A Remote Code Update Mechanism for Wireless Sensor Networks

Thanos Stathopoulos† John Heidemann‡ Deborah Estrin†

CENS Technical Report # 30
Center for Embedded Networked Sensing
† UCLA, Department of Computer Science
‡ USC, Information Sciences Institute
{thanos@cs.ucla.edu, johnh@isi.edu, destrin@cs.ucla.edu}

Abstract

Wireless sensor networks consist of collections of small, low-power nodes that interface or interact with the physical environment. The ability to add new functionality or perform software maintenance without having to physically reach each individual node is already an essential service, even at the limited scale at which current sensor networks are deployed. TinyOS supports single-hop over-the-air reprogramming today, but the need to reprogram sensors in a multihop network will become particularly critical as sensor networks mature and move toward larger deployment sizes. In this paper we present Multihop Over-the-Air Programming (MOAP), a code distribution mechanism specifically targeted for Mica-2 Motes. We discuss and analyze the design goals, constraints, choices and optimizations focusing in particular on dissemination strategies and retransmission policies. We have implemented MOAP on Mica-2 motes and we evaluate that implementation using both emulation and testbed experiments. We show that our dissemination mechanism obtains a 60–90% performance improvement in terms of required transmissions compared to flooding. We also show that a very simple windowed retransmission tracking scheme is nearly as effective as arbitrary repairs and yet is much better suited to energy and memory constrained embedded systems.

1 Introduction

Recently, increasing research attention has been directed toward wireless sensor networks. Large numbers of small, inexpensive devices that integrate sensing, computation and communication will be monitoring environmental changes, water contamination, seismic activity, structural integrity of buildings etc. These devices are quite heterogeneous and can have multiple constraints such as limited CPU power (ranging from micro-controllers to basic microprocessors), narrow bandwidth (short range, low power wireless radios) and limited energy budget.

Once deployed, sensor nodes are expected to operate for extended periods of time, and without any human intervention. In fact, there are several deployment scenarios for which physically reaching all nodes is either impractical (e.g., nodes on treetops) or detrimental to the sensing process (e.g., nodes inside nests). In spite of these difficulties, there is a real need to add or upgrade the software running on those nodes, post-deployment. Sensor network users need remote programmability in order to add new functionality to the nodes—especially when the knowledge of the environment is not complete and predicting the complete set of actions a node has to perform before deployment is impossible. In that sense, remote programmability extends the usefulness of the network. It is also helpful in dealing with software maintenance and in-situ debugging. In addition, by removing the human from the programming loop, we expect to automate the process and thus support large network sizes. With large networks, a *multihop* dissemination mechanism is therefore needed.

A multihop code distribution mechanism can be considered as a special case of *reliable data dissemination*. The need to distribute a *large volume* of data to *all* the nodes in the network means that we cannot apply traditional approaches to the problem, such as those developed for Reliable Multicast [1]. At the same time, this application also includes a more strict notion of *reliability* than Internet real-time audio and video streaming [2], in that *everything* must be received. Finally, the specific nature of sensor networks requires the mechanism to operate using low-power, unreliable radios and very limited memory and storage. Based on the above, multihop code distribution adds to the exploration of the sensor network design space for *reliable communications*.

The TinyOS [3] developers anticipated the remote programming requirements and have included support for single-hop over the air programming since its release [4]. However, single-hop programmability has limited usage, especially as network sizes grow beyond the reach of a single radio. In this paper, we present *Multihop Over-the-Air Programming* (MOAP). MOAP is a code distribution mechanism that is specifically targeted for the Mica-2 mote platform [5]. We explore the design goals and questions related to building such a mechanism. In terms of resources, we focus on energy consumption, memory usage and latency. We analyze the design alternatives in areas such as dissemination protocols, retransmission policy and storage management.

Using emulation as well as an actual standalone mote implementation we show that MOAP results in 60–90% reduction in transmissions compared to flooding. This result represents one point in the design space that balances complexity with efficiency, using a neighborhood-by-neighborhood transport mechanism and a simple sliding window method of tracking retransmissions. It is also rather modest in terms of storage consumption, requiring approximately 700 bytes of RAM and 4.5 Kbytes of program memory.

2 Related Work

We looked in four main areas for related work: Code Distribution for Sensor Networks, Reliability protocols for Sensor Networks, Multicast Reliability, and Data Dissemination in Sensor Networks

2.1 Code Distribution in Sensor Networks

TinyOS has included In-Network Programming support for the Mica-2 motes since its 1.0 release [4]. The mechanism is single-hop only: the basestation (source) transmits code capsules to all nodes within a broadcast domain. After the entire image has been transmitted, the basestation polls each node for missing capsules. Nodes scan the contents of EEPROM to find gaps and then reply with NACKs if necessary. The basestation unicasts missing capsules as required.

In order to reduce the energy consumption of code distribution, the authors of [6] propose a *difference-based* mechanism: instead of sending the entire image, only the differences between the new and old code are sent. By using optimizations like address shifts, padding and address patching, the mechanism is able to substantially reduce the size of the updates, especially when the differences are small. This makes it an ideal choice for incremental updates, or bug fixes. Even though the difference creation mechanism is well developed, at the moment, the distribution part only uses point-to-point communications.

LOBcast [7] is another code distribution mechanism that targets Mica-2 motes which includes support for multiple concurrent versions. Nodes keep a *catalog* of objects available for download, which is periodically updated via advertisements. Applications can request content from the catalog (e.g. updated versions of code). LOBcast is using methods similar to RMST, in that a node first requests content (and repairs) from its immediate neighborhood but moves toward the source (called sensor access point in LOBcast), if local retrieval fails. In addition, it includes duplicate suppression mechanisms similar to SRM.

A completely different approach to code distribution is presented in [8]. Mate, and its successor, Bombilla, is a stack-based *Virtual Machine*. It includes three execution contexts, with two stacks per context. Bombilla programs consist of *capsules*, with each capsule having up 24 instructions. The current implementation allows up to 8 capsules to be present. Bombilla includes special instructions that can *forward* capsules to other nodes. Nodes will install a capsule that has a newer version number than the one currently used. By repeatedly using forwarding instructions, a new capsule can reach the entire network.

2.2 Reliability in Sensor Networks

Reliability is an integral part of MOAP, since code images must be delivered in their entirety to nodes. Recently, several sensor network reliability protocols have been proposed. PSFQ [9](Pump Slowly, Fetch Quickly) is a hop-by-hop reliable transport protocol. It is characterized by two phases: a low-rate data distribution phase (pump) and a high-rate, NACK-based error recovery phase. The data distribution phase is based on a controlled flooding algorithm, using a data cache to suppress duplicates. Like MOAP, generated NACKs are strictly local (single-hop). PSFQ uses broadcast repair requests and suppression mechanisms similar to SRM to reduce the number of duplicates. Even though those mechanisms are effective, they can induce a substantial overhead in terms of complexity.

