Updating Software in Wireless Sensor Networks: A Survey

S. Brown,
Dept. of Computer Science, National University of Ireland, Maynooth

C.J. Sreenan, Mobile & Internet Systems Laboratory,
Dept. of Computer Science, University College Cork, Ireland

Technical Report UCC-CS-2006-13-07

Abstract

Due to recent advances, Wireless Sensor Networks are moving out of the laboratory and into the field. For a number of reasons, there is likely to be a significant need to be able to remotely update the software in sensor nodes after deployment. The need for remote software updates is driven both by the scale of deployment, and the potential 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 been applied to this area. However, there still remain important questions to be answered. In this paper we present the first comprehensive survey of over-the-air WSN software updating. Based on this, we identify the need for a cohesive framework as the key open research question, and identify autonomic computing as a candidate solution technology.

I Introduction

Software updating has always been an important topic for sensor networks. In the 1978 Distributed Sensor Networks Workshop, three important functional requirements were identified: the need for dynamic modifications, the need *for heterogeneous node support*, and the need to integrate new software versions into a running system [1]. More recently, other functional requirements have been identified as: the need for the update to reach all the nodes, and to support fragmentation [2], the ability to update all the code on a node, and cope with packet loss [3], the need for distributed version control, bootloaders, and update builders and injectors [4], the ability to update the update mechanism itself, to reconfigure non-functional parameters (such as performance or dependability) without needing to update all the affected applications, and the need for utilities to provide users with standardized ways to manage online changes [5], and the need for overall management of the software update process [6].

Performance requirements have been identified as follows: unintrusiveness (i.e. remote programming), low overhead, and resource awareness, and especially to minimize flash rewriting [4], minimize impact on sensornet lifetime, and limit the use of memory resources [2], minimize processing, limit communication to save energy and only interrupt the application for a short period while updating the code [3], possibly meeting real-time constraints [7], fit within the hardware constraints of different platforms [8], cope with asymmetric links and intermittent disconnections, have a low metadata overhead during stable periods, provide rapid propagation, and be scalable to very large, high density networks [9].

Security and robustness are also key issues for remote WSN software updates. 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 [10], and intrusion detection [11]. Robustness is critical as the ability to download new software into a node introduces a whole new set of failure modes. These include identifying and handling failure (or corruption) of the updated application, other applications, the operating system, and of the network [12], to tolerate hardware and software failures, to monitor system status [5], and to meet other dependability criteria (availability and reliability) [13].

Ease of use and maintenance have been identified as key factors in the future success of wireless sensor networks [6]. Remote software updating is a core component of both, improving ease of use by providing flexibility in a deployed network, and directly supporting the maintenance function.

In this paper we present the first comprehensive review of software updating in wireless sensor networks, and identify a number of future research challenges and directions.

II Review of WSN Software Updating

WSN software update research is described here under the three categories identified in [6]: the sensor node **execution environment**, the **protocols** for disseminating the update, and **reducing the size** of transmitted updates. Common issues addressed by these are power efficiency, performance, security, and dependability. The relationships between these categories and WSNs is summarized in Figure 1.

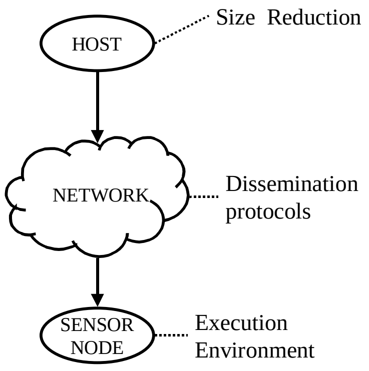

Figure 1 – Relationships between Research Categories

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, and software updating for other classes of computer systems.

A) Protocols

The protocols developed for disseminating a software update in a wireless sensor network are primarily based on data dissemination protocols, for example Directed Diffusion [14] and RMST [15]. The key differences between protocols for data collection and for software updating are: (a) the data flow for software updating is from the gateway to the nodes, and (b) the transport must be reliable. The normal pattern for dissemination protocols is a three-step: advertisement of available software, selection of a source, and reliable download to the target, which may then become a source in turn (Figure 2). This may be reversed by use of a ‘subscription’ approach, but for WSN’s this results in significantly increased overhead at the source/server end of the connection.

