A Retasking Framework For Wireless Sensor Networks

Michael Ruffing*, Yangyang He†, Mat Kelly*, Jason O. Hallstrom†, Stephan Olariu*, Michele C. Weigle*

*Department of Computer Science, Old Dominion University
Norfolk, VA 23529-0162
mruffing@gmail.com,{mkelly,olariu,mweigle}@cs.odu.edu

†School of Computing, Clemson University
Clemson, SC 29631
{yyhe, jasonoh}@clemson.edu

***Abstract*— Wireless sensor networks have been widely used in scientific research, industrial manufacturing, and environmental monitoring over the past decade. Using pre-existing networks to assist in responding to disaster events can be cost-effective. In this paper, we present *Alert*, a software framework for retasking wireless sensor networks, enabling these networks to respond rapidly to unexpected events without neglecting their originally assigned tasks. *Alert*, built upon Deluge [1], is a wireless network code distribution protocol enabling node group management, selective node and group reprogramming, and network state monitoring. We used a testbed of 25 Tmote Sky nodes to evaluate the reprogramming performance and space overhead of *Alert* under different network sizes and densities.**

I. Introduction

Natural and man-made disasters threaten human lives, public infrastructure, supply lines, and economic productivity. The damage can have a profound and lasting effect on the population and the economy. Efficient emergency response systems are crucial for reducing the impact of disasters. To be effective, such systems must enlist modern technological advances and use them wisely to mitigate the effects of such scenarios. One technology that is widely available but not yet widely used in emergency response systems is wireless sensor network technology [2], [3].

In the past decade, wireless sensor network (WSN) technology has evolved significantly. As the technology advances and hardware costs continue to fall, WSNs are increasingly used in a range of applications, including surveillance, environmental, industrial, agricultural, and structural monitoring. However, most applications involve static deployments, where large numbers of unattended, resource-constrained computing nodes cooperate on a single application for extended periods of time, e.g., a network of temperature sensors used to monitor the efficiency and status of a heating and cooling system. If a critical condition not part of the original task is detected, the capabilities of these networks cannot be used to assist in responding. Retasking WSNs, capable of changing behavior rapidly, would be beneficial. Remotely distributing new code through a wireless network is a particularly effective approach to retasking because of the behavioral flexibility it affords. Deluge [1] is one of the approaches used to realize this concept.

Deluge is a reliable and robust data dissemination protocol for remotely reprogramming wireless sensor networks. However, it presents several limitations. First, Deluge is limited only to network-wide dissemination of program binaries. Every node within the network stores and runs the same image. Deluge does not allow different program binaries to execute within a network at the same time. However, when an unexpected event occurs, the nodes that are distant from the disaster site should continue their original tasks, limiting the retasking load to nodes near the event site. Selective retasking is important. Second, Deluge does not provide feedback about the state of the network. When an unexpected event occurs, nodes near the disaster site will be reprogrammed to execute new tasks. Feedback from these nodes is required to verify that the desired programs are running properly and that sensor nodes are assisting in the response effort. Network status monitoring is essential.

In this paper, we present *Alert*, a software framework for retasking wireless sensor networks. *Alert* provides node group management, selective node and group reprogramming, and network state monitoring. The main contributions of this paper are as follows. (i) We describe a significant extension to Deluge, enabling wireless sensor network group management, selective node and group reprogramming, and network status monitoring. (ii) We implement a cross-platform Java user interface, *Deluge-Visualizer*, to visually administer Deluge commands and monitor the status of a deployed network. (iii) We evaluate code size overhead and retasking performance using two representative applications. A testbed consisting of 25 Tmote Sky [4] nodes is used to evaluate the code distribution performance of our framework under different network sizes and densities.

The remainder of the paper is structured as follows: Section II provides relevant background material. Section III describes the design and implementation of *Alert*. Section IV presents an evaluation of *Alert* in term of code size overhead and code distribution performance under different network sizes and densities. Section V summarizes key elements of related work. Finally, Section VI concludes with a summary of contributions and pointers to future work.

II. Background

Our work is built on Deluge for TinyOS 2.x [5]. Platforms compatible with this distribution (i.e., Telosb [6], MicaZ [7], Iris [8], and mulle [9]) can use the framework to realize selective retasking and monitoring. In this section, we survey the architecture and dissemination protocol of Deluge.

