The Emergence of a Networking Primitive in Wireless Sensor Networks

Philip Levis, Eric Brewer, David Culler, David Gay, Sam Madden, Neil Patel, Joe Polastre, Scott Shenker, Robert Szewczyk, and Alec Woo

Abstract

The wireless sensor network community approached networking abstractions as an open question, allowing answers to emerge with time and experience. The Trickle algorithm has become a basic mechanism used in numerous protocols and systems. Trickle brings nodes to eventual consistency quickly and efficiently while remaining remarkably robust to variations in network density, topology, and dynamics. Instead of flooding a network with packets, Trickle uses a "polite gossip" policy to control send rates so each node hears just enough packets to stay consistent. This simple mechanism enables Trickle to scale to thousand-fold changes in network density, reach consistency in seconds, and require only a few bytes of state yet impose a maintenance cost of a few sends an hour. Originally designed for disseminating new code, experience has shown Trickle to have much broader applicability, including route maintenance and neighbor discovery. This paper provides an overview of the research challenges wireless sensor networks face, describes the Trickle algorithm, and outlines several ways it is used today.

1 Wireless Sensor Networks

Although embedded sensing applications are extremely diverse, ranging from habitat and structural monitoring to vehicle tracking and shooter localization, the software and hardware architectures used by these systems are surprisingly similar. The typical architecture is embodied by the mote platforms, such as those shown in Figure 1. A micro-controller provides processing, program ROM, and data RAM, as well as analog-to-digital converters for sensor inputs, digital interfaces for connecting to other devices, and control outputs. Additional flash storage holds program images and data logs. A low power CMOS radio provides a simple link layer. Support circuitry allows the system to enter a low-power sleep state, wake quickly, and respond to important events.

Four fundamental constraints shape wireless embedded system and network design: power supply, limited memory, the need for unattended operation, and the lossy and transient behavior of wireless communication. A typical power envelope for operating on batteries or harvesting requires a 600μW average power draw, with 1% of the time spent in a 60mW active state and the remainder spent in a very low power 6μW passive state.

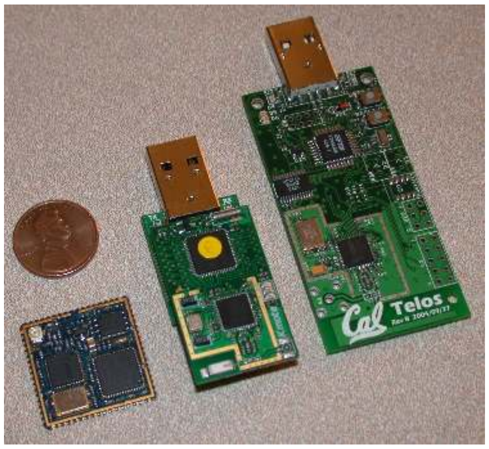


Figure 1: EPIC, KMote, and Telos motes. Each has an 8MHz micro-controller, 10kB of RAM, 48kB of program flash, and a 250kbps radio.

Maintaining a small memory footprint is a major requirement of algorithm design. Memory in low-cost, ultra-low power devices does not track Moore's Law. One indication of this is that micro-controller RAM costs three orders of magnitude more than PC SRAM and five orders more than PC DRAM. More importantly, SRAM leakage current, which grows with capacity, dictates overall standby power consumption and, hence, lifetime. Designs that provide large RAMs in conjunction with 32-bit processors go to great lengths to manage power. One concrete example of such nodes is the Sun SPOT [20], which enters a low-power sleep state by writing RAM contents to flash. Restoring memory from flash on wakeup uses substantial power and takes considerable time. The alternative, taken in most sensor node designs, is to have just a few kilobytes of RAM. This, in turn, imposes limits on the storage complexity of network (and other) protocols, requiring routing tables, buffering, and caches be kept small. The historical trends of monetary and energy costs suggest these constraints are likely to last.

Wireless sensors are typically embedded in the physical environment associated with their application. Communication connectivity varies due to environmental and electromagnetic factors, with the additional constraint that no human being will shepherd the device to a better setting, as with a cell phone or laptop. The degree of the network at a node, i.e., the number of nodes in its communication neighborhood, is determined not by the desired network organization but the physical device placement, which is often dictated by application requirements and physical constraints. There may be thousands of nodes in close proximity, or just a few. A single transmission may be received by many devices, so any retransmission, response, or even a simple acknowledgment, may cause huge contention, interference and loss. Redundancy is essential for reliability, but it also can be a primary cause of loss.

