J. Sens. Actuator Netw. **2013**, *2*, 717-760; doi:10.3390/jsan2040717

OPEN ACCESS

Journal of Sensor and Actuator Networks

ISSN 2224-2708

www.mdpi.com/journal/jsan/

Review

Software Updating in Wireless Sensor Networks: A Survey and Lacunae

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

[1] Callan Institute/Department of Computer Science, National University of Ireland Maynooth, Maynooth, Co. Kildare, Ireland

[2] MISL, Department of Computer Science, University College Cork, Co. Cork, Ireland; E-Mail: cjs@cs.ucc.ie

***** Author to whom correspondence should be addressed; E-Mail: stephen.brown@nuim.ie; Tel.: +353-1-708-3936; Fax: +353-1-708-3848.

Received: 18 September 2013; in revised form: 1 November 2013 / Accepted: 1 November 2013 / Published: 14 November 2013

Abstract: Wireless Sensor Networks are moving out of the laboratory and into the field. For a number of reasons there is often a need to update sensor node software, or node configuration, after deployment. The need for *over-the-air* updates is driven both by the scale of deployments, and by the remoteness and inaccessibility of sensor nodes. This need has been recognized since the early days of sensor networks, and research results from the related areas of mobile networking and distributed systems have been applied to this area. In order to avoid any manual intervention, the update process needs to be autonomous. This paper presents a comprehensive survey of software updating in Wireless Sensor Networks, and analyses the features required to make these updates autonomous. A new taxonomy of software update features and a new model for fault detection and recovery are presented. The paper concludes by identifying the *lacunae* relating to autonomous software updates, providing direction for future research.

Keywords: WSN; software update; configuration update; fault detection; reliability; recovery; management; autonomous

---

1. Introduction

Wireless Sensor Networks (WSNs) are formed from collections of sensing nodes, placed in a sensor field, that collect environmental data and report it via a gateway or base-station to an application. Typically the nodes are energy constrained, and the major focus of research is how to balance acceptable level of performances against network lifetime. Much research has taken place in the pre-deployment/design space, both in hardware and software, with a focus on the physical layer, datalink, and routing issues. There has been comparatively little research into post-deployment issues, but these are becoming more important as real-world deployments are being realized. These issues include debugging, fault isolation, and software updating. *Over-the-air* software updating, being a general term associated with software updates for mobile wireless devices, has been available for many years. Other terms used include OAP/OTAP (Over the Air Programming), remote updates, remote reprogramming. A lack of published results in the many reported deployments led us to speculate that there is not sufficient support for this feature, and that such updates may be regarded as too risky as, if the updated software fails an expensive network, it may become inoperative.

In this paper we present the first broad-based survey that discusses in detail the need for autonomous update support, significantly extending the results of previous surveys [1–4]. This paper extends an earlier technical report [5], and identifies key future research challenges for autonomic software and configuration updates (SCU)—the *lacunae* in current research. In addition to the survey, the key contributions of this paper are: a model for autonomous software updating, a taxonomy for classifying software update features, a model for fault detection and recovery, and the identification of future research topics in the area of autonomous software and configuration updates.

2. Background

This section presents a discussion of the requirements for over-the-air software updates in a WSN, and then examines how the features of autonomic computing can be applied to these, providing for autonomous software update and recovery.

2.1. Software Update Requirements

Software updating has always been an important topic for wireless sensor network (WSN) research. In the 1978 Distributed Sensor Networks Workshop, arguably the root of published WSN research, three important functional requirements were identified. These are: the need for dynamic modifications, the need for heterogeneous node support, and the need to integrate new software versions into a running system [6].

More recently, other functional requirements have been identified: the importance of the update being reliable, and reaching all the nodes [7]; the need to support fragmentation of the software update over the radio network [7]; the ability to monolithically update all the code on a node [8]; coping with packet loss in the context of disseminating data to an entire network [8]; the need to support version control, so that compatible updates take place when and where needed [9]; the need for special boot-loaders to support the update process [9]; and the need for mechanisms to build a software update and inject it into a WSN [9]. In addition to the overall management of the software update process [1],

other requirements relate to the potential need to update the update mechanism itself [10], to reconfigure various parameters (such as performance or dependability) rather than code [10], and the need for utilities to provide users with standardized ways to manage online changes [10].

A number of performance requirements have been identified as follows: for the updates to be *non-intrusive* (*i.e.*, automatic and *over-the-air*, as opposed to manual over a direct connection); for the update protocols to have a low impact on the performance of the WSN sensing application, and to have a minimal impact on the energy resource of the nodes, and to be resource aware (especially for high energy operations such as flash rewriting [9]); the need to minimize the impact on sensor network lifetime, and to limit the use of memory resources [7]; the need to minimize processing, limit communications in order to save energy, and only interrupt the application for a short period while updating the code [8]; the possible need for the update to meet various real-time constraints [11]; the need to fit within the hardware constraints of different platforms [12]; and the need to cope with asymmetric links and intermittent disconnections, meet the conflicting needs of a low overhead for update metadata overhead while providing rapid update propagation, and to be scalable to very large, high density networks [13].

Security and robustness are also key issues for WSNs in general [14], and for WSN software and configuration updates in particular [15]. Security is critical as an insecure software update mechanism would provide a trapdoor to circumvent any other security measures. Three key issues in the secure update of distributed system software are to provide: integrity, authentication, privacy, and secure delegation [16], and intrusion detection [17]. The ability to download new software into a node introduces a whole new set of failure modes. These include detecting and handling failure (or corruption) of the updated application, other applications, the operating system, and of the network [18], to tolerate hardware and software failures, to monitor system status [10], and to meet other dependability criteria (availability and reliability) [19]. This paper does not include a detailed survey of security-related issues.

Finally, ease of use and maintenance are additional key factors in the future success of wireless sensor networks [1]. Remote software/configuration updating is a core component for both of these factors, improving ease of use by providing flexibility in a deployed network, and directly supporting the maintenance function.

2.2. Autonomous Updates

The key feature of an autonomic system is the ability to self-manage. Other, and not entirely unrelated, features are: interacting with users at the policy level, hardware and software self-protection, and interacting with legacy systems. An autonomic system is aware of its environment, and can anticipate and react to environmental changes [20], and these are all features that are essential for applying over-the-air software and configuration updates in large and widely-distributed wireless sensor networks.

Extending the model presented in [2], we identify the following seven software components for autonomous software/configuration updates, as shown in Figure 1:

- On the management-host:
    1 Self-Management

2 Preparation
3 Insertion
4 Monitoring

- Distributed across the network:

5 Dissemination

- On the nodes:

6 Activation
7 Fault Detection and Recovery

Figure 1. Autonomous Update Components.

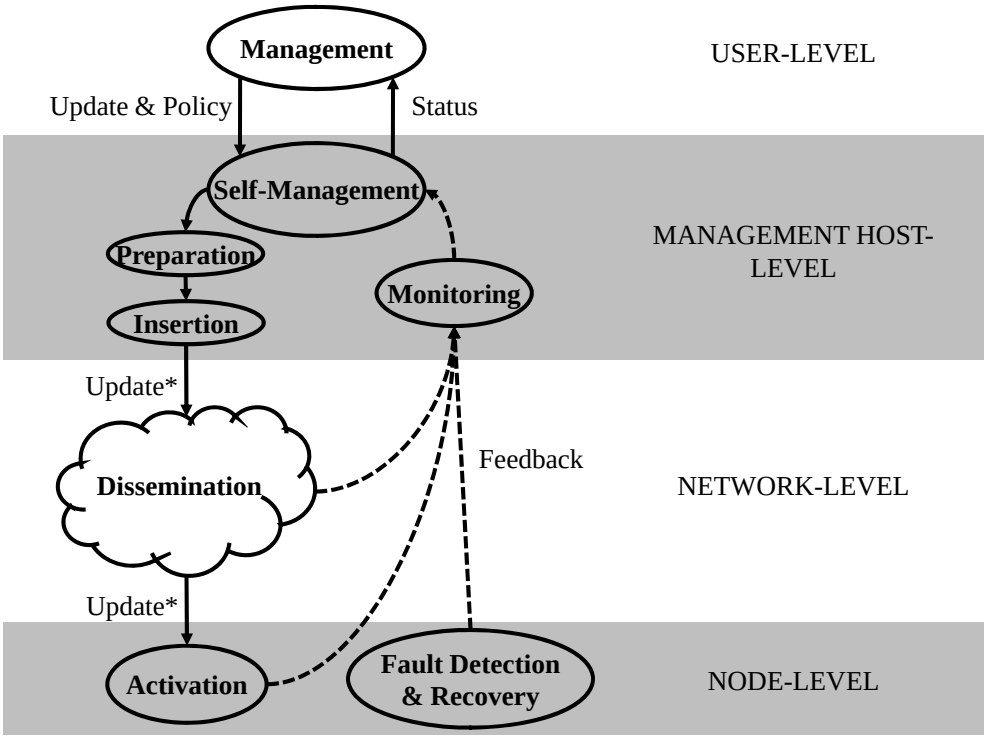


Figure 1 contains the following data components:

- Update—the generated update.
- Policy—specification of the user update policy.
- Update*—the prepared update along the necessary parameters to control dissemination, activation, fault detection, fault recovery, and feedback.
- Feedback—the status of the various activities, returned to the management host.
- Status—the status of the entire update, returned to the user management application.

Some of the key aspects, when compared to a *manual* update, are:

- Support for policy-driven operation: in addition to the data comprising the update, this requires specification of the policies specifying how the update is to be performed, and especially how error conditions are to be handled.

- Self-Management: the management host must incorporate the required policies both in disseminating and activating the update, and also in responding to feedback (again, error handling is the essential component here).
- Planning: the self-management component needs to do planning (for example, using simulated updates) in order to determine whether it is possible to perform an update on the current network according to the specified policies, and also to determine the optimum parameters for doing so (primarily to minimize power consumption).
- Dissemination: the dissemination needs to operate according to the various parameters provided, and to also report feedback.
- Activation: the activation needs to operate according to the various parameters provided, and to also report feedback.
- Autonomous Fault Detection and Recovery: whereas data faults can be detected at the management node or network level, software update faults must be detected at the node level (to enable recovery when communication has been lost). The faults must be automatically detected, and recovery action must be self-initiated (possibly following communication with neighbours).
- Feedback: all the update components need to provide feedback to both allow control of the current update, and to allow planning for future updates.

3. Survey of Techniques for Updating

WSN software update research is described here based on the three categories identified in [1]: the protocols for *disseminating* the update, *reducing the traffic* required for the disseminated updates, and the sensor node *execution environment*, along with an additional category: *fault detection and recovery*.

Common issues addressed by all of these are power efficiency, performance, security, and dependability. In this survey we only consider work directly related to software updating in WSNs—noting, however, that much of this work is built on results from distributed systems, mobile systems, embedded systems, broadcast mechanisms, mobile agents, middleware, routing protocols, data dissemination and collection protocols, security, and software updating for other classes of computer systems. Note that fault detection is only considered when it is integral to the technique being surveyed, as opposed to an underlying, operating system feature.

3.1. Dissemination

The protocols developed for disseminating a software update in a wireless sensor network are based on data dissemination protocols, with key examples being Directed Diffusion [21], RMST, [22], PSFQ [23], Infuse [24], and Sprinkler [25]. An overview of WSN transport protocols is presented in [26]. The key differences between protocols for data collection and for updates are:

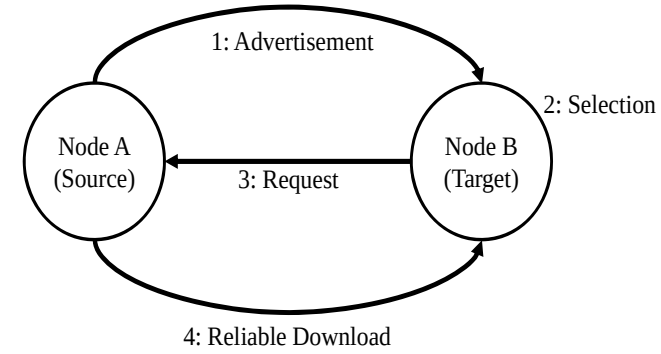
(a) the data flow for software updating is from the gateway to the nodes (see Figure 2),
(b) typically, intermediate nodes both store and forward the data,
(c) the data transfer must be reliable and achieve network-wide coverage.

Figure 2. Update Dissemination.

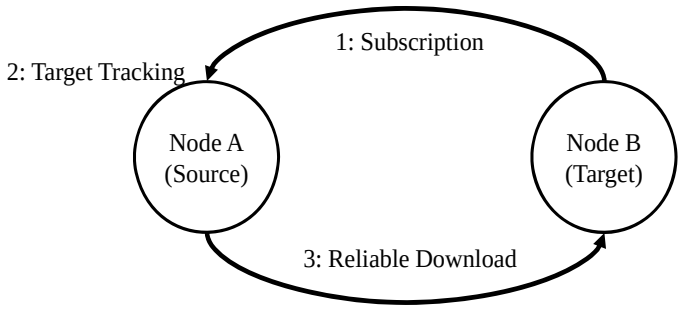
Dissemination may also be performed *out-of-band* (for example, see Deployment Support Network (DSN) [27]), or over an existing network infrastructure—but the vast majority of protocols assume an *ad-hoc* network, and use a form of controlled flooding. Some protocols (see for example, CORD [28]) build a spanning-tree as the first phase of dissemination.

The *advertisement* pattern (Figure 3a) for dissemination protocols typically has four steps: advertisement of available software, selection of a source, requesting the update, and reliable download to the target. The target may then become a source in turn. This may be reversed by use of a *subscription* approach (Figure 3b), where the targets subscribe to sources for new software updates, but for WSN's this results in significantly increased overhead at the source, as every client subscription need to be tracked.

Figure 3. (**a**) Advertisement Pattern for Dissemination Protocols; (**b**) Subscription Pattern for Dissemination Protocols.



(**a**)



(**b**)

The following subsections provide a summary of the operation of the most significant software update dissemination protocols for WSNs (presented in alphabetical order). Each protocol is briefly described, and then autonomic support is reviewed from the following aspects:

- support for pre-dissemination planning, including an energy model and update simulation;
- energy-aware and policy-driven dissemination of the update;
- policy-driven activation of the update;
- feedback.

