INTERNATIONAL JOURNAL OF NETWORK MANAGEMENT
Int. J. Network Mgmt 2005; **15**: 283–294
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/nem.574

Sensor network software update management: a survey

*By Chih-Chieh Han, Ram Kumar, Roy Shea and Mani Srivastava**,†

Software management is a critical task in the system administration of enterprise-scale networks. Enterprise-scale networks that have traditionally comprised of large clusters of workstations are expanding to include low-power ad hoc wireless sensor networks (WSN). The existing tools for software updates in workstations cannot be used with the severely resource-constrained sensor nodes. In this article, we survey the software update techniques in WSNs. We base our discussion around a conceptual model for the software update tools in WSNs. Three components of this model that we study are the execution environment at the sensor nodes, the software distribution protocol in the network and optimization of transmitted updates. We present the design space of each component and discuss in-depth the trade-offs that need to be considered in making a particular design choice. The discussion is interspersed with references to deployed systems that highlight the design choices. Copyright © 2005 John Wiley & Sons, Ltd.

Introduction

The recent advances in MEMS, microelectronics and the advent of ultra-low-power radio technologies such as Zigbee have made sensing, computation and communication truly pervasive. Wirelessly connected networks of resource-constrained sensor nodes in various forms are gradually making their way into the market and our lives. Technologies such as radio frequency identification (RFID) are already changing the nature of supply-chain management for big retailers.[1] Intelligent sensor networks are increasingly being deployed for safeguarding critical infrastructures.[2] M2M (machine-to-machine) networks are being used by Intel to monitor the health of the expensive fabrication machinery.[3] Despite many potential benefits, the success of sensor network technology hinges upon its ease of use and maintenance. Indeed, sensor networks will never succeed if they require constant management from an entire IT department. The large scale of sensor networks, the limited resources at every node, and the deployment in environments with high access cost, make the task of managing and operating these systems extremely challenging. This paper focuses on the task of software updates in wireless sensor network systems. While software updates to fix bugs and improve features are common in all systems, sensor networks come with the additional challenge of updating devices with limited computational and energy resources. Tools to help update software in traditional systems have been used for a number of years. Enterprise network administrators often need to check for software updates, download them, verify the integrity and finally distribute them to hundreds of nodes in the network. A tool that helps with this task is the Microsoft Systems Management Server (SMS).[4] SMS includes support for planning application deployment, selective targeting of application deployment, security

**Correspondence to: M. Srivastava, Department of Electrical Engineering, University of California, 6731-H Boelter Hall, Los Angeles, CA 90095-1594, USA.*
†*E-mail: mbs@ee.ucla.edu*

patch management and the ability to propagate differential changes to software rather than the entire application image. The Tivoli Configuration Manager[5] is another tool from IBM, which provides automatic patch distribution, multicast based software distribution, and pervasive device configuration management. Another example is the HP OpenView Change and Configuration Management[6] tool. In addition to supporting all major operating systems, HP OpenView provides impact analysis prior to software distribution, assures compliance with security and business policy, and supports intermittently connected mobile devices. Existing update management (UM) tools are designed to aid system administrators with two primary problems. First, the diverse set of software and hardware configurations present in a large network presents a great deal of complexity that cannot be effectively handled by a human. UM tools often provide inventory tracking and selective installation based on specific software/hardware combinations to help reduce this complexity. Second, UM tools provide a mechanism for content distribution in the network.

Sensor networks will never succeed if they require constant management from an entire IT department.

In addition to helping with management complexity and content distribution, management tools in the WSN domain must also take into account the severe resource constraints at individual nodes and the wireless medium of communication. These constraints render the existing solutions for the traditional systems ineffective for WSNs. Therefore new techniques need to be developed for software update management in WSNs.

In this article, we propose a distributed architecture model that can be used to design and evaluate software update management tools. There are five components in the model, and we study three components that are related to WSNs: the execution environment at the sensor nodes, the distribution protocol in the network and data compression of transported software. We proceed to describe the design space of each component and discuss in-depth the trade-offs that need to be considered when making a particular design choice. The discussion is interspersed with examples of deployed systems to highlight different design choices.

We continue with a presentation of the components of the UM tool model in the next section. The third section describes three types of execution environments present in WSNs. The forth section describes the choices available for update distribution protocols. The fifth section shows the techniques used for optimizing the size of updates. We conclude in the sixth section.

Update Management Model

Figure 1 shows a conceptual model of an update manager for wireless sensor networks. The *Base Station (BS)* on the left represents a server connected to the sensor network that is not resource constrained. The *Sensor Network (SN)* on the right is made up of many sensor nodes, each of which operates with severe resource limitations. In the context of this paper, we use the term 'resource' in a traditional way to refer to energy, network bandwidth, CPU computation and memory. A sensor node is typically equipped with a low bandwidth radio, an MHz (instead of GHz) micro-controller, and kilobytes (instead of megabytes) of RAM to provide prolonged operations under battery power.