This last point is one of the key observations that has emerged from a decade of development of networking abstractions for wireless sensor networks: the variety of network topologies and densities across which sensor network protocols must operate calls for a polite, density-aware, local retransmission scheme. This paper describes the Trickle algorithm, which uses such a communication pattern to provide an eventual consistency mechanism to protocols and services. In the past ten years, a key insight that has emerged from the wireless sensor network community is that many protocol problems can be reduced to maintaining eventual consistency. Correspondingly, Trickle has emerged as the core networking primitive at the heart of practical, efficient, and robust implementations of many sensor network protocols and systems. Before diving into the details of the Trickle, however, we review how core sensor networking protocols work and differ from conventional networking protocols, with the goal of exploring how a Trickle-like primitive satisfies some of their needs.

2 Networking Protocols

Networking issues are at the core of embedded sensor network design because radio communication – listening, receiving, and transmitting – dominates the active energy budget and defines system lifetime. The standard energy cost metric for multihop protocols, in either link layer meshing or network layer routing, is communication cost, defined as the number of individual radio transmissions and receptions. One protocol is more efficient than another if it can provide equivalent performance (e.g., throughput, latency, delivery ratio) at a lower communication cost. Protocols focus on minimizing transmissions and making sure transmitted packets arrive successfully.

Almost all sensor network systems rely on two multihop protocols for their basic operation: a *collection* protocol for pulling data out of a network and a *dissemination* protocol for pushing data into a network through one or more distinguished nodes or egress routers. Many higher level protocols build on dissemination and collection. For example, reprogramming services such as Deluge [9] use dissemination to deliver commands to change program images. Management layers [22] and remote source-level debuggers [25] also use dissemination. Reliable transport protocols, such as RCRT [18], and rate control protocols such as IFRC [19], operate on collection trees. Point-to-point routing schemes, such as S4 [16], establish overlays over multiple parallel collection topologies.

While collection and dissemination have the opposite communication patterns (all-to-one vs. one-to-all) and differ in reliability (unreliable versus reliable), both maintain eventually consistent shared state between nodes. The rest of this section provides a high-level overview of these two protocol classes. It provides details on the challenging problems they introduce, and how some of them can be solved through eventual consistency.

2.1 Pushing Data In: Dissemination

One problem sensor network administrators face is dynamically changing how a network collects data by changing the sampled sensors, the sampling rate, or even the code running on the nodes by disseminating the change to every node in a network. We begin with a discussion of dissemination protocols because they were the original impetus for Trickle and are its simplest application.

Early systems used packet floods to disseminate changes. Flooding protocols rebroadcast packets they receive. Flooding is very simple – often just a line or two of code – but has many problems. First, floods are unreliable. Inevitably, some nodes do not receive the packet, so users typically repeatedly flood until every node receives it. Second, in high density networks, many nodes end up re-broadcasting packets at the same time. These messages collide and cause a form of network collapse called a “broadcast storm” [17].

Second-generation dissemination and network programming systems like Xnp [3] and TinyDB [15] use an adaptive flood combined with a protocol to request missing messages. Adaptive flooding uses an estimate of the node density to limit the flooding rate. The missing message protocol allows nodes to request the (hopefully few) missing messages from their neighbors. Unfortunately, getting such protocols to work well can be tricky, especially across a range of network densities and object sizes.

Another way to look at dissemination protocols is that they ensure that every node has an eventually consistent version of some shared state, such as the value of a configuration parameter or command. Data consistency is when all nodes have the same version of that state, and nodes resolve inconsistencies by updating neighbors to the newer version.

Inductively, these definitions cause the network to converge on the most recent version. To disseminate a command, a system installs it on one node as a newer version and initiates the consistency protocol.

Casting dissemination as a data consistency problem means it does not provide full reliability. Eventual consistency only promises to deliver the most recent version to connected nodes. Disconnected nodes can and often do miss updates. In practice, however, this limitation is rarely problematic. An administrator who changes the data reporting rate three times then adds some new nodes expects them to receive the most recent reporting rate, not all three. Similarly, when sending commands, users do not expect a new node to receive the entire history of all commands injected into a network. A node that is disconnected for several minutes will still receive the most recent command when it reconnects, however.

