A Dynamic Size Distributed Program Image Cache for Wireless Sensor Networks

Joshua Ellul, Kirk Martinez, and David De Roure

Abstract— **Reprogramming node software over-the-air is an essential requirement in many wireless sensor network applications due to the inaccessibility of the deployed sensor nodes. Transmitting whole software images consumes a high amount of energy in proportion to updates especially when they are small in size. Incremental updates have addressed this, however introduce the potential of a sensor node becoming out of sync when it misses an update. In this paper we present a dynamic size distributed program image cache that provides increased efficiency in reprogramming out of sync nodes and multi-purpose wireless sensor networks.**

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]. An incremental difference-based update can introduce problems when a node misses out an update. A distributed program image cache termed DPICache [12] was presented in aim of providing efficient reprogramming of nodes that missed out incremental updates by caching the incremental updates amongst the sensor nodes.

In this paper we extend on the DPICache algorithm. The DPICache algorithm restricts each node to caching a single program update. The new version of the algorithm, DPICache2, presented in this paper allows nodes to cache a number of updates dynamically according to the free space that is available on each node. Furthermore we broaden the scope for the program image caching algorithm to not only that of incremental updates but also to be able to be used for multi-purpose wireless sensor networks

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

D. De Roure is with the Electronics and Computer Science Department, University of Southampton, Southampton, SO17 1BJ United Kingdom phone: +44 (0)23 8059 2418; e-mail: dder@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.

Previous work on a distributed program image cache demonstrates the benefits gained whilst updating a node that has become out of sync with the current software version. The algorithm involves two stages: the caching stage performed when updates are disseminated through the network and the update request/response stage. The caching stage requires each node to make a decision as to which software version it will cache. Nodes make this choice according to the other choices made by the surrounding nodes. The algorithm uses two simple conditions to determine this: If no other direct neighbor has cached the new update and also if the node is currently caching the oldest update in the neighborhood then it should cache the new update and discard the current cached update. The algorithm can augment any update model since it does not impose any restrictions to the protocol or reprogramming method used. Negligible overhead is incurred since only a few extra bytes would be required to broadcast the cache choice to the neighborhood, and can also be piggybacked with the actual update. The algorithm restricts each node to caching a single update. Thus, if a node is behind by a large

number of updates in comparison to the density of the network, requests to get up to date from the neighboring nodes could lead to no benefit and may even require more transmission than that of a request to the base station.

III. Motivation

As sensor nodes are being developed and released with more and more RAM and program memory, it has become viable to increase the amount of memory dedicated towards caching updates in aim of decreasing energy consumption during program update phases. The algorithm presented in [12] assumes that newer versions are of a higher priority than older versions. By removing this assumption and allowing the developer or sensor network administrator to assign their own priority to updates the caching algorithm becomes not only beneficial to updating out of sync nodes but can also be useful to multi-purpose sensor networks or sensor networks that are reprogrammed on a frequent basis, for example, priority rules could be defined so that popular modules shared between sensor network applications could be assigned a higher priority.

IV. Methodology

This work expands on the DPICache algorithm by enhancing the cache selection process to allow for a dynamic sized program cache size that aims at distributing updates through the network. Each node must be allocated an amount of memory that it can use as an update cache space. Upon receiving an update or information of an update, each node will perform the cache selection algorithm shown in Algorithm 1. If the node has enough free space in its allocated update cache space then it will cache the new update. Nodes will be required to establish the best suited node within its neighborhood to cache the current update. Therefore if a node receives a direction from the sending node then the node must cache the new update and drop the updates which the sender has directed it to. If the receiving node does not have enough free space to cache the update and it has also not received direction from the sending node then the node must determine the best node to cache the update within its network (which includes itself). Once the node has determined which node is best suited to cache the update and if it is the node itself then it will cache the new update and broadcast information required for other nodes to make such decisions. Otherwise it will have determined that another neighbor node is better suited. In this case the node will broadcast its cache information and provide direction to the best suited node to cache the update.

