A New Model for Updating Software in Wireless Sensor Networks

Stephen Brown, National University of Ireland Maynooth
Cormac J. Sreenan, University College Cork

Abstract

Wireless Sensor Networks (WSNs) are expected to be deployed for long periods of time, and the nodes are likely to need software updates during their lifetime, both for bug fixes and in order to support new requirements. But in many cases the nodes will be inaccessible or too numerous to be physically accessed. This drives the need for "over-the-air" support for software updates, and a number of different systems and supporting protocols for performing these updates have been developed. These systems have addressed the problem in a number of different ways, and meet different requirements. This article consolidates this work by determining an integrated set of necessary WSN software update criteria, and presenting a novel model for WSN software update systems.

Wireless Sensor Networks (WSNs) have gained much attention lately due to the recent developments in systems integration and mobile wireless. A critical issue in the effective deployment of these networks is the ability to update software after deployment.

There are a number of reasons why the software may require updating in a WSN. The Software Engineering Institute (SEI) at Carnegie-Mellon University identifies four categories of software updates for dependable systems, which help to provide an insight into these reasons: *maintenance* releases, *minor* releases, *major* releases (technology refresh), and *technology insertion*.

Embedded, wireless sensor systems, programmed by specialists in their own environmental domains rather than software engineers, are likely to experience higher levels of maintenance than normal. Minor releases will be used to improve data collection and performance. As the needs of WSNs are likely to develop dynamically over time, major releases can be expected in response. Finally, due to the active research on WSNs and related technologies (especially wireless), and the associated development of new algorithms and protocols, technology insertion will be an important driver of software updates. In addition, software updating over the network will be an absolute necessity in the development and testing cycle for large-scale WSNs [1].

In this article we provide a consolidated set of WSN software update requirements in the form of criteria to be met, present a new model that meets these criteria, and validate our model against some representative systems. This article does not attempt to provide a detailed comparison of existing algorithms from the viewpoints of performance, energy efficiency,and so forth, but rather provides an encompassing framework within which existing and new research can be placed so that the relationships between them can be better understood.

The rest of this article is organized as follows. First, the problems associated with the software update problem in WSNs are described, and the systems analyzed in developing a model are detailed. Next, the set of derived criteria for software updating is presented. The software update model is then described, and validated against four different implementations. In the conclusion, we propose this model as a framework for future research.

Background

The criteria and model presented in this article are based on a review of the requirements associated with updating software in WSNs [2–6], developments in the associated areas of handling execution failures in WSN updates [7] and managing software updates [8], and on a detailed analysis of the following WSN software update systems, components, and protocols: Agilla, Maté (Bombilla), COMiS, Contiki, cSimplex, Deluge, Deployment-Support Networks (DSN), Zebranet/Impala, Infuse, MNP, MOAP, MANTIS/MOS, Pushpin/Bertha, REAP/RSN, Reijers' *diff*-based optimizations, Jeong and Culler's Rsync-based Incremental Network Programming, ScatterWeb, SensorWare, SOS, Spatial Programming, SINA, Szumel *et al*.'s Mobile Agent Framework, Trickle, and XNP (references for these can be found in [9]).

Traditional software update tools are ineffective for WSNs, as they do not take into account the severe resource constraints at individual nodes, or the impact of wireless communication. As discussed in [8], systems such as HP OpenView, IBM Tivoli Configuration Manager, and Microsoft Systems Management Server provide features such as planning, impact analysis, selective targeting, distributing differential changes, and patches for size reduction, multicast propagation, and some support for pervasive and intermittently connected mobile devices. However, they do this in an environment in which the target systems have significant resources compared to a typical Wireless Sensor Node (e.g., storage space, power, O/S features, etc.), a separate and reliable networking infrastructure exists, and manual intervention is a realistic option.