Dissemination protocols succeed where flooding and its derivatives fail because they cast the problem of delivering data into maintaining data consistency among neighbors. This allows them to provide a very useful form of reliability in arbitrary topologies with no *a priori* topology knowledge or configuration. An effective dissemination protocol, however, needs to bring nodes up to date quick while sending few packets when every node has the most recent version: this is correspondingly a requirement for the underlying consistency mechanism.

2.2 Pulling Data Out: Collection

As the typical sensor network goal is to report observations on a remote environment, it is not surprising that data collection is the earliest and most studied class of protocol. There are many collection protocol variations, similar to how there are many versions of TCP. These differences aside, all commonly used collection protocols provide unreliable datagram delivery to a collection point using a minimum-cost routing tree. Following the general goal of layer 3 protocols, cost is typically measured in terms of expected transmissions, or ETX [2]: nodes send packets on the route that requires the fewest transmissions to reach a collection point.

The earliest collection protocol, Directed Diffusion, proposed dynamically setting up collection trees based on data-specific node requests [10]. Early experiences with low-power wireless, however, led many deployments to move towards a much simpler and less general approach, where each node decides on a single next hop for all forwarded data traffic, thereby creating routing trees to fixed collection points. The network builds this tree by establishing a routing cost gradient. A collection point has a cost of 0. A node calculates the cost of each of its candidate next hops as the cost of that node plus the cost of the link to it. Inductively, a node's cost is the sum of the costs of the links in its route. Figure 2 illustrates an example topology.

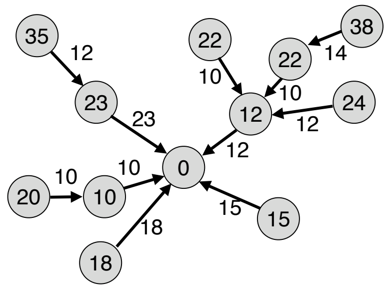


Figure 2: Sample collection tree, showing per-link and node costs. The cost of a node is its next hop's cost plus the cost of the link.

Collection variations boil down to how they quantify and calculate link costs, the number of links they maintain, how they propagate changes in link state amongst nodes, and how frequently they re-evaluate link costs and switch parents. Early protocols used hop-counts [8] as a link cost metric, similar to MANET protocols such as AODV and DSDV; second-generation protocols such as MintRoute [24] and Srcr [2] estimated the transmissions per delivery on a link using periodic broadcasts; third generation protocols, such as MultiHopLQI, added physical layer signal quality to the metric; current generation collection protocols, such as CTP, unify these approaches, drawing on information from multiple layers [6].

Most collection layers operate as anycast protocols. A network can have multiple data collection points, and collection automatically routes to the closest one. As there is only one destination – any collection point – the required routing state can be independent of network density and size. Most protocols use a small, fixed-size table of candidate next hops. They also attempt to strike a balance between route stability and churn to discover new, possibly better parents by switching parents infrequently and using damping mechanisms to limit the rate of change.

As collection protocols have improved and become better at choosing routes, reducing control traffic has become an increasingly important component of efficiency. While nodes can piggyback some control information on data packets, they need to send link-layer broadcasts to their local neighbors to advertise their presence and routing cost. Choosing how often to send these advertisements introduces a difficult design tension. A slow rate imposes a low overhead, but limits how quickly the tree can adapt to failures or link changes, making its data traffic less efficient. A fast rate imposes a higher overhead, but leads to an agile tree that can more accurately find the best route to use.

This tension is especially challenging when a network only collects data in response to events, and so can go through periods of high and low data rates. Having a high control rate during periods of low traffic is highly inefficient, while having a low control rate during periods of high traffic makes the tree unable to react quickly enough to changes. When starting a burst of transmissions, a node may find that link costs have changed substantially necessitating a change in its route and, as a result, its advertised routing cost. Changes in costs need to propagate quickly, or the topology can easily form routing loops. For example, if a link's cost increases significantly, then a node may choose one if its children as its next hop. Since the protocol state must be independent of the topology, a node cannot avoid this by simply enumerating its children (constraining tree in-degree to a constant leads to inefficient, circuitous topologies in dense networks).

Current protocols, such as CTP [21] and ArchRock's routing layer [1], resolve this tension by reducing the routing gradient as a data consistency problem. The gradient is consistent as long as children have a higher cost than their parent. An inconsistency can arise when costs change enough to violate this constraint. As long as routing costs are stable, nodes can assume the gradient is consistent and avoid exchanging unnecessary packets.

