DPICache: A Distributed Program Image Cache for Wireless Sensor Networks

Joshua Ellul and Kirk Martinez

Abstract— **Wireless sensor networks, like other computing platforms, require software updates from time to time due to software bugs, new functionality, better understanding of the surrounding environment or new applications. Physically connecting to and reprogramming each node is usually not feasible and often impossible. A number of wireless code distribution and reprogramming techniques have been presented in recent years. In this paper we present a novel technique to efficiently update any nodes that were lost during programming periods. Our experiments demonstrate a 57% decrease of bytes sent over the air when updating nodes that previously missed updates.**

I. Introduction

Wireless sensor networks consist of many battery-powered sensor nodes usually equipped with a radio transceiver. The nodes are programmed with an application which aims to monitor the environment they are deployed in. Typical sensor network deployments include military [1], habitat monitoring [2] and environmental [3] applications.

From time to time nodes will require reprogramming. This could be due to solve software bugs, changes in requirements, install completely new applications or also perhaps due to a better understanding of the external environment that was not available during the initial deployment. It is unfeasible and often impossible to physically connect to each node and update the software application perfectly at every attempt and therefore a means of reprogramming sensor nodes in an energy efficient manner must be realized.

In recent years a number of sensor network reprogramming techniques were presented including multi-hop code dissemination [4][5][6], virtual machines that allow for smaller program updates due to the higher level instructions in a VM [7][8] and incremental difference-based approaches [9][10][11]. In this paper we present a novel energy efficient method that allows nodes that missed out any previous incremental updates to update to the current version. It is to our best of knowledge that this is the first attempt at an energy efficient updating mechanism for nodes that have previously missed updates.

This work was partly supported by the Malta Government Scholarship Scheme through award ME 367/07/7.

J. Ellul is with the Electronics and Computer Science Department, University of Southampton, Southampton, SO17 1BJ United Kingdom phone: +44 (0)23 8059 4583; e-mail: je07r@ecs.soton.ac.uk

K. Martinez is with the Electronics and Computer Science Department, University of Southampton, Southampton, SO17 1BJ United Kingdom phone: +44 (0)23 8059 4491; e-mail: km@ecs.soton.ac.uk

II. Related Work

Initial work in reprogramming a wireless sensor network by means of an incremental diff update was proposed in [9]. They efficiently encode the program update by generating a script that contains the differences between the existing image on the nodes and the new image to be updated to. They create the update diff scripts by using a solution similar to the UNIX `diff` command. The diff script will consist of *insert*, *copy*, and *repair* commands that the nodes will follow in order to update their program memory to the new version.

A second diff based approach was proposed in [10]. They enhance the approach in [9] by comparing the object dump files for the old and new program version. By doing so, they only need to compare each old version source object with the same object in the new version. This optimizes the encoding process of the diff script that will usually take place on a base station. In addition to the *insert*, *copy* and *repair* commands used in [9], they also add two new commands: *copy_continue* and *copy_reuse*. These two commands will decrease the size of the diff script since they do not require the full memory address of where to copy to since this can be deduced from the last copy. Another diff based approach that uses a tuned version of the Rsync algorithm is presented in [11]. Unlike the above two methods, this method does not rely on prior knowledge of the code structure but compares at the block level and thus is independent of the hardware platform.

By ignoring the program structure the rewriting that is to be performed on the nodes when an update is injected could have an adverse impact [12]. Rewriting a few bytes in flash memory requires that the whole page of memory be flashed. Thus, it is important that flash pages be rewritten only if required. When ignoring the program structure and the fact that functions may have only moved, shrunk or grown in size, unnecessary flash page writes and a larger delta will be required. [12] propose a delta generation method that is tied with the linking procedure. This aims to minimize shifting functions which have not changed. They also add a *slop* region to each function that allows for the function to grow or shrink as necessary.

