Data Discovery and Dissemination with DIP

tinyos-2.x/tos/lib/net/dip

Kaisen Lin
UC San Diego
kaisenl@cs.ucsd.edu

Philip Levis
Stanford University
pal@cs.stanford.edu

Abstract

We present DIP, a data discovery and dissemination protocol for wireless networks. Prior approaches, such as Trickle or SPIN, have overheads that scale linearly with the number of data items. For T items, DIP can identify new items with $O(log(T))$ packets while maintaining a $O(1)$ detection latency. To achieve this performance in a wide spectrum of network configurations, DIP uses a hybrid approach of randomized scanning and tree-based directed searches. By dynamically selecting which of the two algorithms to use, DIP outperforms both in terms of transmissions and speed. Simulation and testbed experiments show that DIP sends 20-60% fewer packets than existing protocols and can be 200% faster, while only requiring $O(log(log(T)))$ additional state per data item.

1 Introduction

Reliable data dissemination is a basic building block for sensor network applications. Dissemination protocols such as XNP [3], Deluge [8], Sprinkler [16] and MNP [21] distribute new binaries into a network, enabling complete system reprogramming. Dissemination protocols such as Maté's capsule propagation [9] and Tenet's task propagation [7] install small virtual programs, enabling application-level reprogramming. Finally, dissemination protocols such as Drip [20] allow administrators to adjust configuration parameters and send RPC commands [22].

Dissemination protocols reliably deliver data to every node in a network using key, version tuples on top of some variant of the Trickle algorithm [11]. We describe these protocols and their algorithms in greater depth in Section 2. The key characteristic they share is a node detects a neighbor needs an update by observing that the neighbor has a lower version number for a data item (key). The cost of this mechanism scales linearly with the number of data items: T data items require T version number announcements. Even though a protocol can typically put multiple announcements in a single packet, this is only a small constant factor improvement. Fundamentally, these algorithms scale with $O(T)$ for T total data items. This linear factor introduces a basic cost/latency tradeoff. Nodes can either keep a constant detection latency and send $O(T)$ packets, or keep a constant cost and have an $O(T)$ latency.

The key insight in this paper is that dissemination protocols can break this tradeoff by aggregating many data items into a single advertisement. Because these aggregates compress information, they can determine that an update is needed, but cannot always determine which data item needs an update. Section 3 outlines existing dissemination algorithms and describes a new algorithm, *search*, that breaks the cost/latency tradeoff, enabling fast and efficient dissemination. By using a hash tree of data item version numbers, a protocol using search can discover an update is needed with $O(log(T))$ transmissions.

In simple collision-free and lossless network models, search works well. However, two problems can make the hash tree algorithm perform poorly in real networks. First, packet losses can make it difficult to quickly traverse the tree. Second, the multiple advertisements caused by packet losses are completely redundant: there is typically only one subtree to explore. Through controlled simulation experiments, we find that in cases of very high loss or when a large fraction of items require updates, the underlying constant factors can cause randomized scans to be more efficient than hash tree searches.

Section 3 presents an analytical framework to understand these tradeoffs. The analysis shows that whether periodic advertisements or searches is more efficient depends on three factors: network density, packet loss ratios, and the percentage of items that need updates. While they have similar efficiency when reasonably close to their equality point, one can be a factor of two more efficient than the other at the edges. This analysis indicates that a scalable dissemination protocol can get the best of both worlds by using a hybrid approach, dynamically switching between algorithms based on run-time conditions.

Section 4 presents such a protocol, called DIP (DIssemination Protocol). DIP continuously measures network con-

ditions and estimates whether each data item requires an updates. Based on this information, it dynamically chooses between a hash tree-based search approach and scoped randomized scanning. DIP improves searching performance by combining hashes over ranges of the key space with a bloom filter. Hashes allow it to detect whether there are version number inconsistencies while Bloom filters let it quickly pinpoint the source of the inconsistency.

Section 5 evaluates DIP in simulation and on a mote testbed. In simulated clique networks, DIP sends up to 30-50% fewer packets than either scanning or searching and is correspondingly 30-50% faster. In the Intel Mirage multihop 80 node testbed, DIP sends 60% fewer packets than scanning or searching. In some cases, DIP sends 85% fewer packets than scanning, the dominant algorithm in use today. By improving its transmission efficiency, DIP is also able to disseminate faster: across real, multihop networks, DIP is 60% faster for a few items and over 200% faster for many items. Section 6 presents how DIP relates to prior work.

These results show that DIP is significantly more efficient than existing approaches. This improvement comes at a cost of an additional $log(log((T))$ bits of state per data item for T items. Section 7 discusses the implications of these findings. The tradeoffs between scanning and searching touch on a basic tension in sensornet protocol design. While searching can find inconsistencies quickly by exchanging higher-level metadata, its deterministic operation means that it cannot leverage the communication redundancy inherent to wireless protocols. While scanning can take advantage of this redundancy through randomization, it does so by explicitly avoiding any complex metadata exchange. DIP's results suggest the complex tradeoffs between randomized versus deterministic algorithms in wireless networks deserve further study.

This paper makes three research contributions. First, it proposes DIP, an adaptive dissemination protocol that can scale to a large number of items. Second, it introduces using a bloom filter as an optimization to update detection mechanisms in dissemination protocols. Third, it evaluates DIP and shows it outperforms existing dissemination protocols, reducing transmission costs by 60% and latency by up to 40%. These results suggest the complex tradeoffs between randomized versus deterministic algorithms in lossy networks deserve further study.

2 Motivation and Background

Efficiently, quickly, and reliably delivering data to every node in a network is the basic mechanism for almost all administration and reprogramming protocols. Maté virtual machines disseminate code capsules [9]; Tenet disseminates tasks [7]; Deluge [8], Typhoon [12] and MNP [21] disseminate binary images; Drip disseminates parameters [20] and Marionette builds on Drip to disseminate queries [22].

2.1 Trickle

All of these dissemination protocols use or extend the Trickle algorithm [11]. Trickle periodically broadcasts a summary of the data a node has, unless it has recently heard an identical summary. As long as all nodes agree on what data they have, Trickle exponentially increases the broadcast interval, thereby limiting energy costs when a network is stable. When Trickle detects that other nodes have different data, it starts reporting more quickly. If a node hears an older summary, it sends an update to that node.

In practice, protocols assign keys to data items and summaries use version numbers to determine if data is newer or older. For example, the Maté VM assigns each code capsule a unique number. Installing new code in the network involves selecting a capsule, incrementing its version number, and installing the new version on a single source node. That node starts quickly advertising it has a new version, shrinking its advertisement interval to a small value (e.g., one second). Neighbors hear the advertisement and quickly advertise they have an old version, causing the source to broadcast the update and spread the new code.[1] This process repeats throughout the network until all nodes have the update. Trickle's transmission rate slows down, following the exponential interval increase rule up to a maximum interval size (e.g., one hour).

