Supporting Concurrent Task Deployment Wireless Sensor Networks*

S. Guo, C. Fan, and T.D.C. Little
Department of Electrical and Computer Engineering
Boston University
8 Saint Mary's St., Boston, MA 02215, USA
{guosong,cxfan,tdcl}@ bu.edu

June 1, 2008

MCL Technical Report No. TR-06-01-2008

Abstract—Deploying large-scale sensor networks involves the programming of many devices based on a desired mission. Techniques for reprogramming devices in situ have been investigated to mitigate the effort required when program updates are required or when the mission of the system changes. We consider a technique that is intended to support multiple concurrent missions by the system by exploiting available resources of the sensor network. In essence, our model is based on a tasking scheme, a common framework for the interchange and instantiation of tasks on multiple devices, and the use of attributes defining the resources in the system. A prototype system has been implemented to demonstrate and validate the concepts using the Imote2, a 32-bit mote architecture that has been configured with embedded Linux enabled with Java. Several applications have been rendered as tasks that are injected into a multi-node sensor network. Results demonstrate the support of concurrent overlaid applications in the system permitting task injection, maintenance, and termination. Performance evaluation of the scheme indicates benefits over an epidemic model of code dissemination.

* *In Proc. Intl. Symp. on Network Computing and Applications (NCA 2008),* July 11, 2008, Cambridge, MA. This work is supported by the NSF under grant No. CNS-0435353. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

1. Introduction

Wireless sensor networks (WSNs) are expected to encompass very large numbers of devices. For example, in a building automation scenario, it is not unreasonable to expect many sensors and actuators to exist in each room. These might be smoke detectors, light switches, lights, HVAC valves and thermostats. History shows us that these devices continue to reduce in size, energy consumption, and performance, and as with other maturing technologies, the dominant lifetime costs become tied to installation and maintenance. In the case of installation, a desirable feature of, for example, a wireless thermostat, is its ability to be installed by a non-specialist and for it to subsequently discover and associate with its corresponding HVAC control loop. We claim that this involves discovery processes enabled by the use of localized information realized as attributes.

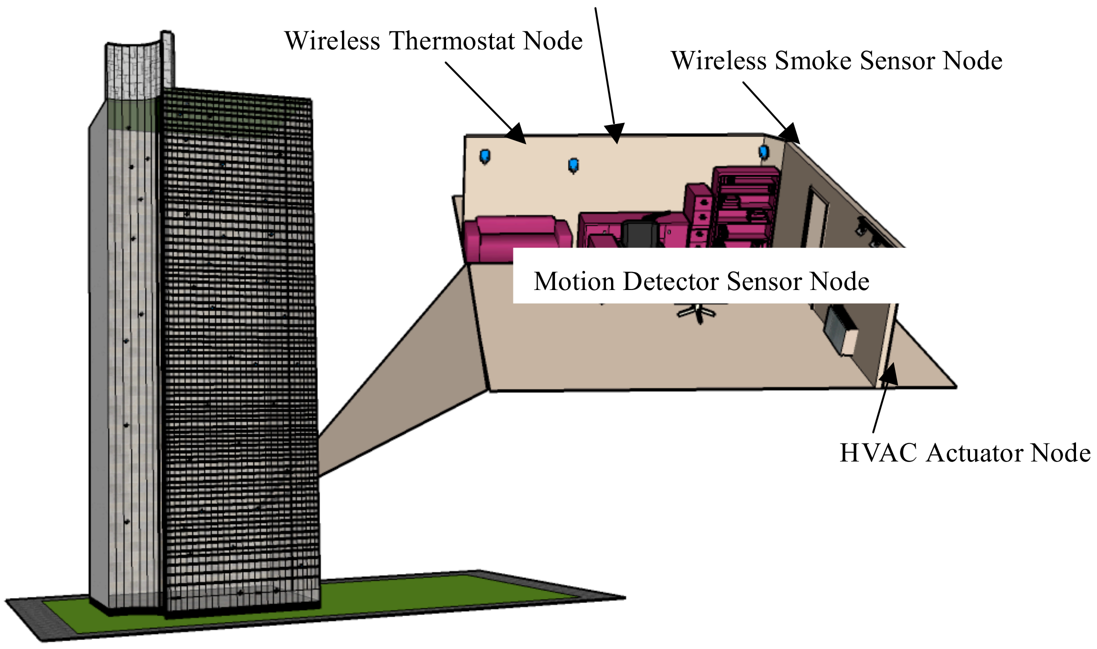


Fig 1: WSN in Building Automation Example

Similarly, maintenance of a WSN can potentially be costly from a labor standpoint. Physically touching each device can be of an equivalent cost as installation. Replenishment of devices, as an option presents heterogeneity challenges as newer devices often rely on more recent code releases or features. Thus efforts have been applied to create code dissemination schemes that permit updates to be propagated wirelessly, "over the air" to all devices in the system.