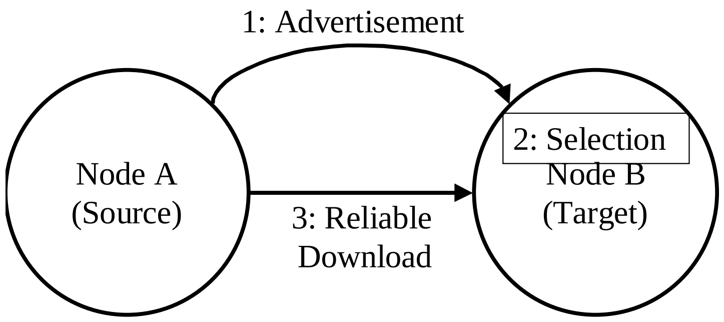

Figure 2 - Dissemination Protocols

- Deluge [16]

  Deluge is a data dissemination protocol and algorithm for propagating large amounts of data throughout a WSN using incremental upgrades for enhanced performance. It is particularly aimed at disseminating software image updates, identified by incremental version numbers, for network reprogramming. There is no support for heterogeneity: the same image is disseminated to all nodes in the network. The program image is split into fixed size pages that can be 'reasonably' buffered in RAM, and each page is split into fixed size packets so that a packet can be sent without fragmentation by the TinyOS network stack. A bit vector of pages received can be sent in a single TinyOS network packet. Nodes broadcast advertisements containing a version number and a bit vector of the associated pages received, using a variable period based on updating activity. If a node determines that it needs to upgrade part of its image to match a newer version, then, after listening to further advertisements for a time, it sends a request to the selected neighbour for the lowest page number required, and the packets required within that page. After listening for further requests, the sender selects a page, and broadcasts every requested packet in that page. When a node receives the last packet required to complete a page, it broadcasts an advertisement before requesting further pages – this enhances parallelisation ('spatial multiplexing') of the update within the network (as the node can now issue further requests in parallel with responding to requests from other nodes). The protocol keeps the state data to a fixed size, independent of the number of neighbours. There are no ACK's or NACK's – requesters either request new pages, or re-request missing packets from a previous page. There is no global co-ordination to select senders; heuristics are used to try and elect relatively remote senders in order to minimise radio network contention. Incremental updating is supported through the use of *Complete* Advertisements which indicate which pages in an image have changed since the previous version; requesters can then request just the changed pages. Future versions of Deluge are expected to address the following issues: control message suppression, running updates concurrently with applications, explicitly reducing energy consumption, and support for multiple types and versions of images.

- Deployment Support Network (DSN) [17]

  An alternative to either accessing the nodes individually, or accessing them over the sensor network, is to provide a parallel, maintenance network – a "Deployment Support Network" (DSN). 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 the nodes' energy. 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.

- Imapla/ZebraNet [18]

  Impala is the middleware layer of the ZebraNet wireless sensor network, which uses wildlife tracking as a target application in the development of a mobile wireless sensor network. Impala provides an event-based middleware layer, which is specially designed to allow applications to be updated and adapted dynamically through the use of *application adapters*. Events are processed by the *Event Filter* and then dispatched through these *Application Adapters* to the appropriate application. Zebranet nodes are intended to be situated in large numbers in places inaccessible to system administrators, and to support this, Zebrenet 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 2K 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 3-step process: firstly the nodes exchange in index of application modules, then they make unicast requests (using node ID as a tie-breaker) for updated modules, and finally they respond to requests from other nodes, by transmitting the first packet for the first requested modules. The backoff timer increases exponentially if all neighbours have the same software versions – this significantly reduces management traffic, but can delay updates when two originally separated groups of nodes (at different version numbers) become connected. If memory space is exhausted, then older incomplete version are deleted. 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.

- Infuse [19]

Infuse is a TDMA-based protocol for disseminating bulk data in location-aware sensor networks. Nodes periodically select *predecessors* and *successors*; energy is reduced by a selective listening policy, turning off the receiver for other TDMA slots; the selective use of *predecessors* and *successors* also prevents broadcast storms. A base station broadcasts a *start-download* message containing a version ID and specifying the number of *capsules* in the new data sequence. It then sends the capsules in subsequent TDMA slots. Receivers forward the *start-download* message; when they receive data modules they save them to flash and forward them. When the last capsule is received, it signals the application that the download is complete. Two different loss recovery mechanisms are discussed in their paper: *Go-Back-N*, based on implicit acknowledgements; and *Selective-Retransmission*, based on explicit, piggybacked acknowledgements. By selecting preferred predecessors, energy use can be reduced further. The authors claim that Infuse has more effective pipelining than Deluge, as all the sensors receive the data capsules at approximately the same time, avoiding the CSMA contention at the middle of the network seen in Deluge.

- MNP [10]

MNP is targeted at MICA2 motes running TinyOS and uses the XNP boot loader along with a dedicated network protocol to provide multi-hop, in-network programming. The MNP protocol operates in 4 phases:

1. Advertisement/Request, where sources advertise the new version of the code, and all interested nodes make requests. Nodes listen to both advertisements and requests, and decide whether to start forwarding code or not (this acts as a suppression scheme to avoid network overload);
2. Forward/Download, where a source broadcasts a *StartDownload* message to prepare the receivers, and then sends the program code a packet at a time (in packet-sized *segments*) to the receivers to be stored in external memory (EEPROM) – there is no ack, the receiver keeps a linked-list of missing segments in EEPROM to save RAM space;
3. Query/Update, where the source broadcasts a *Query* to all its receivers, which respond by unicast by asking for the missing packets (segments) – these are then rebroadcast by the source node, and then another *Query* is broadcast until there are no requests for missing packets. The receivers, having received the full image, now become source nodes and start advertising the new program;
4. Reboot, entered when a source received no *requests* in response to an *advertisement*, where the new program image is transferred to program memory, and the node reboots with the new code.

