**Re-tasking Wireless Sensor Networks**

**Michael Ruffing**

**ECE 698 Master’s Project**

**Introduction**

Wireless sensor networks have evolved from a theoretical concept to an industry standard. As technology has advanced and hardware costs have fallen wireless senor networks have become economically beneficial and widespread. Wireless sensor networks are used in many applications which include surveillance, environmental, industrial, agricultural, and structural monitoring. Most applications of wireless sensor networks are considered static deployments, where the sensors are configured for a specific task and never change. For example a network of temperature sensors used to monitor the efficiency and status of a heating and cooling system. In contrast a re-taskable wireless sensor network can reconfigure itself to dynamically address the ever changing constraints of its environment. For example a building containing a network of cameras originally designed for security related monitoring could be re-tasked during a fire evacuation to assist first responders to people left in the building.

The focus of this project is to develop and implement a re-tasking architecture for wireless sensor networks using the TinyOS operating system and the Deluge framework. The goal of this project is to extend the Deluge framework to implement a reliable re-tasking architecture for wireless sensor networks.

**Project Goals**

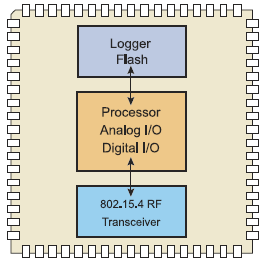
* Evaluate Deluge 2.0 for re-tasking capabilities
* Develop distributed re-tasking architecture using Deluge 2.0
  + Network-wide - disseminate new task to entire network of sensors
  + Selective - disseminate new task to a select group of sensors
* Develop monitoring framework and application
  + Displays real-time information including sensor states and running tasks
  + Allows end-user to initiate network wide and selective re-tasking

**TinyOS Overview**

TinyOS is an event driven operating system designed for low-power wireless devices, specifically sensor networks. TinyOS is written in nesC, a C like programming language that is designed for structured component based applications. Applications written in nesC are built using interfaces and components that promote hardware abstraction and reuse. Components are “wired” together using configuration modules that link specific implementation to dependencies. TinyOS applications are compiled into a single binary (no shared libraries), only including required components and their dependencies. Using this approach reduces application program size and overall memory footprint, which is usually a priority for embedded devices.

The TinyOS operating system is open source and actively developed and supported by academia and industry. TinyOS currently supports many platforms including the IRIS mote. Each new release of TinyOS adds new and updated support for numerous wireless sensor based platforms. The TinyOS community has established working groups for documenting and defining new TinyOS standards called TinyOS Enhancement Proposals (TEP). As of this writing there are over 30 TEPs defining protocols and components of the TinyOS operating system. The TinyOS community also maintains an online Wiki that documents and educates users on TinyOS implementation and application. For this project TinyOS 2.1.2 (August 20, 2012) was used.

**IRIS Mote Platform**

**  
Figure 1 – Iris Mote and Block Diagram**

This project used three IRIS mote devices to develop and test re-tasking and monitoring capabilities within a wireless sensor network. The IRIS mote is a 2.4 GHz Mote module used for enabling low-power, wireless sensor networks. Illustrated in Figure 1 the IRIS mote is composed of three main hardware components: processor, radio transceiver, and external flash (logger). The processor is based on the Atmel Atmega1281 low-power 8-bit microcontroller with 128K bytes of internal flash memory. The radio module is a 2.4 GHz IEEE 802.15.4 compliant transceiver capable of a 250kbps data rate and 500 meter line-of-sight communication. The IRIS’s external flash module supplies the mote with additional 512K bytes of external memory.

