Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks

UCB//CSD-03-1290

Philip Levis, Neil Patel, Scott Shenker and David Culler
{pal,shenker,culler}@eecs.berkeley.edu, neilp@uclink.berkeley.edu
EECS Department
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

We present Trickle, an algorithm for propagating and maintaining code updates in wireless sensor networks. Trickle uses a "polite gossip" policy, where nodes periodically broadcast a code summary to local neighbors but stay quiet if they have recently heard a summary identical to theirs. When a node hears an older summary than its own, it broadcasts an update. Instead of flooding a network with packets, the algorithm controls the send rate so each node hears a small trickle of packets, just enough to stay up to date.

We first analyze Trickle using an idealized single-cell network model, with perfect synchronization and no packet loss. Progressively relaxing these assumptions, we evaluate the algorithm in simulation, first without synchronization, then in the presence of loss, and finally in the multi-cell case. We validate these simulation results with empirical data from a real-world deployment. We show that Trickle scales well, with the aggregate network transmission count increasing as a logarithm of cell density. We show that by dynamically adjusting listening periods, Trickle can rapidly propagate new code, taking on the order of seconds, while keeping maintenance costs on the order of a few sends per hour per node.

1. INTRODUCTION

Composed of large numbers of small, resource-constrained computing nodes ("motes"), sensor networks often must operate unattended for long periods of time, on the order of months or years. As requirements and environments evolve, users need to be able to introduce new code to change the operation of a network. The scale and embedded nature of these systems – buried in bird burrows or collared on roving herds of zebras – requires code be propagated over the network. Networking has a tremendous cost in terms of energy, however, and therefore limits the lifetime of the system.

Two conflicting goals arise. On one hand, changes to running code should propagate quickly and efficiently. On the other, maintaining a consistent code image (detecting when there is a change) throughout the network should have close to zero cost. These two goals are complicated by low power radios, which exhibit high loss and transient disconnection: advertisements of new code can be easily lost.

Maté, a tiny virtual machine designed for sensor networks, represents one instance of this problem [8]. In Maté, certain instructions broadcast code fragments to local neighbors. If a node hears newer code, it installs it. Maintenance and propagation are the same mechanism: code fragment transmissions. This basic primitive, epidemic code distribution, is a powerful one. However, as the authors point out, the Maté algorithm has several limitations that prevent it from being feasibly deployable, the foremost being that it easily saturates a network. In this paper, we propose Trickle, a code maintenance and propagation algorithm which addresses this problem.

Trickle draws on two major areas of prior research relevant to the problem of code maintenance and propagation, each of which assume network characteristics distinct from low-power wireless sensor networks. The first is controlled, density-aware flooding research for wireless and multicast networks [11, 4]. These flooding algorithms, based on IP, assume communication is inexpensive and that loss is present but rare. SRM, for example, is based on wired multicast; while this resembles a wireless cell in some sense (e.g., packets are broadcast to the multicast group), it also does not have all of the same complexities (e.g., the hidden terminal problem). In the wireless case, flooding can adapt to network density but is unreliable [11]. The common technique to provide reliability in wireless flooding is to route acknowledgements back to the flood root. This communication cost can far surpass that of the actual broadcast.

The second area of research is epidemic and gossiping algorithms for maintaining data consistency in wired, distributed systems [2, 3]. These approaches also have assumptions that do not hold in sensor networks. The first is that the basic communication primitive is end-to-end transport. Distant hosts can communicate end-to-end using some network or virtual naming scheme (e.g., IP or a DHT). Regulated one-to-one communication provides scalability; as every node contacts only one other, the expectation is that it is also contacted only once. In wireless networks, the communication medium means a transmission to one neighbor is effectively a transmission to all; communication is not independent.

Although both techniques, broadcasts and epidemics, have assumptions that make them inappropriate to this specific problem – code update propagation in sensor networks – they present powerful techniques that can be borrowed to solve it. An effective algorithm must adjust to local network density as controlled floods do, but continually maintain consistency in a manner similar to epidemic algorithms. Taking advantage of the broadcast nature of the medium, a sensor network can use SRM-like duplicate suppression to conserve transmission energy and precious bandwidth.

One important observation in the Maté work, and a core assumption in Trickle, is that transmitted code is generally very small; the energy cost of transmitting verbose code text demands concise representations. These code representations are not necessarily what are executed on a node. The TinyDB system, for example, encodes queries in a binary format, which it then parses into an on-node data structure for efficient execution [10]. These concise code representations mean that data and metadata can be very close in size: a query fits in three packets, while a complete description of running queries fits in one.

We have developed Trickle, a self-regulating algorithm for propagating code updates in a wireless sensor network. Trickle self-regulates using a local “polite gossip” to exchange code metadata. If a node hears gossip with the same metadata that it has, it stays quiet, instead of spreading something everyone else has already heard. When a node hears old gossip, it broadcasts a code update, so the gossiping node can be brought up to date. To achieve both rapid propagation and a low maintenance overhead, nodes adjust the length of their gossiping attention spans, communicating more often when there is new code. In this study, we present the Trickle algorithm and evaluate it in simulation as well as deployment in a working system.

In Section 2 of this paper, we outline the experimental methodologies we used in this study. In Section 3, we describe the basic primitive of Trickle and its conceptual basis. In Section 4, we present Trickle’s maintenance algorithm, evaluating its scalability with regards to network density. In Section 5, we show how the maintenance algorithm can be modified slightly to enable rapid propagation, and evaluate how quickly Trickle propagates code. We discuss our results in Section 6, review related work in Section 7, and conclude in Section 8.

2. METHODOLOGY

Wireless sensor networks operate under different constraints than traditional network domains. One source of difference is the hardware itself. Sensor nodes are heavily constrained by both communication capability and energy. Node communication hardware is often a simple radio transciever that has limited bandwidth, on the order of a few tens of Kbps. Sending a single bit of data can consume the energy of executing a thousand instructions. As opposed to measuring bandwidth in megabytes or kilobytes, every packet is precious. Energy constraints are further compounded by the fact that the networks are expected to exist unattended for long periods of time. This degree of energy conservation is usually not an issue when discussing wired and even 802.11 networks. Laptops can be recharged, but sensor networks die.

The other point of distinction is in the attributes of the communication network. Wireless sensor networks may operate at a scale of hundreds, thousands, or more. They exhibit highly transient loss patterns which are susceptible to changes in environmental conditions. Irregularities in the network such as asymmetric links often arise. Combined with the precious nature of communication, these characteristics mean that sensor networks do not rely on the same assumptions of more traditional wired and wireless networks, and therefore often require new algorithms and techniques.