However, these schemes are usually an all-or-nothing activity for various reasons, primarily because the devices are resource limited and are challenged to support large or complex process management, and because of the goal to have predictable behavior by limiting the system to a single application. We challenge these assumptions by targeting specific objectives:

1. Allow the WSN to support multiple coexisting missions
2. Allow the missions to be dynamically created, injected, and retired
3. Be energy efficient as a goal but not a primary motivation

Our approach for these objectives is to adopt a tasking model. A mission is defined as a WSN function, such as an HVAC-thermostat-actuator control loop. With a tasking model the function is implemented by code rendered and configured on the motes involved in the control loop. Thus there can be *n* task instances that exist for this function on a subset of the WSN. In order to implement this model we must achieve certain behaviors that emerge:

1. Concurrent functions: multiple control loops in the WSN that are non-intersecting (e.g., multiple rooms)
2. Concurrent tasks on a device: multiple functions that leverage the same sensor
3. Baseline propagation of tasks for installation and propagation in the system
4. Injection of missions, defined by tasks, into the system for dissemination, adaptation, and execution
5. Limiting of task propagation and instantiation to the devices that need the tasks
6. Mobile code translation to accommodate functions not known in advance

With this model, we design general paradigm for programming subsets of a WSN. An artifact of this model is the ability, by leveraging attributes associated with devices and their data, to form specialized communication abstractions [1].

In the remainder of the paper we describe a proposed framework for supporting these concurrent missions with a tasking model. The scheme is supported and demonstrated by the use of Java on the Imote2 platform configured with ARM-Linux. We also demonstrate performance of the scheme as assessed using the OPNET network simulation environment. Although we reference a building automation scheme, the results are expected to generalize to many WSN scenarios.

2. Related Work

Wireless sensor networks are characterized by low cost, low power embedded sensor devices that are network-enabled. For most applications, sensor nodes in a WSN are initialized once and intended to operate for the duration of the application or mission. However, shortcomings of this approach include the costs associated with implementing code updates. Thus there has been considerable interest in alternative models to (a) reprogram, (b) design APIs for programming, (c) better map intereractions among nodes to the WSN via programming techniques. Hood [15] is a neighbor-based programming framework addressing the latter topic. This scheme parameterizes the network application as different attributes and provided filtering mechanism to select and share data based on data attribute. Abstract Regions [1] generalizes and improve the idea of Hood via abstractions above required inter-device communication. It also considers mechanisms for data aggregation. Mate [8], TinyDB [5], Trickle [9], and Squawk [22] address code dissemination with a focus on efficiency. Mate and TinyDB use high-level virtual code representations to reduce the code transmission cost. Mate is quite functional but is limited in the scope of instructions that can be achieved in its virtual machine and is best suited for simple applications. Trickle improves Mate, with a remedy for simple flooding for dissemination by limiting code propagation to subsets of network nodes. Squawk introduces Java as a WSN programming tool and uses Squawk bytecodes to transmit a program.

Unfortunately these code dissemination schemes do not support incremental application updating. References [4] and [24] provide some solutions. Reference [24] develops an algorithm that can efficiently encode program updates. An edit script is generated by host program and corresponding operations are cast on each device at the instruction level. However, this work is essentially a coding scheme with strong processor dependencies. Reference [4] describes a generalized program update scheme. Without prior knowledge of the program code structure, this scheme is designed to distribute key changes of new version of a program. However, a shortcoming is the inability to distribute different application code to different subsets of nodes. MOAP [25] and Deluge [3] are two multi-hop network programming implementations. MOAP succeeds in propagating program code to a selective number of nodes without saturating the whole network. Deluge disseminates the program in an epidemic fashion and improves the throughput using optimization techniques like adjusting transmission rate. Unfortunately, the required rebooting process of new program loading degrades the performance of network application and this is shown in our OPNET simulation results.

3. Task Based Reprogramming

Our proposed tasking scheme relies on three key components: (1) a decomposition of tasks, (2) installation or injection into the system, and (3) message processing based on attributed data at each node. These are each discussed in detail below.

3.1 Task Decomposition

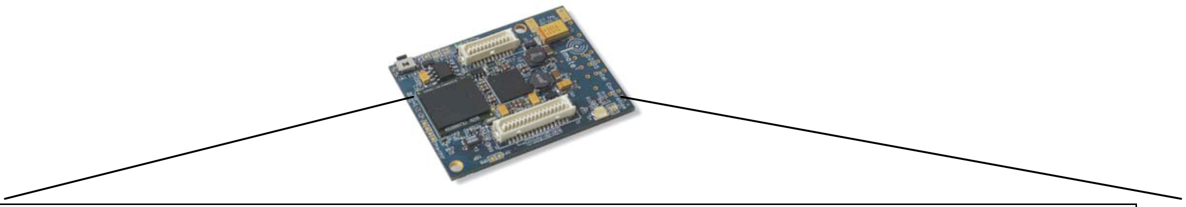
We seek to enable incremental, functional updates, and the ability to support concurrent missions. In our scheme we model a sensor net application as a mission supported by a set of independent and identical tasks that are disseminated on a subset of the WSN devices (a mission is achieved by identical tasks disseminated on a subset of all nodes). However, this subset is guided by attributes associated with the devices and predicates that aid in their instantiation. For example, one can envision a temperature monitoring application that is instantiated on nodes that possess temperature sensors with two independent task modules: a routing task and a temperature measuring task as shown in Fig 2. Other nodes in the network need only to support the routing task. Multiple missions are achieved by the injection of multiple tasks that are disseminated to nodes with matching attributes. The decomposition process is based on the analysis of the running application. Right now we do not have a general algorithm for the decomposition but we require the decomposition to achieve the following goal. We denote the function of the running application as *F(A),* the function of the decomposed task as *F(T_i),* the decomposition requires that:

1) $F(A) = \sum_{i=1}^{N} F(T_i)$

2) All of the decomposed task T_i form a partition of the general application A. A Partition of a set A is a set of disjoint subsets of A, whose union is A

$$T_i \subseteq A, \bigcup_i T_i = A, T_i \bigcap T_j = \phi, \;\; if(i \neq j)$$

The independence of the different tasks in the WSN benefits the task installation since it ensures no interference with currently running tasks. Every node contains a task table and tasks run in parallel on the node. Furthermore, the process of new task installation is eased by only adding a new entry to the node's existing task table.



