2010 29th IEEE International Symposium on Reliable Distributed Systems

# Fixed Cost Maintenance for Information Dissemination in Wireless Sensor Networks

Rajesh Krishna Panta, Madalina Vintila, Saurabh Bagchi
Dependable Computing Systems Laboratory (DCSL), Purdue University
465 Northwestern Avenue, West Lafayette, IN 47907
Email: {rpanta,mvintila,sbagchi}@purdue.edu

***Abstract*—Because of transient wireless link failures, incremental node deployment, and node mobility, existing information dissemination protocols used in wireless ad-hoc and sensor networks cause nodes to periodically broadcast "advertisement" containing the version of their current data item even in the "steady state" when no dissemination is being done. This is to ensure that all nodes in the network are up-to-date. This causes a continuous energy expenditure during the steady state, which is by far the dominant part of a network's lifetime. In this paper, we present a protocol called Varuna which incurs a constant energy cost, independent of the duration of the steady state. In Varuna, nodes monitor the traffic pattern of the neighboring nodes to decide when an advertisement is necessary. Using testbed experiments and simulations, we show that Varuna achieves several orders of magnitude energy savings compared to Trickle, the existing standard for dissemination in sensor networks, at the expense of a reasonable amount of memory for state maintenance.**

***Keywords*-Dissemination, Trickle, steady-state**

## I. Introduction

Wireless ad-hoc and sensor networks use various dissemination protocols [1], [2], [3], [4] for one-to-many communication to disseminate information from the fixed infrastructure to all or a subset of nodes in the network. Examples of such communication are base station sending code updates for *wireless reprogramming* of the network, and sending network commands or queries to nodes in the network. These dissemination protocols incur energy expenditure not only during the information dissemination phase but also during the *steady state* when no dissemination is actually being done. The need for energy expenditure in the steady state arises from the possibility of dynamic changes to the network topology. Such changes are caused by the failure-prone nature of radio communications, node mobility, and incremental node deployment. Unless a network can rule out all these causes of changes to the network (which it rarely can), then it becomes essential to perform some state updates to the network in the steady state, if only to verify that no such change has occurred. For the rest of the paper, we will use the term *steady state* to denote the state when no one-to-many information dissemination is taking place in the network, though the network will be performing other functionality, such as data collection from the sensor nodes toward the base station.

Because of transient failures, nodes may remain disconnected from other nodes in the network for some time and may miss the information dissemination that had occurred during that period. After they come out of disconnection, they must be able to detect the data item inconsistency and then initiate the process to become up-to-date. Inconsistency of information may also happen due to incremental node deployment or node mobility which causes a node to move into a region where its neighbors have received an update. For ease of exposition, we are going to refer to all these events that can cause a node to get out-of-date as *topology changes*. Importantly, since these topology changes can happen at arbitrary time points and are not scheduled, any protocol to keep the network up-to-date needs to execute on a continuing basis.

Inconsistent data items can have serious consequences. For example, in wireless reprogramming, running an old version of the code could lead to wrong computation leading to erroneous aggregation and finally incorrect data being received at the base station. Even worse, different versions of the code in the network can cause the the network to be partitioned.

The traditional way of enhancing the dependability of the dissemination protocols in the presence of the unpredictable topology changes is periodic advertisements (or some variations) of some *metadata* by each node. For example, this is the approach used in the Trickle algorithm [5] which is used as the basic building block by most of the current dissemination protocols [1], [2], [6]. The metadata, a compact representation of the data item, has to be such that, by inspecting the metadata, a node can determine if it needs the corresponding data item for it to become updated. A common case of metadata is a monotonically increasing version number for the data item that the node currently has. When a node hears an advertisement from a neighbor with a newer version of the data item than it currently has, both enter the dissemination phase through which the data item is actually exchanged. The actual dissemination can be accomplished through one of several well-known protocols such as Deluge [1], Stream [2], etc.

Radio communication is often the most significant source of energy consumption in sensor networks. The problem with periodic advertisements is that the steady state energy cost increases linearly with the steady state interval, which is the most dominant phase in a node's lifetime. In fact, in practice, learning *when* to disseminate a data item can be much more costly than disseminating the data item itself and as a result, steady state energy cost is several orders of magnitude higher than the energy cost during actual data item dissemination phase. For example, Deluge, the default reprogramming pro-

1060-9857/10 $26.00 © 2010 IEEE
DOI 10.1109/SRDS.2010.15

tocol for TinyOS-based sensor networks, performs periodic broadcast of the advertisement packets every 2 minutes in the steady state. Periodic advertisements at this rate for one day requires the same amount of radio transmissions as disseminating a 25 KB program code. The steady state energy cost can be reduced by increasing the advertisement interval. However, the interval cannot be increased significantly because it increases the *detection latency*, the time taken by the nodes to determine whether they have inconsistent data items. This in turn increases the probability of the communication between nodes with different versions of the data item, which is a serious concern as we have seen above.

Our holy grail is to break this barrier of continuously increasing energy expenditure for state maintenance in the steady state of the network, and achieve a constant maintenance cost, independent of the duration of the steady state. We achieve this goal in the common case through our protocol called *Varuna* [1]. Common case implies reasonable link reliabilities and reasonable memory allocation for state maintenance. To achieve this, we make a fundamental observation that if the neighborhood topology and the metadata of a node have not changed since its last advertisement transmission, then the node does not need to send any advertisement message. In periodic advertisement schemes like Trickle, practically most of the advertisements in the steady state are, therefore, unnecessary. A node can determine trivially whether its metadata has changed, through a local lookup. However, determining whether the neighborhood topology has changed is difficult and requires wireless communication among the neighboring nodes. The periodic advertisement in Trickle is essentially a way for a node to check if the neighborhood topology has changed, and if so, inform the "new" neighbors about its metadata. In Varuna, on the contrary, a node transmits advertisement messages only when required—either its metadata or local neighborhood or both have changed. Let us group all communication arising from a node into two categories—one-to-many information dissemination kind, and all the others. We will call this latter category *User Application (UA)* traffic. In Varuna, each node observes the communication pattern of UA packets of its neighbors to determine if its neighborhood has changed. Advertisement message is transmitted only when a node hears radio transmission from "new" neighbors.

The problem with the above observation is that it is impossible to determine the change in neighborhood topology based solely on communication pattern of the neighbors. For example, application-specific decisions at a node may cause it not to use a link to its neighbor. Therefore, we complement the first observation with a *second* one. It is critically necessary for a node to be up-to-date only when it is communicating with other nodes. This is so that stale metadata is localized to the out-of-date node only and is not propagated to other nodes. At worst, the out-of-date node may have to discard the results of some local computation that it might have performed while it was out-of-date. Varuna achieves a constant steady state energy expenditure at the cost of a small amount of state maintenance (of the order of 100 Bytes) through which each node keeps track of which nodes it has heard UA packets from, since it last got updated.