Wireless sensor nodes are characterized by very limited resources, and by large-scale deployment. Accessing these

0890-8044/06/$20.00 © 2006 IEEE

nodes in the field to perform software updates can be difficult or impossible. The nodes can be difficult to locate or inaccessible, or the scale of the deployment can preclude individual access. Remote update poses its own problems, however, due primarily to resource constraints. Three key issues are:

- Avoiding interference with data collection while sharing the same communications infrastructure
- Minimizing the cost of upgrades in terms of the impact on sensor network lifetime
- Avoiding the loss of part or all of a sensor network due to an upgrade fault

There are two classes of software updates for WSNs: static and dynamic. A *static* update involves rebooting the node. The update itself may be in the form of a single monolithic image, or incremental in the form of patches to an image, or incremental in the form of full updates to software modules. State may be transferred to the new software through boot-safe memory. The operation of an individual node will be interrupted during the update, but acceptable operation of the network as a whole may continue. A *dynamic* update is made without stopping execution of a node's operational software — though the operation of the software being updated will be necessarily interrupted. Typically, the old code is marked as unavailable and unlinked from the system, and the memory is freed for reuse. Then the new code is loaded into program memory, linked in, initialized, and marked as available for use. Typically, support is provided to transfer state to the new software, and to hide the temporary interruption from other software components — a difficult area to handle effectively. Mobile agents are a distinct form of dynamic updates, where software is moved from node to node to track events and data. Their inherent support for mobility usually provides for state transfer, but updating existing agents is more difficult than injecting new agents.

Remote software updating may be supported in the operating system (e.g., ScatterWeb, SOS), in a middleware layer (e.g., Impala, Sensorware, Maté, COMiS), or in application space (e.g., XNP, MNP, MOAP, Deluge).

Software Update Criteria

Based on a first-principles look at the requirements for updating WSN software, an examination of partial criteria provided in a number of articles, and a review of existing WSN software update mechanisms, we have developed a set of criteria for deriving and evaluating WSN software update solutions. The justification for why these criteria are necessary for a complete and optimal solution is presented for each.

Functionality. After developing a software update, three main phases of activity are required before the update can start executing on the nodes. These are *generating* the update, *propagating* the update to the target nodes, and *activating* the update on these nodes. Planning is an important part of the generation, and feedback from all three phases may be used to improve the quality of future planning in order to provide responsiveness to the actual network configuration and conditions. In addition, continuity of service must be ensured, especially with regards to backwards and forwards version compatibility of updates. There must be support for both monolithic and partial updates.

Security. Authentication is required to ensure that no unauthorized updates can be effected. It may also be needed to provide fine-grained determination over what software can be updated by which end users. Privacy is required to ensure that neither the updated software nor node-specific information (e.g., network address, id, passwords, location) can be intercepted. Data integrity of the update must also be ensured (e.g., checksum or digital signature). The system must also be robust against denial-of-service attacks, which may compromise the operation or lifetime of the network.

Performance. This is a key factor for other WSN applications, and energy/lifetime costs must be balanced against time: one critical difference with respect to data gathering is that data transmission must be reliable. There may also be subtle factors involved in minimizing the impact on overall network lifetime, avoiding underpowered nodes, and using forward error correction effectively. The impact on normal operation must be limited, restricting the use of resources such as memory, CPU, and network bandwidth. Using broadcasting to reduce transmission energy introduces potential problems such as broadcast storms and collisions. The time taken to reprogram an entire network needs to be controlled. There should be no inherent limitations on the size of the software update — implying the need for image fragmentation.

Reliability. Given the inaccessible nature of many sensor networks and the likelihood of very large-scale deployment, the reliability of the software update mechanism is critical. It must be able to complete the update even in the presence of noncritical failures. In addition, reliability of the updated software is an undoubted necessity, and the update mechanism needs to support both local and network-wide fallback to the previous version (or a known good image) if a software update fails. Ideally an update should be autonomous; once injected, it should handle various error conditions and either run to a complete installation or reject the upgrade.

Usability. An upgrade system must provide host tools to support the generation, propagation, and activation of a software update. It must support the update of both the O/S and application software in the target nodes. In addition, maintainability is important for practical use: it must be possible to update the software updating mechanism itself. An additional need is the ability to monitor the status of an update, and possibly take corrective action. An accurate mechanism for the maintenance of software names and version numbers is key here also.