Best Suited Node Calculation

The metric as to which node is the best suited node to cache a particular update is computed according to the amount of free space available, the amount of space consumed by redundant cached updates (i.e. updates that are also cached somewhere else in the neighborhood) and the priority of the updates. If a node receives directions to cache the update then it must do so. Otherwise, the node will compute the best suited node to cache the update in its neighborhood and not according to the sender's neighborhood. Once the node has computed which node is the best suited node to cache the update it can then direct that node to do so in a cache update message that can be piggybacked with the update itself. In order to be able to compute the above, the node must be aware of the sizes and priorities of the updates within the neighborhood and also each update that each neighboring node is caching. So, upon receiving a new update the node will store the following update information tuple, I, in its memory:

$$I = \{u, s, p\}$$

where u is the update id, s the update size and p the update priority. One byte will be used to store the update priority providing a range of 0-255 where 0 implies the least priority and 255 the maximum priority.

The node will also be required to keep track of each neighbor cache information tuple, N, defined as:

$$N = \{n, U, f\}$$

where n is the node id, U is the list of update ids the node is caching and f is the free space available on the node.

The algorithm determines which node is best suited to cache the update by first attempting to find a node with enough redundant cache space to cache the update that has the least aggregate update priority. If no node in the neighborhood has enough redundant space to store the update than the node with the most redundant space and least aggregate update priority will be used. The aggregate update priority for each node is calculated as follows:

$$a = \sum_{1}^{n} s_i \bullet p_i$$

where n is the number of updates the node is caching, s_i is the size in bytes and p_i is the priority of the ith update that the node is caching. The node with the least aggregate update priority is then directed to cache the new update.

Once a node has calculated the best suited node to cache the update it will then require to direct that node to drop a number of updates to make space for the new update. The selection of updates to drop will first include any updates the node is caching that are redundant within the neighborhood. If more updates are required to be dropped to make room for the new update than the updates with the least priority will be

Algorithm 1 Cache Selection

```
1.  Store Update Information
2.  If free space >= update size then
3.      Cache new update
4.  Else If node has been directed to cache this update then
5.      Remove updates directed to remove
6.      Cache new update
7.  Else If no neighbor node has cached this update then
8.      BestSuitedNode To Cache = FindBestSuitedNode()
9.      If BestSuitedNode is this
10.         Cache new update
11.         BroadcastCacheInfo
12.     Else
13.         BroadcastCacheInfoAndDirectNode(BestSuitedNode)
```

included in the list of updates to drop until enough free space will have been made available for the new update.

V. Experimental Evaluation

This section provides an experimental evaluation of the DPICache2 algorithm from a set of simulations. We compare the DPICache2 algorithm with the original DPICache algorithm as well as with a non-caching version. The non-caching algorithm has prior knowledge of the shortest path to the base station unlike both the DPICache algorithms. When an update is required, the non-caching algorithm will send an update request that is sent to the base station. The base station will thereafter transmit the required updates down to the requesting node in a serial fashion. Throughout the simulations the following assumptions and constraints are used: a deployment of 1 base station and 49 nodes was used; the base station was placed at x = 0, y = 0; communication overhead at the base station is ignored; nodes were placed randomly subject to each node being within range of the base station or another node; all nodes have a communication range of 200 cells; the maximum grid size is 1000 x 1000; all nodes were deployed with program number 1; nodes were deployed with an update cache space of 5KB; update sizes were randomly varied between 27 and 4096 bytes. A large range is provided for update sizes; a constant change in code could result in an update size of around 27 bytes according to [13], whilst an update size of 4096 bytes would allow for very large update sizes including whole new applications. Simulations were run for 100 different sensor network layouts which each had 100 different out of sync node positions.