[1]Varuna is a God in the Hindu pantheon who maintains order in the universe.

Our contributions in the paper are:
1. We present the first protocol for maintenance of up-to-date information in a multi-hop wireless network that does not incur a monotonically increasing cost (in terms of energy) with the length of the steady state period.
2. We show how a reasonable amount of local state maintenance can avoid the energy cost of transmissions to determine when a node is (possibly) out-of-date.
3. Our experimental and simulation results show that the actual gains realized over the current state-of-the-art is two orders of magnitude, for a steady state duration of just few days. This benefit grows linearly with increasing duration.

## II. Trickle Overview and Problems

Trickle is the standard steady state algorithm for one-to-many information dissemination in sensor networks [1], [2], [6], [3] and forms our reference comparison point. In Trickle, each node broadcasts its advertisement message once every time interval randomly chosen from $[\tau/2, \tau]$ if it has not heard more than $k$ identical advertisements in that interval. An advertisement contains the metadata about the data item the node has. The metadata in this context is the version number of the data item. When a node hears an advertisement with different metadata than its own, it sets $\tau = \tau_l$. When it hears advertisement with same metadata as its own, it keeps on doubling $\tau$ in the successive intervals. A protocol like Deluge [1] which uses Trickle for code dissemination stops this increment after reaching some threshold, $\tau = \tau_h$. The suppression of advertisement broadcast (if a node has heard more than $k$ identical advertisements in an interval) is necessary to ensure that redundant advertisements are not broadcast and the scheme is scalable with high node density. Clearly, without loss, collision, and with perfect time synchronization of the interval $\tau$ among the sensor nodes, the number of advertisement broadcasts in any time interval within a single hop is bounded by $k$. The authors of Trickle show that with these practical conditions, the number of advertisement broadcasts in a single period $\tau$ is $O(logN)$ where $N$ is the number of nodes within a single hop. However the number of advertisements in a given period $T(>> \tau)$ is $O(T)$. This linear increase in maintenance cost with time results in continuous energy drain in the steady state.

The steady state energy cost can be reduced by increasing the advertisement period. However, this has several problems. First, the increase in the advertisement period also increases the detection latency, the time taken by a node to realize that it is out-of-date. Detection latency and steady state energy cost have an inverse relationship—a smaller advertisement period decreases the detection latency but increases the maintenance cost and vice-versa. Trickle handles this tradeoff by decreasing the advertisement period when data item inconsistency is

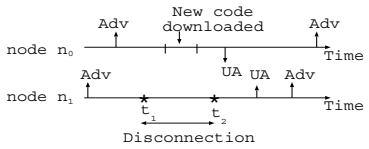



Fig. 1. If advertisement interval is greater than code download time, inconsistent nodes may communicate.

detected and increasing it when nodes are up-to-date. Thus, Trickle decreases the propagation time during the dissemination phase and reduces the maintenance cost during the steady state. To ensure acceptable detection latency, advertisement interval cannot be increased arbitrarily. In Deluge, the maximum interval is 2 minutes by default. Second, although increasing the advertisement period reduces the energy cost, this is only a constant order improvement. The steady state cost still increases linearly with the steady state duration.

Third, if advertisement interval is greater than the time required by a node to download the code, this makes it possible to have communication between the nodes with different versions of the code. As shown in Figure 1, let us suppose that a node $n_1$ goes into a transient *disconnection state* during the time interval $[t_1, t_2]$ during which it misses a code update. We define a node to be in disconnection state if it has no functioning incoming and outgoing link. Let its neighbor $n_0$ download the new code during this interval. Since the advertisement interval is large, $n_0$ and $n_1$ may exchange UA packets before the next advertisement, i.e. before they detect the code inconsistency. As mentioned earlier, this may result in undesired network behavior. To avoid this problem, the advertisement interval must be *less than* the code download time of a node. Code download time of a node is generally in the order of few minutes. Thus the advertisement interval cannot be made arbitrarily large, which increases the steady state energy cost. In this discussion, we have used Trickle as an example. *All one-to-many information dissemination protocols in wireless networks today suffer from the problem mentioned above—monotonically increasing energy cost with the duration of the steady state.*

## III. Strawman Solution Proposals

Without loss of generality, we present Varuna with respect to wireless code dissemination in sensor networks. Varuna, however, is applicable to any one-to-many dissemination protocol in wireless ad-hoc networks. We first explore several intuitively appealing approaches that can reduce the maintenance cost for the steady state interval and point out the flaw that besets each approach.

### A. Piggybacking metadata in UA packets

Instead of periodically advertising the metadata, a node can piggyback the metadata in each UA packet transmission because the energy cost of piggybacking is significantly lower than transmitting a separate advertisement packet. However, since metadata can be quite large, piggybacking reduces the number of bytes available in a packet for the application. For example, in Deluge, for each user application, the metadata is 12 bytes and, for disseminating multiple applications, the metadata is even larger. Instead of piggybacking the entire metadata, only a hash value of the metadata can be piggybacked. However, this is again a constant order improvement and the overhead energy cost due to piggybacking in *each* UA packet transmission increases linearly with the steady-state time like in Trickle. A much simpler approach would be for the neighboring nodes to exchange their metadata before every wireless communication. Obviously, the energy cost increases linearly with the number of communications in such scheme and thus violates our requirement for constant cost quiescent period.

### B. Checking neighborhood periodically

As mentioned above, if a node's metadata and neighborhood have not changed since its last advertisement transmission, it does not need to advertise its metadata. A node can check if its metadata has changed using local information, without communicating with its neighbors. For verifying if its neighborhood has changed, instead of energy-intensive proactive verification by broadcasting advertisement messages periodically as in Trickle, a node can simply listen for UA packet transmissions from its neighbors. However, it is generally impossible for a node to derive the information about the change in neighborhood solely using the traffic pattern from its neighbors. Various application-specific decisions may cause a node not to use a particular link, making it impossible for its neighbor to know if its neighborhood has changed. However, it is practically sufficient for a node to verify the freshness of its metadata, *not with all nodes in its neighborhood, but only with the node with which it has communication (i.e., a node from which it receives a UA packet or to which it sends a UA packet)*. When the nodes are not communicating, the consequence of not being up-to-date is localized to the out-of-data node only and is thus not as serious.

### C. Staying up-to-date only with communicating nodes

An intuitive scheme would be for each node to maintain a neighbor table consisting of ids of the nodes from which it has heard UA packets in the last threshold time duration, call it the refresh interval, $T_{REF}$. In the next $T_{REF}$ interval, if a node $n_1$ receives a UA packet from a node $n_2$ which does not exist in its neighbor table, $n_1$ and $n_2$ exchange advertisements through which they determine if they are up-to-date with respect to each other. If they are, then $n_1$ accepts the UA packet from $n_2$. Otherwise, $n_1$ and $n_2$ enter the dissemination state through which their information is made consistent. If $n_1$ cannot exchange the metadata with $n_2$ after a set number of attempts (due to link failures), it discards the UA packet from $n_2$ and goes back to the steady state.