Portability. Whereas a particular implementation may have limited portability, the general mechanism should be independent of the features of any particular O/S, middleware layer, or MAC layer. It should also be feasible on a wide range of WSN platforms: from very resource-limited nodes, to larger ones with more resources (especially program memory, external memory, and CPU).

A WSN Software Update Model

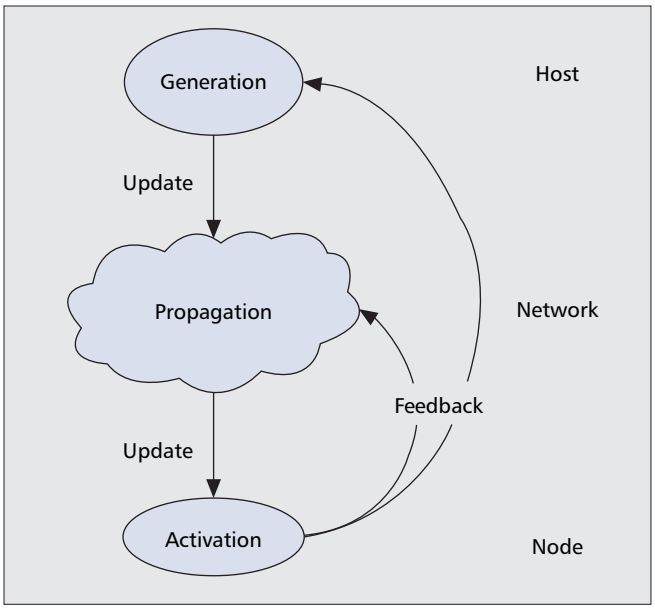
The model presented here has been developed to support the *functionality* criteria identified in the previous section (including some aspects of security). The rest of the requirements identified apply across all the functionality activities. Optimal algorithms are the ones that best meet the needs of performance, reliability, and security; optimal systems are the ones that also best meet the needs of usability and portability.

The high-level data-flow diagram (DFD) shown in Fig. 1 highlights the interactions between the three key elements of software update functionality: *generation*, *propagation*, and *activation*.

Generation

On a host system, this functionality is concerned with planning for the update, generating the update data, and inserting it into the network (Fig. 2).

Plan for a software update — This may involve running a simulation (or using a model) to estimate the required power to do an update, allowing the efficiency of different alterna-



■ Figure 1. *WSN software update model.*

tives to be explored, or verifying that the update is feasible. Current network and node configuration data may need to be fed in. Planning is likely to be a key feature in providing power-efficient updates.

Build the software update — This may involve converting the output from the compiler tool-chain into a suitable format, reducing the image size by reading the old image and creating a differences script or compressing the new image, adding forward error correction, and reading a key to sign or encrypt the image to provide authentication and privacy.

Insert the software update into the network — This involves running a tool on a management host that communicates the image to a base station in the network. It may communicate directly to the base station to do this, emulate a base station, seed the code propagation directly to selected nodes in the network, or communicate directly with the targets.

Note that feedback from the *monitor* activity in the propagation and activation phases may act as feedback for generation or to future planning. For example, the decision of whether to generate a full update, or a diff-style update, may depend on knowledge of which nodes are running which software version. Or network-specific parameters, such as network depth, may be determined by this feedback, and used in planning a future update.

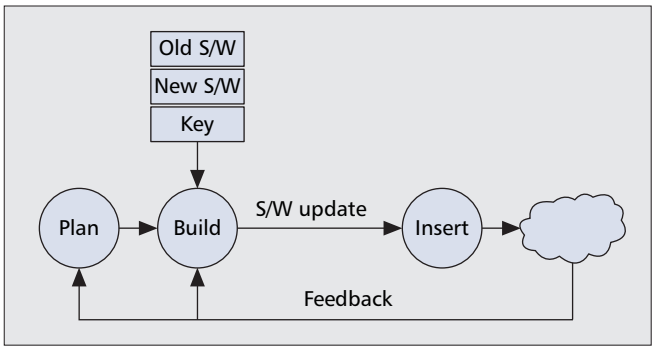
Propagation

This functionality is concerned with transferring the update from the insertion point through the wireless sensor network to the desired destination points or targets. This network-wide functionality is supported by client–server interactions.