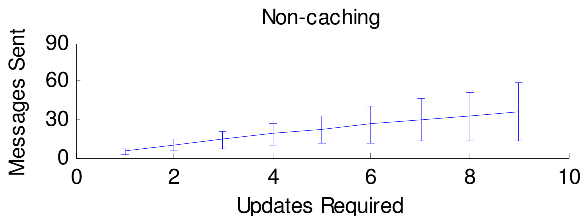

Fig. 1. Update messages sent against the required number of updates when using the non-caching algorithm. The number of update messages sent increases linearly with a large increase as the number of updates required increases.

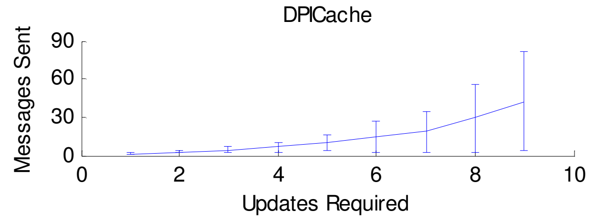

Fig. 2. Update messages sent against the required number of updates when using the DPICache algorithm.

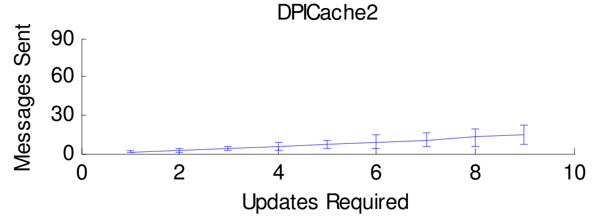

Fig. 3. Update messages sent against the required number of updates when using the DPICache2 algorithm.

Varying Missed Updates

The simulation was run for a varying amount of updates which the out of sync node would require. Simulations for an out of sync node that required 1 to 9 updates were run. Figure 1, 2 and 3 show the number of update messages sent against the number of required updates for the Non-Caching, DPICache and DPICache2 algorithms. As the number of updates required increases the number of update messages sent using DPICache2 increases linearly. The average increase in update messages between each number of required updates was 1.63. The non caching version also demonstrates a linear increase in update messages sent as the number of required updates increases. However, the average increase in update messages sent was 3.72. The original DPICache algorithm eventually increases exponentially. The algorithm actually performs worse than the non-caching algorithm when 9 or more updates are required. The limitations of the original algorithm were not previously identified since such a large number of updates required were not considered. The efficiency of the original DPICache algorithm was highly correlated to the density of the network and thus the limitations of the algorithm for different densities will vary. Figure 4 demonstrates the performance increase of the DPICache2 algorithm compared to the DPICache and non-caching algorithms.

Compared with the non-caching version, a performance increase of 75% is achieved when only 1 update is required and the performance increase achieved when 9 updates is required achieves 58%. Compared to the original DPICache version only a slight increase is gained when a single update is required, however, as the number of updates required increases the performance gained increases reaching up to a performance gain of 66% for 9 missed updates.

Varying Neighborhood Density

Much like the original algorithm, the efficiency of the

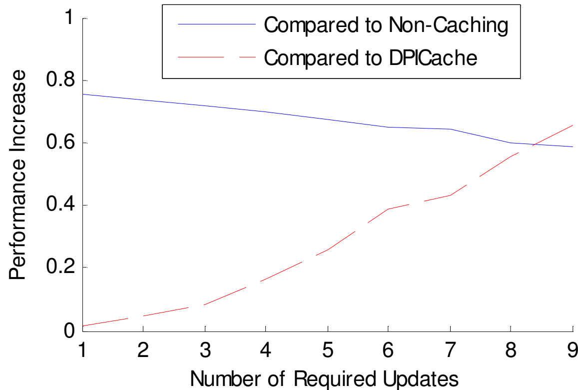

Fig. 4. Performance increase of the DPICache2 algorithm against the number of updates required.

dynamic sized cache algorithm is also highly correlated to the density of the network. The efficiency of the non-caching version has no correlation to the neighborhood density. Figures 5 and 6 show the effect of varying neighborhood density and the number of required updates on the number of update messages sent for the DPICache and DPICache2 algorithms respectively.

