


# Efficient Code Dissemination in Wireless Sensor Networks[☆]

Sain Saginbekov[a,1,*], Arshad Jhumka[a]

[a]*Department of Computer Science*
*University of Warwick*
*Coventry CV4 7AL, UK*

[☆]This is an extended version of a paper [**?** ] that was published in the Proceedings of the International Conference on Ubiquitous Computing and Communications (IUCC) 2012.

[*]Corresponding author
*Email addresses:* sain@dcs.warwick.ac.uk (Sain Saginbekov), arshad@dcs.warwick.ac.uk (Arshad Jhumka)
[1]Tel: +44 24 7657 3780, Fax: +44 24 7657 3024

**Abstract**

Given the dynamic nature of the mission of a wireless sensor network (WSN), and of the environment in which it is usually deployed, network reprogramming is an important activity that enables the WSN to adapt to the mission and/or environment. One important component of a reprogramming protocol is code dissemination and maintenance, during which new code is propagated to relevant WSN nodes. Several dissemination protocols have been proposed, each with a specific objective in mind. Protocols such as Trickle minimise dissemination latency by periodically broadcasting advertisement messages at the expense of energy consumption, while protocols, e.g., Varuna, reduce energy usage by broadcasting advertisement only when needed. In certain type of WSNs, such as *event-based WSNs*, Varuna has high code dissemination latency, while the energy consumption of Trickle does not improve in such WSNs. Further, the efficiency of Varuna drops drastically in the presence of asymmetric links. In this paper, we propose a new code dissemination protocol, called *Triva*, for event-based WSNs, by leveraging the properties of Trickle and Varuna. Our simulation and experimental results show that, for event-based WSNs, Triva outperforms Trickle and Varuna in terms of energy consumption and code dissemination latency respectively. Triva also outperforms Trickle and Varuna when there are unidirectional links in the network. We also show that Triva provides excellent results during bursty traffic in event-based WSNs. Triva is the first information dissemination protocol for event-based WSNs and that tolerates asymmetric links.

**Keywords**: Information dissemination; Wireless Sensor Networks; Event-Based; Asymmetric Links;

## 1. Introduction

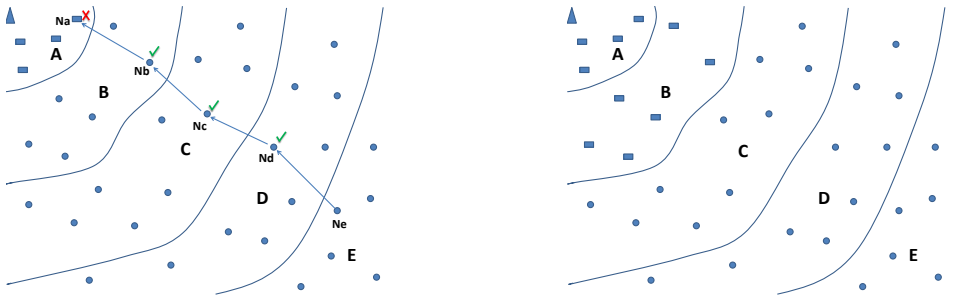
A *wireless sensor network* (WSN) consists of a set of resource-constrained devices, called *nodes* or *motes*, that communicate wirelessly with each other. These networks have enabled novel applications such as monitoring and tracking. These devices are battery-operated, and have limited computational power (e.g., small memory). Given that these networks are generally deployed unattended for long periods, e.g., buried in bird burrows [11] or collared on roving herds of zebras for years possibly, they have to adapt either to their environment or to any change in the WSN mission. To achieve this adaptivity, *network reprogramming* is an important activity during which new code or parameters[2] are uploaded onto the nodes over-the-air. One important component of network reprogramming is *code dissemination*, during which the new code is wirelessly propagated to all the target nodes in the network. Several code dissemination protocols exist (e.g., Trickle [9], MNP [7], Varuna [14]), each focusing on specific dissemination aspects.

Any code dissemination protocol needs to satisfy some important properties: (i) *energy efficiency*: wireless communication has a high energy cost, and primarily defines the system lifetime. Where laptops or mobile phones can be recharged, sensor networks die due to energy exhaustion. Thus, an effective code dissemination protocol for reprogramming must send as few packets as possible, while ensuring that all target nodes receive the code update. (ii) *dissemination latency*: while the new code is being propagated, the network may be in an erroneous, useless state, since interacting nodes may have different code versions running, possibly running different missions. In this case, transition time is wasted time, leading to a waste in energy. Therefore, an effective code dissemination protocol must also propagate new code quickly. The code dissemination protocol typically consists of two components, namely (i) a code maintenance part and (ii) a code download part. The code maintenance enables nodes to determine if they need to download new code or not, whereas the code download part enables relevant nodes to download the code.

To achieve the first goal, viz. energy efficiency, protocols control energy expenditure in various ways, especially during code maintenance. For example, energy efficiency is achieved by either reducing the number of messages being sent [9] (using some form of "polite gossip") or using some well-defined duty-cycling [7] or by integrating the version inconsistency detection during payload communication, thereby avoiding special packet transmission [14]. On the other hand, one way of reducing the code dissemination latency is to periodically perform a "polite gossip", i.e., periodically perform a code maintenance (to determine whether any node needs code updating). However, there seems to be a tradeoff between dissemination latency and energy efficiency: specifically, to reduce latency, version inconsistency needs to be detected fast, requiring periodic transmission of code information. On the other hand, to reduce energy consumption, inconsistencies need only be detected "when needed", i.e., when nodes communicate. Trickle [9] is an algorithm that achieves low dissemination latency through periodic advertisement of the new code. However, there is a steady expenditure of energy, even when the network is in a steady state (i.e., when no node needs updating). This is due to the proactive step to detect inconsistencies. On the other hand, in Varuna [14], inconsistencies are detected during application message communication. In this case, no additional energy is spent in the steady state.

