BDP: A Bloom Filters Based Dissemination Protocol in Wireless Sensor Networks

Tao Chen*, Deke Guo*, Xue Liu†, Honghui Chen*, Xueshan Luo*, and Junxian Liu*
*College of Information Systems and Management, National University of Defense Technology, Changsha, China
†School of Computer Science, McGill University, Montreal, Quebec, Canada

Abstract

There is a growing need for enabling reprogramming in a working sensor network. We prefer to meet the requirements remotely instead of collecting all deployed sensors. Identifying the version difference of data items, having the same key, could significantly reduce the communication overhead, because only those out-of-date items should be updated at each sensor. Previous protocols need to exchange multiple messages to identify a version difference between two items with the same key. In this paper, we propose a reliable and energy efficient data dissemination protocol (BDP) with less propagation delay. BDP uses Bloom filters to identify a version difference between two items with the same key, and find the new one between two items having the same key but different versions. Through comprehensive simulations, we show that BDP outperforms previous work in terms of energy cost and propagation delay of updating new items with high reliability.

1. Introduction

Wireless sensor networks (WSN) are more and more popular and widely used in many applications those days, such as habitat or environment monitoring [1], [2], [3], structural health monitoring [4], [5], medical care [6] and other scientific observations [7]. In those applications, sensors are usually deployed at places where it is not convenient for human to reach. In these scenarios, one of the challenging issues is how to update the applications on sensors by in-network reprogramming as the situation changes.

Data dissemination protocols emerge to solve this issue and are generally classified into two types according to the content of data. The first type of protocols needs to redistribute large file (e.g. binaries or virtual program) into the whole sensor network, including XNP [8], MOAP [9], Deluge [10], MNP [11], Sprinkler [12], Typhoon [13], CORD [14] and the protocols in works of Levis et al. [16] and Gnawali et al. [17]. Those protocols usually mean complete system or application-level reprogramming. The second type of protocols only needs to redistribute configuration parameters. Protocols of this types adjust the behavior of applications on sensors by changing values of the parameters, and thus are more energy efficient than those of the former type because the data needed to be disseminated is much less. Drip [18] and DIP [19] are two representative examples of the second type of protocols, and have been implemented in TinyOS as core components.

Dissemination protocols can also be classified into two types according to the way the protocols disseminate data. The first type disseminates in a two-phase way. The protocols, such as Sprinkler and CORD, choose a set of core nodes (usually chooses a connected dominating set) and update these nodes with new data. Those core nodes are responsible to update the rest of nodes in a parallel way. The two-phase separation reduces the number of message collisions, but it does not support updating newly joined nodes after the updating period for existing nodes has passed. Moreover, it might fail if the topology changes after the first phase, which probably happens because of the instability of wireless channel or mobile circumstances. The second type of protocols disseminates data in a one-phase way likes flooding in a flat (not layered) topology. These protocols include MOAP, Deluge, MNP, Drip and DIP. Data is propagated neighbor by neighbor throughout the whole network. This type of protocol is more reliable and robust than the first type of protocols because it needs no global topology information.

This work focuses on disseminating a number of data items in one-phase way and designs protocols which are robust to instable wireless channels and topologies. In some applications, it needs to update only a few items to change the behavior of nodes. Thus, it may reduce communication overhead if nodes know which item needs to be updated. Previous work argues that the scan and search techniques could reduce message cost due to identifying a new item between two items with the same key [19].

Bloom filters are more space efficient for representing a set of elements than other data structures, such as binary search trees and hash tables. This is a good property in sensor networks because the length of packet payload is limited. Bloom filters also have a fantastic property that they need constant time to check whether an element is in the set, no matter how many elements have already been added to the set. These two properties make it possible to reduce communication overhead in sensor networks.

This paper proposes a Bloom filters based dissemination protocol (BDP), which can be used for not only updating configuration parameters but also distributing bulk data. BDP is more energy efficient than other protocols with less

978-1-4244-5113-5/09/$25.00

delay. The probability that BDP fails to update all sensors is very small if the false positive rate of Bloom filters is well controlled. Thus, BDP is also a reliable protocol.

The main contributions of this work are as follows.

- We propose BDP, an efficient, fast and reliable dissemination protocol, which does not base on global topology information or virtual infrastructure. It uses Bloom filters to identify the version difference even find the new version between two items having the same key. To our knowledge, this is the first attempt to use Bloom filters to find the new version between two items having the same key in wireless sensor networks.
- Through extensive simulations, we evaluate BDP and compare it with DIP under different network topologies. To the best of our knowledge, BDP is the most energy efficient protocol to disseminate a large number of data items.

The rest of this paper is organized as follows. We briefly describe Bloom filters and related work in Section 2. We discuss the detail of BDP in Section 4. Section 5 analyzes the false positive problem of BDP. Section 6 evaluates BDP and compares it with DIP under different network configurations. We conclude the work in Section 7.

2. Preliminaries

2.1. Trickle

Trickle is a self-regulating algorithm for code propagation and maintenance in wireless sensor networks [20]. It exchanges metadata in a gossip way, which does not need to know the global information. Trickle periodically broadcasts messages to all neighbors within transmission range. Motes who hear the message could either get an update or detect a need for update. Trickle also provides a straightforward way to control the transmission overhead. In each propagation period, a mote is suppressed for propagating metadata if the mote has heard some metadata which is identical to its local metadata.