3.1.1. Trickle

Trickle [13] acts as a service to continuously propagate single-packet code updates throughout the network. The packet size is system dependent, with the original example being 30 bytes under TinyOS, enough to hold a Maté capsule (see Section 3.3.1). Periodically (with a *gossiping interval*) using the maintenance algorithm every node broadcasts a code summary if it has not overheard a certain number of neighbours transmit the same information. If a recipient detects the need for an update (either in the sender or in the receiver) then it brings everyone nearby up to date by broadcasting the needed code. Trickle dynamically regulates the per-node, Trickle-related traffic to a particular rate, thus adjusting automatically to the local network density. This scales well, even with packet loss taken into account. A listen-only period is used to minimise the short-listen problem (where unsynchronised nodes may cause redundant transmissions due to a shift in their timer phases). The CSMA hidden-terminal problem does not lead to excessive misbehaviour by Trickle, as long as the traffic rate is kept low. By dynamically changing the gossip interval, Trickle can propagate changes rapidly, while using less network bandwidth when there are no known changes. Trickle provides code dissemination support for Maté, with code *capsules* fitting into a single packet.

Trickle was one of the original code update protocols, and is still used as the basis for other protocols. Apart from its scalability it contains no special features related to autonomous updates.

3.1.2. Deluge

Deluge [29] is a data dissemination protocol and algorithm for propagating large amounts of data throughout a WSN using incremental upgrades for enhanced performance, building on Trickle. It is particularly aimed at disseminating software image updates, identified by incremental version numbers. The same image is disseminated to all nodes in the network. The program image is split into fixed size pages, and each page is split into fixed size packets so suit the PDU size of the TinyOS network stack. A bit vector of pages received can also fit in a single packet. Nodes broadcast advertisements containing a version number and a bit vector for any new pages received, using a variable period based on updating activity. To upgrade part of its image to match a newer version, a node listens to further advertisements for a time, and then requests the page number/packets required from a selected neighbour. A sender collects several requests before selecting a page and broadcasting the requested packets. When a node receives the last packet required to complete a page, it broadcasts an advertisement before requesting further pages—this enhances pipelining of the update within the network. State data takes a fixed size, independent of the number of neighbours. There are no ACKs or NACKs—the requester pulls the date by requesting either packets for a new page, or missing packets from a previous page. Radio network contention is reduced through heuristics used to select more distant senders. Whereas Trickle provides support for single-packet updates, Deluge supports essentially unlimited sizes.

Deluge has support for reducing energy consumption, but is not an energy-aware protocol. It has no planning features, and provides no model for the anticipated energy use. It supports dissemination and activation, but has no specific support for fault detection and recovery (apart from the watchdog timer under TinyOS); it also has no feedback mechanisms.

3.1.3. Rateless Deluge and ACKLess Deluge

These protocols are based on the Deluge protocol, and use rateless-coding to reduce the need for retransmissions [30]. Rateless coding, with random linear codes, allows a receiving node need only inform the transmitting node of the *number* of packets received, rather than the *specific* packets, in order for the sender to retransmit the required data (any *m* out of *n* packets being sufficient to recover the entire message). In addition, other nodes can overhear both the NACK and the coded result, and use this to recover missing data. Pre-coding of subsequent packets is used to reduce wait times. In ACKLess Deluge, forward error correction (FEC) is used to further reduce the need for retransmissions, by adding redundant packets to reduce the probability that any NACKs will be required. This also improves the efficiency of the pre-coding process. These improvements reduce energy use and latency, and improve scalability through reduced communications.

Rateless and ACKless Deluge provide dissemination and activation of updates, with significant performance optimizations that might be useful for autonomous updates, but no other significant features supporting autonomous updates.

3.1.4. Multicast Deluge (MDeluge)

MDeluge [31] disseminates both binary code and Maté capsules (see 3.3.1) over a spanning tree to a designated subset of a WSN. Unlike Deluge, every node does not have to hold every update. Initially gateways (*micro servers*) act as source nodes for a new version. Source nodes broadcast a code version message for groups of sensor nodes throughout the network, each relay node records the upstream link. Designated sink nodes reply over their upstream link with a code request message, which is sent via upstream links to the source; relay nodes record the route to the sinks. This sets up a bi-directional tree between the source and all the sinks. The source node then propagates the code image in segments to the closest sinks; they in turn become source nodes for further dissemination. Nodes not involved in the update only keep minimal routing information to relay the messages. Sink nodes only relay code request messages if they do not already have the code—otherwise they become a source and return the code data. Segments are sent in multiple packets, and loss detection is done at the sink nodes using NACKs. Nodes maintain two routing tables: one to the root node, and one to the sinks. Image IDs and image version numbers are used to identify each update.

The support for updating groups of nodes and for multiple update gateways are key features of MDeluge, which are likely to be of direct applicability to autonomous updates. It is likely that a more sophisticated ID field would be required.

3.1.5. Multihop Over-the-Air Programming (MOAP)

MOAP [7] is a multi-hop, over-the-air code distribution mechanism specifically targeted at MICA2 motes running TinyOS. It uses store-and-forward, providing a *ripple* pattern of updates; lost segments are identified by the receiver using a sliding window, and are re-requested using a unicast message to prevent duplication. A keep-alive timer is used to recover from unanswered unicast retransmission requests: when it expires a broadcast request is sent. The base-station broadcasts publish messages advertising the version number of the new code. Receiving nodes check this against their own version number, and can request the update with subscribe messages. A link-statistics mechanism is used to try to avoid unreliable links. After waiting a period to receive all subscriptions, the sender then starts the data transfer. Missing segments are requested directly from the sender, which prioritises these over further data transmissions. Once a node has received an entire image, it becomes a sender in turn. If a sender receives no subscribe messages, it transfers the new image to program memory from EEPROM, and reboots with the new code.

MOAP provides dissemination and activation of updates, with some performance optimization, but no additional significant features to support autonomous updates.

3.1.6. Multihop Network Reprogramming (MNP)

MNP [32] is targeted at MICA2 motes running TinyOS and the XNP boot-loader. The protocol operates in four phases. During Advertisement/Request sources advertise the new version of the code, and interested nodes make requests. Sources overhear all other advertisements and requests—a suppression scheme to avoid network overload. During Forward/Download a source broadcasts a *StartDownload* message to prepare the receivers, and then sends the program code a packet at a time: there is no ack. During Query/Update the source broadcasts a Query to all its receivers, which respond by unicast asking for the missing packets. Receivers, having received the full image, now become source nodes and start advertising. During Reboot, entered when a source receives no requests in response to an advertisement, the new program image is transferred to program memory, and the node reboots with the new code. Download requests are sent to all sources to reduce the hidden terminal effect, and select only one active sender in a neighborhood. Flow control is rate based, determined by the EEPROM write speed.

MNP provides dissemination and activation of updates, but no additional significant features supporting autonomous updates.

3.1.7. Crossbow in-Network Programming (XNP)

XNP [33] provides a single-hop, in-network programming facility for TinyOS. A special Boot-loader must be resident in a reserved section of program memory, and the XNP protocol module must be wired into an application. A host PC application loads the image, via a base station mote running TOSBase (this acts as a serial-to-wireless bridge) to one (mote-id specific) or many (group-id specific) nodes within direct radio range of the base. The image is sent in capsules, one per packet; there is a fixed time delay between packet transmissions. In unicast mode, XNP checks delivery for each capsule; in broadcast mode, missing packets are handled, after the full image download has

completed, using a follow-up query request (nodes respond with a list of missing capsules). During download, other applications are halted, and the program is loaded into external memory. When a reboot command is issued (via the host program), the boot-loader copies the program from external to program memory, and finally jumps to its start to activate the update.

XNP is a very limited one-hop update mechanism, essentially replaced by MNP. Except for the explicit differentiation of dissemination and activation, it contains no specific features useful in an autonomous update environment.

3.1.8. Efficient Code Dissemination (ECD)

ECD [34] uses three techniques to improve the efficiency and reduce the latency for disseminating updates: variable packet sizes, selection of the highest impact senders, and impact-weighted backoff timers. By measuring the link quality to its 1-hop neighbours, a node can select the most efficient packet size (taking retransmissions into account). The impact of a sender is determined by the number of neighbouring nodes it can reach—weighted by the link quality, so that the sender with the lowest number of required transmissions is selected. A transmission backoff timer is weighted by the impact factor, so that the node with the highest impact is likely to transmit first, providing quick selection of the highest impact sender. Nodes have 5 associated states, extending the Deluge state model: IDLE (advertise available pages) and listen to received advertisements, 'RX' (having requested newer pages), ESTIMATION (used to estimate the impact from hearing further requests), CONTENTION (backoff state before transmission), TX (send requested packets, and return to IDLE). Nodes that overhear update packets being sent when in the CONTENTION state return to the IDLE state—only the *winner* sends the update packets.

The key focus of ECD is improving efficiency, and it has no specific features supporting autonomic operation.

3.1.9. Freshnet

Freshnet [35] uses a number of optimizations to reduce the energy consumed during an update, when compared to protocols such as MNP, MOPA, and Deluge. It allows pipelining: streaming of pages before the update has been fully received, to minimize latency; it supports out-of-order page reception, to maximize the benefits of receiving from multiple sources; it supports an aggressive sleep mode when the *wave* of a dissemination is not expected; and it reduces the exchange of metadata during *quiescent* phases. Dissemination occurs in two phases: in the initial phase, information about a code update is propagated rapidly throughout the network, along with some topology information (including a hop count). Each node then estimates when the *distribution* phase is likely to reach it, using the hop count, and sleeps until that time. A *warning* message injected a fixed number of times in the initial phase. In the distribution phase, the *Deluge* three-way handshake algorithm is used. Once a node ceases to hear any *requests* from neighbours within range, it enters a *quiescent* phase, where new nodes must *pull* data, rather than every node periodically broadcasting its latest version number, and the update activity is only active on a 50% duty cycle. Freshnet also introduces a *multi-page* distribution mode, which reduces the number of *advertisement* message sent when a node sees that all

its neighbours are running out-of-date code. All pages are sent automatically; neighbours only request retransmissions if all the pages are not received (not on a page-by-page basis).

Freshnet contains no features of direct benefit to autonomous updates, but the aggressive optimizations to reduce energy use and latency provide useful flexibility.

3.1.10. Gossip Control Protocol (GCP)

GCP [36] is a controlled flooding algorithm, using Piggy-Backing and Forwarding Control to reduce the energy consumption in mobile WSNs. Periodic hello *beacons* are transmitted so that nodes are aware of their 1-hop neighbours. Available software version numbers are piggy-backed onto these beacons. To balance the load, and reduce energy use, each node transmits beacons a limited number of times, based on available *tokens* (a system constant), renewed when a new software version is received. When a newer beacon is received, a node responds with its own beacon to request the new version. When an older beacon is received, the node responds with the updated software. Neighbours overhear the software update packets to benefit from *free updates*. MD5 signatures are used to verify the updates. The protocol contains no support for multiple packets in a software update.

The key focus of GCD is improving efficiency, but the code update verification through signatures could be a useful feature for autonomic updates in vulnerable situations.

3.1.11. Impala/ZebraNet

Impala [37] is the event-based middleware layer of the ZebraNet [37] wireless sensor network. It is designed to allow applications to be updated and adapted dynamically by dispatching events through application adapters. ZebraNet nodes are expected to be inaccessible, and deployed in large numbers, so ZebraNet supports high node mobility, constrained network bandwidth, and a wide range of updates (from bug fixes, through updates, to adding and deleting entire applications). Applications consist of multiple, shareable modules, organized in 2 kB blocks. The Application Updater allows applications to continue running during updates, and can process multiple contemporaneous updates; version numbering is used to ensure compatibility of updates with existing modules. It also handles incomplete updates, and provides a set of simple sanity checks before linking in a new module. Software updates are performed in a three-step process: firstly the nodes exchange an index of modules, then they make unicast requests for updated modules, and finally they respond to requests from other nodes. An exponentially increasing backoff timer reduces management traffic, but can delay updates when separated groups of nodes reconnect. When software reception is complete, then after performing simple sanity checks, the old version application is terminated, the modules in the new version are linked in, and the new application is initialised prior to use.

Impala provides a reliable update service for ZebraNet. Some of its features, such as compatibility and sanity checks, could be a useful feature for autonomic updates however.

3.1.12. Multicast Code-redistribution Protocol (MCP)

MCP [38] is envisaged to provide support for a Multi-Application WSN (MA-WSN) where nodes switch between different application images as required. Nodes periodically broadcast advertisement

messages for known applications, using an *application information table* built from incoming advertisements. To perform an application switch, a dissemination command is flooded from the sink, containing the information of which active application is to be switched to which new one (probabilities can be used to switch a random subset). If a receiving node has the new application it becomes a *source*; if it needs this new application, it becomes a *requester*; otherwise it becomes a *forwarder*. A requester periodically multicasts request messages to its nearest source, until it has acquired all of the new application. Non-responsive sources are timed out. Sources respond with data, and relays forward both request and data messages. Sensors advertise different applications in a round-robin manner. Requesters and forwarders cache a small number of packets, and respond immediately for any requests to cached content. An associated size reduction scheme, using an *update conscious compiler* is described in [39].

The reduction in traffic and latency reported for MCP are valuable features, and the support for switching selected subsets of nodes with a random component, is a potentially valuable feature for autonomous operation.

3.1.13. Parallel Diffusion Mechanism (PDM)

PDM [40] uses dynamic transmit power control to improve dissemination performance through interference-free simultaneous transmissions. The inverse-squared relationship between distance and power is used to estimate the transmission region of each node. Code is propagated by forwarding nodes (F-nodes), which advertise a *ProgramID* and *SectionID*. Nodes which have received code segments successfully become candidate forwarding nodes (CF-nodes) and may be selected as future F-nodes. Neighbour candidate forwarding nodes (NCF-nodes) are CF nodes with overlapping transmission regions. Unknown nodes (U-nodes) have not yet received the code segments, and public nodes (P-nodes) are U-nodes located in the overlapping regions of CF-nodes. U-nodes and P-nodes respond to advertisements, and CF-Nodes use these responses to estimate the minimum power level required for overlap-free transmissions. When NCF nodes overlap, one goes to sleep to save energy, and reduce overlap.

The reduction in both latency and energy use, are valuable optimisations for software updating. The increased parallelism may provide a useful feature for autonomic behaviour.