Task Table	
Application: Temperature Monitoring	**Task One: Measure Temperature**
	Task Two: Routing Temperature data to gateway

Fig 2: A Mission is Comprised of Multiple Identical Tasks Mapped to Attributes of Nodes

3.2 Task Installation

Unlike existing over-the-air programming (such as Deluge [3] and Sun Spot[22]) which feed all the nodes with the same code, we install different tasks onto different nodes based on the predicates used in dissemination that are applied on node attributes.

Our task installation and forwarding scheme are inspired by Content Based Routing protocol proposed in [10]. In this scheme, a set of predefined predicates on each node can be used to delineate a subset of nodes to target our tasks. These predicates per node are called the Installation Predicate (IP). The Installation Predicate is defined as follows:

$$IP = \{p_{1,} p_2, p_{3,} ...\} \ \ where \ \ p_i \ \ is\ individual\ predicate\ on\ each\ node$$

The Installation Predicate *(IP)* over the received task code T_i defines a task addressing scheme.

$$IP(T_i) \longrightarrow \{\text{true, false}\}$$

If the result is true, the node will install the received task module, otherwise the node ignores the task and forwards it to the proper neighbors.

In the forwarding process, each node forms a Forwarding Predicate (FP) related to each neighbor using the algorithm proposed by [10]. We denote the set of all Neighbors of each node as *N* and give a Task module Forwarding Function (TFF) to identify the neighbor which needs to receive the task module code:

$$TFF(T_i, FP) = \{i : i \in N \cap FP(T_i) = True\}$$

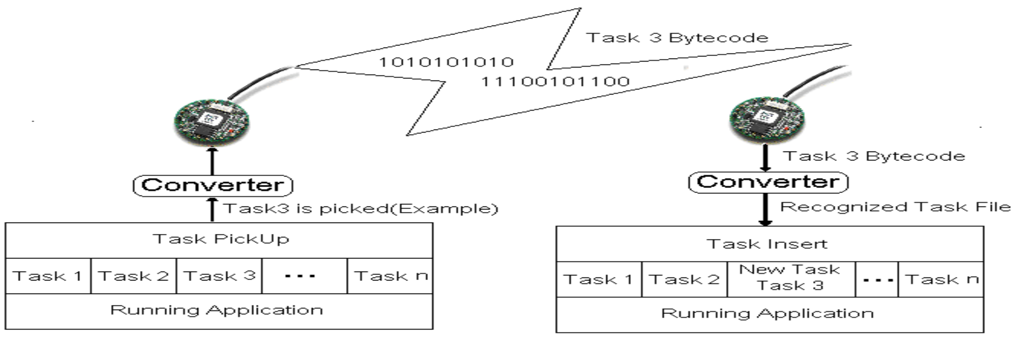


Fig 3. Illustration of Wireless Task Installation

During the task installation process, there is no reboot or reset operation as associated with other types of over-the-air programming. Every application can be modified and updated by stopping, retiring, and adding different tasks. The task installation and deletion process is intended to leverage common data structures on each node (e.g., neighbor connectivity information) yet not interfere with concurrent tasks. Optimization of inter-task cooperation is beyond the scope of our current scheme. The installation process is illustrated in Fig. 3. Here a set of running applications is defined by multiple tasks. The sender converts the updated task (Task 3 in Fig. 3) into byte code and transmits the code over the network. The receiver converts the byte code back to a task module and inserts the task module into its own task table to accomplish the application updating without interfering the other running task modules.

3.3 Message Processing

In our framework, the node behaves as a message forwarder. Fig.4 shows the message-processing framework within a node. Every incoming message is first parsed by an interpreter according to the predefined message format. The Interpreter will then forward the message to a proper task module for further processing based on the attribute forwarding scheme. The processed messages are all sent to a package module for duplicate removal. This module will repackage the received messages and send them out.

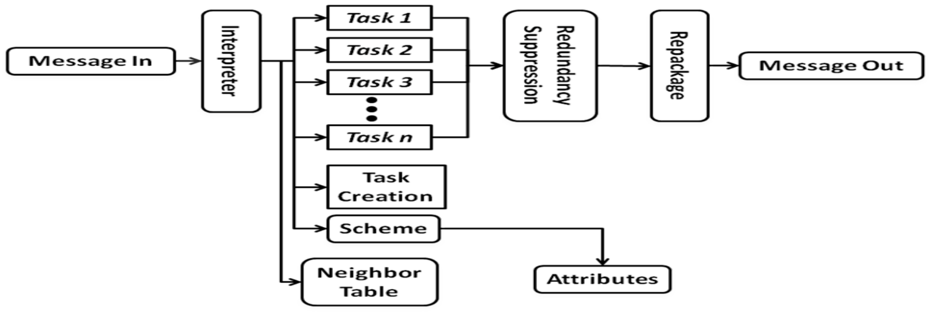


Fig 4. Message Processing within a Node

3.4 Main Features of the Task Based Reprogramming

Independent tasks serve as the logical programming units mapped to nodes in the network. The application can be easily modified or updated by installing, updating and deleting individual Task Modules.