   A node sends a *download request* to all senders, this assists in sender selection, and also allows the hidden terminal effect to be reduced (as other potential senders can overhead this request). The sender selection algorithm attempts to allow only one active sender in a particular neighborhood. Flow control is rate based, determined by the EEPROM write speed (of the MICA2 mote).

- MOAP [2]

MOAP 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 basestation 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 EPROM, and reboots with the new code. Sliding window acknowledgements reduce power consumption (reduced EEPROM reads) at the cost of reduced out-of-order message tolerance. There is no support for rate control, or suppressing multiple senders (apart from link statistics).

- Trickle [9]

Trickle runs under TinyOS/Mate – it acts as a service to continuously propagate code updates throughout the network. Periodically (*gossiping interval* τ) using the *maintenance algorithm* every node broadcasts a code summary (‘metadata’) 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 (rx+tx), 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 desynchronised 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. Programs fit into a single TinyOS packet.

- XNP [21]

  XNP 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 (to allow for subsequent XNP updates). A host PC application *xnp* 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). The program is loaded into external (non-program) memory. Applications are halted during the program download.When a *reboot* command is issued (via the *xnp* host program), then the boot loader is called: this copies the program from external to program memory, and then jumps to the start of the new program.

B) Size Reduction

Size reduction is important for software updates: it has a direct impact on the transmission power used, and thus on sensornet lifetime. The two main approaches for optimizing power usage in this way are (a) by providing modular software, so that just the necessary modules need to be transmitted, and (b) incremental updates, so that just the changes from the previous version need to be transmitted. These approaches have been addressed in other domains, for example the *rsync* algorithm [22], dynamic software updating [23], and software updating using mobile agents [24].

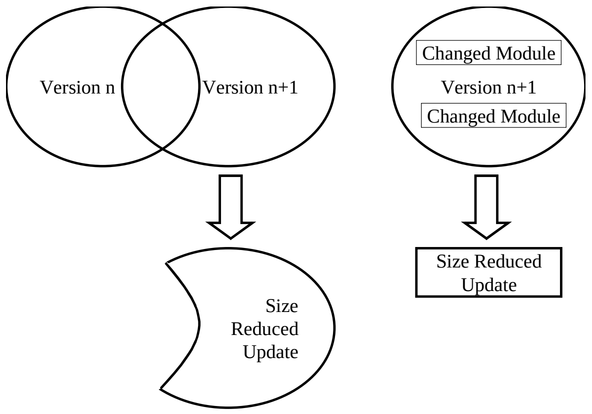

Figure 3 - Size Reduction Approaches

- Reijers [3]

  The code distribution scheme described is principally concerned with reducing the image size – 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.

- Rsync [25]

  This mechanism uses the Rsync algorithm to provide incremental software updates. 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 bootloader (modified from XNP) to select the image to read into program memory at boot time.

- Remote Incremental Linking [4]

  This is a modular updating system, designed for the mica2 mote, support static and dynamic updates. 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: this reduced overhead at the sensor nodes, but increases it at the base station, which may need to keep per-node address maps. 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 is delivered in an incremental manner, using diff scripts generated by the Xdelta algorithm. The diff target size is set as the program memory page size, allowing single pages to be updated, thus 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 the read-while-write functionality of the ATMega128's NVRAM). Both parameter values/data and code can be updated in this manner. The linking and compression are done on a base station, while a boot loader interprets the scripts and updates the program memory pages. The memory management policy attempts to minimize slow NVRAM page writes. 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.

C) The Execution Environment

The execution environment impacts on how the software update may be propagated and executed. The software update functionality will always need some low-level support to write to program memory, but the rest of the functionality may be implemented in device-specific firmware (e.g. Scatterweb), in the operating system (e.g. SOS), in a middleware layer (e.g. Agilla), or as an application (e.g. MOS). 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 are usually supported in a middleware layer to allow access to global data. Network-query approaches (e.g. TinyDB/TAG [26]) are not included unless there is a direct reprogramming aspect. We also do not directly consider hardware-specific aspects; a good overview of WSN platforms is given in [27].