3.1.14. CORD

CORD [28] is a reliable, bulk-data dissemination protocol, with a design focus on minimizing energy consumption. A two-phase approach is used: in the first phase, a connected dominating set (CDS) of *core* nodes is selected and the update reliably propagated to them; in the second phase, the core nodes disseminate the update to their non-core neighbours. This significantly reduces the number of control message that need to be sent, which reduces both energy use and collisions. An aggressive, coordinated sleep schedule, enabled by the use of a CDS, is also used to further reduce energy consumption. Link quality and the sleep schedule are built into the CDS generation. One additional feature for CORD is its suitability for heterogeneous networks, where a subset of more highly powered nodes can be selected for the CDS, extending the lifetime of the entire network.

CORD has support for reducing energy consumption, but is not an energy-aware protocol. It has no planning features, and provides no model for the anticipated energy use. CORD is a data dissemination protocol, so has no built-in support for activation; it also has no feedback mechanisms.

3.1.15. Deployment Support Network (DSN)

An alternative to either accessing the nodes individually, or accessing them over the sensor network, is to provide a parallel network specifically for maintenance [27]. Accessing the nodes individually for a software update is normally impractical for two reasons: the large number of nodes, and the inaccessibility of the nodes. Accessing them over the wireless sensor network itself has several disadvantages: it relies on the network being operational, it has an impact on the performance of the sensor network, and it depletes node energy supplies. A DSN of small, mobile, temporarily attachable nodes can provide a solution to some of these problems. Providing virtual connections from a host PC to the individual nodes allows normal host tools to be used in updating the software.

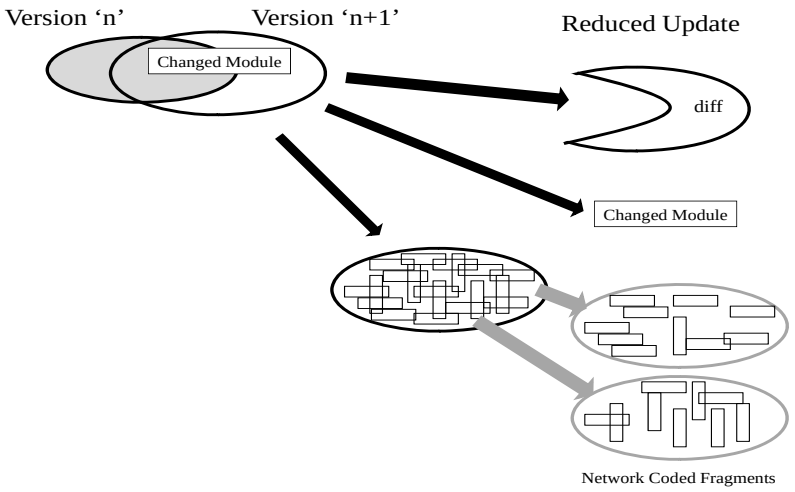
DSN is essentially an architecture rather than a specific implementation for software updates. The concept could provide high reliability, out-of-band support for autonomic operation in networks where the cost was justified. An open question is whether DSN could be provided over-the-air using an independent network overlay.

3.2. Traffic Reduction

Traffic reduction has a direct impact on the wireless energy used, and therefore it both extends sensor network lifetime, and enables updates on networks which are near the end of their life (from an energy viewpoint).

The three main approaches (see Figure 4) for reducing traffic are: (a) Modularity: by using modular software, so that just the necessary changed or new modules need to be transmitted; (b) Size Reduction: by using differential updates, so that just the changes from the previous version need to be transmitted; and (c) Network Coding: using network coding to maximize the benefits of overhearing in a broadcast environment with loss, and thus reduce the traffic. These approaches have been addressed in other domains, for example the rsync algorithm [41], mobile agents [42], and network coding [43].

Figure 4. Update Traffic Reduction Approaches.



In the following subsections, the two direct size-reduction techniques are reviewed (differential updates and network coding). Modular updates which require a supporting execution environment are discussed in Section 3.3.

3.2.1. The Reijers code distribution scheme

This is principally concerned with reducing the image size [8]. It does this by generating and propagating a diff script, rather than the entire image. The script is downloaded as a series of packets, each of which is self contained and states the address range which it builds (allowing for immediate processing of each packet, and easy identification of missing packets). The script is applied against the currently running program, progressively building a new image in external memory (EEPROM). A boot-loader copies this into code memory (Flash), and then control is transferred to the new image. Packets can be processed out-of-order; missing packets can be identified afterwards, during a verification phase, and the binary data requested from its neighbours. As of 2003, network flooding was not supported, and the script is loaded into EEPROM in its entirety before processing. The major limitation of this algorithm is that the scripting language developed is CPU-specific.

Whereas size reduction may inherently increase the flexibility of response, there are no specific features in the Reijers code distribution scheme relevant to autonomous updates.

3.2.2. Incremental Network Programming (INP)

This uses the *rsync* algorithm [41] to provide incremental software updates [44]. It differs from Reijers' work in that the algorithm is processor independent; in fact it does not require any knowledge of the program code structure. Energy is saved by sending software updates as a *differences script*. The script can either specify an unmodified block to be copied (possibly to a new address), or can contain a modified form of S-Records for the new program image. The sender asks the receiver for the checksums of its current blocks, and after the receiver responds, it sends only the changed blocks. The script commands are disseminated using the XNP protocol, and are stored by a node in external memory during the Download operation; the host then queries the node for missing commands; finally a Decode command is sent. The program images are stored in two sections of external memory (EEPROM), one for the old image, and one for the new image (a third is used for the script). The network programming module passes a parameter to the boot-loader (modified from XNP) to select the image to read into program memory at boot time.

As for the Reijers scheme, the only features relevant to autonomous updates are the inherent advantages introduced by the size reduction with reference to autonomous updates.

3.2.3. Deluge with Delta Coding

A greedy delta algorithm is used to produce minimal delta file sizes based on the differences between two software versions (the original version and the update) [45]. This delta file is then disseminated throughout the WSN using Deluge. The delta file is based on four *VCDIFF* instructions: *shift*, *run*, *copy* and *add*, but uses a simpler file format designed to optimize sequential EEPROM accesses. These are then applied by each node to its original software version in order to produce the

update. Energy use is reduced by the effectiveness of the compression algorithm—the authors report figures between 30% and 99% for representative programs and updates under TinyOS.

As for the Reijers' scheme, the only features relevant to autonomous updates are the inherent advantages in terms of extra flexibility introduced by the size reduction with reference to autonomous updates.

3.2.4. AdapCode

AdapCode [46] is a dissemination protocol, using broadcasts to benefit from redundant links, and adaptive network coding to reduce network traffic. Linearly independent packets are coded on each node for propagation, and decoded using Gaussian elimination. When a node receives a packet, it uses Gaussian elimination to see if it can now decode the entire *page*. If so it determines its coding scheme, based on the number of neighbours, and following a random backoff, starts broadcasting the coded packets. Periodically, with progressively doubling timer value, receiving nodes transmit a NACK giving the number of additional packets required. Nodes that can respond now delay for a random time, and only respond if no other node has during this delay. The timer is reset once a response is received. Nodes wait for an inter-page time T before moving to the next page—selection of this value has a significant impact on efficiency and latency. The Gaussian elimination takes 1433 of RAM space with their selected coding parameters (number of messages M and maximum coefficient p). Their results show a traffic reduction of 30%–40%, a reduction in latency of up to 15%, and a better balancing of the load between the nodes, compared to Deluge.

AdapCode has no specific features relevant to autonomous updating, apart from the advantages of traffic reduction through network coding, but it is an open question whether the use of network coding might provide some additional flexibility for autonomous updates, perhaps in environments with limited connectivity or high noise.

3.2.5. Power Efficient Adaptive Network Coding (PEANC)

By adding neighbour discovery to AdapCode [46], higher efficiencies are obtained, as the nodes have a more accurate estimate of the number of neighbours [47]. In the initial deployment, nodes exchange a selected number of beacons in order to evaluate the link quality to each neighbour. Prior to code dissemination, nodes broadcast a beacon so nodes may establish their neighbours. The dissemination is then performed in a manner similar to AdapCode, where, once a node has received enough packets to decode a page using Gaussian elimination, it then uses the current neighbour count to recode the page prior to broadcasting it. NACKs are used to ensure 100% reliability. The reported results indicate an approximate halving of energy used when compared to AdapCode.

As for AdapCode, apart from the improved traffic reduction, it is an open question whether autonomous updates can benefit from the use of network coding to disseminate the update.

3.2.6. R-Code

A Minimum Spanning Tree is used to disseminate the update across the network, using network coding and overhearing to reduce network traffic. The link weights are based on the number of

expected packets. The key idea in R-Code [48] is that each node selects a *guardian* parent node in the MST, which is responsible for ensuring its children receive the full update. Building the MST is not part of the protocol, but is done during periodic *initialization* stages. The data to be disseminated is broken into generations, each of which contains multiple pages. In the *broadcast* stage, nodes broadcast the earliest unfinished generation to their children. After a short, random delay, by the children ACKs (at the message, rather than the packet level) are used to ensure reliability. Guardians move to the next generation once all children have ACKed the current one. Periodic *beacons* are used to determine link quality, and maintain connectivity. Guardians can be switched dynamically, depending on which potential parent receives the full update from overhearing first. Random linear coding and Gaussian elimination are used, as in AdapCode. Their results show a reduction in traffic of 15% and in latency of 50% when compared to AdapCode.

As for AdapCode, apart from the general advantages of traffic reduction, it is an open question whether autonomous updates can benefit from the use of network coding to disseminate the update.

3.2.7. Remote Incremental Linking (RIL)

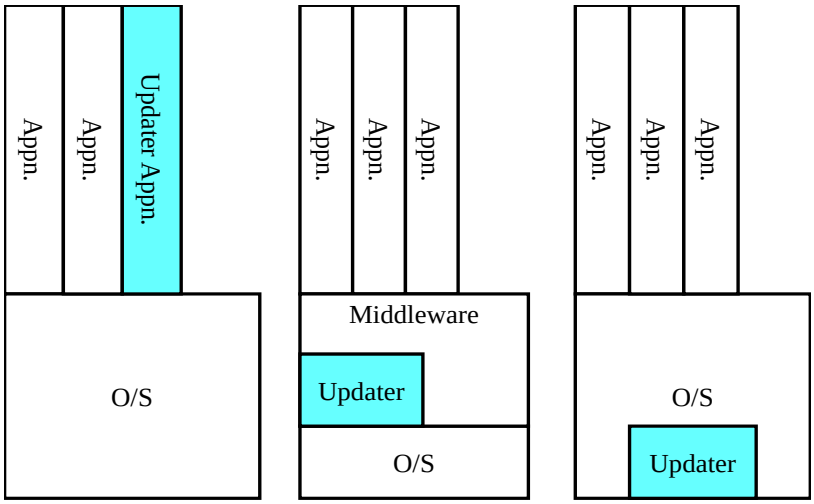
This modular updating system, designed for the mica2 mote, support static and dynamic updates [9]. A program is built from a number of functions, reserving space at the end of each function to allow for some future expansion (slop space). These functions are linked remotely from the target nodes, at a more capable node such as the base station. The new version of a function can be written in program memory at the same start address, as long as it fits, without the need to update references to this function elsewhere in the program. Otherwise, either adjacent functions or the new function are moved: whichever minimizes page writes. The new version (code and/or data) is delivered in an incremental manner, using diff-scripts generated by the Xdelta algorithm. Single pages can be updated, reducing the need for RAM use, and allowing the script for the next page to be downloaded while the previous page is being written (using ATMega128's read-while-write). The boot-loader interprets the scripts and updates the program memory pages. The system is probably more suitable for minor, incremental upgrades, and the need to retain the capability for full system upgrades is identified by the authors.

The system has no specific features relevant to autonomous updates, but the additional granularity associated with the inherent advantages of size reduction might be of value.

3.3. The Execution Environment

The execution environment has a significant impact on how the software update may be disseminated and executed. Software update functionality will always need some low-level support (e.g., overwriting program memory) but the rest of the functionality may be implemented in device-specific firmware (e.g., Scatterweb [49]), in the operating system (e.g., SOS [50]), in a middleware layer (e.g., Agilla [51]), or as an application (e.g., MOS [52]) as shown in Figure 5. Mobile agents also provide potential software upgrade mechanism, though probably more suited to the insertion of new code than the upgrading of existing code. These agents are usually supported in a middleware layer to allow access to global data. Network-query approaches (e.g., TinyDB/TAG [53]) are not included unless there is a direct reprogramming aspect.

Figure 5. Execution Environments.



(a) Application (b) Middleware (c) O/S

We do not directly consider hardware-specific aspects, but there are three key hardware features: re-programmable program memory (e.g., FLASH), a temporary store (e.g., EEPROM), and a secure location for a fallback image (e.g., Boot-ROM). An overview of WSN platforms is given in [54–57].

3.3.1. Maté

Maté [58] is a communication-centric middleware layer for WSNs, based on a virtual machine architecture, which runs on TinyOS (Bombilla [59] is an instance of the Maté virtual machine that runs under TinyOS). Code is divided into 24-instruction capsules; larger programs can be supported using subroutine capsules (currently limited to 4). Programs communicate via message passing, in a similar way to TinyOS, using the Trickle protocol [13]. Code capsules are flooded through the network via viral programming-code capsules can be marked as self-forwarding, and such capsules are flooded throughout the network based on 32-bit version numbers. Every node broadcasts a summary of capsule versions to its neighbours based on a random timer: if a node hears an identical summary, it suppresses its own next transmission; if it hears a version summary with older capsules than it has itself, it broadcasts the newer modules. This flooding continues even after the entire network has been updated. (An earlier version of Maté supported an instruction to initiate code propagation, but experiments showed that this could easily lead to network saturation.) There is support for injecting new modules (or new module versions) into the network via a capsule injector.

The flexibility provided by the support for mobile agents in Maté might prove to be a useful enabling technology for autonomous updates, allowing a self-managing system to inject different behaviours in order to respond to local conditions.