WSN applications can be classified according to their data delivery model as either continuous (periodic), event-driven, observer-initiated or hybrid [18]. For event-based WSNs, messages are sent only when a given event is detected, over a time period [1, 22]. In applications like forest fire detection and flood detection, when a node detects a fire or a flood, it must

[2] We will call a new program, new parameters or new values as new code.




(a) Multihop network divided into regions according to hop distance from the sink.

(b) Network after communications of nodes in region $B$ with the nodes in region $A$.

Figure 1: Varuna in action.

immediately send an alarm to the sink for a certain time. However, if events of interest are far between, i.e., the occurrences of the events are rare, then any code dissemination protocol that are based on regular data communication, e.g., Varuna, can suffer from high dissemination latency. If data packets are sent rarely, like in such event-based sensor networks, nodes further away from the sink will update their code very late, depending on how frequently data packet is sent. Moreover, in Varuna, most of the application data will be discarded by intermediate nodes, during convergecast, due to code incompatibilities, thereby reducing the yield of the network. Figure 1(a) illustrates these drawbacks of Varuna. For the sake of simplicity, consider a multihop network in which nodes located in the same region labeled with $X$ have the same number of hops away from the *sink*. The nodes in neighbouring regions have one hop distance. Assume that triangular node located in the corner is sink, the rectangular nodes in region $A$ are updated nodes and circular nodes in other regions are not updated. According to Varuna protocol, nodes in regions $B, C, D$ and $E$ communicate and accept each others data because they have the same version codes. However, since nodes in region $A$ have bigger version numbers than the nodes in other regions, the nodes in region $A$ should not communicate with other nodes. They will not forward messages received from nodes in $B$ and will discard them. They detect inconsistency and let the nodes in region $B$ update their code. For example, assume that a node $N_e$ in region $E$ has detected a fire and wants to send an alarm message to the $sink$ immediately. It sends to a node $N_d$ in $D$, which accepts and forwards it further to a node $N_c$ in $C$. $N_c$ forwards to node $N_b$. When data is forwarded from $N_b$ to a node $N_a$ in region $A$, $N_a$ will cancel it because node $N_b$ has lower version than $N_a$. In other words, the data originated at $N_e$ will be waste after flowing through the entire network. Similarly, many packets originated at regions $B$, $C$, $D$ and $E$ will be discarded since the time when new code is injected. Therefore, when events are rare, leading to bursts of data packets, Varuna not only waste resources such as energy and bandwidth, but also increases dissemination latency and delivery latency. Figure 1(b) shows the network after communication of nodes in region $B$ with the nodes in region $A$.

Further, in WSNs, link quality can fluctuate[2], causing asymmetric links to exist in the network, and this can be due to several reasons. For example, the network can transmit low-power signals, thus creating links that are often asymmetric. The link quality depends strongly on hardware inaccuracy and environmental factors [2]. Hardware inaccuracy may create asymmetric links between nodes [2]. In [10], the authors observed that transceiver frequency mismatch can also be a reason for asymmetric links. Further, the duration of these asymmetric links may be very small (i.e., transient) or very long (i.e., permanent) [12]. Protocols that assume bidirectional links may not work or may not be efficient in networks with asymmetric links. Specifically, Varuna will not work efficiently if asymmetric links exist in the network since the protocol is based on bidirectional communication. Asymmetric links create several major problems in wireless sensor networks [19]. Overall,

Trickle spends a lot of energy, even in steady state, for code maintenance, while Varuna, which addresses the shortcomings of Trickle, does not perform well in presence of asymmetric links or if the network application is event-based.

Thus, there is a need for a code dissemination protocol performs well in event-based WSNs and that tolerates asymmetric links. Such a protocols needs to have the following properties: (i) low dissemination latency as Trickle, and (ii) does not incur steady energy expenditure during steady state, as Varuna. To achieve this, we propose a new protocol, called *Triva*, which is an adaptive code dissemination protocol that leverages the properties of Trickle and Varuna. Specifically, when dissemination is needed, Triva behaves like Trickle, but when dissemination is not needed (maintenance is needed), it behaves like Varuna. Our results show that Triva outperforms both protocols in general. Further, Triva outperforms both Trickle and Varuna in presence of asymmetric links. We also show that Triva handles bursty traffic much better than Varuna.

Our paper is structured as follows: We present an overview of related work in Section 2. We present our network model in Section 3. We propose Triva, an adaptive algorithm for event-based WSNs in Section 4. We explain our simulation and experimental setup in Section 5, and discuss our results in Section 6. Finally, we conclude the paper in Section 7.

## 2. Related Work

### *2.1. Overview of Related Work*

There exist several dissemination protocols for updating codes. While some of the protocols deliver complete binary image of the code like [3], [16], [5], [7], some deliver only the difference between the new code and the old code [6, 15]. Also there exist protocols which deliver tasks[4], network parameters[20], and queries[23].