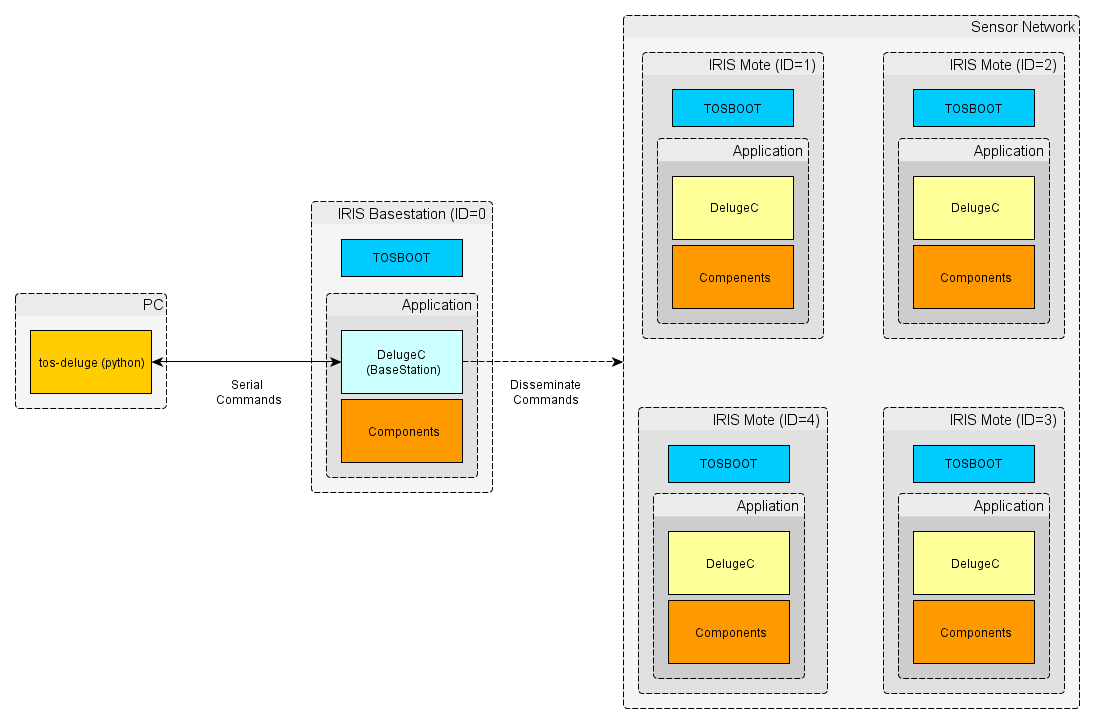
**Deluge Overview**

The Deluge TinyOS component is defined as “an efficient protocol for disseminating large data objects, such as program binaries, to many nodes within a wireless sensor network [1].” Deluge was first described in a paper written by students at Berkeley during the fall of 2003. Deluge’s first Beta version was released in April 2004 and then subsequent releases up to Deluge 2.0 in July of 2005 [6]. At some point the Deluge code base was ported and combined with TinyOS 2.X and titled Deluge T2. The current version of Deluge is 2.1.2 which is packaged and versioned with TinyOS 2.1.2 (August 20, 2012).

Deluge is a nesC based reusable component designed to wirelessly distribute and program binaries within a sensor network. Deluge allows up to four program binaries to be distributed and installed on neighboring motes within a sensor network. The Deluge framework is composed of the following core software components:

* **DelugeC** – The core functionality of the Deluge framework. DelugeC is a nesC component that is added to a user’s application to enable program distribution and reprogramming. DelugeC is a non-interactive service that runs parallel to a user’s application logic.
* **tos-deluge** – A python script that delivers serial based commands to BaseStation configured motes. Tos-deluge is responsible for controlling, querying, and commanding the binary distribution and reprogramming features of the Deluge framework. Specifically tos-deluge injects and erases TinyOS application within the BaseStation mote. Tos-deluge is also responsible for initiating the dissemination and reprogram of the sensor network nodes.

When motes are configured to use the Deluge framework they are either programmed as client or BaseStation motes. By default an application is configured as a Deluge client when it includes the DelugeC component. If an application is compiled with the DELUGE\_BASESTATION flag it is configured as a BaseStation mote. BaseStation motes are responsible for receiving serial commands from the tos-deluge python script. BaseStation motes send Deluge commands and client motes receive the commands. In most cases a sensor network is composed of many client motes and only one BaseStation mote to send Deluge commands for binary distribution and network reprogramming.



**Figure 2 – Deluge Overview**

Figure 2 illustrates a basic IRIS based sensor network configured with the Deluge framework. There is one BaseStation mote and four client motes. Serial Deluge commands are sent between the tos-deluge script from a PC and the single BaseStation mote. The BaseStation mote receives the serial messages and repackages the Deluge commands for wireless communication to the IRIS client motes. Communication between the BaseStation and client motes is done through the TinyOS Dissemination protocol.

**Dissemination**

The Deluge framework uses the TinyOS dissemination protocol to reliably send commands to all client motes of the sensor network. The dissemination protocol is defined in the TEP standard 118. The standard defines dissemination as a mechanism to “reliably deliver a piece of data to every node [2].” Deluge commands are first instantiated and injected into the BaseStation mote using the tos-deluge python script. The BaseStation mote then disseminates the Deluge command by calling a dissemination update command. All client motes in the sensor network then receive a value changed event containing the updated Deluge command. Based on the dissemination implementation various strategies are used to update all client motes within the network. The Deluge framework uses the Drip implementation which is based on a trickle epidemic algorithm [3].

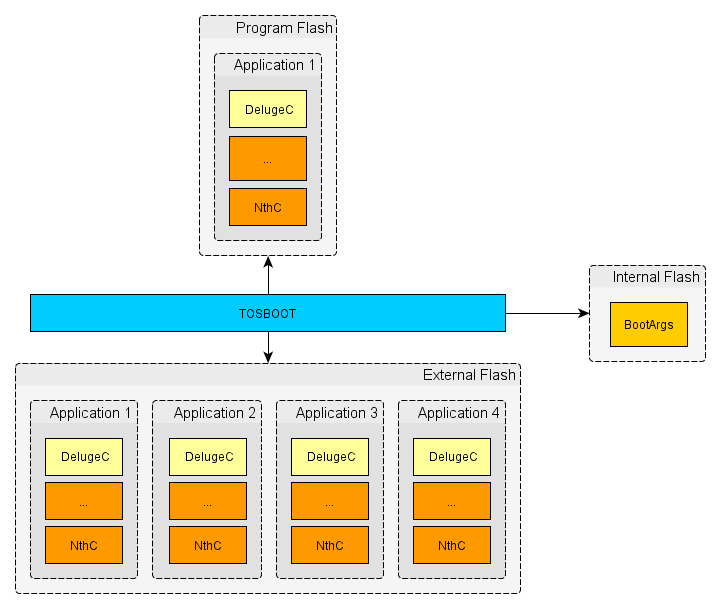
**Deluge Architecture**

As illustrated in Figure 2 the Deluge framework is composed of two core software modules; TOSBoot (blue rectangle) and the DelugeC component (yellow rectangle). The following sections explain each module and their responsibilities within the Deluge architecture.