Systems use Trickle because they are efficient and scale logarithmically with node density. As nodes suppress redundant advertisements, Trickle can scale to very dense networks. This suppression is not perfect: packet losses cause the number of redundant advertisements to scale logarithmically with network density. By constantly adjusting its advertisement interval, Trickle advertises new data quickly yet advertises slowly when a network is consistent.

2.2 A Need for Scalability

While using Trickle enables dissemination protocols to scale to dense networks, no protocol currently scales well to supporting a large number of data items. As sensornet applications grow in complexity and nodes have more storage, administrators will have more parameters to adjust, more values to monitor, and a need for a larger number of concurrent capsules, tasks, or queries.

When an administrator injects new data to a single node, that node knows the data is newer. Therefore, disseminating new data with Trickle is comparatively fast. The more challenging case is when nodes need to detect that there is new data. This case occurs when disconnected nodes rejoin a network. Both the old and new nodes think that the network is up to date, and so advertise at a very low rate.

Because current protocols advertise (key, version) tuples, their transmission costs increase linearly with the number

[1] MNP [21] extends simple trickles in that it uses density estimates to decide which node sends the actual update.

of distinct data items. To detect that a data item is different, a node must either transmit or receive a tuple for that item. This approach causes the cost/latency product of a trickle to scale with $O(T)$, where T is the total number of data items. Some protocols, such as Drip and Deluge, maintain a constant latency by keeping a fixed maximum interval size and disseminating each item with a separate trickle. As the number of items grows, the transmission rates of these protocols grow with $O(T)$. Other protocols, such as Tenet's task dissemination, keep a constant communication rate so detection latency grows with $O(T)$.

As sensor systems grow in complexity, linear scalability will become a limiting factor in the effectiveness of these protocols: it will force administrators to choose between speed and efficiency. The next section quantifies these tradeoffs more precisely, and introduces a new hash-tree based algorithm that resolves this tension, so dissemination protocols can simultaneously be efficient and fast.

3 Protocol Tradeoffs

Dissemination protocols have two main performance metrics: detection latency and maintenance cost. Maintenance cost is the rate at which a dissemination sends packets when a network is up-to-date. Traditionally, these two metrics have been tightly coupled. A smaller interval lowers latency but increases the packet transmission rate. A larger interval reduces the transmission rate but increases latency. Trickle addresses part of this tension by dynamically scaling the interval size, so it is smaller when there are updates and larger when the network is stable. While this enables fast dissemination once an update is detected, it does not help with detection itself.

Protocols today use two approaches to apply Trickle to many data items. The first establishes many parallel Trickles; the second uses a single Trickle that serially scans across the version numbers to advertise. This section proposes a third approach, which uses a hash tree to obtain constant detection latency and maintenance cost. To achieve this, searching introduces an $O(log(T))$ overhead when it detects an update is needed.

3.1 Scanning and Searching

Parallel detection uses a separate Trickle for each data item. Because the maximum trickle interval is fixed, parallel detection provides a detection latency bound independent of the number of items. However, this bound comes at a cost: parallel detection has a maintenance cost of $O(T)$.

Serial detection uses a single Trickle for all items. Each transmission contains a selection of the (key,version) tuples. Because serial detection scans across the tuples, it requires $O(T)$ trickle intervals to transmit a particular tuple. Therefore, serial detection has a latency of $O(T)$.

Protocol	Latency	Cost	Identify
Parallel Scan	$O(1)$	$O(T)$	$O(1)$
Serial Scan	$O(T)$	$O(1)$	$O(1)$
Search	$O(1)$	$O(1)$	$O(log(T))$

Table 1. Scalability of the three basic dissemination algorithms.

Term	Meaning
T	Total data items
N	New data items
D	Node density
L	Loss ratio

Table 2. Network parameters.

The parallel and serial approaches represent the $O(T)$ cost/latency product that basic trickles impose. One can imagine other, intermediate approaches, say where both cost and latency increase as $O(\sqrt{T})$. In all such cases, however, the cost/latency tradeoff persists.

A third approach is to *search* for different items using a hash tree, similar to a binary search. When a node sends an advertisement, it sends hashes of version numbers across ranges of data items. When a node hears an advertisement with a hash that does not match its own, it sends hashes of sub-ranges within that hash. This simple protocol backs up one tree level on each transmission to prevent locking.

When a network is stable, nodes advertise the top-level hashes that cover the entire keyspace. As these hashes cover all items, searching can detect new items in $O(1)$ time and transmissions. Determining which item is new requires $O(log(T))$ transmissions. As these transmissions can occur at a small Trickle interval rate, the latency of identifying the items is insignificant compared to detection.

While searching is much more efficient in detecting a single new item, it can be inefficient when there are many new items. This can occur, for example, if an administrator adds nodes that require all of the software updates

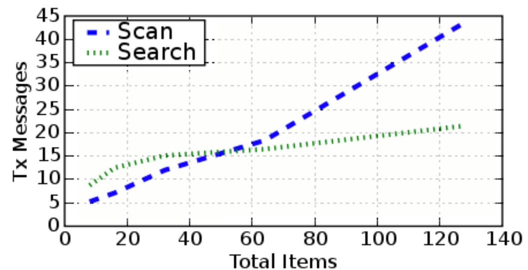


Figure 1. As T increases but N is constant, the chances a scan will find a new item goes down, and searches become more effective.

and programs running in the network. Since searching pays an $O(log(T))$ cost for each item, with N new items its cost will be $O(N \cdot log(T))$. In the worst case, this can be $O(T \cdot log(T))$, which is more expensive than the $O(T)$ of scanning approaches. Table 1 summarizes these tradeoffs, and Figure 1 shows which algorithm is better as T changes.

3.2 Analysis

Loss and density affect protocol performance. In the rest of this paper, we describe networks with the terms in Table 2. Trickle introduces a communication redundancy R of $log_{\frac{1}{L}}(D)$. This comes from the probability that a node with an area of density D will advertise even if $R-1$ nodes have already advertised because it lost those packets.

In the case of a parallel scan protocol, these extra transmissions are completely redundant: there will be $R{\cdot}T$ transmissions per interval, and detection latency remains $O(1)$. In serial scan protocols, these extra transmissions are not completely redundant: nodes may be at different points in their scan, or might be advertising a random subset. Because they are not redundant, scanning's latency goes down: these extra transmissions further cover the keyspace. Therefore, the detection latency of parallel scans are $O(\frac{T}{R})$.