In XNP[3], the base station broadcasts the code image to the nodes which are in the coverage range of it. The nodes which are not in the range cannot receive the code image. The protocol proposed in MOAP[16] is a multi-hop dissemination protocol, which can deliver code images to nodes that are several hops away from the base station. Each node forwards the code image further after receiving the complete code image. Deluge[5] allows large data transmission by fragmenting the data into fixed-size pages. It also supports pipelined page transmission to make dissemination faster. Unlike MOAP, nodes in Deluge should not wait for the complete code image before forwarding it. The authors of MNP[7] proposes another protocol which, like Deluge, fragments the code image and uses a pipelining mechanism. However, unlike Deluge, MNP selects the sender of the code such that there is at most one sender at any time in a neighbourhood, i.e., a sender with the highest coverage will be chosen to avoid message collisions. A sender selection mechanism reduces collisions and avoids the hidden terminal problem. Also, in MNP, some of the nodes can go to the sleep mode to save energy whenever there is no data to receive or transmit.

Because of the features of wireless sensor networks, e.g., transient link failures and node mobility, not all nodes will update their code to the newest one during the dissemination phase. Some of the nodes may not receive any message at all during the dissemination period because of transient link failures or node rejoins. To ensure the delivery of the updated code to all nodes in a network, any updated node should continuously check the consistency of its neighbours, by broadcasting *advertisement* messages. Blindly broadcasting advertisement messages causes the so-called “broadcast storm problem” [13]. There are two algorithms which address this problem. The first one is Trickle [9], which addresses this problem by using a “polite gossip” policy. In Trickle, a node suppresses its advertisement message transmissions when it hears a number of messages identical to its own. The second algorithm that addresses the broadcast storm problem is Varuna [14], which supports code update

maintenance. Varuna consumes constant energy in steady phase when the network is in a steady state, i.e., when all nodes have the same code.

Since our proposed algorithm Triva leverages the properties of Trickle and Varuna, we will discuss both of these algorithms in more details in the subsequent sections.

### *2.2. Trickle*

In Trickle, every node broadcasts advertisement messages that contain a metadata that includes the version number of the code, at most once per period given between [$\tau$/2, $\tau$]. If a node hears more than $k$ identical metadata before it transmits its advertisement, it suppresses its broadcast and doubles the value of $\tau$ up to $\tau_h$, which is an upper bound for $\tau$. If it hears a different metadata, $\tau$ is set to $\tau_l$, which is a lower bound for $\tau$. By increasing the broadcast interval, $\tau$, Trickle sends less number of advertisement messages, thus saving energy. By decreasing $\tau$ ,Trickle can update nodes more quickly. So, there is a trade-off between dissemination latency and energy to be achieved when selecting $\tau$. As noted in [14], in addition to the dissemination latency, increasing the value of $\tau$ to large values causes some other problems such as message communication between two nodes, which have different code versions. In Trickle, the number of advertisement messages increases linearly as a function of time, as the dissemination is irrespective of the mission of the WSN application.

### *2.3. Varuna*

Varuna[14] is another protocol which supports code update maintenance. Varuna saves energy in the network steady phase, a phase where no dissemination is being done as all the nodes have the same code version. Unlike Trickle, where there is a linear increase of energy consumption, energy consumption in Varuna is constant in the steady phase. To achieve this constant energy consumption in the steady phase, nodes, in Varuna, send advertisement messages only when there is a change in their neighbourhood topology or metadata since their last advertisement transmissions. To learn about this change, each node stores its neighbour IDs in its *neighbourhood table*. A node stores only the IDs of neighbours which are consistent with it, i.e., neighbours that have the same code as it. For example, if a node $N_1$ sends a message to a node $N_2$, $N_2$ checks the existence of $N_1$'s ID. If it exists in the table, it is assumed to be consistent with the $N_2$, and then $N_2$ does not send an advertisement message. Otherwise, $N_2$ checks the consistency of its code with the node $N_1$ by sending an advertisement message. If the version numbers are equal, $N_2$ stores the ID of $N_1$ in its table. If they are different, the node which has bigger version number sends an advertisement message to let other nodes request and download the code with the bigger version. After receiving the new code, the node resets its table (as it has to detect new consistent nodes). When all nodes receive the newest code, every node's table will contain the IDs of all neighbouring nodes. This state stops sending advertisement messages, which makes energy drain constant in steady phase after some time. In Varuna, a node detects inconsistency only if it receives a message from a node which has smaller version number. It means that, a node will not update its code unless it communicates with a node with new version number. This makes the *update latency*, the time from the injection of new code to the time when all nodes receive the new code, dependent to a communication rate of a node. Therefore, update latency in Varuna increases linearly with data communication rate or with the event time if the application is event based.

### *2.4. Asymmetric links*

In wireless networks, especially in networks which transmit low-power signals, links are often asymmetric. The link quality, as mentioned in [2], have a strong dependency on hardware inaccuracy and environmental factors. Hardware inaccu-

racy may also create asymmetric links between nodes [2]. In their experiments[10], they observed that transceiver frequency mismatch can also be a reason for asymmetric links. And these asymmetric links may be transient or permanent[12].

Many proposed protocols for WSNs in the literature assume symmetric or bidirectional links, i.e, two nodes can receive each others messages. However, as mentioned above, the links can be asymmetric or unidirectional. Therefore, they may not function properly, may be inefficient in terms of different metrics such as energy and delay as intended, or they may not work at all.

There are also protocols which deal with asymmetric links. In[17], authors propose two WSN MAC protocols that increase reliability, connectivity and network lifetime by utilizing unidirectional links. In[21], authors propose a framework which allows routing protocols designed for bidirectional links to function in the presence of unidirectional links. Another energy efficient reliable transport protocol for WSNs which overcomes the negative impact of asymmetric links is presented in[10].

One of the most important performance criteria when designing WSNs protocols, may it be a routing protocol, MAC protocol or other layer protocol, is energy conservation. And one of the reasons of energy waste is retransmission of packets due to reasons such as collisions and asymmetric links as in Varuna. Recall that, in Varuna, a node sends *ADV* packets until it receives a response for that. The node may never receive the response, if the link beetween them is unidirectional, in other words, if there exists only downlink.