2.3 A General Mechanism

The examples above described how two very different protocols can both address a design tension by reducing a problem to maintaining data consistency. Both examples place the same requirements on a data consistency mechanism: it needs to resolve inconsistencies quickly, send few packets when data is consistent, and require very little state. The Trickle algorithm, discussed in the next section, meets these three requirements.

3 Trickle

The Trickle algorithm establishes a density-aware local broadcast with an underlying consistency model that guides when a node communicates. When a node's data does not agree with its neighbors, it communicates quickly to resolve the inconsistency. When nodes agree, they slow their communication rate exponentially, such that in a stable state nodes send at most a few packets per hour. 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 consistent. Furthermore, by relying only on local broadcasts, Trickle handles network re-population, is robust to network transience, loss, and disconnection, and requires very little state (implementations use 4-11 bytes).

While Trickle was originally designed for reprogramming protocols (where the data is the code of the program being updated), experience has shown it to be a powerful mechanism that can be applied to wide range of protocol design problems. For example, routing protocols can use Trickle to ensure that nodes in a given neighborhood have consistent, loop-free routes. When the topology is consistent, nodes occasionally gossip to check that they still agree, and when the topology changes they gossip more frequently, until they reach consistency again.

τ	Communication interval length
t	Timer value in range $[\frac{\tau}{2}, \tau)$
c	Communication counter
k	Redundancy constant
τ_l	Smallest τ
τ_h	Largest τ

Figure 3: Trickle parameters and variables.

For the purpose of clearly explaining the reasons behind Trickle's design, all of the experimental results in this section are from simulation, in some cases very high-level abstract simulators. In practice, Trickle's simplicity means it works remarkably well in the far more challenging and difficult real world. The original Trickle paper [13], as well as Deluge [9] and DIP [14] report experimental results from real networks.

3.1 Algorithm

Trickle's basic mechanism is a randomized, suppressive broadcast. A Trickle has a time interval of length τ and a redundancy constant k. At the beginning of an interval, a node sets a timer t in the range of $(\frac{\tau}{2}, \tau]$. When this timer fires, the node decides whether to broadcast a packet containing metadata for detecting inconsistencies. This decision is based on what packets the node heard in the interval before t. A Trickle maintains a counter c, which it initializes to 0 at the beginning of each interval. Every time a node hears a Trickle broadcast that is consistent with its own state, it increments c. When it reaches time t, the Trickle broadcasts if $c < k$. Randomizing t spreads transmission load over a single-hop neighborhood, as nodes take turns being the first node to decide whether to transmit. Figure 3 summarizes Trickle's parameters.

3.2 Scalability

Transmitting only if $c < k$ makes a Trickle density aware, as it limits the transmission rate over a region of the network to a factor of k. In practice, the transmission load a node observes over an interval is $O(k \cdot \log(d))$, where d is the network density. The base of the logarithm depends on the packet loss rate PLR: it is $\log_{\frac{1}{PLR}}$.

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 that a node will miss a single packet. If a node misses a packet, it

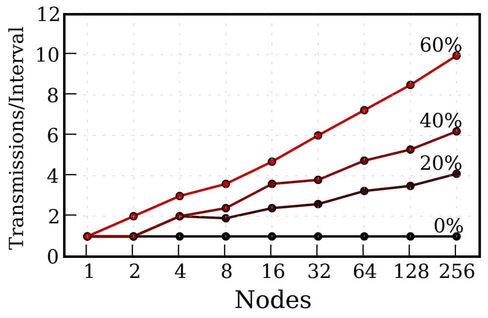


Figure 4: Trickle's transmissions per interval scales logarithmically with density. The base of the logarithm is a function of the packet loss rate (the percentages).

will transmit, resulting in two transmissions. There is correspondingly a 1% chance a node will miss two packets from other nodes, leading to three transmissions. In the extreme case of a 100% loss rate, each node is by itself: transmissions scale linearly.

Figure 4 shows this scaling. The number of transmissions scales logarithmically with density, and the slope the line (base of the logarithm) depends on the loss rate. These results come from a Trickle-specific algorithmic simulator we implemented to explore the algorithm's behavior under controlled conditions. Consisting of little more than an event queue, this simulator allows configuration of all of Trickle's parameters, run duration, and the boot time of nodes. It models a uniform packet loss rate (same for all links) across a single hop network. Its output is a packet send count.

3.3 Synchronization

Figure 4's scaling assumes that all nodes are synchronized, such that the intervals during which they are awake and listening to their radios line up perfectly. Inevitably, this kind of time synchronization imposes a communication, and therefore energy, overhead. While some networks can provide time synchronization to Trickle, others cannot. Therefore, Trickle is designed to work in both the presence and absence of synchronization.