In this scheme each node essentially checks if its neighborhood has changed since the last $T_{REF}$ interval using the already existing UA packet transmissions of the neighboring nodes. This scheme significantly reduces the steady state

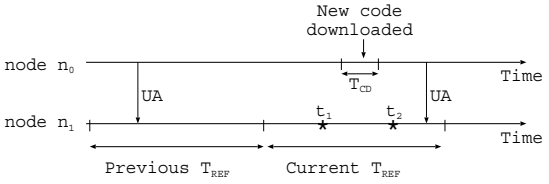



Fig. 2. Correctness issue if $T_{REF} > T_{CD}$.

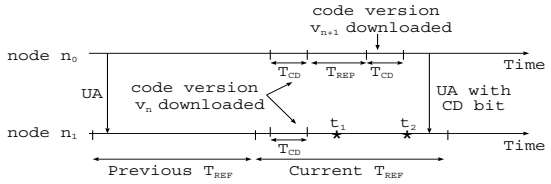



Fig. 3. $T_{REF}$, the refresh interval cannot made larger than $T_{REP}$, the minimum time between two successive code downloads, for correctness reasons.

energy cost because a node needs to advertise its metadata only when its neighbor table—a measure of neighborhood topology—changes. As long as the neighbor table does not change and the node has itself not received an update, advertisements are not transmitted.

The problem with this scheme is the difficulty in choosing $T_{REF}$ properly. It should be sufficiently large so that with a high likelihood, a node hears UA packets from all its neighbors within each $T_{REF}$. Otherwise, the node will needlessly perform the advertisement exchange, only to realize that both were up-to-date. In the extreme case when the neighbor table changes every $T_{REF}$ interval, this scheme is equivalent to Trickle with advertisement period equal to $T_{REF}$. Furthermore, $T_{REF}$ cannot be increased arbitrarily—if $T_{REF}$ is larger than the time to download the code $T_{CD}$ by a node, this scheme fails. Figure 2 illustrates this. Here a node $n_1$ goes into the disconnection state for time interval $[t_1, t_2]$ during which its neighbor $n_0$ downloads the new version of the code, which $n_1$ misses. $n_1$ receives a UA packet from $n_0$ in the previous $T_{REF}$ interval and since $T_{REF} > T_{CD}$, let us suppose $n_1$ receives a UA packet from $n_0$ in the current $T_{REF}$ interval also. As a result, $n_1$ thinks that it is up-to-date with respect to $n_0$ since its neighbor table has not changed. The code inconsistency, thus, goes undetected for a potentially unbounded period of time.

### D. Informing neighbors of code downloads

The above correctness problem can be solved by having each node piggyback a *Code Downloaded (CD)* bit in each UA packet transmission for $T_{REF}$ interval, each time after downloading the new version of the code. For example, in Figure 2, when $n_1$ comes out of disconnection, if it hears a UA packet from $n_0$ in the current $T_{REF}$ interval, it will have the CD bit turned on. Then $n_1$ realizes that it has not downloaded the new version of the code in the last $T_{REF}$ interval, but $n_0$ has. Thus the code inconsistency is detected.

However, even with this revised scheme, $T_{REF}$ cannot be made larger than $T_{REP}$, the minimum time interval between two consecutive reprogramming procedures. Figure 3 illustrates this. Here $n_0$ and $n_1$ download version $v_n$ of the code in the current $T_{REF}$ interval. But after that, $n_1$ goes into the disconnection state for interval $[t_1, t_2]$. During this time, $n_0$ downloads $v_{n+1}$ version of the code, which $n_1$ misses. When $n_1$ comes out of disconnection, it receives a UA packet from $n_0$ with CD bit turned on and believes that it is up-to-date with $n_0$ because $n_1$ also downloaded code in the current $T_{REF}$ interval. Hence the code inconsistency goes undetected again for a potentially unbounded time.

Instead of piggybacking the CD bit, if $n_0$ had piggybacked "the number of times the code was downloaded in the current $T_{REF}$" or "the latest version of the code downloaded in $T_{REF}$", $n_1$ would have detected the inconsistency. But this is generally not possible since a sensor node may be running multiple applications and thus it would need to explicitly say *which versions* of *which applications* were downloaded in the last $T_{REF}$ interval. This information is too large to be piggybacked in every UA packet for a "large" $T_{REF}$ interval.

Any scheme that uses threshold (refresh) time intervals to check if the neighborhood has changed between such intervals has a fundamental performance problem—since UA can be arbitrary, no matter how large a $T_{REF}$ is chosen, a neighbor can be such that it sends UA packet at every other $T_{REF}$ interval, causing the neighbor table to change in every $T_{REF}$ interval. As a result, the node needs to advertise in every $T_{REF}$ interval and thus, the energy expenditure of such scheme is equivalent to Trickle with advertisement period equal to $T_{REF}$.

## IV. Varuna Design

Based on the above observation that a node cannot determine if its neighborhood topology has changed by monitoring the communication pattern for a finite time duration, we arrive at a very simple maintenance algorithm, called Varuna, that does not use the notion of refresh interval. The above approaches (Sections III-B and III-D) try to make sure that the code inconsistency is detected when nodes with different versions of the code communicate. Varuna relaxes this requirement and, instead, guarantees that the following invariant is satisfied—*When a node receives a packet from another node with a lower version of the metadata than its own, the metadata inconsistency is detected by the receiving node.* This invariant also implies that Varuna achieves eventual consistency—even though a node $n_1$ may not detect inconsistency while it is receiving packets from a node $n_2$ which has a higher version of the metadata than $n_1$, eventually when $n_2$ receives the packet from $n_1$, the inconsistency is detected. We believe this relaxed form of consistency is satisfactory in most application contexts and is necessary in practice to achieve a constant cost of state maintenance. With Varuna's invariant, information does not flow from out-of-date nodes to up-to-date nodes, and thus, the erroneous result is not propagated in the network. Note that base stations (or nodes close to them)

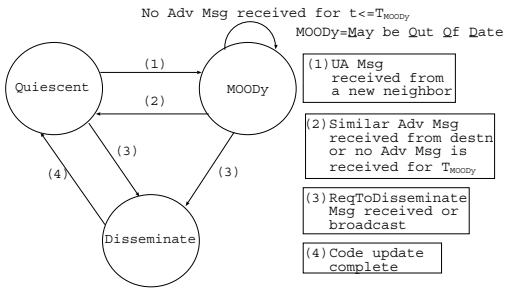



Fig. 4. State transition diagram of Varuna

can be assumed to be always up-to-date. Thus, erroneous results from out-of-date nodes will not be collected by the base station. The relaxed invariant of Varuna is sufficient to most practical application contexts where sensor nodes report the sensed values to base stations or fixed infrastructure. Before presenting a formal description, we first present an overview of Varuna. Figure 4 shows the state transition diagram of Varuna.