## 3. Network Model

We define a wireless sensor node as a computing device equipped with a wireless interface and associated with a unique identifier. The node has limited computational resources, such as limited memory and power. A wireless sensor network is a collection of wireless sensor nodes that communicate via the wireless interfaces, and is modelled as a directed graph $G = (V, A)$ where $V$ is a set of $N$ wireless sensor nodes (i.e., $|V| = N$) and $A$ is a set of arcs, representing the direction of message travel. Each directed link $(m, n)$ represents a pair of distinct nodes, and a directed edge exists between two nodes $(m, n)$ only if node $n$ can receive a message from node $m$. A node $n$ is said to be a 1-hop neighbour (or neighbour) of $m$ if $(m, n) \in A$. We denote by $M$ the set of $m$'s neighbours. We say that two nodes $m$ and $n$ can collide at node $p$ if $(m, p) \in A \wedge (n, p) \in A$[3]. Our model allows the network to have asymmetric links. In such a case, any signal with a low signal-to-noise ratio (SiNR) such that the signal is not correctly received at an intended receiver can be omitted from the graph. However, we also assume that the network remains connected in that every node has at least one arc incident on it and at least one outgoing link, i.e., $\forall m \in V \exists p, q \cdot (m, p) \in A \wedge (q, m) \in A$.

We assume that there are two special types of messages: (i) An *advertisement* message, which is broadcast by a node to inform its neighbourhood about the *version number* of the code it has, as is the case with existing code dissemination protocols, and (ii) a *request* message that is used by a node to request new code from one of its neighbours.

## 4. Triva: Efficient Algorithm for Code Dissemination in Event-Based WSNs

In this section, we explain our proposed protocol, *Triva*, and subsequently give a formal description of its working.

*Triva* is a code maintenance protocol, as part of a code dissemination protocol intended specifically for event-based wireless sensor networks. It works in such a way so as to enable nodes to update their code quickly, very much like Trickle.

[3] We will say two nodes $m$ and $n$ can collide if such a node $p$ exists

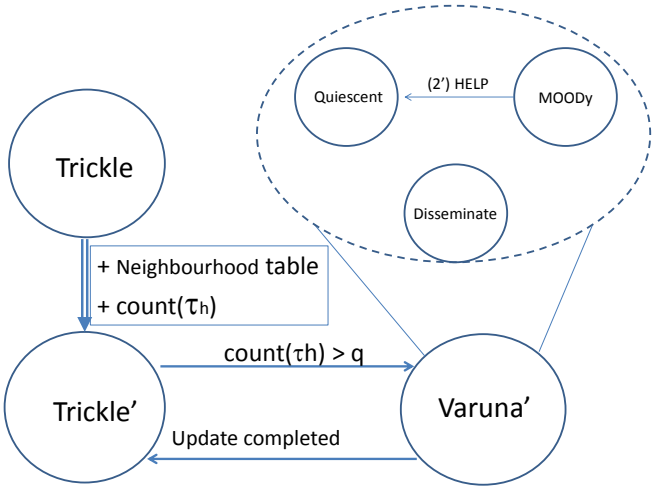



Figure 2: The state machine for Triva, which is a combination of Trickle' and Varuna, where Trickle' is obtained by adding a *Neighbourhood* table and variable $count(\tau_h)$ to Trickle.

However, unlike Trickle, it does not consume much energy in steady state. Specifically, it consumes little energy, like Varuna, when there is no new code in the network. Further, when there unidirectional links exist in the network, Varuna's energy efficiency *drops drastically* as redundant message transmissions are necessary. *Triva* tries to address the unidirectional link problem by making use of other neighbours, circumventing the problem when a relevant neighbour cannot be reached.

Informally, our protocol leverages the working of both Trickle and Varuna to achieve efficient dissemination and works in the following way: When a node $N_1$ updates its code, it tries to quickly disseminate the code to its neighbours. $N_1$ broadcasts advertisement messages at a random time in given period, as in Trickle. If, during these transmissions, a neighbouring node $N_2$ requests the new code, $N_1$ sends the new code to $N_2$. However, unlike in Trickle, if $N_1$ receives an advertisement message with the same version number from $N_2$, it saves $N_2$'s *ID* in its neighbourhood table. After broadcasting advertisement messages for some time, the node stops broadcasting and acts like in Varuna in the steady phase to save energy. In *Triva*, there is no concern with selecting an upper bound value $\tau_h$ (see Section 2.2) as, in *Triva*, a node sends advertisement message, as in Trickle, only for a short period after the node has updated its code.

Specifically, we extend Trickle with some Varuna-like variables, such as neighbourhood table to reduce the broadcasting of advertisement messages. Then, in steady state, Triva behaves as Varuna, while in the dissemination phase, it behaves like Trickle. Further, in the MOODy state of Varuna', we extend the working of Varuna to handle asymmetric links, through HELP messages (see Figure 2).

### *4.1. Formal protocol description*

Figure 2 illustrates the state machine of our protocol, which we now detail.

- **Trickle' state:**

  When a node $N_1$ enters this state, it sets *counter*, which is a Trickle variable, to 0, and sets $\tau$ to $\tau_l$. In this state, $N_1$ sends an advertisement message periodically at a random time between [$\tau$/2, $\tau$], if it has not heard $k$ advertisement messages about the same version number, i.e., if *counter*$< k$, otherwise, it doubles $\tau$ up to $\tau_h$.

  – If node $N_1$'s $\tau$ becomes $\tau_h$, $count(\tau_h)$ is incremented by one.

