Self-Organization in Autonomous Sensor/Actuator Networks

[SelfOrg]

Dr.-Ing. Falko Dressler

Computer Networks and Communication Systems
Department of Computer Sciences
University of Erlangen-Nürnberg

http://www7.informatik.uni-erlangen.de/~dressler/
dressler@informatik.uni-erlangen.de
Overview

- **Self-Organization**
  Basic methodologies of self-organization; comparison of central and hierarchical control, distributed systems, and autonomous behavior; examples of self-organization

- **Mobile Sensor/Actuator Networks**
  Ad hoc routing; reliable communication and congestion control; sensor assistance for mobile robots; applications

- **Coordination of Autonomous Systems**
  Coordination and synchronization; communication aspects; clustering

- **Bio-inspired Mechanisms**
  Swarm intelligence; artificial immune system; intra/inter cellular information exchange
Case Study: Mica2 + TinyOS + Surge

- **Hardware**
  - Mica2 / Mica2dot
  - Sensor boards

- **Programming environment (aka operating system)**
  - TinyOS
  - nesC

- **Sensor application**
  - Surge
The MICA Architecture

- Atmel ATMEGA103
  - 4 Mhz 8-bit CPU
  - 128KB Instruction Memory
- 4KB RAM
- 4 Mbit flash (AT45DB041B)
  - SPI interface
  - 1-4 uj/bit r/w
- RFM TR1000 radio
  - 50 kb/s – ASK
  - Focused hardware acceleration
- Network programming
- Rich Expansion connector
  - i2c, SPI, GIO, 1-wire
  - Analog compare + interrupts
- TinyOS tool chain
- **sub** microsecond RF synchronization primitive
Rich Sensor board

- PHOTO
- TEMP
- ACCELEROMETER
- MAGNETOMETER
- MICROPHONE
- SOUNDER

ADC Signals (ADC1-ADC6)
On/Off Control
I^2C Bus
Interrupt

Microphone  Sounder  Magnetometer

Temperature Sensor

Light Sensor

1.25 in

2.25 in
# Mote Evolution

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Microcontroller</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>AT90LS8535</td>
<td>ATmega163</td>
<td>ATmega128</td>
<td>TI MSP430</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program memory (KB)</td>
<td>8</td>
<td>16</td>
<td>128</td>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAM (KB)</td>
<td>0.5</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Active Power (mW)</td>
<td>15</td>
<td>15</td>
<td>8</td>
<td>33</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sleep Power (µW)</td>
<td>45</td>
<td>45</td>
<td>75</td>
<td>75</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wakeup Time (µs)</td>
<td>1000</td>
<td>36</td>
<td>180</td>
<td>180</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nonvolatile storage</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chip</td>
<td>24LC256</td>
<td>AT45DB041B</td>
<td>ST M24M01S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connection type</td>
<td>I²C</td>
<td>SPI</td>
<td>F^2C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Size (KB)</td>
<td>32</td>
<td>512</td>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communication</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radio</td>
<td>TR1000</td>
<td>TR1000</td>
<td>CC1000</td>
<td>CC2420</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data rate (kbps)</td>
<td>10</td>
<td>40</td>
<td>38.4</td>
<td>250</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modulation type</td>
<td>OOK</td>
<td>ASK</td>
<td>FSK</td>
<td>O-QPSK</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Receive Power (mW)</td>
<td>9</td>
<td>12</td>
<td>29</td>
<td>38</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transmit Power at 0dBm (mW)</td>
<td>36</td>
<td>36</td>
<td>42</td>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Power Consumption</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minimum Operation (V)</td>
<td>2.7</td>
<td>2.7</td>
<td>2.7</td>
<td>1.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Active Power (mW)</td>
<td>24</td>
<td>27</td>
<td>44</td>
<td>89</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Programming and Sensor Interface</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expansion</td>
<td>none</td>
<td>51-pin</td>
<td>51-pin</td>
<td>none</td>
<td>51-pin</td>
<td>19-pin</td>
<td>51-pin</td>
<td>10-pin</td>
</tr>
<tr>
<td>Communication</td>
<td>IEEE 1284 (programming) and RS232 (requires additional hardware)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>USB</td>
</tr>
<tr>
<td>Integrated Sensors</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
Tiny OS Concepts