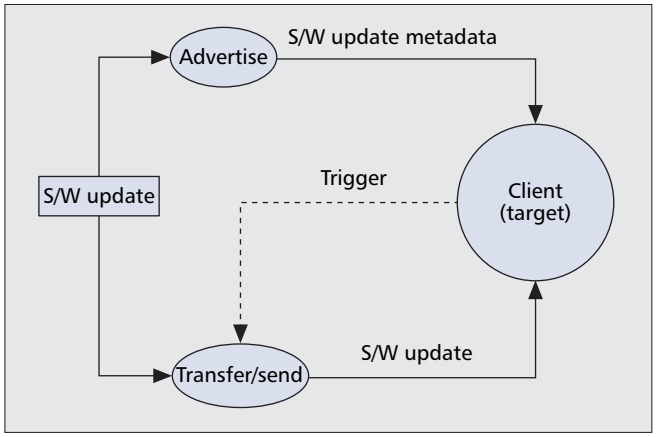
Server-Side Propagation — Following download of a software update, each client may subsequently act as a server, or source, to other nodes (Fig. 3).

The activities identified in the diagram shown in Fig. 3 are as follows:

Advertize: Send image metadata — normally via broadcast to neighboring nodes to minimize transmission energy. The transmission of duplicates can be avoided by overhearing other broadcasts. This functionality normally continues indefinitely, using variable timers for increased efficiency, in order to provide recovery from temporary network partitioning. The *advertize* and *listen* functions can use authentication to enhance security.



■ Figure 2. *Generation.*



■ Figure 3. *Propagation/server side.*

Transfer/send: Transfer the software update to the client — this needs to handle packet loss and delays at the client end (especially writing to slow EEPROM memory). The main design challenge here is preventing network congestion (bandwidth hogging, collisions/hidden terminal problem, and broadcast storms). Denial-of-service attacks must be identified and handled here; requests for retransmissions may be a particular source of weakness. When there are no further transfer requests, a server may trigger activation, and/or may continue as a server.

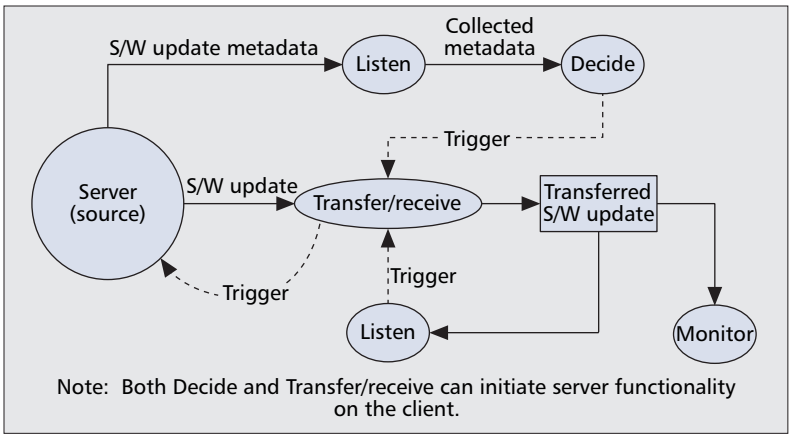
Client-Side Propagation — The client side of the interaction (Fig. 4) is the more complex, as performance concerns mean that connection state is invariably handled more effectively at the client.

The activities identified in the diagram (Fig. 4) are as follows:

Listen: Collect metadata from update advertizements — the *decision* can be more efficient if a number of advertisements are collected first. Also, a node's own advertising strategy can be optimized by overhearing other advertisements.

Decide: Compare metadata from the collected update advertisements with that in the nodes current software. Use rules (e.g., version comparison) to trigger the fetch function. Decide can also trigger the server mode, and effect changes in the advertising. The design challenges here in selecting a source are avoiding both broadcast storms and excessive latency, while minimizing energy use and avoiding excessive delays. Denial-of-service attacks may be identified in the listen and decide functionality.

