DHV: A Code Consistency Maintenance Protocol for Multi-Hop Wireless Sensor Networks

Thanh Dang, Nirupama Bulusu, Wu-chi Feng, and Seungweon Park

Department of Computer Science,
Portland State University,
PO Box 751, Portland, OR, USA
`(dangtx,nbulusu,wuchi,spark)@cs.pdx.edu`

Abstract. Ensuring that every sensor node has the same code version is challenging in dynamic, unreliable multi-hop sensor networks. When nodes have different code versions, the network may not behave as intended, wasting time and energy. We propose and evaluate DHV, an efficient code consistency maintenance protocol to ensure that every node in a network will eventually have the same code. DHV is based on the simple observation that if two code versions are different, their corresponding version numbers often differ in only a few least significant bits of their binary representation. DHV allows nodes to carefully select and transmit only necessary bit level information to detect a newer code version in the network. DHV can detect and identify version differences in $O(1)$ messages and latency compared to the logarithmic scale of current protocols. Simulations and experiments on a real MicaZ testbed show that DHV reduces the number of messages by 50%, converges in half the time, and reduces the number of bits transmitted by 40-60% compared to DIP, the state-of-the-art protocol.

Key words: Code consistency, network reprogramming, sensor networks

1 Introduction

Experience with wireless sensor network deployments across application domains has shown that sensor node tasks typically change over time, for instance, to vary sensed parameters, node duty cycles, or support debugging. Such reprograming is accomplished through wireless communication using reprogrammable devices. The goal of network reprogramming is to not only reprogram individual sensors but to also ensure that all network sensors agree on the task to be performed.

Network reprogramming is typically implemented on top of data dissemination protocols. For reprogramming, the data can be configuration parameters, code capsules, or binary images. We will refer to this data as a *code item.* A node must detect if there is a different code item in the network, identify if it is newer, and update its code with minimal reprogramming cost, in terms of convergence speed and energy.

Network reprogramming must address two main challenges — *node unreliability* and *network dynamics*. In a static network, sensor nodes run on batteries, leading to inconsistent behaviors, with periods of no operation followed by active periods. Radio communication is also known to be unreliable and dynamic with intermittent connectivity and link asymmetry. Sensor networks are becoming more dynamic due to mobility. Mobile sensor nodes might dynamically leave and join a network, requiring that they also vary their tasks dynamically. For instance, a sensor equipped car might be tasked to report carbon emissions within city A but to report temperature in city B. In this case, even if code items are disseminated correctly to all connected nodes during the reprogramming period, some nodes may not have the current code version.

Naturally, this creates the need for a protocol to ensure that all network nodes have the same up-to-date code items, which we refer to as a *Code Consistency Maintenance Protocol* (CCMP). Depending upon the application, the bandwidth and energy consumption of a CCMP can be comparable to sensing, particularly for networks with mobility or large churn. The key requirements for a CCMP are that they:

- ensure all network nodes *eventually have the same updated code*
- enable a node with an old code item to discover a newer code item and update it with *low latency*
- conserve energy and bandwidth
- scale with both the network size and the total number of code items

We represent code items as a set of *(key, version)* tuples. Each *key* uniquely identifies a code item. The corresponding *version* indicates if the code item is old or new (the higher the version, the newer the code). A CCMP allows any node to discover if there is a *(key, version)* tuple in the network with the same key but a different version compared to its own tuple. A natural approach is to allow a node to keep querying its neighbors, to find if there is a new code item by comparing its own tuple to its neighbors tuples. If there are only two nodes and one code item, a node can simply broadcast it own *(key, version)* tuple periodically. In a realistic scenario involving hundreds of dynamic nodes and hundreds of code items, a node should ask *smart questions* at the *right time* to discover if it needs an update. This requires careful CCMP design to reduce code update latency and to conserve both energy and bandwidth.

Previous CCMP protocols like DRIP [1] and DIP [2] incur high latency, high cost and may be complicated. Both DRIP and DIP are built on top of Trickle [3], a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In DRIP, a node randomly broadcasts each of its own *(key, version)* tuples to its neighbors separately. DRIP scales linearly with the total number of items ($O(T)$). DIP improves the total number of messages and latency to $O(log(T))$ by searching for a different *(key, version)* using hashes of the *(key, version)* tuples. A common feature in both approaches is that they try to detect and identify differences of versions as a whole.

In this paper, we propose a new code consistency maintenance protocol called *DHV*. We observe that *if two versions are different*, *they often only differ in a* *few least significant bits of their version number rather than in all their bits.* Hence, it is not necessary to transmit and compare the whole version identifier in the network. DHV aims to detect and identify differences of version-levels for code items with the goal of transmitting much less data in the network compared to DRIP and DIP. The name DHV comes from the three steps in the protocol — Difference detection, Horizontal search, and Vertical search. Each step requires only $O(1)$ total messages and latency with respect to the total number of items. So DHV can identify a difference with as few as 3 transmissions.