3.3.2. Agilla

Agilla [51] is a middleware layer that runs on TinyOS and supports mobile agents for WSNs. It is based on Maté and provides a virtual machine for agent code, but a more controlled migration of agents within the network (as opposed to flooding in Maté), and with support for larger agent code

sizes. The agents communicate via remote access to local tuple space on each node, and can migrate via move and clone instructions. Migration uses small packet sizes (less than 41 bytes) and a store-and-forward mechanism to reduce the impact of packet loss. Currently only assembly language programming is supported, with the obvious disadvantages. Applications are deployed by injecting mobile agents into the WSN, but there is no explicit support for code versioning or updating of agent code. Agents can be stopped, so software updates could probably be achieved on top of Agilla, but this would require explicit application support (in the mobile agent code).

The flexibility provided by the support for larger mobile agents in Agilla might prove to be a useful enabling technology for autonomous updates, allowing a self-managing system to inject different behaviours in order to respond to local conditions.

3.3.3. Melete

Melete [60] is based on the Maté virtual machine, and allows for the dynamic updating of TinyScript-based functionality based on dynamic grouping. By joining and leaving groups, and thus executing the relevant code, this allows a sensor network to respond to the current conditions. Multiple applications are executed concurrently, so a node may perform multiple functions as required. Dissemination is based on modified Trickle functionality: code is selectively and reactively disseminated only to nodes in the group. And additional FORWARD state is added to Trickle, creating a multi-hop *forwarding region*. Only code for its associated groups is stored on a node. To minimize network overhead, while minimizing delay, version information for all groups is propagated throughout the network.

The provision of concurrent applications, and support for dynamic responses to network state, are two features which are of direct interest to an autonomic software update.

3.3.4. Mobile Agent Framework (MAF)

This framework [59] is built on top of Maté which runs under TinyOS. It provides a *breadcrumb* model for communication, where agents may leave per-node data behind them for future agents to access (though an agreed index). An agent's per-agent state is propagated along with the agent code. The agent propagation is under the control of the agents, and supported by a number of *FwdAgent* functions in the framework. These support reliable, unicast, and broadcast forwarding requests. Agents may terminate their own execution on a node by exiting, or may continue indefinite execution in a loop structure with sleep statements.

As for Maté, the flexibility provided by the support for mobile agents might prove to be useful enabling technology for autonomous updates, in addition to the ability to propagate state as well.

3.3.5. Component Oriented Middleware for Sensor Networks (COMiS)

COMiS [61] is a component-based, generic middleware layer developed as part of the TinyMaCLaS project. It allows components to be loaded onto particular nodes, and supports network-wide software updating. Applications are written in the DCL (Distributed Compositional Language) scripting language [61]. Software components are registered locally, and then available for

discovery by other components, using a broadcast algorithm, and subsequent connections. Both middle-ware and application components can be updated. Compiled component binaries are deployed into the network and installed (registered) at deployment nodes. Connection is established with (a subset of) nodes in the network using the discovery component. Then the update method is used to deploy the code to these nodes. On receiving an update, the COMiS listener component checks that the version number is greater before installing and relinking the new component, which is then restarted. A TTL field provides limited flooding control.

COMiS contains no specific features relevant to autonomous updates, but has flexibility in terms of what can be updated and might prove a useful enabler.

3.3.6. Contiki

Contiki [62] provides a core kernel, and supports loadable applications and services that can be dynamically replaced at runtime. These execute as processes, and share a single address space. The core is a single binary image, and typically consists of the kernel, the program loader, commonly used libraries, and a communication stack. It cannot generally be modified after deployment. Typically programs are downloaded into the system over the communication stack, and stored in EEPROM before being loaded/linked into the system using a relocatable binary format. When a service process is replaced, its process ID can be retained to support continuity of service; service state can also be transferred. Over-the-air programming is supported by a simple protocol that allows new binaries to be downloaded to selected concentrator nodes using point-to-point communication (over the communication stack). When the entire binary has been received, it is broadcast to neighboring nodes: these use negative acknowledgements to request retransmissions. Version numbers are used to match service interfaces calls with library functions. Version numbers are also used to support replaceable service processes, so that incompatible versions of a service will not try to load the stored service state.

Contiki contains a number of features that might be of assistance in autonomous updates. The use of concentrator nodes might provide a useful mechanism for controlling updates, and getting feedback. Version number support is a key feature for autonomous updates, to avoid compatibility problems. As for other modular systems, the ability to reload modules might provide useful flexibility for autonomous updates.

3.3.7. Fine-Grained Software Reconfiguration (FiGaRo)

FiGaRo [63] provides fine grained control of software components, providing support for heterogeneous networks, component dependencies, and version constraints. It is based on Contiki, and has two core constituents: a *component model* providing support for reconfigurable components with dependencies and versions, and a *distribution model* that restricts dissemination to a subset of nodes. FiGaRo components are mapped to Contiki *services*, and the Contiki dynamic linking facility is used. Subsets of nodes are selected using attribute/value pairs, and a mesh is created connecting matching nodes. A spanning-tree is selected from the mesh to support software reconfiguration, rooted at the target node closest to the injection point. The current values of a nodes attributes are piggybacked on all transmitted traffic, and is used to populate a simple routing table with hop-based costs. The first stage of reconfiguration is a *marker* message to identify redundant paths, parents being selected based

on minimum hops. Code is distributed in *chunks* on a hop-by-hop basis using explicit ACKs. Nodes buffer every message, and use overhearing to confirm that their children have in turn transmitted it.

FiGaRo provides no specific features related to autonomic updates, but the granularity, version and dependency support, and support for groups of nodes (allowing for heterogeneous updates), along with performance improvements, may be useful features in an autonomic environment.

3.3.8. Mantis Operating System (MOS)

MOS [52] is a lightweight, UNIX-style O/S with support for threads. These threads can be user application threads; the network stack is also implemented as a set of user threads. The MOS operating system library provides support to write a new code image to EEPROM. A commit function may then be called to write a control block for the boot-loader, which then installs the new code on reset. Any part of the code image (except for the boot-loader) may be updated, from a simple patch, to one or more thread, to the entire image. No specific tools are provided to build an update (except for an entire image), and an application-specific protocol would be needed to transfer the update over the network, apply the update, and reboot the system. MOS thus provides a framework to enable remote software updating, rather than the actual implementation.

MOS contains no specific features support autonomous updates, but the UNIX-style environment might provide for more flexibility than the single-threaded environment provided by many WSN operating systems.

3.3.9. Remote Execution and Action Protocol (REAP) [64]

REAP supports a Reactive Sensor Network (RSN) where the network is reprogrammed by packets passing through the network. REAP, originally implemented to support WSNs, is supported by a lightweight mobile code daemon. Packets contain code and data, and can be sent by TCP/IP or Directed Diffusion to other nodes. An efficient object serialization class provides high performance transfer of objects. An index system provides a distributed database of resources. Nodes can search for algorithms that match their architecture and operating system, thus providing heterogeneity support. The packet router determines multihop paths through broadcast requests, gradually increasing the TLL until a limit is reached or the destination is found.

Heterogeneity support is a key feature in enabling autonomous updates.

3.3.10. Sensor Information Networking Architecture (SINA)

This is a database style approach, supporting reprogramming through downloaded scripts [65]. The SEE (Sensor Execution Environment) dispatches incoming messages and executes contained scripts. These are in an extended form of SQL called Sensor Query and Tasking Language (SQTL)—this adds event-based procedural programs, that can be explicitly propagated around the network using the execute primitive, which can select a subset of nodes to propagate the script to. There is no built-in version control, or limits to traffic.

The database-style model of SINA might provide an alternative framework for highly flexible behavior for autonomous software updates.

3.3.11. Spatial Programming (SP)

SP [66] is a network-wide programming model. Script based programs propagate around the network in Active Messages [67]; so in principle a new version of a program can be injected using an Active Message. Programs reference global data through a space/tag tuple defining the geographic scope and a content-based name. Spatial reference bindings are maintained per-application, and there is no built-in mechanism to terminate old scripts, but they can be terminated at the application level (by exiting); so an application could support self-upgrading using the SP mechanisms.

There are no specific features relating to autonomous updating, apart from the flexibility that scripting introduces.

3.3.12. Pushpin

Pushpin [68] supports a dynamic process fragment model, similar to mobile agents: the fragments each support an update function which is called repeatedly. The Bertha operating system can load and unload these 2 K size fragments dynamically. A process fragment contains state and code, and may transfer or copy itself to neighbouring Pushpin systems: each fragment supports install and uninstall functions to support this. The fragments communicate through a local bulletin board system; nodes also maintain a synopsis of neighbouring bulletin boards, which can be used by a fragment to decide to request a move. The Pushpin IDE supports the development and downloading of process fragments to an attached Pushpin node (serial connection).

As for other modular systems with dynamic linking, flexibility provided by the support for dynamic updates agents might prove to be a useful enabling technology for autonomous updates; in addition the bulletin board might also provide features that help to support this.

3.3.13. ScatterWeb

This is a distributed, heterogeneous platform for the *ad-hoc* deployment of sensor networks [49]. Software is divided into a firmware core and modifiable tasks to provide a secure update environment. The software is fragmented into small packets to minimize packet loss. Lost packets are recovered in a two-step process: first from local nodes, and if this fails from the gateway. The update is first copied into EEPROM, and then written into flash. This allows the integrity of the received code to be checked, and allows synchronization of updates across the network (using a flash command, or a timestamp). A host tool supports over-the-air reprogramming via a USB/radio stick (ScatterFlasher) or via a www gateway (Embedded Web Server). Supported protocols, such as Directed Diffusion, can be used to distribute an update.

As for other modular systems with dynamic linking, the updatable tasks provided by ScatterWeb provide flexibility which might prove to be a useful enabling technology for autonomous updates.

3.3.14. SensorWare

SensorWare [69] provides a heavyweight active sensor framework (*i.e.*, supporting code mobility) designed for relatively large memory nodes (e.g., 1 MB Program Memory/128 KB RAM). It supports programming through a high-level scripting language based on tcl, and provides a runtime environment

for dynamic control scripts. Scripts can be injected by *external users*; scripts can contain a replicate command that causes the script to be replicated on another node. Scripts can be replicated to a specific list of nodes, or broadcast to all neighbours. The script replication does not support versioning.

As for other script-based systems, the flexibility provided by SensorWare might prove to be a useful enabling technology for autonomous updates, allowing a self-managing system to inject different behaviours in order to respond to local conditions.

3.3.15. Sensor Operating System (SOS)

SOS [50] is a microkernel-style operating system, supporting 'C' programs, with support for dynamic module loading and unloading. Services (such as routing, applications *etc*.) are provided as dynamic modules. Synchronous communication between modules is provided through function pointers. A function control block (FCB) stores the function prototype and version number. New module advertisements contain the version number, and if this is an updated version, and there is space, the module is downloaded. For an update, a *final* message is sent to the old module, which is then unlinked from the FCBs, before the new version is loaded and init called to repopulate the FCB. Other modules keep running—if an unavailable function is called, then kernel provides a stub to return failure. SOS does not support update of the core kernel image.

The dynamic modules, and versioning support, provided by SOS provide flexibility which might prove to be a useful enabling technology for autonomous updates.

3.3.16. HERMES

HERMES [70] is a lightweight framework for monitoring post-deployment code behavior on WSNs. It has been implemented on the SOS platform. Rather than working at the line-of-code level for developer debugging, such as remote-gdb, HERMES provides a higher level of monitoring, at the level of data-flow interactions using *interposition* to enable this in a non-intrusive manner. Generated stubs are dynamically linked (or interposed) between *target* modules and the O/S runtime. This requires no modifications to the original source code. Features supported are: observing in-field execution, and controlling in-field execution. Observation is supported by monitoring the function calls to selected modules, allowing for conditional watchpoints, and synthetic event generation for in-field testing. In-field execution allows dynamic access control (for example, inserting firewall rules dynamically), traffic shaping (for example, by introducing resource allocation policies dynamically), and fixing isolated failures (for example by invoking fail-safe operation when an illegal interaction is observed, or recovery by invoking stateful rollback using interaction traces). The SOS dynamic linker tracks down all provide-subscribe function pairs at load time of a module, a feature which HERMES leverages. HERMES also provides a transparent update feature, to allow a new version of a module to run in parallel with the old one until it has collected enough state to transparently take over operation.

The key feature for autonomous update support is transparent update, which provides a smooth transition to an updated module. The monitoring feature may also be relevant, though it is possibly too granular for the requirements of update-related software faults. Also it relies extensively on the O/S and network operating correctly, and so is not likely to be useful for autonomous software recovery.

3.3.17. MinTax

MinTax [71] uses *in-situ* compilation of a high-level language, and dynamic linking, to reduce update sizes. The high-level language allows functionality to be coded very efficiently, and in a heterogeneous network the same update can be sent to all nodes, and then compiled on each node for its own architecture (rather than disseminating multi-architecture updates). Lookup tables are used to provide for direct access to hardware I/O ports on different platforms. Code updates are stored in EEPROM, and after compilation is written to Flash memory. Significant energy reductions are shown for a simple *Blink* application when compared to Deluge under TinyOS (from 560 mJ to 1 mJ).

MinTax itself provides no specific features for autonomous updates, but the energy reductions reported are significant.

3.3.18. uFlow

This is an event based framework, where all actions are decomposed into tasks (simple tasks and composite tasks), designed to support software updates without losing runtime status—that is, no reboot is required [72]. Tasks are invoked by an event handler, through a task table, with parameters passed in a manner similar to a function call. All tasks are run to completion, and no task ever interrupts another. Control flow is provided through task hierarchies, Sequence Tasks, Conditional Branch Tasks, and While Loops. The uFlow framework enables dynamic code updates, without losing context, by changing the task event handling structures. New tasks can be added, unwanted tasks removed, and tasks modified by replacing them with a new task.

uFlow provides no direct support for autonomous updates, but the ability to perform on-the-fly software updates without reboots could be a valuable feature.

3.3.19. Dynamic TOS

Dynamic TinyOS [73] is an extension to TinyOS that allows TinyOS components (grouped into ELF objects) to be downloaded individually, and dynamically linked into a TinyOS application. Modifications to the *nesc* compiler maintain the component boundaries, and at run-time a new Tiny Manager components dynamically links in replacement objects—and the components they contain. A file system is provided for downloaded ELF objects, and a global symbol table is maintained to resolve dependencies. An *Interrupt Router* allows for interrupt service routines (ISRs) to be replaced. The results show significant energy benefits over a monolithic TinyOS update using Deluge, with a minimal run-time performance impact.