Extra messages in search protocols are redundant for the same reason they are in serial scans. When nodes detect a hash mismatch, they will all respond with the same set of sub-hashes. Furthermore, if a node does not hear a sub-hash, it assumes consistency and backs up one level in the hash tree: heavy packet loss can slow tree traversal.

Searching is typically advantageous when N is small compared to T. If N is large, then scanning is effective because a random selection of items is likely of finding an inconsistency. In contrast, searching requires traversing the tree, introducing control packet overhead. When N is small, this overhead is less than the number of packets a scan must send to find a new item.

Together, these tradeoffs mean that which of the algorithms performs best depends on network conditions. High R and N values improve scanning performance. But when N is small, searching is more efficient. Furthermore, the two approaches are not mutually exclusive; a protocol can search until it determines the new item is in a small subset, at which point R may make scanning that subset more efficient than continuing the search. Thus an ideal protocol should switch from a scan to a search when nodes are down to their last few items and adjust to network conditions. The next section proposes such as protocol.

4 DIP

DIP is a hybrid data detection and dissemination protocol. It separates this into two parts: detecting that a difference occurs, and identifying which data item is different.

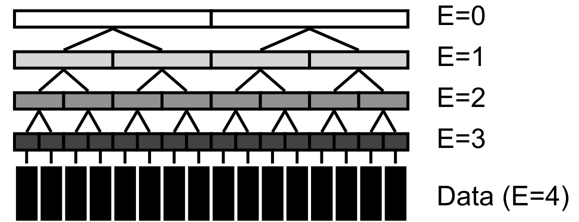


Figure 2. Example estimate values for a 16-item hash tree.

DIP dynamically uses a combination of searching and scanning based on network and version metadata conditions. To aid its decisions, DIP continually estimates the probability that a data item is different. DIP maintains these estimates through message exchanges. When probabilities reach 100%, DIP exchanges the actual data items. It is an eventual consistency protocol in that when data items are not changing, all nodes will eventually see a consistent set of data items.

This section starts with a broad overview of how DIP works. It introduces DIP's metadata structures, the messages it exchanges, its use of Trickle, and the details of its estimate system before finally describing the algorithms DIP applies when receiving and transmitting packets.

4.1 Overview

DIP stores a version number for each data item. In the steady state where all nodes are up to date and have the same versions, DIP uses Trickle to send hashes that cover all of the version numbers. Nodes that receive hashes which are the same as their own know they are consistent with their neighbors. If a node hears a hash that differs from its own, it knows that a difference exists, but does not know which specific item or who has the newer version.

In addition to the version number, DIP maintains a soft-state estimate of whether a given item differs from a neighbor's. It is soft in that if estimates are temporarily inaccurate or lost, the protocol will still proceed. In contrast, version numbers must be correct for consistency and correctness.

When DIP detects a hash of length H that differs, it gives each item covered by the hash a conservative estimate of $\frac{1}{H}$. This estimate is conservative because at least one of the H items is different.

DIP sends advertisements that improve its estimate accuracy by using smaller hashes. For example, a node that receives a differing hash of length H can respond by sending two hashes of length $\frac{H}{2}$. As Figure 2 shows, one can think of these levels of hashes defining a hash tree over the version number set; going down the tree involves sending smaller hashes, while going up the tree involves sending longer hashes.

Identifying which data item is different and which node has the newer version requires exchanging actual version numbers. In the hash tree, version numbers are hashes of length 1. Section 3 showed how if the probability of a version number difference is large enough, then transmitting a random subset of the version numbers can be more efficient than traversing the hash tree. To take advantage of this behavior and determine the transition point, DIP monitors network conditions, such as Trickle communication redundancy. Rather than always walk to the bottom of the hash tree, DIP starts sending precise version information when estimates reach a high enough value that suggest random scanning would be more efficient.

4.2 Metadata

DIP maintains a version number and unique key for each data item. As a result of having a unique key, it also assigns each data item an index in the range of $[0, T-1]$. DIP can describe a data item i as a tuple (k_i, v_i) where k_i is the data item key and v_i is its version number. The implementation of DIP we describe in this paper uses 32-bit version numbers, to preclude wrap-around in any reasonable network lifetime; smaller or larger values could also be used.

In addition to version numbers, DIP maintains estimates of whether an item is different. DIP stores estimates as small integers in the range of $[0, log(T)]$.[2] An estimate value of E means that DIP detected a difference at level E in the hash tree. With a tree branching factor of b, this means a hash that covers $\frac{T}{b^{E+1}}$ items.

Together, these two pieces of metadata are $log(V) + log(log(T))$ bits per data item, where V is the maximum version number. In practice, $log(V)$ is a small constant (e.g., 4 bytes). Compared to the basic Trickle, which requires $O(T)$ state, DIP requires slightly more, $O(T + T \cdot log(log(T)))$, as it must maintain estimates. In practice, the $log(log(T))$ factor is a single byte for simplicity of implementation, so DIP uses 5 bytes of state per data item in comparison to standard Trickle's 4 bytes.

4.3 Messages

The prior section described the per-item state each DIP node maintains with which to make protocol decisions. This section describes the types of messages DIP nodes use to detect and identify differences in data sets among their neighborhood. DIP, like Trickle, is an address-free, single-hop gossip protocol that sends all messages as link-layer broadcasts. DIP seamlessly operates across multihop networks by applying its rules iteratively on each hop. DIP uses three types of messages: data, vector, and summary.

Data Messages: Data messages transmit new data. They have a key k_i, a version number v_i, and a data payload. A data message unambiguously states whether a given item is different. On receiving a data message whose version number is newer than its own, DIP installs the new item.

Vector Messages: Vector messages hold multiple key, version tuples. The tuples may have non-consecutive keys. Vector messages, like data messages, unambiguously state whether a given item is different. They do not actually update, however.

Summary Messages: Figure 3 illustrates a complete summary message. Summary messages contain a set of summary elements and a salt value. Each summary element has two indices describing a set of version numbers, a summary hash over the set, and a bloom filter of the set. The salt value is a per-message random number that seeds the summary hash and bloom filter to protect from hash collisions.[3] When the size of the set covered by the summary hash is small enough, the filter can circumvent several hash tree levels to find which item is inconsistent. The number of summary elements in a summary message determines the branching factor of the DIP hash tree.

The summary hash function SH is $SH(i_1, i_2, s)$ where i_1 and i_2 are two indices representing left and right bounds of the search, and s is a salt value. Its output is a 32-bit hash value. For example, $SH(0, T-1, s)$ would be a hash over all the current version numbers for all data items. The specific function is a one-at-a-time hash using a combination of bit shifts and XOR operations, which an embedded microcontroller can compute without much difficulty.