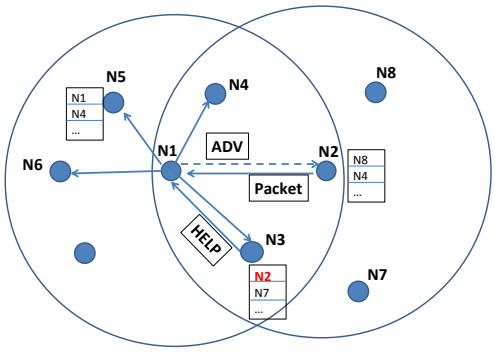



Figure 3: $N_3$ is addressing the asymmetric link between $N_1$ and $N_2$. There exists only one link between $N_1$ and $N_2$: the link from $N_2$ to $N_1$

- If a node $N_1$ receives an advertisement message from a node $N_2$, $N_1$ compares the received version number with its own. If the version number of $N_2$ is bigger, then $N_1$ requests the code after a randomly chosen time between $[0, R]$ seconds, then downloads the code, and updates from $N_2$. If the version number of $N_2$ is smaller, then $N_1$ broadcasts advertisement messages. If the version numbers are equal and if $N_2$ does not exist in its neighbourhood table, $N_1$ adds $N_2$ in its table, then doubles $\tau$ up to $\tau'_h$ and increments *counter* by one.
- If a node $N_1$ receives a request message from $N_2$, $N_1$ sends the new code to $N_2$.
- If a node $N_1$ updates its code from $N_2$, it first clears its neighbourhood table and then adds $N_2$ in its table. It sets *counter* to 0, sets $q$ to 0 and sets $\tau$ to $\tau_l$.
- If, during $\tau$ time, $N_1$ does not send any advertisement message, it doubles $\tau$ up to $\tau_h$. If $\tau$ is already equal to $\tau_h$, it increments $countQ$ by one and sets *counter* to 0.
- If $count(\tau_h) > q$, $N_1$ sets *counter* to 0, $count(\tau_h)$ to 0 and goes to *Varuna*' state.

- **Varuna':**
  - If a node $N_1$ updates its code from a node $N_2$, it clears its neighbourhood table and adds $N_2$ in its table, and then it goes to *Trickle*' state.
  - If a node $N_3$ receives an advertisement message destined to $N_2$ from a node $N_1$, and if $N_2$ is in the neighborhood table of $N_3$ and if $N_1$'s version number is equal to $N_3$'s version number, then $N_3$ sends a *HELP* packet (see Figure 2), which includes the ID of $N_2$, to $N_1$ with a propability $P$ after random time between [0, 1] second.
  - If a node $N_1$ receives a *HELP* packet that is destined to itself, then $N_1$ extracts the node ID $N_2$ from the packet and adds it into its neighborhood table if $N_2$ does not exist in the table (see Figure 3).
  - Otherwise, Varuna' behaves as Varuna.

- **Download state:**
  - After sending a new code, a node returns to the previous state.
  - After downloading a new code, a node goes to the *Trickle*' state.

*4.2. Fast dissemination*

In *Triva*, every node tries to quickly disseminate its code, whenever it receives a new code. It does so by broadcasting advertisement packet periodically between [$\tau$/2,$\tau$]. Since a node broadcasts only a limited number of advertisement packets, $\tau$ can be a small number. Therefore, the amount of energy spent sending advertisement messages is bounded, unlike in Trickle.

*4.3. Constant energy consumption*

In *Triva*, a node eventually fills its neighborhood table with the IDs of all of its neighbours, which means that all its neighbouring nodes have received the new code. Therefore, like in Varuna, in this steady phase, there is no advertisement message transmission, thereby limiting the energy expenditure due to advertisement broadcasts.

*4.4. Addressing asymmetric/unidirectional links*

In *Triva*, a node tries to help two of its neighbouring nodes that may have an asymmetric link between them. It does so by informing them about their code consistency if the node's code is consistent with those two nodes. Figure 3 depicts the Triva process of addressing the unidirectional links between $N_1$ and $N_2$: Node $N_1$, after receiving any packet from $N_2$, checks if $N_2$ exists in its neighbourhood table. If not, then $N_1$ sends an advertisement message to $N_2$. However, $N_2$ will not respond as it cannot hear packets from $N_1$ (due to asymmetric link). If $N_3$ and $N_4$ can overhear messages between the nodes, then they can help $N_1$ receive information about $N_2$'s version number, as long as their codes are consistent with both $N_1$'s and $N_2$'s code. They do it by sending a ***HELP*** packet, which contains the ID of $N_2$, to $N_1$. However, since there may be several nodes that can hear both $N_1$ and $N_2$, they send the ***HELP*** packet with a probability $P$.

In addressing the asymmetric link problem, we assume that the two nodes which have asymmetric links between them share a common neighbour. If there is no such a node, then Triva will not be executed, thereby making no message overhead.

Trickle will not be affected by the presence of asymmetric links as, in Trickle, nodes independently send advertisement packets periodically. In the case of Varuna, using the above example, $N_1$ will send advertisement packets $i$ times periodically until it gets a response from $N_2$. $N_1$ does this every time it receives any data packet from $N_2$. Therefore, in Varuna, a node transmits $i$ *ADV* packets periodically, whenever it gets a data packet and there is no uplink. Thus, if the nodes in the network send packets periodically every $T$ time, then each node transmits $O(A * i * T)$ advertisements packets even when there is no new version in the network. Here, $A$ is the number of unidirectional links (downlinks) between a node and its neighbors.