### A. Design Overview

Each node maintains a neighbor table consisting of the ids of the nodes from which it has received any packet since the last time it updated its metadata (i.e. downloaded the new version of the code). When a node is booted up or its metadata is updated, the neighbor table is cleared and the node goes to the Quiescent state. When a node $n_1$ receives a UA packet from a node $n_2$, it checks if $n_2$ exists in its neighbor table. The case of overflow of the neighbor table is discussed later. If $n_2$ exists in the table, $n_1$ accepts the packet. If it does not, $n_1$ suspects that it (or $n_2$) **M**ay be **O**ut **O**f **D**ate (MOODy) and goes to the MOODy state where it tries to verify if $n_2$ has the same version of the metadata as that of $n_1$. We will shortly explain how this is done. If $n_1$ finds that both have same version of the metadata, it inserts $n_2$ in its neighbor table, goes to the Quiescent state, and accepts the packet from $n_2$. Otherwise, it goes to the Disseminate state. In the Disseminate state, the out-of-date node receives the new version of the code from the up-to-date node(s) using any one of the available dissemination protocols [1], [2], [6], [7], [8]. Varuna's design is orthogonal to that of the dissemination protocol and it can work with any of them.

**Verifying if metadata is up-to-date**: In the MOODy state, $n_1$ broadcasts *Advertisement* packet containing its metadata, source id, and a field called *dest* set to $n_2$. Let $v_i^k$ $(i = 1, 2, ..., n)$ represent the version numbers of $n$ application codes (or data items) present in node $k$. When $n_2$ receives the Advertisement packet with *dest* set to its node id, it compares the received metadata with its own. If $n_2$ finds that it needs an update (i.e. $v_i^{n_2} < v_i^{n_1}$ for any $i$), it broadcasts a *ReqToDisseminate* packet and transitions to the Disseminate state. When $n_1$ and other neighbors of $n_2$ receive the *ReqToDisseminate* packet, they also go to the Disseminate state. If $n_2$ finds that it does not need an update (i.e. $v_i^{n_2} \geq v_i^{n_1}$ for all $i$), it broadcasts an Advertisement packet with *dest* set to NULL. When $n_1$ receives this advertisement packet, it verifies if it needs an update. If it does not (i.e. $v_i^{n_1} = v_i^{n_2}$ for all $i$), it goes back to the Quiescent state, adds $n_2$ to its neighbor table, and accepts the packet received from $n_2$. If $n_1$ needs an update (i.e. $v_i^{n_1} < v_i^{n_2}$ for any $i$), it broadcasts a *ReqToDisseminate* packet. Neighbors of $n_1$ (including $n_2$) go to the Disseminate state after receiving this packet. Advertisement and *ReqToDisseminate* messages are transmitted using random backoff intervals—[0,ADV_RAND] and [0,DISS_RAND] respectively, to avoid collisions due to concurrent transmissions from nearby nodes.

The use of *dest* field in the broadcast Advertisement message avoids the transmission of redundant advertisements in the neighborhood of the MOODy node $n_1$. A node which receives the advertisement message with *dest* set to NULL or *dest* not set to its own id will reply with its own advertisement message only if required (i.e. either the receiver or the sender needs an update) and *suppression threshold* has not been reached. For example, when neighbors of a MOODy node $n_1$ other than $n_2$ hear advertisement from $n_1$ with *dest* set to $n_2$, they do not advertise if they don't have to—i.e. if $v_i^n = v_i^{n_1}$ for all $i$, where $n \in N(n_1)$ and $N(n_1)$ is the set of neighbors of $n_1$ except $n_2$. If any node in $N(n_1)$ needs an update (i.e. $v_i^n < v_i^{n_1}$ for any $i$), it broadcasts *ReqToDisseminate* and goes to the Disseminate state. Otherwise, if a node $n$ in $N(n_1)$ finds that $n_1$ needs an update (i.e. $v_i^n > v_i^{n_1}$ for any $i$) , $n$ broadcasts its advertisement message if it has not heard more than $k$ (suppression threshold) advertisements with same metadata as its own since it heard the advertisement from $n_1$. Because of this advertisement suppression, which is borrowed from Trickle, Varuna scales well with varying node density. When $n_2$ receives advertisement from $n_1$ with *dest* set to $n_2$, it replies by broadcasting its Advertisement message with *dest* set to NULL, irrespective of whether it is up-to-date with $n_1$. This is because $n_1$ wants confirmation about the freshness of its metadata with respect to $n_2$ that caused $n_1$ to become MOODy. Similarly, when $n_2$ replies with its metadata broadcast with *dest* set to NULL, neighbors of $n_2$ will not broadcast advertisement if their metadata is same as $n_2$'s. If they do not match, the neighbor node behaves similarly to the behavior when it heard the advertisement from $n_1$ with *dest*=$n_2$. In this scheme, even if $n_1$ and $n_2$ have the same but outdated versions of the data item, their neighbors help them detect the inconsistency and make them transition to the Disseminate state where they can be updated.

**Retries to deal with link failures**: In the MOODy state, $n_1$ may not receive any response to its Advertisement message from $n_2$ even though $n_1$ had received a UA packet from $n_2$ that triggered $n_1$ to be MOODy. The link between $n_1$ and $n_2$ may be functional in only one direction ($n_2$ to $n_1$), $n_2$ or $n_1$ may have moved after $n_2$'s UA packet is received by $n_1$, or $n_2$ may have had a transient node failure. If $n_1$ does not receive any response, it re-broadcasts the Advertisement message after every $\tau$ interval for $T_{MOODy}$ duration. If no response is received during $T_{MOODy}$, $n_1$ returns to the Quiescent state

and discards the UA packet received from $n_2$.

**Dealing with a full neighbor table**: A node inserts a new neighbor in its neighbor table in the next available slot as long as the neighbor table is not full. When the table is full, it replaces the least recently used (LRU) neighbor with the new neighbor. The LRU node is the one from which it has not received any packet for the longest duration. Thus, in addition to the neighbor id, a neighbor table entry must contain the last time the node received a UA packet from this neighbor. An important design point of Varuna is that there is no notion of refresh time interval for clearing off the local state. Rather, Varuna uses a neighbor table which is cleared in its entirety when the node receives a code update or it is turned on.

### *B. Formal Protocol Description*

Here we describe the local rules followed by each node in Quiescent and MOODy states. In the Disseminate state, nodes follow any of the current protocols used for dissemination [1], [2], [6]. Conceptually, the gain due to Varuna arises from the fact that a node spends most of its time in the Quiescent state, where it does not transmit any advertisement packet. It transitions to the MOODy state only when there is some likelihood that neighborhood topology has changed and therefore it is worthwhile for the node to check if it needs to be updated. Varuna intelligently controls when the transitions to the more expensive MOODy state need to happen.