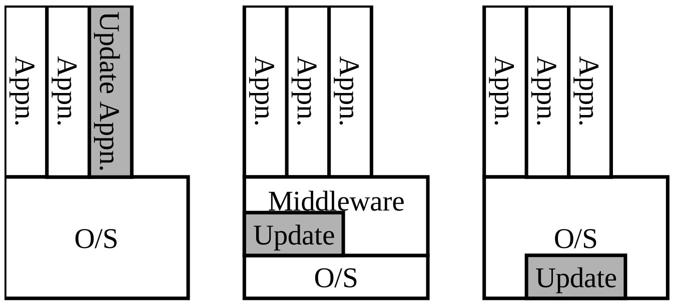

Figure 4 - Execution Environments

- Agilla [28]

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

- COMiS [29]

  COMiS is an component-based, generic middleware layer developed as part of the TinyMaCLaS project. It allows components to be loaded onto particular nodes based on the application semantics, and supports network-wide software updating. Applications are written in the DCL (Distributed Compositional Language) scripting language. Software components are registered locally, and then available for *discovery* by other components, using a broadcast algorithm, and subsequent *connections*. Both middleware and application components can be updated. Compiled component binaries are deployed into the network (through unspecified host tools) 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. There is no flooding control of the broadcast packets (except a hop count/TTL field), or rate control of the software updates (which are unicast to each node from the selected deployment nodes).

- Contiki [30]

  Contiki 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. The kernel supports both event-based programming, and pre-emptive multithreading. Typically programs are downloaded into the system over the communication stack, and stored in EEPROM before being loaded/linked into the system using a runtime relocation function 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 to the new version of the program. *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. There are plans to support an improved protocol in the future, perhaps using the Trickle algorithm. The version number of a service interface (supported by a service process) and of a service stub library (linked into the calling application process) must match; also version numbers are used to support replaceable service processes, so that incompatible versions of a service will not try to load the stored service state.

- cSimplex [31]

  The software architecture *cSimplex* is concerned not so much with the distribution of software updates, but with assessing the stability of code updates, and providing fault detection and error recovery. *Simplex* was developed for the reliable upgrade of control software for mission-critical systems; cSimplex is an extension of this work to communication software in sensor networks. Although cSimplex is targeted at group communication software, the principles would apply for any software functionality. A cSimplex system has a well tested, reliable 'core' communication module (the *safety* module); in addition an *experimental* new communication module may be loaded. These may have both system and application level components. The application level components may be loaded at runtime. Semantic checks are made at the application level on the execution of the experimental module: these checks consist of checks to data messages and management messages between nodes. If the statistics collected using the Multiple Period Method (MPM[1]) indicate faults, then the system can revert to the core module. Management messages are sent to peer nodes to synchronise the startup and termination of experimental modules.

- Mantis/MOS [32]

  The MANTIS operating system (MOS) 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 bootloader, which then installs the new code on reset. Any part of the code image (except for the bootloader) 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 an actual implementation.

- Maté/Bombilla [33][9]

  Maté is a communication-centric middleware layer for WSNs, based on a virtual machine architecture, which runs on TinyOS; Bombilla is a virtual machine that runs under Maté. 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. 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 supresses 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 Mate supported a *forw* 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*. Currently only assembly language/bytecode level programming is supported. Further studies have shown that density aware transmission rates provide an efficient tradeoff between response time and avoiding network saturation.

- REAP [34]

  REAP (Remote Execution and Action Protocol) 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 is provided for high performance transfer of objects. An index system provides a distributed database of resources. The database can be searched for polymorphic algorithms, allowing nodes to select an algorithm that matches 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. Results are cached in

[1] This provides a way to incorporate history into the error rate measurement while discounting past errors (identifying both bursts of errors, and constant low-rate errors).

routing tables, and can be used to reply to a request. There is no mechanism to flush old code versions, but they could voluntarily terminate during a code update.

- SINA [35]

  SINA is a database-style approach, supporting reprogramming through downloaded scripts. The SEE (Sensor Execution Environment) dispatches incoming messages and executes contained scripts. These are in an extended form of SQL called 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.

- SP (Spatial Programming) [36]

  Spatial Programming is a network-wide programming model. Script based programs propagate around the network in Active Messages [37]; 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.

- Szumel *et al*'s Mobile Agent Framework [38]

  This framework is built on top of Maté and 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. New agents may be easily introduced; in principle a new version of an existing agent can be introduced, but there is no direct support for updating of existing agent code, and this would have to be provided with the agents.

- Pushpin [39]

  Pushpin supports a dynamic *process fragment* model, similar to mobile agents – the fragments each support an *update* function is called repeatedly. The *Bertha* operating system can load and unload these ~2K 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 *deinstall* 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). An example program is included in [ref] that shows how a pushpin fragment can propagate itself throughout a network.

- ScatterWeb [40]

  Scatterweb is a distributed, heterogeneous platform for the ad-hoc deployment of sensor networks. Software is divided into a firmware core and modifiable tasks to provide a secure update environment. Tasks can register callbacks with the firmware to handle sensor events or received packet types. If software fails (endless loops or crashes), then the hardware can flush the software and wait for a new download. The software is fragmented into small packets to minimize packet loss. Lost packets are recovered in a two-step process: first retransmission is requested from local nodes, if this fails retransmissions can be requested 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.