RMST [10] (Reliable Multi-Segment Transport) is a transport layer protocol designed to run in conjunction with Directed Diffusion [11]. Its primary goal is the reliable delivery of large pieces of data to all subscribed sinks. RMST is NACK-based; it places responsibility for loss detection at the receivers (which can be intermediate nodes as well as the actual sinks). Missing fragments requests are unicast from the sink to the source. Caches in intermediate nodes allow for fast recovery. However in the worst case, the repair request needs to travel all the way to the source. MOAP is similar to RMST in that it uses unicast repair requests; however MOAP repairs are local, since the dissemination mechanism guarantees that a source will be only one hop away.

ESRT [12](Event-to-Sink Reliable Transport) is a congestion control protocol that tries to meet reliability requirements set by the application, while conserving energy. It is receiver-driven but does *not* guarantee 100% delivery rate, making it problematic as a code distribution mechanism. ESRT operates by adjusting the reporting frequency of the sending nodes to achieve its optimal operating point (Low congestion, high reliability). In that aspect, it has no explicit retransmission scheme. ESRT is an *end-to-end* mechanism (where the edges are the sources and the sink). It is also dependent on control messages; the rate adjustment information has to be pushed all the way to the sources.

2.3 Multicast Reliability and Data Dissemination in Sensor Networks

The design of MOAP's repair and suppression mechanisms was influenced by architectural choices made in the Scalable Reliable Multicast protocol, SRM [1]. SRM imposes only the minimal requirements of reliable multicast: it guarantees eventual delivery of all the data to the group members, but not delivery order. SRM places responsibility about loss detection and recovery on the receiver; it is thus NACK based. It makes use of damping mechanisms to avoid control packet or repair request implosion; hosts wait some time before transmitting a request and do not repeat requests they overheard from their neighbors. The use of opportunistic listening, in addition to the control overhead of SRM, make a direct port to the sensor net domain problematic.

Several aspects of the dissemination methods of MOAP were influenced by Directed Diffusion [11]. Diffusion is a well-known data dissemination mechanism for Sensor Networks, whose main aspects include *data-centric routing*, *in-network aggregation* and *attribute-based data naming*. It implements a *publish-subscribe* interface, by having sinks send out interest packets. Sources whose data matches the interest then reply by sending data packets toward the sink. Interests, as well as exploratory data—used to reinforce a particular path—are disseminated using flooding. This can induce a significant overhead when there are several sinks in the network (as in a code distribution case). Two new variants were therefore proposed [13]: *Push Diffusion*, optimized for many receivers and few senders and *One-Phase Pull*, designed for the reverse case. Even though MOAP does not use any diffusion variant directly, since it is designed to be independent of the routing protocol, it includes a similar publish-subscribe interface.

3 Problem Description

In this section, we describe the Code Distribution problems, in terms of requirements, properties and resource prioritizations.

3.1 Requirements and Properties of Code Distribution

A code distribution mechanism should be designed to fulfill the following:

1. The complete image, starting from specific points in the network, must reach all the nodes. This is a *requirement*. We do not consider the extended problem of reaching only a *subset* of the nodes.
2. If the image cannot fit into a single packet, it must be placed in stable storage until the transfer is complete, at which point the node can be safely reprogrammed. This is also a *required* property.
3. The lifetime of the network should not be severely affected by the distribution operation. This is a *desirable* property.
4. The memory and storage requirements of the mechanism should not be very high since that would limit the available space for the normal application. This property is also *desirable*.

Required properties are necessary in order to ensure the correctness of the mechanism. Desirable properties are not required to ensure correctness, but should not be overlooked in any systems intended for practical use.

The fact that the *complete* image must reach *all* nodes is what makes code distribution a special case of multicast reliability. A reliability mechanism is required to ensure that the entire code is transferred to all nodes, in the presence of link losses and multiple hops. The mechanism should also handle disconnected nodes (as long as there is no permanent network partitioning).

3.2 Resource Prioritization

Satisfying the desirable properties is not overly complex if the sensor nodes are relatively powerful, in terms of computation, power, memory etc (e.g. embedded PCs connected to solar panels). On the other hand, when the target platform is a severely resource-constrained device such as a Mica-2 mote, careful planning is necessary. Since the mote is our target platform, we consider *Resource Prioritization* to be a fundamental design goal for our code distribution mechanism.

The most limited (hence important) resource on a mote is *Energy.* All operations require it and there is only a finite amount available—it is not always possible to equip motes with solar panels or replace their batteries. The most energy-intensive operation on the mote is radio usage and in particular, packet *transmission* (the CC1000 radio consumes 12 mA on transmit mode and 4 mA on receive mode). Another significant energy consumer is *stable storage (EEPROM) access.* A *Write()* operation needs on average approximately one-eighth the amount of energy required for transmitting the same number of bytes. *Reads*() are significantly cheaper than *Write()s* (by at least an order of magnitude) since most FLASH EEPROMs are optimized for *Read()* operations. However, due to the nature of code distribution, *every* code segment has to be stored in EEPROM; therefore, the number of *Write()s* can be a significant factor in the overall energy consumption.

Immediately following energy in terms of importance is *memory usage.* By memory, we primarily refer to the amount of static RAM. The limited amount of SRAM available on the mote platform, in conjunction with the ever-increasing complexity of mote applications has made main memory a highly prized resource. In addition, code distribution is *not* the primary application of a mote. It can be thought of as part of the operating system, a 'utility' service that needs to share memory with the 'real' application. Considering the current lack of dynamic memory allocation in motes, the mechanism's memory usage needs to be rather modest.

Based on the above, it is evident that the main goal is to limit energy consumption—in particular packet transmissions—by as much as possible. However, since optimizations aren't free, something should be traded off for reduced energy usage. For this particular application, *Latency* is a good candidate. Unlike the many sensor net applications, code distribution does not need to respond to real-time phenomena. It isn't sensing the physical world; instead, it's doing a large data transfer. In addition, we assume that code updates don't (or shouldn't) occur very frequently. For those reasons, we assume that latency is the *least important* resource and that it can be traded off without any serious consequences.

4 Design Choices and Alternatives

The first requirement in section 3.1 states that data should reach all the nodes. Intuitively, this suggests using the inherent broadcast capability of the wireless medium, in order to reach a large subset of nodes with one transmission. Broadcasts don't solve the multi-hop issue however; an appropriate dissemination protocol is needed. Requiring *all* data to be present on each node means that a reliability mechanism is needed in order to ensure packet delivery in the presence of lossy links. The second requirement implies that there should be some form of *code segment management* on the receiver. Therefore, one needs to consider the following design questions:

- *Dissemination protocol:* How is data propagated?
- *Reliability mechanism:* Who is responsible for initiating repairs? What is their scope? Is the scheme ACK or NACK based?
- *Segment management:* How are segments stored, retrieved and indexed? How can we detect a missing segment?

4.1 Dissemination Protocol

A common approach to disseminate data is to deliver data to all the nodes at the same time. Traditional IP multicast protocols do so by constructing trees, either rooted at the source or in a rendezvous point [14]. However, all nodes need to be reached in our case, so the tree needs to span the entire network, not just a subset. In addition, the protocol should be tolerant to route and link failures which happen often in the wireless domain.