**Quiescent State:**

Q.1: When a node goes to the Quiescent state upon booting up or updating its metadata, it clears its neighbor table.

Q.2: When a node $n_1$ receives a UA packet from a node $n_2$, $n_1$ checks if $n_2$ exists in its neighbor table. If it exists, $n_1$ accepts the packet. Otherwise, $n_1$ goes to the MOODy state to verify if $n_2$ is up-to-date with $n_1$.

Q.3: If a node hears an Advertisement from a neighbor which is up-to-date with it, the neighbor is added to the neighbor table if it does not already exist in the table.

Q.4: If a node $n_2$ receives an Advertisement packet from a node $n_1$ with *dest* set to $n_2$, it compares the received metadata with its own. If $n_2$ needs an update (i.e. $v_i^{n_2} < v_i^{n_1}$ for any $i$), $n_2$ broadcasts *ReqToDisseminate* packet, after a time interval randomly chosen from [0, DISS_RAND], and goes to the *Disseminate* state. Otherwise, it broadcasts an Advertisement packet with *dest* set to NULL.

Q.5: If a node $n_3$ hears an Advertisement from a node $n_1$ with *dest* set to NULL or *dest* other than $n_3$, it compares the received metadata with its own. If it finds that it has the same version of the metadata as the received one, it ignores the Advertisement message. If they are different and $n_3$ needs an update (i.e. $v_i^{n_3} < v_i^{n_1}$ for any $i$), $n_3$ broadcasts *ReqToDisseminate*, after a time interval randomly chosen from [0, DISS_RAND], and goes to the Disseminate state. Otherwise, if the metadata are different but $n_1$ needs an update (i.e. $v_i^{n_1} < v_i^{n_3}$ for any $i$), $n_3$ broadcasts Advertisement packet with *dest* set to NULL, after a random time from the interval [0, ADV_RAND], conditioned on advertisement suppression. Advertisement suppression implies that if the node has heard more than $k$ advertisements with the same metadata as its own, then it will not broadcast its advertisement message.

Q.6: If a node receives a *ReqToDisseminate* packet, it goes to the Disseminate state.

**MOODy state:** Let $n_1$ be the node which transitions to the MOODy state after receiving a UA packet message from a node $n_2$ that does not exist in $n_1$'s neighbor table.

M.1: As long as $n_1$ does not receive any Advertisement from $n_2$, it broadcasts an Advertisement with *dest* set to $n_2$ after every $\tau$ interval, conditioned on advertisement suppression.

M.2: If $n_1$ does not receive any Advertisement message from $n_2$ for $T_{MOODy}$, it goes back to the Quiescent state and discards the packet received from $n_2$.

M.3: If $n_1$ receives an Advertisement message from $n_2$, it checks its metadata with that of $n_2$. If they match (i.e. $v_i^{n_1} = v_i^{n_2}$ for all $i$) and the neighbor table is not full, $n_1$ adds $n_2$ to its neighbor table, goes to the Quiescent state, and accepts the UA packet received from $n_2$. If the neighbor table is full, $n_1$ replaces the LRU node in its neighbor table with $n_2$, goes to the Quiescent state, and accepts the UA packet received from $n_2$. If the metadata don't match and if $n_1$ finds that it needs an update (i.e. $v_i^{n_1} < v_i^{n_2}$ for any $i$), $n_1$ broadcasts *ReqToDisseminate* packet, after a time interval randomly chosen from [0,DISS_RAND], and goes to the Disseminate state.

M.4 Same as Q.5.

M.5 Same as Q.6.

### *C. Eventual consistency*

Varuna ensures that if a node receives a packet from another node with a lower version of the metadata than its own, the metadata inconsistency is detected by the receiving node. So, in Varuna, communication from a node with a higher version of the metadata to another node with a lower version of the metadata can happen, without the nodes detecting the inconsistency. For example, as shown in Figure 5, let $n_1$ go to disconnection state in the time interval $[t_1, t_2]$, during which its neighbor $n_0$ downloads a new version of the code. After $n_1$ comes out of disconnection, let it receive a UA packet, $UA_1$, from $n_0$. $n_1$ finds $n_0$ in its neighbor table since $n_1$ had earlier received $UA_0$ from $n_0$. Since receiving $UA_0$, $n_1$ has not cleared its neighbor table as its metadata has not changed. Thus the communication from a node with a higher version of the code (here $n_0$) to a node with lower version (here $n_1$) goes undetected. However, Varuna ensures eventual consistency—when $n_0$ receives $UA_2$ from $n_1$ after some time, $n_0$ does not find $n_1$ in its neighbor table as it has been cleared after downloading the code. Thus $n_1$ goes to the MOODy state and detects the code inconsistency. Varuna does not rely on any specific application traffic characteristic. As long as nodes send some application traffic, the above guarantee of Varuna holds.

Note that Varuna's eventual consistency property is different from that of Trickle. In Varuna, an out-of-date node eventually learns that it needs an update when it communicates with an up-to-date node. In Trickle, this realization happens when the

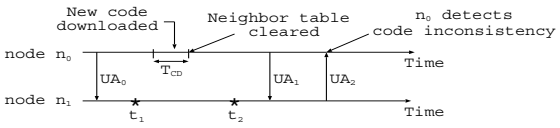



Fig. 5. Eventual consistency in Varuna

out-of-date node receives an advertisement from the up-to-date node. In other words, Trickle tries to ensure that an outdated node knows that it needs an update irrespective of whether it is communicating with other nodes or not. Varuna relaxes this consistency requirement—in the worst case[2], the outdated node knows that it needs an update when it sends a UA packet to the up-to-date node. The energy saving of Varuna is due to the relaxation of the consistency requirement. As mentioned earlier, one consequence of this relaxed consistency guarantee is that if an outdated node takes very long time to communicate with the up-to-date node, the results of the local computations may be lost in Varuna. This problem can also happen in Trickle if the advertisement interval is large. Our thesis in this paper is that in most practical sensor network contexts, Varuna's consistency guarantee is satisfactory to achieve the goal of huge energy savings, which is probably one of the most important aspects of sensor networks. Furthermore, unlike in Varuna, in Trickle a node with a lower version of the code can communicate with a node with a higher version of the code (and vice-versa), without them detecting the inconsistency, as illustrated in Figure 1.

### D. Fixed steady state cost

In Varuna, after a node downloads a new version of the code, it verifies its changed metadata with each of its neighbor *only once*. After this verification, if the neighbor table does not overflow, no further advertisements are necessary. So, in the common case, Varuna incurs *fixed* cost in the steady state, independent of the steady state interval. However, in some cases, a node may need to advertise occasionally due to the *poor neighbor* problem, which we define as follows. In some large networks, a node may occasionally receive a UA packet from "far neighbors" with very poor link reliabilities. Let us call such neighbors poor neighbors. This triggers the node to be MOODy. Since the MOODy node tries to verify the freshness of its metadata with a poor neighbor for a finite time duration ($T_{MOODy}$), the probability that it will succeed is low. As a result, the node goes back to the Quiescent state without success, and the poor neighbor is not added to the neighbor table. Every time a node receives a UA packet from the poor neighbor, though it happens rarely due to the low link reliability, it incurs the cost of transitioning to and back from the MOODy state.