- Scheduler + Graph of Components
  - constrained two-level scheduling model: threads + events
- Component:
  - Commands,
  - Event Handlers
  - Frame (storage)
  - Tasks (concurrency)
- Constrained Storage Model
  - frame per component, shared stack, no heap
- Very lean multithreading
- Efficient Layering
Application = Graph of Components

Example: ad hoc, multi-hop routing of photo sensor readings

3450 B code
226 B data

Graph of cooperating state machines on shared stack
Sensor network programming challenges

- Driven by interaction with environment
  - Data collection and control, not general purpose computation
  - Reactive, event-driven programming model
- Extremely limited resources
  - Very low cost, size, and power consumption
  - Typical embedded OSs consume hundreds of KB of memory
- Reliability for long-lived applications
  - Apps run for months/years without human intervention
  - Reduce run time errors and complexity
- Soft real-time requirements
  - Few time-critical tasks (sensor acquisition and radio timing)
  - Timing constraints through complete control over app and OS
TinyOS

- Very small “operating system” for sensor networks
  - Core OS requires 396 bytes of memory
- Component-oriented architecture
  - Set of reusable system components: sensing, communication, timers, etc.
  - No binary kernel - build app specific OS from components
- Concurrency based on tasks and events
  - Task: deferred computation, runs to completion, no preemption
  - Event: Invoked by module (upcall) or interrupt, may preempt tasks or other events
  - Very low overhead, no threads
- Split-phase operations
  - No blocking operations
  - Long-latency ops (sensing, comm, etc.) are split phase
  - Request to execute an operation returns immediately
  - Event signals completion of operation
TinyOS Execution Model

- Commands request action
  - ack/nack at every boundary
  - call cmd or post task
- Events notify occurrence
  - HW interrupt at lowest level
  - may signal events
  - call cmds
  - post tasks
- Tasks provide logical concurrency
  - preempted by events
- Migration of HW/SW boundary

Diagram:
- Application comp
- Active message
- Radio Packet
- Radio byte
- RFM

Data processing:
- Message-event driven
- Event-driven packet-pump
- Event-driven byte-pump
- Event-driven bit-pump

Encoding/Decoding:
- CRC

Message-event driven:
- Active message

Encoding/Decoding:
- Encode/decode

[SelfOrg], SS 2005
Programming TinyOS: nesC

- Dialect of C with support for *components*
  - Components **provide** and **require** interfaces
  - Create application by wiring together components using **configurations**

- Whole-program compilation and analysis
  - nesC compiles entire application into a single C file
  - Compiled to mote binary by back-end C compiler (e.g., gcc)
  - Allows aggressive cross-component inlining
  - Static race condition detection

- Important restrictions
  - No function pointers (makes whole-program analysis difficult)
  - No dynamic memory allocation
  - No dynamic component instantiation/destruction
  - *These static requirements enable analysis and optimization*
Event-Driven Sensor Access Pattern

```c
command result_t StdControl.start() {
    return call Timer.start(TIMER_REPEAT, 200);
}

event result_t Timer.fired() {
    return call sensor.getData();
}

event result_t sensor.dataReady(uint16_t data) {
    display(data)
    return SUCCESS;
}
```

- clock event handler initiates data collection
- sensor signals data ready event
- data event handler calls output command
- device sleeps or handles other activity while waiting
- conservative send/ack at component boundary
TinyOS Commands and Events

```c
{  
    ...  
    status = call CmdName(args)  
    ...  
}

command CmdName(args) {  
    ...  
    return status;  
}

event EvtName)(args) {  
    ...  
    return status;  
}

{  
    ...  
    status = signal EvtName(args)  
    ...  
}
```
TinyOS Execution Contexts

- Events generated by interrupts preempt tasks
- Tasks do not preempt tasks
- Both essential process state transitions
Tasks