The contributions of this paper are as follows:

- We design and implement DHV, a code consistency maintenance protocol for wireless sensor networks (Sections 3, 4 and 5). DHV is simple, intuitive, and easy to adapt to current systems. DHV does a two dimensional search to identify exactly where the different *(key, version)* tuples are. DHV can detect and identify differences in $O(1)$.
- We evaluate DHV using simulations and real-world experiments on a MicaZ testbed, comparing it to state-of-the-art CCMP protocols (Section 6). Our results show that DHV reduces the number of messages by 50%, converges in half the time, and reduces the number of bits transmitted by 40-60%. Potentially, DHV can conserve significant energy compared to DIP, a state-of-the-art CCMP protocol.

2 Related Work

Code consistency management protocols can be thought of as a complement to data (code) dissemination protocols because a CCMP helps to efficiently discover when a node needs updates. Several dissemination protocols like Deluge [4], Sprinkler [5], and MNP [6] were developed to reprogram the whole network by disseminating new binaries. Maté [7] and Tenet [8] break a program into small code capsules or virtual programs and disseminate them to reprogram the sensors. Finally, DRIP [1], DIP [2], and Marionette [9] disseminate configuration parameters that can change sensor tasks.

Early attempts tried to adapt epidemic algorithms [10] to disseminate code updates during specific reprogramming periods. But there is no way for new nodes to discover past updates. If a node is not updated during the reprogramming period, it will never get updated. To discover if a node needs an update, a natural approach is to query or advertise its information periodically. The network as a whole may transmit an excessive and unnecessary number of query and advertisement messages. To address this problem, Levis *et al* [3] developed the Trickle protocol to allow nodes to suppress unnecessary transmissions. In Trickle, a node periodically broadcasts its versions but politely keeps quiet and increases the period if it hears several messages containing the same information as it has. When a difference is detected, the node resets the period to the lowest preset interval. Trickle scales well with the number of nodes and has successfully reduced the number of messages in the network.

However, the total messaging cost of a naive approach using Trickle scales linearly with the total number of code items, according to [2]. Lin *et al* [2] proposed DIP, a dissemination protocol that scales logarithmically with the total number of items. In DIP, a node periodically broadcasts a summary message, containing hashes of its keys and versions. The use of hashing helps detect if there is a difference in $O(1)$. But, once a difference is detected, DIP requires multiple iterations to hone in on the exact code items that have different version numbers. The search is analogous to a binary search in a sorted array. Therefore, DIP has $O(log(T))$ complexity in both the time and the number of messages required to identify an item that needs an update. DIP uses a bloom filter to further improve the search but it requires extra bytes to be included in every summary message. If there are N new items, then the total number of messages is $O(Nlog(T))$.

Protocol	Total cost	Latency
Scan Serial	$O(T)$	$O(T)$
Scan Parallel	$O(T)$	$O(1)$
DIP	$O(Nlog(T))$	$O(log(T))$
DHV	$O(N)$	$O(1)$

Table 1. Complexity Comparison of different CCMP protocols

Ideally, when there are N new items ($N < T$), we would like to transmit just enough information to identify these N items to update. Both Trickle and DIP transmit redundant information, $O(T)$ and $O(Nlog(T))$ respectively, to identify difference in version numbers. However, if two versions differ by even one bit in their binary representation, the two versions are different from each other. Based on this fact, we develop the DHV protocol, which can detect the differences in $O(1)$ complexity in both time and total number of transmitted messages. The total number of messages required to identify which items have newer versions is $O(N)$. [1] Table 1 shows a complexity comparison of different CCMP protocols.

3 Design Philosophy and Assumptions

3.1 Design Philosophy

Bit-level identification: Previous CCMPs have transmitted the complete version number for a code item. We observe that it may not always be necessary to do so. We treat the version number as a bit array, with the versions of all the code items representing a two dimensional bit array. DHV uses bit slicing to quickly zero in on the out of date code segment, resulting in fewer bits transmitted in the network.

[1] Strictly speaking, the DHV cost has one component with $O(T)$. However, the constant is very small and $O(T)$ is often less than 2 messages for T less than 256. Please refer to 4.1 for further details.

Statelessness: Keeping state in the network, particularly with mobility, is not scalable. DHV messages do not contain any state and usually small in size.
Preference of a large message over multiple small messages: To reduce energy consumption, it is better to transmit as much information possible in a single maximum length message rather than transmit multiple small messages. Sensor nodes turn off the radio when they are idle to conserve energy. Radio start-up and turn-off times (300 microseconds) are much longer than the time used to transmit one byte (30 microseconds) [11]. A long packet may affect the collision rate and packet loss. However, that effect only becomes noticeable under bursty data traffic conditions.

3.2 Assumptions

We assume that the order of code items in the set of *(key, version)* tuples is the same at all nodes, by using the same sorting algorithm at all nodes as the keys are often assigned at a base station and are unique. This assumption allows DHV to identify which items need updates from the indices of the different versions. In rare cases, nodes might have a different number of items, requiring that they compare every *(key, version)* tuple to identify which items are missing. We do not address these rare cases here.[2]

We also assume that the version number is incremented by only one for each update. This assumption is reasonable and implicitly made in both DRIP and DIP.[3] It allows us to infer that, if two version numbers are different, they mostly differ in a few least significant bits. DHV exploits this assumption to restrict the comparison scope to only a few least significant bits of the versions.