The state requirements of multicast protocols make a direct porting to the sensor network world impractical. Routing protocols like Directed Diffusion [11] (and its mote-only implementation, Tiny Diffusion) reduce the memory requirements by taking advantage of *soft state.* Although the abstractions for diffusion support many-to-many communication, Tiny Diffusion is not optimized for disseminating data from many nodes to all nodes.

The last of the concurrent delivery mechanisms is *flooding.* Using flooding, one can expect that all nodes will be reached and its state requirements are minimal. Of course, the penalty is energy consumption since a considerable amount of transmissions are in fact duplicates.

Another approach to the dissemination problem is to transfer the data in a *neighborhood-by-neighborhood* basis. In essence this implies a single-hop mechanism that can be recursively extended to multi-hop. At each neighborhood, only a small subset (preferably, only one) of the nodes is the 'source' while the rest are the receivers. When the receivers have the entire image they can become sources for their own neighborhoods (that were out of range of the original source). A mechanism is required to prevent nodes from becoming sources if another source is present in their neighborhood. This can be done by using a publish-subscribe interface. Sources publish their newer version of the code image and all interested nodes subscribe. If a source has no subscribers it will be *silent*. Therefore, one hopes to take full advantage of the nature of the broadcast medium, where only one transmission can reach all nodes within range in the absence of losses. As long as there is no network partition, all the nodes will eventually receive the full image.

This mechanism, which we call Ripple (from its ripple-like propagation property) has another advantage: it guarantees that, if a data transmission is in progress, the source is *only one* hop away. Since only nodes that have the full image can be sources, all repairs are *local*. However, the potential traffic reduction comes at a price, namely, increased latency. Since data is not delivered to all nodes at the same time anymore—they require the full image to become sources—the operation is definitely slower than a concurrent delivery approach.

The requirement that a node needs the entire image to become a source is not strict. It is possible to have nodes become sources only when a percentage of the image is present. However, this adds additional complexity to the system and might be impractical for a device such as a mote. By decoupling the senders from the receivers and thus forcing a node to be in either state but not in both, we are trading complexity (memory) for latency. Ripple is also consistent with the resource prioritization presented in section 3.2 since it mainly trades off latency for energy.

4.2 Reliability Mechanism

Unlike several sensor network applications in which some packet loss can be tolerated, in code distribution we cannot afford to lose any data—the complete image is required. One approach to reliability is to use forward error-correction: send $N + K$ packets. As long as *any* N packets are received, the full image can be reconstructed. The other, more traditional choice, is to use a retransmission mechanism. For the purpose of our code distribution scheme, we will focus on the retransmission approach.

The first question that needs to be answered is: who is responsible for detecting a loss? If the sender is responsible for all its receivers, then the sender needs to keep state for all of them. On the other hand, a receiver needs to keep state of only one node—the sender. To minimize required sender state, nearly all IP multicast mechanisms make receivers responsible for detecting losses (and initiating repairs). Consequently, the repair mechanism is NACK-based since positive acknowledgment schemes imply that loss detection and repairs are done in the sender. The extra benefit of a NACK-based approach is the significant reduction in control traffic—requiring an ACK per packet sent can have a potentially very high energy overhead.

The next question concerns the scope of repairs. How far along the path to the source do we need to inquire when something is missing? If the missing segment is several hops away, the number of transmissions required for a repair can be considerable. SRM [1] suggests that repairs should be *local* as much as possible. However, since we require *all* the nodes to eventually have the entire (same) image, we can safely assume that given enough time, the missing packet will be only one hop away. We can therefore impose the restriction that *all* repairs be local. Intuitively, this will provide a considerable reduction in energy consumption and complexity (since repair requests and replies don't need to be routed over a potentially long multihop path). The cost, again, is latency.

Finally, we need to choose a retransmission policy. In particular, we need to answer the question: should repair requests be broadcast or unicast? Broadcasting requests gives a higher probability that the requester will receive a reply, since potentially all nodes in the neighborhood will honor the request. But this can cause an excessive number of duplicates and is also subject to 'implosion' effects, so a suitable suppression mechanism is required. This is a non-issue when using a unicast, which can be considered an extreme duplicate suppression mechanism. On the other hand, the probability of receiving a reply is reduced since only one node is honoring the request. In addition, if that node fails or becomes disconnected, the requester will have no way of recovering the remaining segments, unless a 'source discovery' mechanism is applied.

4.3 Segment Management

According to the second requirement in section 3.1, we should place segments of the image in stable storage in order to reconstruct it when the transfer is complete. It is important to know whether a segment is present or not in order to ask for retransmissions or honor repair requests. A simple method is to just read the corresponding EEPROM address and check whether the segment is present or not. However, this involves an I/O operation—a *Read()*—and thus consumes energy. Moreover, if we want to find *all* the missing segments we need to potentially do *Read()s* that span the *entire* segment address range.

We can avoid all those expensive operations by keeping a *record* (a bitmap) of successfully received segments. However, code images are large so storing the bitmap in RAM can consume a considerable amount of memory. This problem is augmented by the lack of dynamic memory allocation on motes—we need to keep a bitmap that can store up to the maximum size of a code image, even if the actual image is much smaller than the maximum. Therefore, we are not only reserving a large amount of memory but are potentially underutilizing it. These observations lead us to consider treating RAM and EEPROM as a memory hierarchy and explore the properties of a *hierarchical data structure.* Parts of the bitmap can reside in RAM and others can be placed in stable storage, with swapping being done as required. Clearly, this approach saves RAM usage but consumes more energy since EEPROM access is now involved.

A completely different approach is to not do any complex segment management but instead use a sliding window. At any point, the receiver knows it has successfully received packets up to the beginning of the window(*base*). It can then receive and successfully store and retrieve up to the size of the window(*offset*). This is similar to a small map size in the hierarchical case, with one important difference: the receiver *cannot* receive a random segment and store it anymore. The segment must fall inside the window, otherwise it will be discarded. For segments smaller than *base* this is not a problem since they are duplicates, but segments larger than *offset* aren't. The advantage of this approach is that it does not involve *any* extra EEPROM I/O. The disadvantage is that its *out-of-order tolerance* is much less than the previous mechanisms. We explore the differences and costs of all those approaches in the next section.

5 Analysis and Comparison

The enumeration of choices in the previous section is by no means exhaustive. However, we believe that they represent a considerable part of the design space. We now proceed to analyze and compare a subset of those choices, using our resource prioritization goals—energy, memory (representing complexity) and latency—as a baseline. As with Section 4 we compare neighborhood-by-neighborhood dissemination (Ripple) to concurrent delivery (Flooding) using a simplified model.

5.1 Ripple vs Flooding

The basic assumptions that we make for this analysis are:

- There is only one 'original' source.
- Each packet reception has a fixed probability of failure p, based on the link quality. This probability is the same for *all* nodes and all transmissions. This assumption is not realistic but it simplifies the analysis considerably.
- The transmission rate is constant.
- A neighborhood is a set of nodes that are in the same broadcast domain with each other. Neighborhoods can overlap.
- All repairs are assumed to be local.

