Disseminating Code Updates in Sensor Networks: Survey of Protocols and Security Issues

Patrick E. Lanigan, Rajeev Gandhi, Priya Narasimhan

October 2005
CMU-ISRI-05-122

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Network reprogramming allows for over-the-air application updates in sensor networks. We describe the operation of a number of network reprogramming protocols that have emerged for the TinyOS sensor network operating system. We go on to discuss potential security issues that arise from the operation of network reprogramming protocols.

We acknowledge support through the NSF CAREER grant CCR-0238381 and the Army Research Office grant DAAD19-02-1-0389 ("Perpetually Available and Secure Information Systems") to the Center for Computer and Communications Security at CMU.

Keywords: network reprogramming, security, sensor networks, code dissemination, code update

Contents

1 Introduction

Wireless sensor networks (WSN) are typically long-lived, large scale and deeply embedded. In many cases, it might be necessary to distribute application updates to the sensor nodes in order to fix bugs or add new functionality. Loading a new application or upgrading an existing application on a sensor node traditionally requires a physical connection to the node, typically via a serial port or some other hardwired back-channel. However, physical access to nodes is in many cases extremely limited following deployment. In low-impact environmental monitoring applications, dispatching maintenance personnel to collect nodes or flash them in the field could adversely affect the environment being monitored. Physical access might even be impossible, as in the case of monitoring sensors poured into concrete in bridges and roadways. Even if physical access were possible, manually updating hundreds or thousands of nodes would be a tedious task indeed. Network reprogramming protocols have recently emerged as a way to distribute application updates without requiring physical access to sensor nodes. These protocols distribute upgrades using the sensor's primary communication channel (i.e. the radio), allowing an entire sensor network to be updated *en masse*. Clearly, network reprogramming is a necessary capability for the maintenance of wireless sensor networks.

1.1 Organization

The field of network reprogramming has been studied over the past few years, and this survey is an attempt to describe and assess the current state-of-the-art. We focus mainly on protocols designed for distributing monolithic application images, the model used by the TinyOS [13] operating system. Under this model, the application and operating system are compiled into a single executable binary file. Section 2 begins with a description of properties that can be used to characterize current network reprogramming protocols and an overview of common mechanisms that are used to build such protocols. A discussion of how these mechanisms are implemented in current protocols is found in Section 2.1. We evaluate and compare these existing protocols in Section 2.2.

As sensor networks pervade safety-critical environments, security and survivability will likely emerge as serious issues. We discuss these issues in Section 3 and propose new metrics for evaluating performance and survivability in Section 4. We conclude by looking at future research directions in Section 5.

2 Network (Re)programming Protocols

A generic network reprogramming protocol might operate as follows. Initially, all of the sensor nodes in a network are running the same version of some distributed application. This *primary application* defines the main functionality of a node, such as data collection, processing, and communication patterns. We differentiate between the primary application and the network reprogramming protocol, which operates as a *secondary service* alongside the primary application. At some point during the operation of the network, a new application image becomes available from an *originating source*. The originating source could be a base station, or a sensor node recently introduced into the network. We will refer to the set of data comprising a new application version as a *data object*. A data object is most commonly a monolithic application image, but could be any large piece of data required to upgrade an application. When two or more nodes have different versions of a data object, we say that they are *inconsistent*.

The originating source announces the availability of the new data object, and its neighboring nodes become aware of an inconsistency. The data object is then distributed from the originating source to each inconsistent neighbor. These neighbors are then *eligible* to become sources themselves, and the process is repeated until no inconsistencies remain in the network. The point at which a node is eligible to become a source varies with the protocol[1]. A data object is *pending* while it is in the process of being distributed. After the data object has been fully received by a node, it is considered *complete*. A reboot is typically required to load and execute the new application image, after which the completed data object becomes the *current* data object.

Properties. Various properties have been proposed to characterize network reprogramming protocols.

- *Reliability.* Unlike some sensor-network applications where packet losses are tolerable, a data object must be received in its entirety in order for the data to be usable. Since network reprogramming protocols typically operate in an inherently lossy network, they must sustain packet loss to provide complete program images.
- *Consistency.* In order to ensure that all distributed nodes are network state, the entire data object should *eventually* reach all of the nodes in the network. Nodes entering the network during or after an upgrade should still be ultimately able to receive the latest version of a data object.
- *Small Memory Footprint.* The update protocol should itself have a small memory footprint so as not to significantly restrict the amount of program space and RAM available to applications on the sensor node.
- *Energy Efficient.* Because energy is often a limited resource in sensor networks, an update protocol should minimize its energy usage so that it does not significantly affect network lifetime.
- *Minimize Disruptions.* The update protocol itself is a secondary service. As such, its operation should not significantly disrupt the functions of any primary application.
- *Fault Tolerance.* Network programming protocols should be tolerant to node failures and node additions. Failed and incoming nodes should not adversely affect the propagation of data objects to established live nodes.