Trickle has some good features. It provides rapid data propagation with low maintenance, and scales well in various network density. The ideal results depend on some assumptions, such as no packet loss and single-hop network. In practical environment, the number of redundant messages grows, however, is still controlled within the acceptable boundary. Another good feature of Trickle is that it uses a scheme to dynamically adjust the gossip interval. Trickle achieves an optimal trade off between communication overhead and propagation speed.

2.2. Bloom Filter

Bloom filter was first introduced in 1970s. It is now widely used in database and networking applications because it is space-efficient and can reduce communication overhead [21]. A Bloom filter (BF) is a compact data structure for probabilistic representation of a set $S = \{s_1, s_2, \ldots, s_n\}$ of n elements. It uses a vector of m bits to represent the elements. These m bits are initially set to 0. Then k independent hash functions h_1, h_2,..., h_k are used to add the elements into the BF. For each element s_i, bit $h_j(s_i)$ is set to 1 for $j = 1, \ldots, k$. To do a membership query, we check all bits $h_j(x)$ of an element x, $j = 1, \ldots, k$. If all these bits were set to 1, we may consider x belongs to the set S with some probability. Otherwise, we may infer that x is not a member of S.

A major aspect of optimizing BF is to maximize the probability of true match, which in turn means to decrease the probability of making the wrong inference that an element belongs to set S (this is also called false positive or false match). False positive is due to a filter collision, in which all associated bits were set to 1 by other elements in set S. Given the assumptions that hash functions are perfectly random and independent to each other, the false positive rate is

$$f \approx (1 - (1 - 1/m)^{kn})^k. \quad (1)$$

The four important parameters of BF are m, n, k, f. Given the combination of values of any two or three parameters, we can optimize the value of remainder parameters. We will discuss this issue in Section 5.

2.3. Related Work

There are some data dissemination protocols in wireless sensor networks. A common target of XNP [8], MOAP [9], Deluge [10], MNP [11], Sprinkler [12], Typhoon [13], Flush [24] and CORD [14], is to realize reliable bulk data dissemination. Performance of some bulk data dissemination protocols are analyzed in [15]. Typhoon and CORD both assume that all the receivers should be updated. Flush is a receiver-initiated transport protocol. Some other protocols aim at updating configuration parameters, such Drip [18] and DIP [19]. They both use Trickle algorithm. Drip has a constant latency of $O(T)$ to disseminate T items, and the transmission rate also grows with $O(T)$. The most related protocol to our work is DIP which is currently the most efficient one to disseminate large number of data items. DIP uses a hybrid method of scan and search combining with Bloom filters to detect data difference and then disseminate data. DIP outperforms scan and search methods, but still causes $O(\log(T))$ messages when identifying a changed item among T data items. Furthermore, DIP accelerates the updating process through an estimation method, however, may fail to update all items completely. DIP uses a BF to recognize whether two items having the same key are different in version, however, cannot directly tell which is the newer item. The major objective of BDP is to directly

Table 1. Main Notations

Term	Definition
S	a set of elements
s_i	an element in S
n	the number of elements in S
h_i	a hash function
BF	a Bloom filter
n_0	the capacity of a BF
k	the number of hash functions of Bloom filter
m	the bits(vector) length of Bloom filter
x	an element to do membership query
f	the false positive rate of Bloom filter
T	the number of data items on each sensor node
N	the number of new data items
key	an exclusive identifier of a data item
ver	a version number of a data item
D_{key}	a data item with an identifier key
M or M_{key}	a metadata tuple of data item (D_{key})
L or L_{key}	a data value (of D_{key})
H_{New}	state value "new"
H_{Old}	state value "old"
$H_{Unknown}$	state value "unknown"
H_{key}	state value of a data item D_{key}
DM	a data message
SM	a summary message
VM	a vector message

identify the version difference even the newer item of two items with the same key. In addition, BDP is more energy efficient and scalable than DIP.

BDP uses Bloom filters in a way similar to direct Bloom filter (DBF) in Bonomi's work [23] under a different application background. Lookup operation is done with a state as a given input; that is, do membership query for a $(id, state)$ pair. If the state of corresponding id is not the lookup input, then one has to check all the possible states. In the scenario of DBF, the number of possible states is not too large and the length of DBF is not limited.

3. Problem Definition and Assumptions

We focus on the problem of disseminating a number of data items in a flat sensor network. Links between nodes may be asymmetric. There are T data items on each sensor node. These items are organized as a list, with an index assigned to each item. Of all T items, only N items need to be updated. Each date item D is associated with a metadata tuple $M = (key, ver)$, where key and ver denote an exclusive identifier and a version number, respectively. We call items with the same key as **associated items** on different nodes. Any version number is set to an integer, and is added by one when content of the associated data item is updated. If a data item has a larger version number than an associated item on another node, the data item is **newer** than the associated item. That is, the associated item is **older** than that item. We use D_{key} to denote a data item with an identifier key. Let M_{key} denote a metadata tuple of D_{key}, respectively. Each sensor node has the same set of items whose versions are initially set to zero. The value of item D_{key} is denoted as L_{key}. For ease of presentation, we treat L_{key} as a numerical value, but our protocol is not limited to applications with numerical data items. The identifier of a data item is unchangeable while the value and version number of an item can be changed.