In addition, we will make use of the following definitions:

- Number of total segments to be transmitted: S
- Data rate: D
- Number of nodes in each neighborhood: $o_i \geq 2$
- Number of sources at each neighborhood: $k_i \geq 1$
- Hop dist. of node i from original source: $h_i \geq 1$
- Total number of nodes: N
- Total number of neighborhoods: O_{tot}
- Expected number of transmissions needed for all nodes to receive the code image: $E[Tx]$
- Expected amount of time needed for all nodes to receive the code image: $E[Time]$

When using flooding, the probability of a node belonging to neighborhood i not receiving a packet is on average $P = p^{o_i}$, since every node in the neighborhood will forward the packet. So one packet will be transmitted $o_i + p^{o_i}$ times in a neighborhood. The sum of nodes of all neighborhoods is N, so the expected number of transmissions for the entire network is:

$$E[Tx_{flooding}] = N \cdot S(1 + \sum_{i=1}^{O_{tot}} p^{o_i}) \qquad (1)$$

Where the second term is the total number of retransmissions required throughout the network. The average time it takes for node i at distance h_i from the source to receive the entire image is:

$$E[Time_{flooding}] = D{\cdot}S{\cdot}E[Retx_i] \qquad (2)$$

Where $E[Retx_i]$ is the expected number of retransmissions required for this particular node. Note that the expected time is *independent* of the distance to the original source—assuming zero forwarding delay. This behavior is expected, given the concurrent nature of flooding.

In the case of ripple, one packet gets transmitted $k_i + p^{k_i}$ times in a neighborhood, so the total number of transmissions is:

$$E[Tx_{ripple}] = S \sum_{i=1}^{O_{tot}} (k_i + p^{k_i}) \qquad (3)$$

The value of k_i depends on the amount of overlap between neighborhoods. Ripple, however, has the property that a potential source will be silent if it has no subscribers. Since there are o_i nodes total in a neighborhood, in the worst case k_i can be no more than $(o_i/2) + 1$. In the absence of losses, this observation means that ripple has at least $\frac{N/2+O_{tot}}{N}$ fewer transmissions than flooding. We also note that as network density increases the reduction in traffic becomes more pronounced, as k_i is in fact *reduced*. In the limit, i.e. a fully connected network with diameter 1 (which is of course a trivial case), ripple requires S transmissions while flooding requires $N{\cdot}S$.

In the presence of losses however, things are less favorable for ripple. Since $k_i \leq (o_i/2) + 1$, the loss probability is higher than flooding. We therefore expect that ripple will require more retransmissions. If the link loss rate is sufficiently high, it will end up sending as many packets as normal flooding.

Our loss model does not take *collisions* into account, but in a real channel they can lead to a substantial number of losses. We expect the average number of collisions, when using Ripple to be much less than Flooding, since there is less contention for the channel.

The average time it takes for node i at distance h_i from the source to receive the entire image, when using ripple is:

$$E[Time_{ripple}] = D{\cdot}S{\cdot}E[Retx_i]h_i \qquad (4)$$

Note that the ripple delay is directly proportional to the distance of node i from the original source, while the equivalent flooding delay is independent of distance.

From the equations above, it is apparent that ripple is *not* suitable for sparse networks with a large diameter: the number of transmissions will be approximately equal to flooding, while the delay will be $max\{h_i\}$ times more.

5.2 Retransmission Policy: Broadcast vs Unicast

In 4.2 we characterized our reliability mechanism as being local-scoped and NACK-based. We now explore the implications of choosing a *retransmission* policy, specifically whether to unicast or broadcast requests when asking for a retransmission. Since a broadcast request can lead to an excessive amount of replies, a suppression mechanism is required. We consider three different suppression mechanisms for broadcasts, in addition to the unsuppressed case. Therefore, the different design options that we investigate are:

1. *Broadcast* RREQ (Repair Request), no suppression (all nodes reply).

2. *Broadcast* RREQ, nodes choose a randomized interval, snoop on the channel for transmissions and reply if the interval expires and no one else has replied. This suppression mechanism was introduced in [1].

3. *Broadcast* RREQ, all nodes reply with a fixed (static) probability.

4. *Broadcast* RREQ, all nodes reply with an adaptive probability, based on some metric like the neighborhood size, or, in the case of ripple, whether the node is a source with subscribers or not.

5. *Unicast* RREQ, only the source replies. Situations where the original source fails are handled by the requester doing some sort of *source discovery.*

The randomized interval algorithm is theoretically the most efficient of the suppression mechanisms when the randomization interval is large. The other two trade off optimality for a reduction in complexity. They are more appropriate for ripple, where there is a clear distinction between nodes that are active sources and nodes that aren't. Option 3 is static in that the probability of generating a reply is not dependent on network dynamics. This can lead to an expensive retransmission policy. Option 4 tries to do better by using an estimation function like, for example, the neighborhood size.

Assuming again that all links have the same, uncorrelated loss probability p, if the requesting node sends an RREQ and that triggers one reply, the probability of getting the missing packet is: $P = (1 - p)^2$. This simplifying assumption ignores collisions when multiple nodes reply at the same time, so it is biased favorably toward broadcasting schemes.

When using Ripple, the set of nodes that can honor a repair request can be split into two parts: k is the

Policy	Expected number of replies	Latency	Complexity
Broadcast request, all nodes reply	$(1-p)^2(k+m)$	0	$O(1)$
Broadcast request, random interval suppression	$(1-p)^2[1+(k+m-1)/C]$	Up to C	$O(neighborhood\ size)$ for a good estimation of C. Several timers
Broadcast request, nodes reply with a static probability	$(1-p)^2(a{\cdot}k+b{\cdot}m)$	Depends on selection of a, b	$O(1)$
Broadcast request, nodes reply using adaptive probability	$(1-p)^2(a{\cdot}k+b{\cdot}m)$	Depends on selection of a, b	$O(neighborhood\ size)$ for a good estimation of a, b
Unicast request, only publisher replies	$(1-p)^2$	Considerable if link to publisher fails, else 0	$O(1)$

Table 1: Comparison of different retransmission policies, in terms of packets generated, latency and complexity.

subset of nodes that has subscribers (the same definition as k_i in 5.1), while m is the subset of nodes that have the whole image but *no* subscribers. In the case of flooding, this distinction doesn't exist. The expected number of packets, $E[A]$, that get sent to the requesting node, for each option is:

1. $E[A] = (k+m)(1-p)^2$
2. $E[A] = [1 + \frac{(k+m-1)}{C}](1-p)^2$, where C is the randomization interval.[1]
3. $E[A] = (a{\cdot}k + b{\cdot}m)(1-p)^2$, a,b are statically assigned probabilities. For flooding, $a = b$
4. $E[A] = (a{\cdot}k + b{\cdot}m)(1-p)^2$, a,b are dynamically assigned probabilities. For flooding, $a = b$
5. $E[A] = (1-p)^2$

If $E[A] < 1$, another RREQ might be required. $E[A] > 1$ indicates that there are potential duplicates generated.