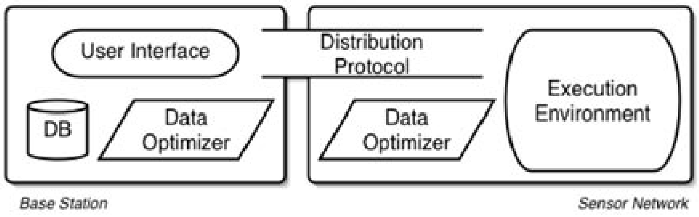

Figure 1. Update manager model

The model is made up of five main components:

1. *User interface (UI)* is used by system administrator to plan for software updates, select potential targets based on system parameter and monitor the progress of the updates.
2. *Database (DB)* stores software images from past updates, potential images or patches for current and future updates, and keeps system parameters from the network needed for target selection and update planning.
3. *Data optimizer (DO)* minimizes the size of updates before they are distributed to the network.
4. *Distribution protocol (DP)* transmits updates into targeted sensor nodes and provides some form of reliability.
5. *Execution environment (EE)* executes programs on a sensor node.

A typical update process that involves all five components could be the following. A system administrator uses the UI to plan a bug fix for a sensor driver installed in the network. The UI pulls the new and old sensor drivers from the DB and passes them to the data optimizer, which compares the two versions and generates a compressed patch. The patch is passed on to the distribution protocol for delivery to nodes in the sensor network. When the patch arrives at a destination node, the execution environment uses the patch to fix the sensor driver and restart the sensing service.

This paper details update operations occurring in wireless sensor network, while abstracting away operations at the UI and DB running on the base station. Although the techniques used in sensor nodes typically involve cross-component optimization, this paper treats the data optimizer, distribution protocol and execution environment as logically separate entities. The next three sections will discuss them each in turn.

Execution Environments

An execution environment supports program execution on top of the hardware provided by a node in a network. Heterogeneous hardware availability and application needs in the WSN domain have led to a variety of specialized execution environments customized to make best use of the available underlying hardware. A significant hardware feature is the memory management unit (MMU). When present, an MMU is able to significantly ease the software update process owing to its ability to present a virtual memory space, making much of the code position independent and adding safety boundaries to programs. Unfortunately many of the small microcontrollers used in WSNs are not equipped with an MMU. Without an MMU, a node operates in a single address space that is vulnerable to incorrect or malicious memory references and is not friendly to position independence. This section examines execution environments with an MMU and those without an MMU. Execution environments running on hardware without an MMU are further subdivided into three categories: monolithic environments, modular environments and virtual machines (VM).

—Execution Environments with an MMU—

Some WSN nodes use a processor equipped with an MMU, but are typically limited to a smaller number of high-power nodes used to manage groups of lower power nodes. Two examples of such systems include the Stargate[7] and the iPAQ. These systems often run a mature operating system such as Linux.[8] Example execution environments for this class of device are EmStar[9] from UCLA, Impala*[10,11] from Princeton, and SensorWare[12] from UCLA. EmStar uses a modular abstraction implemented on top of Linux. Individual modules are executed as separate processes that each benefit from the memory safety provided by the MMU. Since processes run in a virtual address space they can be easily upgraded at run time without needing to worry about references to absolute memory address. SensorWare implements a modified TCL interpreter to provide a VM for users, also running on top of Linux. While not critical for its operation, the MMU significantly eases the implementation of the TCL interpreter used in SensorWare. Impala allows updates in the form of a software module to be installed on a running system. Each module implements event handlers to process the events dispatched by Impala middleware.

**Impala has been ported to an MMU-less device without update functionality.*

—Execution Environments without an MMU—

Many low-power WSN devices are based upon microcontrollers without MMUs. Execution environments for these systems must be designed to operate in the single address space provided by the microcontroller. Three possible execution environments include: monolithic environments, modular environments and VM. Each of these environments optimizes for slightly different program needs.

Monolithic environment—Monolithic environments strive to provide efficient program execution by statically optimizing the entire program executable during compilation. This results in a system image with the application and system kernel intertwined to make for efficient use of CPU and memory. Examples of this are TinyOS[13] from Berkeley and Mantis[14] from the University of Colorado, Boulder.

Because the executable is a monolithic image, the update process typically constructs a new image in a secondary storage such as external flash or EEPROM.[15,16] It is therefore important for monolithic environments to include algorithms and protocols required for updates in the static image build. While monolithic environments provide a very efficient execution environment, the energy cost of updates can be high. The entire new system image needs to be on the destination node, often resulting in large patches that increase network expenses and image reconstruction costs on costly external storage. Under 'Date Optimizer', below, we will talk about techniques to relieve this by minimizing the size of the update.