Trickle chooses t in the range of $(\frac{\tau}{2}, \tau]$ rather than $(0, \tau]$ because the latter causes the transmission load in unsynchronized networks to scale with $O(\sqrt{d})$. This undesirable scaling occurs due to the *short listen problem*, where some subset of motes gossip soon after the beginning of their interval. They listen for only a short time, before anyone else has a chance to speak up. This is not a problem if all of the intervals are synchronized, since the first gossip will quiet everyone else. However, if nodes are not synchronized, a node may start its interval just after another node's broadcast, resulting in missed messages and increased transmission load.

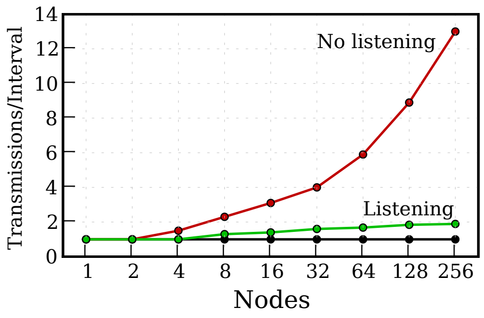


Figure 5: Without a listen-only period, Trickle's transmissions scale with a square root of the density when intervals are not synchronized. With a listen-only period of duration $\frac{\tau}{2}$, the transmissions per interval asymptotically approach $2k$. The black line shows how Trickle scales when intervals are synchronized. These results are from lossless networks.

Unlike loss, where the extra $O(\log(d))$ transmissions keep the worst case node that missed several packets up to date, the additional transmissions due to the short listen problem are completely wasteful. Choosing t in the range of $(\frac{\tau}{2}, \tau]$ removes this problem: it defines a "listen-only" period of the first half of an interval. A listening period improves scalability by enforcing a simple constraint. If sending a message guarantees a silent period of some time T that is independent of density, then the send rate is bounded above (independent of the density). When a mote transmits, it suppresses all other nodes for at least the length of the listening period. Figure 5 shows how a listen period of $\frac{\tau}{2}$ bounds the total sends in a lossless single-hop network to be $2k$. With loss, transmissions scale as $O(2k \cdot \log(d))$ per interval, returning scalability to the $O(\log(d))$ goal.

3.4 Controlling τ

A large τ (gossiping interval) leads to a low communication overhead, but propagates information slowly. Conversely, a small τ imposes a higher communication overhead, but propagates data 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 quickly make data consistent with a very small cost. τ has a lower bound, τ_l, and an upper bound τ_h. When τ expires without a node receiving a new update, τ doubles, up to a maximum of τ_h. When a node detects a data inconsistency (e.g., a newer version number in dissemination, a gradient constraint violation in collection), it resets τ to be τ_l.

Essentially, when there's nothing new to say, motes gossip infrequently: τ is set to τ_h. However, as soon as a mote

Event	Action
τ Expires	Double τ, up to τ_h. Reset c, pick a new t.
t Expires	If $c < k$, transmit.
Receive consistent data	Increment c.
Receive inconsistent data	Set τ to τ_l. Reset c, pick a new t.

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

Figure 6: Trickle pseudocode.

hears something new, it gossips more frequently, so those who haven't heard the new data receive it quickly. 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 consistency, and low overhead when the network is consistent. The cost per inconsistency (shrinking τ), is approximately $log(\frac{\tau_h}{\tau_l})$ additional sends. For a τ_l of one second and a τ_h of one hour, this is a cost of eleven packets to obtain a three-thousand fold decrease in the time it takes to detect an inconsistency (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 that the network is consistent as well as quickly inform nodes when there is an inconsistency.

Figure 6 shows pseudocode for the complete Trickle algorithm.

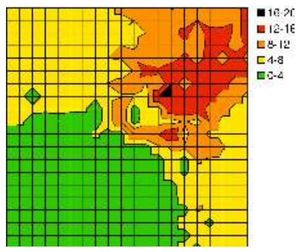
3.5 Case Study: Maté

Maté is a lightweight bytecode interpreter for wireless sensornets [11]. Programs are tiny sequences of optimized bytecodes. The Maté runtime uses Trickle to install new programs in a network, by making all nodes consistent to the most recent version of a script.