Inspired by [23], we use H_{New}, H_{Old}, and $H_{Unknown}$ to describe three different states for each data item. H_{New} means that the item is newly updated or newer than an associated item on some neighbor nodes. H_{Old} denotes that there is no newer associated item in the neighborhood. H_{Old} is the default state value for each item on all nodes. $H_{Unknown}$ denotes that the item has a different version number with some associated items in the neighborhood. Each data item is assigned one of the three state values. Main notations used in this work are listed in Table 1.

4. BDP

4.1. Overview

The difference between BDP and any other dissemination protocols is that BDP mainly uses BF to identify the version difference of any two associated items. If the version difference varies in a small range, BDP can find which node has the newer data item by conducting membership queries locally at the cost of only one message. BDP speeds up the update process and depresses redundant broadcast by using state values of data items and achieves high energy efficient. The last but not least strongpoint of BDP is that it needs no topology or other global information and thus is robust to environment change and node failures. Sensor nodes are supposed to be autonomous and communicate with one hop neighbors in a broadcast manner. Initially, all nodes have the same data items with the same version number of zero. Each node uses a Trickle timer to fire an event periodically. An event triggers a node to check all state values of its data items, and then sends one category of message accordingly. In the ideal cases, all nodes only send summary messages with maximum gossip interval after the dissemination process ended. If the whole network keeps this status for a relative long time, we say the network reaches a *steady state*. At this time, either all nodes share the same data items with the same versions, or there exists version difference(s) but the dissemination protocol does not recognize it (them). In the rest of this Section, we will introduce three important part of BDP, namely message types, decision making and state value updating strategies.

4.2. Message Types

The three types of messages used by BDP are as follows.

Data message: Data message is used to deliver new data to neighbors. It is similar to the same term used by many other dissemination protocols, such as Drip and DIP. A data message, denoted as a tuple $DM = (M, L)$, includes data value and metadata of an item.

Summary message: Summary message is the major difference between BDP and other dissemination protocols. A summary message is used to discover a difference in version numbers of some associated items in neighborhood. A summary message is denoted as a tuple $SM = (BF, d, c)$, where d is an index position of the first item of the set in local list which stores all items, c is a counter which records the number of items contained in the BF. This work uses the exclusive identifier (key) of an item as the index. A node represents the metadata of each item by a BF, and broadcasts the BF as a summary message when needed. A receiver can conduct the membership queries of the metadata of its local items on the basis of a received summary message. If the queries show that a metadata is not represented by the received BF, a local item is different in the version with an associated item in a neighbor. Due to the false match problem of BF, a different metadata might be missed.

To make the false positive rate of a BF under a threshold, more bits should be allocated such that the BF can contain more elements. Due to the limitation of packet size, a BF needs multiple packets to accommodate the m bits if m is larger than the payload size of a packet. Here, we borrow the idea from dynamic Bloom filters [25], and allocate a constant length of bytes to store a BF. To ensure that f is less than a given value, BF could contain at most n_0 elements. We call n_0 the capacity of BF. If $T > n_0$, we select the first n_0 items to construct a BF for in first summary message, then use the next n_0 ones to construct another BF for the second summary message, and so on. When all items in the items list have been selected, the process repeats by selecting items from the head of the items list. Note that d and c indicate the index and number of items in BF, respectively.

Vector message: A summary message is usually enough for a receiver to determine whether each local data item is newer than an associated item on sender. A receiver, however, cannot draw a conclusion and thus marks the state value as $H_{Unknown}$ in some special conditions. To address this issue, a vector message is employed to send metadata of items whose state value are $H_{Unknown}$. A vector message contains an array of metadata and a counter which records the number of associated tuples in the message. A vector message can be denoted as $VM = (c, M_{key_1}, M_{key_2}, ..., M_{key_c})$, where $M_{key_1}, M_{key_2}, ..., M_{key_c}$ are a series of metadata tuples and c recorders the number of metadata tuples in a message.

4.3. Decision Making

After introducing the three type of messages used by BDP, we propose the following rules for each node to make decision on which type of message it should send. All decisions are made based on the states of local data items.

- If any local data item has the state value of H_{New}, the node sends a data message which contains the key and the value of the item, then changes the state value of the item to H_{Old}.
- If no local item has the state value of H_{New} but some items have the state value of $H_{Unknown}$, the node sends a vector message.
- If all the state values of local items equal to H_{Old}, the node sends a summary message.

Note that we allocate different priority levels for those state values. Specifically, H_{New} has the highest priority level. This favors the dissemination of new data items. If more than one item has the state value of H_{New}, we choose the first one in index sequence to construct a message. The second highest priority level is assigned to $H_{Unknown}$. If no newer item is discovered, BDP chooses those items which may be newer to construct a message. This is also more likely to favor the dissemination of potential newer items. It uses the same index sequence to choose items if there are multiple items with the state value of $H_{Unknown}$. The lowest priority level is assigned to H_{Old}. If all state values are H_{Old}, it means that there is no existing clue for new data items. BDP tries to produce clue itself by sending summary message.

4.4. State Value Updating

After receiving a message, a node updates the state value even the data and version number of each respective item, according to the type and content of the received message. If the type of received message is **data**, the node compares the version number of an item in the message with that of a local item which has the same key as showed in Algorithm 1. If the local item is newer, the node changes the state to H_{New}. If the local item is older, the node updates the data value and sets the state value to H_{New}. In other conditions, the node sets the state value to H_{Old}.