In this study, we used three different platforms to investigate and evaluate our algorithm, Trickle. The first is a high-level, abstract algorithmic simulator written especially for

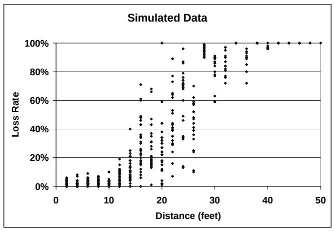


Figure 1: TOSSIM Packet Loss Rates over Distance

Figure 2: The TinyOS mica2

this study. The second is TOSSIM, a bit-level node simulator for TinyOS, a sensor network operating system [7, 9]. TOSSIM compiles directly from TinyOS code. Finally, we used TinyOS mica-2 nodes for empirical studies. The same implementation of Trickle ran in TOSSIM and on nodes.

Each of our three platforms offer useful information about the behavior of Trickle at differing levels of control. The abstract simulator allows us to quickly evaluate change in the algorithm's behavior when its tunable parameters are tweaked. At a looser level of control, TOSSIM let us test our tuned algorithm repeatedly on a network, producing a complete and precise data set for use in observing the performance of Trickle. Finally, empirical study was necessary to confirm the real-world effectiveness of Trickle, and to validate our simulation studies..

2.1 Abstract Simulation

To quickly evaluate Trickle under controlled conditions, we implemented a Trickle-specific algorithmic simulator. Little more than an event queue, it allows configuration of all of Trickle's parameters, run duration, the boot time of nodes, and a uniform packet loss rate (same for all links) across a single cell. Its output is how many packets were sent in the cell.

2.2 TOSSIM

The TOSSIM simulator compiles directly from TinyOS code, simulating complete programs from application level logic to the network at a bit level [9]. The bit-level network simulation captures the entire TinyOS stack, including its CSMA MAC protocol, its data encodings, packet CRC checks, and packet timing. The simulation models the network as a directed graph, where vertices are nodes and edges are links; each link has a bit error rate. The networking stack (based on mica hardware) can handle approximately forty packets per second, with each carrying a 36 byte payload.

TOSSIM models the network as a directed graph, where vertices are nodes and edges are links; each link has a bit error rate. To generate network topologies, we used a network model TOSSIM provides, based on empirically gathered data from TinyOS nodes [5]. Figure 1 shows an experiment demonstrating this model's packet loss rates over distance (in feet)[1]. Each link (a, b) is sampled independently from (b, a). For intermediate distances such as twenty feet, this can lead to link asymmetry, where only one direction has good connectivity. To generate a loss topology, one first generates a physical topology, from which distances are taken and fed into the loss distribution to generate a loss topology. Signal strength is uniform in a 50 foot disc; although closer nodes have a lower packet loss rate, they cannot overwhelm the signal of a further node. Link bit error rates are constant for the duration of a simulation, but packet loss rates can be effected by dynamic interactions such as the hidden terminal problem.

In addition to standard bit-level simulation, we modified TOSSIM to support packet-level simulations, which model loss due to bit errors, but not model collisions (the hidden terminal problem).

2.3 TinyOS nodes

We used TinyOS mica-2 nodes, with a 900Mhz radio [2]. These nodes provide 128KB of program memory, 4KB of RAM, and a 7MHz 8-bit microcontroller for a processor. The radio transmits at 19.2 Kbit, which after encoding and media access, is approximately 40 TinyOS packets. For propagation experiments, we instrumented mica nodes with a special hardware device that bridges their UART to TCP; other computers can connect to the node with a TCP socket to read and write data to the node. We used this functionality to obtain accurate (i.e., with a few milliseconds) timestamps

[1]Figure courtesy of TOSSIM developers at UC Berkeley.
[2]There is also a 433 MHz variety.

on events through the network. Figure 2 shows a picture of one of the mica-2 nodes used in our experiments.

We performed two empirical studies. One involved placing varying number of nodes on a table, with the transmission strength set very low to create a lossy cell. The other was a nineteen node network in an office area, approximately 160' by 40'. Section 5 presents the latter experiment in greater depth.

3. TRICKLE OVERVIEW

In the next three sections, we introduce and evaluate Trickle. In this section, we describe the basic algorithm primitive colloquially, as well as its conceptual basis. In Section 4, we describe the algorithm more formally, and evaluate the scalability of Trickle's maintenance cost, starting with an ideal case – a single lossless and perfectly synchronized cell. Incrementally, we will remove each of these three constraints, quantifying the resulting scalability in simulation and validating the simulation results with an empirical study. In Section 5, we show how, by adjusting the length of time intervals, Trickle's maintenance algorithm can be easily adapted to also rapidly propagate code while imposing a minimal overhead.

Trickle has two goals: propagation and maintenance. Propagation is to quickly install new code on nodes that need it, while maintenance is to detect that a propagation is needed. Propagation should be quick, but maintenance should approach zero cost. We say approach because, in the presence of transient and variable network loss, maintenance cannot be free; at some point nodes must communicate to see if they have the latest code. The algorithm makes two assumptions. First, it assumes reprogramming events are not common (e.g., at most every few minutes). Second, it assumes that nodes can succinctly describe their code with metadata, and by comparing two different pieces of metadata can determine which node needs an update.

Trickle's basic mechanism is simple: every so often, a node transmits its information if it has not heard a few other nodes transmit the same thing. This primitive allows Trickle to scale to thousand-fold variations in cell density, quickly propagate updates, distribute transmission load evenly, be robust to transient disconnections, handle network repopulations, and impose a low maintenance overhead on the order of a few packets per hour per node.

Trickle broadcasts all messages to the local radio cell. There are two possible cell states: either every node is up to date, or the need for an update is detected. Detection can be the result of either an old node hearing someone has new code, or a new node hearing someone has old code. As long as every node in the cell communicates somehow – either receives or transmits – the need for an update will be detected.

For example, if node A broadcasts that it has code ϕ_x, but B has code ϕ_{x+1}, then B knows that A needs an update. Similarly, if B broadcasts that it has ϕ_{x+1}, A knows that it needs an update. If B broadcasts updates, then all of its neighbors can receive them without having to advertise their need. Some of these recipients might not even have heard A's transmission.

In this example, it doesn't matter who first transmits, A or B; either one will cause the inconsistency to be detected and resolved. All that matters is that there is some rate at which nodes*communicate* with one another, either receiving or transmitting. No matter how many nodes are in a single cell, as long as there is some minimum rate of communication for each node, everyone will stay up to date.

The fact that communication can be either transmission or reception allows it to scale to high density cells; in the ideal case of a single lossless cell of arbitrary size, only one node need transmit for every node to communicate. Keeping the communication rate independent of density allows Trickle to scale to any network topology, as the amount of bandwidth consumed is independent of the cell size. This is an important behavior in wireless networks, where the channel itself is a valuable shared resource. Maintaining a constant communication rate also conserves overall cell energy – in dense cells, individual nodes must transmit fewer packets to stay up to date.

We begin in Section 4 by describing Trickle's maintenance algorithm, which tries to keep a constant node communication rate. We analyze its performance (in terms of transmissions and communication) in the idealized case of a single, lossless network cell with perfect time synchronization. We then relax each of these assumptions, showing how it changes the behavior of Trickle, and, in the case of synchronization, modify the algorithm slightly to accommodate.