## 5. Simulation Setup

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $q$ | 15 | $\tau'_h$ | 20 sec | $T_{MOODY}$ | 60 sec | *DIS_RAND* | 2 sec |
| $\tau_l$ | 1 sec | $k$ | 2 | $\tau_v$ | 8 sec | *ADV_RAND* | 2 sec |
| $\tau_h$ | 60 sec | $b$ | 1, 30, 60 | $R$ | 1 sec | $P$ | 0.5 |

Table 1: Parameters for the simulation

We perform TOSSIM[8] simulations on a 20x20 grid network to evaluate *Triva*. We used a network topology generator tool given on tinyos.net to construct the network. We set the distance between neighboring nodes to 10 feet. By appropriately choosing the power decay value for the reference distance, we constructed the network such that a node has a communication radius of around 30 feet. Each node is given a noise model from the "casino-lab" noise trace file, which is real noise trace

taken in the Casino Lab of Colorado School of Mines. The file itself can be found in Tossim/noise folder. The parameter values used in our simulation are given in Table 1, while all other parameter values used are the same as in Trickle and Varuna.

The value of $\tau_h'$ can be set to the value of $\tau_h$. However, since *Triva* runs Trickle for a short period, we keep the value of $\tau_h'$ small. The value of $q$ can be set according to how fast a node's neighborhood table can be filled with its neighboring nodes' *IDs*. Simulations show that almost all ($\geq 90\%$) neighboring nodes' *IDs* can be saved when $q$ is between 15 to 20. The parameter $\tau_v$ is $\tau$ used in Varuna. The probability $P$ of sending a *HELP* packet can be set according to the density of the network. If the network is dense, the $P$ can be set to a small value. If the network is sparse, then $P$ can be set to larger values. In our simulations, we set $P$ to 0.5.

In our simulations, we ran three protocols: (i) Triva, (ii) Trickle and (iii) Varuna, to assess (i) the number of advertisement packets transmitted, (ii) the total number of discarded data packets (relevant for Varuna) from the time when the new version is injected and (iii) the *completion time* or dissemination latency, the time from the point when a new version is injected to the point when the last node receives the update. We simulated three scenarios:

1. *Periodic traffic* : Each node periodically sends data packet, with the period randomly selected between $[0 \ldots 1]$ minute.
2. *Event-based traffic* : Each node sends only one packet at a randomly selected time between $[\lambda, \lambda + 1]$ minute, where $\lambda$ is the time the event occurred. Events occur every $\lambda$ minutes.
3. *Event-based bursty traffic* [24] : Each node sends $b$ packets, one packet per second, after a randomly selected time between $[\lambda, \lambda + 1]$ minute, where $\lambda$ is the time the event occurred. Events occur every $\lambda$ minutes.

To evaluate the performance of the three protocols on networks with different link symmetry, we simulated them on a network (i) with symmetric links and (ii) with asymmetric links.

In all our simulations, all nodes boot randomly in the first minute and a packet with a new version is injected into the top-left node (the sink) after 2 minutes. We also assume that the network starts in a "Varuna" state.

## 6. Results

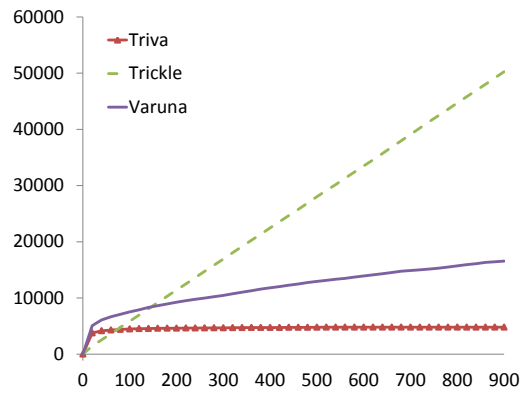
We assume that Triva is in the Varuna' state (see Figure 2) at the beginning of execution of the simulations. We show the simulation results of Triva, Varuna and Trickle.

### *6.1. Metrics*

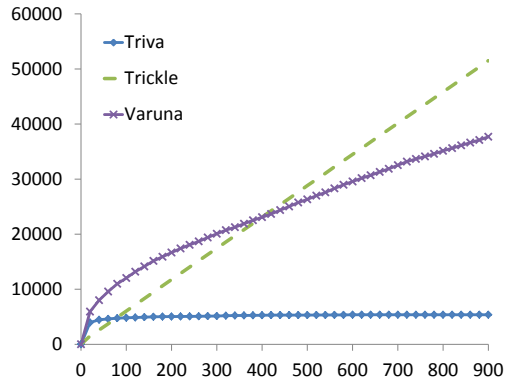
In our simulations, we used the following metrics: (i) the number of advertisement packets transmitted, as the number of transmissions is directly related to energy consumption, (ii) the number of discarded application, packets as this metric captures the amount of resources wasted as bandwidth and energy, and (iii) completion time (dissemination latency), which is another important criteria in network reprogramming, to enable the network to work efficiently.

### *6.2. Number of Advertisement Packets*

Figure 4 shows the number of advertisement packets transmitted for completely disseminating the code update in the network. Figure 4(a) shows the number of advertisement packets where a node periodically sends one data packet, with the period randomly chosen between $[0 \ldots 1]$ minute in the network with symmetric links. As can be observed, the number of advertisement packets Trickle sends increases linearly with time, even when there is no new code in the network, while the amount of transmitted advertisement messages in Varuna decreases eventually. As it can be observed from Figure 4(a), in




(a) Scenario 1 - **Periodic Traffic, Symmetric Links**: Data packets generated randomly every $0 \ldots 1$ minute for periodic traffic




(b) Scenario 2 - **Periodic Traffic, Asymmetric Links**: Data packets generated randomly every $0 \ldots 1$ minute for periodic traffic.