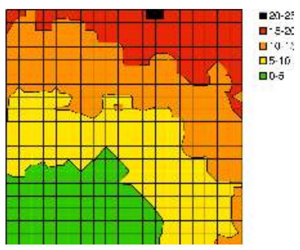
Maté uses Trickle to periodically broadcast version summaries. In all experiments, code routines fit in a single packet (30 bytes). The runtime registers routines with a Trickle propagation service, which then maintains all of the necessary timers and broadcasts, notifying the runtime when it installs new code. Maté uses a very simple consistency resolution mechanism. It broadcasts the missing routines three times: one second, three seconds, and seven seconds after hearing there is an inconsistency.

Figure 7 shows simulation results of Maté's behavior during a reprogramming event. These results come from the TOSSIM simulator [12], which simulates entire sensornet applications and models wireless connectivity at the bit level. In these experiments, τ_l is one second and τ_h is one minute.

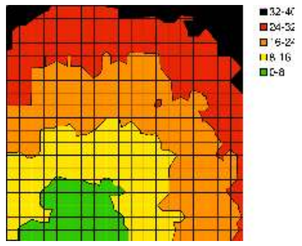
Each simulation had 400 nodes regularly placed in a square grid with node spacings of 5, 10, 15, and 20 feet. By varying network density, we were able to examine how Trickle's propagation rate scales over different loss rates



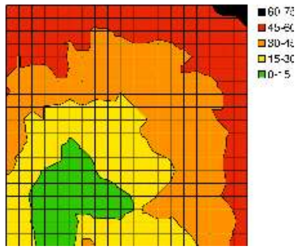
(a) 5' Spacing, 6 hops



(b) 10' Spacing, 16 hops



(c) 15' Spacing, 32 hops



(d) 20' Spacing, 40 hops

Figure 7: Time to consistency in 20x20 TOSSIM grids (seconds). The hop count values in each legend are the expected number of transmissions necessary to get from corner to corner, considering loss.

and physical densities. Density ranged from a five foot spacing between nodes up to twenty feet (the networks were 95'x95' to 380'x380'). Crossing the network in these topologies takes from six to forty hops. [1] Time to complete propagation varied from 16 seconds in the densest network to about 70 seconds for the sparsest, representing a latency of 2.7 and 1.8 seconds per hop, respectively. The minimum per-hop Trickle latency is $\frac{\tau_l}{2}$ and the consistency mechanism broadcasts a routine one second after discovering an inconsistency, so the best case latency is 1.5 seconds per hop. Despite an almost complete lack of coordination between nodes, Trickle still is able to cause them them to cooperate efficiently.

Figure 8 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 trade-off 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 trig-

[1] These hop count values come from computing the minimum cost path across the network loss topology, where each link has a weight of $\frac{1}{1-loss}$, i.e. the expected number of transmissions to successfully traverse that link.

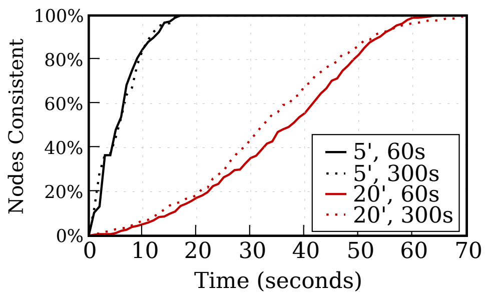


Figure 8: Rate nodes reach consistency for different τ_hs in TOSSIM. A larger τ_h does not slow reaching consistency.

gers nodes to reset τ to τ_l, bringing them quickly to a consistent state.

The Maté 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 code routines. Trickle itself requires only eleven bytes for its counters; the remaining RAM is for internal coordination (e.g. pending and initialization flags). The executable code is 1.8K (90 lines of code). Other implementations have similar costs. The algorithm requires few CPU cycles, and can operate at a very low duty cycle.

3.6 Uses and Improvements

Trickle is not just used by Maté; it and its derivatives are used in almost every dissemination protocol today. The Deluge binary dissemination protocol for installing new sensor node firmware uses Trickle to detect when nodes have different firmware versions [9] (τ_l =500ms, τ_h =1.1h). The MNP binary dissemination protocol (τ_l =16s, τ_h =512s) adjusts Trickle so that nodes with more neighbors are more likely to send updates by preventing low degree nodes from suppressing high degree ones [23]. The Drip command layer of the Sensornet Management System uses Trickle (τ_l =100ms,τ_h =32s) to install commands [22]. The Tenet programming architecture uses Trickle (τ_l =100ms, τ_h =32s) to install small dynamic code tasks [7].