4 The DHV Protocol

DHV has two main phases — *detection* and *identification*. In detection, each node broadcasts a hash of all its versions called a *SUMMARY message*. Upon receiving a hash from a neighbor, a node compares it to its own hash. If they differ, there is at least one code item with a different version number.

In identification, the *horizontal search* and *vertical search* steps identify which versions differ. In horizontal search, a node broadcasts a checksum of all versions, called a *HSUM message*. Upon receiving a checksum from a neighbor, the node compares it to its own checksum to identify which bit indices differ and proceeds to the next step. In vertical search, the node broadcasts a bit slice, starting at the least significant bit of all versions, called a *VBIT message*. If the bit indices are similar, but the hashes differ, the node broadcasts a bit slice of index 0 and increases the bit index to find the different locations until the hashes are the same. Upon receiving a VBIT message, a node compares it to its own VBIT

[2] We plan to address these cases in the future as we integrate DHV with current dissemination protocols.

[3] In the implementations of DRIP and DIP, each time a code item is updated, its version number is incremented by 1.

to identify the locations corresponding to the differing *(key, version)* tuples. After identifying which *(key, version)* tuples differ, the node broadcasts these *(key, version)* tuples in a *VECTOR message*. Upon receiving a VECTOR message, a node compares it to its own *(key, version)* tuple to decide who has the newer version and if it should broadcast its DATA. A node with a newer version broadcasts its DATA to nodes with an older version.

Figure 1 illustrates the DHV protocol steps to complete a search to identify which vectors differ. DHV requires $O(1)$ in both latency and cost to identify all the differences, in contrast to DIP, which requires $O(log(T))$ in both latency and cost for searching.

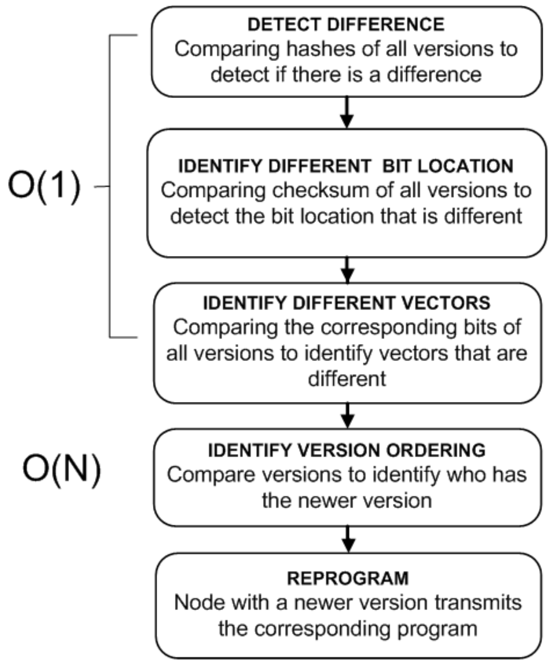

O(1)
O(N)
DETECT DIFFERENCE
Comparing hashes of all versions to detect if there is a difference
IDENTIFY DIFFERENT BIT LOCATION
Comparing checksum of all versions to detect the bit location that is different
IDENTIFY DIFFERENT VECTORS
Comparing the corresponding bits of all versions to identify vectors that are different
IDENTIFY VERSION ORDERING
Compare versions to identify who has the newer version
REPROGRAM
Node with a newer version transmits the corresponding program

Fig. 1. Main steps in the DHV protocol. DHV completes a search in only three steps, having 0(1) complexity in both latency and cost.

Figure 2 illustrates how DHV works. Node 1 and Node 2 have a set of keys and versions, in which key number 2 has different versions. Node 1 first broadcasts its SUMMARY hash of all the versions. Node 2 receives hash 1 and sees that hash 1 is different from hash 2 of node 2. Hence, node 2 broadcasts its HSUM message, checksum of all versions. Node 1 receives the HSUM message 2 and compares it to its own checksum. Node 1 identifies that the 2nd bits differ. Hence, node 1 copies the 2nd bit of all the versions into one or more VBIT messages and broadcasts them. Node 2 receives a VBIT message, compares it to its own VBIT message and detects that the 2nd bits differ. Hence, node 2 knows that the item

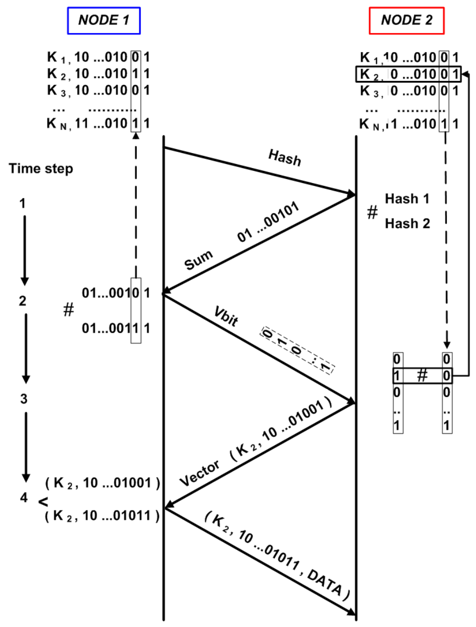


Fig. 2. DHV protocol example. Nodes identify a difference in only 3 steps.

with key index 2 is different from its neighbors. Node 2 broadcasts a VECTOR message containing *(key 2, version 2)*. Node 1 receives *(key 2, version 2)* from node 2 and sees that node 1 has a newer version of this item. Hence, node 1 broadcasts the DATA of key 2.