- provide concurrency internal to a component
  - longer running operations
- are preempted by events
- able to perform operations beyond event context
- may call commands
- may signal events
- not preempted by tasks

```c
{  
  ...  
  post TskName();
  ...
}
```

```c
task void TskName {
  ...
}
```
Typical Application Use of Tasks

- event driven data acquisition
- schedule task to do computational portion

```cpp
event result_t sensor.dataReady(uint16_t data) {
    putdata(data);
    post processData();
    return SUCCESS;
}

task void processData() {
    int16_t i, sum=0;
    for (i=0; i < maxdata; i++)
        sum += (rdata[i] >> 7);
    display(sum >> shiftdata);
}
```

- 128 Hz sampling rate
- simple FIR filter
- dynamic software tuning for centering the magnetometer signal (1208 bytes)
  - digital control of analog, not DSP
- ADC (196 bytes)
Task Scheduling

- Currently simple FIFO scheduler
- Bounded number of pending tasks
- When idle, shuts down node except clock

- Uses non-blocking task queue data structure

- Simple event-driven structure + control over complete application/system graph
  - instead of complex task priorities and IPC
TinyOS Active Messages

- **Sending**
  - Declare buffer storage in a frame
  - Request Transmission
  - Name a handler
  - Handle Completion signal

- **Receiving**
  - Declare a handler
  - Firing a handler
    - automatic
    - behaves like any other event

- **Buffer management**
  - strict ownership exchange
  - tx: done event => reuse
  - rx: must rtn a buffer
bool pending;
struct TOS_Msg data;

command result_t IntOutput.output(uint16_t value) {
    IntMsg *message = (IntMsg *)data.data;
    if (!pending) {
        pending = TRUE;
        message->val = value;
        message->src = TOS_LOCAL_ADDRESS;
        if (call Send.send(TOS_BCAST_ADDR, sizeof(IntMsg), &data))
            return SUCCESS;
        pending = FALSE;
    }
    return FAIL;
}

• Refuses to accept command if buffer is still full or network refuses to accept send command
• User component provide structured msg storage
Send Done Event

```c
event result_t IntOutput.sendDone(TOS_MsgPtr msg,  
    result_t success)  
{
    if (pending && msg == &data) {
        pending = FALSE;
        signal IntOutput.outputComplete(success);
    }
    return SUCCESS;
}
```

- Send done event fans out to all potential senders
- Originator determined by match
  - free buffer on success, retry or fail on failure
- Others use the event to schedule pending communication
Receive Event

```c
event TOS_MsgPtr ReceiveIntMsg.receive(TOS_MsgPtr m) {
    IntMsg *message = (IntMsg *)m->data;
    call IntOutput.output(message->val);
    return m;
}
```

- Active message automatically dispatched to associated handler
  - knows the format, no run-time parsing
  - performs action on message event
- Must return free buffer to the system
  - typically the incoming buffer if processing complete
Maintaining Scheduling Agility

- Need logical concurrency at many levels of the graph
- While meeting hard timing constraints
  - sample the radio in every bit window

⇒ Retain event-driven structure throughout application
⇒ Tasks extend processing outside event window
⇒ All operations are non-blocking
nesC Interfaces

- nesC interfaces are bidirectional
  - **Command**: Function call from one component requesting service from another
  - **Event**: Function call indicating completion of service by a component
  - Grouping commands/events together makes inter-component protocols clear

```c
interface Timer {
    command result_t start(char type, uint32_t interval);
    command result_t stop();
    event result_t fired();
}

interface SendMsg {
    command result_t send(TOS_Msg *msg, uint16_t length);
    event result_t sendDone(TOS_Msg *msg, result_t success);
}
```
nesC Components

- Two types of components
  - **Modules** contain implementation code
  - **Configurations** wire other components together
  - An application is defined with a single top-level configuration

```c
module TimerM {
  provides {
    interface StdControl;
    interface Timer;
  }
  uses interface Clock;
} implementation {
  command result_t Timer.start(char type, uint32_t interval) {
    ... }
  command result_t Timer.stop() {
    ... }
  event void Clock.tick() {
    ... }
}
```