Each summary message has a bloom filter, a probabilistic data structure that can be used to test an item's membership. Computing a bloom filter of length B bits involves taking a hash $BH(i, v, s)$ of each covered item, where i is the item index, v is the version number, and s is the salt. The result of each BH modulo B is a single bit position that is set to 1 in the filter[4]. If two bloom filters share an (index, version) tuple, they will both compute the same bit to be 1: there are no false negatives, where both sets agree but the filters differ. Bloom filters can have false positives, where two sets differ but have the same filter.

Each of the three message types provides DIP with different information. Data and vector messages can identify which data items are different. However, the detection cost of finding the item is $O(T)$. Summary messages can tell DIP that there is a difference, but not definitively which item or items differ. Summary messages have a $O(1)$ detection cost and a $O(log(T))$ identification cost. In practice, for reasonable T values (below 1,000), the bloom filter significantly reduces the identification cost.

[2] The implementation we describe in Section 5 actually stores values in the range $[0, log(T) + 2]$ to save a few bits in transmit state.

[3] We borrow the term "salt" from UNIX password generation [15].

[4] Bloom filters generally use k hash functions to set k bits. In DIP, $k = 1$

Salt s			
Begin_1	End_1	BloomFilter_1	SummaryHash_1
...			
Begin_n	End_n	BloomFilter_n	SummaryHash_n

Figure 3. DIP Summary Message

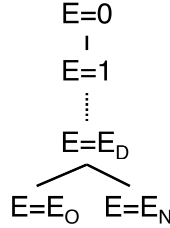


Figure 4. Estimate values. $E_D = \lfloor log_b(T) \rfloor$, **where** b **is the number of summary elements in a summary message (the branching factor).** E_O **denotes a neighbor has an older item, while** E_N **denotes a neighbor has a newer item.**

4.4 Updating Estimates

The prior two sections explained the state that DIP maintains and the messages it exchanges. As version numbers only change on data updates, estimates constitute most of the complexity of DIP's state management. This section explains how DIP adjusts its estimates in response to receiving packets. The next section explains how DIP decides what packets to send.

Section 4.2 stated that estimate values E are in the range of $[0, log(T)]$, where a 0 represents a belief that the data item is consistent with all neighbors, and value of $log(T)$ represents certainty that the item is different We denote $log(T)$ as E_D, as it denotes a difference exists. In addition to this range, DIP reserves two extra values: E_O, which denotes a nearby node has an older version, and E_N, which denotes a neighbor has a newer version. These two values are necessary because E_D only denotes that a node has a different version than one of its neighbors, but not the difference direction. Figure 4 summarizes the estimate values.

DIP uses the information in data, vector, and summary packets to update its estimates. On receiving a packet that indicates a neighbor has the same version number for an item, DIP decrements that item's estimate, to a minumum of zero. In the base case, when all nodes agree, estimates converge to zero. On receiving a packet that indicates a neighbor has a different version number, DIP improves its estimates for the relevant items as well as it can from the information it receives. More precisely, DIP adjusts estimate values with the following rules:

1. Receiving a vector or data message with an older version number sets that item to E_0 unless it is E_N.
2. Receiving a vector or data message with the same version number decrements that item's E, to a minimum of 0.
3. Receiving a data message with a newer version number, it updates the item and sets it to E_O.
4. Receiving a vector with a newer version number sets that item to E_N.
5. Receiving a summary element with a differing hash of length H sets E for all items that hash covers to be the maximum of their current E and $log(T) - log(H)$.
6. Receiving a summary element with a differing hash and differing bloom filter sets all items with a differing bloom filter bit to E_D.
7. Receiving a summary element with a matching hash decrements the E of all items the hash covers, to a minimum of 0.

The first two rules are identical for vector and data messages. In the case of receiving an older version number, DIP sets the item to be E_O, denoting it should send an update, unless the item is already E_N, denoting it should receive an update. Sending an update with an out-of-date item is a waste, so a node prefers to wait until it is up-to-date before forwarding updates.

The third and fourth rules define what occurs when DIP receives a packet with a newer version number. If the version number is in a vector message, DIP knows it needs an update, so sets the item to E_N. If the version number is in a data message, then DIP has received the update, which it installs. DIP then sets the item to E_O, as chances are another node nearby needs the update as well.

The last three rules define how DIP responds to summary messages. Like data and vector messages, receiving a matching summary decrements estimate values. When DIP receives a differing hash that can provide a more precise estimate, it increases its estimates to the value the hash will allow. If the bloom filter allows DIP to pinpoint that an item is different, then it sets that item's estimate to E_D. It determines this by checking the bloom filter bit for each item index, version pair. If the bit is not set, then there is certainly a difference. Because summary messages contain multiple summary entries, a single message can trigger one, two, or all three of rules 5, 6 and 7.

4.5 Transmissions

Section 4.4 described what happens on a message reception. This section describes how DIP decides which message types to transmit and what they contain.

DIP uses a Trickle timer to control message transmissions. In the maintenance state when no data items are different, the DIP Trickle timer is set at a maximum interval size of τ_h. As soon as a difference is detected, the Trickle

interval shrinks to the minimum interval of τ_l. When all estimates return to 0, DIP doubles its Trickle interval until τ_h is reached again.

DIP also uses hierarchical suppression to further prevent network flooding. Messages of the same type may suppress each other, but summary messages cannot suppress vector messages. This is because vector messages are more precise and are often used near the end of a search. This hierarchical suppression prevents nodes from suppressing more precise information.

DIP's transmission goal is to identify which nodes and items need updates. It accomplishes this goal by increasing estimate values. In the steady state, when all nodes are up to date, DIP typically sends summary messages whose summary elements together cover the entire set of version numbers. All DIP transmissions are link-layer broadcasts.

All tuples in vector messages and summary elements in summary messages have the same estimate value: a packet always represents a single level within the DIP hash tree. DIP always transmits packets which contain information on items with the highest estimate values. Together, these two rules mean that DIP transmissions are a depth-first, parallelized, search on the hash tree.

We now describe the decisions DIP makes. DIP's decisions are made based on local information to each node. This, coupled with the soft-state properties of the estimates, allow nodes running DIP to seamlessly leave and join both singlehop and multihop networks.

If an item has an estimate of $E = E_O$, DIP sends a data message. Because receiving an update causes DIP to set E to E_O, forwarding an update takes highest priority in DIP. Of course, hearing the same data message, or other packets that decrement E may prevent this data transmission; however, since DIP is based on randomized Trickle timers, in practice it soon discovers if a node is out of date and increases E to E_O.