III. Methodology

The distributed program image caching technique we present is independent of the actual reprogramming method and dissemination protocol used and therefore can augment any existing update model. In order to provide efficient reprogramming to nodes lost during reprogramming phases the program image updates are scattered over the wireless sensor network. Thus, when a node regains its link to the network it will be possible to update the node from its neighbors rather than having to depend on the base station and the path to it. The algorithm consists of two stages:

1) Cache Selection
2) Node Update

A. Cache Selection

The number of updates injected into the sensor network are cached and distributed amongst the sensor nodes. Due to the memory constraints of sensor nodes our current model allows for each node to cache one program image update. Upon a node receiving a new update (or information of a new update), a decision must be made as to whether to cache the new update or continue to cache the current cached version. The node will then send the cache choice to the neighboring nodes around it to facilitate the cache selection decisions to be made on the peer nodes. The overhead of the above communication is negligible as it can be piggybacked with a 'new software update information' message being disseminated through the network before the software update is sent or even piggybacked with the actual software update. The following priorities are used in our caching technique:

1) The more recent a version is the more important it is.
2) Caching of as many program image updates as possible.
3) Every program image update version should be as close as possible to any node.

Cache Selection Algorithm

A simple decentralized algorithm is used to determine which program update version a node should cache. This is shown below in Figure 1.

Each node will determine whether to cache the new update or the current cached update according to the following rules:

1) If another neighboring node is caching the new update then continue to cache the current cached update.
2) If the current cached update is not the oldest cached version in the neighborhood then continue to cache the current cached update.
3) Otherwise cache the new update.

After the program update selection choice is made the node will broadcast its choice to the other nodes. The simple decentralized algorithm allows the network to adapt to any network topology changes and ensures that any neighborhood will always have the most recent updates available, whilst caching as many different update versions as possible in the neighborhood and thus offering any neighboring nodes the maximum amount of versions possible. Due to the simplicity of the choice procedure very little computation is added to the reprogramming model.

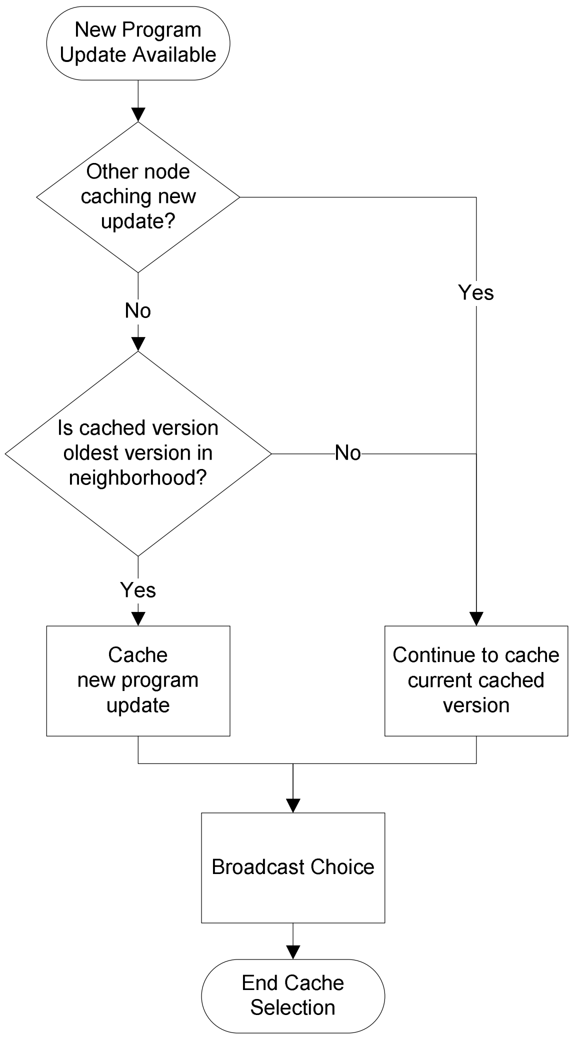
New Program Update Available
Other node caching new update?
Yes
No
Is cached version oldest version in neighborhood?
No
Yes
Cache new program update
Continue to cache current cached version
Broadcast Choice
End Cache Selection

Fig. 1. Cache selection algorithm used to determine which program update version a node should cache upon receiving a new program update.