Based on the above, option 5 is the best in terms of reducing duplicates, and 1 is the worst. Option 3, with its statically assigned probabilities is not as flexible as 2 and 4, so for large values of a and k it can degenerate to option 1.

The success of the suppression mechanisms depends largely on a correct estimation of the values of k and m, especially when Ripple is used. The probabilistic techniques are affected more directly than option 2, but, since C depends on the number of nodes capable of replying, it is also not immune.

Options 1, 3 and 5 don't involve any estimation algorithm so their space complexity is $O(1)$. For 2 and 4 complexity is $O(neighborhood\ size)$, for a good estimation of their subsequent parameters. Option 2 also requires several software timers.

In terms of latency, option 1 is the fastest since replies are generated immediately. Option 5 also has zero latency assuming that the source doesn't fail. If the source fails, a source discovery mechanism needs to be triggered. That can incur a considerable delay. The latency of option 2 depends on the randomization interval, C (it is in fact at most C). Options 3 and 4 have a latency of zero with probability $max\{a, b\}$, one with probability $max\{a(1-a), b(1-b)\}$, etc. In reality, however, methods that create an abundance of duplicates can end up having considerable latency, due to the increased probability of collisions. In essence, the channel is operating above capacity and a large number of transmissions fail. This increases the probability of no replies making it back to the requester, which in turn sends another repair request-therefore adding to latency [13]. In that sense, duplicate suppression is another congestion control [12] method.

The comparative results of all five options are shown in table 1.

5.3 Segment Management: Hierarchical Data Structures vs Sliding Window

We have identified five segment management alternatives, which are categorized based on the technique used to determine the presence or absence of a segment.

1. *No indexing:* No data structure is used to indicate the presence of segments in EEPROM. To find if segment i is missing, we need to read the corresponding entry from EEPROM.

[1] If we have K nodes each picking a reply interval uniformly from the range $[1..C]$ (with discrete slots), then, each slot has been picked, on average, by K/C nodes. So we have 1 reply and $(K-1)/C$ duplicates.

Segment management	RAM (bytes)	TX Cost	RX Cost	Gap detection Cost	Out-of-order tolerance
No indexing	0	R	$R + W$ always	Up to C/S	Complete
Full indexing	$C/(8S)$ Typical 1024	R	W when segment missing, else 0	0	Complete
Partial indexing	$C/(8kS)$ Typical 256	R	Up to kR if entry empty Up to $kR + W$ if segment missing 0 if bitmap entry full	Up to kR Minimum R	Complete
Hierarchical full indexing	$C/(64E)$ Typical 8	R	R if entry empty $R + 2W$ if segment missing Else 0	R	Complete
Sliding window	$M/8$ Typical 4-8	R	W always	0	Up to bitmap size

Table 2: Comparison of different segment management techniques in terms of RAM usage, transmission, reception and gap detection I/O cost and out-of-order tolerance. R denotes a *Read()* and W represents a *Write()*.

2. *Full indexing:* The entire segment bitmap is kept in RAM. It has one entry per segment (one-to-one, or full, mapping). To find if segment i is missing, we need to just look at entry i.

3. *Partial indexing:* Each entry in the bitmap represents a set of k consecutive segments. An entry is full if all its corresponding segments are present, otherwise it's empty. To find if segment i is missing, we need to look at entry i *div* k. If the entry is full the segment is there, otherwise we need to do up to k sequential *Read()s* to determine the status of i.

4. *Hierarchical full indexing:* Similar to full indexing, but the bitmap is stored in two levels, using both RAM and EEPROM. The bottom level, which is kept in EEPROM is using full indexing for a *subset* of the code image, which we call a *page*. The page size can be arbitrary, but, for ease of analysis, we consider it to be equivalent to the size of a physical EEPROM page. A complete record of all pages is kept in RAM—the top level. Since pages are relatively large, the RAM usage of this method is minimal. To find if segment i is missing, we need to first associate it with a page entry (i *div* m, where m is the number of bits in a page) and then look in this particular page for entry i *mod* m.

5. *Sliding window:* A bitmap of up to w segments is kept in RAM, starting at the last segment successfully received *in order*. To find if segment i is missing, we check if $i < last$; if so, the segment is present. If $last < i \leq last + w$, we check the bitmap to see whether the segment has been received. If $i > last + w$, the segment is considered to be missing, so the out-of-order tolerance of this approach is limited.

We are interested in analyzing the above approaches in terms of EEPROM access which directly translates to energy usage, but also in terms of RAM cost. We define *TX Cost* to be the cost, in terms of EEPROM I/O, of transmitting a segment. When receiving a segment, we need to determine if it is a duplicate, in which case we discard it, or not, in which case we store it. This is the *RX Cost*. We also define the cost of finding out the *first* missing segment—so as to ask for a retransmission—as the *Gap Detection Cost*.

The first method doesn't require any memory, so its RAM cost is zero. Full indexing needs an entry for each segment. If the total image size is C bytes and each segment contains S bytes, then the RAM required is $\frac{C}{8S}$. The maximum value of C on the mote is 128KB and with $S = 16$ *bytes*, a typical value when the packet size is kept small, the RAM cost of full indexing is 1K; a quarter of the total RAM available on the current generation of motes. Partial indexing keeps k segments per bitmap entry, so its cost is $\frac{C}{8kS}$. With $k = 4$, this becomes 256 bytes. Hierarchical full indexing requires $\frac{C}{8(8E)}$ bytes in RAM, where E is the page size in bytes. If the page size is equivalent to the physical page size its value is 256, so by using bitmaps for both levels, the RAM cost is 8 bytes. The sliding window method's RAM requirements are $\frac{M}{8}$ where M is the window size. Typical values are 4-8 bytes, corresponding to window sizes of 32 up to 128.

The transmission cost for all methods is one *Read()* since to transmit segment i we always need to read the corresponding EEPROM entry—assuming of course that we *have* segment i. In terms of RX Cost, the first method requires a *Read()* before a *Write()*, since it has no other way of knowing if the segment is a duplicate. Full indexing and the sliding window method do a *Write()* only when the segment is missing and that has zero cost in terms of EEPROM I/O. Partial indexing needs to do up to k *Read()s* when the corresponding entry is empty, in order to determine *which* segment is missing. We need to pay that cost when at least one of the k segments is missing, even when the received segment is actually a duplicate. If the segment is not present we of course have to do a *Write()* as well. If the bitmap entry is full, the cost is zero. Hierarchical full indexing requires a *Read()* to determine if the segment is a duplicate, assuming that the top-level record indicates an incomplete page. If the segment is indeed missing, we need two *Write()s*—one to update the bottom-level bitmap in EEPROM and one to actually write the segment. Again, if the top-level record indicates the page is full, the cost is zero.

In terms of Gap Detection Cost, the first method needs to read all EEPROM entries sequentially to find the first missing segment, since it doesn't use a bitmap. This means that it might have to do up to $\frac{C}{S}$ *Read()s every time* a segment is missing. Full indexing and Sliding Window have a gap detection cost of zero. Partial indexing needs to do up to k *Read()s* to find the missing segment, with a minimum of one *Read()*. Hierarchical full indexing only needs to do one *Read()* and then locate the missing segment from the bitmap fetched from EEPROM.