4.1 Message Formats

DHV uses 5 message types (Figure 3). One byte is used to indicate message type.
SUMMARY: This message contains the hash of all versions as well as the random seed used for hashing. It contains the least amount of information. A node can only detect if there is a difference or not using this message.
HSUM: This message contains the checksum of all versions, the hash of all versions as well as the random seed used for hashing. It is used to identify which bit indices differ.
VBIT: This message contains the corresponding bits of all versions and is used to identify which vectors differ. The bits corresponding to the differing indices of the VBIT messages are identified from the HSUM messages. If two HSUM messages are similar but the SUMMARY messages differ, the VBIT messages for the least significant bit indices are compared. This approach is suitable as the versions mostly differ in a few least significant bits. The VBIT size can be varied. DHV prefers large messages over multiple small messages. The VBIT message often has either the maximum packet length or the smallest size that can fit all the corresponding bits of all versions. The size of this message scales

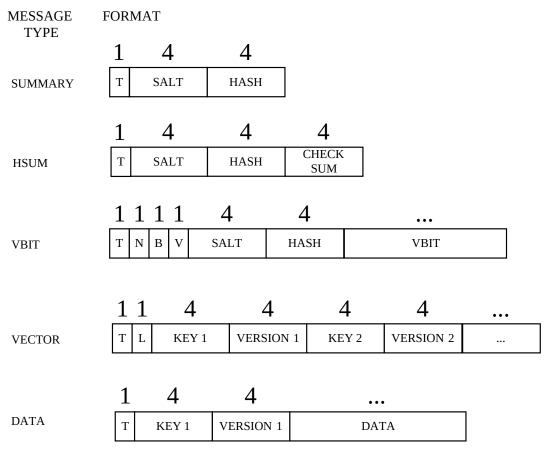


Fig. 3. DHV message format

linearly with the total number of code items. However, the constant factor is very small. Hence, DHV transmits only a few VBIT messages for searching.
VECTOR: This message contains one or more *(key, version)* tuples, and is used to identify who has newer versions to and send the code items.
DATA: This message contains the actual code items to be updated.

4.2 Suppression mechanism

Suppression mechanisms must be carefully designed to avoid flooding the network. As in Trickle, we develop our own suppression mechanism appropriate for DHV. A decision if to send out a message is made when a Trickle timer fires after a specified time interval. When the timer fires, if a node has DATA, VBIT, or HSUM messages to send, it will send one message out, selected by order of importance, DATA, VBIT, and HSUM messages. If there are no DATA, VBIT, HSUM messages to send and the node has heard two or more messages in the last interval, it decides to suppress its own messages. Otherwise, the node will send either the VECTOR or the SUMMARY message. If a SUMMARY message is sent, the next time interval is doubled as it expects that the network is stable.

5 Implementation

We have implemented DHV in TinyOS 2.1 and tested the protocol with MicaZ motes. DHV uses the Trickle timer [7] to control transmission suppression. Similar to DIP, both key and version in DHV are 4 bytes. The key and version sizes can be adjusted for the application. Table 2 compares DHV memory usage to DIP. They are largely similar, but DHV uses more RAM than DIP. We plan to carefully optimize the code implementation and believe that the RAM usage can be reduced significantly in subsequent versions.

Item	DIP	DHV
ROM (Byte)	20K	19K
RAM (Byte)	534	8739
Compiled code size (Byte)	63K	81K

Table 2. DHV Implementation Statistics

6 Experimental Design and Analysis

6.1 Goals and Metrics

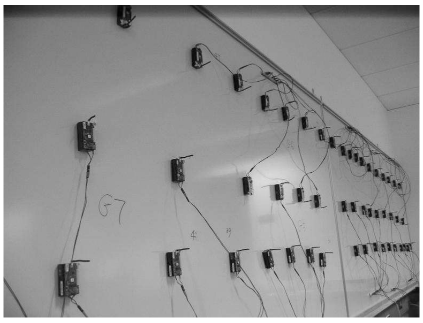
Previous work has shown that DIP outperforms other CCMP protocols [2]. Our experimental goals are to study if DHV performs better than the state-of-the-art DIP protocol over different scenarios. We use the following metrics to evaluate the performance, with a lower value indicating better performance for all metrics.

– Total latency to update new items. This metric indicates how fast a CCMP can help the network converge.
– Total numbers of transmitted messages and transmitted bytes to update new items. These metrics indirectly represent energy and bandwidth consumption.
– Total number of transmitted DATA messages. DATA can be as small as a few bytes or as big as several KBytes.
– Latency of the first transmitted DATA message. This metric indicates how quickly a new item is discovered.