B. Node Update

A node that has missed out on a program update or a number of program updates will be required to send an update request to its peers once its communication link is reestablished. Two processes are required in order to update a node, being the node update request process running on the node requesting to get updated and the request response process on neighboring nodes.

Node Update Request

The node requiring updates will initially request the next version update from the nodes within its communication range. Should the update be received it will continue to request the following update until it has received all updates and finally sends a message to the neighboring nodes informing them of the successful update. If the requesting node does not receive an update within the estimated response time it will request an update from all nodes that are one more hop away (the estimated response time depends on the network protocols and communication latency in the network). This process will continue until the node has successfully been updated. The algorithm is shown below.

Algorithm 1 Node Update Request

```
1.  Set Hops = 1
2.  Send 'update request' message for current program + 1 to
    nodes that are Hops hops away
3.  If update is received within expected response time then
4.      Update to program received
5.      If still require more updates then
6.          GoTo 2
7.      Else
8.          Send 'updated' message to nodes that are Hops
              hops away
9.  Else
10.     Increment Hops
11.     GoTo 2
```

Node Update Response

Nodes that receive update requests are to discard any pending update messages that were going to be sent to the same requesting node if any. Each node will then check to see if the requested update is in its cache. If it is then the cached update will be sent to the requesting node (by sending it to the node that sent the request message). Otherwise if the node does not have the requested program update in its cache and the request message is for more hops away the node will resend a request message to its other neighboring nodes. Upon receiving an update, destined for the node requesting the update, a node will forward the program update to the node that it received the request from. Algorithm 2 demonstrates this.

Algorithm 2 Node Update Response

```
1.  If was ready to send a previous update
2.      Discard previous update
3.  If the requested update is cached
4.      Send cached update to requesting node
5.  Else
6.      If the request is for more than one hop
7.          Decrement the request message hops
8.          Send request to neighboring nodes
```

IV. Experimental Evaluation

In this section we provide initial results of the DPICache technique produced from a set of simulations. The following assumptions and constants were used in the simulations: a deployment of 1 base station and 49 nodes was used; the base node was always placed at position x = 0, y = 0; any communication overhead at the base station is ignored; nodes were placed in a random fashion subject to each node being within range of the base station or any other node; all nodes have a communication range of 200 cells; the maximum grid size is 1000 width x 900 height; 5 program updates were injected into the network and all nodes are deployed with program 1 and updated to program 6 after the 5 updates. Each simulation was run over 50 different sensor network layouts for 100 different lost node positions.

A. Program Image Update Selection Distribution

We start by demonstrating the distribution of the selected program image updates over the sensor network. Figure 2 depicts a particular simulation run at the end of disseminating 5 program updates. The x and y axes are the actual x and y coordinates respectively where the nodes were placed. Each shape depicts a node where its position is indicated according to the coordinate on the axes and the program image being cached according to the shape. Those nodes that have not yet cached any programs are depicted by a triangle, the nodes caching program 2 update by a diamond, program 3 update by an asterisk, program 4 update by a circle, program 5 update by a square and program 6 update by a plus. As can be seen in the figure in the denser areas some nodes have not yet cached any updates as their neighboring peers have cached all the program updates required, whilst in the less dense areas nodes have chosen to cache the more recent program updates.

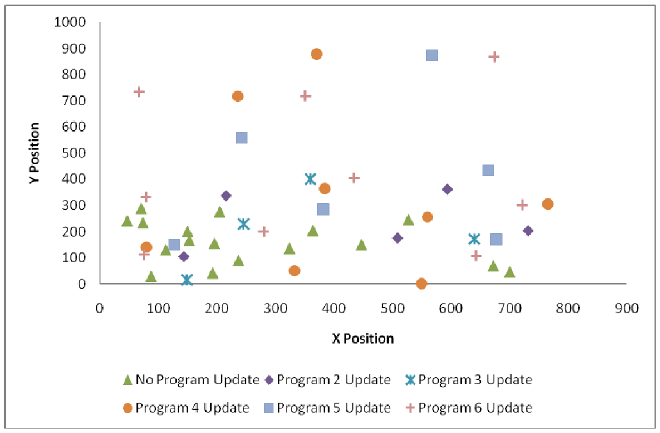

Fig. 2. Layout and program update cache selection of nodes.