Transfer/recv: Transfer the software update from the server into storage — this needs to rate-limit the server in order to handle delays in writing so as to slow memory. The main design challenge here is preventing network congestion (bandwidth hogging, collisions/hidden terminal problem, and broadcast storms). When transfer is complete (see *verify* below), the



■ Figure 4. *Propagation/client side.*

execution phase of the update can occur. The *fetch* functionality will also trigger the nodes' own *advertize* in order to further propagate the update upon completion of a complete image. The delay time can be reduced for little or no energy cost by triggering advertize on completion of each download segment, providing for pipelining (spatial multiplexing) of the image segments over the network.

Verify: Verify that the download is complete, and trigger additional transfers if required. This may occur on a packet-by-packet basis, using positive or negative acknowledgments, or be a request for missing segments once the initial transfer has completed. Once complete, the execution phase can be entered. Efficient handling of lost packets can have a significant impact on both the total download time and the network bandwidth used.

Monitor: This provides feedback on the progress of the update. This may be used to populate a end-user display, used as feedback in future update planning, or used to make decisions (e.g., canceling an upgrade, etc.).

Activation — This functionality (Fig. 5) is associated with initiating the execution of the software update on the destination nodes; it may be triggered locally, through consensus, or from the host node, based on various rules. The update is checked, loaded into program memory, and then executed (or made available for execution).

It is assumed, with no loss of generality, that a node has both storage memory (to store the update) and program memory (to store the new S/W) available. In cases where there is no actual external memory, program memory may be used, or an update may be loaded in small sections into RAM (acting as storage) and each immediately loaded into program memory.

Decode: As necessary, decompress the image and decrypt using the key; these may be done on-the-fly to save memory space. If the update is a "differences script," then decode and execute the instructions against the old software to generate the new software. The new software may be an entire image, an application, a software component (or mobile agent), or just a patch applied to the old image.

Check: Check the new software for security and correctness. For security, authentication and data integrity can be checked. For correctness, the code can be checked against a template (for example, make sure that each function has a return statement). The security checks may be made before decode. The operating system may be a protected component, either with password protection, or allowing no updates "over-the-air." The following *load* and *execute* functions may either be triggered locally, through a set of rules potentially involving neighborhood and network-wide status, or remotely from the host system.

Load: Copy the code into program memory; this may involve format conversion. In some systems, where program memory cannot be easily written, this may be part of a bootloader; in other systems, this may be performed inline. The space for the old code may need to be de-allocated. For a dynamic update, the new code will need to be linked into the existing image (and the old code unlinked). This load may be part of the decode activity for a differences script if the new image is loaded on-the-fly into program memory.

Execute: For static updates, the load and execute functionality is often handled by a dedicated boot-loader (due to the programming models of the associated CPU). For dynamic updates, the activate functionality consists of initializing the new component, and marking it as available for execution.

Monitor: Following the activation of the new software, its progress may be monitored for correct operation. Status may also be returned to the host to provide feedback. Checks may be embedded in the code (internal consistency checks), or may be performed externally (ensuring that the code behavior matches a model). Following a failure of the software, the new software may be restarted, the system may revert to the previous software version, or a "known good" image may be invoked. This will involve collaboration with the *compare* functionality to make sure that future update advertisements are correctly interpreted. If the "known good" image is not stored locally, there may be direct communication with the host to acquire it.

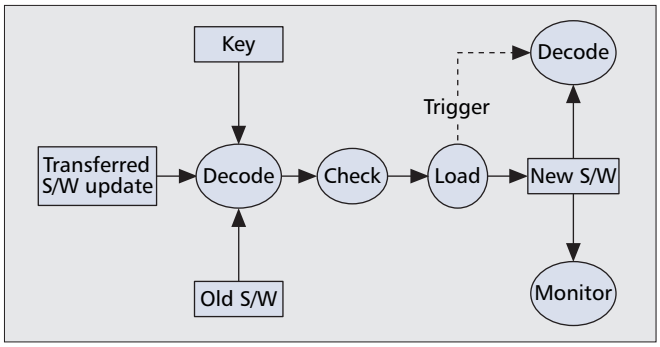
Validating the Model

Space does not allow for a full validation of the model against the many systems, protocols, and mechanisms reviewed in preparing the criteria and model. Instead, the model is validated here against three different systems, representing three different classes of software update: static/monolithic updating (MOAP), dynamic/mobile agent-based updating (Maté), and dynamic/component-based updating (Impala). In addition, the model is validated against a distribution protocol (Deluge).