4. MAINTENANCE

Trickle uses "polite gossip" to exchange code metadata with nearby network neighbors. It breaks time into intervals, and at a random point in each interval, it considers broadcasting its code metadata. If Trickle has already heard several other nodes gossip the same metadata in this interval, it

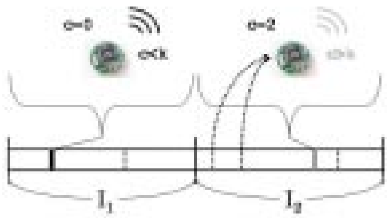


Figure 3: Trickle Maintenance with a k of 1. *Dark boxes are transmissions, grey boxes are suppressed transmissions, and dotted lines are heard transmissions. Both I_1 and I_2 are of length τ.*

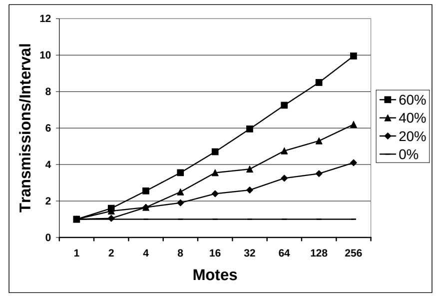


Figure 4: Number of Transmissions as Cell Density Increases for Different Packet Loss Rates

politely stays quiet: repeating what someone else has said is rude.

When a node hears that a neighbor is behind the times (it hears older metadata), it brings everyone nearby up to date by broadcasting the needed pieces of code. When a node hears that it is behind the times, it repeats the latest news it knows of (its own metadata); following the first rule, this triggers nodes with newer code to broadcast it.

More formally, each node maintains a counter c, a threshold k, and a timer t in the range $[0, \tau]$. k is a small, fixed integer (e.g., 1 or 2) and τ is a time constant (to be varied later). When a node hears metadata identical to its own, it increments c. At time t, the node broadcats a summary of its program if $c < k$. When the interval of size τ completes, c is reset to zero and t is reset to a new value in the range $[0, \tau]$. If a node with code ϕ_x hears a summary for ϕ_{x-y}, it broadcasts the code necessary to bring ϕ_{x-y} up to ϕ_x. If it hears a summary for ϕ_{x+y}, it broadcasts its own summary, triggering the node with ϕ_{x+y} to send updates.

Figure 3 has a visualization of Trickle in operation for two intervals of length τ with a k of 1 and no new code. In the first interval, I_1, the node does not hear any transmissions before its point t, and broadcasts. In the second interval, I_2, it hears two broadcasts of metadata identical to its, and so suppresses its broadcast.

Using the Trickle algorithm, each node broadcasts a summary of its data at most once per period τ. If a node hears k nodes with the same program before it transmits, it suppresses its own transmission. In perfect network conditions – lossless, non-interfering cells – there will be k transmissions every τ in each cell, regardless of density. If there are n nodes and m cells, there will be km transmissions, which is independent of n. Instead of fixing the per-node send rate, Trickle dynamically regulates its communication rate, adjusting the send rate to the network density, requiring no a priori assumptions on the topology. In each interval τ, the sum of receptions and sends of each node is k.

The random selection of t uniformly distributes the choice of who broadcasts in a given interval. This evenly spreads the transmission energy load across the network. If a node in a cell needs an update, the expected latency to discover this is $\frac{\tau}{n+1}$. This either happens because the node transmits its summary, which will cause others to send updates, or because another node transmits a newer summary. A large τ has a lower energy overhead (in terms of packet send rate), but also has a higher discovery latency. Conversely, a small τ sends more messages but discovers updates more quickly.

The suppression mechanism is very similar to (and is inspired by) the one used in SRM [4]. However, SRM is based on the idea of an IP multicast group. A message, when transmitted, will reach (barring loss) every node in the multicast group. SRM's choice of timers is based on estimates of transmission latency; in contrast, since Trickle's messages are local wireless broadcasts, they arrive at essentially the same time, but Trickle must propagate across many cells.

This km transmission count behavior depends on three assumptions: no packet loss, perfect interval synchronization, and independent cells. We visit and then relax each of these assumptions in turn. Discussing each assumption seperately allows us to examine the effect of each, and in the case of interval synchronization, will help us in making a slight modification to restore scalability.

4.1 Maintenance with Loss

The above results assume that a node hears every transmission made in its cell; in real-world sensor networks, this is rarely the case. Figure 4 shows how packet loss rates af-

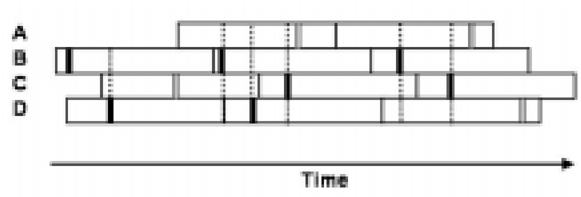


Figure 5: The Short Listen Problem. *Dark bars represent transmissions, light bars suppressed transmissions, and dashed lines are receptions. Note that node B transmits in all three intervals.*

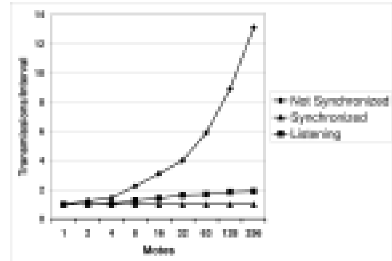


Figure 6: The Short Listen Problem's Effect on Scalability, $k = 1$. *Without synchronization, Trickle scales with $O(\sqrt{n})$. A listening period restores this to asymptotically bounded by a constant.*

fect the number of Trickle transmissions per interval in a cell as density increases. These results are from our abstract simulator, with a k of 1. Each line is a uniform packet loss rate across the cell. For a given loss rate, the number of transmissions grows with $O(log(n))$ with cell density.

This logarithmic behavior represents the probability that a single node misses a number of transmissions. For example, with a 10% loss rate, there is a 10% chance a node will miss a single packet. If a node misses a packet, it will transmit, resulting in two transmissions for the cell. There is correspondingly a 1% chance it will miss two, leading to three transmissions, and a 0.1% chance it will miss three, leading to four transmissions. In the extreme case of a 100% loss rate, each node is in a singleton cell: transmissions increase linearly.

Unfortunately, to maintain a per-interval minimum communication rate, this logarithmic scaling is inescapable. The increase in communication represents satisfying the requirements of the worst case node in a cell; in order to do so, the expected case must transmit a little bit more. Some nodes don't hear the gossip the first time someone says it, and need it repeated.

4.2 Maintenance without Synchronization

The above results assume that all nodes have synchronized intervals. Inevitably, time synchronization imposes a communication, and therefore energy, overhead. While some networks can provide time synchronization to Trickle, others cannot. Therefore, Trickle should be able to work in the absence of this primitive.