- SensorWare [41]

  SensorWare is a heavyweight active sensor framework (i.e. supporting code mobility) designed for relatively large memory nodes (e.g. 1MB 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 (static applications can coexist, and use O/S services directly). SensorWare is an event driven framework; the network communication model is based on message passing. Scripts can be injected by 'external users'; scripts can contain a *replicate* command that causes the script to be replicated on another node (an *intention to replicate* message is sent first to determine whether the script needs to be transferred first). Scripts can be replicated to a specific list of nodes, or broadcast to all neighbours (via an empty list). Whereas the SensorWare code itself has support for different software variants, there is no explicit support for this in the script replication functionality. There is no support for different versions of a script; though this could probably be achieved at the application level.

- SOS [42]

  SOS is a microkernel-style operating system, supporting 'C' programs, with support for dynamic module loading and unloading. Device drivers, and kernel services are provided in SOS: other services (such as routing, applications etc.) are provided as dynamic modules. Modules provide handlers for initialization, finalization, timers, sensor readings, and message reception. Synchronous communication between modules is provided through *registered* functions: functions can be called by another module via pointers in a *function control block (FCB)* which stores information on the function prototype and version as well as the address. 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, the *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. If an unavailable function is called, then kernel provides a stub to return failure (and do any required deallocation). If a function prototype changes, then new calls to *get_handle* must provide the new prototype in order to access the function. SOS does not support update of the core kernel image.

III Open Research Questions

There are a number of open research problems common to all the classes:

1. Planning – before initiating an update, it would be invaluable to be able run a model (such as a simulation) to determine the energy cost of different update options on the current network configuration, and thus allow the user to make informed tradeoffs against energy. For example: which is the best injection strategy for the current configuration? What size reduction technique will result in the quickest update? etc.
2. Size reduction – there is not yet a definitive answer to the best way to reduce the size of updates, with modular, differences-based, and straightforward compression all showing promise in different circumstances. Also the question of incremental vs monolithic updates is still open. In order to provide future-proofing and flexibility, it is possible that software update might include a small code module to correctly apply the update.
3. Injection strategies – there are a number of different injection strategies in use: simulating a base station, sending the update to a base station for dissemination, sending the update to a number of seed nodes for further dissemination, and sending the update individually to each node.
4. Dissemination protocols – there have been many recent advances in these protocols, in fact this is probably the most advanced area of the field. As in other WSN protocols, it is likely that an energy-aware approach will have to be taken in order to respond to current energy patterns in a sensornet (ref. energy-aware MAC layers, and energy-aware Routing).
5. Activation control – there is little work on controlling the activation of software updates following download. In most cases the software update is either automatically activated (perhaps after a timeout), or manual activation is required. Impala provides for rule-based switching of protocols: this mechanism might also be suitable for rule-based activation. In order to support the various possible patterns in which software updates may be received, and to support any requirements for backwards and forwards version compatibility, tighter control over the order of node activation will be required.
6. Dependability – there are a number of aspects of this which are not directly related to software updating, but the key ones which are related are: checking the downloaded software before activation (integrity, version mismatches, platform mismatches) and dynamically checking the operation of the

downloaded software after is has been activated. It is likely that further advances will be necessary in this area, probably using techniques from autonomic computing, to increase the robustness of software updates.

7. Monitoring – there is a need for tools to monitor the 'version' state of a WSN and report status and problems to an operator/user. These will be able to use existing techniques for fusing data to reduce the overhead, and for tracking update-related faults.
8. Security – this is going to be a key concern for software updates in the field, with the normal issues of: key-distribution, authentication, secrecy, integrity, and authorisation needing to be addressed. Results from existing WSN security research will be needed, along with other work specific to the software update problem.
9. Support for very small nodes – it is likely that large WSNs will be deployed with very small nodes, with very limited code memory, and perhaps almost no RAM or EEPROM for storing new code. In this case, techniques will be need to be developed to allow for incremental building of new code straight into code memory (usually FLASH-RAM). This technique would also be useful for reducing the power consumption of a software update – EEPROM writes are an expensive option for temporary code storage.
10. Version control – this is already support by some systems, with SOS having quite a sophisticated mechanism. Further developments in this area will be needed to ensure that there is full control over what software versions are downloaded/activated, and to prevent version mismatch (both inter-node and intra-node) problems.
11. Heterogeneity support – it is likely that deployed networks will have a mix of platforms, for cost reasons, and perhaps for robustness also. A strategy of a small number of high powered nodes for a 'Spine'/data backbone (shown to be optimal for data delivery) and a large number of lower powered nodes (for data collection) seems a very likely scenario. In this case, the backbone nodes will have to handle not only different versions of their own codebase, but different codebases also.
12. Performance – there is very little direct control over the update time in current systems, and no capability of trading off time against energy (perhaps in an emergency situation). It seems likely that research into supporting tradeoffs, and also provision of closed-loop control of total network-wide update time ill be needed.
13. Energy reduction – the protocols used need to be energy-aware, so that the current energy-state of both individual nodes and the entire network can be taken into account during an update.
14. Recovering from faulty updates – some initial work in both checking the new software before execution and during execution has been done.