MOAP

MOAP [6] is a multihop, over-the-air code distribution mechanism specifically targeted at MICA2 motes running TinyOS.

Generation — Only static/monolithic image updates are supported. To *build* the update, a packetizer converts the software image to 16-byte segments. *Insert* is supported by delivering the software update to a base station, along with the software version number.



■ Figure 5. *Activation.*

Propagation — To advertize the update, the base station (and, subsequently, secondary sources) broadcast *publish* messages advertising the version number of the new code. This *ripple* approach explicitly increases latency in order to reduce transmission energy. Network partitioning is handled by a *late joiner* mechanism — nodes continuously send publish messages to advertise their version. Receivers *listen* for a period to receive all subscriptions, in order to optimize the selection of sender. Decide is implemented by receiving nodes checking the advertised version number. Transfer/client requests the update with *subscribe* messages. A link-statistics mechanism is used to try to avoid unreliable links. If transfer/server receives no subscribe messages within a timeout period, it "commits" by triggering *activation*. The store-and-forward approach provides a "ripple" pattern of updates. Once a node has received an entire image, it becomes a sender in turn. The verify performance is enhanced by using a bitmap of received segments to reduce EEPROM reads. Missing segments are identified by the receiver using a sliding window, and are re-requested directly from the receiver using a unicast message to prevent duplication. NACK-based re-requests reduce both energy usage and memory space requirements for the sender. Re-requests are prioritized to reduce delay by quick recovery. Sliding window acknowledgments reduce power consumption (reduced EEPROM reads) at the cost of reduced out-of-order message tolerance. If unicast retransmission requests are not responded to, broadcasts are used.

Activation — A *decode* function transforms the downloaded text records into binary data. The *check* is provided by the underlying link mechanisms on a per-packet/segment basis. A bootloader performs load and execute by transferring the new image to program memory from EPROM, and then rebooting. The version advertisements provide a limited monitor, which gives version feedback to neighbors.

MOAP focuses on the performance of the propagation protocol, and provides limited support for advanced generation or activation features.

Maté

Maté [10] is a communication-centric WSN middleware layer for TinyOS, based on a virtual machine (VM) architecture.

Generation — Maté supports code in 24 instruction capsules; larger programs can be supported using subroutine capsules (currently limited to four). To *generate* the update, programs are assembled, and then insert takes place through the injection of the new modules (or new module versions) into the network using the *capsule injector* to a directly connected mote.

Propagation — Every node broadcasts a summary of 32-bit capsule version numbers to its neighbors, based on a random timer, to advertize an update. 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. This flooding continues even after the entire network has been updated. Decreasing the advertisement frequency when an identical vector is heard — at density-aware transmission rates — can provide an efficient trade-off between response time and avoiding network saturation. Each node *listens* for broadcast version vectors. If it hears a version summary with older capsules than it has itself, then Decision triggers transfer/send to broadcast the newer modules. Complete capsules are transferred in each message; the target VM does a transfer/recv for these. No *verify* is required, as code capsules fit into single packets; the continuous flooding handles lost packets.

Activation — When the target VM receives a new capsule version, it unloads the previous version, and *loads* the update. The new capsule version will be triggered for execute by an associated event. Neighbors effectively *monitor* the version update through the advertisements.

Maté uses "viral code propagation" to deliver mobile code updates to all nodes; it focuses on the propagation and mobile-code related activation aspects. It also provides limited support for advanced generation and activation features.

Impala/ZebraNet

Impala is the event-based middleware layer of ZebraNet [3], which uses wildlife tracking as a target application in the development of mobile wireless sensor networks.