Algorithm 1 reduces the communication overhead effectively in any case. First, nodes who overhear a data message with older items will set the state values of respective local items to H_{New}. According to rules discussed in Section 4.3, nodes actively update their neighbors so as to reduce the communication overhead due to useless update of an item with older version or identify a version difference. Second, nodes who overhear a newer data could make themselves be updated and propagate the new data to their neighbors. Third, nodes who hear a data which has the same version with the local one, are suppressed to send messages to avoid flooding. Although a node may fail to update their neighbors who host older data, the neighbors might be updated with the supplement of summary and vector messages or by other neighbors.

Algorithm 1 Receive (DM)

Preconditions: H_{key_1} stands for the state value of D_{key_1}.

1: $key_1 \leftarrow DM.M.key$
2: $ver_1 \leftarrow DM.M.ver$
3: $ver_0 \leftarrow M_{key_1}.ver$
4: **if** $ver_0 > ver_1$ **then**
5: $\quad H_{key_1} \leftarrow H_{New}$
6: **else if** $ver_0 < ver_1$ **then**
7: $\quad L_{key_1} \leftarrow DM.L$
8: $\quad H_{key_1} \leftarrow H_{New}$
9: **else**
10: $\quad H_{key_1} \leftarrow H_{Old}$
11: **end if**

If the type of received message is **summary**, the receiver tries to identify the difference of version according to Algorithm 2. First, it gets the range information of elements set in the BF, and obtains a set of local associated items. Then, the node conducts membership query for each of these items. If the query result is true, the local associated item is identical to the one on the sender. If the query result is false, a version difference is found. Furthermore, the node tries to identify whether the local data item is newer than that on the sender only based on the received BF and local information. Given a certain key, for each $ver \in [ver_2 - \alpha, ver_2)$ (α is a positive integer, ver_2 is the version number of the local item with a different version), node does a membership query again. We call this procedure as *test scan* and α as *scan range*. If it responses a positive result, the node can tell whether the local item is newer according to the difference between ver and ver_2.

Algorithm 2 reveals that BDP is efficient since it can find the version difference between items having the same key and even identify which is newer at the cost of a single summary message. For other protocols for example DIP, the same process needs to transmit several messages. The limitation of Algorithm 2 is that it might induce false match problem in two scenarios (line 6 and line 11). The fundamental cause is that BF only provides a probabilistic membership query result.

In the first scenario, an item with different version may be wrongly considered that it has the same metadata as a local item, and thus Algorithm 2 cannot reveal the existing difference in versions. It is well-known that this problem can not be avoided in theory and practice. The false positive rate, however, can be controlled at a low level if we assign values for those parameters of BF carefully. We will discuss how to control the false positive rate in detail in Section 5.

In the second scenario, Algorithm 2 wrongly reckons a local item which may be an older one as a newer one. This mistake could be corrected by algorithm 1. When an item is wrongly considered as a newer one, the node will broadcast a data message with the item. If some neighbors

Algorithm 2 Receive (SM)

Preconditions: $MQuery(M)$ denotes a membership query of a BF. It returns true if M is represented by the BF, otherwise returns false. A function $GetKey(d)$ can get the exclusive identifier of an item with an index d. Let α indicate the range for test scan and $TestM$ represent a temporal metadata tuple.

1: $d_1 \leftarrow SM.d$
2: $c_1 \leftarrow SM.c$
3: $BF_1 \leftarrow SM.BF$
4: **for** $i \leftarrow d_1, d_1 + c_1$ **do**
5: $\quad key_2 \leftarrow GetKey(i)$
6: $\quad$ **if** $MQuery(M_{key_2})$ is $false$ **then**
7: $\quad\quad ver_2 \leftarrow M_{key_2}.ver$
8: $\quad\quad TestM.key \leftarrow key_2$
9: $\quad\quad$ **for** $j \leftarrow 1, \alpha$ **do**
10: $\quad\quad\quad TestM.ver \leftarrow ver_2 - i$
11: $\quad\quad\quad$ **if** $MQuery(TestM)$ is $true$ **then**
12: $\quad\quad\quad\quad H_{key_2} \leftarrow H_{New}$
13: $\quad\quad\quad\quad$ Return $null$
14: $\quad\quad\quad$ **end if**
15: $\quad\quad$ **end for**
16: $\quad\quad H_{key_2} \leftarrow H_{Unknown}$
17: $\quad$ **end if**
18: **end for**

holding a newer item hear the message, it may broadcast a data message immediately to clear the difference. But if we conduct *test scan* in a range $[ver_2 - \alpha, ver_2 + \alpha]$, an item may be falsely considered as an older one. As shown in Figure 1, nodes A and B have a data item D_{key} while D_{key} on node A is newer than that on node B. When performing the test scan operation, node A and node B wrongly believes that their items $D_{key}$are older at the same time. It is clear that this result leads to deadlock. Both of nodes A and B know that there is a different item between them, however, neither of them actively claims to be updated. To address this scenario, BDP only conducts *test scan* operation on single side, and sets the state value to $H_{Unknown}$ once the *test scan* operation ends without returning null.

If the type of a received message is **vector**, the receiver simply compares the version numbers in received message to the local ones as shown in Algorithm 3. If the local item is newer, the receiver sets the state value of the item to H_{New}, otherwise sets the value to H_{Old}. A vector message deals with the state value of $H_{Unknown}$ and is a supplement of a summary message.

In aforementioned algorithms, the state value of each item might be updated only when receiving a message. Actually, a sender might also change the state value of an item to H_{Old} when sending a corresponding data message. The objective is to avoid transmitting the same data message many times.