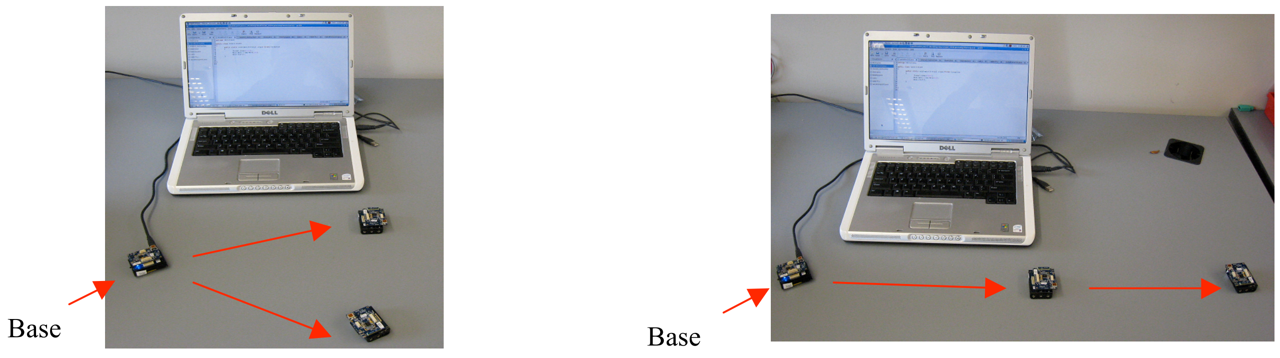
We use predicates, which are predefined installation rules, to guide the installation of the task module. Predicates enable us to install different tasks on different nodes based on the nodes' property. This feature prevents the node from installing the code unrelated to its function and thus saves resources.

There is no reboot or reset operation on the node. The node's function is achieved by combining different independent and parallel running task modules together. Task independence along with the message processing scheme guarantees that the update of one particular task module will not interfere with other running tasks. Thus there is no direct impact on the network's performance.

Finally, the support for independent tasks allows the injection of multiple missions represented by independent tasks. Different missions are achieved by injecting unique tasks, or identical tasks with different injection predicates.

4. Proof of Concept Implementation of Task Based Reprogramming

We implemented our framework on the Crossbow Imote2 platform with attached sensor boards. The motes were installed with embedded Linux v.2.6.14 including Java v.1.3.0. Java was selected due to its popularity, convenient APIs, and the potential to support code mobility. Support for handling multiple tasks is achieved by Java multithreading operations. Code injection is achieved using Java's mobile code transmission architecture. However, some effort was required to adapt the Java VM to support instantiations from foreign nodes that received injected code.



a) single-hop scenario **b) multi-hop scenario**

Fig 6. Prototype Setup

In our demonstration of the framework we programmed a base station with two parallel tasks that are injected into a set of autonomous WSN nodes. The first task enables each mote toggle its blue LED once per second. The second task leveraged the Imote2 sensor board to periodically sample temperature. This drives each node to act as an aggregator of temperature values from each of the other nodes in the network. The node with the highest value illuminates its own red LED. In lab tests we successfully instantiated these concurrent tasks in single-hop (star) and multi-hop configurations shown in Fig 6. For the first scenario (Fig. 6a), the two satellite motes are within one hop range of the base station. We observe that the two satellite motes would install new task simultaneously which is a gain from the code broadcast of the base station. For the second scenario (Fig. 6b), one of the satellite motes is out of range of the base station; the three motes consist of a two hop count network. Periodically broadcasting of task list makes sure the farther mote from base station could correctly install the new tasks. In our demonstration we implemented some basic Java APIs for Imote2 programming, such as Task interface, Sensing interface, Listening_Message interface, Data_Sending interface, etc and these APIs could be easily ported to any other sensor node which is equipped with a standard java virtual machine. We hope in the future we could provide more and more standard Java APIs of our scheme for the users to make the programming of WSN simpler and thus encourage more effort to be given to investigate the potential of WSN application instead of worrying about the implementation complexity.

5. Simulation

Although the focus of our efforts have been directed towards specific functional goals, we have sought to understand the practical limitations to our design and implementation strategy. In this section we investigate the performance of the TBR scheme in a comparative analysis with other code dissemination techniques. We investigated performance by modeling the TBR scheme and its instantiation in the Opnet network simulation environment. The details of our performance study are shown below.

5.1 Performance Metrics

We quantify performance using the following dimensions:

1. *Task Dissemination Percentage*: Fraction of nodes that have been injected required tasks in the network during simulation
2. *Application Dissemination Time*: Average time cost for a complete task dissemination over the entire network

These two parameters are critical to understanding the behavior of task injection. We expect the use of Java to incur penalties in terms of bytes transmitted and do not dwell on this metric.

5.2 Task Dissemination Protocol

There are two main timers in our protocol: TAD_TIMER(Task Advertising timer) which controls the period of advertising the node's task list and RCV_TASK_Timer(Receive Task timer) which control the expiration time of receiving a task block code. Two kinds of packets are applied: RQT_Packet(Request Task Packet), which inquiry a particular task and RQN_Packet(Request Neighbor Packet) which is used to detect the neighborhood. We perform analysis based primarily on this Task Dissemination Protocol which is specified below.