During a software update, it is sometimes necessary to preserve the data or states associated with the program. Examples of such program states are neighbor tables associated with link characteristics, routing tables, and parameters of sensor calibration. Saving and restoring these states can be a complex operation during updates of monolithic environments when memory is allocated statically for performance reasons. Since memory allocation for software states is resolved at compile time, monolithic environments have no inherent semantic knowledge about memory layout. Data access is often accomplished through direct RAM addressing, resulting in a system that depends on data being at specific absolute addresses. Thus, simply copying RAM contents to persistent storage before an update and copying it back after an update does not work when there is any rearranging of memory. While we are not aware of any work in this domain, there appear to be a number of possible solutions. Each piece of state in memory can have an owner responsible for storing the state before an update and restoring the state after the update. This approach allows the system to only store states that are needed and store each in a subsystem-specific manner. Another approach is to name all state data so that they are independent of RAM addresses. This indirect addressing of state allows for easy storage and restoration of state by the kernel, and provides an easy means to share state between independent system components.

A combination of no MMU, CPU constraints, and the entire system image being replaced during an update makes safety very challenging in monolithic environments.

A combination of no MMU, CPU constraints, and the entire system image being replaced during an update makes safety very challenging in monolithic environments. Traditional solutions based on proof-carrying code[17] no longer apply. The type of processor used in these lower-power sensor nodes is highly unlikely to be able to execute a compiler or even verify proof-carrying code. Further, the lack of an MMU makes fault isolation difficult. One solution might be based on software sand boxing in a single address space, as described in Wahbe *et al*.[18] to catch illegal memory references at run time via checks added around instructions at compile time. However, this solution will not work for malicious code that has no illegal memory references.

Modular environment—Modular environments provide an environment where individual modules can be independently loaded on demand. This results in a very flexible system. Examples of modular systems include Contiki[19] from the Swedish Institute, Bertha[20] from MIT and SOS[21] from UCLA.

In a modular environment the system is divided into two parts: a static image (kernel) and the loadable component images (modules). The kernel provides facilities such as communications, I/O and memory management that modules use to extend the functionalities of the system. Modules access kernel services at predefined address or through a system jump table. Since modules are loosely coupled, communication between them tends to be indirect and more expensive than simple function calls. Direct memory access from modules is difficult because memory allocation is resolved during module loading time instead of compilation time. Compared to monolithic environments, the efficiency of the execution is lower due to lack of global optimizations at compilation time.[21] However, updates can be inexpensive since patches are localized and external memory is rarely needed during patching. Modular environments can save energy compared to monolithic environments when they are used for a deployment needing updates to its code base. One challenge for modular environments is to have a kernel that supports flexible operations while minimizing the kernel size and complexity, so that a kernel update is a rare event. At the same time the kernel should minimize the overhead from the provided flexibility.

Updates in modular environments are relatively easy compared to those in monolithic environments. The kernel can be updated using the same techniques used in monolithic environments but with a lower energy cost. The kernel, once stabilized, requires less frequent updates than monolithic environments because the static image consists of select hardware drivers and resource management routines that are can undergo more extensive testing and expect a stable development cycle. Even when an update is necessary, the size of static image is smaller than its monolithic counterpart because the application-specific part of the system does not affect the kernel update. When updating modules there is no need to save the states of other modules. This feature comes naturally in a modular environment because modules are inherently relocatable and address independent. Since some form of dynamic memory is often used for module state, it is a simple task for the kernel to isolate the state of a module. Finally, the smaller size of modules allows them to be updated in-place, without the need for external storage that typically requires high energy for writing.

Modular environments do no better than monolithic environments for code safety. However, if a fault within a module can be found, removing the faulty module is an easy task in a modular system.

Virtual machine environment—VM environments virtualize underlying hardware to provide high-level operations to applications through an instruction interpreter. This enables the creation of scripts that can be loaded onto a sensor node at any time after deployment. An example of this type of system is the Maté VM[22] from Berkeley.

Since the environment is implemented in software, it is possible to support very high-level instructions. An example could be the creation of a single instruction that reads a sensor and sends the returned value to the network. High-level instructions allow for the construction of compact scripts with small network overhead that can be run directly out of RAM. Finding the right level of abstraction for high-level instructions is a challenging problem. If an instruction is too specific it may not be useful after a software update, but overly general instructions result in increased script sizes, reducing the benefits of using a VM.

Perhaps the most beneficial property of a VM is code safety. VM typically interprets a program in a sandbox where direct access to hardware is not possible. Since all accesses to resources are managed indirectly through VM, code safety is provided without additional cost.

Updates in a VM environment can take the form of new application scripts. Since these scripts can be very small the network cost is minimized and the script can run directly out of RAM, resulting in a very low update cost. The interpretation of VM instructions requires more work than executing native programs, resulting in a higher execution cost.[21] As updates become more frequent, the savings in update costs of a VM environment make up for the increased execution cost.