Algorithm 3 Receive (VM)

Preconditions: M_{key_i} stands for metadata of local data item with identifier key_i.

1: $c_1 \leftarrow VM.c$
2: **for** $i \leftarrow 1, c_1$ **do**
3: $ver_1 \leftarrow VM.M_{key_i}.ver$
4: $key_1 \leftarrow VM.M_{key_i}.key$
5: $ver_0 \leftarrow M_{key_i}.ver$
6: **if** $ver_0 > ver_1$ **then**
7: $H_{key_i} \leftarrow H_{New}$
8: **else**
9: $H_{key_i} \leftarrow H_{Old}$
10: **end if**
11: **end for**

4.5. Example

To demonstrate the logic of identifying a version difference in BDP, we compare it with DIP [19] under a simple example with two nodes. We assume that DIP only sends vector messages at the bottom of the hash tree. Figure 2 shows that two nodes running DIP need to exchange multiple messages to identify a difference and get an update. The two nodes running BDP, however, only need to exchange two or three messages if all items can be contained in only one summary message. In BDP, node A sends a summary message with a BF which contains metadata of all data items. Node B can discover that the local item 5 is newer after invoking Algorithm 2, and then sends a data message to update Node A. If Algorithm 2 identifies a version difference but does not know whether the local item is newer, Node B sends a vector message with the metadata of item 5. Node A receives the vector message and makes decision on sending the data or not. Intuitively, BDP is more efficient than DIP since it results in less transmission of messages.

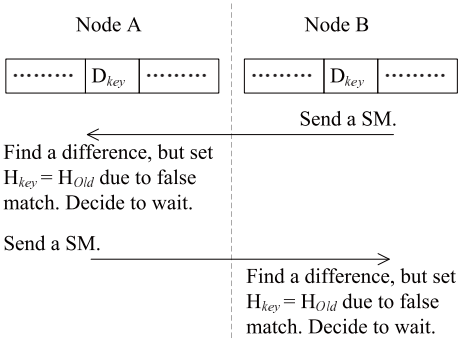


Figure 1. An illustrative example of deadlock.

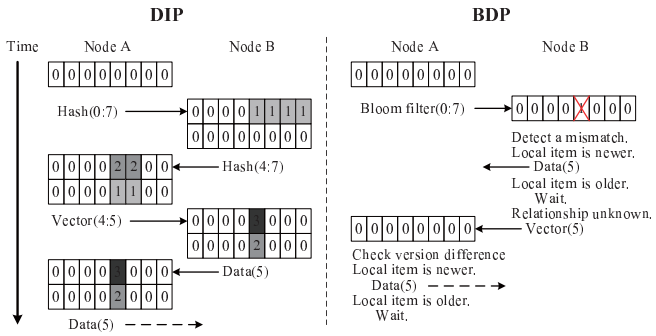


Figure 2. Processes of DIP and BDP with two nodes.

5. False Positive Control

Despite the benefits offered by BF, it may yields a false positive due to hash collisions, for which it wrongly determines an item belongs to a data set when it is actually not. In BDP, the following two factors dominate the false positive rate of BF.

5.1. Parameters Settings of BF

False positive problem of BF is considerably serious in wireless sensor networks because of the limitation of the packet size. Specifically, the capacity of a BF suffers the limited payload of a single packet. The false positive rate of a BF increases to an unacceptable level if the cardinality of set represented by the BF exceeds the capacity of the BF.

To make the false positive rate of a BF stay at an acceptable level, we impose constraint on the number of items represented by a BF. That is, we reduce the value of n. After constructing a BF, a node only represents part of local items using the BF as described in section 4.2. Given the value of m and f, the maximum value of n could be easily calculated by using some derivative techniques. According to equation (1), we have the following expressions

$$n \approx \frac{\ln(1 - f^{1/k})}{k \ln(1 - 1/m)}. \tag{2}$$

Assuming that equation (2) is continuous, we may approximately infer that when

$$\frac{dn}{dk} = \frac{\frac{f^{1/k-1}}{k(1-f^{1/k})} - \ln(1 - f^{1/k})}{k^2 \ln(1 - 1/m)} = 0, \tag{3}$$

n achieves an optimal value.

For example, if 24 bytes are allocated to store a BF ($m = 192$), to restrict the false match rate below 0.1%, as shown in Figure 3 the optimal value of n varies according to the value of k. When $k = 10$, n reaches the maximum value 13, that means the BF can contain up to 13 elements.

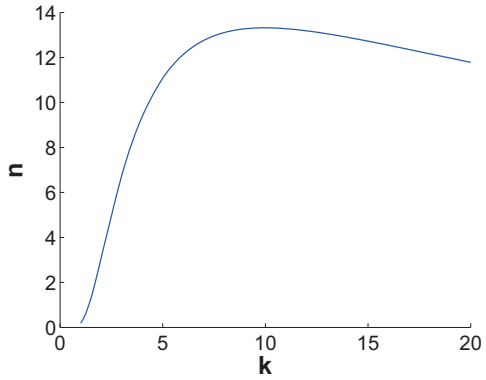


Figure 3. The value of n when m=192 and f=0.001.

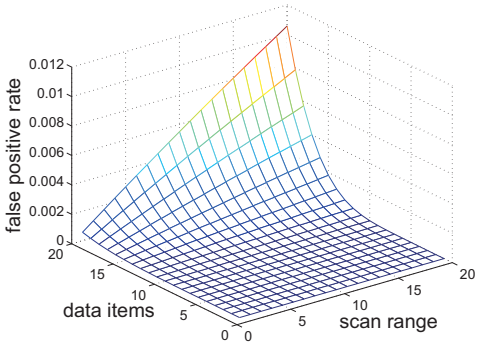


Figure 4. False positive rate of test scan, where m=192, k=10.