Table 2 shows the results of the analysis. We can easily discern that having no record at all is quite expensive in terms of energy, even though its memory usage is zero. The three different indexing schemes all trade memory usage for I/O. Hierarchical full indexing has the lowest memory usage but it requires two *Write()s* per new segment received. This can be quite expensive, considering that the cost of a *Write()* is about an order of magnitude higher than the cost of a *Read()*. Partial indexing consumes less energy than hierarchical full indexing, for small k (around 8 or less), at the cost of using more memory. Finally, the sliding window method has the best combination of energy and memory cost, but trades off out-of-order tolerance. If link losses are high, this can lead to an *increase* in energy consumption due to unnecessary retransmissions. Nevertheless, if the link losses are such that the probability of a receiver losing synchronization with the sender (and thus receiving packets outside the window) is small enough, the sliding window method seems like the most appropriate choice.

6 Implementation

Based on the design goals and priorities described in the previous sections, we made the following implementation choices for MOAP: *Ripple* dissemination protocol, *Unicast* retransmission policy and *Sliding Window* for segment management. Energy consumption remains our primary constraint, while RAM and program memory usage are also important.

The process for programming a mote over the air is as follows. First, the programmer builds the new code, using the standard TinyOS tools. The binary image is then passed to a *packetizer* that divides the Motorola SREC-format binary into actual segments. A segment has a 2-byte address field indicating its address in program memory and a 16-byte data field. In the current version of MOAP, each packet contains one segment.

One mote attached to the PC becomes the *original source* (a basestation). It sends PUBLISH messages, advertising the new version of the code. Nodes check their version number and send SUBSCRIBE messages if it is smaller than the advertised versions. Nodes also use a *link statistics* mechanism so as to not subscribe to sources that have very lossy, intermittent, or otherwise unreliable links. Once the original source receives a subscribe message, it waits for a small amount of time to allow other nodes to send in their subscriptions as well and then starts the data transfer.

As Ripple suggests, once a mote has the complete image stored in EEPROM, it will send PUBLISH messages itself, becoming a *secondary source*. If it doesn't receive any subscribe messages in a specific amount of time it *Commits* and invokes the bootloader to transfer the code from EEPROM to program memory and then restart the mote with the new code.

Active sources don't stay active forever, otherwise they would never commit. Instead, after transmitting the entire code image and waiting a predetermined amount of time in order to handle potential retransmissions of the last segments they also commit. Eventually, assuming the network does not partition, all nodes (besides the basestation) will commit the new code.

When a node detects a lost segment (using the sliding window method), it will ask the source for a retransmission, using a unicast packet. Retransmission requests have higher priority than regular packets; thus, a source will first honor all its retransmission requests and then resume the regular data transfer. Sources suppress duplicate requests, i.e. if N nodes request segment k within a given time period, the source will only transmit k once. Nodes keep track of their sources' activity using a *keepalive* timer.

The keepalive timer has a dual purpose: it solves the 'last packet' problem inherent in NACK-based schemes and is also used as a contingency mechanism in the case the source dies or the receiver loses its connection. If after a certain amount of time the receiver hasn't heard from its source, it will transmit a *broadcast* repair request. All sources within range will reply and then the node will select a new source, based on configurable properties, like link statistics—the same technique used to subscribe to the previous source.

If there are no sources within range, the mote continues to send broadcast repair requests. After a maximum number has been reached, the node will perform an *Abort*—it will reset all MOAP-relevant state, but will not erase segments from EEPROM. It then will wait for a new neighbor to become a source, in which case it will subscribe and continue, or it will invoke the *Late Joiner* mechanism.

The purpose of the Late Joiner mechanism is to allow nodes that are disconnected, have just recovered from failure or have been in any way detached from the code transfer operation to also receive the new image. It requires all nodes to periodically send publish messages, advertising their version. If a node detects a version mismatch and its version number is smaller, it will send a publish message. Instead of using a new packet to only send a 2-byte version number, we could *piggyback* it on existing periodic messages—such as neighbor beacons, in a neighbor discovery protocol, or interests, in Tiny Diffusion. The current version of MOAP does not use piggybacking.

The RAM footprint of MOAP is currently approximately 700 bytes, while the ROM (program memory) footprint is approximately 4.5 Kbytes. Careful optimization should reduce the RAM usage even further. However, a potential caveat lies into the fact that memory optimizations—for example multiplexing a single timer instead of using several—can increase the ROM footprint, since additional control instructions are required. Increasing the ROM footprint leads to *increased* energy consumption. The *entire* MOAP code needs to be transferred, since, to retain the ability to reprogram, every new code image must include MOAP. This problem can be solved to an extent by using difference-based techniques [6].

The current version of MOAP has been successfully used to repeatedly reprogram motes up to four hops away from the basestation, using code images of various sizes, ranging from 600 up to 30K bytes.

7 Evaluation

Our analysis of MOAP (Section 5) provides only steady-state performance estimations given several assumptions. To evaluate real-world performance of the various design choices, we implemented MOAP in the EmStar [15] emulation environment. We then focused on validating a subset of those choices using the native mote implementation.

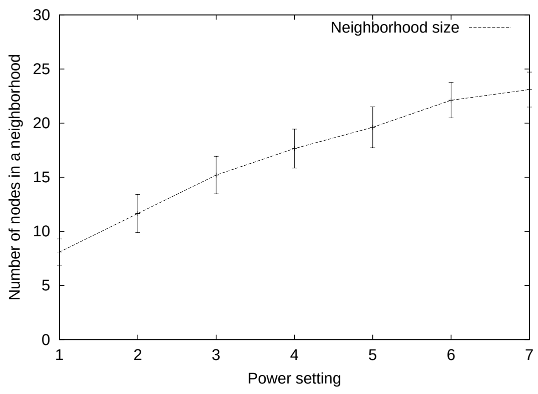

Figure 1: Mean neighborhood size for different power settings. Increasing the power results in an almost linear increase in the neighborhood size. In the highest power setting, approximately 80% of the network is connected.

7.1 Emulation

EmStar is running on a 32-bit platform with dynamic memory support, so methods like Full indexing are not as expensive as in the mote case. However, the real radio channel allows us to evaluate different dissemination methods, as well as retransmission policies. Mote-dependent details such as EEPROM management and writing into program memory via a bootloader are abstracted away but other than that, the functionality that the EmStar implementation provides is identical to that of code running on real motes.

The EmStar experimental setup consisted of 30 Mica-1 motes placed at the ceiling of our laboratory. Since the placement of the nodes is fixed (they are attached, via serial cables, to an EmStar node), we changed their radio power in order to capture effects of variable density.

The average neighborhood size (an indication of network density) for the seven different power levels used throughout our experiments is shown in Figure 1. In our setup, two nodes are considered neighbors when the *bidirectional* loss rate, provided by the link statistics mechanism, is no more than 15%. The neighborhood size increases almost linearly as power settings increase. We use this result in future figures, by just reporting power settings. In addition, in this and all subsequent figures, data points are taken by averaging over 12 experimental runs, for each power setting; error bars represent 95% confidence intervals.