These properties can be classified as *required* or *desired*. *Required* properties are those that define the correctness of a given protocol. *Desired* properties are not necessary for correctness, but enhance the usefulness and practicality of a protocol [23]. Generally, *reliability* and *consistency* are required, while the other properties are desirable. Since certain desirable properties are at odds with others, existing protocols make various tradeoffs between them. A discussion of the tradeoffs made by specific protocols is found in Section 2.2.

Components. Existing update protocols are typically comprised of a few core components that provide basic functionality. These core components can be summarized as follows.

- *Preparation Mechanism.* Since data objects are often larger than a single packet, they must be prepared for distribution before being injected into a network. Some protocols also require that certain initialization steps be performed among nodes before the dissemination process begins.
- *Dissemination Mechanism.* The dissemination mechanism allows nodes to learn of inconsistencies among sensor nodes and enables data propagation across multiple hops.

[1]See the description of *pipelining*, later in this section.

- *Reliability Mechanism.* An update is not useful unless it is received in its entirety. Since sensor networks often run over characteristically lossy wireless links, it is necessary to include a reliability mechanism to ensure that lost packets are recovered and all of the data is received.

In addition to the core components, different optimizations can be employed to increase performance in accordance with the specific design goals of a particular protocol. A few of the more common optimizations are:

- *Message Suppression.* Since all of the sensor nodes in the network are involved in disseminating the same data object, duplicate control and data messages may be sent by multiple nodes. Duplicate messages can lead to poor performance due to collisions and wasted energy. Various protocols take advantage of the broadcast medium to suppress redundant messages.
- *Sender Selection.* Often, more than one node is capable of forwarding data in any given neighborhood. Sender-selection mechanisms limit the number of active senders in a neighborhood.
- *Pipelining.* Allowing nodes to forward data before an object is complete propagates data in a *pipelined* fashion across the network, which reduces dissemination latency[2].

2.1 Overview of Existing Protocols

2.1.1 Reijers et al.

Preparation. In [21], the authors propose a `diff`-like[3] scheme to reduce the amount of data that needs to be transmitted. Instead of disseminating an entire program image, the authors propose creating an *edit script* by comparing the current data object with the pending data object. The edit script consists of commands such as *copy*, *insert* and *repair*. The edit script is divided into packets, such that each packet describes a certain number of program bytes and can be processed independently of other packets.

Dissemination. While the preparation mechanism is well developed, [21] describes only a simple point-to-point distribution mechanism.

Reliability. Each packet contains the starting address of the block of program bytes it describes. Nodes use this address information to determine where edit-script data is missing. Since each packet contains all of the information required to build an arbitrary number of bytes and can be processed independently, packets can arrive in any order.

2.1.2 Multi-hop Over-the-Air Programming (MOAP)

Preparation. MOAP [23] divides a data object into packets, and these packets are distributed through the network. Once received, packets are placed in stable storage until the entire update has been completed.

[2]We define *latency* to be the time from when the dissemination process begins at the originating source, until all live nodes have received the complete data object

[3]`diff` is a unix utility that identifies differences between two files. See http://www.gnu.org/software/diffutils/

Dissemination. MOAP uses a mechanism called *Ripple* to distribute code through the network on a ”neighborhood-by-neighborhood” basis. Ripple uses a publish-subscribe mechanism similar to Directed Diffusion [7], where sources advertise updated code images to which other nodes register interest. After nodes receive a full image, they become publishers and may propagate the image to other nodes out of range of the original source. This process is applied iteratively until the update has propagated across the network.

Reliability. MOAP places the responsibility for detecting packet loss at the receiver and uses a sliding-window approach to keep track of lost packets. When a missing packet is detected, the receiver sends a unicast retransmission request. If the source does not respond within a certain amount of time, the receiver broadcasts a retransmission request to which all nodes within range reply. This allows the receiver to choose a new source in case the original source fails. Duplicate requests arriving at a source within a given time period are suppressed.

2.1.3 Deluge

Preparation. Deluge [6] introduced the notion of *pages* as a unit of transfer. An update is broken up into a number of fixed-size packets, which are then grouped into segments (or *pages*) of N packets each. Each page can then be compared against a previous version to determine whether it has changed. This metadata is encapsulated in an *age vector*, which allows nodes to determine when a page changed, and whether they need to request it.

Dissemination. Deluge employs a three-phase *advertisement-request-data* handshake to propagate updates between nodes.

In the *advertisement* phase, all nodes broadcast summaries that consist of a version number and the number of the highest page available for transfer. If a node learns of inconsistencies (i.e. a neighbor advertises a lower version number), it transmits an object profile containing the age vector and a version number. Deluge borrows techniques from Trickle [14] to suppress redundant control messages, while dynamic advertisement intervals allow adjustment of the advertisement rate during upgrades versus during steady state operation. When a node hears an advertisement for a page it does not have, it will transition to the *request* phase unless it has recently overheard (1) a previous request for, or data packet of, a lower numbered page or (2) a data packet of the desired page.