—Summary—

This section briefly examined execution environments for nodes with an MMU and then focused on execution environments for nodes without an MMU. A summary of the specialized

Energy costs	Monolithic environment	Modular environment	Virtual machine environment
Network transmission	Worst	Middle	Best
External storage	Worst	Middle	Best
On-chip storage	Worst	Middle	Best
Program execution	Best	Middle	Worst
Update flexibility	Best	Middle	Worst
Example system	TinyOS	SOS	Mate

Table 1. Relative comparison of different execution environments. Note that data optimization through compression can be used for data transmission in all three types of environments

Energy costs	VM + monolithic environment	VM + modular environment
Update tasks in VM	Low	Low
Energy cost of updating VM	High	Low
Cost of interpretation	Low	High

Table 2. Relative comparison of multi-environments. The table presents the trade-offs when VM is implemented on top of the other two execution environments

categories is presented in Table 1. An interesting area to examine is the role of multiple execution environments in a single deployment. An example includes running a VM on top of a modular system or splitting, changing the amount of functionality supplied by the VM compared to the underlying system. This would allow efficient execution of core services that rarely change while continuing to support easy modification to simple applications running on the VM.

The choice of environment used by VM presents similar trade-offs, as has been shown between monolithic and modular environments. When a monolithic environment is used, adding one VM instruction would mean updating the whole monolithic environment. On the other hand, a modular environment allows ease of adding new instructions in the form of a module with slightly higher interpretation overhead. We are currently investigating VM on top of a modular environment. In this model, each module implements one or more VM instructions. Since modules can be added on the fly, the capability of VM can be enhanced at run time with lower cost of update. Table 2 shows a comparison of multi-environments.

Distribution Protocol

Distribution protocols deliver software updates to intended target sensor nodes. From a network management point of view, this can be divided into three conceptual steps:

1. *Select* intended targets.
2. *Deliver* software updates.
3. *Verify* the completion of update.

Unless the target is intended for every sensor node, a naming convention is required. This can be an attribute–value pair used in Directed diffusion.[23] Once the target is selected, delivering software updates can be thought of as a variant of data multicast. Although data distribution protocols have been studied in both wireless ad hoc networks[24,25] and in distributed systems,[26–28] unique characteristics of sensor networks and the nature of software update limit the applicability of these solutions.[29–35]

First, communication is very expensive in WSNs. This is a result of the combined cost of the radio itself and, when the radio is driven by a software driver on the main microprocessor, CPU time is dedicated to managing the radio. Second, the battery-powered sensor nodes require minimization of distribution protocols for overall energy usage rather than only optimizing for network overhead. Although radio is the major energy consumer, CPU and external flash can consume significant energy during the active state. For example, software updates can be large enough to overflow into costly secondary storage that can consume an order of magnitude more energy.[33,36]

Third, wireless link failures are a common event[37] and sensor nodes can lose connectivity for a prolonged amount of time.[38] Such a link failure can cause a node to miss an update during distribution. Hence, the members of sensor network are considered to be dynamic rather than static. Fourth, software updates are different from pure data delivery. Software, once updated, will produce data, and it is possible that the updated data format is different from old one. Lastly, the nature of software updates requires successful delivery of the entire update in a timely manner.[34] In contrast to best effort delivery used in some data-gathering applications, an update is only valuable when it is complete.

In the following, we will summarize the research efforts in target selection, software delivery and update completion verification. It is important to keep in mind that the sensor network research community has largely focused on efficient data delivery. Many of the works assume that the targets of an update are selected and the routing tree has been built, or the intended targets are simply all of the sensor nodes. We have only seen one paper that deals with completion verification, using report generation.[30] Therefore, we will highlight the challenges and sketch possible solutions.

—Target Selection—

To enable target selection, management tools often provide naming services for grouping target sensor nodes. The naming scheme can be as simple as the address of the node or as complex as attribute–value pair with operators on attributes. Because the topology of the sensor network is not fixed owing to wireless link failure, a multicast tree (i.e., path to intended sensor nodes) cannot be statically built. Further, some attributes can only be evaluated at run time, such as battery level and sensor readings. The best the system can do is to compress the name and defer the evaluation of target membership to run-time evaluation. Such a name becomes the metadata attached to the update for distribution. The metadata is then used during distribution to decide whether to install the update. Although it is possible to use metadata for forwarding decisions, in many situations only the geographic information (i.e., the location of a sensor node) is of use for routing. That is, when metadata carries desired location information as a qualifier, it is possible to evaluate only the metadata for nodes in the bounding area. For most other cases, the metadata must be evaluated on individual sensor nodes. This means that the metadata has to be delivered to every sensor node in the network, bounded by any established location properties, to guarantee the coverage of intended targets. Further, to avoid expensive flash writing of large updates for sensor nodes that are not intended, it is beneficial to deliver metadata first instead of appending metadata in the update.