The transmitted code image consisted of 100 segments, with one segment per packet. For methods using sliding window, the window size was 16 bits. The experiments ran until each node had received the image in its entirety.

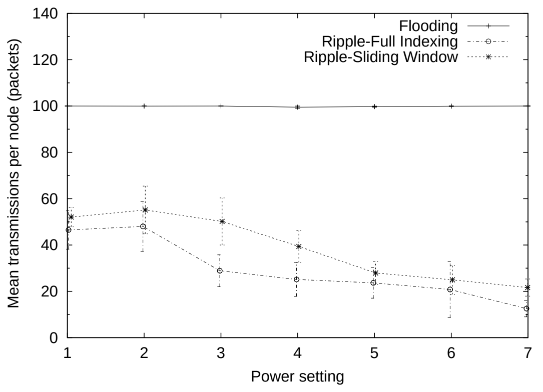

Figure 2: Mean packets transmitted per node versus radio power (different network density), for Flooding using Sliding Window, Ripple using Full Indexing and Ripple using Sliding Window. As network density increases, the energy savings obtained when using Ripple become more profound. File size is 100 segments with one segment per packet. The window size is 16 bits.

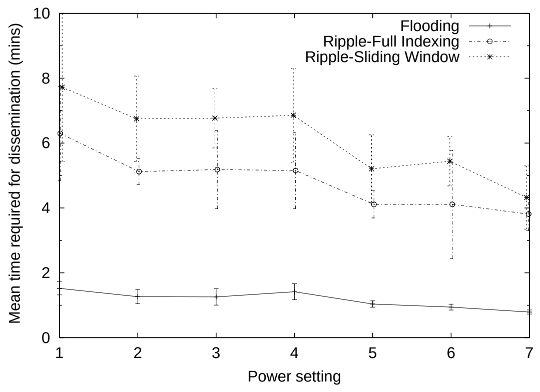

Figure 3: Mean time required for the entire file to reach all the nodes versus radio power, for Flooding using Sliding Window, Ripple using Full Indexing and Ripple using Sliding Window. The two Ripple variants are approximately five times slower than Flooding.

7.1.1 Energy consumption

In order to determine the energy consumption of different dissemination strategies we compared Ripple with Sliding window against Flooding with Sliding window and Ripple with Sliding window against Ripple with Full indexing. Since Full indexing is identical to the other indexing methods portrayed in section 4.3 in terms of out-of-order tolerance, we expect those methods to exhibit the same behavior in terms of packets transmitted. The average number of packets transmitted per node, including retransmissions, for different power settings, are shown in Figure 2.

Flooding transmissions are always very close to 100—each of the thirty nodes ends forwarding the entire file. Changes for different power settings are very small. In contrast, the Ripple variants are quite sensitive to changes in network density. When network connectivity is sparse, they incur an average of 50% reduction in traffic as opposed to flooding. The difference becomes more pronounced as the neighborhood size increases. In relatively dense networks, Ripple can result in an order of magnitude reduction in traffic, leading to substantial energy savings. The results match the simple models presented in section 5.1. Flooding results show very little fluctuation since the number of retransmissions for flooding is minimal; there is so much redundancy that repairs are rarely needed.

Ripple using Full Indexing performs, on average, 5–15% better that its Sliding Window counterpart. The difference is primarily due to the limited out-of-order tolerance of the Sliding Window mechanism, which results in more retransmissions. However, in high density networks, the differences are not large enough to warrant incurring the cost of an Indexing method—the Sliding window is a better alternative.

7.1.2 Latency

The energy savings of Ripple are not free, however. Figure 3 shows the average time required for the code image to reach all the nodes, for a transmission rate of 2 packets per second. The transmission rate is quite modest; an implementation could increase it up to five times or more and still expect not to over-utilize the channel, when using Ripple. Using flooding is different since the excessive number of transmissions can saturate the channel quickly. This can substantially increase the number of retransmissions needed, as the collision probability is large [16]. For the rate used in the experiments, the Ripple variants are significantly slower than Flooding. The results are consistent with the analysis presented in section 5.1. Ripple's latency is reduced at higher densities, while flooding is not extensively affected by it. Full indexing performs better than Sliding window, requiring on average 20–30% less time. Again, the reason is the increased number of retransmissions.

7.1.3 Retransmission policies

Using the Ripple-Sliding Window variant, we compared the unicast retransmission mechanism (the one used in the mote implementation) with an unsuppressed broadcast scheme. The results, in terms of total number of retransmissions performed in the network, for different power settings, are shown in Figure 4. Using unicast instead of broadcast leads to a very

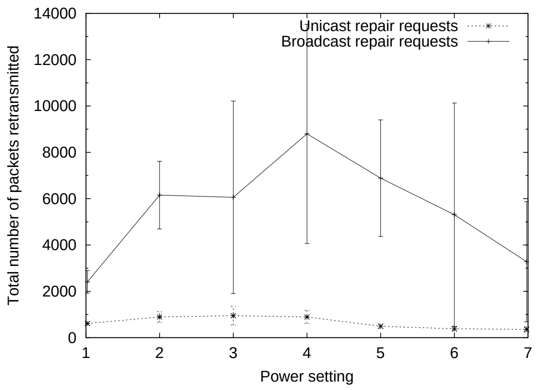

Figure 4: Total number of packets transmitted versus radio power, for two retransmission policies: Broadcast with no suppression and Unicast. Using unicast results in massive gains in terms of duplicate suppression, hence energy savings.

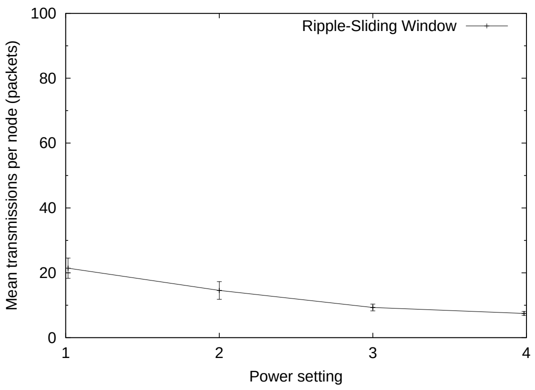

Figure 5: Mean packets transmitted per node versus radio power, for Ripple with Sliding Window on standalone motes. The more powerful and reliable CC1000 radio on the Mica-2 motes is the main reason for the reduction in transmissions, compared to Figure 2.

significant reduction of packets retransmitted, especially in higher power settings which correspond to larger densities and better links.

The advantage of using unicast requests over broadcast becomes more pronounced when using a MAC that provides link-level retransmissions, such as S-MAC [17], since the unicast reliability is then substantially higher. Even though neither of our implementations currently use S-MAC, we are planning on using it in future releases of MOAP.