6.2 Methodology

We conducted both simulations using TOSSIM [12], a discrete event simulator tool for wireless sensor networks, and experiments using our DHV implementation on a real MicaZ sensor network testbed (Figure 4 Left). TOSSIM does not allow simulation of packet loss directly. Instead, the packet loss depends on several parameters like receiving gain, noise, and clear channel access threshold. As a first step, we studied the effect of these parameters on packet loss. We simulated a two-node network. The noise is simulated using the state-of-the-art closest pattern matching approach with noise traces from the Stanford Meyer library, available in the TinyOS source code. Due to memory limitations, we only use the first 1000 entries in the trace, which is well above the minimum requirement of 100 entries. Figure 4 (Right) shows the packet loss rate versus receiving gain. The receiving gains correspond to packet loss rates of 5, 10, 15, 20, 25, 30, 35, 40, and 45% are -70, -74, -76, -78, -81, 84, -87,-88, -89dBM, respectively.

We conducted three sets of experiments. Two sets are simulation-based and one is on a real MicaZ sensor network. The first set studies how DHV performance is impacted by different parameters including the total number of items T, the total number of new items N, the packet loss rate L, and the density D. Density refers to the number of radio communication neighbors. We compare



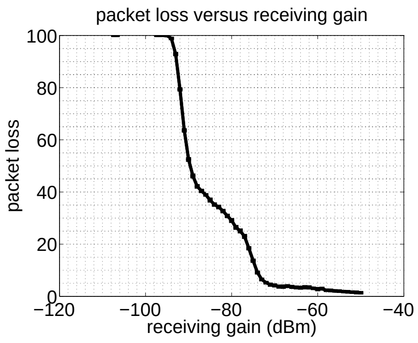


Fig. 4. (Left) Real MicaZ Testbed. (Right) Packet loss versus receiving gain using TOSSIM simulation.

DHV and DIP in a clique network. The default setting is $D = 32$ (nodes), $T = 64$ (keys), $N = 8$ (keys), $L = 5\%$. We vary D, T, N, and L and observe the total numbers of transmitted messages, transmitted bytes, and DATA messages, the true dissemination time, and the latency until the first data is sent out. Each experiment is repeated 10 times to account for randomness in timing. Like DIP, DHV uses 2 key/version tuples per VECTOR message to ensure comparability.

The second set includes two experiments with a medium and a tight density multi-hop network. The total number of nodes is 225. The topology and link configurations are extracted from example files in TOSSIM (15-15-medium-mica2-grid.txt and 15-15-tight-mica2-grid.txt in tossim/topologies directory). We observe the total number of transmitted messages and bytes required to complete updating as well as the update progress in terms of time.

Finally, we experimented with a real MicaZ sensor network testbed. We varied the number of sensor nodes and observed the total number of transmitted messages and total time required to complete updating the whole network. Unfortunately, the latest version of DIP on TinyOS 2.1 did not function properly on the MicaZ platform. Due to time constraints and lack of hardware, the TinyOS maintainers were unable to verify the problem. Hence, we can only evaluate DHV experimentally, and not compare it to DIP.

6.3 Experimental Results

DHV Performance versus Total Number of Items Figure 5 compares DHV and DIP when we vary the total number of items, T. DHV performance is relatively constant with T. Meanwhile, the cost and latency in DIP increases as T increases. As an example case, when $T = 64$, nodes using DHV transmit only about 40% of the total number of messages and complete reprogramming within 45% of the time taken by nodes using DIP. The discovery latency for the first new item in DHV is also a constant. Nodes using DHV also transmit only about 50% of the total number of DATA messages and less than 50% of the total number of bytes compared to nodes using DIP.