Dynamic TOS provides no additional features related to autonomic support when compared with TOS, but the increased granularity of updates, and performance improvement, may be a useful feature in this regards.

3.3.20. Dynamic Virtual Machine (DVM)

The DVM [74] architecture provides a dynamically modifiable virtual machine on top of SOS. Event handlers are written in a portable bytecode scripting language, and Operation Libraries implement its instruction set. A Scheduler is used to control script execution. An event manager maps

events to handlers, allowing new events to be defined dynamically. Both scripts and binary modules implementing an Extension Library can be updated (using for example Deluge to perform the dissemination) allowing new bytecodes to be introduced. A Resource Manager implements Admission Control is used to ensure that only compatible scripts that can fit in memory are admitted into the system. Scripts are installed (after a wait if required) only when there are no scripts executing.

DVM provides no specific features related au autonomic updates, but as for Dynamic TOS, it provides additional granularity and performance improvements that may be useful features in an autonomic environment.

3.3.21. ELUS

The ELUS [75] framework provides a level of indirection between function calls and function invocation. This allows components marked as *reconfigurable*, or functions within those components, to be dynamically updated at runtime. Data transfer between the old and new component's attributes is achieved using *get* and *set* methods. A *RECONFIGURATOR* receives new code, and informs an *AGENT* to load the component code and update related lookup tables. Updates are disseminated using a publish/subscribe protocol, with unicast requests and broadcast retransmissions. Nodes publish all their code versions periodically, and interested nodes request them, and the publishing nodes become *senders* and transmit the data. Unicast NACKs are used to request missing packets. The ELUS TRANSPORT PROTOCOL is used to send reprogramming requests for a component to nodes.

ELUS provides no specific features related to autonomic updates, but it provides additional granularity and performance improvements that may be useful features in an autonomic environment.

3.3.22. FlexCup

FlexCup [76] enables on-the-fly reinstallation of software components in TinyOS. In a manner similar to Dynamic TinyOS, components can be linked into separate *binary components*, which are subsequently linked at runtime into a TinyOS system. FlexCup holds the following metadata: generic program information, a program-wide symbol table, and a relocation table for the binary components. A 5-step algorithm is used for updates: storage of new metadata and code, symbol table merge, relocation table placement, reference patching, and installation of the new code followed by a reboot. FlexCup Basic transmits entire updates; FlexCup Diff only transmits incremental changes—this requires the base-station to know the exact configuration of the sensor node to prepare a diff script.

FlexCup provides no specific features related to autonomic updates, but as for Dynamic TinyOS, it provides additional granularity and performance improvements that may be useful features in an autonomic environment.

3.3.23. Stream

Stream [77] reduces the downloaded program image size, compared to *Deluge*, by separating the image into two parts: the reprogramming protocol and the application. The sensor node must support booting from multiple images. The reprogramming image is preloaded as a separate image from the application image—the application image is therefore significantly smaller. For an update to occur, the

application detects that an update is requested, and reboot into the reprogramming image. This only requires a small amount of extra code in the application image, rather than the full reprogramming protocol. In addition, image metadata is only broadcast by the reprogramming image during the reprogramming phase. The dissemination is based on the *Deluge* protocol, with changes to ensure a node keeps running the reprogramming image until all its neighbours have received the update. This is done by broadcasting an ACK once the new image has been received—the neighbours can now remove this node from their *requesting set*. Once a node has downloaded all the update pages, and its requesting set is empty, it reboots into the updated application image.

As well as the flexibility benefits introduced by energy reduction, the idea of using a separate image to support the update operation introduces significant benefits into the activity from the viewpoint of autonomous behaviour. It would allow an autonomous update to have significant functionality, without impacting the size of the application image. Also, the time period during which an autonomous update was active, could allow for much more efficient energy use (not subject, for example, to any power-cycling schemes determined by the application).

3.3.24. Zephyr

Zephyr [78] improves code-size reduction by performing application-level modifications to the updated code, which reduces the size impact of function shifts. A modified version of the *rsync* [41] algorithm is used to generate a differences script. Dissemination has two stages: first all the nodes are requested to reboot the *reprogramming component* using flooding, then the *Stream* approach is used to distribute the script to all nodes. After the script has been applied to build the new image, and as the new image is loaded into program memory, any indirect function calls (*i.e.*, using a function call table) are replaced by direct calls to improve performance. The nodes then reboot into the new image.

Zephyr has no specific support for autonomic features, but the authors report a very significant reduction in code size, which makes updates more flexible in general.

3.4. Update Fault Detection and Recovery

In IEEE terminology, a software release can introduce a defect, which is a fault encountered during software execution; a fault may cause one or more failures, and a problem may be caused by one or more failures [79]. Fault detection therefore relies on identifying failures in the executing software. Identifying these software failures is not an easy task, but is a fundamental problem for software updates as even *minor* faults in the software can cause application failure, data corruption, or system failure which may prevent further updates.

There are two possible approaches to fault detection. One is to identify all the possible faults, and make sure each is checked for. The problem with this is that there is (and probably never will be) a definitive list of faults. cSimplex [80], for example, identifies some common faults as: malformed messages, messages dropped, packet storms, group management errors, improper data delivery, message duplication, routing loops, and local reception errors. In [81] faults are classified by their symptoms and ease of replication: *Bohrbugs* simple, are easily isolated, and manifest consistently under a well-defined set of conditions, and *Mandelbugs* have complex activation or propagation behavior, are difficult to isolate, and failures are not systematically reproducible. A third category

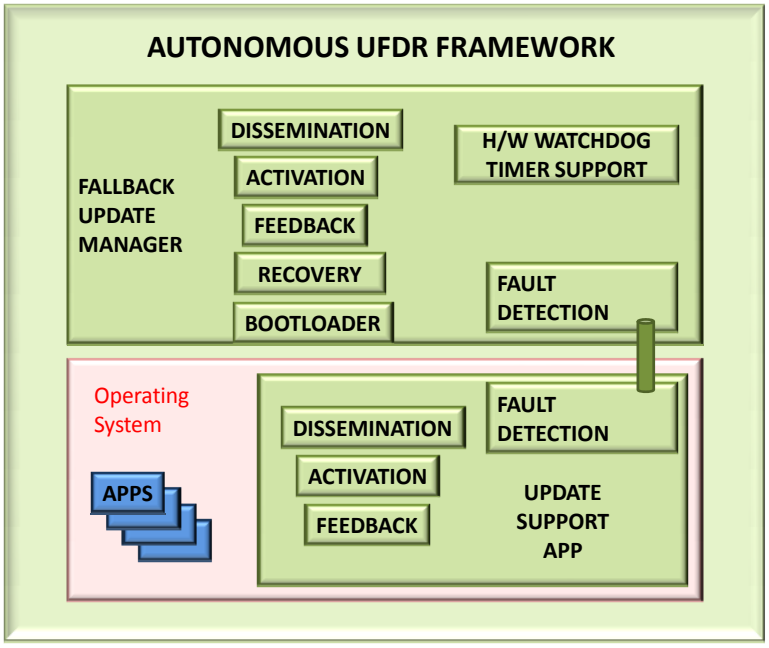
Aging-Related Mandelbugs are a sub-type of *Mandelbugs*, and cause an increasing failure rate or degraded performance.

The other approach, identified in [82] and used in [83], is to focus on a *single* failure, with an easily identified symptom: that of being unable to perform future software updates. The reasoning is that if a software update *can* be performed, the system is not in need of recovery (from a software update viewpoint), and that if it *cannot* be performed, then recovery is needed (irrespective of whether the application is currently performing correctly). This does not detect faults that may result in corrupted or missing application data—but this can be done completely orthogonally to the update failure detection, and application correctness checking can run in parallel with the software update failure detection and recovery functionality.

Software (or configuration) update faults are considered to be fundamentally different from faults in the original software—even though the symptoms may be the same—under the assumption that the original software has been extensively tested prior to deployment, whereas updates are applied post-deployment and may prove harder to test (or be subject to a less rigorous process). No matter how faults are detected, and how recovery takes place, there needs to be high-reliability recovery software available as an ultimate fallback if permanent network failure (*i.e.*, loss of part, or all, of the sensor network) is to be avoided.

A number of models have been presented for detection and recovery from *data* faults in WSNS—for example [84–87]. In this paper we propose a model for detection and recovery from **execution** faults, designed for use in an autonomous software/configuration update environment: WSN Update Fault Detection and Recovery (WSN-UFDR) shown in Figure 6. This model is based on an analysis of the required functionality.

Figure 6. A Model for Software Update Fault Detection and Recovery.



The model includes the usual operating system and applications: this may be in any execution environment (monolithic, modular, script-based, *etc*.)—subject to three requirements: there must be a

watchdog timer reserved for use by the UFDR (the update mechanism may chose to delete any such references, using for example null-operations as in [83]), there must be read-only space reserved to hold the *Fallback Update Manager* which includes the bootloader, and there must be some non-volatile memory reserved (for example in EEPROM). An additional *Update Support Application* (this may alternatively be an O/S module) is defined, which provides software/configuration update support, and interacts with the *Fallback Update Manager*, an independent software component outside of the operating system. Under non-fault conditions, a beacon originating from one or more management stations in injected into the WSN via one or more gateways, and is propagated by the *Support Application* to all nodes, using either the O/S supported network stack, or directly using the Wireless Datalink Layer. Via the special communication channel between the fault detection components this resets the watchdog timer. The UFDR builds a model of beacon arrival rates, and if the beacon fails to arrive after a period (based on this model, and user-defined policy parameters) then the watchdog timer is allowed to expire. The behaviour of the *Recovery* component is then based on the defined recovery policy and the stored fault recovery history—a reboot, a fallback to a previous software version, or ultimately a fallback to the *Fallback Update Manager* image. If the *Update Support Application* uses the O/S stack, network-wide recovery may require all nodes to revert to the *Fallback Update Manager*; but if the *Update Support Application* and the *Fallback Update Manager* use a compatible Datalink, then network-wide recovery may be able to occur with only some nodes reverting. The *Feedback* component provides feedback from all the modules, as shown in Figure 1.

This model provides the basis for surveying recent research results. Only results directly relating to software updates are included; many others (e.g., Neutron [88]) consider the more general problem of detecting and recovering from software failures unrelated to software updates.

3.4.1. Software Update Recovery for Wireless Sensor Networks (SUR)

This work [82] identifies a single, critical, failure symptom—*loss-of-control*—that complements exception-based schemes, and supports failsafe recovery from faults in software updates. It includes a software update recovery mechanism that uses loss-of-control to provide high-reliability, low energy, recovery from software update failures. It has two phases: a *trial* phase (with lower latency) for use immediately following an update, and an *operational* phase (with lower energy) for use once an update has been determined to be stable. A *beacon* injected from a management station and *polycast* throughout the network, using a dedicated flooding algorithm (Flooding with Duplicate-suppression, Missing packet regeneration, and Timeouts), is used to determine loss-of-control. Nodes run in one of four states: OPERATIONAL, TRIAL, FALLBACK, and MAINTENANCE. In the FALLBACK state, a previous version of the software is run; in the MAINTENANCE state, a high-reliability image is executed which supports the software update process (allowing further updates to be made). Transition between the states is controlled by commands in the *beacon* and watchdog timeouts. A protocol is defined for controlling and monitoring the software update process, which allows version information to be propagated, and separates code dissemination from activation. Low energy feedback on the software status of nodes is provided through overhearing; aggregated status is piggybacked onto the *beacon*, and used to report via the gateway node back to a management station.

Most of the components of the UFDR model are covered, but support for ensuring unique access to the Watchdog timer, or persistent data storage for the recovery process, are not included. Also, the entire fault detection software is placed in the O/S image, potentially reducing the reliability of the detection and processes.

3.4.2. Over-the-Air Programming (OTAP)

This was developed to support the requirements of scalability, fail-safety, reliability, and applicability for resource constrained hardware [83]. Two firmware images are stored in EEPROM, one for temporary storage of new software, and one for recovery (a *fallback* image). The OTAP Agent uses existing network stacks for communication: it requires 1-to-1 bi-directional communication with a management station for control, and 1-to-N communication from the management station for data (the update). Before dissemination, a preparation request is sent to the selected targets. The update is then transmitted using the 1-to-N network functionality, in multiple packets sized according to the underlying protocol capabilities, using Reed-Solomon coding for redundancy to reduce the performance impacts of unreliable transmission, which are restored using Gaussian elimination and stored in the temporary image section of EEPROM. Afterwards, the network manager polls each target for missing segments, which are re-sent using 1-to-N communication to enable the efficiencies of overhearing. The bootloader is designed to be fail-safe, the image is verified, power-failure and recovery will never result in an inconsistent state (stored in EEPROM), and unexpected resets are handled. Beacons are periodically broadcast by the management station; a sequence of missing or invalid beacons lead to a reset and a reprogram—connectivity loss is used as an indicator for a possible software failure. The OTAP Agent runs as an application, receiving these beacons (sent using the 1-to-N protocol), and storing them in an area of shared memory to be checked by the bootloader periodically (using its reserved timer). The hardware watchdog and a timer are reserved for the bootloader; updates are checked to ensure they do not attempt to access the watchdog timer, and the interrupt vector table is overwritten to ensure the bootloader maintains access for resets.

Direct support for autonomous operation is provided through the use of beacons (as in [82]) where loss of software update communication indicates a software fault and results in recovery to a fallback image. Feedback of the dissemination is provided through end-to-end ACKs, and a network-wide fallback can be achieved by halting the transmission of beacons from the management station.

3.4.3. cSimplex

cSimplex [80] supports the reliable updating of multicast communication software in Wireless Sensor Networks. Fault detection is performed by monitoring the new communications software for entry into inadmissible states, using statistical checks and a stability definition. Recovery takes place by reverting to a well-tested, reliable, communications *safety module*. When a new (or *experimental*) communications software module is uploaded, its behavior is compared to that of a baseline (high assurance) module. Three basic critical requirements are introduced: *agreement* (all valid nodes receive the same set of messages), *validity* (all messages sent by valid nodes are received, and *integrity* (no invalid messages are received). From this, eight basic invariants are defined for a *stable* protocol—these can be augmented by application-specific requirements (e.g., timeouts, performance,

application-specific features such as causal ordering). Received and transmitted messages are checked, and cSimplex state is added to every message (in a special header); cSimplex *Session Messages* are also exchanged between nodes. Stability checking is distributed over the network, using statistical methods, evaluating: the *Long Term Error Rate*, *Per-Period Error Rate*, *Per-Period Error Rate with History*, and a *Multiple-Period Method*. Following fault detection, a publish-subscribe scheme is used to allow communication software to be dynamically replaced without shutting down the software.