The poor neighbor problem occurs very rarely because of various reasons. First, for the MOODy node to be unsuccessful in verifying the freshness of its metadata with the poor neighbor, the link reliability between them should be very poor. This means that the node will hear from the poor neighbor very rarely in the first place. Second, according to rule Q.3, whenever a node overhears an Advertisement packet with the same metadata as its own, it adds that neighbor to its neighbor table if it does not already exist in the table. This means that a node $n_1$ gets multiple opportunities to add a poor neighbor $n_2$ to its neighbor table—not only when $n_1$ hears directly from $n_2$, but also when $n_2$ broadcasts an advertisement in response to an advertisement message from a neighbor of $n_2$. This in turn means a stray message from $n_2$ does not necessarily cause $n_1$ to transition to the MOODy state and expend energy. If overhearing is not possible due to duty-cycling (to save energy), the performance of Varuna degrades. Note that overhearing is possible even with many duty-cycling MAC protocols in sensor networks.

Varuna's advantage over Trickle is even more pronounced in networks for rare event detection. Nodes generate packets rarely in response to the occurrence of certain events. Thus in Varuna, nodes need to verify their metadata very rarely. The effect of the poor neighbor problem is also significantly reduced. Trickle, however, needs to advertise periodically. The period cannot be chosen to be as large as the average event occurrence period because if events occur faster than the estimated average, the likelihood of communication between the inconsistent nodes increases, as shown in Figure 1.

### E. State maintenance cost

Trickle does not require any state maintenance, while in Varuna, each node maintains a neighbor table. The amount of memory available in low cost commercially available sensor nodes has been increasing (512 bytes RAM in Rene mote to 256KB RAM in IMote2), and this trend is likely to continue in future. Thus this tradeoff makes sense because reducing the energy consumption and increasing the network lifetime are, generally, more important than saving some memory. Neighbor table is a localized data structure and its size does not increase with the size of the network, but with the size of the neighborhood of the node. Also, as we will show from our experiments, for most practical deployments, the neighbor table consumes less than 200 bytes of memory. For "very large" and "very dense" networks, less than 600 bytes are enough. Consider the memory available in current sensor nodes: TelosB, micaz, IMote2, IRIS, BTNode, and SunSPOT have 10KB, 4KB, 256KB, 8KB, 180KB, 512KB RAM, respectively. Furthermore for many sensor networks, neighbor table is a fundamental already existing data structure. It is used by many protocols and services—MAC protocols [9], AODV based routing protocols [10], 6LoWPAN [11] standard, ZigBee, and many other applications. Thus, Varuna can leverage the existing data structure. The size of the neighbor table can be adjusted dynamically based on runtime observations of the table overflow.

[2]This is the worst case scenario because in Varuna, a node can learn that it needs an update before it sends the UA packet to the up-to-date node if it hears advertisement from an up-to-date node, say, in response to the advertisement request from some other node.

TABLE I
PARAMETERS FOR THE EXPERIMENT.

| | | | |
|---|---|---|---|
| $T_{UA}$ | U[0,60sec] | $(\tau_l, \tau_h)$ | (2sec,2min) |
| $\tau$ | 4,8,and 20sec | $k$ | 2 |
| $ADV\_RAND$ | 1,2,and 5sec | $T_{MOODy}$ | 1min |

### F. Detection latency

Traditionally, detection latency is defined as the time duration from the instant when a node becomes out-of-date to the instant when it *knows* that it is out-of-date. Clearly, higher advertisement rate and UA packet rate decrease detection latency in Trickle and Varuna, respectively. However, one of the basic ideas of Varuna is that nodes need not be up-to-date with *all* neighbors *all the time*. Rather, information should not flow from a lower version node to a higher version node. Thus, in this context, a more suitable definition of detection latency is the time interval from when an out-of-date node communicates with an up-to-date node for the first time to the instant when it realizes that it is out-of-date. As is obvious from Varuna's design, this detection latency is zero, because whenever an out-of-date node communicates, its inconsistency is detected. Even with this new definition of detection latency, for Trickle this is still a function of advertisement period.

## V. IMPLEMENTATION AND EVALUATION

We implement Varuna on TinyOS-2.1. In order to evaluate the performance of Varuna and compare it with Trickle, we perform testbed experiments using TelosB sensor nodes. For large scale evaluation, we use TOSSIM [12] simulations. We run the network in the steady state. This means in the experiments no information dissemination is taking place. This is a valid experimental method because the only job of Varuna is to let the node know *when* it needs to go to the disseminate state. Varuna does not have any impact on the actual dissemination. We use steady state energy cost as a metric to compare Varuna and Trickle. Since the energy cost is directly proportional to the number of advertisement transmissions in the steady state, we use the total number of advertisement packets transmitted by all nodes in the network as a measure of steady state energy cost. We also quantify the memory cost for state maintenance in Varuna for various node densities and network sizes. Each entry in the neighbor table takes 6 bytes (2 bytes for neighbor-id and 4 bytes for the time when this neighbor was last heard from).

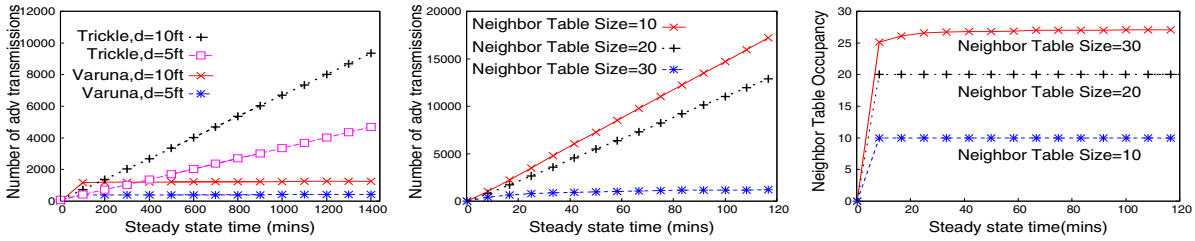
### A. Testbed Results

We compare Trickle and Varuna using a 30-node testbed of TelosB nodes arranged in a 5X6 grid. The output transmission power of each node is set to the minimum possible value. Each node broadcasts UA packets after every $T_{UA}$ interval uniformly distributed between 0 and 60 seconds. Table I shows the values of the parameters used in the experiments.