IV Conclusions

This review identifies the large number of different dissemination protocols have been developed for WSN software updates. Many of these have features designed for particular platforms or execution environments, or for particular types of network configuration. 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 format**: 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**.

Overall this progress through discrete solutions reveals the fundamental research challenge in WSN software updating: to bring all this together into a *cohesive framework*. Most of the discrete solutions are optimal in some situations, but not in others, and thus a framework needs the flexibility to react to the current state of the WSN in order to operate dependably, efficiently, effectively, and securely. The inaccessibility and scale planned for sensor networks argue for an *autonomic* approach [43] to solving the software update problem. Users will, at least in the short-to-medium term, still need to implement the code for software updates; but once injected into the sensornet, the software updates should be handled with minimal user intervention (see Figure 5). In particular, the WSN as a whole should recover from failures associated with the software updates. Identifying failures is not an easy task, and lessons can be learned from the Simplex system [12] in monitoring the software for entry into inadmissible states, and differentiating between experimental (high performance) and baseline (high assurance) software to provide for recovery.

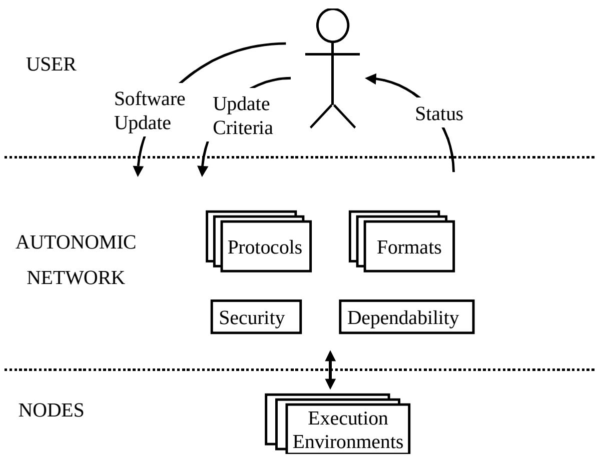

Figure 5 - The Autonomic WSN

Once available, software update functionality is likely to be heavily used in order to optimize the use of deployed networks. To support more frequent reprogramming energy consumption models, such as those presented in [44] and [45], 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 [46], 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.

References

[1] Habermann, A.W.: “Dynamically Modifiable Distributed Systems”. In: *Proc. of a Workshop on Distributed Sensor Nets*, Pittsburgh, Pennsylvania, Dec. 7-8 1978. Carnegie-Mellon University (1978) pp. 111-114

[2] Stathopoulos, T., Heidemann, J. Estrin, D.: “A Remote Code Update Mechanism for Wireless Sensor Networks”. *CENS Tech. Report #30*. Centre for Embedded Networked Sensing, UCLA (2003)

[3] Reijers, N., Langendoen, K.: “Efficient Code Distribution in Wireless Sensor Networks”. In: *Proc. of the Second ACM Intl. Workshop on Wireless Sensor Networks and Applications* (WSNA’03), San Diego, CA, Sept. 2003. ACM (2003) 60-67

[4] Koshy, J., Pandy, R.: “Remote Incremental Linking for Energy-Efficient Reprogramming of Sensor Networks”. In: *Proc. of the 2nd European Workshop on Wireless Sensor Networks* (EWSN’05). IEEE (2005) 354-365

[5] Sha, L.: “Evolving Dependable Real-Time Systems”. In*: Proc. of Aerospace Applications Conf.*, vol. 1. IEEE (1996) 335-346

[6] Han, C.-C., Kumar, R., Shea R., Srivastavam, M.: “Sensor Network Software Update Management: a Survey*”. Intl. Journal of Network Management*, no. 15. John Wiley & Sons (2005) 283-294

[7] Stankovic, J.A., Abdelzaher, T.E., Chenyang Lu, Sha, L., Hou, J.C.: “Real-time communication and coordination in embedded sensor networks”. In: *Proc. of the IEEE*, Volume: 91, Issue: 7, July. IEEE (2003) 1002-1022

[8] Liu, T., Sadler, C.S., Zhang, P. Martonosi, M.: “Implementing Software on Resource-Constrained Mobile Sensors: Experiences with Impala and ZebraNet”. In: *Proc. of the 2nd Intl. Conference on Mobile Systems, Applications and Services* (MobiSys’04), Boston, MA, June. ACM (2004) 256-269

