Policy-Based Reprogramming for Wireless Sensor Networks

Sang Hoon Lee, Lynn Choi
School of Electrical Engineering
Korea University
Seoul, Korea
{smile97, lchoi}@korea.ac.kr

Yunmook Nah
Department of Electronics and Computer Engineering
Dankook University
Yongin, Korea
ymnah@dku.edu

Seungki Hong, Jong-Arm Jun
IT Convergence Technology Research Laboratory
ETRI
Daejeon, Korea
{skhong, jajun}@etri.re.kr

Abstract—Program codes running on the sensor nodes need to be updated from time to time to fix bugs, tune the parameters, insert new functions, or delete useless codes. In this paper, we propose a policy-based reprogramming framework that can optimize the reprogramming process based on the execution characteristics of modules that constitute the WSN applications. To accomplish this we model the characteristics of a module as a tuple of the following four criteria; region, residence, synchronization and real-time properties of the module. According to the class of a module and the type of a reprogramming operation, we adaptively use 3 different reprogramming techniques; efficient code dissemination using connected dominating set, selective code dissemination, and code acquisition. Experimentation results confirm that this policy-based reprogramming can achieve substantial improvement in both the energy consumption and the reprogramming latency compared to existing solutions for various reprogramming scenarios.

Keywords—sensor network, reprogramming, policy, network management

I. INTRODUCTION

In wireless sensor networks (WSN), sensor nodes are designed to operate unattended for a long period of time. However, after the deployment, program codes running on the sensor nodes need to be updated from time to time to fix bugs, tune the parameters, insert new functions, or delete useless codes. The process of these program updates is called the reprogramming. WSN designers must provide the sensor network administrators with the reprogramming capability to handle the program updates as needed. Manual reprogramming is the most primitive approach but may not be a feasible solution for a large-scale network since it is labor intensive and time-consuming. Even for a small network, nodes may not be physically accessible since sensors are often deployed in a hostile environment such as disaster zone, biohazard area, or deep canyons.

Network reprogramming, which manages the reprogramming process over wireless or wired communication media, can overcome the problems of the manual reprogramming since it does not require physically accessing the nodes. The network reprogramming can be classified into code dissemination and code acquisition depending on who initiates the reprogramming process. In the code dissemination, a program image or a source code is disseminated by a system administrator. All the nodes in a network have the same version of a program code. In contrast, the code acquisition is initiated by individual sensor nodes. Each sensor node fetches and installs program codes through the network dynamically. Since each node acquires necessary codes on demand, program codes running on a particular node may be different from the codes in other nodes.

TinyOS [5] is one of the most popular platforms used for WSN. Since TinyOS [5] requires all the program codes to be integrated into a single executable image, most of the existing reprogramming schemes [7, 9, 12] assume that all the nodes in a network have the same image. Therefore, these schemes simply use only the code dissemination approach. However, in real-life scenarios, the role of each sensor may be different from each other. Thus, each node may not require the entire code or may run different program codes. For example, only a subset of nodes may act as a cluster head [20], virtual sink [1], disseminating node [23], or location server [10]. Also, each sensor may have different sensors in a heterogeneous sensor network. Under these circumstances, the code acquisition approach might be more efficient. However, the single image which contains the entire program codes will possess excessive memory space and increase the communication overhead for delivering the new version of the image. Moreover, it is hard to deliver and reprogram only the difference from the previous version since applications and TinyOS are statically compiled and integrated into a single image for an efficient execution.

The module-based reprogramming [6, 15, 18, 25] addresses those limitations of the single image environment. It allows each node to run different program codes. And, the reprogramming can be performed on a module-by-module basis on demand.

In this paper we further extend the module-based reprogramming to improve both the energy efficiency and the reprogramming performance. To accomplish this, we use different reprogramming policies based on the execution characteristics of each module and the type of reprogramming operation. We characterize the execution patterns of a program module by using the following

...
classification criteria: region, residence, synchronization, and real-time properties of the module. First, we classify modules into global modules and local modules according to the region of the nodes where the module is distributed. Global modules run on all the sensor nodes while local modules run only on a subset of the nodes. Second, we classify modules into static modules and dynamic modules according to the residence of the module. Static modules always run during the entire lifetime of nodes. A sensing module is a typical example. Dynamic modules are acquired dynamically and run only when the pre-specified conditions are satisfied. Third, we classify modules into synchronous modules and asynchronous modules. Synchronous modules run simultaneously by multiple nodes while asynchronous modules run by each node asynchronously at different times. Finally, we classify modules into real-time modules and non-real-time modules. Real-time modules have hard or soft deadlines while non-real-time modules may not have such deadlines. Based on these execution characteristics of each module, the target nodes and the timing of the reprogramming operation are determined.