It is instructive to compare the data naming proposed in Directed diffusion[23] against software naming. To execute a piece of software, additional resources are needed. Typically, software occupies the CPU and controls blocks of memory for a certain amount of time. Therefore, metadata for software updates should also serve the purpose of preventing over-subscription of resources.

—Update Delivery—

Once the targets are selected, delivering updates becomes a multicast problem. This is a two-step process: metadata flooding and multicast update. Metadata can be used to determine the membership of a multicast group, and then the update is delivered through the multicast group. Since it is possible for a sensor node to lose connectivity for a prolonged amount of time,[38] most application domains require that the update process needs to be continuous throughout the lifetime of the network.[34] This means that the system must try to synchronize the metadata throughout the network and form multicast groups on demand when metadata is evaluated to be true. For this architecture to work, a lightweight flooding algorithm and on-demand multicast protocol are needed. The current state of sensor network research focuses on efficient flooding, with very little work on on-demand multicast. In the following, we will focus on efficient flooding. Interested readers can find on-demand multicast routing in ad hoc network research such as ODMRP.[39]

Probably the earliest work of efficient flooding in WSN is SPIN (Sensor Protocols for Information via Negotiation).[29] SPIN follows a three-way hand-

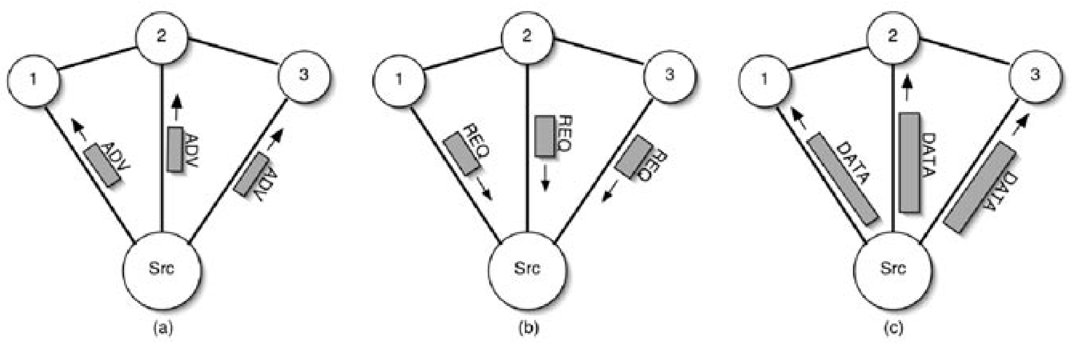

Figure 2. SPIN protocol handshake: (a) source node broadcasts advertisement; (b) those nodes that need the data request from source; (c) source node broadcasts data

shake as shown in Figure 2: *ADV*, *REQ*, and *DATA*. In ADV phase, each sensor node does a one-hop broadcast to advertise what it has in a compact format. If the node needs the data, it then sends REQ back to advertiser. Advertiser, upon hearing REQ, does a one-hop broadcast DATA. The benefit of this scheme is that a node will not receive redundant data, as happened in classic flooding.

Trickle[34] improves the scheme by noticing that ADV needs to be periodically broadcast, and this can consume a significant amount of energy over time. It solves the problem by borrowing the suppression mechanism in SRM (Scalable Reliable Multicast).[40] Specifically, the node will only broadcast ADV when it has not heard a few other nodes transmit the same ADV. Trickle further uses implicit REQ in that it will broadcast DATA when it hears others sending old ADV. This allows Trickle to be scalable even in a high-density network.

Deluge[33] extends Trickle to support efficient flooding of large data. It divides large data into segments that are bigger than radio MTU (maximum transmit unit) and restricts each sensor node to request data from each segment sequentially. That is, each sensor node will request data from one segment until the transfer of the segment is completed. In a sense, the flooding is staged into each segment. The key idea is to localize the error (i.e., packet loss) and hence lower the cost of recovery.[30] The benefits are twofold: the state required for recording current available data in the segment is minimized, and it is possible to spatially multiplex the data transmission in a segment.

—Completion Verification—

The last step in software distribution is to verify the completeness of the update. From a management point of view, the information can be used to as certain the stableness of the network. For example, one of the primary functions of WSN is to gather data. It is necessary to know whether the program used to gather data is up to date. It might sometimes be desirable to know which and when nodes are up to date—information that can be used to reject invalid data. Based on the above, we believe that completion verification should consist of three types of information:

1. Percentage of nodes that are up to date for tracking the progress of update.
2. The ID of the nodes that are up to date.
3. The time the nodes were last updated.

The implementation of completion verification can be any data-gathering application, such as Directed diffusion[23] or TinyDB.[41]

—Summary—

A well-designed update distribution protocol consists of three components: target selection, update delivery and completion verification. All three components have been implemented separately in the context of WSNs. A successful implementation is likely to include a subset of all three components. This means that a significant amount of program memory will be occupied for enabling update distribution. For updates in a monolithic environment, the protocol will be transmitted over the air. For a modularized and VM environment, this implies an even bigger kernel. It is important to note that the distribution protocol is a service to allow updates, but itself does not provide a direct utility to users. The challenge of the distribution protocol is to minimize the required functionalities while keeping it flexible to handle future updates.