A. Deluge

Deluge is a nesC module designed to wirelessly distribute and install program binaries within a WSN. Up to four program binaries can be distributed and stored, but only one may be activated (throughout the network) at any time. The Deluge framework is composed of three core components: `DelugeC`, `tos-deluge`, and `TOSBoot`. `DelugeC` contains the core functionality of the Deluge framework, enabling code distribution and reprogramming. `DelugeC` is a non-interactive service added to a user's application and runs parallel to the application logic. `tos-deluge` is a Python script that delivers serial-based commands to a base-station node. A base-station node is a sink node serially-connected to a PC. It receives the program binaries via the serial port, and then disseminates them to the rest of the network. `tos-deluge` is responsible for controlling the binary distribution and reprogramming features of nodes running `DelugeC`. `TOSBoot` is a bootloader executed when a wireless node is reset or powered on. When `TOSBoot` starts execution, it checks to see if the currently loaded application (in program flash) is the desired application based on boot arguments saved to flash. If `TOSBoot` detects the correct application, the application is executed. If the application does not match, `TOSBoot` copies the new application from external flash to the MCU's internal flash. After the copy is complete, `TOSBoot` executes the newly loaded application. Figure 1 illustrates the Deluge `TOSBoot` memory model.

When nodes are configured to use the Deluge framework, they are either programmed as *clients* or *base-stations*. In most cases, a network is composed of only one base-station node and many client nodes. A base-station node is responsible for receiving Deluge commands from the `tos-deluge` Python script via a serial port and sending Deluge commands to client nodes for binary distribution and network reprogramming via radio.

In Figure 2, a sensor network containing a base-station node and four client nodes is configured with the Deluge framework. Sensor nodes that run Deluge will include the `DelugeC` and `TOSBoot` components. Serial Deluge commands are sent from the `tos-deluge` script on a PC to a base-station node. The base-station node receives the serial messages and repackages the Deluge commands for wireless communication to the client nodes.