B. Comparison with non-caching version

In order to evaluate the distributed program image cache we executed simulations of our technique in comparison to a normal incremental update algorithm. Without the caching technique, the node requesting to get up to date will send a request that is routed to the base station and thereafter the base station sends down all the incremental updates to the requesting node. We assume in this non-caching algorithm that each node is aware of the shortest path to the base station and that route is used for communication. Since the request messages and updates of both versions are of the same size we chose to compare the efficiency of the algorithms according to the number of messages sent and received from sensor nodes in relation to the updates. The DPICache 'update complete' message is of the same size as the 'request update' messages so it will be classified in the same group. We will begin by comparing how the algorithms differ according to the number of updates a node has missed, we then look at how neighborhood density affects the efficiency, and finally at how varying the shortest path to the base station affects transmission requirements.

Varying Missed Updates

The simulation was set for a variable number of missed programs that the requesting node would require. A simulation run of 50 layouts for 100 node positions was executed for each number of missed programs required.

Table 1 displays the total number of messages sent throughout the simulation in regards to the program updates.

TABLE I
COMPARISON OF TRANSMISSIONS

Missed Programs	DPICache Request	Normal Request	DPICache Update	Normal Update
1	12,985	27,052	6,285	22,048
2	22,813	27,142	13,242	44,175
3	36,255	26,697	21,787	64,404
4	52,606	27,999	31,419	90,582

Since the normal incremental algorithm always requests updates from the base station and the shortest path is assumed to be known the number of requests sent remain the same regardless of the number of missed programs. This is due to the fact that a request is sent up to the base station once and thereafter the base station will send down all the updates required in a serial fashion. The DPICache algorithm on the other hand sees an increase in request messages as the number of missed programs increases, due to the requesting node being the coordinating entity requesting each incremental update one at a time. A 52% decrease in messages sent is seen for requests when only one program was missed, however as the number of missed programs increase the tables are turned and the algorithm actually requires more messages to be sent. A request message is generally of the size of a few bytes whilst an update is much larger (depending upon the changes). A single constant change in the `Blink` TinyOS application would generate a 27 byte delta according to [12]; looking at a larger change as described in [12], the delta from the `Blink` to the `CntToLedsAndRfm` application would generate a 10,846 byte delta. According to the simulation the DPICache decreased the number of updates sent by 71% for 1 missed program, 70% for 2 missed programs, 66% for 3 missed programs and 65% for 4 missed programs. As an example let's assume that a node has missed 4 updates and the 4 updates are all just constant changes. Each update would have a delta size of ~27 bytes [12]. Let us assume a request message size of 5 bytes. The total number of bytes transmitted using the DPICache algorithm would equal to 263,030 bytes of request messages and 848,313 bytes of update messages, totaling to 1,111,343 bytes. The base originated updates would total to 2,585,709 bytes. Thus, we can see an overall 57% decrease in bytes sent using the DPICache algorithm, even better results can be seen when the program updates are larger than 27 bytes.

Looking at the messages received in Table 2 we a see a similar trend. The number of update request messages received for the normal incremental algorithm remains in the same region similar to that of the transmissions. The reason is owing to the fact that each node is assumed to know the shortest path to the base station and that other nodes do not overhear the communication between the requesting nodes.

TABLE II
COMPARISON OF MESSAGES RECEIVED

Missed Programs	DPICache Request	Normal Request	DPICache Update	Normal Update
1	84,991	22,052	6,310	27,045
2	140,351	22,142	13,726	54,155
3	207,622	21,697	23,447	79,290
4	276,407	22,999	34,335	110,360

The actual update messages sent for the normal incremental method also reflect the number of updates sent. Since we have decided to exclude the base station's transmissions from the calculations it can be seen that the number of normal updates received is larger than then number of normal updates sent.