In addition, we classify reprogramming tasks into three types of operations: inserting a module, deleting a module and updating a module. Each operation type has different network traffic requirement. On an insert, the entire program module has to be transmitted over the network. On an update, the difference from the previous version needs to be transmitted for an incremental update. And a delete operation can be performed by transmitting a small control message.

Based on the characteristics of the module and the type of reprogramming operation, we adaptively use three different reprogramming policies: code dissemination using connected dominating set [3, 4], selective code dissemination, and code acquisition. We call our policy-based reprogramming scheme PBR in the following discussion.

To evaluate the energy and network performance of PBR, we have modeled all the proposed reprogramming techniques of PBR on NS-2 and measured the traffic, delay, and the energy consumption. For comparative evaluation, we have also modeled two existing reprogramming solutions called Trickle [15] and LACONIC [6]. Simulation results show that PBR reduces unnecessary traffic by dynamically selecting a reprogramming policy based on the module class and the type of operation for various reprogramming scenarios while the existing schemes use techniques that are partially optimized for specific scenarios. According to the results, the proposed scheme can reduce the total dissipated energy by up to 83.87% and can complete the reprogramming operation as much as 11.6 times faster for a code acquisition scenarios compared to Trickle. In addition, PBR reduces the total energy consumption by up to 33.6% and also can reduce the reprogramming delay by as much as 29.1% for a selective code dissemination scenario.

The rest of this paper is organized as follows. Section II summarizes the related works. Section III discusses the module classification and the related classification criteria. Section IV presents the reprogramming policies and their algorithms. Section V discusses the energy and network performance of our policy-based reprogramming techniques compared to existing reprogramming solutions. Section VI concludes the paper.

II. RELATED WORKS

Early network reprogramming studies [7, 12, 15] for wireless sensor networks assumed that all the sensors execute the same code. They all focus on reducing the redundant code updates each time a new version of code is disseminated. Deluge [7] uses the advertise-request-data handshaking which was introduced by SPIN [11]. Each node periodically advertises the metadata which contains the version information of its codes. If a node recognizes the existence of a new code from the received metadata, it sends a request message. Then, the sender of the metadata broadcasts a new code. Since only the nodes that have received a request message broadcast a new code, Deluge can reduce the number of such broadcast packets. In addition, Deluge divides a code image into the equally-sized pages and pipelines the transfer of the pages to increase the network throughput. To guarantee that there are no missing packets, Deluge uses a selective negative acknowledgment packet with a bit vector that specifies which packets in a page are lost.

MNP [12] extends Deluge and proposes an additional technique called the sender selection to reduce the sender redundancy. If all the nodes, which receive a request for a new code, broadcast a code packet, the sender redundancy will be increased. Therefore, in MNP each node advertises the number of requesters so that the node which has the most requesters becomes a new sender of a code.

Trickle [15] proposes a more advanced code dissemination technique assuming script-based codes for a virtual machine called Mate [14]. Since script codes will be interpreted dynamically, it can deliver only the modified codes rather than the entire code, reducing the communication overhead. In addition, Trickle also reduces the number of metadata transmissions. Like the previous schemes [7, 12], each node periodically broadcasts a metadata to advertise a new code. However, the periodic transmission wastes energy when there is no code update. To reduce unnecessary updates, Trickle introduces a scheme called the polite gossip where a node can request no more update request to its neighbor if it receives the same metadata from the same neighbor multiple times.

Like Trickle, Melete [25] and LACONIC [6] use the same virtual machine assuming script-based codes. However, unlike the previous schemes Melete and LACONIC are based on the module-based reprogramming approach where each module can be updated independently. Therefore, they assume that different sensor nodes may run different application codes. If an application runs only on a subset of nodes, the broadcasting approach used in the previous schemes increases the number of unnecessary transmissions. Therefore, Melete introduces a group-based reprogramming which can update a specific node or a group of nodes, reducing the communication overhead. In addition, Melete can limit the range of code dissemination by introducing a technique called multi-hop code dissemination with a hop
count specification. Melete also introduces two additional techniques called lazy forwarding and progressive forwarding to enhance the reprogramming performance. The lazy forwarding allows a neighbor node of a requester to respond if it has the requested code while the progressive flooding limits the flooding region of the code request message.

LACONIC [6] is an improved version of Melete. LACONIC exploits the application calling history and code dissemination history. The application calling history is used to find the predecessor code and the code dissemination history is used to find the previous code in the forwarding path of the predecessor code. With these two history information, a node can speed up the search for a new code by sending the request message to the previous path.

Existing reprogramming schemes are partially optimized for a specific application type. They either use broadcast-based code dissemination [7, 12, 15] approach assuming that an identical monolithic code will be run by all the nodes or use module-based code acquisition approach [6, 25] assuming that each node may run a different code. PBR is different from the existing schemes in that by pre-analyzing the module characteristics we can selectively employ an optimal reprogramming procedure depending on the module class and the type of reprogramming operation.