**Deluge Architecture - TOSBoot**

TOSBoot is a bootloader application that is executed first when the IRIS mote is reset or powered on. When TOSBoot first starts it checks to see if the currently loaded application (program flash) is the desired application based on boot arguments saved to external flash. If TOSBoot detects the correct application is loaded (based on image address) then the application is executed. However, if the loaded application does not match, then TOSBoot copies the new application from external flash to the internal program flash. After the copy is completed TOSBoot executes the newly loaded application.

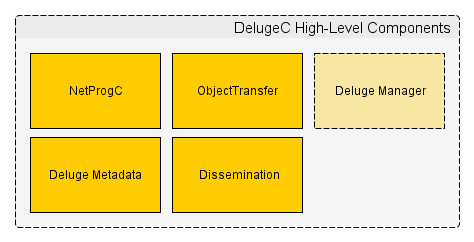
TOSBoot is also responsible for detecting user input to load the GoldenImage. If a user quickly presses the IRIS reset button three times then the TOSBoot will reprogram the mote to the GoldenImage. The GoldenImage is designated as a stable and predictable image that the device can rollback to in case of a malfunctioning or corrupted application. The GoldenImage is located within one of the four external application slots and designated at compile time using the GoldenImage tag in the volumes-at45db.xml file.



**Figure 3 - Deluge-TOSBoot Memory Model**

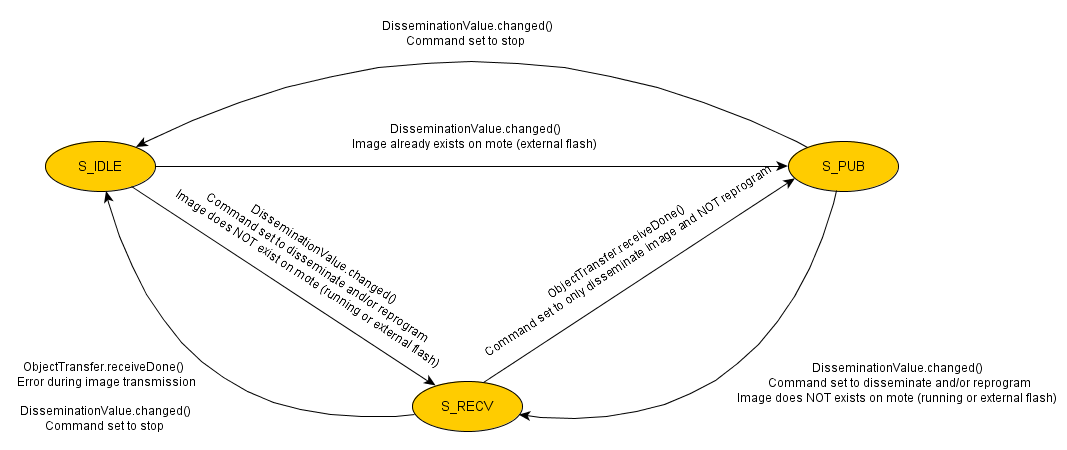
Illustrated in Figure 3 is the Deluge TOSBoot Memory model. The TOSBoot application moves application code between external flash and program flash. The application that runs at boot startup is defined by the BootArgs structure stored within the mote’s internal flash. When the IRIS mote boots up the TOSBoot application runs and reads from internal flash accessing the BootArgs structure for the application address.

**Deluge Architecture – DelugeC Components**



**Figure 4 – DelugeC’s High-Level Components**

Illustrated in Figure 4 the DelugeC component is composed of 5 modules. Each module performs specific tasks and operations in the overall goal of disseminating and reprogramming binaries within the sensor network. Each component is detailed further in the subsequent sections. The DelugeC component has three states: IDLE, RECV, and PUB. When in the IDLE state, DelugeC is waiting for any new Deluge commands. If the mote receives the disseminate and reprogram command then the mote either goes into the RECV (receive) or PUB (publish) state based on its installed images and currently executing application. Figure 5 is a state diagram that illustrates all the possible state transitions, events, and resulting actions for the DelugeC component.

****

**Figure 5 – DelugeC State Diagram**

**Deluge Architecture – DelugeC - Dissemination**

The Dissemination component of Deluge is responsible for sending and receiving Deluge command messages. The Deluge command messages are used to initiate and execute the reprogramming of the sensor network. Applications compiled using the DELUGE\_BASESTATION flag, update and initiate the dissemination of the Deluge command payload. Deluge components compiled as clients receive the updated Deluge command payload and execute the new command accordingly (i.e. reprogram/reboot).