[SelfOrg], SS 2005
Configuration Example

- Allow aggregation of components into “supercomponents”

```
configuration TimerC {
    provides {
        interface StdControl;
        interface Timer;
    }
}
```

```
implementation {
    components TimerM, HWClock;
    // Pass-through: Connect our "provides"
    // to TimerM "provides"
    StdControl = TimerM.StdControl;
    Timer = TimerM.Timer;
    // Normal wiring: Connect "requires" to "provides"
    TimerM.Clock -> HWClock.Clock;
}
```
Top-Level Configuration

[Diagram of Top-Level Configuration]
Concurrency Model

- **Tasks** used as deferred computation mechanism

```c
// Signaled by interrupt handler
event void Receive.receiveMsg(TOS_Msg *msg) {
    if (recv_task_busy) {
        return; // Drop!
    }
    recv_task_busy = TRUE;
    curmsg = msg;
    post recv_task();
}
task void recv_task() {
    // Process curmsg ...
    recv_task_busy = FALSE;
}
```

- Commands and events cannot block
- Tasks run to completion, scheduled non-preemptively
- Scheduler may be FIFO, EDF, etc.
struct main(\text{void}) { 
    RealMain$\text{hardwareInit}() ; 
    TOSH$\text{sched\_init}() ; 

    RealMain$\text{StdControl\_init}() ; 
    RealMain$\text{StdControl\_start}() ; 
    RealMain$\text{Interrupt\_enable}() ; 

    while ( 1 ) { 
        TOSH$\text{run\_task}() ; 
    } 
} 

\text{Hardware and Kernel Initialization} 

\text{Application Initialization} 

\text{Infinite Loop} 

bool TOSH$\text{run\_next\_task}() \{ 
    uint8_t old\_full; 
    void (*func)(\text{void}); 
    if (TOSH$\text{sched\_full} == TOSH$\text{sched\_free}) \{ 
        return 0; 
    \} \text{else} \{ 
    old\_full = TOSH$\text{sched\_full} ;
    TOSH$\text{sched\_full}++ ;
    TOSH$\text{sched\_full} & = TOSH$\text{TASK\_BITMASK} ;
    func = TOSH$\text{queue}[ (int) old\_full ] . tp; 
    TOSH$\text{queue}[ (int) old\_full ] . tp = 0; 
    func(); 
    return 1; 
\} 

\text{1. First Run All Tasks in the Task Queue (Strictly a FIFO)} 
\text{2. Then Sleep (In Low Power Mode)} 
\text{3. And Wait for an Interrupt} 

\text{Task Runs To Completion (But May Be Interrupted By An Event)} 

[SelfOrg], SS 2005
Compilation of a TinyOS Application
Evaluation

- TinyOS component census
  - Core TinyOS: 401 components (235 modules, 166 configurations)
  - Average of 74 components per app
  - Modules between 7 and 1898 lines of code (avg. 134)

- Race condition analysis on TinyOS tree
  - Original code: 156 potential races, 53 false positives
  - Fixed by using atomic or moving code into tasks

- Race condition false positives:
  - Shared variable access serialized by another variable
  - Pointer-swapping (no alias analysis)
Inlining and dead code elimination saves both space and time.

<table>
<thead>
<tr>
<th>Application</th>
<th>Size</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>optimized</td>
<td>unoptimized</td>
</tr>
<tr>
<td>Base TinyOS Runtime</td>
<td>396</td>
<td>646</td>
</tr>
<tr>
<td></td>
<td>1081</td>
<td>1091</td>
</tr>
<tr>
<td>Habitat monitoring Surge</td>
<td>11415</td>
<td>19181</td>
</tr>
<tr>
<td></td>
<td>14794</td>
<td>20645</td>
</tr>
<tr>
<td>Object tracking</td>
<td>23525</td>
<td>37195</td>
</tr>
<tr>
<td>Maté</td>
<td>23741</td>
<td>25907</td>
</tr>
<tr>
<td>TinyDB</td>
<td>63726</td>
<td>71269</td>
</tr>
</tbody>
</table>

Inlining benefit for 5 sample applications.