In the past few years, as collection protocols have improved in efficiency, they have also begun to use Trickle. The Collection Tree Protocol (CTP), the standard collection layer in the TinyOS operating system distribution [21], uses Trickle timers (τ_l =64ms, τ_h =1h) for its routing traffic. The 6LoWPAN IPv6 routing layer in Arch Rock's software uses Trickle to keep IPv6 routing tables and ICMP neighbor lists consistent [1]. As protocols continue to improve, Trickle's efficacy and simplicity will cause it to be used in more protocols and systems.

One limitation with Trickle as described in this paper is that its maintenance cost grows $O(n)$ with the number of data items, as nodes must exchange version numbers. This growth may be a hindering factor as Trickle's use increases. Recent work on the DIP protocol addresses this concern by using a combination of hash trees and randomized searches, enabling the maintenance cost to remain $O(1)$ while imposing a $O(log(n))$ discovery cost [14].

4 Discussion

Wireless sensor networks, like other ad hoc networks, do not know the interconnection topology *a priori* and are typically not static. Nodes must discovered it by attempting to communicate and then observing where communication succeeds. In addition, the communication medium is expected to be lossy. Redundancy in such networks is both friend and foe, but Trickle reinforces the positive aspects and suppresses the negative ones.

Trickle draws on two major areas of prior research. The first area is controlled, density-aware flooding algorithms for wireless and multicast networks [5, 17]. The second is epidemic and gossiping algorithms for maintaining data consistency in distributed systems [4]. Although both techniques – broadcasts and epidemics – have assumptions that make them inappropriate in their pure form to eventual consistency in sensor networks, they are powerful techniques that Trickle draws from. Trickle's suppression mechanism is inspired by the request/repair algorithm used in Scalable and Reliable Multicast (SRM) [5]. Trickle adapts to local network density as controlled floods do, but continually maintains consistency in a manner similar to epidemic algorithms. Trickle also takes advantage of the broadcast nature of the wireless channel, employing SRM-like duplicate suppression to conserve precious transmission energy and scale to dense networks.

Exponential timers are a common protocol mechanism. Ethernet, for example, uses an exponential backoff to prevent collisions. While Trickle also has an exponential timer, its use is reversed. Where Ethernet defaults to the smallest time window and increases it only in the case of collisions, Trickle defaults to the largest time window and decreases it only in the case of an inconsistency. This reversal is indicative of the different priorities in ultra-low power networks: minimizing energy consumption, rather than increasing performance, is typically the more important goal.

In the case of dissemination, Trickle timers spread out packet responses across nodes while allowing nodes to estimate their degree and set their communication interval. Trickle leads to energy efficient, density-aware dissemination not only by avoiding collisions through making collisions rare, but also by suppressing unnecessary retransmissions.

Instead of trying to enforce suppression on an abstraction of a logical group, which can become difficult in multi-hop networks with dynamically changing connectivity, Trickle suppresses in terms of an implicit group: nearby nodes that hear a broadcast. Correspondingly, Trickle does not impose the overhead of discovering and maintaining logical groups, and effortlessly deals with transient and lossy wireless links. By relying on this implicit naming, the Trickle algorithm remains very simple: implementations can fit in under 2K of code, and require a mere 11 bytes of state.

Routing protocols discover other routers, exchange routing information, issue probes, and establish as well as tear down links. All of these operations can be rate-controlled by Trickle. For example, in our experiences exploring how wireless sensor networks can adopt more of the IPv6 stack in 6LoWPAN, Trickle provides a way to support established ICMP-v6 mechanisms for neighbor discovery, duplicate address detection, router discovery, and DHCP in wireless networks. Each of these involve advertisement and response. Trickle mechanisms are a natural fit: they avoid loss where density is large, allow prompt notifications of change and adapt to low energy consumption when the configuration stabilizes. By adopting a model of eventual consistency, nodes can locally settle on a consistent state without requiring any actions from an administrator.

Trickle was initially developed for distributing new programs into a wireless sensornet: the title of the original paper is "Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks." [13] Experience has shown it to have much broader uses. Trickle-based communication, rather than flooding, has emerged as the central paradigm for the basic multi-hop network operations of discovering connectivity, data dissemination, and route maintenance.