[9] Levis, P.,Patel, N., Culler, D., Shenker, S.: Trickle: “A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks”. In: *Proc. of the First USENIX/ACM Symposium on Network Systems Design and Implementation* (NSDI 2004), San Francisco, CA, Mar. USENIX(2004) 15-28

[10] Devanbu, P., Gertz, M., Stubblebine, S.: “Security for Automated, Distributed Configuration Management”. ICSE99 *Workshop on Software Engineering over the Internet*. USC(1999) 1-10

[11] Racherla, G., Saha, D.: “Security and Privacy Issues in Wireless and Mobile Computing”. In: *Proc. IEEE Intl. Conf. on Personal Wireless Communications* (ICPWC'2000). IEEE (2000) 509-513

[12] Sha, L.: “Upgrading Real-Time Control Software in the Field”. In: *Proc. of the IEEE*, Vol. 91, No. 7, July 2003. IEEE (2003) 1131-1140

[13] Avizienis, A., Laprie, J.-C., Randell, R., Landwehr, C.: "Basic Concepts and Taxonomy of Dependable and Secure Computing". In: *IEEE Transactions on Dependable and Secure Computing*, Vol. 1, No. 1, Jan.-Mar. 2004. IEEE (2004) 11-33

[14] Intangonwiwat, C., Govindan, R., Estrin, D.: "Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks". In: *Proc. 6th Annual Conference on Mobile Computing and Networking* (Mobicom'00). ACM (2000) 56-67

[15] Stann, F., Heidemann, J.: "RMST: Reliable Data Transport in Sensor Networks". In: *Proc. of the 1st IEEE Intl. Workshop on Sensor Network Applications and Protocols*. IEEE (2003) 102-112

[16] Hui, J., Culler, D.: "The Dynamic Behavior of a Data Dissemination Protocol for Network Reprogramming at Scale". In: *Proc. of the 2nd international conference on Embedded Networked Sensor Systems*, Baltimore, Maryland, USA. ACM (2004) 81-94

[17] Beutel, J., Dyer, M., Meier, L., Ringwald, M., Thiele, L.: *Next-Generation Deployment Support for Sensor Networks*. TIK-Report No: 207. Computer Engineering and Networks Lab, Swiss Federal Institute of Technology (ETH), Zurich (2004)

[18] Liu T., Martonosi, M.: "Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems". In: *Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Progamming* (PPoPP'03), June 11–13, San Diego, California, USA. ACM(2003) 107-118

[19] Kulkarni, S.S., Arumugam, M.: *Infuse: A TDMA Based Data Dissemination Protocol for Sensor Networks*. Technical Report MSU-CSE-04-46. Dept. of Computer Science and Engineering, Michigan State University, MI (2004)

[20] Kulkarni, S.S., Wang, L.:"MNP: Multihop Network Reprogramming Service for Sensor Networks". In: *Proc. of the 25th IEEE Intl. Conf. on Distributed Computing Systems* (ICDCS'05). IEEE(2005)

[21] Jeong, J.,Kim, S., Broad, A.:*Network Reprogramming*. TinyOS documentation. Available at: http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/NetworkReprogramming.pdf. Berkeley (2003)

[22] Trigdell, A., Mackerras, P.: *The rsync algorithm*. Technical report TR-CS-96-05, Australian National University, Canberra (1998)

[23] Hicks, M.: *Software Updating*. Ph.D. Dissertation. Univ. of Pennsylvania (2001)

[24] Bettini, L., De Nicola, R., Loreti, M.: "Software Update via Mobile Agent Based Programming". In: *Proc. 2002 ACM Symposium on Applied Computing* (SAC 2002). ACM(2002) 32--36

[25] Jeong, J., , D.: "Incremental Network Programming for Network Sensors", In: *Proc. of the 1st Annual IEEE Communications Society Conf. on Sensor and Ad Hoc Networks and Communications* (SECON 2004). IEEE(2004) 25-33

[26] Madden, S., Franklin, M.J., Hellerstein, J.M.: TAG: "A Tiny AGgregation Service for Ad-Hoc Sensor Networks". In 5th *Annual Symposium on Operating Systems Design and Implementation* (OSDI'02), USENIX, December 2002

[27] Hill, J., Horton, M., Kling, R., Krishnamurthy, L.: "The Platforms Enabling Wireless Sensor Networks". In: *Communications of the ACM*, Vol. 47, No. 6, June 2004. ACM (2004) 41-46

[28] Fok, CL., Roman, G.-C., Lu, C.: "Mobile Agent Middleware for Sensor Networks: An Application Case Study". In: Proc. *4th Intl. Symposium on Information Processing in Sensor Network*s (IPSN'05). IEEE (2005) 382--387