<table>
<thead>
<tr>
<th>Cycles</th>
<th>optimized</th>
<th>unoptimized</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work</td>
<td>371</td>
<td>520</td>
<td>29%</td>
</tr>
<tr>
<td>Boundary crossing</td>
<td>109</td>
<td>258</td>
<td>57%</td>
</tr>
<tr>
<td>Total</td>
<td>480</td>
<td>778</td>
<td>38%</td>
</tr>
</tbody>
</table>

Clock cycles for clock event handling, crossing 7 modules.
## Overhead of TinyOS Primitive Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost (cycles)</th>
<th>Time (uSecs)</th>
<th>Normalized to Byte Copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte Copy</td>
<td>8</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Signal an Event</td>
<td>10</td>
<td>2.5</td>
<td>1.25</td>
</tr>
<tr>
<td>Call a Command</td>
<td>10</td>
<td>2.5</td>
<td>1.25</td>
</tr>
<tr>
<td>Schedule a Task</td>
<td>46</td>
<td>11.5</td>
<td>6</td>
</tr>
<tr>
<td>Context Switch</td>
<td>51</td>
<td>12.75</td>
<td>6</td>
</tr>
<tr>
<td>Hardware Interrupt (hw)</td>
<td>9</td>
<td>2.25</td>
<td>1</td>
</tr>
<tr>
<td>Hardware Interrupt (sw)</td>
<td>71</td>
<td>17.75</td>
<td>9</td>
</tr>
</tbody>
</table>

## Code and Data Size of the TinyOS kernel

<table>
<thead>
<tr>
<th>Component</th>
<th>Code Size (bytes)</th>
<th>Data Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Init</td>
<td>172</td>
<td>30</td>
</tr>
<tr>
<td>Scheduler</td>
<td>178</td>
<td>16</td>
</tr>
<tr>
<td>C runtime</td>
<td>82</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>432</td>
<td>46</td>
</tr>
</tbody>
</table>
Surge Application

- Surge is a java program for displaying multihop routing topology in sensor networks.
- When working with multihop routing applications, Surge provides the following information to users:
  - Detects the existence of all the motes in a wireless network.
  - Displays mote information, including the mote identification number (ID), the number of messages sent from each mote, etc.
  - Displays the topology of network.

[SelfOrg], SS 2005
Surge Application: multi-hop data collection

- Periodically collect sensor readings and route to base
  - Timer component fires event to read from ADC
  - ADC completion event initiates communication
- Multihop routing using adaptive spanning tree
  - Nodes route messages to parent in tree
  - Parent selection based on link quality estimation
  - Snoop on radio messages to find better parent
Raw Data Packet

<table>
<thead>
<tr>
<th>Byte #</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Packet frame synch</td>
<td>Always 0x7E</td>
</tr>
<tr>
<td></td>
<td>byte</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Packet Type</td>
<td>There are 5 known packet types:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- P_PACKET_NO_ACK (0x42) - User packet with no ACK required.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- P_PACKET_ACK (0x41) – User packet. ACK required. Includes a prefix byte. Receiver must send a P_ACK response with prefix byte as contents.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- P_ACK (0x40) – The ACK response to a P_PACKET_ACK packet. Includes the prefix byte as its contents.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- PUNKNOWN (0xFF) – An unknown packet type.</td>
</tr>
<tr>
<td>2…n-1</td>
<td>Payload Data</td>
<td>In most cases will be a TinyOS Message of varying length, which is described below.</td>
</tr>
<tr>
<td>n</td>
<td>SYNC_BYTE</td>
<td>Always 0x7E</td>
</tr>
</tbody>
</table>
# TOS_Msg Data Packet