Looking forward, we expect that the use of these kind of techniques to be increasingly common throughout the upper layers of the wireless network stack. Such progress will not only make existing protocols more efficient, it will enable sensor networks to support layers originally thought infeasible. Viewing protocols as a continuous process of establishing and adjusting a consistent view of distributed data is an attractive way to build robust distributed systems.

Acknowledgements

This work was supported, in part, by the Defense Department Advanced Research Projects Agency (grants F33615-01-C-1895 and N6601-99-2-8913), the National Science Foundation (grants No. 0122599, IIS-033017, 0615308 and 0627126), California MICRO program, Intel Corporation, DoCoMo Capital, Foundation Capital, and a Stanford Terman Fellowship. Research infrastructure was provided by the National Science Foundation (grant No. 9802069).

References

[1] Arch Rock Corporation. An IPv6 Network Stack for Wireless Sensor Networks. http://www.archrock.com.

[2] D. D. Couto, D. Aguayo, J. Bicket, and R. Morris. A High-Throughput Path Metric for Multi-Hop Wireless Routing. In *Proceedings of the Ninth Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2003.

[3] Crossbow, Inc. Mote In Network Programming User Reference. http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/Xnp.pdf.

[4] A. Demers, D. Greene, C. Hauser, W. Irish, and J. Larson. Epidemic Algorithms for Replicated Database Maintenance. 1987.

[5] 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 (SIGCOMM)*, 1995.

[6] R. Fonseca, O. Gnawali, K. Jamieson, and P. Levis. Four Bit Wireless Link Estimation. In *Proceedings of the Sixth Workshop on Hot Topics in Networks (HotNets VI)*, 2007.

[7] O. Gnawali, B. Greenstein, K.-Y. Jang, A. Joki, J. Paek, M. Vieira, D. Estrin, R. Govindan, and E. Kohler. The TENET Architecture for Tiered Sensor Networks. In *Proceedings of the Fourth International Conference on Embedded Networked Sensor Systems (Sensys)*, 2006.

[8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[9] J. W. Hui and D. Culler. The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In *Proceedings of the Second International Conference on Embedded Networked Sensor Systems (SenSys)*, 2004.

[10] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: a Scalable and Robust Communication Paradigm for Sensor Networks. In *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2000.

[11] P. Levis, D. Gay, and D. Culler. Active Sensor Networks. In *Proceedings of the Second USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2005.

[12] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2003.

[13] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A Self-Regulating Algorithm for Code Maintenance and Propagation in Wireless Sensor Networks. In *Proceedings of the First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2004.

[14] K. Lin and P. Levis. Data Discovery and Dissemination with DIP. In *Proceedings of the Seventh International Symposium on Information Processing in Sensor Networks (IPSN)*, 2008.

[15] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *Transactions on Database Systems (TODS)*, 2005.

[16] Y. Mao, F. Wang, L. Qiu, , S. Lam, and J. Smith. S4: Small State and Small Stretch Routing Protocol for Large Wireless Sensor Networks. In *Proceedings of the Fourth USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[17] 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 International Conference on Mobile Computing and Networking (MobiCom)*, 1999.

[18] J. Paek and R. Govindan. RCRT: rate-controlled reliable transport for wireless sensor networks. In *Proceedings of the Fifth International Conference on Embedded Networked Sensor Systems (SenSys)*, 2007.

[19] S. Rangwala, R. Gummadi, R. Govindan, and K. Psounis. Interference-aware Fair Rate Control in Wireless Sensor Networks. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, 2006.

[20] Sun Microsystems Laboratories. Project Sun SPOT: Small Programmable Object Technology. http://www.sunspotworld.com/.

[21] TinyOS Network Protocol Working Group. TEP 123: The Collection Tree Protocol. `http://www.tinyos.net/tinyos-2.x/doc/txt/tep123.txt`, 2007.

[22] G. Tolle and D. Culler. Design of an Application-Cooperative Management System for Wireless Sensor Networks. In *Proceedings of the Second European Workshop of Wireless Sensor Netw orks (EWSN)*, 2005.

[23] L. Wang. MNP: Multihop Network Reprogramming Service for Sensor Networks. In *Proceedings of the Second International Conference On Embedded Networked Sensor Systems (SenSys)*, 2004.

[24] A. Woo, T. Tong, and D. Culler. Taming the Underlying Challenges of Multihop Routing in Sensor Networks. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2003.

[25] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: a Comprehensive Source-level Debugger for Wireless Sensor Networks. In *Proceedings of the Fifth International Conference on Embedded Networked Sensor Systems (SenSys)*, 2007.