Figure 4: The number of advertisement packets during dissemination.

Triva, the transmission of advertisement and *HELP* packets stop after some time and never transmitted again, as long as there is no new code in the network. In the network with symmetric links, Triva sends advertisement and *HELP* packets 10 times less than Trickle and 3 times less than Varuna.

Figure 4(b) shows the number of advertisement packets transmitted in the network with high asymmetric links. Here, we can also see that Triva transmits advertisement packets 10 times less than Trickle and 7 times less than Varuna.

In Figure 6, Triva, and Varuna are compared in terms of the number of advertisement packets sent, when each node in the network sends 30 packets back-to-back at 1 packet/second, with $\lambda = 5$ minutes. In this figure, Triva again stops transmitting advertisement packets after some time.

### *6.3. Number of Discarded Application Packets*

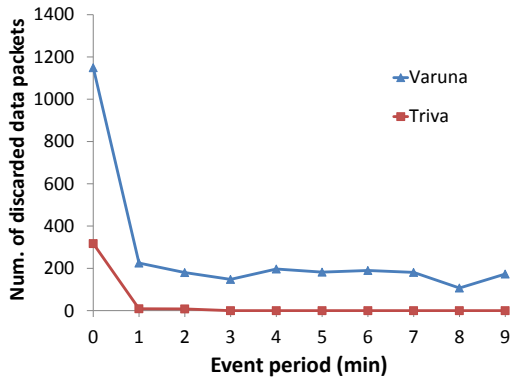
Figure 5 shows the number of discarded application data packets after the time when a packet with a new version is injected, i.e., after 2 minutes. Figure 5(a) shows the values obtained from the network in which a node periodically sends only one, i.e, $b$=1, data packet at randomly selected time between $[\lambda, \lambda + 1]$ minutes. In both *Triva* and Varuna, values are relatively high when $\lambda$=0, as the randomly selected time by a node could be very small such as 150 ms, which forces nodes to send more data packets in a small amount of time. This is shown in Figure 5(a).

In Figure 5(b), a node sends $b$=60 packets back-to-back every 1 second. The number of data packets discarded by Varuna remains constant, with increasing idle time. On the other hand, the number of dropped packets in *Triva* is very low, due to the fact that the nodes quickly obtain the updated code.

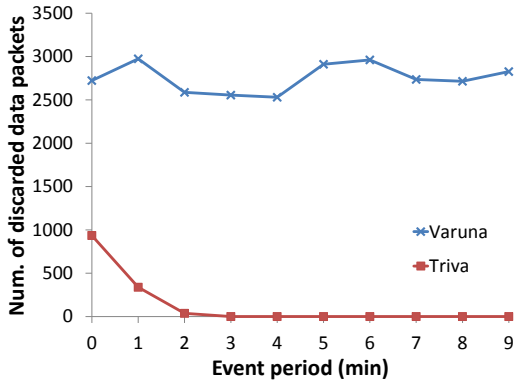
Overall, when the number of discarded data packets is low, it captures the fact that the all nodes have the same code version, i.e., code dissemination has completed quickly.

### *6.4. Code Dissemination Latency*

Figure 7 shows the time taken from the point when a packet with new version is injected to the point when the last node in a network receives that packet, i.e., the code dissemination latency. We investigated the latency under two scenarios: (i) periodic traffic, and bursty traffic. *Triva* and Trickle have very low dissemination latency (for both scenarios), whereas Varuna has increasing latency with increasing idle time.




(a) Scenario 3 - **Event-Based Traffic**: ($\lambda$, $\lambda$+1) minutes, $\lambda$ represents x-axis, Data packets: One data packet is sent for every event.




(b) Scenario 4: **Event-Based Bursty Traffic**: ($\lambda$, $\lambda$+1) minutes, $\lambda$ represents x-axis, Data packets: 60 packets are sent at 1 packet/second.

Figure 5: Number of discarded data packets after injecting new code.

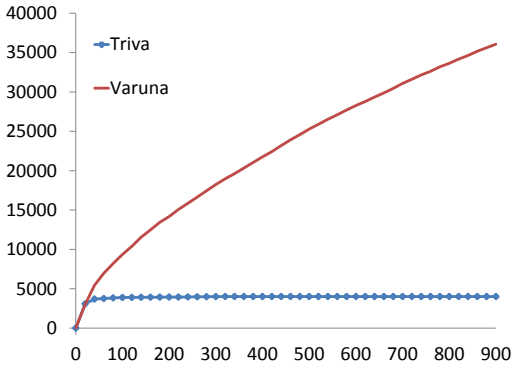



Figure 6: Scenario 5 - **Bursty Traffic**: 30 packets are sent at 1 packet/second, $\lambda = 5$ minutes for bursty traffic.

### *6.5. Experimental Results*

We have implemented Triva and Varuna on TelosB platform based nodes. Our implementation of Triva takes 989 bytes in memory, which includes two tables, each of size 20 entries of 2 bytes, and other algorithm related variables. To have an asymmetric link between nodes, we set the power levels of all nodes to 31 except one which is set to 2. In our experiment, we used 11 nodes and we measured the number of advertisement packets. Table 6.5 shows the results of our experiment. Triva, after around 100 minutes, stopped sending advertisement packets at 104. On the other hand, Varuna, about every minute, sends 11 advertisement packets.

| | |
|---|---|
| Triva | 104 |
| Varuna | 661 |

Table 2: The number of transmitted advertisement packets.