**Deluge Architecture – DelugeC - ObjectTransfer**

The ObjectTransfer component is an ActiveMessage (AM) based service used for sending and receiving program binaries. The ObjectTransfer service operates within three states: idle, publishing, and receiving. When the ObjectTransfer component is IDLE there is no activity; the mote is neither sending nor receiving. When the mote first comes up the ObjectTransfer is in the IDLE state. Setting the ObjectTransfer to the receive state allows the mote to receive program binaries. When the ObjectTransfer is in the publishing state the mote is available for sending a program binary to neighboring motes. The ObjectTransfer sends and receives binaries by breaking them up into manageable pages. Before two motes exchange a program binary the size and number of pages are shared to ensure optimal, reliable, and valid image transfer.

**Deluge Architecture – DelugeC - DelugeMetadata**

The DelugeMetadata component is used to read the mote’s external flash for information regarding the stored images. In the process of reprogramming a mote the DelugeMetadata is used to determine if an application already exists on the device. This is accomplished by comparing the metadata about the stored application with the image being disseminated. The application metadata includes the following: unique ID, size, number of pages, CRC, application name, user who compiled application, user hash, and the machine name that compiled the application.

**Deluge Architecture – DelugeC - NetProg**

The NetProg component is responsible for reprogramming and rebooting the motes. Since Deluge supports multiple platforms (IRIS, micaz, etc.) the NetProg components provides a level of abstraction between the hardware layer and programming interface. The NetProg component allows the Deluge framework to hardware reset the device using a hardware independent interface. In order to reprogram the device NetProg component first writes the BootArgs structure to internal flash and then initiates a hardware reset. Once the device reboots the TOSBoot bootloader reads the BootArgs from internal flash and executes the new application (see **Deluge Architecture - TOSBoot** section for more details about the boot process).

**Deluge Architecture – DelugeC - DelugeManager**

The DelugeManager component provides the Deluge framework with serial communication between the mote and connecting device (Figure 2). More specifically the DelugeManager receives serial commands from the tos-deluge python script and then instantiates Deluge commands for network-wide dissemination. The DelugeManager is responsible for mapping all serial messages to equivalent Deluge commands. The final commands are then disseminated network-wide using the Dissemination component. This component is only available when the mote is configured as a BaseStation.

**Deluge Limitations**

Deluge is a reliable and robust framework for wirelessly reprogramming a sensor network. However, there are several limitations with the current implementation (2.1.2).

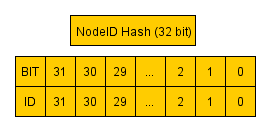
Deluge is limited only to network-wide dissemination of program binaries. When a user wants to issue a new program binary within the sensor network Deluge executes a network-wide dissemination. Every mote within the network will store and run the same image. Deluge does not allow for different program binaries executing within the network at the same time. The ability to selectively re-task certain motes is not supported within the Deluge framework.

Deluge does not provide feedback about the state of the sensor network. When a network is reprogrammed there is no feedback channel to verify that the motes are running the desired program binary. There is also no way of monitoring the current state of each device. The only available option is to monitor the LED status of each mote, which is an explicit implementation requirement that an application must include.

**Overcoming Deluge Limitations**

In order to mitigate the limitations of the Deluge framework the following features were added to the Deluge framework: selective re-tasking using node and group identification, sensor device monitoring using the Collection interface.

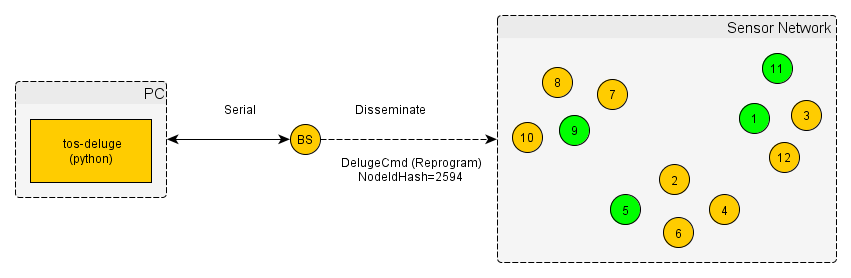
**Selective Re-tasking using Node ID Hash**



|  |  |
| --- | --- |
| **Node IDs Set** | **Node ID Hash (32-Bit)** |
| 1,2,3,4,5 | 62 |
| 29,7,18 | 537133184 |
| 10 | 1024 |
| 31 | 2147483648 |

**Figure 6 – Node ID Hash Variable**

In order to allow users to selectively re-task certain motes within the sensor network two identification strategies were implemented. The first strategy uses the device’s TOS\_NODE\_ID combined with a nodeID hash. When the disseminate and reprogram command is initiated a nodeID hash is included with the command message. The nodeID field is a 32 bit hash that supports up to 32 devices. Each bit of the hash represents a node ID, which is illustrated in Figure 6. When a mote receives the new command it first checks to see if it’s TOS\_NODE\_ID is set within the hash. If the bit is set then the command is executed, otherwise the command is ignored. When a command is ignored the mote has determined that it was not part of this current re-tasking phase.