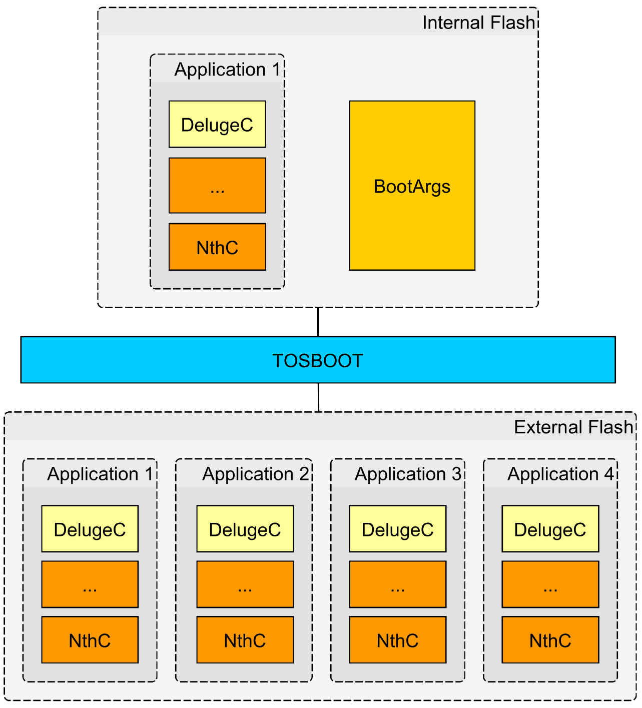


Fig. 1: Deluge TOSBoot Memory Model

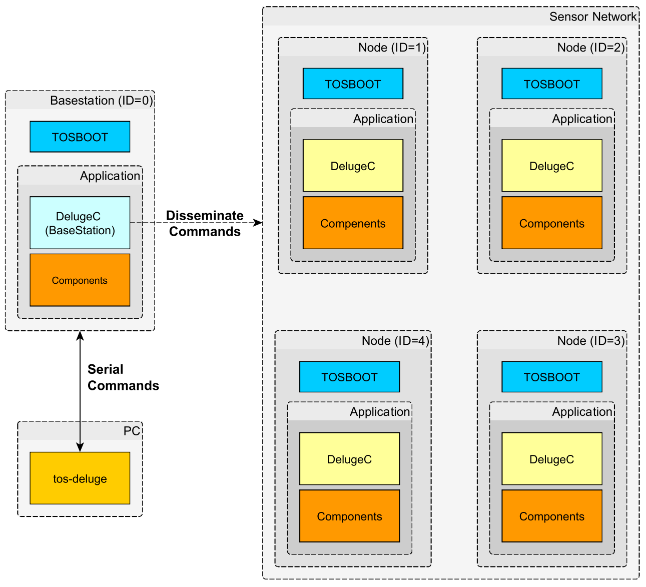


Fig. 2: Deluge Overview

B. Dissemination

The Deluge framework uses the TinyOS Dissemination Protocol to reliably send commands to all client nodes within the sensor network. A protocol called Drip [10], which is based on the Trickle [11] epidemic algorithm, is used. In Deluge, a program binary is represented as a set of small, fixed-size structures called *pages*. Nodes running Deluge periodically broadcast a code summary to their neighbors, advertising the available pages for the currently executing binary. If a node receives an advertisement for an older program binary version than its own, it broadcasts an update. If a node receives a summary identical to its own, it stops broadcasting for a fixed period. In the Drip algorithm, three different node states are defined: `IDLE`, `RECV`, and `PUB`. `IDLE` denotes that a node is running the correct program required by the base-station and is periodically broadcasting its code summary. `RECV` denotes that a node is requesting a page for its incomplete image file. `PUB` denotes that a node is broadcasting its stored image. With

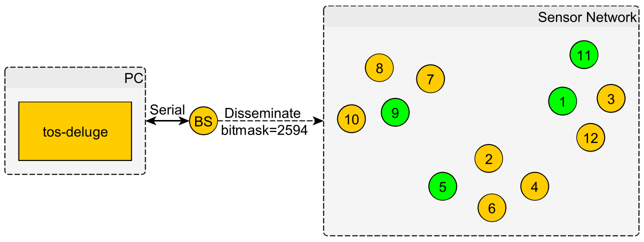


Fig. 3: Retasking Specific Nodes using Node ID

this algorithm, nodes within the network keep their program binaries up to date by transmitting small numbers of messages instead of flooding the network.

III. Design and Implementation

The primary limitations of Deluge include a lack of support for selective network reprogramming and feedback about network status. To mitigate these limitations, selective retasking using node and group identification was integrated, as well as sensor device monitoring using the `Collection` [12] interface. In this section, we present the design and implementation of these features.

A. Selective Retasking using Node ID

To allow users to selectively retask specific nodes within a sensor network, two identification strategies are supported. The first strategy uses each device's `TOS_NODE_ID`, which is typically set during program installation and is used to identify the node in packet transmissions. The `tos-deluge` script is modified to support a selective node ID reprogramming command. When the command is initiated, a node ID bitmask is included with the command message. This 32-bit bitmask supports up to 32 devices, where each bit represents a distinct node ID. The DelugeC component is also modified. When a node running the modified `DelugeC` receives this command, it checks to see if its `TOS_NODE_ID` is set within the mask. If it is, the command is executed; otherwise, the command is ignored.

Figure 3 illustrates a node ID retasking example. The disseminate and reprogram command is initiated using the modified `tos-deluge` Python script. The modified `tos-deluge` sends a serial command to the base-station, which then disseminates the equivalent command to the sensor network. The command is set to disseminate and reprogram with the node ID bitmask field set to 2594 (101000100010). The bitmask corresponds to nodes with `TOS_NODE_ID` set to 1, 5, 9, and 11 (green). These nodes will be reprogrammed with the new image, while the remaining nodes (yellow) will continue to run their current application.

B. Selective Retasking using Group ID

The second retasking strategy assigns each node to a group using a group ID. A new field, `DELUGE_GROUP_ID`, is added to the `DelugeC` component. `DELUGE_GROUP_ID` is typically set in the MAKEFILE of an application and represents the group a node is associated with. It is used to group nodes based on similar tasks or responsibilities and to retask these groups uniformly when an unexpected event occurs. When the disseminate and reprogram command is initiated by the modified `tos-deluge` script, a group ID field is included. When a node running the modified `DelugeC` component receives this command, it checks to see if its `DELUGE_GROUP_ID` matches the group ID within the command. If it is, the command is executed; otherwise, the command is ignored.