<table>
<thead>
<tr>
<th>Byte #</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0 - 1  | Message Address | One of 3 possible value types:  
- Broadcast Address (0xFFFF) – message to all nodes.  
- UART Address (0x007e) – message from a node to the gateway serial port. All incoming messages will have this address.  
- Node Address – the unique ID of a node to receive message. |
| 2      | Message Type | Active Message (AM) unique identifier for the type of message it is. Typically each application will have its own message type. Examples include:  
- AMTYPE_XUART = 0x00  
- AMTYPE_MHOP_DEBUG = 0x03  
- AMTYPE_SURGE_MSG = 0x11  
- AMTYPE_XSENSOR = 0x32  
- AMTYPE_XMULTIHOP = 0x33  
- AMTYPE_MHOP_MSG = 0xFA |
| 3      | Group ID     | Unique identified for the group of motes participating in the network. The default value is 125 (0x7d). Only motes with the same group id will talk to each other. |
| 4      | Data Length  | The length (l) in bytes of the data payload. This does not include the CRC or frame synch bytes. |
| 5...n-2| Payload data | The actual message content. The data resides at byte 5 through byte 5 plus the length of the data (l from above). The data will be specific to the message type. Specific message types are discussed in the next section. |
| n-1, n | CRC          | Two byte code that ensures the integrity of the message. The CRC includes the Packet Type plus the entire unescaped TinyOS message. A discussion on how the CRC is computed is included in the Appendix. |
## Multihop Message

<table>
<thead>
<tr>
<th>Source Address</th>
<th>Origin Address</th>
<th>Sequence Number</th>
<th>Hop Count</th>
<th>Application Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
<td>7...n</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Byte #</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1</td>
<td>Source Address</td>
<td>The address of the forwarding node.</td>
</tr>
<tr>
<td>2,3</td>
<td>Origin Address</td>
<td>The address of the node that originated the packet.</td>
</tr>
<tr>
<td>4,5</td>
<td>Sequence Number</td>
<td>Used for determining route and calculating missed packets.</td>
</tr>
<tr>
<td>6</td>
<td>Hop Count</td>
<td>Used for calculating route. Number of nodes traversed.</td>
</tr>
<tr>
<td>7...n</td>
<td>Application Data</td>
<td>The specific application data. Such as SurgeMsg (see below).</td>
</tr>
</tbody>
</table>
### Surge Message

<table>
<thead>
<tr>
<th>Byte #</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Type</td>
<td>The type of message that indicates the action. Known values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• SURGE_TYPE_SENSORREADING (0x00) – The message contains sensor data.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• SURGE_TYPE_ROOTBEACON (0x01) – The message contains the root beacon.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• SURGE_TYPE_SETRATE (0x02) – Changes the rate a mote will send packets.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• SURGE_TYPE_SLEEP (0x03) – Puts the mote to sleep.</td>
</tr>
<tr>
<td>1-2</td>
<td>Reading</td>
<td>Does not appear to be used.</td>
</tr>
<tr>
<td>3-4</td>
<td>Parent Addr</td>
<td>The address of the Parent Node.</td>
</tr>
<tr>
<td>5-8</td>
<td>Sequence Number</td>
<td>The upper 9 bits represent the battery voltage. The remaining</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bits count the number of packets sent since the application was last reset.</td>
</tr>
<tr>
<td>9</td>
<td>Light</td>
<td>The raw light sensor reading.</td>
</tr>
<tr>
<td>10</td>
<td>Temp</td>
<td>The raw thermistor reading.</td>
</tr>
<tr>
<td>11</td>
<td>Mag X</td>
<td>The raw sensor reading for the x-axis magnetometer.</td>
</tr>
<tr>
<td>12</td>
<td>Mag Y</td>
<td>The raw sensor reading for the y-axis magnetometer.</td>
</tr>
<tr>
<td>13</td>
<td>Accel X</td>
<td>The raw sensor reading for the x-axis accelerometer.</td>
</tr>
<tr>
<td>14</td>
<td>Accel Y</td>
<td>The raw sensor reading for the y-axis accelerometer.</td>
</tr>
</tbody>
</table>