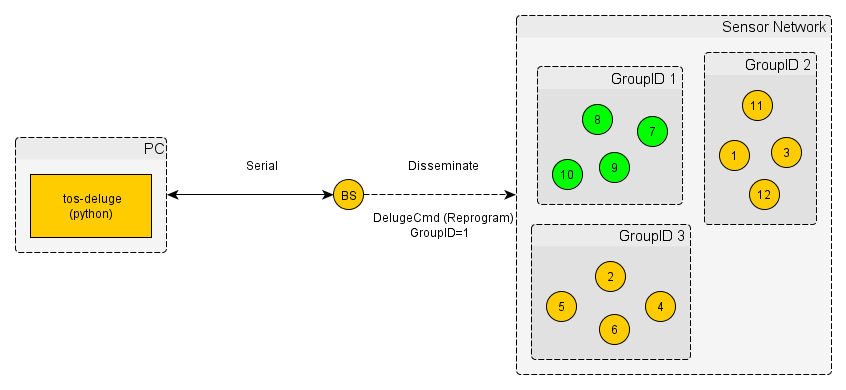


**Figure 7 – Re-tasking Specific Motes**

Illustrated in Figure 7 is a nodeID hash re-tasking example. The disseminate and reprogram command is initiated using the tos-deluge python script. Tos-deluge sends a serial command to the BaseStation, which then disseminates the equivalent DelugeCmd to the sensor network. The DelugeCmd is set to disseminate and reprogram with the nodeID hash field set to 2594. The nodeID hash corresponds to motes with TOS\_NODE\_ID set to 9, 5, 1, and 11 (Green). These motes will be reprogrammed with the new image while the rest of the motes (Yellow) will remain running their current application.

**Selective Re-tasking using Group ID**

The second re-tasking strategy assigns each mote to a group using the DELUGE\_GROUP\_ID. This allows motes to be grouped together based on similar tasking or responsibility within the sensor network. When the disseminate and reprogram command is initiated a groupID field is included with the command message. When a mote receives the new command it first checks to see if it’s DELUGE\_GROUP\_ID matches the group ID within the command. If there is a match the command is executed, otherwise the command is ignored.

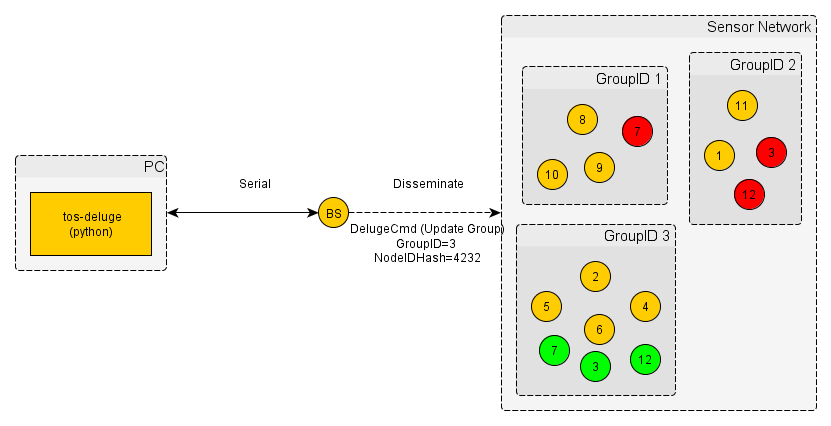


**Figure 8 – Re-tasking Group of Motes**

Illustrated in Figure 8 is a groupID re-tasking example. The disseminate and reprogram command is initialed using the tos-deluge python script. Tos-deluge sends a serial command to the BaseStation, which then disseminates the equivalent DelugeCmd to the sensor network. The DelugeCmd is set to disseminate and reprogram with the groupID field set to 1. Motes (Green) that have their groupID set to 1 will be reprogrammed with the new image while the rest of the motes (Yellow) will remain running their current application.

**Updating Groups**

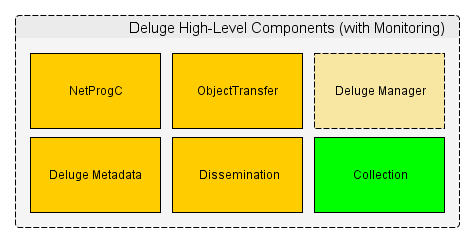
In order to have the ability to update mote groups after compile time the update group command was created. Using the update group command a sensor network’s group topology can be configured dynamically during live operation. When the update group command is initiated the NodeID hash and groupID field is included. Both the nodeID hash and groupID fields enable motes to be added or removed from a particular group.