III. MODULE CLASSIFICATION

In this work we define an application as a set of modules which consists of a set of functions. Even if modules are parts of the same application, each module may have different execution characteristics. In module-based reprogramming, a module is a basic reprogramming unit. If applications are developed with a script language such as TinyScript [22], a set of script codes can be a reprogramming unit.

A. Classification Criteria

Various execution properties of a module can be used to classify modules: the region of nodes where this module is distributed, the residence of the module, the synchronization property of the module execution, the real-time property of the module, the dependency relationship with other modules, and the privilege level of the module in the system such as user-level modules or kernel-level modules. Among these criteria, we use the region, residence, synchronization, and real-time properties of a module for classification since these criteria are the major factors that can determine the traffic volume and pattern of the reprogramming process.

The execution region: First, we classify modules into global modules and local modules according to the region. Modules in the class global are executed on all the nodes in the network. This class includes code modules for MAC and routing protocols, task management modules, device drivers, and sensing modules used in homogeneous sensor networks. Modules in the class local are executed on a subset of nodes. Each node may run a different module based on its role. A module for a cluster head or a dominator in a connected dominating set is a good example of this class. In heterogeneous sensor networks, the sensing module may depend on the type of sensors attached to each node.

The execution residence: Second, we classify modules into static modules and dynamic modules according the execution residence. Static modules are permanently resident during the node lifetime. A sensing module is a typical example. Most of sensor networks assumed that a sensing module is always active. Modules in class dynamic must be acquired dynamically when the pre-specified conditions are satisfied. Conditions for installing a dynamic module have to be specified by the network administrator. This dynamic class includes code modules for routing topology repair for node failures [1], tree setup for wakeup scheduling [17], etc.

The execution synchronization: Third, we classify modules into synchronous and asynchronous modules depending on whether the module will be run by multiple nodes simultaneously or independently. Synchronous modules run at the same time simultaneously while asynchronous modules run asynchronously at different times. Modules for packet flooding and clock synchronization module might be examples of the synchronous class.

The real-time execution: Finally, modules can be divided into real time modules and non-real time modules. The execution of the real time modules must be completed before a specified deadline. To meet the deadline, real time modules may need to be classified as static module or if a real-time module is classified as a dynamic module, the location of this module should be close enough to complete the dynamic loading in time.

B. Reprogramming Operation Classification

We classify reprogramming tasks into three types of operations: inserting a new module, deleting an old module and updating an existing module. Each operation type may have different network traffic requirement.

On an insert, the entire module has to be transmitted over the network. Since sensor nodes may not know the existence of a new module, the administrator should also distribute the metadata about the module through a sink. Metadata includes the version number, the module class, and the code size, etc.

On an update, only the difference from the previous version needs to be transmitted for an incremental update. If WSN assumes a small executable image, an administrator can deliver only the image difference [9]. If WSN assumes a module-based program with script-based codes such as TinyScript [22], an administrator may deliver only the updated script codes. Then, each node may interpret the codes to run the new codes.

Delete operation can be performed by transmitting a small control message containing a program module ID. Since a sink may not know the list of the installed program modules in each node, it cannot send the message directly to a specific node. Therefore, a sink has to broadcast the message when it wants to delete a particular module from the network.
We call this process policy-based reprogramming (PBR).

Reprogramming policy according to the module classification. A network administrator classifies program modules and selects an optimal one on the characteristics of a module. For example, an operating system may be composed of a task management module, a memory management module, an event handler module, and device driver modules. Therefore, an operating system may have various execution characteristics.

### IV. Reprogramming Policies

An optimal reprogramming process may vary depending on the characteristics of a module. A network administrator classifies program modules and selects an optimal reprogramming policy according to the module classification. We call this process policy-based reprogramming (PBR).

Before proposing the policies, we will explore basic reprogramming techniques.

#### A. Basic Techniques for Reprogramming

As discussed in Section I, the reprogramming processes are largely classified into code dissemination and code acquisition.

**CDS-based code dissemination:** We use the flooding-based code dissemination algorithm to reprogram global modules. However, a simple flooding scheme requires all of the nodes to broadcast at least once, which incurs excessive data and sender redundancies. These redundancies may increase the energy consumption and degrade the network performance due to their excessive transmissions and collisions. To address this problem, various efficient flooding algorithms have been proposed [2, 8, 16]. The number of broadcast packets can be minimized by using a minimum connected dominating set [3, 4]. However, finding the minimum connected dominating set is known as NP-complete. Instead we use a greedy algorithm [13] that can efficiently approximate the minimum connected dominating set. We also use the pipelining scheme proposed in Deluge to accelerate the code dissemination process. Since code dissemination reprograms the entire network, we do not need to propagate metadata at all.

**Selective code dissemination:** We use a selective dissemination to reprogram local modules. For this purpose we use the on-demand path setup technique similar to the one used in AODV [19]. After receiving the metadata from a sink, a node sends a request message to the sink if it needs a new version. When the sink disseminates the module, only the nodes who have delivered the request message participate in forwarding the new module, reducing unnecessary packet transmissions.