1. Each node maintains its own task table and periodically broadcasts its task list to its neighbor until TAD_TIMER (task advertising timer) is out. There are two states of the node: normal and listening to task code.
   Pseudo Code:

```
If(TAD_TIMER expire && node_state == normal state)
    {   Broadcast(TAD_Message)   }
```

2. Receiver receives TAD message and checks the task injection rule to decide whether to inject this task or not. By default, the rule is to inject the first new task found in the task list. When a task is needed to be injected, the receiver sets the new task parameters to the inject phase state and jumps to listen task state: three task parameters are used: Name, Source and State.

Name: task name to be injected;

Source: where the task comes from, eliminating the duplicate task injection from other node.

State: the state of the task:(0: temp task; 1: permanent task; -1: destroyed task)

Pseudo Code:

```
If(Receive TAD_Message && nodestate== normal state)
  {
      Received_Task_List=getTaskList(TAD_Message);
      If(CheckTaskInjectRule(Received_Task_List)==true)
      {
          Jump State to Listening Task Code;
          Set RCV_TASK_TIMER;
          Send(RQT_Message)
          Waiting Task;
      }
  }
```

3. Receiver jumps from normal state to L_Task state (Listening Task Code State) waiting for incoming task and ignoring other interrupts
4. Receiver sends back RQT_PACKET to the specified task source node
5. Sender receives the first RQT_PACKET will fall back a random time in order to achieve a multicast of task code. The nodes in the neighbor list with the same task requirement could benefit from the broadcast of task code.

Pseudo Code:

```
If(Receive RQT_Message && state == normal state)
  {
     Fall back a slight time;
Broadcast(Task_Code);
  }
```

6. Receivers with different task requirements (different desired tasks) will simply discard the code and wait for the next round task inquiry
7. When the task injection succeeds or injection times out, the receiver will jump back to normal state. The node will add a new task in its local task list (Task Map) if the injection is successful. Otherwise, the node jumps back to normal state, deletes the temp task from task list and resets the task state to -1

Pseudo Code:

```
If((RCV_TASK_Timer expire || Task Inject Success) && state == listening task
 state)
        {
```

```
        Jump back to normal state;
    }
```

8. During normal state, the node periodically broadcasts RQN_PACKET to push itself into the neighbor's neighbor list.
9. Receiver adds the sender of RQN_PACKET to its neighbor list.

5.3 OPNET Simulation Model

We built a simulation environment for the Task Dissemination Protocol using OPNET based on grid-shaped network topology shown in Fig 7: (node_0 is the base station).

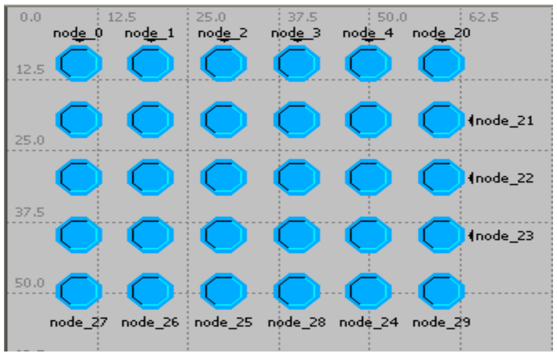


Fig 7. Grid Topology for Simulation

Radio transceivers in the node model have the following parameters: a 2.4 GHz center frequency, a single physical channel with a bandwidth of 5 MHz, and the raw bit rate of 250 kbps. These parameters are consistent with the standard physical channel setup used in the IEEE 802.15.4 used by the Imote2s in our testbed (Section 4). We also customized the receiver group pipeline stage function such that the transmission range of a node is set to 10 meters for simulation

Each node contains four components: radio receiver, processor, radio transmitter, and antenna as shown in Fig 8.

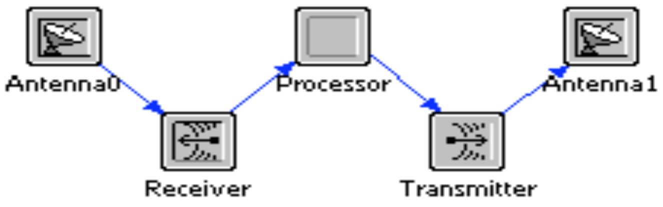


Fig 8. Node Model

Fig 9 illustrates the implementation of the processor module, which is the core implementation of the Task Injection Protocol