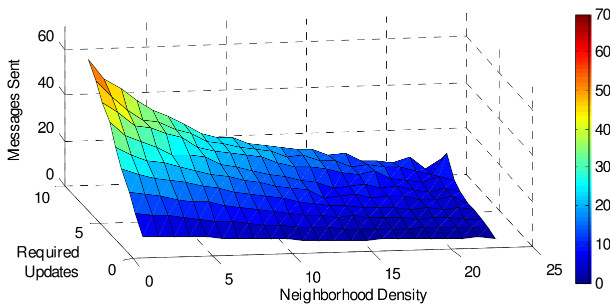

Fig. 5. Update messages sent against the neighborhood density using the DPICache algorithm.

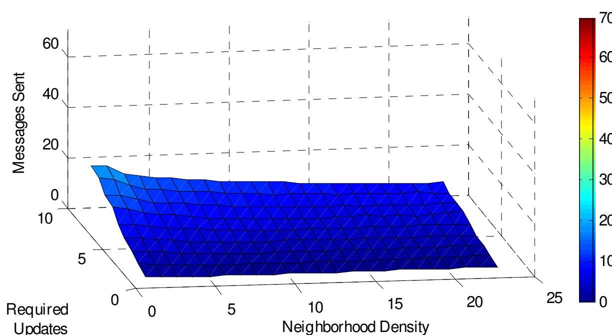

Fig. 6. Update messages sent against the neighborhood density using the DPICache2 algorithm.

Both the DPICache and DPICache2 algorithms demonstrate a high dependency on the neighborhood density. As the neighborhood density increases the efficiency increases. The DPICache2 algorithm performs marginally better than that of its predecessor for less dense areas and performs the same for very dense areas.

VI. Discussion

The cache selection process requires that each node broadcasts its cache information when an update is injected into the network. This overhead is of the size of tens of bytes which is very small compared to the size of software updates. However, this overhead must be modeled.

Prior to this work the DPICache algorithm was only tested for scenarios that required a few number of updates. In this work we have broadened the scope of the new algorithm for that not only of updating nodes that have become out of sync but also to that of multi-purpose sensor networks or sensor networks that frequently update their software. Due to this it may be required that more updates are required to be cached in the sensor network. Our experiments have shown that the efficiency of the DPICache algorithm falls below the non-caching algorithm for cases where the required number of updates is large and the neighborhood density is small. The DPICache2 algorithm may also be subject to such a limitation due to its sensitivity to the neighborhood density and required number of updates.

Caching more updates in the sensor network minimizes the number of update messages sent when updating a requesting node. However, more storage space must be allocated to each node. With more memory being made available to sensor nodes this can be justified.

VII. Conclusion

We presented an extension to the distributed program image cache that allows nodes to dynamically cache updates according to the amount of space allocated to the update cache. The results demonstrated a substantial decrease of energy cost when updating out of sync nodes compared to a non-caching algorithm. We have identified a problem with the original DPICache algorithm when the number of required updates is large and the neighborhood density is small. When such conditions occur the number of update messages required to update the requesting node exceeds that of a non-caching version. The DPICache2 algorithm under the same conditions continues to provides a substantial decrease of update messages required to update the lost node.

In the future we plan to implement a test bed of nodes running the DPICache2 algorithm and analyze actual gains achieved when using the algorithm.

Like the original DPICache algorithm, the efficiency of DPICache2 algorithm is sensitive to the neighborhood density and the number of required updates that a node requests. We plan to perform extensive experiments in order to find the limitations of the algorithm and also formulate the algorithm's sensitivity to the neighborhood density and the required number of updates. We would like to include in our model the overheads introduced in the selection process due

to each node being required to broadcast their cache information.

ACKNOWLEDGEMENT

The authors would like to thank the Government of Malta for supporting this research through the Malta Government Scholarship Scheme award 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. Ellul and K. Martinez, "DPICache: A Distributed Program Image Cache for Wireless Sensor Networks," in *IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications,* Beijing, China, October 2008, pp. 170–175.

[13] 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.