Unfortunately, without synchronization, Trickle can suffer from the *short-listen* problem. Some subset of nodes gossip soon after the beginning of their interval, listening for only a short time, before anyone else has a chance to speak up. If all of the intervals are synchronized, the first gossip will quiet everyone else. However, if not synchronized, the gossiping could occur just before another node begins an interval in which it might also listen for a short period.

Figure 5 shows an instance of this phenomenon. In this example, node B selects a small t on each of its three intervals. Although other nodes transmit, node B never hears those transmissions before its own, and its transmissions are never suppressed. Figure 6 shows how the short-listen problem effects the transmission rate in a lossless cell with $k = 1$. A perfectly synchronized cell scales perfectly, with a constant number of transmissions. In a cell without any synchronization between intervals, however, the number of transmissions per interval increases significantly.

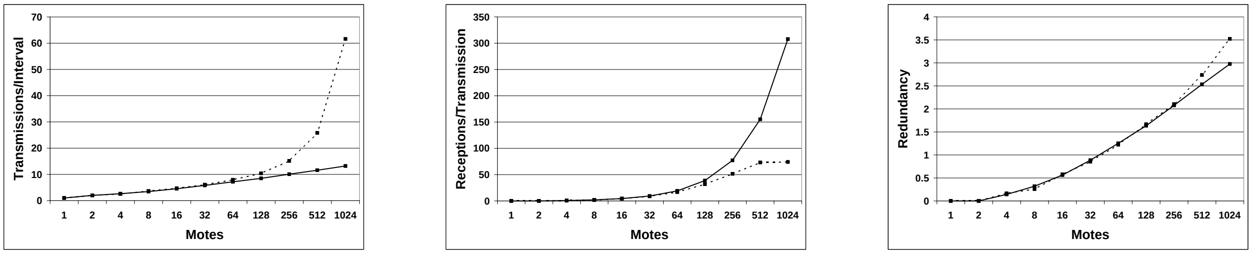
The short-listen problem causes the number of transmissions to scale at $O(\sqrt{n})$ with cell density. Unlike with loss, where extra $O(log(n))$ transmissions are sent to keep the worst case node up to date, the additional transmissions due to a lack of synchronization are completely redundant, and represent avoidable inefficiency.

O($\sqrt{n}$): Assume the network of n nodes with an interval of size τ is in a steady state. If interval skew is uniformly distributed, then in an interval of length of $\frac{\tau}{n}$, the expectation is that one node will start its interval. For time t after a transmission, $\frac{nt}{\tau}$ will have started their intervals. For a node that starts at time $t - k$, the probability its transmission time will be before t is $\frac{k}{\tau}$. From this, we can compute the expected time after a transmission that another transmission will occur. This is when

$$\cdot \prod_{t=0}^{n} \left(1 - \frac{t}{n}\right)$$

is less than 50%, which is when $t \approx \sqrt{n}$, that is, when $\frac{\sqrt{n}}{\tau}$ time has passed. There will therefore be $\sqrt{n}$ transmissions.

To remove the short-listen effect, we modified Trickle slightly.



(a) Total Transmissions per Interval (b) Receptions per Transmission (c) Redundancy

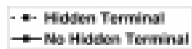


Figure 8: Simulated Trickle Scalability for Multiple Cells with Increasing Density. *nodes were uniformly distributed in a 50'x50' square area.*

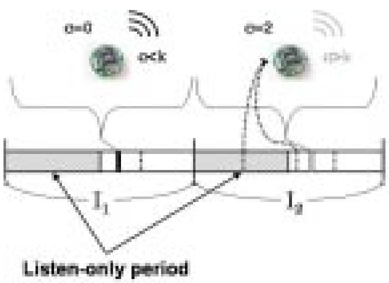


Figure 7: Trickle Maintenance with a k of 1 and a Listen-Only Period. *Dark boxes are transmissions, grey boxes are suppressed transmissions, and dotted lines are heard transmissions***.**

Instead of picking a t in the range $[0, \tau]$, t is selected in the range $[\frac{\tau}{2}, \tau]$, defining a "listen-only" period of the first half of an interval. Figure 7 depicts the modified algorithm. A listening period improves scalability by enforcing a simple constraint. If, whenever a message is sent in the cell, one can guarantee a silent period of some time T that is independent of the density, then the sending rate in the cell is bounded above (independent of the density). When a node transmits, it suppresses all other nodes for at least the length of the listening period. With a listen period of $\frac{\tau}{2}$, it bounds the total sends in a lossless cell to be $2k$, and a lossy cell scales as $2k \cdot log(n)$.

The "Listening" line in Figure 6 shows the number of transmissions in a cell with no synchronization when Trickle uses this listening period. As cell density increases, the number of transmissions per interval asymptotically approaches two. The listening period does not harm performance when the cell is synchronized: there are k transmissions, but they are all in the second half of the interval.

One possible conclusion from this is that nodes should listen for their entire interval, and decide whether to transmit at the end. However, this breaks down if the network happens to be synchronized; all nodes will hear silence and then try to transmit at once. To work properly, Trickle needs a source of randomness; this can come from either the selection of t or from a lack of synchronization. By using both, it works in either circumstance, or any point between the two (e.g., partial or loose synchronization).

4.3 Maintenance with Multiple Cells

To understand Trickle's behavior in the presence of multiple overlapping network cells, we used TOSSIM, randomly placing nodes in a 50'x50' area with a uniform distribution. To capture the effect of the hidden terminalproblem, we ran TOSSIM as both a packet-level simulation that does not model collision, and as a bit-level simulation that does. Drawing from the loss distributions in Figure 1, a 50'x50' grid will have several cells. Figure 8 shows the results of this experiment.

Figure 8(a) shows how the number of transmissions per interval scales as the number of nodes increases. In the absence of the hidden terminal problem, Trickle scales as expected, at $O(log(n))$. This is also true with the hidden terminal problem for low to medium densities; however, once there is over 128 nodes, the number of transmissions in-

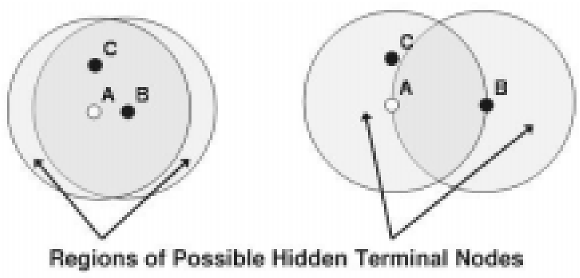


Figure 9: **The Effect of Proximity on the Hidden Terminal Problem.** *When C is within range of both A and B, CSMA will prevent C from interfering with transmissions between A and B. But when C is in range of A but not B, B might start transmitting without knowing that C is already transmitting, corrupting B's transmission. Note that when A and B are farther apart, the region where C might cause this "hidden terminal" problem is larger.*

creases significantly.

This result is troubling – it suggests that Trickle cannot scale to very dense networks. However, this turns out to be a limitation of the TinyOS's CSMA, and not Trickle itself. Figure 8(b) shows the average number of receptions per transmission for the same experiments. Without packet collisions, as network density increases exponentially, so does the reception ratio. However, in the presence of the hidden terminal, the reception/transmission ratio plateaus around seventy-five.