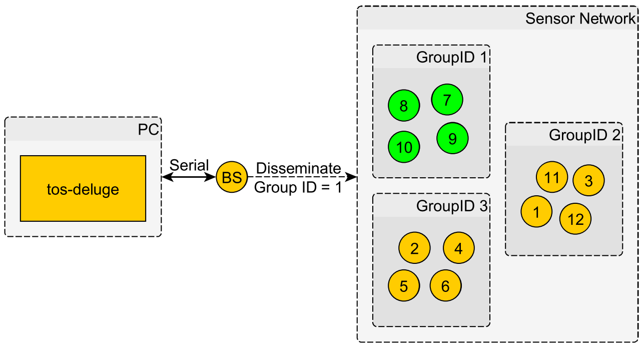


Fig. 4: Retasking Group of Nodes using Group ID

Figure 4 illustrates a group ID retasking example. The disseminate and reprogram command is initiated using the modified `tos-deluge` Python script. The process is analogous to the node ID case. Nodes that have their group ID set to 1 (green) will be reprogrammed with the new image, while the remaining nodes (yellow) will continue to run their current application.

C. Updating Groups

To update node groups post-deployment, the *update group* command was created. Using this command, a sensor network's group topology can be configured dynamically. When the command is initiated, either in *add* or *remove* mode, the node ID bitmask and group ID field are included. When a node running `DelugeC` receives this command, it checks to see if its `TOS_NODE_ID` is set within the mask. If it is, the command is executed; otherwise, the command is ignored. This enables nodes to be added or removed from a particular group.

D. Network Status Monitoring

To provide real-time feedback on a sensor network's operation, the `Collection` component was added to the Deluge framework. `Collection` implements a protocol based on a tree topology using a link estimator to build efficient and reliable routes between client nodes and a base-station. In this context, `Collection` is used to gather information about the sensor network and to deliver node status messages from client nodes to the base-station. Each client node periodically sends a status message containing the following fields: *Node ID* (`TOS_NODE_ID`), *Group ID* (`DELUGE_GROUP_ID`), *State* (`IDLE`, `RECV`, and `PUB`), *App UID* (a unique ID generated when an application is compiled), *App Name* (the currently running application's name), and *App TimeStamp* (denoting when the running application was compiled).

Figure 5 illustrates the updated Deluge framework. The `Collection` component provides a feedback channel (red

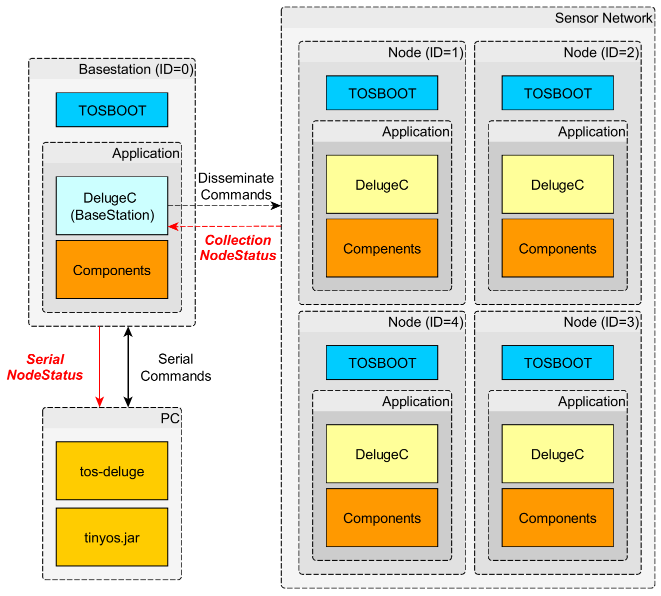
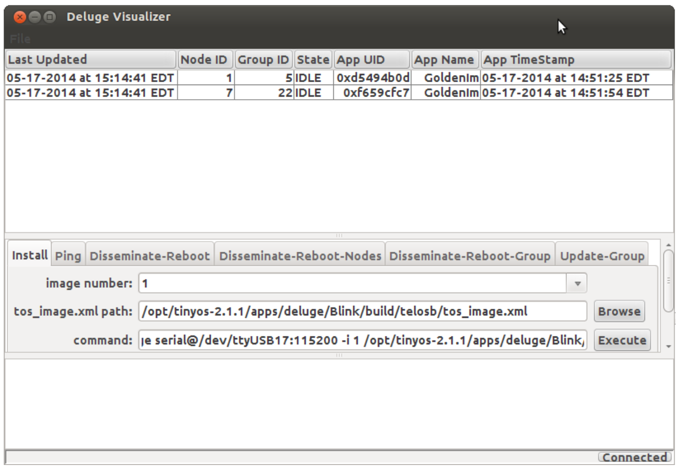