The cSimplex system also defines a complementary application/communication fault detection and recovery mechanism that could run in parallel to UFDR.

3.4.4. Autonomous and Distributed Node Recovery (ADNR)

Detection of, and recovery from, of faults deliberately inserted into nodes in a WSN is considered in this work [89]. A network is divided into clusters, and for intrusion detection (IDS) each node observes the behavior of its cluster neighbours. The IDS may be of any form—for example: *malignancy counters*, active probing, rule-based, reputation-based, or locally verified correctness properties. Two key protection features are identified: *logical protection* to ensure that the recovery system remains available and uncompromised, no matter what malignant software does (for example, using a dedicated hardware interrupt vectoring to write-protected memory); and *physical protection* to make the hardware (reasonably) tamper-proof. As long as the recovery component of a faulty node is intact, recovery is through a restart, or a code update. If the node does not respond to these measures, then it is expelled through a majority decision in order to avoid inconsistencies. Recovery takes places in two phases: an *accusation phase* in which nodes accuse all neighbours which have been identified as faulty, and a *recovery phase* whereby if a node is accused by at least two or three of its neighbours, countermeasures are executed. This may consist of a node reloading the correct software update from its neighbours, using hash codes to ensure the integrity of each fragment, and using rounds to distribute the load. Or a node may reboot, or shut down permanently.

Autonomous and Distributed Node Recovery defines a complementary mechanism, with many similarities, targeted at identifying and recovering from malicious code deliberately inserted into a node as an update.

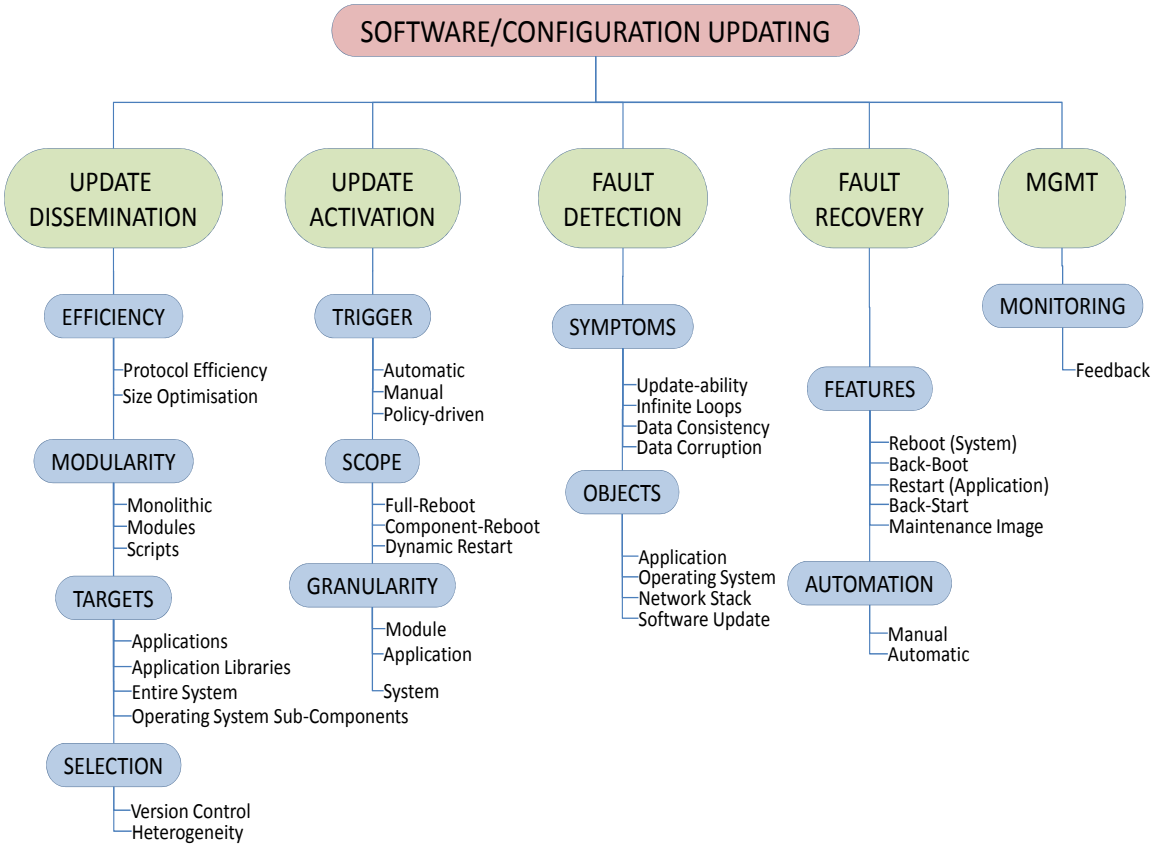
4. Taxonomy and Comparison

A framework for the classification of Software/Configuration Update features in a WSN, covering the four key areas of Dissemination, Activation, Fault Detection, and Fault Recovery is shown in Figure 7, identifying key characteristics of each.

Notes on Figure 7:

- Reboot (System)—reboot the current (failed) image after a failure
- Back-Boot—reboot, going back to the previous image after a failure
- Restart (Application)—restart the current (failed) application component after a failure
- Back-Start—restart, going back to the previous application component after a failure
- Maintenance Image—rebooting a special image after a failure that supports software updating, but not the WSN application

Figure 7. Taxonomy of Update Features.



A comparison of the features shown in Figure 7, as addressed by the papers surveyed here, is shown in Table 1.

This classification of features clearly shows the major streams of research to date: dissemination and activation. Fault detection and recovery are, in general, relatively recent streams; and management features relevant to the update process are only just starting to receive attention.

Table 1. Software and Configuration Updates (SCU) Feature Comparison.

	Dissemination											Activation									Detection									Recovery									Mgmt	
	Efficiency		Modularity			Targets				Selection		Granularity		Scope				Trigger			Symptoms				Objects				Features						Automation		Monitoring	
Name	Protocol Efficiency	Traffic Reduction	Monolithic	Modular	Scripts	Applications	Application Libraries	Entire System	O/S Subcomponents	Version Control	Heterogeneity	Module	Application	System	Full-Reboot	Component-Reboot	Dynamic Restart	Automatic	Manual	Policy Driven	Update-ability	Infinite Loops	Data Consistency	Data Corruption	Application	Operating System	Network Stack	Software Update	Reboot (system)	Back-Boot	Reboot Application	Back-Start	Maintenance Image	Manual	Automatic	Feedback
CORD [28]	✓		✓					✓		✓	✓																										
Deluge [29]	✓		✓					✓		✓				✓	✓			✓																			
DSN [27]																																					
ECD [34]	✓		✓					✓		✓				✓	✓			✓																			
Freshnet [35]	✓		✓					✓		✓																											
GCP [36]	✓		✓					✓		✓																											
Impala [37]				✓		✓	✓			✓		✓				✓		✓																			
MCP [38]	✓		✓					✓		✓	✓			✓	✓			✓																			
MDeluge [31]	✓		✓					✓		✓	✓			✓	✓			✓																			
MNP [32]	✓		✓					✓		✓				✓	✓			✓																			
MOAP [7]	✓		✓					✓		✓				✓	✓			✓																			
PDM [40]	✓		✓					✓		✓	✓			✓	✓			✓																			

Table 1. *Cont.*

Name	Dissemination											Activation									Detection								Recovery									Mgmt
	Efficiency		Modularity			Targets				Selection		Granularity		Scope			Trigger			Symptoms				Objects				Features					Automation		Monitoring
	Protocol Efficiency	Traffic Reduction	Monolithic	Modular	Scripts	Applications	Application Libraries	Entire System	O/S Subcomponents	Version Control	Heterogeneity	Module	Application	System	Full-Reboot	Component-Reboot	Dynamic Restart	Automatic	Manual	Policy Driven	Update-ability	Infinite Loops	Data Consistency	Data Corruption	Application	Operating System	Network Stack	Software Update	Reboot (system)	Back-Boot	Reboot Application	Back-Start	Maintenance Image	Manual	Automatic	Feedback
Rate/ACKless Deluge [30]	✓	✓						✓		✓				✓	✓			✓																		
Trickle [13]	✓				✓	✓				✓		✓					✓		✓																	
XNP [33]			✓					✓						✓	✓					✓																
AdapCode [46]	✓	✓								✓																										
Delta Deluge [45]	✓	✓	✓					✓		✓				✓	✓			✓																		
INP [44]		✓		✓				✓		✓				✓	✓			✓																		
PEANC [47]	✓	✓								✓																										
R-Code [48]	✓	✓								✓																										
Reijers [8]		✓	✓					✓		✓				✓	✓			✓																		
RIL [9]		✓		✓		✓				✓				✓	✓			✓																		
Agilla [51]		✓			✓	✓			✓				✓			✓		✓																		

Table 1. *Cont.*

Name	Dissemination											Activation							Detection								Recovery									Mgmt
	Efficiency		Modularity			Targets				Selection		Granularity			Scope			Trigger			Symptoms			Objects				Features						Automation		Monitoring
	Protocol Efficiency	Traffic Reduction	Monolithic	Modular	Scripts	Applications	Application Libraries	Entire System	O/S Subcomponents	Version Control	Heterogeneity	Module	Application	System	Full-Reboot	Component-Reboot	Dynamic Restart	Automatic	Manual	Policy Driven	Update-ability	Infinite Loops	Data Consistency	Data Corruption	Application	Operating System	Network Stack	Software Update	Reboot (system)	Back-Boot	Reboot Application	Back-Start	Maintenance Image	Manual	Automatic	Feedback
COMiS [61]		✓			✓	✓				✓			✓			✓		✓																	
Contiki [62]		✓		✓		✓				✓		✓					✓	✓																	✓
DVM [74]		✓			✓	✓				✓		✓					✓	✓																	
Dynamic TOS [73]		✓		✓		✓	✓		✓	✓		✓					✓	✓																	
ELUS [75]		✓		✓		✓	✓			✓		✓					✓	✓																	
FlexCup [76]		✓		✓		✓	✓			✓		✓			✓			✓																	
FiGaRo [63]	✓	✓		✓		✓	✓			✓	✓	✓					✓	✓																	✓
HERMES [70]		✓		✓		✓				✓		✓					✓	✓					✓		✓								✓		✓
MAF [59]		✓			✓	✓							✓				✓	✓																	
Maté [58]		✓			✓	✓				✓			✓				✓	✓																	
Melete [60]		✓			✓	✓				✓	✓		✓				✓	✓																	
MinTax [71]		✓		✓		✓				✓	✓			✓	✓			✓																	
MOS [52]		✓	✓	✓		✓	✓	✓	✓																										
Pushpin [68]		✓		✓		✓	✓			✓		✓					✓	✓																	

Table 1. *Cont.*

Name	Dissemination										Activation							Detection							Recovery								Mgmt
	Efficiency		Modularity			Targets				Selection	Granularity			Scope			Trigger			Symptoms				Objects				Features					Automation		Monitoring
	Protocol Efficiency	Traffic Reduction	Monolithic	Modular	Scripts	Applications	Application Libraries	Entire System	O/S Subcomponents	Version Control	Heterogeneity	Module	Application	System	Full-Reboot	Component-Reboot	Dynamic Restart	Automatic	Manual	Policy Driven	Update-ability	Infinite Loops	Data Consistency	Data Corruption	Application	Operating System	Network Stack	Software Update	Reboot (system)	Back-Boot	Reboot Application	Back-Start	Maintenance Image	Manual	Automatic	Feedback
REAP [64]		✓		✓		✓	✓			✓	✓	✓	✓				✓	✓																	
SINA [65]		✓			✓	✓							✓				✓	✓																	
ScatterWeb [49]		✓		✓		✓	✓						✓			✓		✓	✓																
SensorWare [69]		✓			✓	✓							✓				✓	✓																	
SOS [50]		✓		✓		✓				✓		✓					✓	✓																	
SP [66]					✓	✓							✓				✓	✓																	
Stream [77]		✓	✓					✓		✓				✓	✓			✓																	
uFlow [72]		✓		✓		✓	✓					✓		✓			✓	✓																	
Zephyr [78]		✓	✓					✓		✓				✓	✓			✓																	
ADNR [89]			✓					✓		✓			✓	✓									✓		✓				✓		✓			✓	
cSimplex [80]				✓					✓			✓						✓					✓			✓			✓					✓	
SUR [82]														✓				✓			✓						✓	✓	✓	✓	✓	✓		✓	✓
OTAP [83]		✓		✓				✓						✓				✓	✓		✓								✓			✓		✓	✓

5. *Lacunae*

Autonomic behavior has been identified as a valuable, if not essential, feature for Wireless Sensor Networks in general [90]. In [5], the conclusion is that autonomous behavior is required for over-the-air updates in a Wireless Sensor Network. Reviewing a large number of research results, with regard to this requirement, identifies the following *lacunae*:

(a) Fault Detection and Recovery.
(b) Policy-driven Update Management.
(c) Feedback (of the SCU process).
(d) Update Planning.
(e) Configuration Management (including Version Control).

5.1. Fault Detection and Recovery

Self-managing and reliable software and configuration updating requires fully automatic detection of faults and recovery. A *one-symptom* approach to this is described in the Update Failure Detection and Recovery Model presented in this paper. There are however three general classes of faults associated with a software update:

1. a fault in the original software,
2. a fault in the updated software, and
3. a fault in the interaction between the two versions.

Further research is required, extending the work presented in Section 3.4, to validate the model, and provide reference implementations to determine performance characteristics. Research is also required into these three general classes of fault, not only to discover whether identifying the symptoms associated with the different classes can be of value in fault *detection*, but also whether they can be of value in fault *recovery*. For a particular recovery policy, there may be different actions available for handling recovery depending on the class of fault.