*1) Comparison of steady state energy cost:* Figure 6-a compares the number of advertisement transmissions by Trickle and Varuna as a function of steady state time. In this experiment, each node allocates a neighbor table of size 30, i.e. 180 bytes. As expected, the steady state energy cost increases linearly with time in Trickle. However, in Varuna, it does not increase after some time because once a node verifies the freshness of its metadata with each of its neighbors, it does not need to transition to the MOODy state and advertise anymore. In just 1 day, the steady state energy cost in Trickle is about 8 and 11 times more than that of Varuna for node spacing, $d = 10$ ft and $d = 5$ ft, respectively. A simple extrapolation of the data in Figure 6-a shows that in one month, Trickle consumes about 223 and 336 times more energy than Varuna, for $d = 10$ ft and 5 ft, respectively. Note that in our experiments nodes are not duty-cycled. With duty cycling, identical energy savings due to Varuna will take longer to be realized, increasing in inverse proportion to the duty cycle. If we had data dissemination in our experiments, this would not have changed the results. In both Varuna and Trickle, a node would have realized it is out-of-date and transitioned to the Disseminate state, at which point the same protocol would have been used. Varuna incurs the same cost whether a node determines it is out-of-date or up-to-date. Varuna's contribution is to provide a low cost way for the node to determine either of these two conclusions.

When the distance $d$ between successive nodes in the grid is increased, the energy consumption increases both in Trickle and Varuna. The increase in $d$ (equivalently, decrease in node density) causes the link quality to be poor between the neighboring nodes. As a result, in Trickle, a node may not hear $k$ or more identical advertisement broadcasts in its neighborhood, even though that many may have been broadcast in reality. Consequently, the node will not suppress its own advertisement. This leads to more redundant advertisements for $d$=10ft than $d$=5ft. In Varuna, poor link qualities cause many retransmissions of advertisement packets in the MOODy state and thus the energy cost is higher for the sparser network.

*2) Effect of neighbor table size:* In the above experiment, Varuna incurs fixed cost in the steady state because the neighbor table does not overflow as it is sufficiently large. If the neighbor table is small, a node $n_1$ may need to evict an LRU node $n_2$ from the neighbor table to accommodate a "new" neighbor $n_3$. Later, when $n_1$ receives a UA packet from $n_2$ , $n_1$ goes to the MOODy state (even though $n_2$ is up-to-date with $n_1$) as $n_2$ does not exist in its neighbor table. So, the steady state energy cost cannot be a fixed value if the neighbor table is small. Figure 6-b shows the steady state energy cost in Varuna for different neighbor table sizes for $d$ =10 ft. When the size of the neighbor table is 30, the steady state energy cost is fixed. But when the neighbor table is 10 or 20, it increases linearly with time. Although not shown in the figure, obviously the slope of this linear relationship increases with the increase in the UA packet reception rate because UA packet reception causes the node to be MOODy if the source of the UA packet is not in the neighbor table. For small or moderate size networks, the neighbor table can be made sufficiently large to ensure fixed energy cost in the steady state. In the next section, our simulation results show that, even for very large and dense networks, the memory requirement for state maintenance is




(a) (b) (c)

Fig. 6. Testbed results: Steady state energy cost as a function of time for (a) neighbor table size=30, and different grid spacings $d$, and (b) $d$=10ft and different neighbor table sizes; (c) Neighbor table occupancy vs time for $d$ =10ft.

reasonable. It is fundamentally because the neighbor table size does not grow with the network size. Rather, it grows with increasing number of nodes in a neighborhood, or equivalently, the network density.

Figure 6-c shows how the actual neighbor table occupancy (the number of entries in the neighbor table) varies with time. The neighbor table occupancy shown in this graph is an average over all nodes in the network. When 30 slots are allocated for the neighbor table, on average each node uses about 26 slots, but with 10 or 20 slots, the occupancy reaches the capacity and there is overflow. Thus, 10 or 20 slots are not sufficient, which causes the linear increase in steady state energy consumption shown in Figure 6-b.

### B. Simulation Results

In small and moderate size networks, the size of the neighbor table can be made sufficiently large. In order to find the appropriate size of the neighbor table to ensure fixed steady energy cost in large networks, we perform TOSSIM [12] simulations on a 20x20 grid network. Distance $d$ between the grid points is taken as 5ft, 10ft, and 20ft. The values of various parameters used for the simulation experiments are same as those used for testbed experiments (Table I). In our simulations, all UA packets are broadcast. In practice, not all UA traffic is of broadcast nature. If the amount of broadcast traffic is less, then as explained in Section IV-D, the number of MOODy transitions of the node, steady state energy cost, and the necessary neighbor table size are all reduced.

As Figure 7-a shows, the steady state energy cost increases linearly with time in Trickle, whereas it is fixed in Varuna for $d$ =10ft and 20ft, for a neighbor table size of 50 slots. Trickle consumes about 5 and 147 times more energy than Varuna in one day and one month, respectively, for $d$ =20ft. For a dense network with $d$=5ft, Figure 7-b shows that a neighbor table of 100 slots is required to achieve fixed steady state energy cost in Varuna. For the dense network, Trickle's performance is better than for the sparse network because of its good advertisement suppression mechanism. But as the steady state time increases, Varuna outperforms Trickle. Figure 7-c shows the steady state energy cost for various values of neighbor table size and $d$ =10 ft. Identical to the conclusion from the testbed experiments, if the neighbor table is kept sufficiently large, the steady state energy cost becomes independent of time in Varuna.

Figure 7-d, e, and f show the actual neighbor table size occupancy for $d$ =5ft, 10ft, and 20ft. For sparse network with $d$ =20ft, neighbor table with less than 10 slots is sufficient. For a denser network with $d$ =10ft, less than 30 slots are sufficient. For a very dense network with $d$ =5ft (and the large 400 node network), less than 100 slots (i.e. less than 600 Bytes) are sufficient.

In the TOSSIM simulation model, each node has a transmission radius of 50ft, and the bit error rates are modeled using the empirical results from TinyOS experiments. To evaluate the size of the neighbor table necessary to ensure fixed steady state energy cost for different node densities, we introduce the notion of *density factor*. It is the ratio of actual network node density (nodes/ft$^2$) to *connectivity density* (nodes/ft$^2$). Connectivity density is the node density in a *minimally connected network* where every pair of neighboring nodes are at the farthest possible distance that allows direct one-hop communication between them. For example, in the grid network for our TOSSIM simulation, connectivity density is 4/2500 nodes/ft$^2$ since a square of 50ft side requires four nodes at four corners of the square to be minimally connected. Thus, density factor is also a measure of node redundancy in the network. For example, if density factor is $k$, then the network has $k$ times more nodes than that minimally required for connectivity. For our experiments with $d$ =20ft, 10ft, and 5ft, density factors are 6.25, 25, and 100, respectively. Note that because of failures and the fact that sensing range is generally smaller than the transmission range, a minimally connected network is generally not suitable for practical deployments. Nevertheless, a redundancy of factor 100 (for $d$ =5ft) will likely be more than sufficient for most deployments. For such a deployment, a neighbor table of 100 slots is sufficient. Figure 7-g shows how the size of the neighbor table required for fixed steady state energy cost in Varuna increases with the density factor. This relationship looks linear and we plan to investigate this further in our future work.