If $E \neq E_O$, DIP compares the vector message and summary message costs of identifying a difference. If v key/version pairs can fit in a vector message and d data items need to be covered (computed from E), then DIP requires $\frac{d}{v}$ transmissions to scan through all d items. For summary messages, DIP requires $E_D - E$ (the number of levels until the bottom of the hash tree) transmissions. Assuming lossless communication, DIP transmits summaries when $(E_D - E) > \frac{d}{v}$ and vectors when $(E_D - E) < \frac{d}{v}$. This inequality means that if an item has an estimate of E_D or E_N, DIP always transmits a vector message.

Because real networks are not lossless, DIP adjusts its decision based on the degree of communication redundancy it heard in the last trickle interval. Hearing more messages in a given interval is related to a failure in the suppression mechanism and most likely caused by a loss rate or dense network. DIP accounts for this by changing the vector message calculation from $\frac{d}{v}$ to $\frac{d}{c \cdot v}$ where c is the redundancy.[5] If it takes $\frac{d}{v}$ vector messages to determine a new data item, but it is receiving c messages, then weight the required number of vector messages by a factor of c.

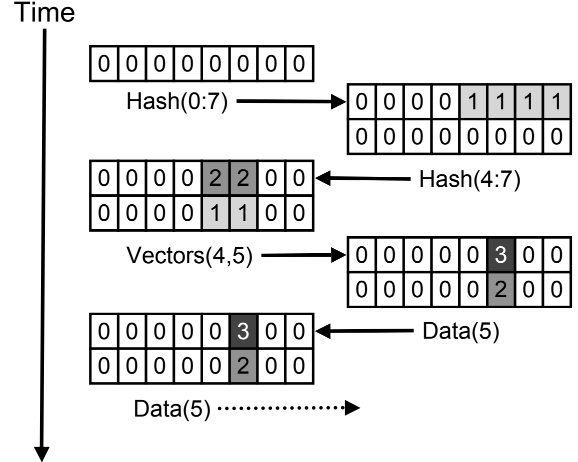


Figure 5. Two nodes exchanges summaries and vectors to determine that the left node needs an update for item 5. The arrays show each node's estimate values during each step of the packet exchange. In this example, summaries contain 2 summary elements.

When DIP transmits version messages, it selects a random subset of the data items which have the highest estimate value. As Section 3 showed, randomization is critical as communication redundancy increases with density.

When DIP transmits summary messages, it performs a single linear pass across the data items to find contiguous runs of items that have the highest estimate value. It generates summary elements for these items that are one level lower on the hash tree. For example, if DIP finds a run of items with an estimate of E, it generates summary elements which each cover $\frac{T}{b^{E+1}}$ items. While the items within each summary element must be contiguous, the summary elements themselves do not need to be.

Finally, when DIP transmits, it decrements the estimate values of all data items the transmission covers. In the case of data and vector messages, it decrements the estimates of the items version numbers that are in the packet. In the case of summary messages, it decrements the estimates of all items covered by a summary element.

4.6 Example

To give a concrete example of what a DIP search looks like and how the estimate update rules work, Figure 5 shows two nodes running DIP detecting that one node needs an up-

[5] We borrow the term c from Trickle's communication counter.

date for item 5. For simplicity, this example assumes bloom filters never help, and DIP only sends vector messages at the bottom of the hash tree. Each time a node transmits a summary message or a vector message, it reduces the estimate of the covered items. First, the left node transmits a full summary. The right node sees that the right summary hash (covering items 4–7) is different, so replies with summary hash of 4–5 and 6–7. The left node sees that 4–5 is different, so replies with the version numbers of 4 and 5. The right node replies with the data for item 5. The left node rebroadcasts the data in case any neighbors do not have it, which propagates the update acros the next hop. The nodes exchange a few more summary messages to make sure that the different summary hashes covered only one update.

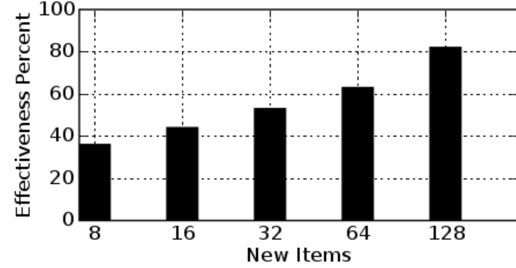
5 Evaluation

We divide our evaluation of DIP into two parts. In the first part, we look at improvements. We measure the effectiveness of DIP's two primary optimizations: bloom filters and using scans when the chance of hitting a new item is high. These experiments are all in simulation. In the second part, we look at comparisons. We compare DIP against a scan and search protocols in simulation and a real network. In simulation, we measure how parameters affect DIP's transmission costs by measuring the cost for the whole network to complete all updates. On a mote testbed, we compare the performance of the three algorithms for different N.

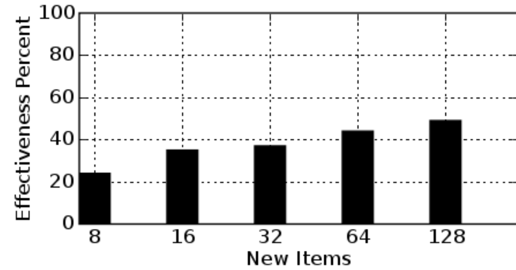
5.1 Methodology

We implemented DIP in TinyOS 2.0: it compiles to 3K of program code. In our implementation of DIP, we had 2 summary elements per summary message, and 2 key/version tuples per vector messages. The maximum Trickle interval was set at one minute, and the minimum interval at one second, though these numbers can be adjusted on a real deployment. We also implemented intelligent versions of the serial scan and search algorithms that use DIP-style estimates. Using estimates greatly improves the performance of these algorithms as they maintain some state on what to advertise. All three protocols use a single underlying Trickle. Scanning sends two (key,version) tuples per packet and sequentially scans through items. Searching uses a binary hash tree. When search identifies a different item, it updates the item and resets to the root of the tree.

To evaluate simple clique networks and multihop simulations, we used TOSSIM, a discrete event network simulator that compiles directly from TinyOS code [10]. In TinyOS 2.0, TOSSIM uses a signal-strength based propagation and interference model. Nodes have uniformly distributed noise along a range of 10dB and a binary SNR threshold of 4dB. The model distinguishes stronger-first from stronger-last



(a) D = 2.



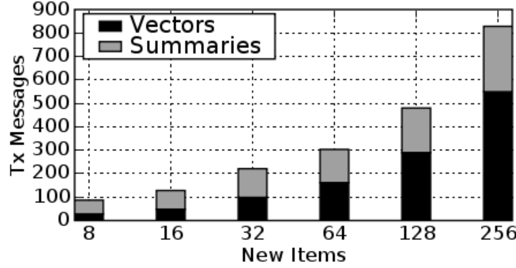
(b) D = 32.