5.2. Range of Test Scan

Another important factor which may increase the false positive rate is the length of *test scan*. In the worst case, BDP has to scan all the versions in the range $[ver_2-\alpha, ver_2)$ by conducting α membership queries. To make query result reliable, we should ensure that the false positive rate of all these α membership queries is under a certain threshold. Intuitively, the false positive rate increases as α grows. According to equation (1) and Algorithm 2 , we have the expression

$$f_{scan} \approx f(1-(1-f)^{\alpha}), \tag{4}$$

where f_{scan} denotes the maximum false positive rate induced by a test scan operation.

Figure 4 plots the distribution of false positive rate when $m = 192$. We can see that for any given value of n, the false positive rate increases almost linearly as the scan range increases. A large scan range may reduce communication overhead, but also has chance to increase false positive rate. More false positive judgments, however, results in additional communication overhead according to Algorithm 2. Therefore, the scan range should be restricted to a small value. Actually, the version difference should not be very large as long as the environment (wireless channel) does not change severely and no new node joins the network. Simulation results show that the version difference is only one in most cases. No matter how large is the range assigned for the test scan, it is always possible that no false positive result is returned even we scan the entire range. It is caused by the large version difference between a local item and an associated item on the sender. In this case, a node sets the state value of an unidentified item as $H_{Unknown}$, and adds the metadata into a vector message for further identification.

6. Evaluation

We evaluate the performance of BDP from three aspects and compare BDP against DIP which is the most energy efficient among previous dissemination protocols.

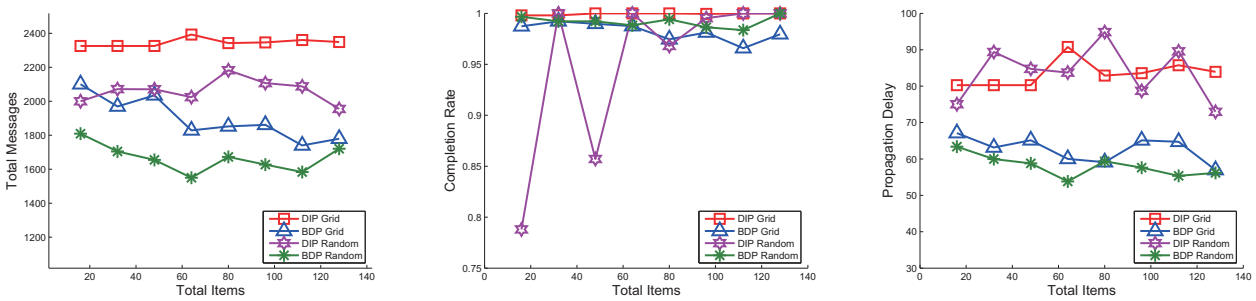
6.1. Simulation Methodology and Metrics

We implement BDP in TinyOS 2.0. In our implementation, we use 24 bytes for a BF in summary message. Vector messages can store metadata of no more than 3 data items. We use the same Trickle timer as DIP. The maximum and minimum intervals are set to 1 minute and 1 second, respectively. We adopt TOSSIM [22] to simulate the multihop dissemination process in grid and random topologies. We set the path loss exponent to 4.7, and shadowing standard deviation to 3.2, reference distance to 1 meter, and power decay for the reference distance to 55.4db. The radio noise is set to -105dBm. The standard deviation of additive white Gaussian noise is set to 4. We compare the performance of BDP and DIP in the terms of the three metrics.

- **Total Message Cost**. The energy cost of communication is a critical metric in WSNs. Here, we use the total number of messages that all nodes in a WSN have sent as an approximate energy cost.
- **Completion rate**. Dissemination protocol should be reliable. Once a group of items are injected to the network, associated items on all nodes should be updated. We define the completion rate as the ratio of nodes which have updated the group of items to all nodes.
- **Propagation Delay**. We measure the whole network propagation delay, which is defined as the time needed for the entire network to get the last update before the network reaches a steady state defined in Section 4.1.

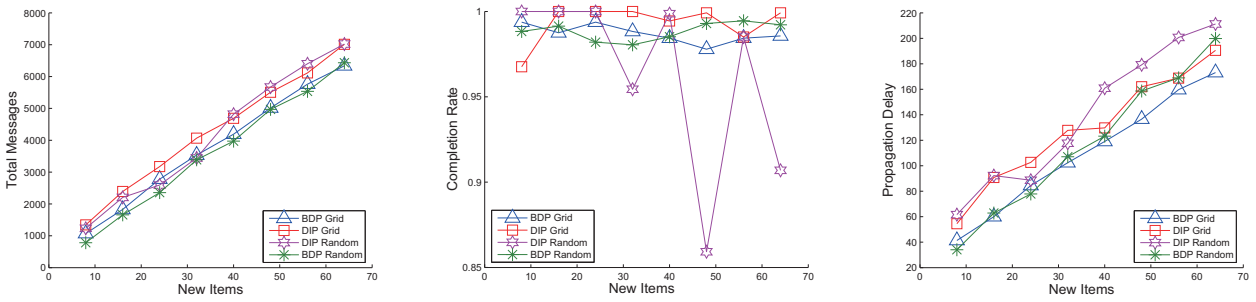
6.2. TOSSIM Evaluation

In our simulations based on TOSSIM, the following three topologies are employed. The first one is a 16×16 grid topology in which the inter-node distance is 2 units and each node has about 16 neighbors in its communication range.