[29] Janakiram, D., Venkateswarlu, R., Nitin, S.: "COMiS : Component Oriented Middleware for Sensor Networks." In: *Proc. 14th Intl. Workshop on Local and Metropolitan Area networks* (LANMAN05). IEEE (2005)

[30] Dunkels, A., Grovall, B., Voigt, T.: "Contiki – a Lightweight and Flexible Operating System for Tiny Networked Sensors". In: *Proc. First IEEE Workshop on Embedded Networked Sensors*, Nov. 2004. IEEE (2004)

[31] Krishnan, P.V., Sha, L., Mechitov, K.: "Reliable Upgrade of Group Communication Software in Sensor Networks". In: Proc. of the *First IEEE Intl. Workshop on Sensor Network Protocols and Applications*. IEEE(2003) 82-92

[32] H. Abrach, H., Bhatti, S., , J., Dai, H., Rose, J., Sheth, A., Shucker, B., Deng, J., Han, R.: "MANTIS: System Support for MultimodAl NeTworks of In-situ Sensors". In: *Proc. of the Second ACM Intl. Workshop on Wireless Sensor Networks and Applications* (WSNA'03), San Diego, CA, Sept. 2003. ACM (2003) 50-59

[33] Levis, P., Culler, D.: "Mate: a Virtual Machine for Tiny Networked Sensors". In: *Proc. of the ACM Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), Oct. ACM(2002) 85-95

[34] Brooks, R.R., Keiser, T.E.: "Mobile Code Daemons for Networks of Embedded Systems". In*: IEEE Internet Computing*, Jul/Aug 2004. IEEE (2004) 72-79

[35] Shen, C-C., Srisathapornphat, C., Jaikaeo, C.: “Sensor Information Networking Architecture and Applications”. In: *Proc. of the International Workshop on Pervasive Computing*, Toronto, Canada, August. IEEE(2004) 52-59

[36] Iftode, L., Borcea, C., Kochut, A., Intanagonwiwat, C., Kremer, U.: “Programming Computers Embedded in the Real World”. In: Proc. *of the Ninth IEEE Workshop on Future Trends of Distributed Computing Systems* (FTDCS’03). IEEE (2003) 78-85

[37] von Eicken, T., Culler, D., Oldstein, S., Schauser, K.: “Active Messages: A Mechanism for Integrated Communication and Computation”. In: *Proc. of the 19th International Symposium on Computer Architecture*. (1992). 256-266

[38] Szumel, L., LeBrun, J., Owens, J.D.:”Towards a Mobile Agent Framework for Sensor Networks”. In: *Proc. of the 2nd IEEE Workshop on Embedded Networked Sensors* (EmNets-II). IEEE (2005) 79--88

[39] Lifton, J., Seetharam, D., Broxton, M., Paradiso, J.: “Pushpin computing system overview: A Platform for Distributed, Embedded, Ubiquitous Sensor Networks”. In: Mattern, F. , Naghshineh, M. (eds.): In. *Proc. of the International Conference on Pervasive Computing*, August 2002, Lecture Notes in Computer Science 2414. Springer-Verlag (2002) 139-151

[40] Schiller, J., Liers, A., Ritter, H., Winter, R., Voigt, T.: “ScatterWeb – Low Power Sensor Nodes and Energy Aware Routing”. In: *Proc. of the 38th Hawaii International Conference on System Sciences*. IEEE (2005) 1-9

[41] Boulis, A., Srivastava, M.: “A Framework for Efficient and Programmable Sensor Networks”. In: *Proc. of the 1st Intl. Conf. on Mobile Systems, Applications, and Services* (MobiSys'03). ACM (2003) 187--200

[42] Han, C.-C., Kumar, R., Shea, R., Kohler, E., Srivastava, M.: “A Dynamic Operating System for Sensor Nodes”. In: *Proc. of the 3rd Intl. Conf. on Mobile Systems, Applications, and Services* (MobiSys'05). ACM (2005) 163-176

[43] Kephart, J.O., Chess, D.M.: “The Vision of Autonomic Computing”. In: *IEEE Computer*, vol. 36, iss. 1, Jan. 2003. IEEE (2003) 41-50

[44] Dong, Q.: “Maximizing System Lifetime in Wireless Sensor Networks”. In: *Proc. of the 4th Intl. Symposium on Information Processing in Sensor Networks* (IPSN'05). IEEE (2005) 13-19

[45] Calinescu, G. Kapoor, S. Olshevsky, A., Zelikovsky, A. (original authors), Hyvarinen, A.: “Summary of Network Lifetime and Power Assignment in ad hoc Wireless Networks”. In: *Proc. 11th Annual European Symposium on Algorithms* (ESA'03), LNCS 2832. Springer-Verlag (2003) 114-126

[46] Wentzloff, 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* (VLSID’04). IEEE (2004) 361-367