Fig. 5: Modified Deluge Framework with `Collection`



Last Updated	Node ID	Group ID	State	App UID	App Name	App TimeStamp
05-17-2014 at 15:14:41 EDT	1	5	IDLE	0xd5494b0d	GoldenIm	05-17-2014 at 14:51:25 EDT
05-17-2014 at 15:14:41 EDT	7	22	IDLE	0xf659cfc7	GoldenIm	05-17-2014 at 14:51:54 EDT

image number: 1
tos_image.xml path: /opt/tinyos-2.1.1/apps/deluge/Blink/build/telosb/tos_image.xml
command: ͜je serial@/dev/ttyUSB17:115200 -i 1 /opt/tinyos-2.1.1/apps/deluge/Blink/

Fig. 6: Deluge Visualizer

arrow) back to the base-station, and ultimately, a serially-connected PC. The messages are then parsed, processed, and analyzed by the *Deluge Visualizer*, a Java user interface running on the PC, using `tinyos.jar`, the Java SDK for TinyOS. This is detailed in Section III-E.

E. Deluge Visualizer

The *Deluge Visualizer* was developed to automate Deluge command-line tasks, and to monitor and analyze the status of a sensor network. The Visualizer provides a front-end GUI to the `tos-deluge` Python script. Users can easily initiate and verify Deluge commands while monitoring the state of their network. As commands are executed by the user, all status messages and associated information are displayed in the status window, notifying the user of any problems or exceptions. The Visualizer also communicates with the base-station node to receive and parse status messages transmitted by client nodes (via `Collection`). Each status message is presented within a dynamic table that displays each field of the message. As shown in Figure 6, Node 1, associated with Group 5, and Node 7, associated with Group 22, are running the *GoldenImage* application and are in the `IDLE` state. The last times the base-station received status messages from these nodes are shown in the *Last Updated* column. The first tab, *Install*, in the Deluge Visualizer, is used to issue a command to inject program images to a base-station node. The *Ping* tab is used to query the status of the images in external flash (i.e., Program Name, UID, Compilation Time, Platform, User ID, Host Name, User Hash, Size, Number of Pages). The *Disseminate-Reboot* tab is used to disseminate an image (already) in external flash, and to reprogram the network. The *Disseminate-Reboot-Nodes* tab is used to selectively disseminate an image in external flash to specific nodes, and to reprogram them. The *Disseminate-Reboot-Group* tab is used to selectively disseminate the image in external flash to a specific group, and to reprogram the nodes in that group. The *Update-Group* tab is used to update node groups within the network. The Visualizer reduces the learning curve to interact with the Deluge framework and with networks that rely on it.

IV. Evaluation

In this section, we present our evaluation of *Alert*. We first analyze the code size overhead for two representative applications using Tmote Sky motes. We then measure the retasking performance of *Alert* under different network sizes and densities using a physical sensor network testbed. This testbed is tailored to support system debugging, profiling, and experimentation and consists of 25 Tmote Sky sensors accessible via a web interface [13]. The experiments were run on a machine running Ubuntu 12.04, with Linux kernel version 3.2, and TinyOS 2.1.2.