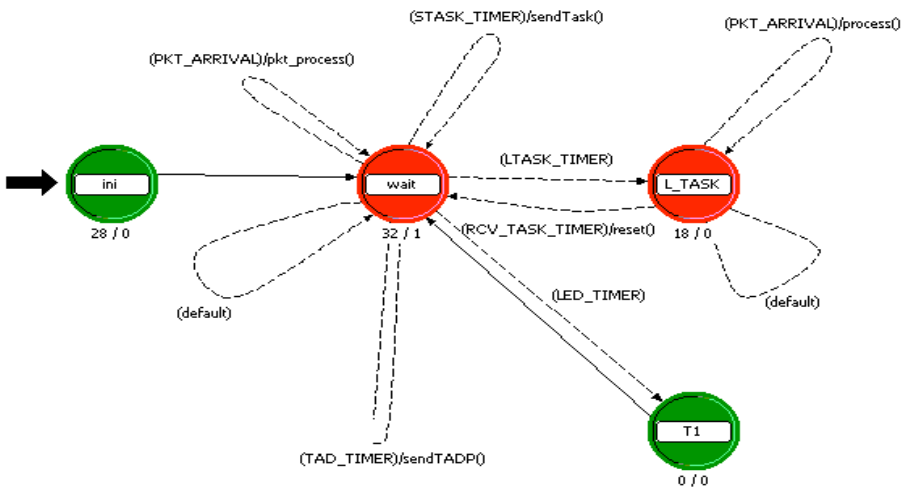


Fig 9. Processor Module Implementation

5.4 Opnet Simulation Results

For simplicity in the simulation we do not consider radio collision problems and assume that they are addressed by an underlying MAC layer. We regulate the size of each task to be identical and equal to 1kB. In order to avoid collision, we set the TAD_TIMER expiration to be a random time uniformly distributed between [5, 10] seconds. The node backoff time is also random, uniformly distributed between [0.2, 0.4] seconds.

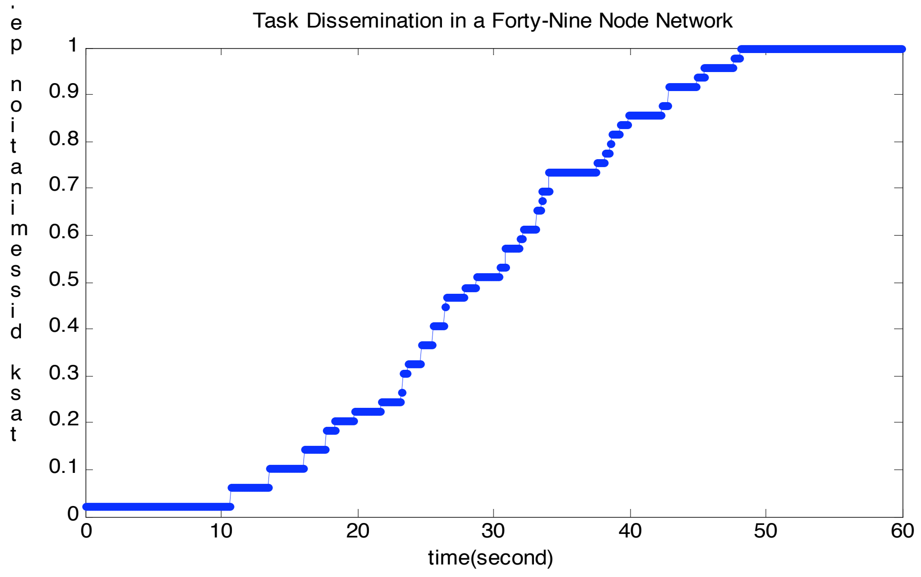


Fig 10. Task Dissemination Time w.r.t Network Size(size = 49)

We first analyze the single task dissemination time over different sized network. We apply our task injection scheme on a square mesh network with the size range from 4 (2x2) nodes to 49

(7x7) nodes. Fig 10 shows the result for the largest network size (7x7). From the results, we observe that a task can be completely installed on all nodes in a 49-node network spanning an area of 360 square meters within one minute. This time is negligible compared to the lifetime of a typical WSN and little impact on the overall performance of the network.

We further measured the task dissemination speed on different sized networks. The definition of average task dissemination speed is defined as follows: The result is provided in Fig 11.

$$\text{AVG Speed} = \frac{\text{Total \# of Tasks Need to Install}}{\text{Application Diseemiantion Time}}$$

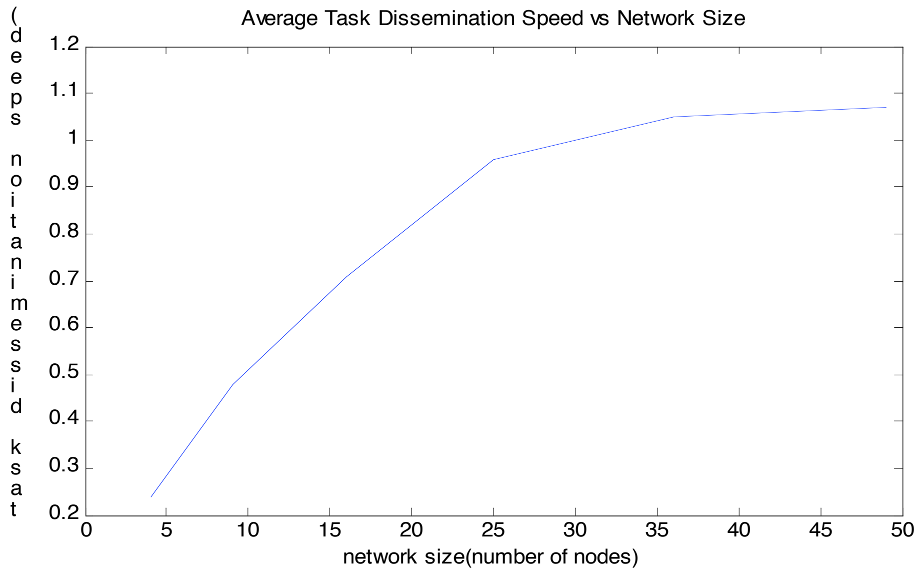


Fig 11. Task Dissemination Speed vs. Network Size

Fig. 11, illustrates an advantage of our scheme; the task dissemination speed increases with the size of the network as would be expected as participation by nodes increases as propagation radiates. By further analyzing the relationship between Task Injection Completion Time and Network size we reach the results shown in Fig. 12.