Figure 6. Bloom filter effectiveness with L = 0%, T = 256 with two different densities and varying N.

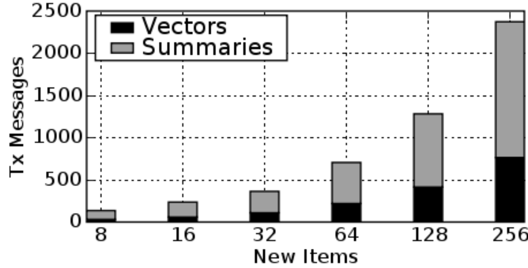
collisions, such that if a stronger packet arrives in the middle of a weaker one, the radio does not recover it (this is how the CC2420 radio on Telos and micaZ nodes behaves). Therefore, lossless communication in TOSSIM does not mean all packets arrive successfully: there can still be collisions. Fully connected networks are not collision-free because TOSSIM models radio RX/TX turnaround times.

To set a uniform loss rate, we configured TOSSIM to have noise values in the range of -110dBm to -100dBm and tuned the signal strength to obtain the desired loss rate. For a loss rate of 10%, we set link strengths to be -96.5dBm; for 20%, -97dBm; for 30%, -97.5dBm; for 40%, -98dBm, and for 50% we set the signal strength to be -99dBm. These values are not a simple linear progression as uniform loss distributions would suggest because TOSSIM samples noise several times during a packet. We ran several iterations for validity and averaged their results when applicable.

To collect empirical data, we ran our experiments on the Mirage testbed [2]. It is composed of 100 MicaZ motes that are distributed throughout an office environment. Each MicaZ consists of an Atmel Atmega128L microcontroller, 128K program flash, and an 802.15.4 compatible radio transceiver that transmits at 250kbps. We instrumented the DIP code to write to the UART various statistical information when events occur. Then using a UART to TCP bridge, we listened on on the port of each node and collected the information at millisecond granularity. Although there are



(a) L = 0.



(b) L = 20.

Figure 7. Message types used by DIP with T = 256, D = 32, two different loss rates, and a varying N.

100 nodes in the network, we could only connect to 77 of them for data gathering purposes.

5.2 Improvements

We evaluate how bloom filters improve DIP's performance by measuring how many summary messages successfully used a bloom filter to identify which item needed an update. Detecting with a bloom filter enables DIP to circumvent subsequent summary exchanges to traverse the tree, reducing the number of summary transmissions.

Figure 6(a) shows results from the very simple case of a pair of TOSSIM nodes with lossless communication. For different N, bloom filter optimizations have a hit rate of 35%-80%.

Figure 6(b) shows that bloom filters are also effective in a 32-node network, albeit less so than a single node pair. At higher densities, different nodes will require different items over time. Thus the issue is not that a single node requires many items, but rather many nodes require a few items, making N small for each node.

To better understand the decisions that DIP makes, we measured the distribution of transmitted summary and vector messages for different loss rates and new items in a 32 node clique. Figure 7(a) shows that as the number of new items increases, DIP uses more vectors and at a higher proportion. This is a result of DIP's dynamic adjustment to the number of new items. When the network becomes lossy as shown in Figure 7(b), an increase in N causes DIP to use more vectors and at a larger proportion, but this increase is not as fast as in a lossless network. High loss slows the increase of estimate values, leading to more summary messages and delaying when it uses vector messages. Although the X-axis on Figures 7 show N doubling, DIP's total transmission count does not double, as using pure searches might suggest: its message adaptation allows it to take advantage of high hit rates.

These results show that bloom filters improve DIP's detection and DIP dynamically improves its transmission policy based on network conditions.

5.3 Protocol Comparisons (TOSSIM)

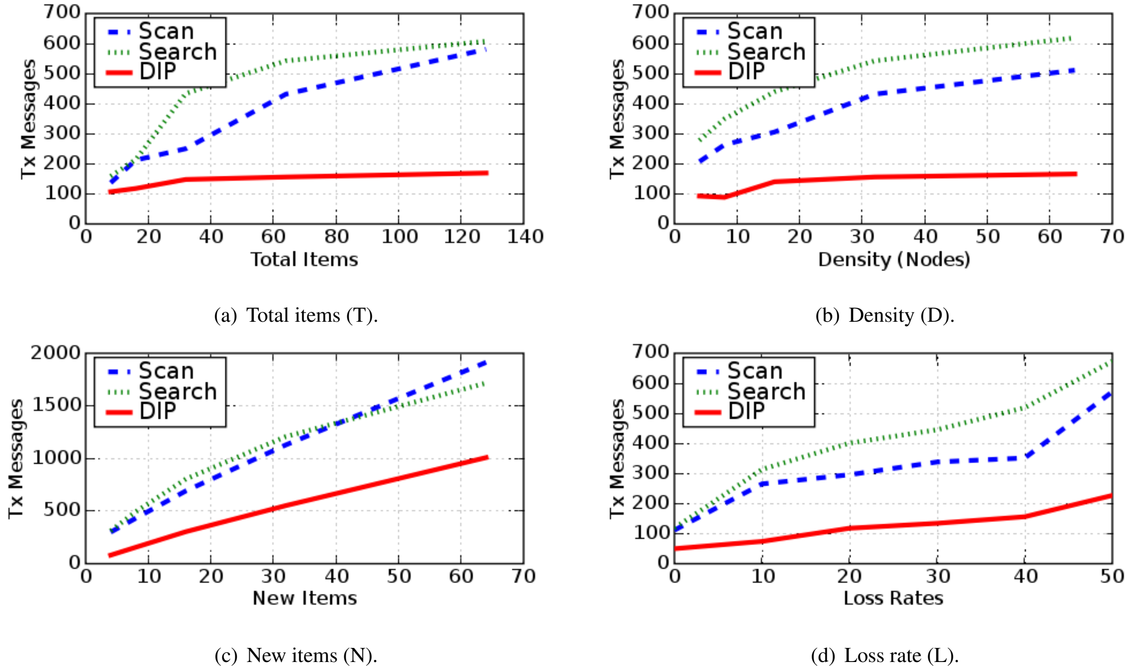
Because there are four possible parameters, we explore each one independently using TOSSIM. For simplicity, these experiments are all fully connected networks with a uniform loss rate.