A. Space Overhead

To analyze the code size overhead of *Alert*, the *Blink* application and the *Basestation* application, included in the TinyOS distribution, were used. We first compiled both applications using the original Deluge framework, and then measured the ROM and RAM usage to establish a baseline. We then recompiled the applications using our framework with and without the `Collection` component and measured the ROM and RAM usage again for comparison with the baseline. The results are shown in Table I. Table I(a) summarizes the ROM and RAM usage of the two applications when compiled using the original Deluge framework. Table I(b) summarizes the ROM and RAM usage when compiled with the *Alert* framework, without the `Collection` component. From Table I(b), we can see that the ROM and RAM overhead both increase by approximately 1% when the `Collection` component is not included. Table I(c) summarizes the ROM and RAM usage when the `Collection` component is included in the application. The ROM overhead increases by approximately 20%, and the RAM overhead increases by over 80%. Based on these results, adding network status feedback to the Deluge framework has a high impact on the resulting memory footprint. This is because the `Collection` component requires additional RAM space to support its link estimation and route construction algorithms [4]. Still, the overhead is manageable given the available

Application	ROM (bytes)	RAM (bytes)
Blink	24090	1030
Basestation	29090	1434

(a) Memory Usage of Original Applications

Application	ROM (bytes)	RAM (bytes)	ROM Δ	RAM Δ
Blink	24426	1042	+1.39%	+1.17%
Basestation	29270	1450	+0.62%	+1.11%

(b) Memory Usage with Selective Retasking, w/o `Collection`

Application	ROM (bytes)	RAM (bytes)	ROM Δ	RAM Δ
Blink	29346	2304	+21.81%	123.69%
Basestation	34184	2696	+17.51%	88.00%

(c) Memory Usage with Selective Retasking, w/ `Collection`

TABLE I: Memory Overhead

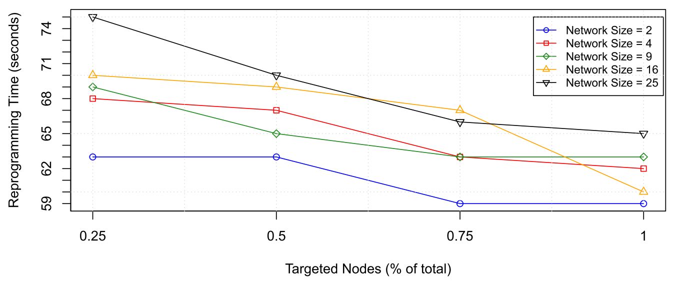


Fig. 7: Group Retasking Time

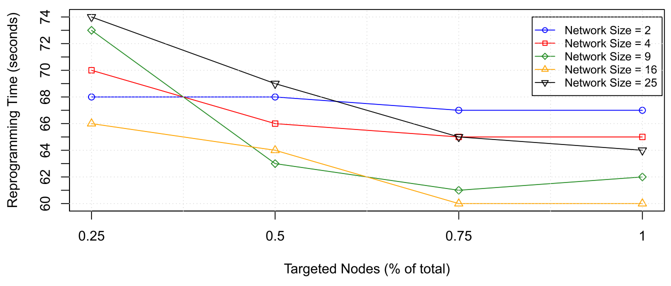


Fig. 8: Node ID Retasking Time

ROM/RAM capacity (48K bytes and 10K bytes, respectively, for Tmote Sky Motes). For many applications and disaster scenarios, the benefits significantly outweigh the memory cost.

B. *Retasking Performance*

We next evaluate the code dissemination and reprogramming performance of the framework as a function of network size and density. A physical testbed consisting of 25 Tmote Sky motes was used. Network sizes of 1, 4, 9, 16, and 25 were considered. Nodes were deployed in a regular grid, with nodes spaced approximately 12 inches apart. To ensure fairness, radio power was set to the maximum for all nodes, allowing single-hop communication for all network sizes considered. To measure code dissemination and reprogramming performance under different network densities, 25%, 50%, 75%, and 100% of the available nodes were targeted for reprogramming in each network setup. The Deluge Visualizer was used to measure code dissemination and reprogramming time. Each time a reprogramming command was sent to the base-station node through the Visualizer, the system time was recorded. After all the target nodes finished receiving (`RECV`) and transitioned to `PUB`, the system time was recorded again. The time spent for code dissemination and reprogramming is the time elapsed between these two timestamps.

