Maté: A Tiny Virtual Machine for Sensor Networks

Philip Levis and David Culler
{pal,culler}@cs.berkeley.edu

Computer Science Division
University of California
Berkeley, California

Intel Research: Berkeley
Intel Corporation
Berkeley, California

ABSTRACT

Composed of tens of thousands of tiny devices with very limited resources ("motes"), sensor networks are subject to novel systems problems and constraints. The large number of motes in a sensor network means that there will often be some failing nodes; networks must be easy to repopulate. Often there is no feasible method to recharge motes, so energy is a precious resource. Once deployed, a network must be reprogrammable although physically unreachable, and this reprogramming can be a significant energy cost.

We present Maté, a tiny communication-centric virtual machine designed for sensor networks. Maté's high-level interface allows complex programs to be very short (under 100 bytes), reducing the energy cost of transmitting new programs. Code is broken up into small capsules of 24 instructions, which can self-replicate through the network. Packet sending and reception capsules enable the deployment of ad-hoc routing and data aggregation algorithms. Maté's concise, high-level program representation simplifies programming and allows large networks to be frequently reprogrammed in an energy-efficient manner; in addition, its safe execution environment suggests a use of virtual machines to provide the user/kernel boundary on motes that have no hardware protection mechanisms.

1. INTRODUCTION

Wireless sensor networks pose novel problems in system management and programming. Networks of hundreds or thousands of nodes, limited in energy and individual resources, must be reprogrammable in response to changing needs. A spectrum of reprogrammability emerges, from simple parameter adjustments to uploading complete binary images. As network bandwidth is limited and network activity is a large energy draw, a concise way to represent a wide range of programs is needed. A virtual machine is a natural way to achieve this, but characteristics of sensor networks, such as their large numbers, require an approach distinct from prior VM architectures.

Traditionally, virtual machines have focused on virtualizing real hardware [3][5], intermediate program representation [4], or bytecode interpretation [20]. Each technique has its own design goals and advantages as well as drawbacks. All have generally focused on traditional uniprocessor or multiprocessor environments, although some work has dealt with bringing bytecode interpreters to small consumer devices [15]; one solution, PicoJava, has been to design hardware that natively runs the VM instruction set [16]. Virtual machines for tiny devices have been proposed, but their de-

sign goals have focused on byte-code verification and online compilation [30].

Sensor networks are a domain characterized by resources that are orders of magnitude smaller than what current virtual machines require. For example, current sensor network nodes (**motes**[1]) have 8 to 128 KB of instruction memory and 512B to 4KB of RAM, while the K Virtual Machine targets devices with a memory budget of 160KB to 512 KB [15]. Energy is a critical resource, especially in terms of communication; sending a single bit can consume the same energy as executing 1000 instructions. The hardware/software boundary of an individual mote is currently a topic of open research [10][11].

Although sensor networks are distinct from other computing domains in several ways, they have similarities to parallel architectures and distributed systems; notably, they are all composed of many separate processing elements that must communicate. There are, of course, important differences; as opposed to a reliable message passing system in parallel architectures, sensor networks have lossy irregular wireless networks.

Content-based routing and naming in sensor networks have been examined for bandwidth conservation [9][14]; this work borrows ideas from active networks. By taking a more general approach, however, active networks allow not only these mechanisms but arbitrary computation within a network [32]. There is growing interest in data aggregation as a mechanism for making the most of precious sensor network bandwidth [13] – a problem that could be easily investigated with an active sensor network. While incremental code migration has been achieved using XML [6], motes lack the RAM to store even a few simple XML elements.

The basic question of how sensor networks will be programmed remains largely unanswered. A spectrum exists between minor application adjustments (tweaking constants) and uploading entire programs (binary images). The former can be achieved with a tiny RPC-like mechanism and a handful of packets, while the latter requires sending kilobytes of data at high energy cost. Our experience has shown that most sensor network applications are composed of a common set of services and sub-systems, combined in different ways. A system that allows these compositions to be concisely described (a few packets) could provide much of the flexibility of reprogramming with the transmission costs of parameter tweaking.

To establish that a wide range of sensor network applica-

---

[1]**mote**: n.; A very small particle; a speck: "Dust motes hung in a slant of sunlight" (Anne Tyler).

tions can be composed of a small set of high level primitives, we built Maté[2], a byte-code interpreter to run on motes. Capsules of 24 instructions fit in a packet and can forward themselves through a network. Complex programs, such as an ad-hoc routing layer, can be composed of several capsules and represented in under 100 bytes. Maté therefore allows a wide range of programs to be installed in a network quickly and with little network traffic, conserving energy. We show that for a common class of frequently reprogrammed systems, the energy cost of the CPU overhead of a bytecode interpreter is outweighed by the energy savings of transmitting such concise program representations.

The following two sections present sensor networks and TinyOS, an operating system designed specifically for sensor networks; from these two sections a clear set of requirements emerge for a mote reprogramming system. Section 4 describes the design, structure, and implementation of Maté. Section 5 evaluates the system's behavior and performance. Section 6 and 7 discuss our evaluations of Maté and the conclusions one can draw from it. The full Maté instruction set and an implementation of a beacon-less ad-hoc routing algorithm are included in the appendix.

2. SENSOR NETWORKS

The progression of Moore's Law has led to the design and manufacture of small sensing computers that can communicate with one another over a wireless network [11]. Research and industry indicate that motes will be used in networks of hundreds, thousands, or more [2][7][18].