During the *request* phase, a node sends requests containing a bitmap of the packets it needs to complete a given page. A node will return to the advertisement phase once it receives all of packets that constitute the page, or the reception rate drops below a certain threshold.

A node enters the *data* phase when it receives a request for data. A node in the data phase is responsible for transmitting packets from a particular page. The node adds requested packets to a set, and transmits them in round-robin order. Packets requested in subsequent request messages are added to the set as well. After a packet is transmitted, it is removed from the set and the node returns to the advertisement phase when the set is empty.

Reliability. Cyclic redundancy checks (CRCs) at the packet- and page-levels help protect against data corruption. If a CRC fails, all of the data represented by it must be retransmitted.

Retransmissions are enabled by a bitmap that tracks packets received for a given page. As in [23], receivers are responsible for detecting their own losses and retransmission requests function as ”selective negative acknowledgments” by including the page bitmap in messages during the *request* phase.

Optimizations. Deluge introduced the notion of "spatial multiplexing", or *pipelining*, whereby nodes are allowed to transmit page n as soon as pages $0 \ldots n-1$ are complete as opposed to waiting for an entire data object to be completed before forwarding data. Deluge also attempts to reduce the transfer of redundant data by proposing the use of an age vector, which allows nodes to know when pages have last changed. Pages that have not changed since the last update are not requested by nodes in the *request* phase.

2.1.4 Multi-hop Network Programming (MNP)

Preparation. As in Deluge, data objects are divided into fixed-size segments each containing an equal number of fixed-size packets.

Dissemination. MNP [12] is designed as a state machine that incorporates the dissemination and reliability mechanisms. The state machine consists of 5 basic functional states (`IDLE`, `DOWNLOAD`, `ADVERTISE`, `FORWARD` and `SLEEP`), as well as an ephemeral `FAIL` state from which nodes transition immediately back to `IDLE`. The dissemination mechanism also introduces a *parent / child* relationship between senders (the parents) and receivers (the children).

Nodes in the `IDLE` state listen for *advertisement* messages and respond with unicast *download requests*. Nodes in the `ADVERTISE` state send *advertisements* with the ID of the highest segment that they have available for transmission. If a *start download* message is received while a node is in either `IDLE` or `ADVERTISE`, the node sets its parent to be the source of the message and transitions to `DOWNLOAD`. Nodes in the `DOWNLOAD` state receive packets and write the data directly to stable storage, instead of storing the entire segment in a RAM buffer. If all packets have been received correctly when the node receives an *end download* message, it transitions to `ADVERTISE`. Otherwise, it transitions to `FAIL`.

Nodes in the `ADVERTISE` state listen for *download requests*. When an advertising node decides to become a sender, it broadcasts a *start download* message and transitions to the `FORWARD` state. In this state, the node broadcasts requested packets from a particular segment. Once all of the requested packets have been sent, the node broadcasts an *end download* message and goes to `SLEEP` for a predetermined length of time.

Reliability. MNP's reliability mechanism is very similar to Deluge's. MNP places the responsibility for detecting packet loss on the receiver and retains in memory a bitmap of the current segment. The bitmap tracks which packets in the current segment are missing and is included in download requests; when a node receives a download request, it performs the union of the new request with a set of outstanding requested packets to determine which packets it needs to forward.

MNP also provides an optional *query/update* phase, which adds two new states: `QUERY` and `UPDATE`. After a sender in the `FORWARD` state has finished forwarding all of its requested packets, it will enter the `QUERY` state instead of going to `SLEEP`. The sender broadcasts a *query* message, to which its children respond with unicast NACKs. This allows children to recover a small number of packets more quickly than immediately transitioning to `FAIL` and waiting for a new advertisement.

To handle failed senders, nodes in the `DOWNLOAD` and `UPDATE` states use timeouts to determine whether a parent is still alive. If a child does not hear from its parent for a predetermined period of time, the child node will transition to `FAIL`. Note that the parent-child relationship is stricly one-way; parents have no knowledge of their children.

Optimizations. MNP introduced a *sender-selection* mechanism that attempts to explicitly limit the number of nodes that are transmitting data in a particular broadcast neighborhood. Eligible source nodes (ie. nodes in the `ADVERTISE` state) compete with each other to become senders; nodes that lose go to `SLEEP` for a predetermined period of time. Specifically, potential senders keep track of the number of *download requests* they have received, and the node with the highest number of requests wins. Potential senders announce their local request-count in *advertisement* messages. When a node responds to an *advertisement* with a *download request*, it includes the request count of the advertising node in the request. This allows nodes to learn the request counts of other potential senders up to two hops away. Additionally, MNP takes advantage of a pipelining mechanism similar to that used by Deluge.

2.1.5 Infuse

Preparation. Infuse [11] splits the data object into fixed-size packets prior to dissemination. Infuse also requires a Time Division Multiple Access (TDMA) schedule to be known prior to dissemination.