Many packets are being lost due to collisions from the hidden terminal problem. In the perfect cell scaling model, the number of transmissions for m isolated and independent cells is mk. In a network, there is a number of *physical* cells (defined by the radio range), but the hidden terminal problem causes there to be a much larger number of *effective* cells, as cell sizes are smaller than ideal due to loss. A physical cell represents who can hear a transmission in the absence of any other traffic, while an effective cell is a function of other, possibly conflicting, traffic in the network. Increasing network density increases the number of effective cells m, and correspondingly, the number of transmissions (mk).

The set of nodes in a node's effective cell is tied to physical proximity. The set of nodes that can interfere with communication by the hidden terminal problem is larger when two nodes are far away than when they are close. Figure 9 depicts this relationship.

From each node's perspective in the 512 and 1024 node experiments, it is in a cell of 75 nodes. This does not change significantly as density increases. Returning to Figure 8(b), adding more nodes to the area increases the number of effective cells, therefore increasing the number of transmissions.

To better understand Trickle in multi-cell networks, we use the metric of *redundancy*. Redundancy is the portion of messages heard in an interval that were unnecessary communication. Specifically, it is each node's expected value of $\frac{c+s}{k} - 1$, where s is 1 if the node transmitted and 0 if not. A redundancy of 0 means Trickle works perfectly; every node communicates k times. A redundancy of 1.5 means there were $1.5k$ redundant communications. The maintenance overhead is independent of density, but the update latency is inversely proportional to density.

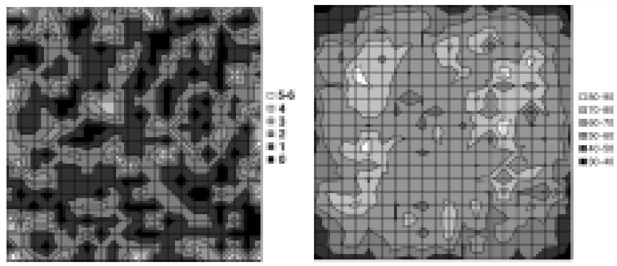
The redundancy can be easily computed for the single cell experiments with uniform loss (Figures 4 and 6). For example, in a cell with a uniform 20% loss rate, 3 transmissions/interval has a redundancy of 1.4 ($(3 \cdot 0.8) - 1$), as the expectation is that each node receives 2.4 packets, and three nodes transmitted.

Figure 8(c) shows a plot of Trickle redundancy as network density increases. For a one-thousand node network – larger than any wireless sensor network yet deployed – with multiple cells, in the presence of link asymmetry, highly variable packet loss, and the hidden terminal problem, the redundancy grows to be just over 3. This redundancy grows with a simple logarithm of the cell size, and is due to the simple problem outlined in Section 4.1: packets are lost. To maintain a communication rate for the worst case node, the average case must communicate a little bit more. Although the communication increases, the actual per-node transmission rate shrinks. The presence of multiple, overlapping cells does not disrupt Trickle's scalability.

4.4 Load Distribution

One of the goals of Trickle is to impose a low overhead. The above simulation results show that few packets are sent in a network. However, this raises the question of which nodes sent those packets; 500 transmissions evenly distributed over 500 nodes does not impose a high cost, but 500 messages by one node does.

Figure 10(a) shows the transmission distribution for a simulated 400 node network in a 20 node by 20 node grid with a 5 foot spacing (the entire grid was 95'x95'), run in TOSSIM. Drawing from the empirical distributions in Figure 1, a five foot spacing forms a six hop network from grid corner to corner. This simulation was run with a τ of one minute, and ran for twenty minutes of virtual time. The topology shows that



(a) Transmissions (b) Receptions

Figure 10: Communication Topography of a Simulated 400 node Network in a 20x20 Grid with 5 foot spacing (95'x95').

some nodes send more than others, in a mostly random pattern. Given that the predominant range is one, two, or three packets, this non-uniformity is easily attributed to statistical variation. A few nodes show markedly more transmissions, for example, six. This is the result of some nodes just being poor receivers. If many of their incoming links have high loss rates (drawn from the distribution in Figure 1), they will perceive a small cell size, as they receive few packets.

Figure 10(b) shows the reception distribution. Unlike the transmission distribution, this shows clear patterns. nodes toward the edges and corners of the grid receive fewer packets than those in the center. This is due to the non-uniform network density; a node at a corner has one quarter the neighbors as one in the center. Additionally, a node in the center has more neighbors that cannot hear one another; it straddles multiple cells, so that a transmission in one will not suppress a transmission in another. In constrast, almost all of the nodes in a corner node's cell can hear one another. Although the transmission topology is quite noisy, the reception topography is smooth. The number of transmissions is very small compared to the number of receptions: the communication rate across the network is fairly uniform.

4.5 Empirical Study

To evaluate Trickle's scalability in a real network, we recreated, as best we could, the experiments shown in Figures 6 and 8. We placed nodes on a small table, with their transmission signal strength set very low, making the table a noisy cell. With a τ of one minute, we measured Trickle efficiency over a twenty minute period for increasing numbers of nodes.

Figure 11 shows the results. Although much bumpier than

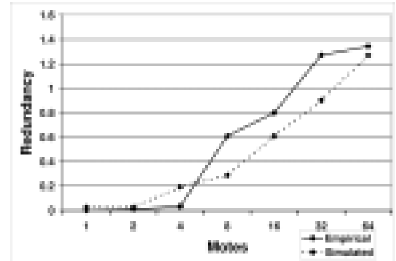


Figure 11: Empirical and Simulated over Density. *The simulated data is the same as Figure 8.*

Event	Action
τ Expires	Double τ, up to τ_h. Reset c, pick a new t.
t Expires	If $c < k$, transmit.
Receive same metadata	Increment c.
Receive newer metadata	Set τ to τ_l. Reset c, pick a new t.
Receive newer code	Set τ to τ_l. Reset c, pick a new t.
Receive older metadata	Send updates.

t is picked from the range $[\frac{\tau}{2}, \tau]$

Figure 12: Trickle Pseudocode.

the results from TOSSIM, they show similar scaling. For example, the TOSSIM results in Figure 8(c) show a 64 node network having an redundancy of 1.1; the empirical results show 2.35. These were by no means under identical network conditions; the noise inherent in real world sensor network systems makes any equivalence with TOSSIM impossible. Among other things, the base of the scaling logarithm comes from the network loss rate. However, the empirical results show that maintenance scales as the simulation results indicate it should: logarithmically.

The above results quantified the maintenance overhead. Evaluating propagation requires an implementation; among other things, there must be code to propagate. It also require more complex tools that can model multihop sensor networks. In the next section, we present an implementation of Trickle, evaluating it in simulation and empirically.