## 7. Conclusion

In this paper, we proposed a new code dissemination protocol called Triva that leverages the properties of two well-known code dissemination protocols, namely Trickle and Varuna. It adapts both Trickle and Varuna to achieve energy efficiency and low dissemination latency in event-based sensor networks and in the presence of asymmetric links. Our results show that Triva

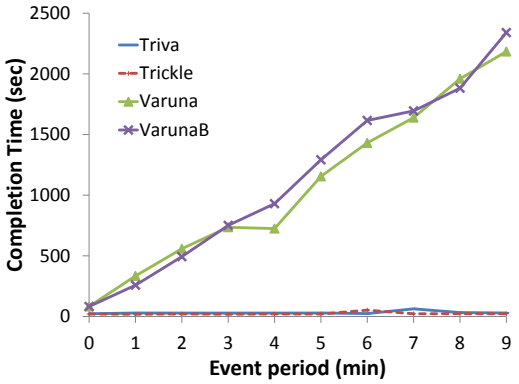



Figure 7: Scenarios 6 and 7: Completion time as a function of $\lambda$, the event period. Scenario 1: Data packets generated every $(0 \ldots 1)$ minute, Scenario 3: 30 packets per event at 1 packet/second.

outperforms both Trickle and Varuna (i) in periodic traffic, (ii) event-based traffic, (iii) bursty traffic, and (iv) in networks with asymmetric links.

**References**

[1] A. Arora and et al. A line in the sand: A wireless sensor network for target detection, classification, tracking. *Computer Networks (Elsevier)*, 46(5), 2004.

[2] Nouha Baccour, Anis Koubaa, Luca Mottola, Marco Zuniga, Habib Youssef, Carlo Boano, and Mario Alves. Radio Link Quality Estimation in Wireless Sensor Networks: a Survey. *ACM Transactions on Sensor Networks*, 2011.

[3] Inc. Crossbow Technology. Mote in-network programming user reference, www.tinyos.net/tinyos-1.x/doc/xnp.pdf. 2003.

[4] 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. In *SenSys*, pages 153–166, 2006.

[5] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys '04, pages 81–94, 2004.

[6] Jaein Jeong and David Culler. Incremental network programming for wireless sensors. In *IEEE Sensor and Ad Hoc Communications and Networks (SECON*, pages 25–33, 2004.

[7] Sandeep S. Kulkarni and Limin Wang. Mnp: Multihop network reprogramming service for sensor networks. In *In Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS*, pages 7–16, 2005.

[8] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, SenSys '03, pages 126–137, 2003.

[9] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, pages 2–2, 2004.

[10] Ren Ping Liu, Zvi Rosberg, Iain B. Collings, Carol Wilson, Alex Y. Dong, and Sanjay Jha. Overcoming radio link asymmetry in wireless sensor networks. In *Proceedings of the IEEE 19th International Symposium on Personal, Indoor and Mobile Radio Communications, PIMRC 2008, 15-18 September 2008, Cannes, French Riviera, France*, pages 1–5. IEEE, 2008.

[11] A. Mainwarig and et al. Wireless sensor networks for habitat monitoring. In *Proceedings WSNA*, pages 88–97, 2002.

[12] Luca Mottola, Gian Pietro Picco, Matteo Ceriotti, Ştefan Gună, and Amy L. Murphy. Not all wireless sensor networks are created equal: A comparative study on tunnels. *ACM Trans. Sen. Netw.*, 7(2):15:1–15:33, September 2010.

[13] Sze-Yao Ni, Yu-Chee Tseng, Yuh-Shyan Chen, and Jang-Ping Sheu. The broadcast storm problem in a mobile ad hoc network. In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, MobiCom '99, pages 151–162, 1999.

[14] Rajesh Krishna Panta, Madalina Vintila, and Saurabh Bagchi. Fixed cost maintenance for information dissemination in wireless sensor networks. In *Proc. SRDS*, pages 54–63, 2010.

[15] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *Proc. of the* $2^{nd}$ *Int. Conf. on Wireless Sensor Networks and Applications (WSNA)*, 2003.

[16] Thanos Stathopoulos, John Heidemann, and Deborah Estrin. A remote code update mechanism for wireless sensor networks. Technical Report CENS-TR-30, UCLA, Center for Embedded Networked Computing, 2003.

[17] Jorg Nolte Stephan Mank, Reinhardt Karnapke. Mac protocols for wireless sensor networks: Tackling the problem of unidirectional links. *International Journal on Advances in Networks and Services*, 2(4):218–229, 2009.

[18] S. Tilak, N. Abu-Ghazaleh, and W. Heinzelman. A taxonomy of wireless micro-sensor network models. *ACM Mobile Computing and Communication Review*, 6(2), 2002.

[19] T.Masuzawa and S.Tixeuil. Stabilizing locally maximizable tasks in unidirectional networks is hard. In *Proceedings of International Conference on Distributed Computing Systems*, 2010.

[20] Gilman Tolle and David E. Culler. Design of an application-cooperative management system for wireless sensor networks. In *EWSN*, pages 121–132, 2005.

[21] R. Chandra V. Ramasubramanian and D. Mosse. Providing a bidirectional abstraction for unidirectional ad-hoc networks. In *In Proceedings of the IEEE Infocom*, 2002.

[22] Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 381–396, 2006.

[23] Kamin Whitehouse, Gilman Tolle, Jay Taneja, Cory Sharp, Sukun Kim, Jaein Jeong, Jonathan Hui, Prabal Dutta, and David Culler. Marionette: using rpc for interactive development and debugging of wireless embedded networks. In *Proc. IPSN*, 2006.

[24] H. Zhang, A. Arora, Y. Choi, and M. Gouda. Reliable bursty convergecast in wireless sensor networks. *Computer Communications*, 30(13), 2007.