## VI. Related Work

Reliable data dissemination using unreliable wireless channels has been the focus of several research efforts. This is more challenging in sensor networks where transient failures occur more frequently and unpredictably than in other types of wireless networks. Also energy consumption is a major

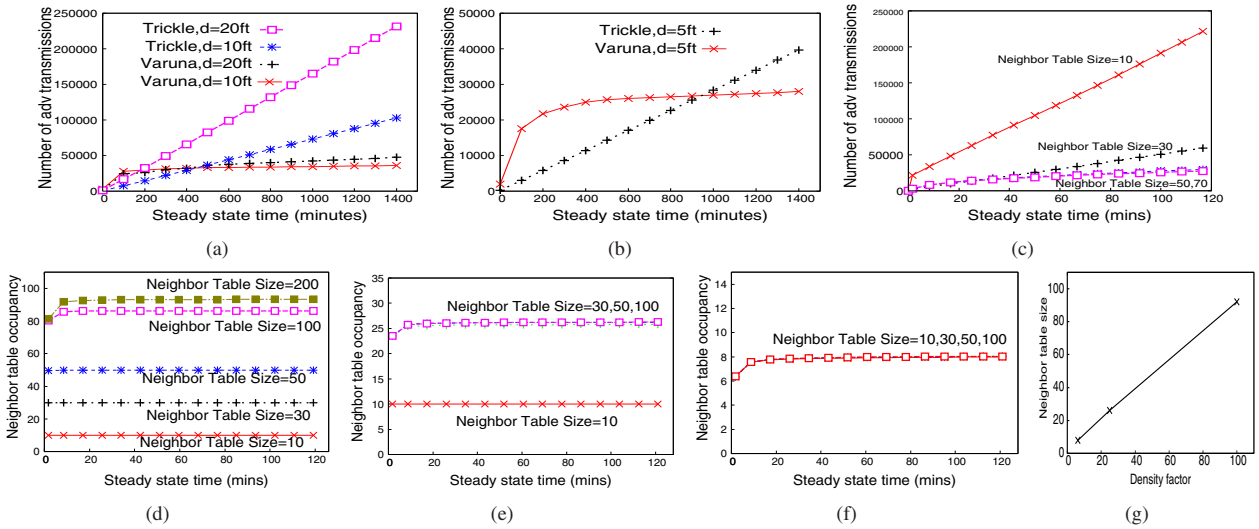



Fig. 7. Simulation results: Steady state energy cost as a function of time for (a) neighbor table size=50 and (b) neighbor table size=100; (c) Steady state energy cost for different neighbor table sizes for $d$=10ft; Neighbor table occupancy vs time for (d) $d$=5ft, (e) $d$=10ft, and (f) $d$=20ft; (g) Neighbor table size as a function of density factor.

concern. Various protocols have been proposed for information dissemination in sensor networks. Drip [13] disseminates network parameters to the network nodes; Marionette [4] distributes network queries for debugging; [1], [2], [6] disseminate code binaries; Tenet [14] disseminates tasks. All these protocols use Trickle [5] or some modified form of the Trickle algorithm. Trickle's design achieves fast and energy efficient distribution of the data items in the dissemination phase under varying node densities. Dip [3] reduces the size of the metadata that has to be transmitted in an advertisement message if the information to be disseminated consists of several sub data items. However the steady state energy expenditure of *all* of these protocols increases linearly with the steady state time, the most dominant phase in the lifetime of a network. To the best of our knowledge, Varuna is the first protocol to address the issue of steady state resource expenditure.

## VII. Conclusions

To maintain data item consistency, existing systems cause nodes in the network to advertise their metadata periodically in the steady state when no dissemination is actually being done. As a result, the steady state energy cost increases linearly with the steady state time—the most dominant part of a node's lifetime. We presented the design and implementation of Varuna, whose steady state cost is independent of the steady state interval for most practical cases. Varuna achieves energy savings of several orders of magnitude compared to the existing standard algorithm called Trickle, as demonstrated through our testbed and simulation results. The tradeoff in Varuna are the memory requirement for storing neighbor table and guarantee of a relaxed invariant that information cannot flow from a higher version node to a lower version node. As is evident from our experiments, even for a very dense network with 100x redundancy, the memory requirement is reasonable for the currently available commercial sensor nodes. Also, in most sensor network deployments, the flow of information from up-to-date nodes to out-of-date nodes for some time (before it is eventually detected by Varuna) is acceptable. The possibly erroneous information does not affect the up-to-date nodes, and cannot force wrong decisions at the base station, which can be assumed to be always up-to-date.

## References


[1] J. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," *SenSys*, pp. 81–94, 2004.

[2] R. Panta, I. Khalil, and S. Bagchi, "Stream: Low Overhead Wireless Reprogramming for Sensor Networks," *Infocom*, pp. 928–936, 2007.

[3] K. Lin and P. Levis, "Data discovery and dissemination with dip," *IPSN*, 2008.

[4] 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," *IPSN*, 2006.

[5] P. Levis, N. Patel, D. Culler, and S. Shenker, "Trickle:A Self Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks," *NSDI*, pp. 15–28, 2004.

[6] M. Krasniewski, R. Panta, S. Bagchi, C. Yang, and W. Chappell, "Energy-efficient on-demand reprogramming of large-scale sensor networks," *ACM Transactions on Sensor Networks*, 2008.

[7] R. Panta, S. Bagchi, and S. Midkiff, "Zephyr: Efficient incremental reprogramming of sensor nodes using function call indirections and difference computation," *USENIX ATC*, 2009.

[8] R. Panta and S. Bagchi, "Hermes: Fast and Energy Efficient Incremental Code Updates for Wireless Sensor Networks," *IEEE Infocom*, 2009.

[9] "Wireless systems for industrial automation, process control, and related applications," *ISA 100.11a, Draft standard*, 2009.

[10] C. Perkins, E. Royer, and S. Das, "Ad hoc on-demand distance vector (AODV)," in *IEEE WMCSA*, 1999.

[11] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks," *RFC 4944*.

[12] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: accurate and scalable simulation of entire tinyOS applications," *SenSys*, 2003.

[13] G. Tolle and D. Culler, "Design of an application-cooperative management system for wireless sensor networks," *EWSN*, pp. 121–132, 2005.

[14] O. Gnawali, K. Jang, J. Paek, M. Vieira, R. Govindan, B. Greenstein, A. Joki, D. Estrin, and E. Kohler, "The tenet architecture for tiered sensor networks," *Sensys*, pp. 153–166, 2006.