5. PROPAGATION

A large τ has a low communication overhead, but slowly propagates information. Conversely, a small τ has a higher communication overhead, but propagates more quickly. These two goals, rapid propagation and low overhead, are fundamentally at odds: the former requires communication to be frequent, while the latter requires it to be infrequent.

By dynamically scaling τ, Trickle can use its maintenance algorithm to rapidly propagate updates with a very small cost. τ has a lower bound, τ_l and an upper bound τ_h. When

τ expires, it doubles, up to τ_h. When a node hears a summary with newer data than it has, it resets τ to be τ_l. When a node hears a summary with older code than it has, it sends the code, to bring the other node up to date. When a node installs new code, it resets τ to τ_l, to make sure that it spreads quickly. This is necessary for when a node receives code it did not request, that is, didn't reset its τ for. Figure 12 shows pseudocode for this complete version of Trickle.

Essentially, when there's nothing new to say, nodes gossip infrequently: τ is set to τ_h. However, as soon as a node hears something new, it gossips more frequently, so those who haven't heard it yet find out. The chatter then dies down, as τ grows from τ_l to τ_h.

By adjusting τ in this way, Trickle can get the best of both worlds: rapid propagation, and low maintenance overhead. The cost of a propagation event, in terms of additional sends caused by shrinking τ, is approximately $log(\frac{\tau_h}{\tau_l})$. For a τ_l of one second and a τ_h of one hour, this is a cost of eleven packets per cell to obtain a three-thousand fold increase in propagation rate (or, from the other perspective, a three thousand fold decrease in maintenance overhead). The simple Trickle policy, "every once in a while, transmit unless you've heard a few other transmissions," can be used both to inexpensively maintain code and quickly propagate it.

We evaluate an implementation of Trickle, incorporated into SNLR (Sensor Network Language Runtime), a dynamic runtime for TinyOS sensor networks. We first present a brief overview of SNLR and its Trickle implementation. Using TOSSIM, we evaluate how how rapidly Trickle can propagate an update through reasonably sized (i.e., 400 node) networks of varying density. We then evaluate Trickle's propagation rate in a small (20 node) real-world network.

5.1 SNLR, a Trickle Implementation

SNLR (the Sensor Network Language Runtime) has a small, static set of code routines. Each routine can have many versions, but the runtime only keeps the most recent one. By replacing these routines, a user can update a network's program. Each routine fits in a single packet and has a version number. The runtime installs routines with a newer version number when it receives them.

Instead of sending entire routines, nodes can broadcast version summaries. A version summary contains the version numbers of all of the routines currently installed. A node determines that someone else needs an update by seeing who (if anyone) has a newer version.

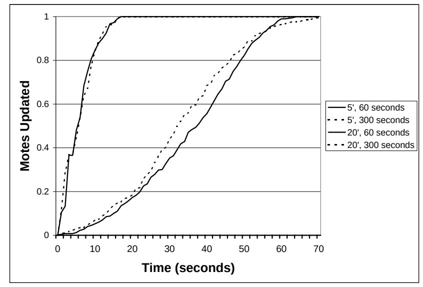


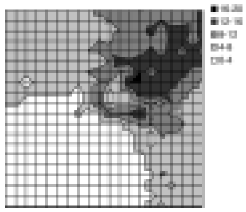
Figure 13: Simulated Code Propagation Rate for Different τ_hs.

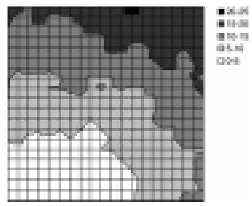
SNLR uses Trickle to periodically broadcast version summaries. In all experiments, code routines fit in a single TinyOS packet (30 bytes). The runtime registers routines with a propagation service, which then maintains all of the necessary timers and broadcasts, notifying the runtime when it installs new code. The actual code propagation mechanism is outside the scope of Trickle, but we describe it here for completeness. When a node hears an older vector, it broadcasts the missing routines three times: one second, three seconds, and seven seconds after hearing the vector. If code transmission redundancy were a performance issue, it could also use Trickle's suppression mechanism. For the purpose of our experiments, however, it was not.

The SNLR implementation maintains a 10Hz timer, which it uses to increment a counter. t and τ are represented in ticks of this 10Hz clock. Given that the current node platforms can transmit on the order of 40 packets/second, we found this granularity of time to be sufficient. If the power consumption of maintaining a 10Hz clock were an issue (as it may be in some deployments), a non-periodic implementation could be used instead.

5.2 Simulation

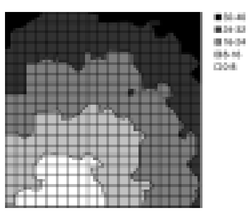
We used TOSSIM to observe the behavior of Trickle during a propogation event. We ran a series of simulations, each of which had 400 nodes regularly placed in a 20x20 grid, and varied the spacing between nodes. By varying network density, we could examine how Trickle's propagation rate scales over different loss rates and cell sizes. Density ranged from a five foot spacing between nodes up to twenty feet (the networks were 95'x95' to 380'x380'). We set τ_l to one second and τ_h to one minute. From corner to corner, these topologies range from six to forty hops. These hopcount measure-

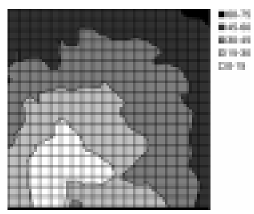




(a) 5’ Spacing, 6 hops

(b) 10’ Spacing, 16 hops





(c) 15’ Spacing, 32 hops

(d) 20’ Spacing, 40 hops

Figure 14: Simulated Time to Code Propagation Topography in Seconds. *Hopcounts are the expected number of transmissions necessary to get from corner to corner.*

ments come from computing the minimum cost path across the network loss topology, where each link has a weight of $\frac{1}{1-loss}$, or the expected number of transmissions to successfully traverse that link.

The simulations ran for five virtual minutes. Nodes booted with randomized times in the first minute, selected from a uniform distribution. After two minutes, a node near one corner of the grid advertised a new SNLR routine. We measured the propagation time (time for the last node to install the new routine from the time it first appeared) as well as the topographical distribution of routine installation time. The results are shown in Figures 13 and 14. Time to complete propagation varied from 16 seconds in the densest network to about 70 seconds for the sparsest. Figure 13 shows curves for only the 5’ and 20’ grids; the 10’ and 15’ grid had similar curves.

Figure 14(a) shows a manifestation of the hidden terminal problem. This topography doesn’t have the wave pattern we see in the experiments with sparser netorks. Because the network was only a few hops in area, nodes near the edges of the grid were able to receive and install the new capsule quickly, causing their subsequent transmissions to collide in the upper right corner. In contrast, the sparser networks exhibited a wave-like propagation because the sends mostly came from a single direction throughout the propagation event.

Figure 13 shows how adjusting τ_h changes the propagation time for the five and twenty foot spacings. Increasing τ_h from one minute to five does not significantly affect the propagation time; indeed, in the sparse case, it propagates faster until roughly the 95th percentile. This result indicates that there may be little tradeoff between the maintenance overhead of Trickle and its effectiveness in the face of a propagation event.