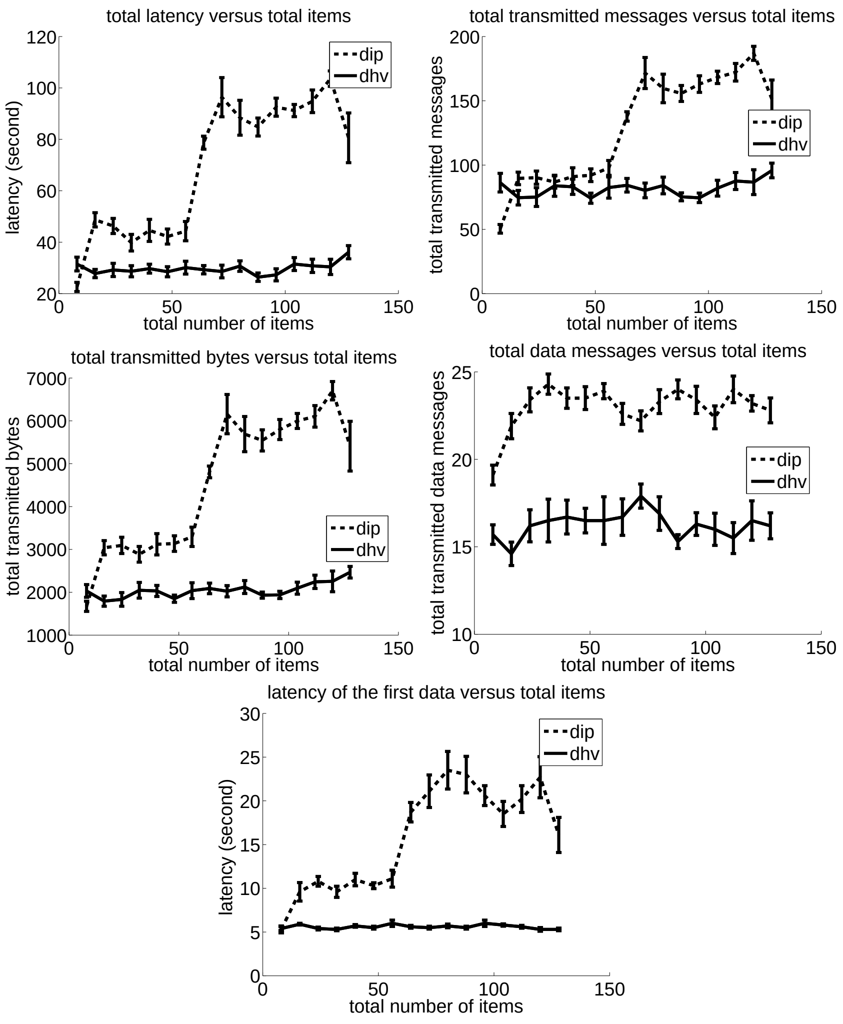


Fig. 5. Total latency versus total items: $D = 32$, $N = 8$, $L = 5\%$. T varies from 8 to 128.

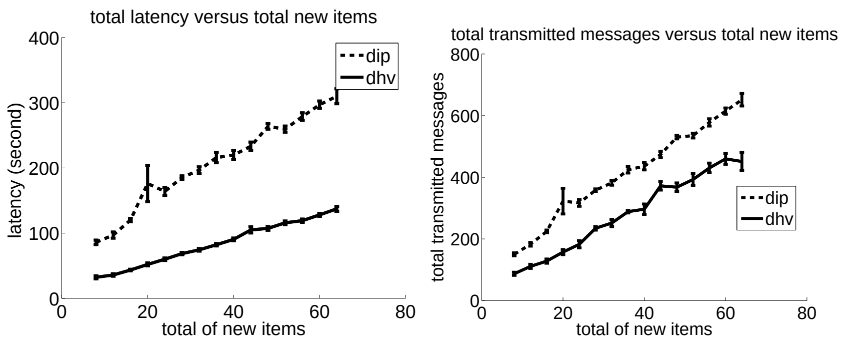


Fig. 6. Total latency versus total new items: $D = 32$, $T = 64$, $L = 5\%$. N varies from 8 to 64.

DHV Performance versus Total Number of New Items Figure 6 compares DHV and DIP when we vary the number of new items. DHV always uses fewer messages than DIP and uses only half the time to complete updating the network. For example, when $N = 32$, DHV uses about 70% of the messages of DIP and completes reprogramming in half the time.

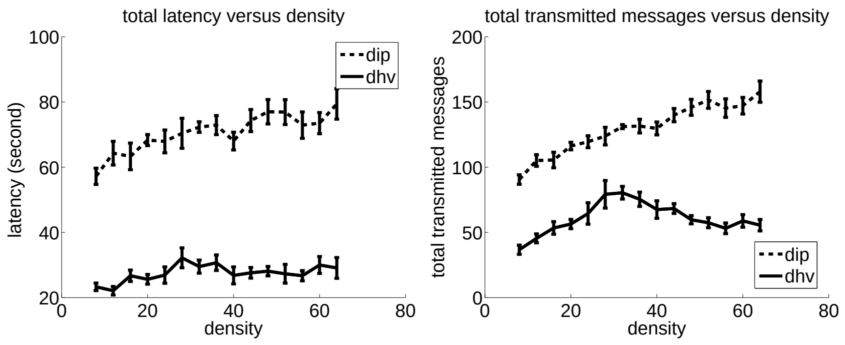


Fig. 7. Total latency versus network density: $T = 64$, $N = 8$, $L = 5\%$. D varies from 8 to 64.

DHV Performance versus Network Density Figure 7 compares DHV and DIP when we vary the density of the network. DHV completes updating twice faster and uses 50% fewer messages than DIP. Since, DHV uses smaller size messages than DIP except for the VBIT message, the total transmitted bytes in DHV are also much smaller than DIP.