Dissemination. Infuse introduces a TDMA-based approach to code dissemination. Infuse assumes TDMA slots are assigned by the base station and that each time slot is long enough to transmit a single packet. Each node groups its neighbors into *predecessors* and *successors*. Predecessors of a node are those neighbors who forward a majority of packets before the node. Successors of a node are neighbors who forward a majority of packets after the node. Nodes listen during their predecessor's time slots to receive packets, and forward packets during their own time slots.

Reliability. Infuse uses *go-back-n* with implicit acknowledgments, placing the responsibility for detecting and recovering from lost packets on the sender. After forwarding a packet, the sender listens in its successors' time slots. If a successor does not forward the packet, it is assumed to be lost and retransmitted in accordance with the go-back-n algorithm. If a node does not hear from a successor after a certain number of retransmission attempts, it assumes the successor has failed and no longer listens for implicit acknowledgments from it.

Optimizations. Infuse suggests a notion of *preferred predecessors* to reduce the number of predecessors receiving implicit acknowledgments from a single node. Nodes choose a predecessor from whom they prefer to recover lost packets, and piggyback this information on forwarded data packets. Once the preferred status of a node is known, non-preferred predecessors will only listen for implicit acknowledgments with a certain probability (to allow recovery from a failed preferred predecessor). The preferred predecessor continues to listen for implicit acknowledgments during its successors' slot, although the authors note that message receptions can be further reduced by having the preferred predecessor listen probabilistically as well.

2.1.6 Sprinkler

Preparation. Sprinkler [18] splits the data object into fixed-sized packets prior to dissemination. Clustering and coloring algorithms must be performed over the network to compute a connected dominating set (CDS) and TDMA schedule. Sprinkler assumes that nodes are grouped into clusters, and that a single node from each cluster is designated as the cluster head. A connected dominating set (CDS) is computed over cluster heads. Members of the CDS are designated as senders, and are the only nodes responsible for forwarding data. A D-2 coloring algorithm is then run over the CDS nodes to determine a TDMA transmission schedule.

Protocol	*Dissemination*	*Data Granularity*	*Pipelining*	*Recovery*
Reijers et. al.	point to point	packet	no	N/A
MOAP	publish/subscribe	packet	no	sliding window + unicast NACK
Deluge	three phase (adv-req-data)	page	yes	bitmap + unicast NACK
MNP	three phase (adv-req-data)	page	yes	bitmap + unicast NACK
Infuse	TDMA - parent / child	packet	yes	go-back-n + implicit ACK
Sprinkler	TDMA - parent / child	packet	yes	implicit & unicast NACK

Table 1: A summary of the protocols discussed so far.

Dissemination & Reliability. The Sprinkler dissemination mechanism has 2 phases, *streaming* and *recovery*, each of which employ reliability mechanisms. Each node in the network selects a neighboring node as its parent. Parents are responsible for recovering lost packets at their children; however, the mechanism for doing so differs according to the phase.

During the *streaming* phase, CDS nodes forward newly heard data packets during their respective TDMA slots. Negative acknowledgments (NACK) and parent IDs are piggybacked on these data packets. When a parent receives a NACK, it retransmits the corresponding packet in its next slot.

At the end of the streaming phase, all of the CDS nodes have a complete data object, but non-CDS nodes may still be missing packets. During the *recovery* phase, non-CDS nodes unicast requests for missing packets to their parents. These requests continue to be sent at specific intervals until each non-CDS node has a complete data object. In this phase, messages can be sent by both CDS (recovery data packets) and non-CDS (recovery data requests) nodes, so the previously computed TDMA schedule cannot be used. Instead, Sprinkler specifies that a simple RTS/CTS mechanism be used for coordination.

Optimizations. Sprinkler introduced local algorithms to compute a connected dominating set (CDS) among clusters of nodes, and D-2 vertex coloring among cluster heads. These algorithms are used to determine the set of senders and their transmission schedules.

2.2 Evaluation & Comparison of Existing Protocols

One way to improve energy efficiency and end-to-end dissemination latency is to reduce the amount of data that needs to be transferred. Deluge attempts to reduce the size of data objects though the age vector. The age vector allows for nodes which may be multiple versions behind the pending data object. However, even small source-code changes can cause large shifts in binary data that span multiple pages. While [21] is very effective at reducing the amount of data to be transferred, it takes advantage of instruction-set details to build the edit script, and so, is architecture dependent. A platform independent approach based on the `rsync` algorithm is presented in [8]. This approach effectively handles code shifts and small binary changes, but presents a negligible speedup with large code changes. Both [21] and [8] assume nodes to be in a consistent state, and can only update nodes that are a single version behind the pending data object.

An important aspect of reliability mechanisms is keeping track of which packets a node has received correctly, and which packets have been lost. A simple solution would be to use a bitmap of all of the packets in a data object. However, this would require an inordinate amount of state to be stored in memory. Moreover, since dynamic memory allocation is typically not well supported on sensor nodes, it would be necessary to statically allocate a bitmap that is large enough for the largest possible data object. MOAP handles this through the use of a fixed-size sliding window. By splitting a data object into a number of fixed size pages, Deluge and MNP can use a small bitmap per page as opposed to a large bitmap per data object, thereby reducing the memory required to store recovery state. Also, since pages have a fixed number of packets, the page bitmap can be statically allocated without wasted space.