Figure 7 summarizes the selective reprogramming performance using group ID as a function of the percentage of targeted nodes in the network. The X-axis represents the percentage of targeted nodes in the network. The Y-axis represents the time required, in seconds, for all targeted nodes to finish retasking. Figure 8 similarly summarizes the selective reprogramming performance using node ID as a function of the percentage of targeted nodes in the network. The X-axis represents the percentage of targeted nodes in the network, and the Y-axis represents the time required for all targeted nodes to finish retasking.

From Figures 7 and 8, we observe that as the size of the network increases, the overall code dissemination and reprogramming time increases. This is because the probability of message collision increases as network size grows, decreasing the performance of the network. However, as the percentage of targeted nodes increases, the overall code dissemination and reprogramming time decreases. This is a surprising result. It is explained as follows: Nodes that are targeted for retasking will transition to `RECV` and stop sending their status to the base-station, decreasing the number of messages in the network, increasing retasking performance. The results reveal an interesting trade-off among network size, percentage of targeted nodes, and retasking performance during unexpected events. Increasing network size offers better coverage for disaster events, but reduces retasking performance, leading to increased response time during emergencies. Increasing the percentage of targeted nodes during retasking offers better performance, but neglects more originally assigned tasks. Knowledge of this trade-off can be used to help WSN managers balance network retasking performance and resource utilization.

V. Related Work

Levis et al. present TinyOS [14], which uses Crossbow Network Programming (XNP) [15][16] as its network reprogramming protocol. XNP transmits a complete program image to nodes within the radio communication range of the base-station. Hui et al. present Deluge [1], which addresses many of the drawbacks of XNP. Deluge supports multi-hop code distribution through epidemic dissemination. Deluge provides redundant data integrity checks and an improved bootloader (TOSBoot), with a lower RAM footprint than XNP [17]. Neither approach can control the scope of code distribution as ours do.

Maté [18] transmits byte code for execution on a virtual machine, enabling significantly faster reprogramming. It is not, however, useful when the virtual machine itself needs to be reprogrammed. Trickle [11] is used to transmit Maté virtual machine scripts, which are significantly smaller. However, Maté scripts are limited to the functions of the virtual machine, and are not as flexible as nesC/C constructs, which are used by our approach.

Stathopoulos et al. present Multihop Over-the-Air Programming (MOAP) [19], which uses a data dissemination protocol called Ripple to distribute code among sensor devices. Ripple selectively forwards packets to nodes while utilizing a sliding window protocol for controlling retransmissions. Nodes have the ability to transmit segments of the program code they have already received to new nodes, while waiting for retransmission of lost packets. Sprinkler [20] and Firecracker [21] both use a hierarchical reprogramming strategy. They first send code updates to nodes in the upper layer of the node hierarchy (i.e., super nodes), and these nodes reprogram other nodes in their local areas. Super nodes can be cluster heads, or a set of connected dominating nodes that can cover the whole network, as in Sprinkler. Super nodes in Firecracker are "corner" nodes, or are randomly selected. None of these approaches consider dissemination scope.

VI. Conclusions

Retaskable WSNs offer an effective and efficient solution to responding rapidly to disaster events. In this paper, we presented *Alert*, a software retasking framework for WSNs, enabling these networks to respond rapidly to unexpected events without neglecting their originally assigned tasks. *Alert* provides node group management, selective node and group reprogramming, and network state monitoring for wireless sensor networks. We presented the design and implementation of these features, as well as a Java user interface used to simplify interaction with the network. We also presented an evaluation of memory overhead and retasking performance under different network sizes and densities. Experimental results show that our framework can efficiently retask wireless sensor networks with tolerable memory overhead.

Our future work spans four paths. The first path is focused on improving the space usage of the framework. As shown in Table I, ROM and RAM overhead is fairly high (though still tolerable) when the `Collection` component is included in the user application to support network status monitoring. To address this problem, the `Collection` component must be optimized or replaced. In the evaluation section, we noticed that the overhead traffic from status messages affected reprogramming performance. This leads us to the second path, which is focused on investigating the trade-offs between status messages size, intervals, etc and reprogramming performance to improve the *Alert*'s performance The third path is focused on modifying the Deluge framework. The current Deluge framework acts as an independent service layer running by itself, offering no external control, which is limiting. It offers no external facing API to be used as a control interface within an application. This would make Deluge a more flexible and reusable library, allowing for looser coupling and tighter integration with application functionality. The third path is focused on removing the dependency on `tos-deluge` to directly command the base-station node from the Deluge Visualizer. Further, if the TinyOS SDK is ported to a smartphone-based operating system, such as Android, Deluge commands can be issued from a tiny mobile device, instead of a PC, offering benefits to emergency responders.