The DPICache algorithm does not assume any prior knowledge of the surrounding nodes and thus when an update request is sent it must be heard by all the surrounding neighbors. For this reason the number of requests received is much greater than the number of requests sent. The DPICache update messages received on the other hand can be forwarded down to the requesting nodes without having the surrounding nodes overhearing. Using the example as before, i.e. with a request message size of 5 bytes and an extremely small update size of 27 bytes, the normal incremental update totals to 840,475 bytes received for 1 missed program update for all the simulation runs whilst the DPICache entailed 595,325 bytes received, i.e. a 29% decrease in bytes received. Looking at the 4 missed update program scenarios the normal method totals to 3,094,715 bytes received whilst the DPICache method totals to 2,309,080 bytes providing a 25% decrease in bytes received.

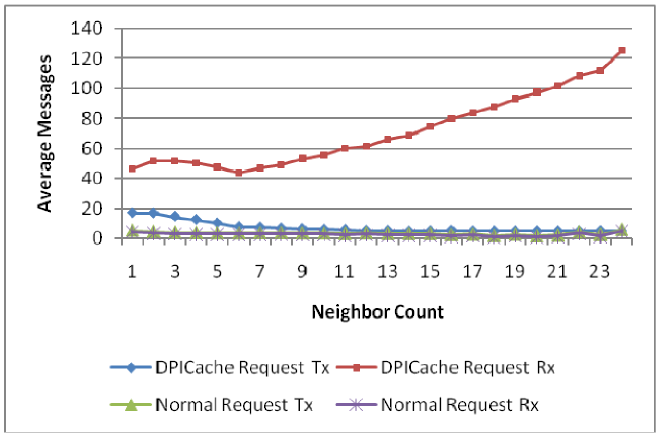

Fig. 3. Request messages sent and received against the neighborhood size. Request messages sent and received using non-caching update algorithm are independent of the number of neighboring nodes.

Varying Neighborhood Density

We now analyze the effects of varying neighborhood density on the algorithms. Simulation runs for a node requesting 4 missed program updates will be used to better demonstrate the benefits gained. The simulations resulted in neighbor densities between 1 and 24 nodes in the same communication range as the node requesting to get up to date. The average number of messages sent and received against the number of neighbors the requesting node is in direct transmission with is plotted in Figure 3 and 4.

The number of request messages send and received using the non-caching algorithm have no dependency on the number of surrounding neighbors. The DPICache request messages sent for a smaller neighborhood are seen to be more than 3 times the size of the non-caching request messages sent. As the neighborhood around the requesting node grows the number of DPICache request messages sent decrease. Since the DPICache update request does not assume any knowledge of the surrounding nodes all nodes must take part in the request process. Due to this the number of DPICache request messages received greatly outnumbers that of the non-caching version, also it can be seen that as the neighborhood increases the number of messages received increases linearly.

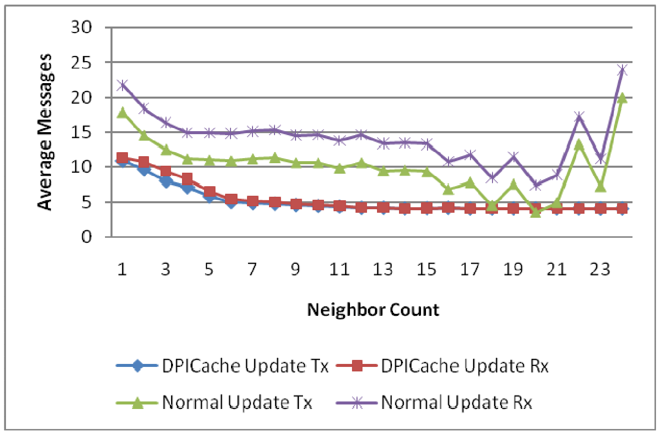

Fig. 4. Update messages sent and received against the neighborhood size. As the neighborhood size increases the number of DPICache update messages decrease.