In our first experiment, we evaluated the scalability of each protocol by having a constant number of new data items ($N = 8$) while varying up the total number of data items (T). We measured the total number of transmissions required to update each node. Figure 8(a) shows the results. As expected, searching requires $O(log(T))$ transmissions and scanning requires $O(T)$. DIP performs much better because it detects items quickly through the bloom filter. Even though messages are lost, a single successful bloom filter will identify the items and thus not require a full search. Furthermore, DIP keeps a narrow search by estimate-based back outs, making bloom filters more effective.

Figure 8(b) shows how density affects scanning and searching performance. Because both improved protocols handle redundancy, they have similar lines. Scanning, however, is better than searching because there is no overhead associated with scanning. At high densities, the improved scan protocol does not pay extra to cover its entire keyspace. DIP again performs the best because it has the benefits of both low overhead through bloom filters, but also being able to find items quickly through searching.

Figure 8(c) shows how performance changes as the number of new data items changes. When there are many new data items, the $log(T)$ search overhead becomes noticeable, but not significant due to the fact our search protocol implementation handles redundancy. The scan protocol scales linearly because each additional new item requires a constant amount of identification overhead. When almost all data items are new, it may seem counterintuitive that searching is better. The reason for this is because after nodes have been updating, the few remaining items at the end are hard to identify. Thus, the early scan advantage cancels out at the end. DIP scales linearly, but at a smaller constant factor.

Figure 8(d) shows how performance changes as the loss rate increases and the results are similar to that of figure 8(b)



(a) Total items (T).

(b) Density (D).

(c) New items (N).

(d) Loss rate (L).

Figure 8. DIP compared against improved scan and search protocols. By default, N=8, T=64, D=32, and L=40%: there are 8 new items out of 64, the clique has 32 nodes, and the packet loss rate is 40%. Each figure shows how varying one parameter affects DIP relative to other protocols.

due to redundancy.

In a multihop network, we are interested in how many transmissions are required for the whole network to detect new items. In multihop topologies, nodes must request new data from neighbors, while at the same time servicing other neighbors. We used the 15 by 15 sparse grid in the TinyOS 2.0 TOSSIM libraries, which uses Zuniga et al.'s hardware covariance matrix [23] to model link asymmetries and other real-world effects. We modified the multihop implementations of the scan and search protocols to perform better in multihop situations as well. The scan protocol re-advertises items 3-4 times after receiving an item, while the search protocol uses estimates to back out rather than resetting.

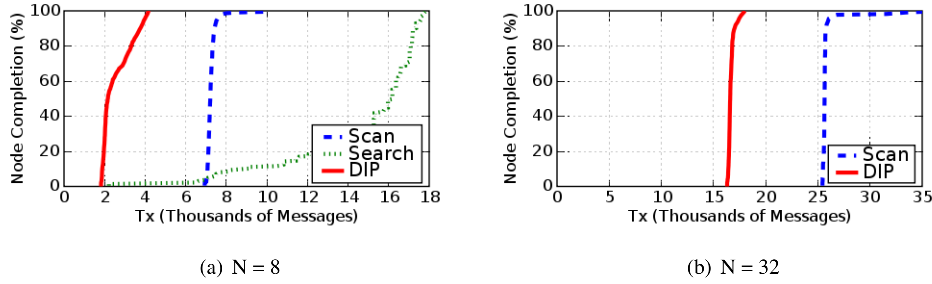
Our first experiment examines transmission costs when $N = 8$ and $T = 256$. Figure 9(a) shows the results. The scanning protocol completes a large majority of nodes very close together, but the last few nodes take exceedingly long. When different items are discovered, the scan protocol repeatedly transmits high estimate items until estimate levels have decreased. At the end, when only a few nodes have old items, scanning cannot find those few items very well. Searching performs far worse because nodes are both senders and receivers in multihop topologies and searching requires efficient pairwise communication. DIP has similar performance to scanning until around half the nodes are complete. Afterwards, it begins searching and completes the remaining nodes. The gap between when the first and last node completes is marginally smaller in DIP. DIP's tail begins to occur when a majority of the nodes have finished. Finally, DIP uses only 40% as many transmissions as scanning, a 60% improvement.

In the second experiment, shown in Figure 9(b), there were 32 new items instead of 8. DIP completed disseminating to all nodes within 18,000 transmissions. The overhead of searching caused the search protocol (not shown) to not finish within 35,000 transmissions. Multihop topologies force nodes to send and receive in different cells. This is problematic for searching, which requires pairwise communication to succeed. In contrast, the scan protocol finished with just over 35,000 transmissions and exhibited a very long tail, as scans are inefficient at finding the last few items. DIP has a shorter tail due to its ability to identify items through the bloom filter.

5.4 Protocol Comparisons: Mirage

We ran two testbed experiments with $T = 64$. We measured how many transmissions updated the whole network, and timed out each experiment after 400 seconds.

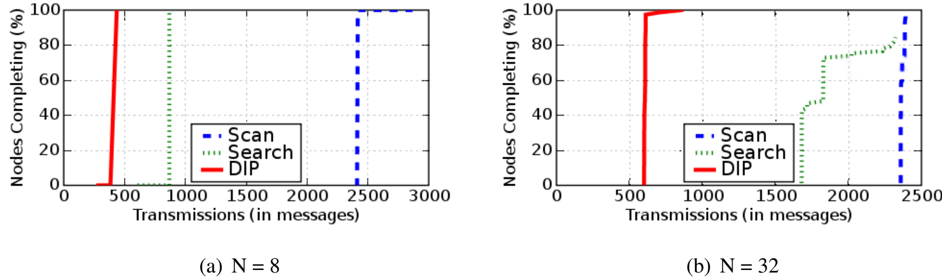
Figure 10(a) shows the per-node completion CDF for $N = 8$. DIP completed with 436 transmissions, while the search and scan protocols required 868 and 2867 respectively. DIP and the search protocol had steep curves meaning the nodes completed within a short time of each other. This is because DIP and the search protocol (which was



(a) N = 8

(b) N = 32

Figure 9. Transmission costs to complete reception of all new data items on 15 by 15 grid with 64 total items and two different values of new items (TOSSIM).



(a) N = 8

(b) N = 32

Figure 10. Transmission costs to complete reception of all new data items on Mirage with 64 total items and two different values of new items.

modified) back out after receiving or sending new items, keeping searches narrow enough for nodes in other cells to find items. The modified scan protocol is also steep, but has a long tail, which is caused from being unable to find the last few items quickly after most of the network has finished. Furthermore DIP finished with a much better overall time compared to the other modified protocols. This is due to DIP's ability to identify items faster through its bloom filters. In terms of time, DIP took 86 seconds to deliver all 8 items while scanning took and searching took 107 and 143 seconds, respectively, a speedup of 24-60%.