(a) (b) (c)

Figure 5. Total messages, completion rate and propagation delay of DIP and BDP in grid and random topologies when N=16.



(a) (b) (c)

Figure 6. Total messages, completion rate and propagation delay of DIP and BDP in grid and random topologies of 256 nodes when T=64.

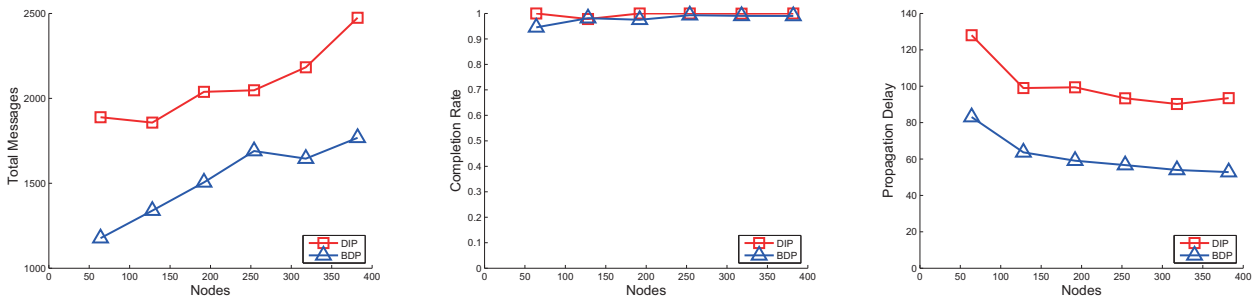
The second one is a random topology of 256 nodes within a terrain of $30units \times 30units$. The third topology has the same setting as the second topology except the density of nodes. The number of nodes ranges from 64 to 384.

After determining the topology, there still exist two parameters which influence the aforementioned three metrics. They are the number of total items (denoted as T) and that of new items (denoted as N). We fist use a basic scenario with 64 total items and 16 new items, and then measure the effect of variable N and T on the three metrics, respectively. In each individual scenario, we run the simulation 10 rounds and calculate the average value of each metric. In each simulation, source nodes are chosen randomly.

In our first experiment, we evaluate the three metrics under grid and random topologies with 256 nodes. The number of new items is 16, while the total items ranges from 16 to 128. The simulation results are plotted in Figure 5. Figure 5(a) shows that BDP saves about 10% to 20% messages than DIP to update all nodes. The reason is that BDP mainly uses BF to discover the version difference between two items and identify which one is newer. Figure 5(b) shows that neither DIP nor BDP could guarantee that each node finally updates all items, but both of them achieve high completion rate in most cases. BDP is more stable than DIP since DIP sometimes performs terribly and only updates a few items especially in the random topology. A possible reason is inaccurate use of the estimate value in DIP and is out of the scope of this work. Figure 5(c) shows that BDP is obviously faster and more stable than DIP in both topologies. BDP reduces the propagation delay of DIP by about 15% on average. We can also see that the communication overhead and propagation delay almost do not change (or not increase considerately) as T increases. It means that the value of T does not affect the performance too much.

As shown in Figure 5, the performance improvement of BDP fluctuates in random topology. This is because the performance of both DIP and BDP are influenced by topological features and the positions of source nodes, but the influences on the two protocols may be different. In a static topology, as we could see from Figure 5, curves of grid topology are smoother than those of random topologies.

Our second experiment uses the same topologies as the first experiment, sets the number of total items to 64, and varies the number of new items from 8 to 64. Figure 6 shows that BDP outperforms DIP in terms of the three metrics. The experimental results are in accord with that in the first experiment. That is, BDP is more energy efficient and faster than DIP and the completion rate of BDP is also



(a) (b) (c)

Figure 7. Total messages, completion rate and propagation delay of DIP and BDP in random topologies when T=64, N=16.

high. BDP is even more stable than DIP in the random topology. The communication overhead and propagation delay increase linearly as the number of new items(N) increases. It indicates that the value of N is the critical factor which influences the communication overhead and propagation delay, despite the influence of network topology.

Note that the above two experiments are carried out in topologies with constant density of nodes. To achieve more comprehensive results, we change the number of nodes from 64 to 384 in the random topology such that the node density ranges from 4 to 24. As shown in Figure 7, BDP overwhelms DIP in terms of communication overhead and propagation delay, without trade-off of reliability. The experimental results are in accordance with the first two experiments. Figure 7(a) shows that the communication overhead tends to increase linearly but not so sharp as the density increases. It means that the node density is another factor which influences the communication overhead. Figure 7(c) shows that the propagation delay decreases slightly as the node density increases.

In summary, BDP outperforms DIP in terms of the communication overhead and propagation delay in both grid and random topologies as the number of total items, the number of new items and the node density varies. Both BDP and DIP achieve high reliability in most cases and BDP is more stable since DIP sometimes can only update a few items.