**Figure 9 – Updating/Deleting Mote(s) from Group(s)**

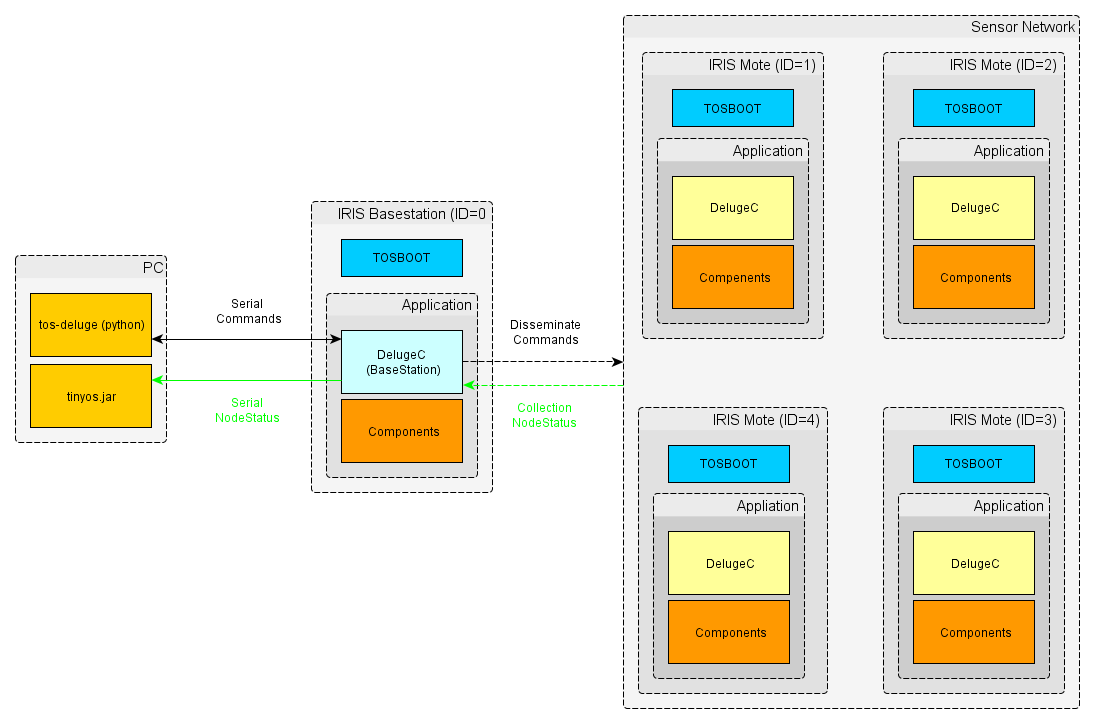
Illustrated in Figure 9 is an example of adding and removing motes from a specific group. The update group command is initiated by the tos-deluge python script. Tos-deluge sends a serial command to the BaseStation, which then disseminates the equivalent DelugeCmd to the sensor network. The DelugeCmd is set to update group with the groupID field set to 3 and the nodeID hash set to 4232. The motes corresponding to the NodeID hash (7, 3, and 12) are removed from their current group and added to GroupID 3. Figure 9 illustrates this exchange with the Green circles representing the new additions to groupID 3 and Red circles representing the removal of motes.

**Collection (Feedback from the Sensor Network)**

****

**Figure 10 - DelugeC’s High-Level Components with Collection**

In order to provide feedback about the sensor network the Collection component was added to the Deluge framework (Illustrated in Figure 10). Deluge Collection is used to gather information about the sensor network; client motes send information to the BaseStation mote. Collection is based on the TEP 119 which documents and defines the collection protocol within the TinyOS operating system. “Collection provides a best-effort, multihop delivery of packets to the root of a tree [4].” The Collection protocol is based on a tree architecture that uses a link estimator algorithm to build efficient and reliable routes between client nodes and a BaseStation. Deluge uses the collection protocol to efficiently and reliably deliver NodeStatus messages from the client nodes to the BaseStation mote. Each client mote periodically sends a NodeStatus message which contains the following fields: Node ID, Group ID, State, and the current running application’s unique ID, name, and compile timestamp. The NodeStatus structure is limited to a size of 20 bytes. This limitation is due to the Collection protocol not allowing more than 20 bytes for the payload (IRIS platform specific).

****

**Figure 11 – Deluge with Monitoring Overview**

Figure 11 represents the updated Deluge framework including monitoring. The original Deluge framework only pushed information and commands into the sensor network. Adding the collection component provides a feedback channel (green arrow) back to the BaseStation and ultimately a serially connected PC. NodeStatus messages are sent from the senor network, through the BaseStation, and back to the PC. Using TinyOS’s SDK the NodeStatus message are parsed, processed, and analyzed by a JAVA application running on the PC. Section **Deluge Visualizer** details the JAVA application developed for this project.

**Deluge Code Sizes**

|  |  |  |
| --- | --- | --- |
| **Application** | **ROM (Bytes)** | **RAM (Bytes)** |
| Blink (Deluge) | 26074 | 945 |
| Base station (Deluge) | 32544 | 1300 |

**Table 1 - Original Deluge (2.1.2)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Application** | **ROM (Bytes)** | **RAM (Bytes)** | **ROM** Δ | **RAM** Δ |
| Blink (Deluge) | 26666 | 957 | +2.3% | +1.3% |
| Base station (Deluge) | 32840 | 1315 | +1.0% | +1.2% |