Figure 10(b) shows results for $N = 32$. With more new items, search's overhead grows, and its performance relative to scanning degrades. While DIP was able to disseminate to every node in approximately 860 packets, neither scanning nor searching completed in over 2500 packets: both timed out. As dissemination layers today use scanning algorithms, DIP reduces the cost by up to 60%. While neither scanning nor searching was able to complete in 400 seconds, DIP took 131 seconds, a speedup of over 200%. This means, for example, when introducing new nodes to a network, DIP will bring them up to date to configuration changes up to 200% faster than existing approaches.

6 Related Work

DIP draws heavily from prior work in reliable data dissemination. Unlike protocols for wired networks, such as SRM [6] and Demers' seminal work on epidemic database replication [5], DIP follows Trickle's approach of using local wireless broadcasts [11]. Trickle's suppression and rate control protects DIP from many of the common problems that plague wireless protocols, such as broadcast storms [17]. Existing systems, such as Deluge [8], Maté [9], and Tenet [7] use protocols that assume the number of data items is small. DIP relaxes this assumption, enabling scalable dissemination for many items. Unlike flooding and broadcast protocols such as RBP [19], DIP provides complete reliability as long as the network is connected.

DIP's hashing is similar to Merkle hash trees [14], a common mechanism in secure systems. As Merkle hash trees need to minimize computation, each level is a hash of the hashes of the level below it. As the tree stores all of these hashes, changing a single leaf requires only updating $log(n)$ hashes. In contrast, DIP dynamically computes hashes of leaf values on demand. This approach stems from how the resource tradeoffs between sensor nodes and traditional storage systems differ: on a sensor node, the RAM to store a hash tree is expensive, while CPU cycles to hash a range of version numbers is cheap.

Bloom filters have a long history in networked sys-

tems, including web caching, Internet measurements, overlay lookups, and keyword searches [1]. Keyword searches are the most similar, but with an opposite purpose: while they find similarities in filters, DIP seeks to find differences. Bloom filters are commonly used in distributed and replicated IP systems (e.g., PlanetP [4]), but to our knowledge DIP represents their first use in wireless dissemination.

The tradeoffs between deterministic and randomized algorithms appear in many domains. At one extreme of data reliability, standard ARQ algorithms repeat lost data verbatim. At the other extreme, fountain codes [13] used randomized code blocks to reliably deliver data. At the cost of a small data overhead ϵ, a fountain code requires no explicit coordination between the sender and receiver, trading off a bit of efficiency for simplicity and robustness. There are also many techniques that lie between these extremes, such as incremental redundancy (or Hybrid ARQ) [18], which randomizes code bits sent on each packet retransmission. Similarly, Demers et al.'s landmark paper on consistency in replicated systems explored the tradeoffs between deterministic and randomized algorithms [5]. The Trickle algorithm adds another level of complexity to these tradeoffs due to its inherent transmission redundancy.

7 Conclusion

This paper presents DIP, an adaptive dissemination algorithm that uses randomized and directed algorithms to quickly find needed updates. To achieve this, DIP maintains estimates of the probability that data items are different and dynamically adapts between its algorithms based on network conditions. This adaptation, combined with bloom filters, enables DIP to efficiently support disseminating a large number of data items and achieve significant performance improvements over existing approaches. The tradeoffs between searching and scanning show a basic tension between deterministic and randomized algorithms. Acting optimally on received data works best in isolation, but in the case of redundancy, multiple nodes each taking a sub-optimal part of the problem can together outperform a locally optimal decision. DIP leverages this observation to greatly improve dissemination efficiency; it remains an open question whether other wireless protocols can do the same.

Acknowledgements

This work was supported by generous gifts from Intel Research, DoCoMo Capital, Foundation Capital, the National Science Foundation under grants #0615308 ("CSR-EHS") and #0627126 ("NeTS-NOSS"), and a Stanford Terman Fellowship.

References

[1] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. 1(4):485–509, 2004.

[2] B. Chun, P. Buonadonna, A. AuYoung, C. Ng, D. Parkes, J. Shneidman, A. Snoeren, and A. Vahdat. Mirage: A microeconomic resource allocation system for sensornet testbeds. In *Proceedings of the 2nd IEEE Workshop on Embedded Networked Sensors (EmNets)*, 2005.

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

[4] 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. In *HPDC '03: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, page 236, Washington, DC, USA, 2003. IEEE Computer Society.

[5] 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.

[6] 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.

[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 ACM Conference on Embedded Networked Sensor Systems (Sensys)*, 2006.

[8] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94, New York, NY, USA, 2004. ACM Press.

[9] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *Second USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2005.

[10] 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.

[11] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code maintenance and propagation in wireless sensor networks. In *First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2004.

[12] C.-J. M. Liang, R. Musaloiu-Elefteri, and A. Terzis. Typhoon: A reliable data dissemination protocol for wireless sensor networks. In *Proceedings of 5th European Conference on Wireless Sensor Networks (EWSN)*, pages 268–285, 2008.

[13] M. Luby. Lt codes. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science*, pages 271–282, 2002.

[14] R. Merkle. Secrecy, authentication, and public key systems. Ph.D. dissertation, Dept. of Electrical Engineering, Stanford University, 1979.

[15] R. Morris and K. Thompson. Password security: a case history. *Commun. ACM*, 22(11):594–597, 1979.

[16] V. Naik, A. Arora, P. Sinha, and H. Zhang. Sprinkler: A reliable and energy efficient data dissemination service for extreme scale wireless networks of embedded devices. *IEEE Transactions on Mobile Computing*, 6(7):777–789, 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 ACM/IEEE international conference on Mobile computing and networking*, pages 151–162. ACM Press, 1999.

[18] E. Soljanin. Hybrid arq in wireless networks. DIMACS Workshop on Network Information Theory, 2003.

[19] F. Stann, J. Heidemann, R. Shroff, and M. Z. Murtaza. Rbp: robust broadcast propagation in wireless networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 85–98, New York, NY, USA, 2006. ACM.

[20] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless se nsor networks. In *Proceedings of the Second European Workshop of Wireless Sensor Netw orks (EWSN 2005)*, 2005.

[21] L. Wang. MNP: multihop network reprogramming service for sensor networks. In *Proceedings of the Second ACM Conference On Embedded Networked Sensor Systems (SenSys)*, pages 285–286, New York, NY, USA, 2004. ACM Press.

[22] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: using rpc for interactive development and debugging of wireless embedded networks. In *IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks*, pages 416–423, New York, NY, USA, 2006. ACM.

[23] M. Zuniga and B. Krishnamachari. Analyzing the transitional region in low power wireless links. In *First IEEE International Conference on Sensor and Ad hoc Communica tions and Networks (SECON)*, 2004.