**Metadata dissemination:** Unlike the code dissemination, without version information it is difficult for a sensor node to decide when it needs to initiate the on-demand code acquisition process. For this purpose, the previous schemes [7, 12] assume that each node periodically broadcasts its metadata. However, if a reprogramming is rarely carried out, this metadata transmission increases energy consumption and wastes network bandwidth. In our scheme, only a sink node floods metadata once whenever a new code is announced. And, each node maintains the version information of all the modules. For an efficient flooding, our scheme uses an efficient broadcast based on a connected dominating set where only the nodes who are the members of the connected dominating set broadcasts a packet.

**Code acquisition:** For the on-demand code acquisition, we use a two-way request-reply handshaking. Since each node knows the version information through the metadata from a sink, it sends the request message to the sink when a new code is needed. If an intermediate node on the path to the sink has the latest version of the requested code, the node sends the code to the requester instead of forwarding the request to a sink [5]. When a new node joins the network, it may carry out the on-demand reprogramming process. To get the version information, this node sends a metadata request message to one of its neighbors. If this new node decides

<table>
<thead>
<tr>
<th>Region Class</th>
<th>Module Class</th>
<th>Real-time Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>TGlobal</td>
<td>RGlobal</td>
</tr>
<tr>
<td>Local</td>
<td>TLocal</td>
<td>RLocal</td>
</tr>
<tr>
<td>Static</td>
<td>TStatic</td>
<td>DNon-real</td>
</tr>
<tr>
<td>Dynamic</td>
<td>TDynamic.Sync</td>
<td>DReal</td>
</tr>
<tr>
<td>Synchronous</td>
<td>TStatic</td>
<td>DNon-real</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>TDynamic_Async</td>
<td>DNon-real</td>
</tr>
</tbody>
</table>

### C. Module Classification

Table I shows our module classification parameters and their corresponding notations. We classify a module using a tuple notation, \( \{ \text{region class, residence class, real-time class} \} \). From the reprogramming purpose, static modules can be considered synchronous since they are permanently resident. Therefore, we further classify only dynamic modules into synchronous and asynchronous modules. Table II classifies various WSN modules using the classification criteria. Since an application code is composed of multiple modules, a WSN application may exhibit different execution characteristics. For example, an operating system may be composed of a task management module, a memory management module, an event handler module, and device driver modules. Therefore, an operating system may have various execution characteristics.

### Module Classification Parameters

<table>
<thead>
<tr>
<th>Region Class</th>
<th>Module Class</th>
<th>Real-time Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>TGlobal</td>
<td>RGlobal</td>
</tr>
<tr>
<td>Local</td>
<td>TLocal</td>
<td>RLocal</td>
</tr>
<tr>
<td>Static</td>
<td>TStatic</td>
<td>DNon-real</td>
</tr>
<tr>
<td>Dynamic</td>
<td>TDynamic.Sync</td>
<td>DReal</td>
</tr>
<tr>
<td>Synchronous</td>
<td>TStatic</td>
<td>DNon-real</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>TDynamic_Async</td>
<td>DNon-real</td>
</tr>
</tbody>
</table>

### Module Classification Examples

<table>
<thead>
<tr>
<th>Modules</th>
<th>Region Class</th>
<th>Module Class</th>
<th>Real-time Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensing in a homogeneous network</td>
<td>{ RGlobal, TStatic, DReal }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sensing in a heterogeneous network</td>
<td>{ RLocal, TStatic, DReal }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task management</td>
<td>{ RGlobal, TStatic, DReal }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory management</td>
<td>{ RGlobal, TStatic, DNon-real }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Event handler</td>
<td>{ RLocal, TDynamic_Async, DReal }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Device driver</td>
<td>{ RGlobal, TStatic, DNon-real }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Networking protocol</td>
<td>{ RLocal, TStatic, DNon-real }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reprogramming module</td>
<td>{ RGlobal, TDynamic_Sync, DNon-real }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual machine</td>
<td>{ RGlobal, TDynamic_Async, DNon-real }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reporting module for fire alarm</td>
<td>{ RGlobal, TDynamic_Async, DReal }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cluster header module</td>
<td>{ RLocal, TStatic, DNon-real }</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CDS-based code dissemination: We use the flooding-based code dissemination algorithm to reprogram global modules. However, a simple flooding scheme requires all the nodes to broadcast at least once, which incurs excessive data and sender redundancies. These redundancies may increase the energy consumption and degrade the network performance due to their excessive transmissions and collisions. To address this problem, various efficient flooding algorithms have been proposed [2, 8, 16]. The number of broadcast packets can be minimized by using a minimum connected dominating set [3, 4]. However, finding the minimum connected dominating set is known as NP-complete. Instead we use a greedy algorithm [13] that can efficiently approximate the minimum connected dominating set. We also use the pipelining scheme proposed in Deluge to accelerate the code dissemination process. Since code dissemination reprograms the entire network, we do not need to propagate metadata at all.