5.2. Policy-Driven Update Management

For autonomous behavior to meet the needs of the user, it requires a *policy* to be specified. Some of the issues, such as an update policy might address, are:

- Connectivity: what to do if only some nodes receive an update, if connectivity to neighbours is maintained but is lost to the base-stations, if a software update fails repeatedly, *etc*.
- Performance: the required update latency, the allowed application performance impact, *etc*.
- Energy: the allowed energy impact, energy-performance tradeoffs, minimum node energy levels, what to do if only some nodes have the energy to perform an update, *etc*.
- Dependability: checks to be performed before activating the update (integrity, version mismatches, platform mismatches), *etc*.

Further research is required to fully determine the required policies, and then to implement these policies both on the management node, distributed throughout the WSN, and locally on each node. In

addition, research is required into a *policy specification language* for policy driven update behaviour in a WSN environment.

5.3. Feedback

Both for manual and automatic software/configuration updates, feedback on the software status (as opposed to the application or environment status) is critical in order to control and manage the updates. This feedback needs to be available at different levels of granularity, from network-wide statistics, to identifying the specific software and configuration active on an individual node. This feedback is needed both to control the current update activity, and to provide feedback for use in planning for future updates. All software/configuration update components need to provide this feedback:

- *dissemination*—which nodes have what software available,
- *activation*—what version is currently running on each node,
- *fault-detection*—what faults have been detected on which nodes, and
- *recovery*—what is the recovery status of different nodes: *i.e.*, which have rebooted the current software, fallen back to a previous version, or dropped back to the maintenance image.

Further research is required on how to efficiently monitor the software state of WSN nodes. The traffic pattern is closer to the more familiar many-to-one pattern used by WSN applications to report data, and it is an open question how existing results can best be applied to monitoring software/ configuration updates. Also, this feedback feature should probably be incorporated into a broader scheme for the monitoring, control and management of WSNs.

5.4. Update Planning

This allows the energy costs of different update options with the current network configuration and status to be determined in advance. To support more frequent reprogramming, energy consumption models (such as those presented in [91,92]) need to be developed to allow the energy costs of different update policies to be compared, either analytically or through simulation, prior to the initiation of an update. The dissemination and execution policies then need to be controlled through power-aware protocols [93], as has been found to be effective in the development of datalink and routing protocols for wireless sensor networks. Power-awareness will provide one of the feedback paths required to provide an intervention-free software update environment for wireless sensor networks. Update planning includes the determination of the best update injection strategies: how to select the optimal injection strategy: simulating a base station, sending the update to a base station, sending the update to a number of seed nodes, and sending the update to individual nodes.

5.5. Configuration Management

Most systems currently use a simple monotonically increasing version number for controlling the software update process—nodes update if they become aware of a higher version number. This approach has three problems associated with it:

1. it does not cope with heterogeneous deployments;

2. the update facility is disabled once the version number reaches its maximum value (unlikely in normal use, this might occur through a mistake);
3. and it does not handle configuration data (as opposed to code).

Real deployments of heterogeneous WSNs will require more complex configuration management. Namespace management is required to provide full control in a heterogeneous environment: for different hardware platforms, or for different required functionality on identical hardware platforms. Probably a more flexible scheme than just *individual*, *group-based*, and *network-wide* updates should be supported. Version control is required to ensure the latest software/configuration versions are downloaded, and to prevent *intra-node* version mismatch problems (for both code and data). Enhanced activation may be required to control the ordering of activation of a software update throughout a network in order to avoid *inter-node* version incompatibilities. Further research is required to determine exactly what is required, and how best to provide it in the resource constrained environment of a WSN.

6. Conclusions

This paper provides an extensive and novel survey of software update support for Wireless Sensor Networks. The new Software Update Taxonomy presented in the paper, the feature comparison in the context of this taxonomy, the focus on autonomous behavior, and the breadth, differentiate it from previous surveys.

A summary is provided of a large number of different solutions that have been developed for WSN software updates, allowing them to be compared from the viewpoint of autonomous behavior. This review identifies three different classes of how software updates are executed (or made available for execution): static/monolithic updates, dynamic/modular updates, and dynamic/mobile agent-based updates. This review also identifies two classes of update formats: incremental updates (which are dependent on the current version), and full updates (which are not). Finally this review has identified two connectivity solutions: using the sensor network itself, and using a parallel maintenance network.

These different solutions reveal the fundamental research challenge in WSN software updating: to bring all this together into a cohesive, and self-managing, framework. Most of the discrete solutions are optimal in some situations, but not in others: an autonomous framework needs the ability to react both to different environments, and to the current state of the WSN, in order to operate dependably, efficiently, effectively, and securely. The remoteness, inaccessibility and scale planned for sensor networks, along with need for high dependability, argue for an autonomic approach to solving the software update problem. Identifying this key issue, and identifying the associated research lacunae, is the principal contribution of this paper.

Users (or, rather, programmers) will, at least in the short-to-medium term, still need to develop the code for software updates; but once injected into the sensor network, the software updates should be handled with minimal user intervention.

In particular, the WSN as a whole should recover from failures associated with the software/ configuration update process. Detecting failures is not an easy task, and this paper reinforces the conclusion that it is an easier task to identify that a software update component has lost communications

with a management station than to detect every possibly type of failure. A model for update fault detection and recovery in Wireless Sensor Networks is presented.

Once software update capability is available as a reliable service, it is likely to be heavily used in order to optimize the use of deployed networks. We hope that our paper will encourage the necessary research to enable this vision.

Conflicts of Interest

The authors declare no conflict of interest.

References

1. Han, C.-C.; Kumar, R.; Shea, R.; Srivastava, M. Sensor network software update management: A survey. *Int. J. Netw. Manag.* **2005**, *15*, 283–294.
2. Brown, S.; Sreenan, C.J. A new model for updating software in wireless sensor networks. *IEEE Netw.* **2006**, *20*, 42–47.
3. Wang, Q.; Zhu, Y.; Cheng, L. Reprogramming wireless sensor networks: Challenges and approaches. *IEEE Netw.* **2006**, *20*, 48–55.
4. Mottola, L.; Picco, G.P. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Comput. Surv.* **2011**, *43*, 19:1–19:51.
5. Brown, S.; Sreenan, C.J. *Updating Software in Wireless Sensor Networks: A Survey*; Technical Report UCC-CS-2006-13-07, University College Cork: Cork, Ireland, 2006.
6. Habermann, A.W. Dynamically Modifiable Distributed Systems. In Proceedings of a Workshop on Distributed Sensor Nets, Pittsburgh, PA, USA, December 1978; Carnegie-Mellon University: Pittsburgh, PA, USA; pp. 111–114.
7. Stathopoulos, T.; Heidemann, J.; Estrin, D. *A Remote Code Update Mechanism for Wireless Sensor Networks*; Technical Report CENS #30 for the Center for Embedded Network Sensing, UCLA: Los Angeles, CA, USA, 2003.
8. Reijers, N.; Langendoen, K. Efficient Code Distribution in Wireless Sensor Networks. In Proceedings of the 2nd ACM International Workshop on Wireless Sensor Networks and Applications, San Diego, CA, USA, 19 September 2003; pp. 60–67.
9. Koshy, J.; Pandy, R. Remote Incremental Linking for Energy-Efficient Reprogramming of Sensor Networks. In Proceedings of the 2nd European Workshop on Wireless Sensor Networks, Istanbul, Turkey, 31 January–2 February 2005; pp. 354–365.
10. Sha, L.; Rajkumar, R.; Gagliardi, M. Evolving Dependable Real-Time Systems. In Proceedings of the IEEE Conference on Aeropsace Applications, Aspen, CO, USA, 3–10 February 1996; pp. 335–346.
11. Stankovic, J.A.; Abdelzaher, T.E.; Lu, C.; Sha, L.; Hou, J.C. Real-time communication and coordination in embedded sensor networks. *Proc. IEEE* **2003**, *91*, 1002–1022.
12. Liu, T.; Sadler, C.S.; Zhang, P.; Martonosi, M. Implementing Software on Resource-Constrained Mobile Sensors: Experiences with Impala and ZebraNet. In Proceedings of the 2nd International Conference on Mobile Systems, Applications and Services, Boston, MA, USA, 6–9 June 2004; pp. 256–269.

13. Levis, P.; Patel, N.; Culler, D.; Shenker, S. Trickle: A Self Regulating algorIthm for Code Propagation and Maintenance in Wireless Sensor Networks. In Proceedings of the First USENIX/ACM Symposium on Network Systems Design and Implementation, San Francisco, CA, USA, 29–31 March 2004; pp. 15–28.
14. Kavitha, T.; Sridharan, D. Security vulnerabilities in wireless sensor networks: A survey. *J. Inf. Assur. Secur.* **2010**, *5*, 31–44.
15. Law, Y.W.; Zhang, Y.; Jin, J.; Palaniswami, M.; Havinga, P. Secure rateless deluge: Pollution-resistant reprogramming and data dissemination for wireless sensor networks. *EURASIP J. Wirel. Commun. Netw.* **2011**, *685219*, 1–22.
16. Devanbu, P.; Gertz, M.; Stubblebine, S. Security for Automated, Distributed Configuration Management. In Proceedings of the ICSE 99 Workshop on Software Engineering over the Internet, Los Angeles, CA, USA, 25 April 1999; pp. 1–11.
17. Racherla, G.; Saha, D. Security and Privacy Issues in Wireless and Mobile Computing. In Proceedings of the IEEE International Conference on Personal Wireless Communications, Hyderabad, India, 17–20 December 2000; pp. 509–513.
18. Sha, L. Upgrading Real-Time Control Software in the Field. *Proc. IEEE* **2003**, *91*, 1131–1140.
19. Avizienis, A.; Laprie, J.-C.; Randell, R.; Landwehr, C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.* **2004**, *1*, 11–33.
20. Kephart, J.O.; Chess, D.M. The vision of autonomic computing. *IEEE Comput.* **2003**, *36*, 41–50.
21. Intangonwiwat, C.; Govindan, R.; Estrin, D. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In Proceedings of the 6th Annual Conference on Mobile Computing and Networking, Boston, MA, USA, 6–11 August 2000; pp. 56–67.
22. Stann, F.; Heidemann, J. RMST: Reliable Data Transport in Sensor Networks. In Proceedings of the 1st IEEE International Workshop on Sensor Network Applications and Protocols, Anchorage, AK, USA, 11 May 2003; pp. 102–112.
23. Wan, C.Y.; Campbell, A.T.; Krishnamurthy, L. PSFQ: A Reliable Transport Protocol for Wireless Sensor Networks. In Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications, Atlanta, Georgia, USA, 28 September 2002; pp. 1–11.
24. Kulkarni, S.S.; Arumugam, M. *Infuse: A TDMA Based Data Dissemination Protocol for Sensor Networks*; Technical Report MSU-CSE-04-46 for the Department of Computer Science and Engineering, Michigan State University: East Lansing, MI, USA, 2004.
25. Naik, V.; Arora, A.; Sinha, P.; Zhang, H. Sprinkler: A Reliable and Energy Efficient Data Dissemination Service for Wireless Embedded Devices. In Proceedings of the 26th IEEE International Real-Time Systems Symposium, Miami, FL, USA, 5–8 December 2005; pp. 286–296.
26. Yick, J.; Mukherjee, B.; Ghosal, D. Wireless sensor network survey. *Comput. Netw.* **2008**, *52*, 2292–2330.
27. Beutel, J.; Dyer, M.; Meier, L.; Ringwald, M.; Thiele, L. *Next-Generation Deployment Support for Sensor Networks*; Technical Report TIK 207 for the Computer Engineering and Networks Lab, Swiss Federal Institute of Technology (ETH): Zurich, Switzerland, 2004.
28. Huang, L.; Setia, S. CORD: Energy-Efficient Reliable Bulk Data Dissemination in Sensor Networks. In Proceedings of the 27th Conference on Computer Communications, Phoenix, AZ, USA, 13–18 April 2008; pp. 574–582.

29. Hui, J.; Culler, D. The Dynamic Behavior of a Data Dissemination Protocol for Network Reprogramming at Scale. In Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, Baltimore, MD, USA, 3–5 November 2004; pp. 81–94.
30. Hagedorn, A.; Starobinski, D.; Trachtenberg, A. Rateless Deluge: Over-the-Air Programming of Wireless Sensor Networks Using Random Linear Codes. In Proceedings of the 7th International Conference on Information Processing in Sensor Networks, Washington, DC, USA, 22–24 April 2008; pp. 457–466.
31. Zheng, X.; Sarikaya, B. Task dissemination with multicast deluge in sensor networks. *IEEE Trans. Wirel. Commun.* **2009**, *8*, 2726–2734.
32. Kulkarni, S.S.; Wang, L. MNP: Multihop Network Reprogramming Service for Sensor Networks. In Proceedings of the 25th IEEE International Conference on Distributed Computing Systems, Baltimore, MD, USA, 14–16 June 2005; pp. 7–16.
33. Jeong, J.; Kim, S.; Broad, A. *Network Reprogramming*; University of California: Berkeley, CA, USA, 2003.
34. Dong, W.; Liu, Y.; Wang, C.; Liu, X.; Chen, C.; Bu, J. Link Quality Aware Code Dissemination in Wireless Sensor Networks. In Proceedings of the 19th IEEE International Conference on Network Protocols, Vancouver, BC, Canada, 17–20 October 2011; pp. 89–98.
35. Krasniewski, M.D.; Panta, R.K.; Bagchi, S.; Yang, C.L.; Chappell, W.J. Energy-efficient on-demand reprogramming of large-scale sensor networks. *ACM Trans. Sensor Netw.* **2008**, *4*, 2:1–2:38.
36. Busnel, Y.; Bertier, M.; Fleury, E.; Kermarrec, A.M. GCP: Gossip-Based Code Propagation for Large-Scale Mobile Wireless Sensor Networks. In Proceedings of the 1st International Conference on Autonomic Computing and Communication Systems, Rome, Italy, 28–30 October 2007; pp. 11:1–11:5.
37. Liu, T.; Martonosi, M. Impala a Middleware System for Managing Autonomic, Parallel Sensor Systems. In Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, CA, USA, 11–13 June 2003; pp. 107–118.
38. Li, W.; Zhang, Y.; Childers, B. MCP: An Energy-Efficient Code Distribution Protocol for Multi-Application WSNs. In Proceedings of the 5th IEEE International Conference on Distributed Computing in Sensor Systems, Marina del Rey, CA, USA, 8–10 June 2009; pp. 259–272.
39. Li, W. Software Update Management in Wireless Sensor Networks. PhD Thesis, University of Pittsburgh, PA, USA, February 2011.
40. Wen, T.; Li, Z.; Li, Q.F. An Efficient Code Distribution Protocol for OTAP in WSNs. In Proceedings of the 5th International Conference on Wireless Communications, Networking and Mobile Computing, Beijing, China, 24–26 September 2009; pp. 3980–3983.
41. Trigdell, A.; Mackerras, P. *The Rsync Algorithm*; Technical Report TR-CS-96–05 for the Australian National University: Canberra, Australia, 1998.
42. Bettini, L.; de Nicola, R.; Loreti, M. Software Update via Mobile Agent Based Programming. In Proceedings of the 2002 ACM Symposium on Applied Computing, Madrid, Spain, 10–14 March 2002; pp. 32–36.
43. Ahlswede, R.; Cai, N.; Li, S.Y.; Yeung, R.W. Network information flow. *IEEE Trans. Inf. Theory* **2000**, *46*, 1204–1216.