A page-based approach also allows Deluge and MNP to take advantage of pipelining to reduce latency. On the other hand, MOAP takes the position that reducing latency is less important than minimizing energy and memory usage, and makes tradeoffs accordingly. Not allowing nodes to become senders until they have received a complete update precludes the use of pipelining to decrease latency. However, this reduces the protocol's program complexity. Using an initial unicast retransmission request prevents duplicate replies without any additional suppression mechanisms. The tradeoff is a reduction in program complexity (no additional suppression code) and energy usage (no duplicate replies) for increased latency if the original source should fail.

Due to the hidden terminal problem[4], Deluge displays dynamic propagation behavior where data propagates faster along the edges of a network than across a diagonal. MNP's sender-selection mechanism effectively accounts for hidden terminals, so the dynamic propagation seen by Deluge is absent. The use of a `SLEEP` state in MNP allows nodes to avoid idle listening, thereby conserving energy.

Since Infuse and Sprinkler use a TDMA approach, the transmission schedule is deterministic and the hidden terminal problem does not apply. Hence, dynamic propagation effects are absent here as well. The use of implicit acknowledgments in Infuse reduces control-traffic overhead. Using TDMA also allows for reduced idle listening, since nodes only need to listen in their neighbor's slots. While the use of preferred predecessors reduces the number of slots that a node must listen in, it introduces probabilistic recovery guarantees. Also, it may be difficult for nodes to determine their predecessors and successors in non-uniform networks.

While [18] introduced novel ways to optimize the set of senders, Sprinkler's approach has a number of drawbacks. First, energy usage is not distributed evenly through the network. Because the set of senders (nodes in the CDS) is static, these nodes will be depleted of energy much faster than non-CDS nodes. Also, location information is required to compute the CDS and nodes must remain stationary. The overhead required to perform localization is not discussed in [18].

While it is desirable that an update protocol not significantly impact the performance of the primary application, this property is not well evaluated in the protocols discussed so far. One way that protocols attempt to minimize disruptions is by minimizing dissemination latency, under the assumption that the sooner that dissemination completes, the sooner the nodes can get back to work. Some protocols [6, 12] also leverage a dynamic advertisement rate to allow for fast propagation (shorter advertisement period) during the upgrade process and reduced overhead (longer advertisement period) once the network reaches consistency.

3 Security

While network reprogramming has emerged as a necessary mechanism for maintaining sensor networks, current protocols have not been designed with security in mind. Consequently, it may be possible for an adversary to subvert the operation of such protocols to prevent the dissemination of needed updates, waste network resources, disrupt the operation of current applications or execute arbitrary malicious code. Moreover, an adversary may be able to leverage for malicious purposes the same mechanisms that contribute to the efficiency of current protocols. For example, pipelining allows correct nodes to quickly propagate

[4]The hidden terminal problem refers to the situation where, given three nodes a, b and c, $a \leftrightarrow b$ and $b \leftrightarrow c$ can communicate but a and c are out of range from each other. Thus, if a and c can not sense the other is sending, they may both try transmitting to b at the same time, where the packets will collide and be lost.

legitimate data objects across a network. An adversary that is able to hijack this mechanism could use it to quickly propagate *malicious* data objects across the network instead.

3.1 Attacks

A discussion of attacks on sensor network routing protocols is presented in [10]. Many of these attacks are applicable to network reprogramming protocols as well. However, network reprogramming protocols have several distinguishing characteristics.

- While routing protocols typically focus on finding a path through the network to a specific destination, *every* node is a target in network reprogramming. Thus, the operation of a network reprogramming protocol affects every single node in the network.
- Network reprogramming protocols affect the actual execution state of network nodes. A vulnerability in a network reprogramming protocol could compromise the execution state of a node.
- In many protocols, there is an implicit ordering whereby messages have to be received (i.e. pipelining). Disrupting this order could lead to the failure of the protocol.

Therefore, the nature of network reprogramming protocols makes the network vulnerable to new types of attacks and could allow for serious consequences in the event of an attack. Some of the attacks that we envisage are described below. While it is difficult to quantify the severity of these potential attacks, these issues should not be overlooked when designing secure network programming protocols.

Malicious Data Injection. Some form of advertisement is necessary for nodes to learn of inconsistencies in the network, and to determine a need to upgrade. Thus, the very nature of network reprogramming makes current protocols vulnerable to their most serious attack. If a malicious node is able to inject packets into a network, it may also be able to induce other nodes to download an arbitrary data object by advertising the availability of a bogus update. This not only wastes network resources, but it also may allow for the execution arbitrary code.