Selective code dissemination: We use a selective dissemination to reprogram local modules. For this purpose we use the on-demand path setup technique similar to the one used in AODV [19]. After receiving the metadata from a sink, a node sends a request message to the sink if it needs a new version. When the sink disseminates the module, only the nodes who have delivered the request message participate in forwarding the new module, reducing unnecessary packet transmissions.

Metadata dissemination: Unlike the code dissemination, without version information it is difficult for a sensor node to decide when it needs to initiate the on-demand code acquisition process. For this purpose, the previous schemes [7, 12] assume that each node periodically broadcasts its metadata. However, if a reprogramming is rarely carried out, this metadata transmission increases energy consumption and wastes network bandwidth. In our scheme, only a sink node floods metadata once whenever a new code is announced. And, each node maintains the version information of all the modules. For an efficient flooding, our scheme uses an efficient broadcast based on a connected dominating set where only the nodes who are the members of the connected dominating set broadcasts a packet.

Code acquisition: For the on-demand code acquisition, we use a two-way request-reply handshaking. Since each node knows the version information through the metadata from a sink, it sends the request message to the sink when a new code is needed. If an intermediate node on the path to the sink has the latest version of the requested code, the node sends the code to the requester instead of forwarding the request to a sink [5]. When a new node joins the network, it may carry out the on-demand reprogramming process. To get the version information, this node sends a metadata request message to one of its neighbors. If this new node decides
reprogramming, it requests a sink to disseminate the latest code.

B. Additional Techniques for Further Optimization

The techniques presented in Section IV.A are basically improved versions of the existing techniques [6, 15, 19, 25]. To further enhance the energy and network performance, we introduce the following additional techniques.

Early update: During the on-demand code acquisition process, intermediate nodes between the requestor and a sink may have an older version of the transmitted code. Instead of waiting for a future reprogramming, these intermediate nodes also update their modules in advance, which can save the energy and may improve the performance by eliminating future code acquisitions.

Code caching for real time modules: If a module is in the class \{RLocal, TDynamic, DReal\}, each node may not have the latest code for the module since it is in the class ‘local’ and ‘dynamic’. In this situation, the code acquisition process must be completed fast enough to meet the deadline. To accelerate the reprogramming process we use code caching for real time modules. Assume that the average per-hop delay is \(L\) and a caching node is \(N\) hops away from the requestor. Then, the time to deliver the first packet of the requested code is at least \(2LN\). If the cached module consists of \(M\) packets, each packet has to be buffered to avoid a collision with the previous transmission. In other words, a node can start transmitting a packet after the previous packet arrives at the third successor node. Since all the packets except the first one suffer from this additional delay, the it will take at least \(3L(M-1)\) for the last packet to arrive at the requestor. Therefore, delivering the cached module to the requestor will take \(2LN + 3L(M-1)\). If the time remaining to the specified deadline is \(D\), then the delivery delay has to be smaller than \(D\) and the maximum distance to the nearest caching node \(N_{MAX}\) has to be smaller than \(\frac{D-3L(M-1)}{2L}\). To deploy codes for real time modules to the ideal position, a sink node requires all the connectivity information of the network and massive processing overload. In this work, we use a time-to-live (TTL) value to approximate the selection of caching nodes. A sink sets a TTL in the actual code packet as \(N_{MAX} - 1\) and broadcasts the code. Each node, which receives the code with non-zero TTL, rebroadcasts the code after reducing the TTL by one. Then the TTL decreases as the hop distance from a sink node increases. If a node receives the code whose TTL is zero, it caches the code. The selected caching node will reset the TTL as \(N_{MAX} - 1\) and rebroadcasts the code to select other caching nodes. Finally, all the nodes have at least one caching node among \(N_{MAX} - 1\) hop neighbors.

C. Reprogramming Policies for Each Module Class

1) CDS-based code dissemination

Class \{RGlobal, TStatic, DReal or DNon-real\}: A module in the class TStatic has to be reprogrammed right after a new version of the module is announced. The most efficient reprogramming policy for this class is the pipelined code dissemination from a sink. Since a sink floods a new module without transmitting metadata, we can eliminate the communication overhead for control packets such as metadata and request messages.

Class \{RGlobal, TDynamic_Sync, DReal or DNon-real\}: If a module runs on all the nodes simultaneously, we can regard the execution of the module as a scheduled execution. For example, a clock synchronization process is carried out according to the pre-specified interval. Temperature sensing modules can be classified as this class if their report schedule is predetermined. An administrator will initiate the reprogramming right before the execution. Thus, we can avoid the transmission of unnecessary version updates if several updates are carried out between the executions.