Data Optimizer

While the distribution protocol tries to minimize the overhead associated with delivering updates to the sensor nodes, the data optimizer minimizes the size of updates. To minimize the size of updates over the network, the encoder on the base station performs lossless compression on the updates and then passes the update to the distribution protocol. When this compressed update arrives in sensor nodes, a decoding process will be used to reconstruct the updates for the targeted environment. In sensor networks, resource constraints complicate this process.

Sensor nodes are typically equipped with a CPU having limited processing power to conserve energy. Decoding can be CPU intensive, increasing CPU active time during the update process and potentially offsetting energy savings due to decreased transmission expenses. Additionally, access to external storage consumes a significant amount of energy.[36,42] Since the size of an update can be much larger than the size of in-system volatile memory, the decoding process may have to run in external memory, which provides larger capacity at a higher energy cost. In a wireless environment where packet loss is common, buffering updates can be costly owing to external storage access. It would be ideal to allow each fragment of an update to be processed independently to avoid buffering.

A significant challenge to software updates in WSNs is choosing a lossless data encoding that can be efficiently done on highly resource-constrained sensor nodes.

A significant challenge to software updates in WSNs is choosing a lossless data encoding that can be efficiently done on highly resource-constrained sensor nodes. We describe three potential approaches to solving this challenge: data compression, differential patching, and using high-level languages.

—Compression—

Compression algorithms are used to minimize the size of updates. Typical usage would compress updates at the base station and then distribute the compressed update to sensor nodes for decompression. Although this scheme is fairly easy to implement, as compression libraries are widely available, care must be taken when choosing decompression algorithms. Barr and Asanovic[42] have shown that simply sending fewer bits may not reduce the total energy of a task. This is seen by examining the causes of energy consumption that occur when sending compressed data: CPU, memory, and network. Energy saved from sending smaller network packets may be overwhelmed by increased energy costs from the CPU. For example, choosing a compression algorithm with a slightly better compression ratio might be bad because of an order of magnitude greater CPU usage during decompression. In choosing compression algorithms, it is important to be aware of the marginal gain of network energy and the marginal loss of memory and CPU energy. Thus, a compression algorithm must be chosen such that it is optimized for memory and CPU usage in addition to compression ratios.

—Differential Patching—

When an update is an incremental improvement on a deployed procedure, such as a bug fix or parameter change, the new image is often very similar to the old version. When this is the case, the difference between two images can be transmitted as the update. Typical operation in this scheme is to describe the difference between two versions in terms of an edit script. Such a script often uses commands that can copy data from the old image, modify data from the old image, and insert new data.[15,16] This edit script is sent to the sensor node for reconstruction. Although this basic scheme works quite well for minor changes, larger changes require a more complicated edit script language.

Often a small change is made in the source code, resulting in a similar but shifted new binary

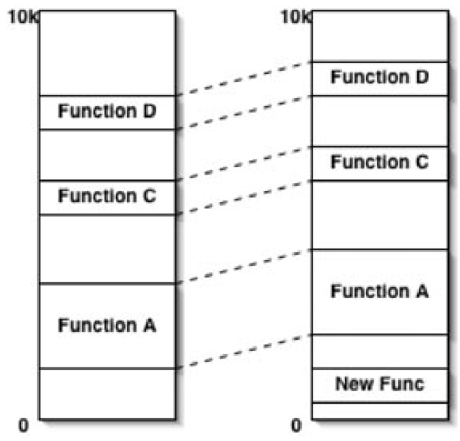

Figure 3. Address change due to new function

image. Figure 3 shows this situation. In Reijers and Langendoen,[15] the authors suggested a new script command called 'patch list' to shift the address from the old image to the new image, moving groups of functions together. A list of {*begin address, end address, offset*} can be used to reconstruct the new image instead of listing all the differences in terms of addresses. Unfortunately, this solution is instruction format specific. In Jeong and Culler,[16] the authors describe an approach derived from rsync[43] that is independent of instruction set. The key idea is to divide the image into a fixed size block, and then try to match the old block at any offset using a rolling checksum. A third solution to this problem is the use of relocatable updates. For example, SOS[21] supports dynamically loadable modules that are relocatable.

—High-Level Instructions—

While not strictly a compression technique, the use of high-level instructions for a virtual machine[12,22] can optimize the amount of data sent over a network. The key idea is to abstract the real hardware with commonly used and high-level instructions that support common combinations of underlying tasks, such as reading a sensor value and then sending the result to a base station. When there is a need for an update, a new script can be written to describe the task. Since the instructions are very high level, this script is often very compact despite its complex functionality.