A very large τ_h can increase the time to discover inconsistencies to be approximately $\frac{\tau_h}{2}$. However, this is only true when two stable subnets ($\tau = \tau_h$) with different code reconnect. If new code is introduced, it immediately triggers nodes to τ_l, bringing the network to action.

5.3 Empirical Study

As Trickle was implemented as part of SNLR, several other services run concurrently with it. The only one of possible importance is the ad-hoc routing protocol, which periodically sends out network beacons to estimate link qualities. However, as both Trickle packets and these beacons are very infrequent compared to channel capacity (e.g., at most 1 packet/second), this does not represent a significant source of noise.

We deployed a nineteen node network in an office area, approximately 160’ by 40’. We instrumented fourteen of the nodes with the TCP interface described in Section 2, for precise timestamping. When SNLR installed a new piece of code, it sent out a UART packet; by opening sockets to all of the nodes and timestamping when this packet is received, we can measure the propagation of code over a distributed area.

Figure 16 shows a picture of the office space and the placement of the nodes. nodes 4, 11, 17, 18 and 19 were not instrumented; nodes 1, 2, 3, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, and 20 were. Node 16 did not exist.

As with the above experiments, Trickle was configured with a τ_l of one second and a τ_h of one minute. The experiments began with the injection of a new piece of code through a TinyOS GenericBase, which is a simple bridge between a PC and a TinyOS network. The GenericBase broadcast the new piece of code three times in quick succession. We then logged when each node had received the code update, and calculated the time between the first transmission and installation.

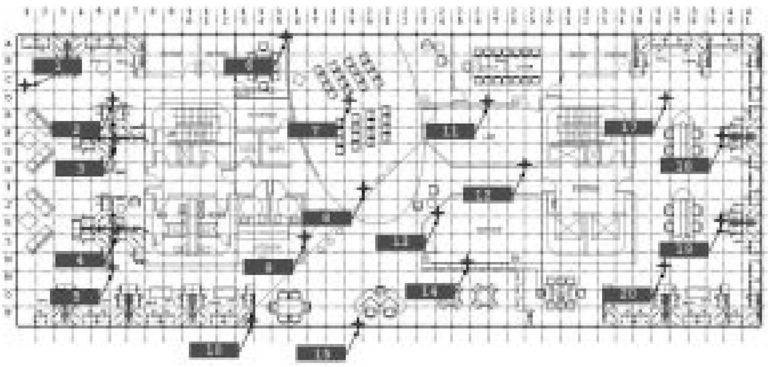
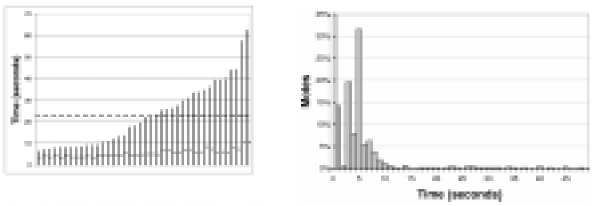
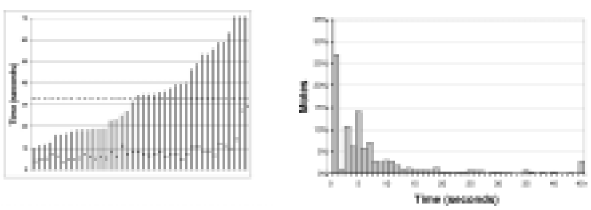


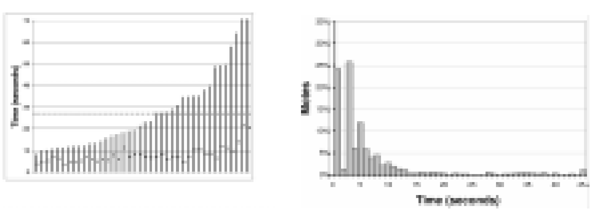
Figure 16: Empirical Testbed



(a) τ_h of 1 minute, $k = 1$



(b) τ_h of 20 minutes, $k = 1$



(c) τ_h of 20 minutes, $k = 2$

Figure 15: Empirical Network Propagation Time. *The graphs on the left show the time to complete reprogramming for 40 experiments, sorted with increasing time. The graphs on the right show the distribution of individual node reprogramming times for all of the experiments.*

The left hand column of Figure 15 shows the results of these experiments. Each bar is a separate experiment (40 in all). The worst-case reprogramming time for the instrumentation points was just over a minute; the best case was about seven seconds. The average, shown by the dark dotted line, was just over twenty-two seconds for a τ_h of sixty seconds (Figure 15(a)), while it was thirty-two seconds for a τ_h of twenty minutes (Figure 15(b)).

The right hand column of Figure 15 shows a distribution of the time to reprogramming for individual nodes across all the experiments. This shows that almost all nodes are reprogrammed in the first ten seconds: the longer times in Figure 15 are from the very long tail on this distribution. The high loss characteristics of the node radio, combined with t's exponential scaling, make this an issue. When scaling involves sending only a handful (e.g., $log_2(60)$) of packets in a single cell in order to conserve energy, long tails are inevitable.

One interesting observation in Figure 15 is that very few nodes reprogram between one and two seconds after code is introduced. This is an artifact of the granularity of the timers used, the capsule propagation timing, and the listening period. Essentially, from the first broadcast, three timers expire: $[\frac{\tau l}{2}, \tau_l]$ for nodes with the new code,$[\frac{\tau l}{2}, \tau_l]$ for nodes saying they have old code, then one second before the first capsule is sent. This is approximately $2 \cdot \frac{\tau_l}{2} + 1$; with a τ_l of one second, this latency is two seconds.

5.4 State

The SNLR implementation of Trickle requires few system resources. It requires approximately seventy bytes of RAM; half of this is a message buffer for transmissions, a quarter is pointers to the SNLR routines. Trickle itself requires only eleven bytes for its counters; the remaining RAM is used by coordinating state such as pending and initialization flags. The executable code is 2.5 KB; TinyOS's inlining and optimizations can reduce this by roughly 30%, to 1.8K. The algorithm requires few CPU cycles, and can operate at a very low duty cycle.

6. DISCUSSION

A tradeoff emerges between energy overhead and reprogramming rate. By using a dynamic transmission rate, Trickle achieves a reprogramming rate comparable to frequent transmissions while keeping overhead very low. However, as Figure 15 shows, the exact relationship between constants such as τ_h and k is unclear in the context of these high loss networks.

In this study, we have largely ignored the actual policy used to propagate code once Trickle detects the need to do so: SNLR merely broadcasts code routines three times. There's little reason that Trickle suppression techniques couldn't be used in this situation, to control the rate of code transmission. In the current SNLR implementation, the blind code broadcast is a form of localized flood; Trickle acts as a flood control protocol. This behavior is almost the inverse of protocols such as SPIN [6], which transmits metadata freely but controls data transmission.