2) Selective code dissemination

Class \{RLocal, TStatic, DReal or DNon-real\}: Similar to the class \{RGlobal, TStatic, DReal or DNon-real\}, modules in this class have to be reprogrammed when a new module is announced. Since only a subset of nodes requires a module in this class, we can use the selective code dissemination technique to reduce the traffic.

Class \{RLocal, TDynamic_Sync, DReal or DNon-real\}: Since we know the execution time of the modules in this class, we use the selective code dissemination. Although a module is included in the class RLocal, the selective code dissemination consumes more energy than the CDS-based code dissemination when most of the nodes run this module.

### TABLE III. REPROGRAMMING POLICIES FOR EACH MODULE CLASS

<table>
<thead>
<tr>
<th>Module class</th>
<th>Reprogramming policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>{RGlobal, TStatic, DReal}</td>
<td>CDS-based code dissemination</td>
</tr>
<tr>
<td>{RGlobal, TStatic, DNon-real}</td>
<td>CDS-based code dissemination</td>
</tr>
<tr>
<td>{RGlobal, TDynamic_Sync, DReal}</td>
<td>CDS-based code dissemination</td>
</tr>
<tr>
<td>{RGlobal, TDynamic_Sync, DNon-real}</td>
<td>CDS-based code dissemination</td>
</tr>
<tr>
<td>{RGlobal, TDynamic_Async, DReal}</td>
<td>Insert/update operation: On-demand code acquisition Delete operation: CDS-based control packet flooding</td>
</tr>
<tr>
<td>{RGlobal, TDynamic_Async, DNon-real}</td>
<td>Insert/update operation: On-demand code acquisition Delete operation: CDS-based control packet flooding</td>
</tr>
<tr>
<td>{RLocal, TStatic, DReal}</td>
<td>Selective code dissemination</td>
</tr>
<tr>
<td>{RLocal, TStatic, DNon-real}</td>
<td>Selective code dissemination</td>
</tr>
<tr>
<td>{RLocal, TDynamic_Sync, DReal}</td>
<td>Selective code dissemination</td>
</tr>
<tr>
<td>{RLocal, TDynamic_Sync, DNon-real}</td>
<td>Selective code dissemination</td>
</tr>
<tr>
<td>{RLocal, TDynamic_Async, DReal}</td>
<td>Insert/update operation: On-demand code acquisition Delete operation: CDS-based control packet flooding</td>
</tr>
<tr>
<td>{RLocal, TDynamic_Async, DNon-real}</td>
<td>Insert/update operation: On-demand code acquisition Delete operation: CDS-based control packet flooding</td>
</tr>
</tbody>
</table>

\(L\) and a caching node is \(N\) hops away from the requestor.
Assume that the number of nodes in a set is \( n \), the number of nodes in a connected dominating set is \( d \). Then the total transmission costs of these two techniques are the following:

- **CDS-based code dissemination**: \( dc \)
- **selective code dissemination**: \( dm + nh(r+c) \),

where \( c \) is the cost for transmitting the code, \( r \) is the cost for the request message, \( m \) is the cost for the metadata, and \( n \) is the average hop distance from a node in a set to a sink. If \( (dm + nh(r+c)) \) is bigger than \( dc \), we use the CDS-based code dissemination technique even though a module is included in the class \( R_{\text{Local}} \).

3) **Code acquisition**

- **Class \([R_{\text{Global}} \text{ or } R_{\text{Local}}, T_{\text{Dynamic Async}}, D_{\text{Real}}]\)**: Even if a module runs on all the sensor nodes, we will use the on-demand code acquisition since we assume that the module will run rarely and asynchronously. A module in this class has the pre-specified deadline. Therefore, a sink selectively disseminates the new version of the module to reduce the reprogramming delay. During the selective dissemination, nodes in the disseminating region will employ early updates if they also have the same module.

- **Class \([R_{\text{Global}} \text{ or } R_{\text{Local}}, T_{\text{Dynamic Async}}, D_{\text{Non-real}}]\)**: Similar to the previous class, the reprogramming process for this class is carried out through the on-demand code acquisition with the early update technique.

Table III lists the reprogramming policies for each module classes. Note that a sensor node uses the on-demand code acquisition only for insert and update operations. For a delete operation CDS-based flooding should be used since the delete operation is initiated by a sink by transmitting a small control message containing a program module ID.

V. EXPERIMENTATION AND RESULTS

To evaluate both the energy and the network performance of PBR, we have modeled all of our proposed reprogramming techniques using NS-2 simulator platform [21]. For comparative evaluation, we also model two representative existing reprogramming schemes known as Trickle [15] and LACONIC [6]. For this experimentation we assume that sensor nodes are stationary and randomly deployed. We also assume that a sink is located at the center of the sensor field. We use S-MAC [24] as an underlying MAC protocol. For each scenario, we measure the total dissipated energy and the delay for a single reprogramming operation.