—Summary—

We have described three approaches for minimizing the size of updates. Note that the above three options are not exclusive. Compression could be used on a differential update of a script running on a VM. In this case a differential update is generated between the new update and current task for VM, then the differential is passed to a compression algorithm to produce the final form of update.

This section has described the techniques independent of distribution protocols. In practice the data optimizer typically interacts with the distribution protocol to reduce the cost of update. For example, decompression may be done on packet boundaries to avoid buffering.[15] The distribution protocol transmits compressed fragments of updates, while the data optimizer generates new fragments. The potential benefit is that there is no need to wait for all fragments before beginning decompression. However, such an approach may not be suited for distributed distribution protocols where in-network caching is used heavily. Specifically, if the update is decompressed on an intermediate node without caching the compressed packet, retransmission from the cache has either to transmit a decompressed version or perform compression on the fly.

Conclusion and Future Directions

Execution environments present a trade-off between execution efficiency and update overhead. Total power consumption of an update process can be defined as the long-term (execution) power consumption plus the short-term (update) power consumption. Long-term power consumption depends on energy consumption of an execution environment and the frequency of execution. Short-term power consumption depends on energy consumption of the update process and the frequency of update requests. It is clear that when the frequency of updates is high, it makes sense to lower the cost of the update process at the expense of a more expensive execution environment. The challenge is then to know in advance the types and frequencies of updates that will be likely, so that an appropriate execution environment can be used.

Update distribution protocols trade off between update reliability and the cost of an update. When inconsistency can be handled internal to the network, the value of a completely reliable update mechanism decreases. With research efforts toward eventual consistency, the challenge is to provide better language support for handling inconsistencies.

Compression and differential patching provide a way to reduce the size of an update. Since the cost of accessing external storage is high, it might be better to avoid caching all together and patch the software directly on top of the running program. On the other hand, the cost of distributing an update becomes higher due to lack of local cache for quick recovery. One simple improvement may be caching some fragments of an update in some nodes so that the cost of recovery is lower.

The area of update management is still young in sensor networks and requires more effort. We hope this article serves as an initial call for help.

References

1. Rfid walmart way. http://www.themaelstrom.net/log/2003/06/27/rfid_walmart_way.php, 2003.
2. Structural health monitoring of the Golden Gate Bridge. http://www.eecs.berkeley.edu/binetude/ggb/.
3. Chhabra J, Kushalnagar N, Metzler B, Sampson A. Sensor networks in Intel fabrication plants. In *SenSys '04: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*. ACM Press, 2004; 324.
4. Microsoft Systems Management Server. http://www.microsoft.com/smserver.
5. IBM Tivoli ConfigurationManager. http://www-306.ibm.com/software/tivoli/products/config-mgr/.
6. HP OpenView Change and Configuration Management. http://www.managementsoftware.hp.com/solutions/ascm/index.html.
7. Linux on Stargate. http://platformx.sourceforge.net/Documents/manuals/LinuxFeatures04.pdf.
8. Familiar project. http://familiar.handhelds.org/.
9. Girod L, Stathopoulos T, Ramanathan N, Elson J, Estrin D, Osterweil E, Schoellhammer T. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems*, Baltimore, MD, 2004.
10. Liu T, Martonosi M. Impala: a middleware system for managing autonomic, parallel sensor systems. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 107–118, San Diego, CA. ACM Press, 2003; 107–118.
11. Liu T, Sadler CM, Zhang P, Martonosi M. Implementing software on resource-constrained mobile sensors: experiences with impala and zebranet. In *MobiSYS '04: Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services*. ACM Press, 2004; 206–269.
12. Boulis A, Han C-C, Srivastava MB. Design and implementation of a framework for efficient and programmable sensor networks. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services*, San Francisco, CA. ACM Press, 2003; 187–200.
13. Hill J, Szewczyk R, Woo A, Hollar S, Culler D, Pister K. System architecture directions for networked sensors. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA. ACM Press, 2000; 93–104.
14. Abrach H, Bhatti S, Carlson J, Dai H, Rose J, Sheth A, Shucker B, Deng J, Han R. Mantis: system support for multimodal networks of in-situ sensors. In *Proceedings of the 2nd ACM International Conference on Wireless Sensor Networks and Applications*. San Diego, CA. ACM Press, 2003; 50–54.
15. Reijers N, Langendoen K. Efficient code distribution in wireless sensor networks. In *Proceedings of the 2nd ACM International Conference on Wireless Sensor Networks and Applications*, pages 60–67, San Diego, CA. ACM Press, 2003; 60–67.
16. Jeong J, Culler D. Incremental network programming for wireless sensors. In *Proceedings of the First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (IEEE SECON)*, Santa Clara, CA, 2004.
17. Necula GC. A scalable architecture for proof-carrying code. In *FLOPS*, Tokyo, 2001; 21–39.
18. Wahbe R, Lucco S, Anderson TE, Graham SL. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, **27**(5): 203–216.
19. Dunkels A, Gronvall B, Voigt T. Contiki: a lightweight and flexible operating system for tiny networked sensors. In *1st IEEE Workshop on Embedded Networked Sensors (EmNetS-I)*, Tampa, FL, November 2004.
20. Lifton J, Seetharam D, Broxton M, Paradiso J. Pushpin computing system overview: a platform for distributed embedded, ubiquitous sensor networks.