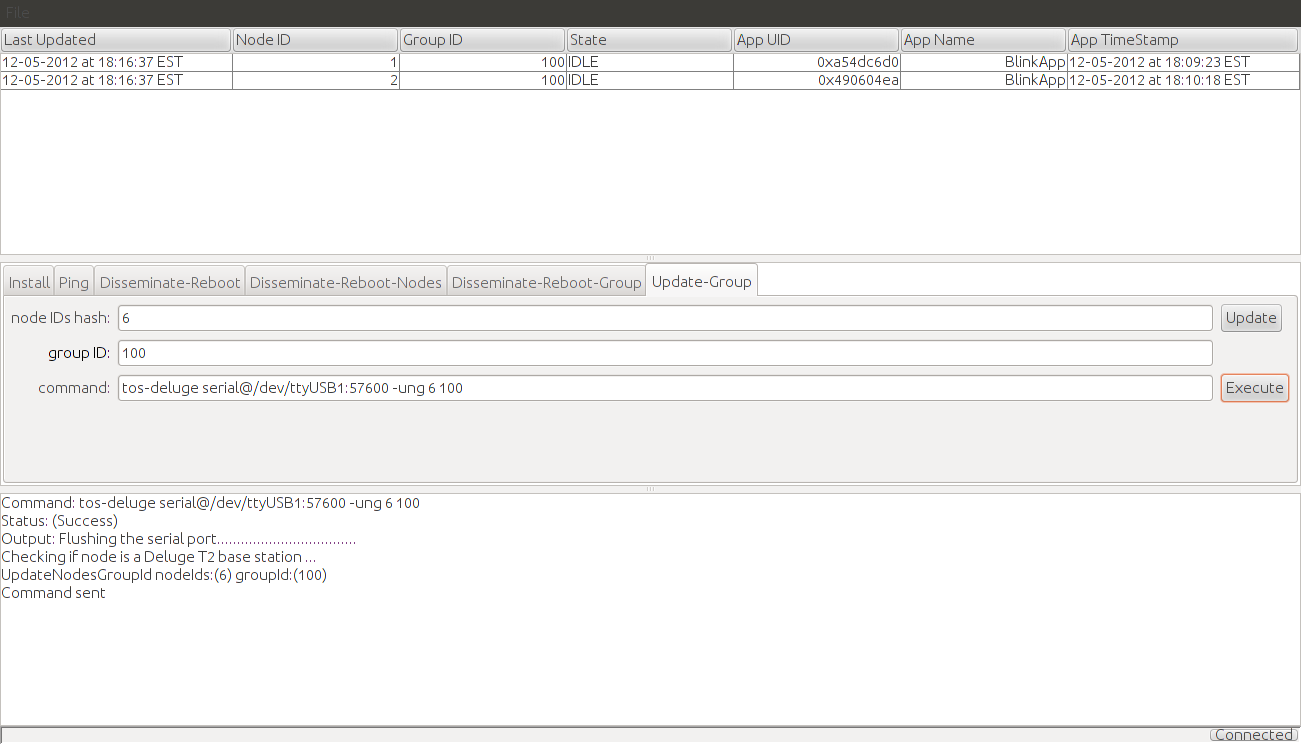
**Table 2 - Selective Re-tasking changes (w/o Monitoring)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Application** | **ROM (Bytes)** | **RAM (Bytes)** | **ROM** Δ | **RAM** Δ |
| Blink (Deluge) | 33294 | 2127 | + 28% | 125% |
| Base station (Deluge) | 39614 | 2463 | + 22% | 89% |

**Table 3 - Deluge Re-tasking changes including Monitoring**

Illustrated in Tables 2-3 are the code size results compared against Table 1 the original Deluge code base. The second table details the ROM and RAM size of the selective re-tasking changes, not including the Collection component (feedback). Overall the selective re-tasking code changes added less than 3% to the program size (ROM) and variable allocation (RAM). The next section shows the memory size when the Collection component is added to the Deluge framework. The program size increased over 30% and variable allocation (ram) increased over %100. Based on these results adding node status feedback to the Deluge framework has a significant impact on the resulting memory footprint. The Collection protocol doubles the RAM usage which is attributed to array and variable allocation to support the link estimation and route building algorithms [4].

**Deluge Visualizer**



**Figure 12 - Deluge Visualizer**

The Deluge Visualizer was developed to automate Deluge command line based tasks and most importantly to monitor and analyze the status of a sensor network. The Deluge Visualizer provides a frontend GUI to the tos-deluge python script. It also provides a single point to administer Deluge commands and monitor the status of a sensor network. Users can easily initiate and verify Deluge commands while monitoring the state of the sensor network. As commands are executed by the user all status messages and information are displayed in the status window, notifying the user of any problems or exceptions. The Deluge Visualizer also serially communicates with the BaseStation mote to receive and parse NodeStatus messages that are transmitted by the client motes (via Collection). Each node status message is presented within a dynamic table of the Deluge Visualizer application. The dynamic table displays each field of the NodeStatus message as a column in the table (Figure 12). The Deluge Visualizer simplifies and reduces the learning curve to interact with the Deluge framework.

The Deluge Visualizer is a cross-platform swing based application developed using the Java programming language. The application is built using the TinyOS Java SDK and MIG tool. The TinyOS SDK is used to receive and deserialize TinyOS serial message objects within the application. The MIG tool is a code generating tool that builds language specific serialization code mapping between a TinyOS message structure and a language specific message structure. For this project the MIG tool was used to generate a Java equivalent NodeStatus message based on the structure definition found in the Deluge framework. Not only are fields generated but also specific primitive type conversions and type casting code.