Deluge, for example, is designed so that any node can initiate an upgrade. If an adversary sends a malicious object profile containing a very high version number, other nodes will request the malicious code object. This code object could disable future reprogramming attempts and shut down the network, or simply retask it for the adversary's gain. Deluge's dual CRC checks do not provide any protection against this attack; they simply ensure that the malicious data object is not corrupted. Once the data object has propagated, the adversary could broadcast a reboot message to have network nodes load and execute the code. The malicious code could either place the network under the adversary's control, or turn off network reprogramming altogether, necessitating a manual maintenance to correct the affected nodes.

The advertisement and publishing mechanisms employed by MNP and MOAP can be likewise subverted. While [11] and [18] do not specify how nodes learn of inconsistencies, similar attacks could be possible.

Timing/State Attacks. In protocols that rely on actions taken by other nodes to determine local state and timing, malicious nodes can broadcast fake information to disrupt the operation of other nodes. Deluge allows nodes to dynamically change their advertisement period during steady-state operation versus during an active update. This is accomplished by listening for inconsistencies in the network. If a malicious node falsely advertises an inconsistency, it will force neighboring nodes to send advertisements rapidly in an attempt to update the malicious node. This rapid advertisement wastes energy, and could have an adverse impact on the operation of the primary application.

Deluge is also designed so that a node will not request a needed page if the transfer of a lower numbered page is in progress. This is intended to prioritize lower numbered pages in order to limit interference caused by the transfer of different pages in a single neighborhood. However, this restriction could be abused by a node requesting low numbered pages with a high enough frequency that neighboring nodes never enter the request phase. These nodes will still be able to save overheard data packets, but the inability to request data will break the chain of propagation in certain network topologies.

Sybil Attacks. In a Sybil attack [3], a single node presents multiple identities to its neighbors. In the context of network reprogramming protocols, this could allow attacks against parent/child relationships. For example, Infuse allows a node to identify predecessors and successors by determining which nodes forward a majority of packets before or after it. A malicious node could be pose as both a successor and a predecessor to another node by using different identities to forward a packet at different times.

Sinkhole Attacks. Sinkhole attacks allow malicious nodes to attract a disproportionate amount of traffic from neighboring nodes. This could allow an adversary to propagate malicious code more quickly, or to block the progress of a legitimate update.

An effective sinkhole attack is enabled by the sender-selection mechanism used in MNP. By advertising with an artificially high request counter, a malicious node is able to "cheat" in the sender-selection competition. All of the potential receivers in the malicious node's neighborhood will assign it as their parent, putting other eligible senders to sleep. This causes a denial of service at other potential sender nodes and allows for a *selective forwarding* attack where the malicious node refuses to forward data to its children, preventing the propagation of an update to that particular neighborhood.

3.2 Current Approaches

To the best of our knowledge, there has been little work directly addressing security issues in network programming protocols. However, security research in other areas (e.g. code attestation, authenticated broadcast, secure routing, and cryptography) is relevant to this field.

Remote Verification. Software attestation techniques such as SWATT [22] exist to verify the program image and memory contents of embedded devices. However, SWATT does not prevent attacks; tampering can only be detected after it has already occurred. By the time an attack has been detected, precious energy would have been wasted propagating and loading a malicious image. It may then be necessary to reprogram each node by hand, which could be infeasible in large or deeply embedded networks.

Authenticated Broadcast. TESLA [19], and its WSN counterpart μTESLA [20], are authenticated broadcast protocols. μTESLA uses symmetric primitives with delayed key disclosure to provide authenticated broadcast. However, the bootstrapping process requires the sender to unicast keychain commitments to each receiver. This limits μTESLAs scalability in large networks. On the other hand, TESLA uses digital signatures to broadcast keychain commitments. Both schemes require loose time synchronization across the network, which is incompatible with current dissemination protocols which do not place time bounds on completion.

Symmetric Key Cryptography. Symmetric-key protocols like TinySec [9] and SNEP [20] provide confidentiality, message integrity, and access control. These schemes protect againt eavesdropping, message tampering, and message injection. However, the use of these protocols alone does not provide sufficient assurance against compromised nodes.

Symmetric link-layer security protocols protect against packet injection by arbitrary nodes by using a message authentication code (MAC) on each packet to authenticate traffic as coming from a legitimate source. If a node is physically compromised, its cryptographic material is potentially compromised as well; thus, an adversary can use the compromised node to inject packets into the network that pass the MAC, but that are nevertheless malicious. In a network reprogramming setting, these seemingly legitimate nodes can propagate malicious updates. If a single node is compromised, an adversary could leverage its cryptographic material to broadcast "authentic" updates that are propagated through the network. Hence, a *single* compromised node can effectively allow for the remote execution of arbitrary code on *every* node, giving the adversary complete control of the network.

Clearly, any secure network reprogramming protocol must be robust against node compromise. Pairwise key schemes [16, 24] attempt to mitigate the threat of compromised nodes by establishing shared keys that are held by only a small subset of network nodes. Unfortunately, this strategy is not sufficiently effective for network reprogramming protocols. A compromised node might propagate updates to nodes that it is paired with, which, in turn, might forward the updates to nodes they are paired with, and so on. As long as the network is connected, the malicious update can eventually reach all nodes.