Figure 4 displays the number of average update messages sent and received against the neighborhood size directly in communication with the requesting node.

From figure 4 it can be seen that the update messages of non-caching algorithm do not have any dependency on the neighborhood. The above graph clearly shows the benefits of the DPICache algorithm, i.e. as the number of neighboring nodes increase the number of update messages decrease. The DPICache algorithm sacrifices a higher number of requests for a lower number of update messages, which is justified due to the small request message size compared to the larger update message size.

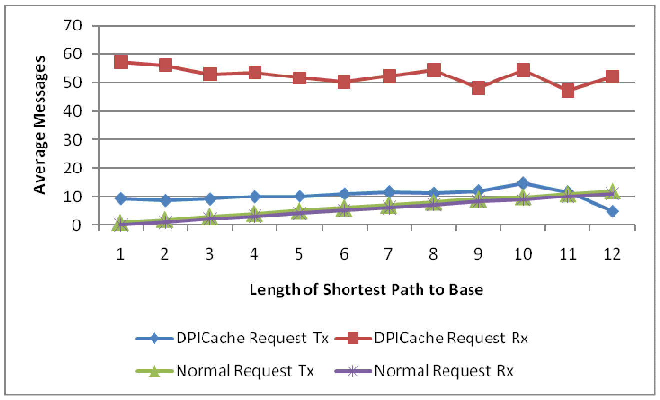

Fig. 5. Request messages sent and received against the length of the shortest path to the base station. As the hop count to the base station increases, the number of request messages sent and received using the non-caching algorithm increase.

Varying Shortest Path to Base Station

The normal incremental update algorithm is dependent upon the shortest path to the base. For this reason we analyze the effect of a varying shortest path on the two algorithms. Simulations were run again for a node requesting 4 missed updates. Shortest paths of sizes between 0 and 11 nodes were seen (excluding the base station and requesting node).

Figure 5 shows the effects of varying the shortest path to the base station on request messages whilst figure 6 shows the effects on the update messages.

The DPICache algorithm has no dependency on the shortest path as can be seen in the graphs. The non-caching algorithm, however, suffers from an increase in messages sent and received as the length to the base station increases. It is interesting to note, though, that for the short path lengths of 1 and 2 the non-caching algorithm requires less update messages than that of the DPICache. For this reason it would be ideal to augment a base station update into the DPICache model if the node requesting updates is close enough to the base station.

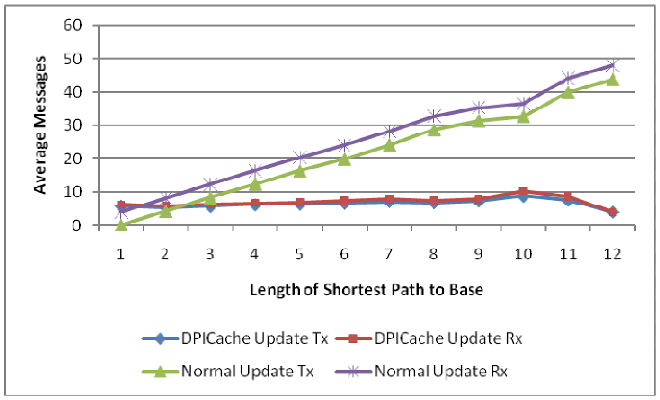

Fig. 6. Update messages sent and received against the length of the shortest path to the base station. As the hop count to the base station increases, the number of update messages sent and received using the non-caching algorithm increase.

V. Discussion

From the experiments above it can be seen that the DPICache algorithm provides a substantial decrease of communication overheard in reprogramming nodes that have previously missed updates. We compared the DPICache technique with a normal update mechanism that sends down the incremental updates from the base station to the requesting node. The overheads of using DPICache must be taken into consideration. Each node is required to have double the amount of memory allocated for program updates, one half used for receiving updates and the other half for storing the cached update. However memory availability is increasing on nodes, so the gains received by using the algorithm can easily be justified.

The program caching technique can be used not only for incremental updates but could also be used for systems that disseminate whole program image updates. Each node could cache a fragment of the program image and thus any neighboring nodes would be able to combine the whole image together.