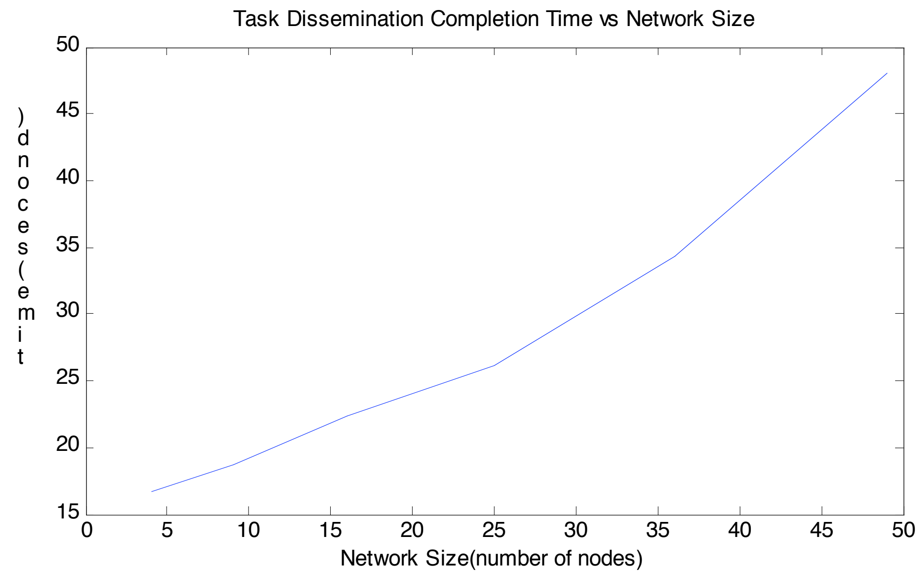


Fig 12. Task Dissemination Time vs Network Size

Fig. 12 demonstrates that task dissemination time complexity in our scheme grows almost linearly with respect to the size of the network. This result indicates good scalability of our scheme.

Clearly injecting smaller, autonomous tasks is more efficient than injecting large monolithic applications. Fig. 13 shows a comparison of task-based injection vs. injection of a single large application that can be represented as five tasks with the scheme which is adopted from reference [22] across a 7x7 square mesh network (Fig. 14). These results favor the tasking model in terms of installation time.

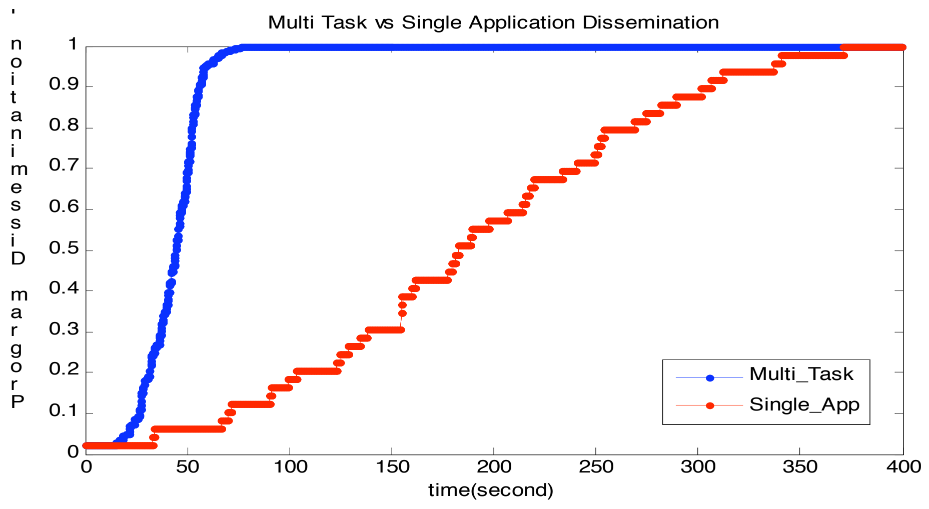


Fig. 13 Multi Task Dissemination vs Single App Dissemination

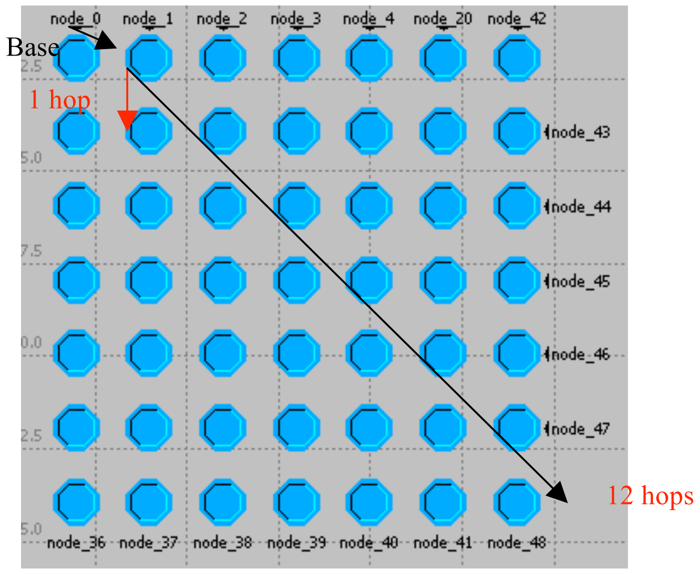


Fig 14 Topology of 49 Nodes

Based on the simulation results we claim that compared to the traditional program dissemination scheme, our tasking approach has beneficial functional and performance characteristics. Additionally, by eliminating mote rebooting processes as required for other schemes, we can reduce the time for code installation.