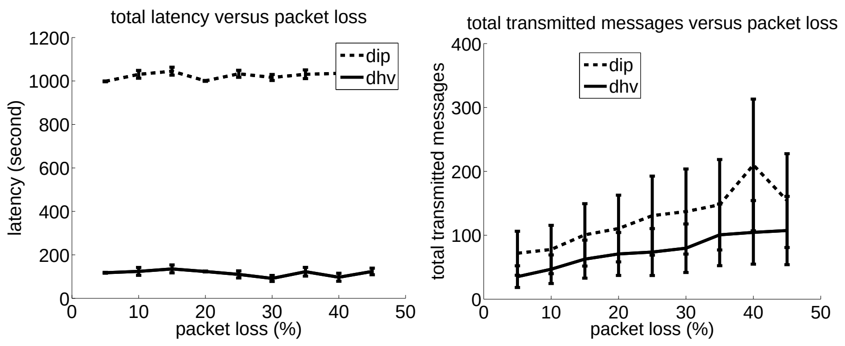


Fig. 8. Total latency versus network density: $D = 32$, $T = 64$, $N = 8$. Packet loss rate L varies from 5% to 45%.

DHV Performance versus Packet Loss Rate Figure 8 compares DHV and DIP when we vary the packet loss rate. DHV completely outperforms DIP in terms of latency. DHV completes updating ten times faster than DIP while using only half the number of messages. Unlike previous experiments, DHV and DIP use similar numbers of DATA messages. But both DHV and DIP exhibit a high performance variation, that increases with the packet loss rate.

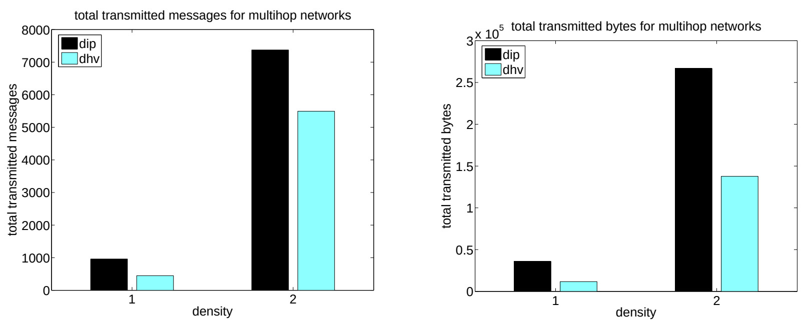


Fig. 9. Total transmitted messages for multi-hop networks.

DHV Performance in Multi-hop Networks Figure 9 shows the total number of transmitted messages and bytes to complete updating a multi-hop network using DHV and DIP. As expected, a denser network requires fewer total number of messages for updating data. In both medium and tight density networks, DHV uses about 60% to 70% number of messages and about 50% number of bytes compared to DIP. Figure 10 plots the update progress time versus the number of completed nodes for both DHV and DIP. In update completion time, DHV is at least two times faster than DIP in medium density networks and nearly

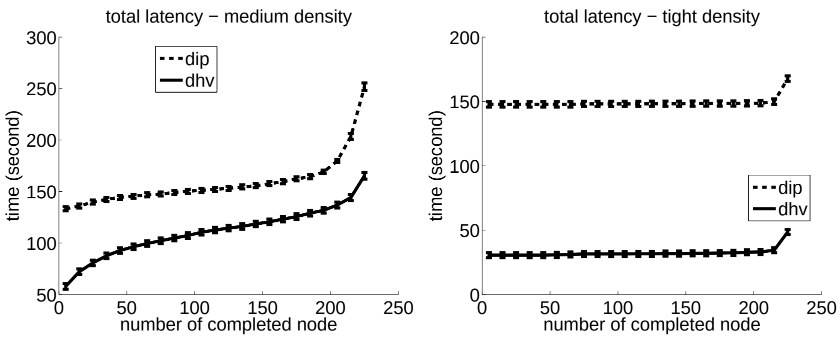


Fig. 10. Convergence time for multi-hop networks.

five times faster in tight density networks. In the medium density network, both DHV and DIP update completion times grow linearly with the completed nodes because in each transmission, only a few nodes can receive the messages. The update progresses from the node with the new items and spreads out to the whole network. Hence, the number of completed nodes grows linearly with time. In contrast, in the tight density network, when a node broadcasts, most other nodes receive the message. Hence, the network converges quickly. The inflection point when the number of completed nodes is around 200 can be explained by the fact that some nodes always receive messages with high noise. It takes much longer to complete updating these nodes.

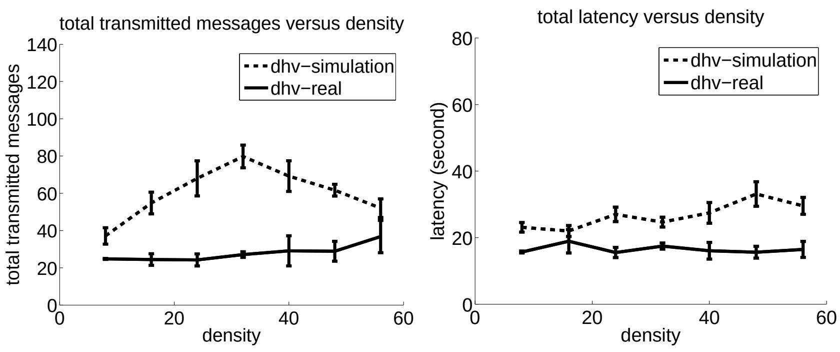


Fig. 11. Total transmitted messages versus network density: $T = 64$, $N = 8$, D is varied from 8 to 56 nodes.