Asymmetric Key Cryptography Digital signatures could be useful for verifying the source of a code object, but asymmetric cryptography has long been thought to be impractical on severely constrained sensor nodes. However, recent work [25, 4, 17, 15, 5] suggests that, with careful implementation and judicious use, public-key cryptography can be quite feasible on sensor nodes.

TinyPK [25] implements the RSA cryptosystem for TinyOS. The authors propose securing the XNP module [2], the original single-hop TinyOS network-reprogramming protocol, by running TinyPK verification after initializing the XNP process. However, the verification protocol only seems to verify that the broadcaster is an authorized member of the network. This does not address the issue of an already verified node malfunctioning or being compromised and subsequently running the update protocol.

Also, it has been shown that elliptic-curve cryptography (ECC) provides substantial performance gains over RSA on constrained platforms [5]. ECC versions of cryptographic functions are becoming more common, e.g., ECDSA [1] represents the elliptic curve version of DSA. The authors of EccM [17] provide a Diffie-Helman (DH) key-distribution mechanism based on ECDLP, but do not address digital signatures. The developers of Sizzle [4] successfully implemented SSL using ECC on the MICA2 andTelos mote platforms. Their highly optimized code can complete a full SSL handshake (which includes ECDSA and ECDH operations) in under 4 seconds. While this work shows the viability of ECC in highly resource-constrained systems, a full SSL handshake is neither necessary nor appropriate to secure network reprogramming protocols. SSL provides one-to-one semantics, while network reprogramming is by nature one-to-many. The authors of [15] provide an ECC implementation for TinyOS that includes an ECDSA module. This module is optimized for the MICAz platform, and is able to verify a digital signature in under 15 seconds.

4 Metrics for Evaluation

Most network reprogramming protocols to date have been evaluated in terms propagation latency and energy usage. To judge their suitability for safety- or time-critical environments, additional evaluation metrics are needed.

4.1 Performance

The network reprogramming protocol is a utility service, rather than a primary application. The protocol may need to operate concurrently with the primary application, or may block the primary application until a data object's transmission is completed. Simply measuring the propagation latency is not sufficient to quantify the impact of network reprogramming on an application. If the protocol operates concurrently, it may introduce jitter into the timing of periodic tasks within the application. If an application is time-critical, this could cause missed deadlines.

While some protocols have recognized that their impact on the application should be small, most empirical evaluation has taken place under the assumption that no other applications are running. To quantify the effect of a reprogramming protocol on the primary application (and vice versa), we believe application semantics should play a role in evaluation.

- In time-critical applications, missed deadlines should be quantified under operation with and without network programming. The jitter of periodic tasks should be measured as well.
- Non-time-critical applications should still be able to make progress. The throughput of such applications could be measured as an indicator of how much impact network reprogramming has on their operation.

4.2 Survivability

While some reprogramming protocols have been evaluated under a fail-stop fault model, no reprogramming protocols have taken misbehaving or malicious nodes into account. The possibility exists that misbehaving or malicious nodes could seriously disrupt the operation of a network reprogramming protocol. As mentioned in the previous section, this could, in turn, seriously disrupt the operation of the primary application as well. For example, since current network reprogramming protocols do not place any bounds on propagation time, a misbehaving node could cause denial of service at the application level.

- Nodes should not be assumed to behave correctly or fail stop. Performance metrics should be evaluated in the presence of incorrect nodes.
- Since malicious data injection wastes network resources as well as allows for the possibility of arbitrary code execution, metrics are needed to quantify the severity of this type of attack. Possible metrics include the number of hops that an arbitrary data object is allowed to propagate before it is halted, and the amount of arbitrary data that a single node can be coerced into downloading.
- In the presence of incorrect nodes, the eventual consistency guarantee of some protocols may not be met. Consistency should be quantified by the percentage of correct nodes receiving a correct data object in the presence of incorrect nodes.

Actually, it might be desirable to modify the assumption of eventual consistency. Instead of guaranteeing the eventual dissemination of a data object to all of the nodes, secure protocols could instead guarantee *correct state* at all nodes. Either all of the nodes will download and begin execution with a correct pending data object or all of the nodes will continue executing with a correct current data object. Nodes should not execute arbitrary data objects, and they should not indefinitely block on pending objects.

5 Looking Ahead

It may be possible to prevent certain attacks by incorporating new security mechanisms into current protocols. However, in order to have truly secure network reprogramming that is robust against a range of attacks, it will likely be necessary to design new protocols with security in mind from the beginning. These new protocols should not only be evaluated using traditional metrics like propagation latency and energy usage, but also using the additional metrics proposed in Section 4.

References

[1] American National Standards Institute. ANSI X9.62-1998: Public key cryptography for the financial services industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), 1998.

[2] Crossbow Technology Inc. Mote In-Network Programming User Reference, 2003.