6. Conclusion and Future Work

We described a task-based programming methodology that permits the execution of concurrent missions in a WSN. This model is demonstrated with a Java implementation and shows potential for performance gains due to the reduction in the unit of reprogramming of an application. Network updating becomes an easier job with the help of proper application decomposition and individual update of the task module on each mote; Run-time updating is achieved and new task installation has no interference with other running applications; Network programming and reprogramming are simplified by using Java and its standard APIs. Future work involves the creation of more complex WSN missions as implemented as tasks and to identify the performance limit of concurrency using Java in this manner. It is our goal to simplify the process of deploying sensor networks using with the task model and Java.

References

[1] M. Welsh, Geoff Mainland, “Programming Sensor Networks Using Abstract Regions”, *NSDI* 2004, p. 29-42.

[2] Boulis and M. B. Srivastava, "A Framework for Efficient and Programmable Sensor Networks", In *Proc. of OPENARCH* 2002, NY, p.117-118.

[3] A. Chlipala, J. Hui and G. Tolle, “Deluge: Data Dissemination in Multi-Hop Sensor Networks,” *UC Berkeley CS294-1 Project Report*,Dec. 2003.

[4] J. Jeong and D. Culler, “Incremental Network Programming for Wireless Sensors”, *IEEE SECON 2004*, p. 25-33.

[5] S. R. Madden, M. J. Franklin, J. M. Hellerstein and W. Hong, “TinyDB: An Acquisitional Query Processing System for Sensor Networks”, *ACM Transactions on Database Systems*, v 30, n 1, March, 2005, p. 122-173.

[6] P. J. Marrón et al., “Management and Configuration Issues for Sensor Networks,” *Int’l. J. Network Mgmt*, vol. 15, no. 4, July 2005, p. 235–253.

[7] A. Carzaniga, M.J. Rutherford, and A.L. Wolf, "A Routing Scheme for Content-Based Networking", *Proceedings of IEEE INFOCOM 2004*. HK p.918-928.

[8] P. Levis and D. Culler, “Mat´e: A Tiny Virtual Machine for Sensor Networks,” *ASPLOS, ACM SIGOPS Operating Systems Review* v.36, p. 85 – 95.

[9] P. Levis, N. Patel, S. Shenker, and D. Culler “Trickle: ASelf-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks”, *NSDI 2004*, p.15-28.

[10] A. Carzaniga and A.L. Wolf, "Forwarding in a Content-Based Network", *Proceedings of ACM SIGCOMM 2003* Karlsruhe, Germany. August, 2003, p. 163-174.

[11] R. Tynan, , D. Marsh, D. O'Kane, and G. M. P. O’Hare, “Agents for Wireless Sensor Network Power Management”, *Proc. of the 2005 ICPPW*, p. 413-418.

[12] L. L. Petrea, D. Grigoras, “Towards Introducing Code Mobility on J2ME”. *ISPDC 2005*, p. 173-182.

[13] C. Tschudin, H. Gulbrandsen, H. Lundgren, “Active Routing for Ad-hoc Networks”, *IEEE Communications Magazine,* v 38, n 4, Apr, 2000, p 122-127.

[14] K. Whitehouse et al, "Marionette: Using RPC for Interactive Development and Debugging of Wireless Embedded Networks". *IPSN/SPOTS '06*. Nashville, TN, April 21, 2006. P.416-423.

[15] K. Whitehouse et al. "Hood: a Neighborhood Abstraction for Sensor Networks." *MobiSys’04*. Boston, MA, June, 2004. p.99-110.

[16] C. Intanagonwiwat, R. Govindan, D. Estrin, “Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks” *Proc. of the Sixth Annual International Conference on Mobile Computing and Networking*, 2000, p 56-67.

[17] J. M. Hellerstein, W. Hong, S. Madden, and K. Stanek, “Beyond Average: Towards Sophisticated Sensing with Queries”, *Information Processing in Sensor Networks. Second International Workshop*, 2003. p 63-79.

[18] U. P. Schultz, K. Burgaard, F.G. Christensen, J.L. Kristensen, “Compiling JAVA for Low-End Embedded Systems”, *ACM SIGPLAN Notices*, v 38, n 7, July, 2003, p 42-50.

[19] Q. Wang, Y. Zhu, L. Cheng, “Reprogramming wireless sensor networks: challenges and approaches”, *IEEE Network 20(3)*, 2006, p. 48-55.

[20] C.C. Han, et al, “A dynamic operating system for sensor nodes”, *MobiSys 2005*, p. 163-176.

[21] D. Simon, et al, “JavaTM on the bare metal of wireless sensor devices: the Squawk Java virtual machine” In *VEE ’06: Proc. of the 2nd international conference on Virtual execution environments*, New York, NY, USA, 2006. p78–88.

[22] SUN Spot Project: http://www.sunspotworld.com/.

[23] OPNET University Program http://www.opnet.com/ university_program/.

[24] N. Reijers and K. Langendoen, “Efficient Code Distribution in Wireless Sensor Networks,” *WSNA ’03* San Diego, CA, USA 2003, p. 60 – 67.

[25] T. Stathopoulos, J. Heidemann and D. Estrin, “A Remote Code Update Mechanism for Wireless Sensor Networks,” *CENS Technical Report # 30*.