DHV Performance on a real MicaZ test-bed Figure 11 shows the total number of transmitted messages and the update completion time on a real MicaZ network. There are total T=64 items and N=8 new items. The number of nodes is varied from 8 to 56 nodes. Surprisingly, DHV performs better in the real testbed than the simulation in section 6.3. DHV uses both fewer total messages and completes updating earlier in the real testbed than in the simulation. This might be explained by the fact that in the simulation the packet loss rate is 5% while the packet loss rate of the testbed is pretty low.

7 Discussion

Unlike dissemination protocols like Deluge or MNP [4, 6], which address the problem of how to disseminate code items in the network, code consistency management protocols address the problem of when to perform code updates. Therefore, it is important to ensure that the two protocols can be integrated well. The DHV protocol enables a dissemination protocol to decide when to begin transferring a new code item in the network so that the network can be updated quickly while still conserving energy. This decision making process can be independent from the operation of the dissemination protocol. Hence, the DHV protocol should work well with the existing dissemination protocols. We plan to integrate the DHV protocol with Deluge in the next source code release. Sensor networks are becoming more heterogeneous with devices based on different hardware/software platforms, having different numbers of code items. Ensuring code consistency for such networks is challenging, and a future research topic.

8 Conclusion

We have proposed and evaluated the DHV protocol to maintain code consistency in wireless sensor networks. The key innovation in DHV is that it reduces the number of transmitted bytes in the network by carefully selecting and transmitting only absolutely necessary information at the bit level to detect and identify which code items need updates. Together with a carefully designed suppression mechanism, DHV is able to reduce the total number of messages significantly. Theoretically, DHV can identify differences with $O(1)$ complexity in the total number of items instead of logarithmically compared to DIP. Simulations and real-world experiments validate that DHV performs at least twice better than the state-of-the-art DIP protocol. We believe that DHV can not only be used in wireless sensor networks but also in other distributed applications that require data consistency. The preliminary version of the DHV source code can be downloaded at `http://sys.cs.pdx.edu/home/dhv`.

Acknowledgments — This research was supported in part by National Science Foundation grants 0514818 and 0722063. Any opinions, findings, conclusions or recommendations it contains are those of the authors, and do not necessarily reflect the views of the Foundation.

References

1. Tolle, G., Culler, D.: Design of an application-cooperative management system for wireless sensor networks. In: Proceedings of the 2nd European Workshop on Wireless Sensor Networks (EWSN 2005), Istanbul, Turkey (2005)
2. Lin, K., Levis, P.: Data discovery and dissemination with dip. In: Proceedings of the 2008 International Conference on Information Processing in Sensor Networks (IPSN 2008), Washington, DC, USA, IEEE Computer Society (2008) 433–444
3. Levis, P., Patel, N., Culler, D., Shenker, S.: 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 (NSDI 04), Berkeley, CA, USA, USENIX Association (2004) 2–2
4. Hui, J.W., Culler, D.: 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), New York, NY, USA, ACM (2004) 81–94
5. Naik, V., Arora, A., Sinha, P., Zhang, H.: Sprinkler: A reliable and energy efficient data dissemination service for wireless embedded devices. In: Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS 05), Washington, DC, USA, IEEE Computer Society (2005) 277–286
6. Kulkarni, S.S., Wang, L.: Mnp: Multihop network reprogramming service for sensor networks. In: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS 05), Washington, DC, USA, IEEE Computer Society (2005) 7–16
7. Levis, P., Gay, D., Culler, D.: Active sensor networks. In: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation (NSDI 05), Berkeley, CA, USA, USENIX Association (2005) 343–356
8. Gnawali, O., Jang, K.Y., Paek, J., Vieira, M., Govindan, R., Greenstein, B., Joki, A., Estrin, D., Kohler, E.: The tenet architecture for tiered sensor networks. In: Proceedings of the 4th international conference on Embedded networked sensor systems (SenSys 06), New York, NY, USA, ACM (2006) 153–166
9. Whitehouse, K., Tolle, G., Taneja, J., Sharp, C., Kim, S., Jeong, J., Hui, J., Dutta, P., Culler, D.: Marionette: using rpc for interactive development and debugging of wireless embedded networks. In: Proceedings of the fifth international conference on Information processing in sensor networks (IPSN 06), New York, NY, USA, ACM (2006) 416–423
10. Akdere, M., Çagatay Bilgin, C., Gerdaneri, O., Korpeoglu, I., Ulusoy, Ö., Çetintemel, U.: A comparison of epidemic algorithms in wireless sensor networks. Computer Communications **29**(13-14) (2006) 2450–2457
11. Kramer, M., Geraldy, A.: Energy measurements for micaz node. In: Technical Report, Technical University Kaisers Lautern,GI/ITG KuVS, (2006). 1–7
12. Levis, P., Lee, N., Welsh, M., Culler, D.: Tossim: accurate and scalable simulation of entire tinyos applications. In: Proceedings of the 1st international conference on Embedded networked sensor systems (Sensys 03), New York, NY, USA, ACM (2003) 126–137