[3] John R. Douceur. The Sybil Attack. In *First International Workshop on Peer-to-peer Systems (IPTPS '02)*, March 2002.

[4] Vipul Gupta, Matthew Millard, Stephen Fung, Yu Zhu, Nils Gura, Hans Eberle, and Sheueling Chang Shantz. Sizzle: A Standards-Based End-to-End Security Architecture for the Embedded Internet (Best Paper). In *PERCOM '05*, pages 247–256, Washington, DC, USA, 2005. IEEE Computer Society.

[5] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In *International Workshop on Cryptographic Hardware and Embedded Systems*, volume 3156, pages 119–132, August 2004.

[6] Jonathan W. Hui and David Culler. The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In *SenSys '04: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, pages 81–94, New York, NY, USA, 2004. ACM Press.

[7] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *MobiCom '00: Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, pages 56–67, New York, NY, USA, 2000. ACM Press.

[8] Jain Jeong and David E. Culler. 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*, 2004.

[9] Chris Karlof, Naveen Sastry, and David Wagner. TinySec: A Link Layer Security Architecture for Wireless Sensor Networks. In *SenSys '04: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, pages 162–175, New York, NY, USA, 2004. ACM Press.

[10] Chris Karlof and David Wagner. Secure Routing in Wireless Sensor Networks: Attacks and Countermeasures. *Elsevier's AdHoc Networks Journal, Special Issue on Sensor Network Applications and Protocols*, 1(2–3):293–315, September 2003.

[11] Sandeep S. Kulkarni and Mahesh Arumugam. Infuse: A TDMA Based Data Dissemination Protocol for Sensor Networks. Technical Report MSU-CSE-04-46, Department of Computer Science, Michigan State University, East Lansing, Michigan, 2004.

[12] Sandeep S. Kulkarni and Limin Wang. MNP: Multihop Network Programming for Sensor Networks. In *International Conference on Distributed Computing Systems*, pages 7–16, Washington, DC, USA, June 2005. IEEE Computer Society.

[13] Philip Levis, Sam Madden, David Gay, Joseph Polastre, Robert Szewczyk, Alec Woo, Eric Brewer, and David Culler. The Emergence of Networking Abstractions and Techniques in TinyOS. In *First Symposium on Networked System Design and Implementation (NSDI '04)*, pages 1–14, San Francisco, California, USA, 2004.

[14] Philip Levis, Neil Patel, David E. Culler, and Scott Shenker. Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In *First Symposium on Networked System Design and Implementation (NSDI '04)*, pages 15–28, 2004.

[15] A. Liu and P. Ning. TinyECC: Elliptic curve cryptography for sensor networks, September 2005.

[16] D. Liu, P. Ning, and R. Li. Establishing pairwise keys in distributed sensor networks. *ACM Transactions on Information and System Security*, 8(1):41–77, February 2005.

[17] David J. Malan, Matt Welsh, and Michael D. Smith. A Public-Key Infrastructure for Key Distribution in TinyOS Based on Elliptic Curve Cryptography. In *First IEEE International Conference on Sensor and Ad Hoc Comminications and Networks*, pages 71–81, October 2004.

[18] Vinayak Naik, Anish Arora, and Prasun Sinha. Sprinkler: A Reliable and Scalable Data Dissemination Service for Wireless Embedded Devices. Technical Report ExScal-OSU-EN04-2005-05-11, ExScal Note Series, 2005.

[19] Adrian Perrig, Ran Canetti, J. D. Tygar, and Dawn Song. The TESLA Broadcast Authentication Protocol. *RSA CryptoBytes*, 5(Summer), 2002.

[20] Adrian Perrig, Robert Szewczyk, J. D. Tygar, Victor Wen, and David E. Culler. SPINS: Security Protocols for Sensor Networks. *Wireless Networks*, 8(5):521–534, 2002.

[21] Niels Reijers and Koen Langendoen. Efficient Code Distribution in Wireless Sensor Networks. In *WSNA '03: Proceedings of the 2nd ACM International Conference on Wireless Sensor Networks and Applications*, pages 60–67, New York, NY, USA, 2003. ACM Press.

[22] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep K. Khosla. SWATT: SoftWare-based ATTestation for Embedded Devices. In *IEEE Symposium on Security and Privacy*, pages 272–282, 2004.

[23] Thanos Stathopoulos, John Heidemann, and Deborah Estrin. 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.

[24] A. Wacker, M. Knoll, T. Hieber, and K. Rothermel. A new approach for establishing pairwise keys for securing wireless sensor networks. In *ACM International Conference on Embedded Networked Sensor Systems*, pages 27–38, November 2005.

[25] Ronald Watro, Derrick Kong, Sue fen Cuti, Charles Gardiner, Charles Lynn, and Peter Kruus. TinyPK: Securing Sensor Networks with Public Key Technology. In *SASN '04: Proceedings of the 2nd ACM*

Workshop on Security of Ad Hoc and Sensor Networks, pages 59–64, New York, NY, USA, 2004. ACM Press.