44. Jeong, J.; Culler, D. Incremental Network Programming for Network Sensors. In Proceedings of the 1st Annual IEEE Communications Society Conference on Sensor and *Ad Hoc* Networks and Communications, Santa Clara, CA, USA, 4–7 October 2004; pp. 25–33.
45. Von Rickenbach; P. Wattenhofer, R. Decoding Code on a Sensor Node. In Proceedings of the 4th IEEE International Conference on Distributed Computing in Sensor Systems, Santorini, Greece, 11–14 June 2008; pp. 400–414.
46. Hou, I.-H.; Tsai, Y.-E.; Abdelzaher, T.F.; Gupta, I. AdapCode: Adaptive Network Coding for Code Updates in Wireless Sensor Networks. In Proceedings of the 27th Conference on Computer Communications, Phoenix, AZ, USA, 13–18 April 2008; pp. 1517–1525.
47. Shwe, H.Y.; Adachi, F. Power Efficient Adaptive Network Coding in Wireless Sensor Networks. In Proceedings of the IEEE International Conference on Communications, Kyoto, Japan, 5–9 June 2011; pp. 1–5.
48. Yang, Z.; Li, M.; Lou, W. R-code: Network Coding Based Reliable Broadcast in Wireless Mesh Networks with Unreliable Links. In Proceedings of Global Telecommunications Conference, Honolulu, HI, USA, 30 November–4 December 2009; pp. 1–6.
49. Schiller, J.; Liers, A.; Ritter, H.; Winter, R.; Voigt, T. ScatterWeb-Low Power Sensor Nodes and Energy Aware Routing. In Proceedings of the 38th Hawaii International Conference on System Sciences, Waikkoloa, HI, USA, 3–6 January 2005; pp. 1–9.
50. Han, C.-C.; Kumar, R.; Shea, R.; Kohler, E.; Srivastava, M. A Dynamic Operating System for Sensor Nodes. In Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services, Seattle, WA, USA, 6–8 June 2005; pp. 163–176.
51. Fok, C.-L.; Roman, G.-C.; Lu, C. Mobile Agent Middleware for Sensor Networks: An Application Case Study. In Proceedings of the 4th International Symposium on Information Processing in Sensor Networks, Los Angeles, CA, USA, 25–27 April 2005; pp. 382–387.
52. Abrach, H.; Bhatti, S.; Carlson, J.; Dai, H.; Rose, J.; Sheth, A.; Shucker, B.; Deng, J.; Han, R. MANTIS: System Support for MultimodAl NeTworks of *In-situ* Sensors. In Proceedings of the 2nd ACM International Workshop on Wireless Sensor Networks and Applications, San Diego, CA, USA, 19 September 2003; pp. 50–59.
53. Madden, S.; Franklin, M.J.; Hellerstein, J.M. TAG: A Tiny Aggregation Service for *Ad-Hoc* Sensor Networks. *SIGOPS Oper. Syst. Rev.* **2002**, *36*, 131–146.
54. Hill, J.; Horton, M.; Kling, R.; Krishnamurthy, L. The platforms enabling wireless sensor networks. *Commun. ACM* **2004**, *47*, 41–46.
55. Hempstead, M.; Lyons, M.J.; Brooks, D.; Wei, G.-Y. Survey of hardware platforms for wireless sensor networks. *J. Low Power Electron.* **2008**, *4*, 11–20.
56. Dwivedi A.K.; Vyas, O.P. Wireless Sensor Network: At a Glance. In *Recent Advances in Wireless Communications and Networks*; Lin, J.-C., Ed.; InTech: Rijeka, Croatia, 2011; pp. 299–326.
57. Karani, M.; Kale, A.; Kopekar, A. Wireless Sensor Network Hardware Platforms and Multi-Channel Communication Protocols: A Survey. In Proceedings on 2nd National Conference on Information and Communication Technology, New York, NY, USA, September 2011; pp. 20–23.
58. Levis, P.; Culler, D. Maté: A virtual machine for tiny networked sensors. *ACM SIGPLAN Not.* **2002**, *37*, 85–95.

59. Szumel, L.; LeBrun, J.; Owens, J.D. Towards a Mobile Agent Framework for Sensor Networks. In Proceedings of the 2nd IEEE Workshop on Embedded Networked Sensors, Sydney, Australia, 30–31 May 2005; pp. 79–88.
60. Yu, Y.; Rittle, L.J.; Bhandari, V.; LeBrun, J.B. Supporting Concurrent Applications in Wireless Sensor Networks. In Proceedings of the 4th International Conference on Embedded Networked Sensor Systems, Boulder, CO, USA, 31 October–3 November 2006; pp. 139–152.
61. Janakiram, D.; Venkateswarlu, R.; Nitin, S. *COMiS: Component Oriented Middleware for Sensor Networks*; Department of Computer Science and Engineering, Indian Institute of Technology: Chennai, India, 2005.
62. Dunkels, A.; Grovall, B.; Voigt, T. Contiki—A Lightweight and Flexible Operating System for Tiny Networked Sensors. In Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks, Tampa, FL, USA, 16–18 November 2004; pp. 455–462.
63. Mottola, L.; Picco, G.P.; Sheikh, A.A. FiGaRo: Fine-Grained Software Reconfiguration for Wireless Sensor Networks. In Proceedings of the 5th European Wireless Sensor Networks Conference (EWSN 2008), Bologna, Italy, 30 January–1 February 2008; pp. 286–304.
64. Brooks, R.R.; Keiser, T.E. Mobile code daemons for networks of embedded systems. *IEEE Internet Comput.* **2004**, *8*, 72–79.
65. Shen, C.-C.; Srisathapornphat, C.; Jaikaeo, C. Sensor information networking architecture and applications. *IEEE Pers. Commun.* **2001**, *8*, 52–59.
66. Iftode, L.; Borcea, C.; Kochut, A.; Intanagonwiwat, C.; Kremer, U. Programming Computers Embedded in the Physical World. In Proceedings of the 9th IEEE Workshop on Future Trends of Distributed Computing Systems, San Juan, Puerto Rico, 28–30 May 2003; pp. 78–85.
67. Von Eicken, T.; Culler, D.; Oldstein, S.; Schauser, K. Active Messages: A Mechanism for Integrated Communication and Computation. In Proceedings of the 19th International Symposium on Computer Architecture, Queensland, Australia, 26–28 May 1992; pp. 256–266.
68. Lifton, J.; Seetharam, D.; Broxton, M.; Paradiso, J. Pushpin computing system overview: A Platform for Distributed, Embedded, Ubiquitous Sensor Networks. In Proceedings of the 1st International Conference on Pervasive Computing, Zurich, Switzerland, 26–28 August 2002; pp. 139–151.
69. Boulis, A.; Srivastava, M. Design and Implementation of A Framework for Efficient and Programmable Sensor Networks. In Proceedings of the 1st International Conference on Mobile Systems, Applications, and Services, San Francisco, CA, USA, 5–8 May 2003; pp. 187–200.
70. Kothari, N.; Nagaraja, K.; Raghunathan, V.; Sultan, F.; Chakradhar, S. Hermes: A Software Architecture for Visibility and Control in Wireless Sensor Network Deployments. In Proceedings of Information Processing in Sensor Networks, St. Louis, MI, USA, 22–24 April 2008; pp. 395–406.
71. Galos, M.; Navarro, D.; Mieyeville, F.; O'Connor, I. Energy-Aware Software Updates in Heterogeneous Wireless Sensor Networks. In Proceedings of the IEEE 9th International Conference on New Circuits and Systems Conference, Bordeaux, France, 26–29 June 2011; pp. 333–336.
72. Chi, T.Y.; Wang, W.C.; Kuo, S.Y. uFlow: Dynamic Software Updating in Wireless Sensor Networks. In Proceedings of the 8th International Conference on Ubiquitous Intelligence and Computing, Banff, AB, Canada, 2–4 September 2011; pp. 405–419.

73. Munawar, W.; Alizai, M.H.; Landsiedel, O.; Wehrle, K. Dynamic TinyOS: Modular and Transparent Incremental Code-Updates for Sensor Networks. In Proceedings of the IEEE International Conference on Communications, Cape Town, South Africa, *23–27* May 2010; pp. 1–6.
74. Balani, R.; Han, C.C.; Rengaswamy, R.K.; Tsigkogiannis, I.; Srivastava, M. Multi-Level Software Reconfiguration for Sensor Networks. In Proceedings of the 6th ACM & IEEE International Conference on Embedded software, Seoul, Korea, 22–25 October 2006; pp. 112–121.
75. Steiner, R.; Gracioli, G.; de Cassia Cazu Soldi, R.; Frohlich, A.A. An Operating System Runtime Reprogramming Infrastructure for WSN. In Proceedings of the IEEE Symposium on Computers and Communications, Cappadocia, Turkey, 1–4 July 2012; pp. 000621–000624.
76. Marrón, P.J.; Gauger, M.; Lachenmann, A.; Minder, D.; Saukh, O.; Rothermel, K. FlexCup: A Flexible and Efficient Code Update Mechanism for Sensor Networks. In Proceedings of the Third European Workshop, EWSN 2006, Zurich, Switzerland, 13–15 February 2006; pp. 212–227.
77. Panta, R.K.; Khalil, I.; Bagchi, S. Stream: Low Overhead Wireless Reprogramming for Sensor Networks. In Proceedings of the 26th IEEE International Conference on Computer Communications, Anchorage, AK, USA, 6–12 May 2007; pp. 928–936.
78. Panta, R.K.; Bagchi, S.; Midkiff, S.P. Zephyr: Efficient Incremental Reprogramming of Sensor Nodes Using Function Call Indirections and Difference Computation. In Proceedings of the USENIX Annual technical conference, San Diego, CA, USA, 14–19 June 2009; p. 32.
79. *IEEE Guide to Classification for Software Anomalies*; IEEE Std. 1044.1–1995; IEEE: New York, NY, USA.
80. Krishnan, P.V.; Sha, L.; Mechitov, K. Reliable Upgrade of Group Communication Software in Sensor Networks. In Proceedings of the 1st IEEE International Workshop on Sensor Network Protocols and Applications, Anchorage, AK, USA, 11 May 2003; pp. 82–92.
81. Grottke, M.; Nikora, A.P.; Trivedi, K.S. An Empirical Investigation of Fault Types in Space Mission System Software. In Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Chicago, IL, USA, 28 June–1 July 2010; pp. 447–456.
82. Brown, S.; Sreenan, C.J. Software Update Recovery for Wireless Sensor Networks. In Proceedings of the 1st International Conference on Sensor Networks Applications Experimentation and Logistics (SENSAPPEAL 2009), Athens, Greece, 24–25 September 2009; pp. 107–125.
83. Unterschutz, S.; Turau, V. Fail-Safe Over-the-Air Programming and Error Recovery in Wireless Networks. In Proceedings of the 10th Workshop on Intelligent Solutions in Embedded Systems, Klagenfurt, Austria, 5–6 July 2012; pp. 27–32.
84. Lee, M.H.; Choi, Y.H. Fault detection of wireless sensor networks. *Comput. Commun.* **2008**, *31*, 3469–3475.
85. Yu, M.; Mokhtar, H.; Merabti, M. Self-Managed Fault Management in Wireless Sensor Networks. In Proceedings of the 2nd International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies, Valencia, Spain, 29 September–4 October 2008; pp. 13–18.
86. Asim, M.; Mokhtar, H.; Merabti, M. A self-managing fault management mechanism for wireless sensor networks. *Int. J. Wirel. Mob. Netw.* **2010**, *2*, 184–197.
87. Bourdenas, T. Self-Healing in Wireless Sensor Networks. PhD Thesis, Imperial College London, London, UK, September 2011.

88. Chen, Y.; Gnawali, O.; Kazandjieva, M.; Levis, P.; Regehr, J. Surviving Sensor Network Software Faults. In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, Big Sky, MO, USA, 11–14 October 2009; pp. 235–246.
89. Strasser, M.; Vogt, H. Autonomous and Distributed Node Recovery in Wireless Sensor Networks. In Proceedings of the 4th ACM workshop on Security of ad hoc and sensor networks, Alexandria, VA, USA, 30 October 2006; pp. 113–122.
90. Marsh, D.; Tynan, R.; O'Kane, D.P.; O'Hare G.M. Autonomic wireless sensor networks. *Eng. Appl. Artif. Intell.* **2004**, *17*, 741–748.
91. Dong, Q. Maximizing System Lifetime in Wireless Sensor Networks. In Proceedings of the 4th International Symposium on Information Processing in Sensor Network, Los Angeles, CA, USA, 24–27 April 2005; pp. 13–19.
92. Calinescu, G. Kapoor, S. Olshevsky, A.; Zelikovsky, A. Network Lifetime and Power Assignment in ad hoc Wireless Networks. In Proceedings 11th Annual European Symposium on Algorithms, Budapest, Hungary, 16–19 September 2003; pp. 114–126.
93. Wentzlo.; D.D.; Calhoun, B.H.; Min, R.; Wang, A.; Ickes, N.; Chandrakasan, A.P. Design Considerations for Next Generation Wireless Power-Aware Microsensor Nodes. In Proceedings of the 17th International Conference on VLSI Design, Mumbai, India, 5–9 January 2004; pp. 361–367.