Generation — Impala supports wide range of updates (from bug fixes, through updates, to adding and deleting entire applications). It uses version numbering to ensure compatibility of updates with existing modules. In *generate* applications can be formed of multiple modules (with an 8 KB module limit). Linking takes place on the target nodes, so unlinked binaries are prepared for injection. To insert the update, it is sent to the base station.

Propagation — Nodes advertize the version numbers of both complete applications and their constituent modules. Nodes *listen* to their neighbors' advertisements, and a decide triggers advertize to increase a backoff timer exponentially if all neighbors have the same software versions — this significantly reduces management traffic, but can delay updates if network connectivity to nodes with earlier software versions is lost and recovered. In transfer/recv, nodes make unicast requests (using node ID as a tie-breaker) to trigger transmission of newer modules only, thus saving on network bandwidth. Transfer/send nodes respond to requests from other nodes, by transmitting the code for the first requested modules. In transfer/recv, if memory space is exhausted, then older incomplete applications are deleted. Multiple contemporaneous updates (different versions of the same module) can be processed. To verify updates, lists are used to keep track of which modules have been received; partially received modules are re-requested on the next advertisement through co-operation with a decide. When all the modules in a particular update have been received, activation is triggered.

Activation — Simple sanity *checks* are performed (e.g., only valid memory accesses, and a return statement). To load an update, the old version application is terminated, the modules in the new version are linked in, and the pointers in the *application activation table* are updated. Code memory management is provided in 2 K blocks to cope with multiple applications each consisting of multiple modules. To execute after loading, the new application is first initialized. Applications can continue running while updates are being downloaded. *Monitor* is effectively provided to neighbors through the advertisements.

Impala concentrates primarily on the propagation of the update, but also provides somewhat more comprehensive support for generation and activation. The explicit support for modules can significantly decrease overhead.

Deluge

Deluge [1] is a data dissemination protocol for reliably propagating large amounts of data throughout a WSN using incremental upgrades for enhanced performance.

	Generation				Propagation				Activation							
	Plan	Build	Insert	Feed-back	Adver-tize	Xfer/ send	Listen	Decide	Xfer/ recv	Verify	Monitor	Decode	Check	Load	Exe-cute	Moni-tor
MOAP		✓	✓		✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓
Maté		✓	✓		✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓
Impala		✓	✓		✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓
Deluge					✓	✓	✓	✓	✓	✓						

■ Table 1. *Software update functionality comparison.*

Generation — Deluge does not itself support this.

Propagation — A 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. Nodes advertize using a broadcast containing a version number and a page bit vector, using a variable period based on updating activity. Nodes *listen* to their neighbors' advertisements, and if the decide determines that it needs to upgrade part of its image to match a newer version, then, after listening to further advertisements for a time, transfer/recv sends a request to the selected neighbor for the lowest page number required and the packets required within that page. After listening for further requests, transfer/send selects a page, and broadcasts every requested packet in that page. Verify triggers transfer/recv to either request new pages, or re-request missing packets from a previous page. When a node receives the last packet required to complete a page, verify triggers advertize to broadcast an advertisement before requesting further pages to allow parallelization of the update within the network. Heuristics are used by transfer/send to try and select relatively remote senders (to minimize radio contention). Incremental updates are supported by advertize, indicating which pages have changed since the previous image version; the decide then triggers transfer/recv to request just the changed pages.

Activation — Deluge itself does not support this: an associated bootloader, TOSBoot, allows Deluge objects to be invoked at boot time.

Deluge concentrates on the efficient propagation of the update, and includes many optimizations to minimize the reprogramming time and power used.

Comparison of Functionality Supported

The functionalitiy supported by each of the different WSN software update systems examined earlier is given in Table 1 for comparison. Note that only the areas of functionality addressed by each system are identified; the degree of support is not. A more complete comparison would need to address the other identified criteria: security, performance, reliability, usability, and portability.