7. Conclusion

In this paper, we propose BDP, a Bloom filter based dissemination protocol for large number of data items in multihop wireless sensor networks. BDP provides algorithms for autonomous nodes to update their data items in a neighbor by neighbor way without global topology information. Bloom filters are used to identify version differences even find the newer item from items on neighbor nodes, and make BDP work more efficiently with less propagation delay. Although false match problem is inevitable, we can associate critical parameters with appropriate values so as to control the false match rate at an acceptable level. We evaluate BDP and DIP via extensive simulations in grid and random topologies. The simulation results show that BDP is more energy efficient and faster than DIP and is more reliable in random topologies.

Acknowledgments

This work was supported in part by the Ph.D. Candidates Research Foundation of National University of Defense Technology under Grant No.0615, NSERC Discovery Grant 341823-07, NSERC Strategic Grant STPGP 364910-08 and FQRNT Grant 2010-NC-131844. The authors would like to thank Yunhao Liu and Xiangyang Li for their constructive comments.

References

[1] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk and J. Anderson. Wireless sensor networks for habitat monitoring. In *Proc. of the 1st ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, Atlanta, GA, September 2002.

[2] M. Li and Y. Liu. Underground Structure Monitoring with Wireless Sensor Networks. In *Proc. of the 6th International Conference on Information Processing in Sensor Networks (IPSN)*, Cambridge, MA, USA, April 2007.

[3] G. Barrenetxea, F. Ingelrest, G. Schaefer and M. Vetterli. SensorScope: Out-of-the-Box Environmental Monitoring. In *Proc. of the 7th Information Processing in Sensor Networks (IPSN)*, 2008.

[4] S. Pakzad, S. Kim, G. Fenves, S. Glaster, D.Culler and J. Demmel. Multi-Purpose Wireless Accelerometers for Civil Infrastructure Monitoring. In *Proc. of the 5th*

International Workshop on Structural Health Monitoring (IWSHM '05), Stanford, CA, September 2005.

[5] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenvs, S. Glaser and M. Turon. Health Monitoring of Civil Infrastructures Using Wireless Sensor Networks. In *Proc. of the 6th International Conference on Information Processing in Sensor Networks (IPSN)*, Cambridge, MA, USA, April 2007.

[6] CodeBlue. http://fiji.eecs.harvard.edu/CodeBlue

[7] Z. Yang, M. Li, and Y. Liu. Sea Depth Measurement with Restricted Floating Sensors. In *Proc. of the 28th IEEE International Real-Time Systems Symposium (RTSS)*, Tucson, Arizona, USA, December 2007.

[8] Crossbow, Inc. Mote in network programming user reference. http://www.tinyos.net/tinyos-1.x/doc/Xnp.pdf.

[9] T. Stathopoulos, J. Heidemann and D. Estrin. A Remote Code Update Mechanism for Wireless Sensor Networks. Technical Report CENS-TR-30, University of California, Los Angeles, November, 2003.

[10] J. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proc. of the 2nd ACM Conference On Embedded Networked Sensor Systems*, USA, 2004.

[11] L. Wang. MNP: multihop network reprogramming service for sensor networks. In *Proc. of the 2nd ACM Conference On Embedded Networked Sensor Systems*, USA, 2004.

[12] V. Naik, A. Arora, P. Sinha and H. Zhang. Sprinkler: A reliable and energy efficient data dissemination service for wireless embedded devices. In *Proc. of the 26th IEEE International Real-Time Systems Symposium (RTSS)*, 2005.

[13] C. Liang, R. Musaloiu-Elefteri and A. Terzis. Typhoon: A reliable data dissemination protocol for wireless sensor networks. In *Proc. of the 5th European Conference on Wireless Sensor Networks (EWSN)*, 2008.

[14] L. Huang and S. Setia. CORD: Energy-efficient Reliable Bulk Data Dissemination in Sensor Networks. In *Proc. of the 27th INFOCOM*, 2008.

[15] W. Dong, C. Chen, X. Liu, J. Bu and Y. Liu. Performance of Bulk Data Dissemination in Wireless Sensor Networks. In *Proc. of the 5th IEEE/ACM International Conference on Distributed Computing in Sensor Systems (DCOSS'09)*, Marina Del Rey, CA, USA, June 8-10, 2009.

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

[17] O. Gnawali, B. Greenstein, K. Jang, A. Joki, J. Paek, M. Vieira, D. Estrin, R. Govindan and E. Kohler. The TENET architecture for tiered sensor networks. In *Proc. of the First ACM Conference on Embedded Networked Sensor Systems (Sensys)*, 2006.

[18] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. In *Proc. of the Second European Workshop of Wireless Sensor Networks (EWSN)*, 2005.

[19] K. Lin and P. Levis. Data Discovery and Dissemination with DIP. In *Proc. of the 7th Information Processing in Sensor Networks (IPSN)*, 2008.

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

[21] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, volume 1, pages 485-509, 2004.

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

[23] F. Bonomi, M. Mitzenmacher and R. Panigrahy. Beyond bloom filters: from approximate membership checks to approximate state machines. In *Proc. of ACM SIGCOMM*, 2006.

[24] S. Kim, R. Fonseca, P. Dutta, A. Tavakoli, D. Culler, P. Levis, S. Shenker and I. Stoica. Flush: A Reliable Bulk Transport Protocol for Multihop Wireless Networks. In *Proc. of the 2nd ACM Conference On Embedded Networked Sensor Systems (SenSys)*, Australia, 2007.

[25] D. Guo, J. Wu, H. Chen and X. Luo. Theory and Network Applications of Dynamic Bloom Filters. In *Proc. of the 25th IEEE International Conference on Computer Communications(INFOCOM)*,Barcelona, Catalunya, Spain, April 2006.