In this experimentation we focus on three reprogramming policies, the CDS-based code dissemination, the selective code dissemination, and the on-demand code acquisition since any reprogramming scheme can be described as combinations of these three techniques. We generate reprogramming scenarios using three types of reprogramming operation; an insert, an update and a delete since each operation may have a different traffic pattern.

A. Experimentation Scenarios

To focus on the energy and delay performance of the proposed three policies, we have implemented only the communication part of the reprogramming scheme. We assume 10 Kbyte module which consists of 80 128-byte dummy packets. We also assume that each sensor node has 20 modules in its memory. We use two metrics: total dissipated energy and average operation delay. The total dissipated energy measures the total energy dissipated throughout the network for a single reprogramming operation. This metric shows the communication overhead required for a reprogramming operation. The average operation delay measures the average delay from the time when the first packet is transmitted to the time when the last code packet is received. We use Trickle and LACONIC as reference schemes for the comparative evaluation of PBR.

B. Experimentation Results

1) **CDS-based code dissemination**

   **Insert** Fig. 1 compares the energy and delay performance of PBR against Trickle and LACONIC. For this simulation we vary the number of sensor nodes from 100 nodes to 500 nodes assuming the same physical network dimension, thus changing the network density. Fig. 1(a) shows the total dissipated energy of each scheme. Since PBR uses the CDS-based flooding, the number of broadcast nodes remains stable regardless of the node density. However, the total energy consumption increases as the number of nodes increases, because the number of overhearing nodes is proportional to the node density. Likewise, the number of broadcast nodes does not increase in Trickle as the node density increases since only the nodes that have received a
request message broadcast the new code. However, Trickle consumes more energy than PBR since PBR does not propagate metadata before distributing a new module. As a result, PBR reduces the energy consumption by up to 10.4% compared to Trickle. Note that LACONIC consumes more energy than the other schemes. Since LACONIC is based on the on-demand code acquisition, it transmits more control packets during the reprogramming process. Although every node sends a code request message to a sink in LACONIC, the early reply technique used by LACONIC can reduce the number of such transmissions. PBR reduces the energy consumption by up to 31.8% compared to LACONIC. Fig. 1(b) shows the average delay of a reprogramming operation. As demonstrated in the figure, PBR has the best performance since it does not have to broadcast metadata. The difference between the results of Trickle and PBR is due to the metadata overhead. For this scenario LACONIC shows the longest delay since each node initiates the code acquisition process after receiving metadata. Therefore, PBR can reduce the reprogramming delay as much as by 47% compared to LACONIC.

**Update**

Fig. 2 compares the energy and delay performance of three schemes when a sink updates a module globally throughout the network. We assume that the difference from the previous version to the new one is 256 bytes. Similar to the insert operation scenario, we vary the number of sensor nodes from 100 nodes to 500 nodes. As shown in Fig. 2(a), the results show similar pattern as in the insert operation. For this scenario PBR reduces the energy consumption by up to 25.27% and 9.1% compared to LACONIC and Trickle respectively. According to the results, an update operation consumes 58 times lower energy than an insert operation even though an insert operation delivers 50 times bigger data than an update operation. The different ratio of the energy consumption to the payload size is due to transmission failures caused by collisions and transmission errors. Fig. 2(b) compares the operation delay of three schemes. Like the energy consumption results, the average delay of an update operation is 60.1 times smaller than that of an insert operation.

**Delete**

Since all the nodes run the identical module, a sink can carry out a delete operation by using the CDS-based control packet flooding. Table IV shows the energy and delay performance of a delete operation. The total number of nodes does not affect the operation delay since we use a CDS-based flooding scheme. However, the energy consumption increases since the number of receivers increases.

**2) Selective code dissemination**

**Insert**

Fig. 3 shows the energy and delay performance of selective reprogramming where reprogramming takes place only for a subset of nodes in the network. For this simulation we assume a fixed size 400-node network but vary the number of nodes in the subset. Nodes are randomly selected for the reprogramming. Fig. 3(a) shows the total dissipated

![Figure 2. The performance of an insert operation for selective reprogramming for Trickle, LACONIC, and PBR: (a) the total dissipated energy and (b) the average operation delay.](image2)

![Figure 3. The performance of an update operation for Trickle, LACONIC, and PBR: (a) the total dissipated energy and (c) the average operation delay](image3)

**TABLE IV. THE ENERGY AND DELAY PERFORMANCE OF A DELETE OPERATION.**

<table>
<thead>
<tr>
<th>Number of nodes</th>
<th>100</th>
<th>200</th>
<th>300</th>
<th>400</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total dissipated energy (J)</td>
<td>0.0155</td>
<td>0.0326</td>
<td>0.0489</td>
<td>0.0650</td>
<td>0.0772</td>
</tr>
<tr>
<td>Average operation delay (s)</td>
<td>9.5625</td>
<td>9.9375</td>
<td>9.6375</td>
<td>9.8125</td>
<td>9.915</td>
</tr>
</tbody>
</table>

![Figure 2](image2)

![Figure 3](image3)
energy and Fig. 3(b) shows the average operation delay. The total dissipated energy is generally proportional to the number of target nodes. Note that LACONIC consumes less energy than Trickle since Trickle cannot support multi-hop code dissemination. This makes all the nodes in Trickle to participate in the reprogramming process. Therefore, Trickle cannot exploit the execution region characteristic of a module and it consumes 58.8% more energy than PBR.