Conclusions

Efficient software updating is in many ways one of the most challenging features to provide on a Wireless Sensor Network (WSN). It requires relatively large amounts of data to be reliably disseminated to the nodes, sophisticated mechanisms to minimize the cost of this dissemination, and aggregated status to be returned to a host. It may also require tracking of software failures and recovering from these failures in a network-wide manner. It must provide support to handle network partitioning, node failures, software failures, data transmission failures, and other intermittent and persistent faults. It is a mission critical application: its failure may cause loss of an entire WSN.

Dr. G. Parulkar from the National Science Foundation (NSF) highlighted the need for the consolidation of disparate solutions in WSN research in his keynote address to the Second IEEE Workshop on Embedded Networked Sensors (EmNetS-II) in Sydney, Australia on May 30, 2005. The work presented in this article addresses this need by providing a framework for consolidating research into WSN software updating. Future work will involve further validation of the model, and use of the model in integrating the best features of existing solutions to provide a more comprehensive approach for software updates.

References

[1] J. Hui and D. Culler, "The Dynamic Behavior of a Data Dissemination Protocol for Network Reprogramming at Scale," *Proc. 2nd Int'l. Conf. Embedded Networked Sensor Sys.*, Baltimore, MD, Nov. 2004, pp. 81–94.

[2] P. Levis *et al.*, "Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks," *Proc. First USENIX/ACM Symp. Network Sys. Design and Implementation*, San Francisco, CA, Mar. 2004, pp. 15–28.

[3] T. Liu *et al.*, "Implementing Software on Resource-Constrained Mobile Sensors: Experiences with Impala and ZebraNet," *Proc. 2nd Int'l. Conf. Mobile Sys., Apps. and Svcs.*, Boston, MA, June 2004, pp. 256–69.

[4] N. Reijers and K. Langendoen, "Efficient Code Distribution in Wireless Sensor Networks," *Proc. 2nd ACM Int'l. Wksp. Wireless Sensor Networks and Apps.*, San Diego, CA, Sept. 2003, pp. 60–67.

[5] L. B. Ruiz, J. M. Nogueira, and A. A. F. Loureiro, "MANNA: A Management Architecture for Wireless Sensor Networks," *IEEE Commun. Mag.*, vol. 41, no. 2, Feb. 2003, pp. 116–25.

[6] T. Stathopoulos, J. Heidemann, and D. Estrin, "A Remote Code Update Mechanism for Wireless Sensor Networks," CENS tech. rep. #30, Dept. Comp. Sci., UCLA, Nov. 2003.

[7] P.V. Krishnan, L. Sha, and K. Mechitov, "Reliable Upgrade of Group Communication Software in Sensor Networks," *Proc. 1st IEEE Int'l. Wksp. Sensor Network Protocols and Apps.*, Anchorage, AK, May 2003, pp. 82–92.

[8] C.-C. Han *et al.*, "Sensor Network Software Update Management: A Survey," *Int'l. J. Network Mgmt.*, no. 15, 2005, pp. 283–94.

[9] S. Brown and C. Sreenan, "A Survey of Software Updating in Wireless Sensor Networks," Tech. rep. UCC-CS-2006-13-07, Dept. Comp. Sci., University College Cork, Ireland, July 2006.

[10] P. Levis and D. Culler, "Maté: A Tiny Virtual Machine for Sensor Networks," *Proc. 10th Int'l. Conf. Architectural Support for Programming Languages and Op. Sys.*, San Jose, CA, Oct. 2002, pp. 85–95.

Biographies

STEPHEN BROWN (stephen.brown@nuim.ie) is a senior lecturer in computer science at the National University of Ireland, Maynooth, where he was acting head of department for several years. Previously he worked for Digital Equipment Corporation. His research interests are in embedded networked systems, with a particular emphasis on wireless sensor networks. He is a member of the Institute of Microelectronic and Wireless Systems in Ireland, and an IEI Chartered Engineer.

CORMAC J. SREENAN [M] is a professor of computer science at University College Cork, where he also served as head of department from 2000 to 2004 inclusive. Previously he was a researcher at AT&T (Bell) Labs, New Jersey. His research interests are in mobile and multimedia networking. He holds a Ph.D. in computer science from Cambridge University, and is a member of ACM and a Fellow of the British Computer Society.