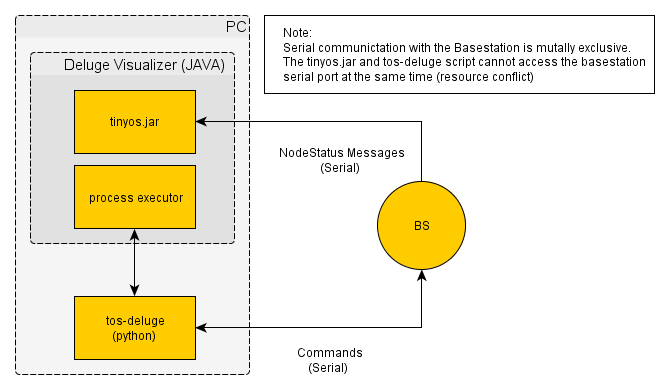


Figure 13 - Deluge Visualizer Components and Interaction

Figure 13 illustrates the Deluge Visualizer’s basic components and interaction with the Deluge framework. The Deluge Visualizer uses a process executor class to handle user input and commanding between the frontend GUI and the tos-deluge python script. Once all conditions of the command have been met and the user initiates a execute command the Deluge Visualizer calls the equivalent tos-deluge command with the appropriate arguments. The tos-deluge script then sends the commands to the BaseStation for dissemination to the sensor network. The tinyos.jar is the TinyOS Java SDK used by the Deluge Visualizer to receive and handle NodeStatus messages received from the BaseStation. As detailed in the **Collection** section the collection component is used to gather NodeStatus messages from the client motes and deliver them to the BaseStation mote. Once the BaseStation receives the NodeStatus message it is then sent serially to the Deluge Visualizer for display.

**Development Tools and Debugging Process**

In order to efficiently and adequately develop a selective re-tasking framework multiple development and software tools were used. The Mercurial source control system was used to maintain and document code changes. Each changeset to the Deluge code base was accompanied by a detailed commit message. Chiliproject, a web based project management system, was used to store project notes and specifics relating to TinyOS, nesC, and Deluge. All notes stored in Chiliproject use a wiki syntax and are easily shared to other contributors or interested personnel.

All development and debugging was conducted on a laptop running Ubuntu 12.04. Some development was also completed on a Windows 7 PC running a virtualized Ubuntu 12.04 using VirtualBox. Both environments successfully ran TinyOS applications and tools without any issues.

The process of debugging a TinyOS application can be very limited and rudimentary. Listed below are the various debugging methods used for this project:

* **Printf** – The printf library provides the ability to write messages to the mote’s serial port. In most cases the library worked without any issues, but for some unknown reason the printf library conflicted with BaseStation motes. The conflict is probably due to the serial communication requirement of the DelugeManager component. Both the printf library and DelugeManager component are competing for the serial port resource.
* **TOSSIM** – TOSSIM is a python based simulator that allows TinyOS applications to be executed independent of any mote hardware. The application executes within a simulated environment on the host computer (PC). Debug messages (similar to printf) can be compiled into the application and written to standard out and/or a logging file during the simulation. TOSSIM works well with unit testing specific methods or algorithms but does not support the simulation of Deluge program binary dissemination.
* **LEDS** – The last strategy to debug a TinyOS application is through the mote’s three LEDs. Users can turn on/off LEDS based on certain conditions or states of the device. The LEDS strategy was used as a last resort for debugging and validation.

**Conclusion**

All project goals were successfully completed which includes: selective re-tasking using the Deluge framework and architecture, sensor network feedback, and a monitoring application. Two selective re-tasking implementations were developed; re-tasking using node and group identification. A monitoring framework was developed through the addition of the collection component and the Deluge Visualizer application. The Deluge Visualizer provides feedback of the sensor network, validating the effectiveness and operation of the Deluge framework. Also, all original Deluge functionality remains valid, fully functional, and compatible with all the new changes.

There are many aspects of this project that can be further improved and researched. The selective re-tasking functionality needs to be tested with a larger network of motes (compared to only 3 motes). There are no results on how the system operates as the network scales. Further development is needed to reduce and improve the memory footprint of the Deluge code changes described in Tables 1-3. However, as technology advances and hardware costs go down this may not be a significant issue. The tos-deluge python script dependency should be removed in favor of direct commanding between the Deluge Visualizer and the BaseStation mote. This can be achieved by porting the python based serial commands to Java. In an effort to stay relevant to current mobile trends the TinyOS SDK should be ported to a smartphone based operating system, such as Android. This would allow a user to issue Deluge commands from a small portable device.

The Deluge framework contains all the functionality and features needed to reprogram a wireless sensor network. However, the Deluge framework is very limited and inflexible on how it’s integrated into a TinyOS application. It is very hard to use as a library within an application. Deluge has no external facing API to allow an application to control its state or modify its transitions. Currently, Deluge acts more like an independent service layer that runs by itself, with no outside control. Deluge can only be extended and controlled from within its own component. Due to these limitations, many applications cannot use the Deluge framework as a solution to disseminate program binaries. An external facing interface should be developed to allow for tighter integration into an application’s functionality and overall requirements. This would transition Deluge into a more flexible and reusable library.

**References**

[1] Hui, Jonathan. Deluge 2.0 – TinyOS Network Programming. July 28, 2005.

[2] http://www.tinyos.net/tinyos-2.x/doc/html/tep118.html

[3] Philip Levis, Neil Patel, David Culler, and Scott Shenker. "Trickle: A Self-Regulating Algorithm for Code Maintenance and Propagation in Wireless Sensor Networks." In Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004)

[4] <http://www.tinyos.net/tinyos-2.x/doc/html/tep119.html>

[5] <http://docs.tinyos.net/tinywiki/index.php/Main_Page>

[6] <http://www.cs.berkeley.edu/~jwhui/deluge/index.html>