In addition, reprogramming in Trickle takes longer than two other schemes. This is due to the overhead of flooding-based code dissemination used by Trickle. The selective dissemination used in PBR and LACONIC can reduce unnecessary transmissions. By reducing the contention among the forwards the selective dissemination can accelerate the reprogramming process compared to the flooding-based code dissemination approach used in Trickle.

Update Fig. 4 shows the results for an update operation. Similar to the insert operation scenario, Trickle consumes 59.8% more energy than PBR when 100 nodes are selectively updated. According to the results shown in Fig. 3 (a) and Fig. 4 (a), the selective code dissemination consumes only 40% of the energy required for the CDS-based code dissemination.

Delete Since Trickle and LACONIC cannot selectively deliver the delete message to specific nodes, they have to use the CDS-based flooding scheme whose performance is shown in Table V. However, PBR can deliver the message to the nodes in a set by using the selective dissemination. Therefore, PBR can reduce the communication overhead for a delete operation as shown in Table V.

3) The on-demand code acquisition

Insert Fig. 5 shows the energy and delay performance of the on-demand code acquisition process for the three reprogramming schemes. 400 sensor nodes are used for the simulation. To evaluate the impact of code caching we vary the number of caching nodes for PBR. Caching nodes are randomly selected. Fig. 5(a) shows the total dissipated energy while Fig. 5(b) shows the average operation delay. Since Trickle uses a flooding-based algorithm, both the energy consumption and the reprogramming delay remain stable. Different from Trickle, LACONIC and PBS can reprogram a target node through the on-demand code acquisition. Therefore, these two schemes outperform Trickle in terms of both energy consumption and reprogramming delay. Since both schemes basically use the same approach, they show comparable performance. In addition, the results show that the operation delay steadily declines as the number of caching nodes increases.

<table>
<thead>
<tr>
<th>Number of nodes in a set</th>
<th>20</th>
<th>40</th>
<th>60</th>
<th>80</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total dissipated energy (J)</td>
<td>0.0149</td>
<td>0.0151</td>
<td>0.0153</td>
<td>0.0152</td>
<td>0.0154</td>
</tr>
<tr>
<td>Average operation delay (s)</td>
<td>0.1839</td>
<td>0.3875</td>
<td>0.5804</td>
<td>0.7726</td>
<td>0.9173</td>
</tr>
</tbody>
</table>

Figure 5. The performance of an update operation for selective reprogramming for Trickle, LACONIC, and PBR: (a) the total dissipated energy and (b) the average operation delay.

Figure 4. The performance of an insert operation for on-demand code acquisition with code caching: (a) the total dissipated energy, (b) the average operation delay.
Flooding should be employed. In dissemination scenarios, the same CDS-based control packet sends the delete message to a target node. Therefore, like code installed program modules in each node, it cannot directly perform the update operation. Although PBR uses the early update technique, the energy and delay performances of PBR are almost same as those of LACONIC. Since we randomly select a requester, the path for code delivery does not usually overlap with the previous path history. Therefore, for this random reprogramming scenario the performance improvement by the early update technique is negligible.

**Delete** Since a sink has no information about the list of installed program modules in each node, it cannot directly send the delete message to a target node. Therefore, like code dissemination scenarios, the same CDS-based control packet flooding should be employed.

VI. CONCLUSION

In this paper we investigate a policy-based reprogramming framework that can improve the energy and delay performance of the network reprogramming for wireless sensor networks. To accomplish this, we analyze the execution characteristics of program code modules and define a module class as a tuple of the region, residence, execution synchronization, and real-time properties of modules. Since the execution characteristics of each module determine the traffic patterns and volume of its reprogramming process, we propose a policy-based reprogramming framework that can adaptively select an optimal reprogramming process for each module class. We use 3 different reprogramming techniques; efficient code dissemination using connected dominating set, selective code dissemination, and code acquisition. To evaluate both the energy and the network performance of the new reprogramming framework, we have implemented the proposed scheme on NS-2 simulation framework. For comparative evaluation, we also implemented Trickle and LACONIC which are representative script-based and module-based reprogramming algorithms respectively. Our simulation results confirm that the proposed policy-based reprogramming can substantially improve both the energy and the network performance compared to the existing solutions for various reprogramming scenarios.

REFERENCES