7.2 Mote Implementation

We also conducted experiments using the actual (mote-only) implementation of MOAP in order to evaluate it in its target platform. The setup consisted of 15 Mica 2 motes. Again, a file size of 100 segments was used, with 1 segment per packet and a 16-bit window. Figure 5 shows the average number of transmissions per node for Ripple with sliding window—as described in section 6. Since we don't currently have a TinyOS implementation of Flooding or Ripple with Full indexing, there are no comparative results for those methods in the mote-only version. In addition, there are fewer power settings in the mote experiments since after level 4 all nodes were in the same broadcast domain. The more powerful and reliable CC1000 radio present on the Mica-2 motes—as opposed to the RFM radio on the Mica-1s—results in better link qualities, fewer retransmissions and a more rapid change in the neighborhood size. The number of transmissions per node is therefore significantly fewer than the corresponding emulation graph. Still, the trend of Ripple is preserved: higher network density leads to a reduction in transmissions.

We do not present experimental results for flooding on standalone motes because we can safely assume that they would again be close to 100, as in the emulation case—with a very small number of retransmissions. The reason is again the nature of flooding—every node in the network will forward all the packets.

8 Future Work

In the future, there are several important features that can improve the performance and functionality of MOAP:

Sending differences between versions. Recent results [6] have shown that sending differences between versions instead of an entire new version can result in an order of magnitude decrease in the size that is to be transferred. Although we cannot perform the image reconstruction directly into the mote's program memory without the help of the bootloader, we can transfer the code image in EEPROM and construct the new image there. This approach is complementary to the MOAP mechanism; together they can lead to even greater reduction in energy usage.

Support for selective node updates. Currently, MOAP tries to update every node to the same version of the code. However this isn't always desirable. Selective updating is possible if we don't require each node to commit the new code after receiving it. Therefore, intermediate nodes that are not interested in the new version can still act as Ripple sources. If the nodes that need the update are dense enough, there is no significant energy penalty. But for small sets of nodes that are topologically distant, other dissemination techniques might be required.

9 Conclusions

As sensor networks mature and grow larger in size, remote programmability will become a critical system service. In this paper we presented MOAP, a Multihop Over-the-Air Programming mechanism that is specifically targeted at large networks of Mica-2 motes. MOAP is designed to be energy and memory efficient, at the expense of increased latency. Our design choices are focused on three areas: dissemination protocol, retransmission mechanism and storage management of code segments. We analyzed an array of different options using some simple models and then evaluated our implementation using results from emulation as well as a mote-only testbed. The reliability mechanisms of MOAP also help explore the design space for reliable communications.

By using the Ripple dissemination protocol, MOAP achieves a significant reduction in transmitted traffic as opposed to flooding, ranging from 60–90%. It accomplishes this by selecting only a small subset of nodes within a broadcast domain to act as sources for the code image, via a publish-subscribe interface. A simple sliding window scheme is used as a repair mechanism. Even though it has limited out-of-order tolerance, we showed that it performs adequately, compared to substantially more complex schemes. We also showed that a unicast retransmission policy was very effective in suppressing duplicates.

Using our mote implementation, we were successful in reprogramming motes several hops away from the basestation. In the next few months we expect to deploy an improved version of MOAP in the field.

Acknowledgments

MOAP was made possible with support from DARPA grant #442511-ED-25621 and the National Science Foundation Cooperative Agreement #CCR-0120778, supporting the Center for Embedded Networked Sensing. For this work, John Heidemann is partially supported by the National Science Foundation (NSF) under grant number ANI-0220026, "MAC Protocols Specific for Sensor Networks".

We would like to thank Crossbow Inc. for providing the source code for their bootloader, as well as David Culler and Jason Hill for making valuable comments that helped improve the functionality of MOAP.

References

[1] Sally Floyd, Van Jackobson, Ching-Gung Liu, and Lixia Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. In *Proceedings of the ACM SIGCOMM Conference*, pages 342–356, Cambridge, MA, USA, August 1995. ACM.

[2] Schulzrinne, Casner, Frederick, and Jacobson. RTP: A transport protocol for real-time applications. *Internet-Draft ietf-avt-rtp-new-01.txt (work in-progress)*, 1998.

[3] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proceedings of the Ninth International Conference on Arhitectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 93–104, Cambridge, MA, USA, November 2000. ACM.

[4] Crossbow Technology Inc. Mote in-network programming user reference, http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/xnp.pdf.

[5] Crossbow Technology Inc. Mica2 wireless measurement system datasheet, http://www.xbow.com/products/product_pdf_files /datasheets /wireless/6020-0042-03_a_mica2.pdf.

[6] Niels Reijers and Koen Langendoen. Efficient Code Distribution in Wireless Sensor Networks. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 60–67. ACM Press, 2003.

[7] Vladimir Bychkovsky, Bret Hull, Kyle Jamieson, Stanislav Rost, and Hari Balakrishnan. Reliable Data Dissemination in Wireless Sensor Networks. *Poster in SOSP '03*, October 2003.

[8] Philip Levis and David Culler. Mate: a Tiny Virtual Machine for Sensor Networks. In *Proceedings of the 10th international conference on architectural support for programming languages and operating systems (ASPLOS-X)*, pages 85–95. ACM Press, 2002.

[9] C.Y. Wan and A.T. Campbell. PSFQ: A Reliable Transport Protocol For Wireless Sensor Networks. In *Proceeedings of First ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, Atlanta, Georgia, USA, September 2002. ACM.

[10] Fred Stann and John Heidemann. RMST: Reliable Data Transport in Sensor Networks. In *Proceedings of the First International Workshop on Sensor Net Protocols and Applications*, page to appear, Anchorage, Alaska, USA, April 2003. USC/Information Sciences Institute, IEEE.

[11] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building Efficient Wireless Sensor Networks with Low-Level Naming. In *SOSP 2001*, Lake Louise, Banff, Canada, October 2001.

[12] Yogesh Sankarasubramaniam, Özgür B. Akan, and Ian F. Akyildiz. ESRT: Event-to-Sink Reliable Transport in Wireless Sensor Networks. In *Proceedings of*

MobiHoc 03, Annapolis, Maryland, USA, June 2003. ACM.

[13] John Heidemann, Fabio Silva, and Deborah Estrin. Matching Data Dissemination Algorithms to Application Requirements. Technical Report ISI-TR-571, USC/Information Sciences Institute, April 2003.

[14] James F. Kurose and Keith W. Ross. *Computer Networking*. Addison-Wesley, 2000.

[15] J. Elson, S. Bien, N. Busek, V. Bychkovskiy, A. Cerpa, D. Ganesan, L. Girod, B. Greenstein, T. Schoellhammer, T. Stathopoulos, and D. Estrin. EmStar: An Environment for Developing Wireless Embedded Systems Software. Technical report, CENS-TR-9, March 2003.

[16] Deepak Ganesan, Bhaskar Krishnamachari, Alec Woo, David Culler, Deborah Estrin, and Stephen Wicker. Complex Behavior at Scale: An Experimental Study of Low-Power Wireless Sensor Networks. Technical report, UCLA/CSD-TR 02-0013, 2001.

[17] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *Proceedings of IEEE INFOCOM*, 2002.