In our experiments we compared the DPICache technique with one that sends down the updates required in a serial fashion. In some cases, it may make more sense to simply send down a whole new image update rather than sending down incremental updates if the changes in the code are large enough. This factor should also be considered when updating a lost node. However, updates that are sent down from the base station are dependent on the base station and the path to it unlike the DPICache technique. Updates sent down from the base also incur a burden on the base station and the nodes in the path. Using DPICache the burden on the path to the base station is relieved.

VI. Conclusion

In this paper we presented a novel simple incremental program update caching technique that provides efficient reprogramming of nodes that missed any number of program updates. The experimental results demonstrated great improvements over a non-caching update system with a 57% decrease of bytes sent and a 25% decrease of bytes received whilst updating nodes that previously missed updates. Our results also show that as the neighborhood around the requesting update increases the efficiency of the algorithm also increases unlike a non-caching update model. The algorithm is also seen to be independent of the requesting node's distance to the base station.

In the future we would like to implement a test bed of nodes running the DPICache algorithm and actually analyze the power gains achieved. We also plan to add several features that will provide better results including:

1) If the requesting node is close enough to the base station than it should receive updates from the base station rather than the neighboring nodes.
2) Currently a high amount of update request messages are sent amongst neighboring nodes. This amount of messages can be decreased since the nodes should be aware of what versions are available on their neighbors from the cache selection phase.
3) Assign a cache size and allow the nodes to cache a dynamic number of updates.

Acknowledgement

The authors would like to thank the Government of Malta for supporting this research through the Malta Government Scholarship Scheme aware ME 367/07/7.

References

[1] A. Arora et al, "A Line in the Sand: A wireless sensor network for target detection, classification and tracking," in *Computer Networks (Elsevier), 46(5)*, 2004, pp. 605–634.

[2] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler and J. Anderson, "Wireless Sensor Networks for Habitat Monitoring," in *First ACM Workshop on Wireless Sensor Networks and Applications,* Atlanta, GA, USA, September 2002.

[3] K. Martinez, J. Hart and R. Ong, "Environmental Sensor Networks," in *IEEE Computer, 37(8),*pp. 50-56.

[4] T. Stathopoulos, T. McHenry, J. Heidemann and D. Estrin, "A Remote Code Update Mechanism for Wireless Sensor Networks," *Technical Report CENS*, Technical Report 30*,* 2003.

[5] P. Levis, N. Patel, D. Culler and S. Shenker, "Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks," in *First Symposium on Network Systems Design and Implementation (NSDI),* March 2004.

[6] S. Kulkarni and L. Wang, "MNP: Multihop Network Reprogramming Service for Sensor Networks," in *Proc. 25th IEEE International Conference on Distributed Computing Systems,* Washington, DC, USA, 2005, pp. 7-16.

[7] P. Levis and D. Culler, "Maté: A Tiny Virtual Machine for Sensor Networks," in *International Conference on Architectural Support for Programming Languages and Operating Systems,* San Jose, CA, USA, 2002.

[8] P. Levis, D. Gay and D. Culler, "Active Sensor Networks," in *Proc. 2nd USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI),* May 2005.

[9] N. Reijers and K. Langendoen, "Efficient Code Distribution in Wireless Sensor Networks," in *Proc. 2nd ACM International Workshop on Wireless Sensor Networks and Applications (WSNA ;03),* 2003.

[10] T. Yeh, H. Yamamoto and T. Stathopolous, "Over-the-air Reprogramming of Wireless Sensor Nodes," in *UCLA EE202A Project Report (2003),* 2003.

[11] J. Jeong and D. Culler, "Incremental Network Programming for Wireless Sensors," in *Proc. 1st IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks,* 2004, pp. 25-33.

[12] J. Koshy and R. Pandey, "Remote incremental linking for energy-efficient reprogramming of sensor networks," in *Proc. 2nd European Workshop on Wireless Sensor Networks*, 2005, pp. 354–365.