One limitation of Trickle is that it currently assumes nodes are always on. To conserve energy, long-term node deployments often have very low duty cycles (e.g., 1%). Correspondingly, nodes are rarely awake, and rarely able to receive messages. TDMA schemes can reserve time slots for code communication, and incorporating TDMA scheduling into Trickle algorithms would allow it to operate in low duty cycle networks. Essentially, the Trickle time intervals become logical time, spread over all of the periods nodes are actually awake. Understandably, this might require alternative tunings of τ_h and k.

Trickle was designed as a code propagation mechanism over an entire network, but it could have greater applicability. Trickle's applicability can certainly go beyond code propagation; it could theoretically be used to disseminate any sort of data. Additionally, one could change propagation scope by adding predicates to summaries, limiting the set of nodes that consider them. For example, by adding a "hop-count" predicate to local routing data, summaries of a node's routing state could reach only two-hop network neighbors of the summary owner; this could be used to propagate copies of node-specific information.

7. RELATED WORK

Prior work in network broadcasts has dealt with a different problem than the one we tackle: delivering a piece of data to as many nodes as possible within a certain time period. Early work showed that in wireless networks, simple broadcast retransmission could easily lead to the broadcast storm problem [11], where competing broadcasts saturate the network. This observation led to work in probabilistic broadcasts [13], and adaptive dissemination [6]. Just as with earlier work in bimodal epidemic algorithms [1], all of these algorithms approach the problem of making a best-effort attempt to send a message to all nodes in a network.

This is insufficient for our needs. For example, it is not clear what happens if a node reconnects three days after a broadcast is sent. For configurations or code, the new node should be brought up to date, but the only way to do so is periodically rebroadcast to the entire network. This imposes a significant cost on the entire network. In contrast, our algorithms locally distribute the data where needed.

The problem of propagating data updates through a distributed system has similar goals, but prior work has been based on traditional wired network models. Demers et al. proposed the idea of using epidemic algorithms for managing replicated databases [3], while the PlanetP project [2] uses epidemic gossiping fo

dex. Our techniques and mechanisms draw from these efforts. However, while traditional gossiping protocols use unicast links to a random member of a neighbor set, our algorithms use the broadcast medium to communicate with neighbors.

The notion of a "polite gossip" can be traced to the request/repair algorithm used in SRM, another system built on a wired IP model. The work focused on reliable delivery of data through a multicast network. Trickle borrows many of its suppression techniques from SRM, adapting them to the domain of sensor networks. SRM, using IP multicast as a primitive, has a single cell of communication in which latency is a concern.

As Trickle is based on this polite gossiping through the exchange of metadata, it is reminiscent of SPIN's three-way handshaking protocol. Specifically, Trickle is similar to SPIN-RL, which works in broadcast environments and provides reliability in a lossy network. Trickle takes has three distinctions from SPIN. First, SPIN-RL suppresses redundant data, but not metadata; their results show over 95% of metadata exchanges to be redundant. SPIN assumes metadata transmissions are inexpensive in comparison to data; in node-based wireless sensor networks, often the two have comparable cost and metadata is the real cost. Second, although SPIN-RL improves data reliability over basic flooding, it transmits just as much redundant data. Third, the SPIN work notes that improved reliability can be provided by periodically re-advertising metadata, but does not suggest a policy for doing so.

Ni et al. propose a counter-based algorithm to prevent the broadcast storm problem by suppressing retransmissions [11]. This algorithm operates on a single interval, instead of continuously. As results in Figure 15 show, the transient loss rates in the class of wireless sensor network we study preclude a single interval from being sufficient. Additionally, their studies were on lossless, disc-based network topologies; it is unclear how they would perform in the sort of connectivity endemic to sensor networks.

In the sensor network space, Reijers proposes energy-efficient code distribution by only distributing changes to currently running code [12]. The work focusses on developing an efficient technique to detect and update changes to a node's code image through node memory manipulation, but does not address the question of how to distribute the code updates in a network or how to validate the network's consistency.

The Maté VM depends on epidemic code distribution [8]. One of the results in this work, however, is that the VM's algorithm (imperative forwarding through a VM bytecode) does not scale; cells easily saturate, making propagation time increase tremendously. Maté, and other similar systems, can benefit from the same algorithms that SNLR uses.

The TinyDB sensor network query system uses an epidemic style of code forwarding [10]. However, it depends on periodic data collection with embedded metadata. Every tuple routed through the network has a query ID associated with it and when a node hears a new query it requests it. In this case, the metadata has no cost, as it would be required anyways: this is not always the case. Also, this approach does not handle event-driven queries for rare events well; the query propagates when the event occurs, which may cause some nodes to miss the event.

8. CONCLUSION

Using listen periods and dynamic τ values, Trickle meets the requirements set out in Section 1. It can quickly propagate new code into a network, while imposing a very small overhead. It does so using a very simple mechanism, and requires very little state. Scaling logarithmically with cell size, it can be used effectively in very dense networks. In one of our empirical experiments, Trickle imposes an overhead of less than three packets per hour, but reprograms the entire network in thirty seconds, with no effort from an end user.

As sensor networks move from research to deployment, from laboratory to the real world, issues of management and reconfiguration will grow in importance. We have identified what we believe to be a core networking primitive in these systems, update distribution, and designed a scalable, lightweight algorithm to provide it.

9. REFERENCES

[1] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems (TOCS)*, 17(2):41–88, 1999.

[2] F. M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen. PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities. Technical Report DCS-TR-487, Department of Computer Science, Rutgers University, Sept. 2002.

[3] A. Demers, D. Greene, C. Hauser, W. Irish, and J. Larson. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM Press, 1987.

[4] S. Floyd, V. Jacobson, S. McCanne, C.-G. Liu, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 342–356. ACM Press, 1995.

[5] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker. An empirical study of epidemic algorithms in large scale multihop wireless networks, 2002. Submitted for publication, February 2002.

[6] W. R. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proceedings of the fifth annual ACM/IEEE international conference on Mobile computing and networking*, pages 174–185. ACM Press, 1999.

[7] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000. TinyOS is available at http://webs.cs.berkeley.edu.

[8] P. Levis and D. Culler. Maté: a Virtual Machine for Tiny Networked Sensors. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2002.

[9] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Simulating large wireless sensor networks of tinyos motes. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, 2003.

[10] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proceedings of the ACM Symposium on Operating System Design and Implementation (OSDI)*, Dec. 2002.

[11] S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu. The broadcast

storm problem in a mobile ad hoc network. In *Proceedings of the fifth annual ACM/IEEE international conference on Mobile computing and networking*, pages 151–162. ACM Press, 1999.

[12] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *Proceedings of the Second ACM International Workshop on Wireless Sensor Networks and Applications (WSNA '03)*, 2003.

[13] Y. Sasson, D. Cavin, and A. Schiper. Probabilistic broadcast for flooding in wireless networks. Technical Report IC/2002/54, 2002.