VII. Acknowledgment

This work is supported by the National Science Foundation through awards CNS-1116238, CNS-1116976, CNS-0745846, and CNS-1126344.

References

[1] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *Proceedings of ACM SenSys*, 2004, pp. 81–94.

[2] E. Cayirci and T. Coplu, "SENDROM: Sensor networks for disaster relief operations management," *Wireless Networks*, vol. 13, no. 3, pp. 409–423, 2007.

[3] N. A. A. Aziz and K. A. Aziz, "Managing disaster with wireless sensor networks," in *Proceedings of the International Conference on Advanced Communication Technology (ICACT)*, 2011, pp. 202–207.

[4] Moteiv. Tmote sky. [Online]. Available: http://www.eecs.harvard.edu/~konrad/projects/shimmer/references/tmote-sky-datasheet.pdf

[5] C. Liang and R. Musaloiu. Deluge t2. [Online]. Available: http://tinyos.stanford.edu/tinyos-wiki/index.php/Deluge_T2

[6] MEMSIC. Telosb mote. [Online]. Available: http://www.memsic.com/userfiles/files/Datasheets/WSN/telosb_datasheet.pdf

[7] MEMSIC. Micaz mote. [Online]. Available: http://www.memsic.com/userfiles/files/Datasheets/WSN/micaz_datasheet-t.pdf

[8] MEMSIC. Iris mote. [Online]. Available: http://www.memsic.com/userfiles/files/Datasheets/WSN/IRIS_Datasheet.pdf

[9] EISTEC. Mulle mote. [Online]. Available: http://www.eistec.se/docs/Mulle_Expansion_Board_User_Manual.pdf

[10] P. Levis and G. Tolle. (2004) Dissemination of small values. [Online]. Available: http://www.tinyos.net/tinyos-2.x/doc/html/tep118.html

[11] P. A. Levis, N. Patel, D. Culler, and S. Shenker, "Trickle: A self regulating algorithm for code propagation and maintenance in wireless sensor networks," in *Proceedings of NSDI*, 2004.

[12] R. Fonseca, O. Gnawali, K. Jamieson, and P. Levi. (2006) Collection. http://www.tinyos.net/tinyos-2.x/doc/html/tep119.html.

[13] A. R. Dalton and J. O. Hallstrom, "An interactive, source-centric, open testbed for developing and profiling wireless sensor systems," *International Journal of Distributed Sensor Networks*, vol. 5, no. 2, pp. 105–138, 2009.

[14] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer *et al.*, "Tinyos: An operating system for sensor networks," in *Ambient Intelligence*. Springer, 2005, pp. 115–148.

[15] Crossbow Technology, Inc. Mote in-network programming user reference. http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/Xnp.pdf.

[16] J. Jeong, S. Kim, and A. Broad. Network programming. http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/Xnp.pdf.

[17] J. Hui. (2004) Tinyos network programming. http://www.cs.virginia.edu/~jwang/STIL_files/Devices/deluge-manual.pdf.

[18] P. Levis and D. Culler, "Maté: A tiny virtual machine for sensor networks," in *Proceedings of ASPLOS X*, vol. 37, no. 10, 2002, pp. 85–95.

[19] T. Stathopoulos, J. Heidemann, and D. Estrin, "A remote code update mechanism for wireless sensor networks," UCLA Center for Embedded Networked Sensing, Tech. Rep., 2003.

[20] V. Naik, A. Arora, P. Sinha, and H. Zhang, "Sprinkler: A reliable and energy efficient data dissemination service for wireless embedded devices," *IEEE Transactions on Mobile Computing*, vol. 6, no. 7, pp. 777–789, July 2007.

[21] P. Levis and D. Culler, "The firecracker protocol," in *Proceedings of the ACM SIGOPS European Workshop*, 2004.