In *Proceedings of the Pervasive Computing Conference*, London, 2002.

21. Han C-C, Kumar R, Shea R, Kohler E, Srivastava M. A dynamic operating system for sensor nodes. *Technical Report NESL-TM-2004-11-01*, University of California Los Angeles, Networked Embedded Systems Lab, November 2004.
22. Levis P, Culler D. Mate: a tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October, 2002.
23. Intanagonwiwat C, Govindan R, Estrin D. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Mobile Computing and Networking* 2000; 56–67.
24. Luo J, Eugster PT, Hubaux J-P. Route driven gossip: probabilistic reliable multicast in ad hoc networks. In *INFOCOM 2003*, San Francisco, CA, 2003.
25. Ni S-Y, Tseng Y-C, Chen Y-S, Sheu J-P. The broadcast storm problem in a mobile ad hoc network. In *MobiCom '99: Proceedings of the 5th annual ACM/IEEE International Conference on Mobile Computing and Networking*. ACM Press, 1999; 151–162.
26. Byers JW, Considine J, Mitzenmacher M, Rost S. Informed content delivery across adaptive overlay networks. *IEEE/ACM Transaction on Networking* 2004; **12**(5): 767–780.
27. Cuenca-Acuna FM, Peery C, Martin RP, Nguyen TD. Planetp: using gossiping to build content addressable peer-to-peer information sharing communities. In *HPDC '03: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, IEEE Computer Society, 2003; 236.
28. Demers A, Greene D, Hauser C, Irish W, Larson J, Shenker S, Sturgis H, Swinehart D, Terry D. Epidemic algorithms for replicated database maintenance. In *PODC '87: Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*. ACM Press, 1987; 1–12.
29. Heinzelman WR, Kulik J, Balakrishnan H. Adaptive protocols for information dissemination in wireless sensor networks. In *MobiCom '99: Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*. ACM Press, 1999; 174–185.
30. Wan C-Y, Campbell AT, Krishnamurthy L. Psfq: a reliable transport protocol for wireless sensor networks. In *WSNA '02: Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications*. ACM Press, 2002; 1–11.
31. Stann F, Heidemann J. Rmst: reliable data transport in sensor networks. In *Proceedings of the First International Workshop on Sensor Net Protocols and Applications*, Anchorage, AK, IEEE, 2003; 102–112.
32. Stathopoulos T, Heidemann J, Estrin D. A remote code update mechanism for wireless sensor networks. *Technical Report CENS-TR-30*, University of California, Los Angeles, Center for Embedded Networked Computing, November 2003.
33. Hui JW, Culler D. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the Second International Conference on Embedded Networked Sensor Systems*, Baltimore, MD. ACM Press, 2004.
34. Levis P, Patel N, Culler D, Shenker S. Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of the First Symposium on Networked Systems Design and Implementation*, San Francisco, CA. USENIX Association, 2004; 15–28.
35. Levis P, Culler D. The firecracker protocol. In *Proceedings of the 11th ACM SIGOPS European Workshop*, Leuven, Belgium, 2004.
36. Shnayder V, Hempstead M, Chen B, Welsh HM. Powertossim: efficient power simulation for TinyOS applications. In *Sensor Networks: Proceedings of ACM SenSys*, Los Angeles, CA, 2003.
37. Zhao J, Govindan R. Understanding packet delivery performance in dense wireless sensor networks. In *Proceedings of the First International Conference on Embedded Networked Sensor Systems*, Los Angeles, CA. ACM Press, 2003; 1–13.
38. Szewczyk R, Polastre J, Mainwaring A, Culler D. Lessons from a sensor network expedition. In *Proceedings of the First European Workshop on Sensor Networks (EWSN)*, Berlin, 2004.
39. Lee S, Su W, Gerla M. On-demand multicast routing protocol in multihop wireless mobile networks, *Mobile Networks and Applications* 2002;**7**(6):441–453.
40. Floyd S, Jacobson V, Liu C-G, McCanne S, Zhang L. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking* 1997; **5**(6): 784–803.
41. Madden S, Franklin MJ, Hellerstein JM, Hong W. The design of an acquisitional query processor for sensor networks. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. ACM Press, 2003; 491–502.
42. Barr K, Asanovic K. Energy aware lossless data compression. In *First International Conference on Mobile Systems, Applications, and Services*, San Francisco, CA, May 2003.
43. Tridgell A. Efficient algorithms for sorting and synchronization. PhD thesis, Australian National University, 1999. ■