Sensor networks are distinct from traditional computing domains. Their design assumes being embedded in common environments (e.g., a corn field, a bathroom), instead of dedicated ones (e.g., a server room, an office). Mean time to failure combined with large numbers leads to routine failure; a network must be easy to repopulate without interrupting operation. Compared to infrastructure or even mobile systems, power is scarce and valuable. One can easily recharge a laptop or a handheld; recharging thousands of motes (or even *finding* all of them!) is much more difficult.

Communication in sensor networks is more precious than in other computing domains. Sending a single bit of data can consume the energy of executing a thousand instructions. This disparity has led to investigation of passive networking systems [12]. Additionally, the power cost of producing sensor data is orders of magnitude lower than the cost of transmitting it, and its rate of production (e.g. 100Kbit/sec·mote) is far greater than can be communicated through a network. This has led to investigation into content-specific routing and in-network data aggregation [9][21]; centralized control and data collection is inefficient and wasteful.

Once deployed, motes often cannot be easily collected; reprogramming cannot require physical contact. Being able to re-task a network as analysis techniques or environmental conditions evolve is desirable. For example, we are in the process of deploying a sensor network to monitor storm petrels[3] on Great Duck Island off the coast of Maine. Deploying motes involves placing them in possible nesting sites; once birds nest, disturbing them to reach a mote is infeasible. Being able to monitor the birds this closely (inches) for continuous periods is unheard of in biological research; scientists are still unsure exactly how, what and at what frequency they want the network to sense [23]; the ability to reprogram is invaluable.

Our experiences working with civil engineers present similar issues in programming. By monitoring building structures with embedded motes, expensive tasks such as earthquake damage assessment can be made fast and simple [1]. Such a network would be very useful for a wide variety of tasks, such as water damage detection or sound propagation testing. As not all of these tasks would be anticipated, installing software for all of them at deployment time is impossible; additionally, as the network is embedded in the building, the motes are unreachable and not feasibly redeployable.

Being able to reprogram a network quickly is also very useful for issues such as data aggregation and adaptive query processing [22]; dynamically installing aggregation functions can provide a more flexible system than a predefined set of aggregates. Examining these sample use cases (habitat monitoring, building instrumentation, and query processing), it is clear that a flexible, rapid and powerful reprogramming mechanism is needed.

Although computation is inexpensive in comparison to communication, it is not abundant. For example, the latest generation of motes we have designed (the `mica` platform) has a 4MHz 8-bit processor. An individual mote cannot perform large computations rapidly by itself – they must be performed in a distributed manner. The scarcity of RAM and network bandwidth inhibits the use of algorithms that depend on global state, such as many ad-hoc routing algorithms designed for mobile computers [28].

3. TINYOS

TinyOS is an operating system designed specifically for use in sensor networks [11]. Combined with a family of wireless sensor devices, TinyOS is currently used as a research platform by several institutions. Examining sensor network challenges and the limitations of TinyOS produces a set of requirements for a flexible and concise sensor network programming model.

3.1 Hardware: `rene2`, `mica`

Two mote hardware platforms, the `rene2` and the `mica`, are currently available for general development. The `rene2` is the older of the two, and has correspondingly smaller resources. The complete family of TinyOS motes is summarized in Table 1. The `mica` supports 40Kbit communication on its radio, while the `rene2` is limited to 10Kbit; the `mica` radio can be put into 10 Kbit mode for backwards compatibility. All mote platforms are Harvard architectures, with separate instruction and data memory. Installing new binary code requires a reset to take effect.

To change the behavior of a TinyOS program, one must either hardcode a state transition in a program (when one

---

[2]**Maté** (mah-tay) n.; Maté is as tea like beverage consumed mainly in Argentina, Uruguay, Paraguay and southern Brazil. It is brewed from the dried leaves and stemlets of the perennial tree *Ilex paraguarensis* ("yerba maté"). The name "maté" derives from the Quechua word "mati" that means cup or vessel used for drinking.

[3]**Storm Petrel**: Genus *Oceanodroma*. Habitat: Open ocean and oceanic islands. Nests are built in burrows, among colonies on oceanic islands. After mating and hatching, *Oceanodroma* return to the open ocean.

Mote Type		WeC	rene2	rene2	dot	mica
Date		9/99	10/00	6/01	8/01	2/02
Microcontroller						
Type		AT90LS8535		ATMega163		ATMega103
Prog. mem. (KB)		8		16		128
RAM (KB)		0.5		1		4
Nonvolatile storage						
Chip		24LC256				AT45DB041B
Connection type		I2C				SPI
Size (KB)		32				512
Default Power source						
Type		Li	Alk		Li	Alk
Size		CR2450	2xAA		CR2032	2xAA
Capacity (mAh)		575	2850		225	2850
Communication						
Radio		RFM TR1000				
Rate (Kbps)		10	10	10	10	10/40
Modulation type		OOK				OOK/ASK

Table 1: The family of Berkeley TinyOS motes

receives a type of packet, start reading light data instead of temperature), or one must modify source code, recompile a TinyOS image, and place the entire new image on a mote.

3.2 Software Architecture

A TinyOS program is composed of a graph of software components. At the component level, TinyOS has three computational abstractions: commands, events, and tasks. Commands are used to call down the component graph: for example, telling the network component to send a packet. Events are calls up the component graph: for example, signaling the packet has been sent. From a traditional OS perspective, commands are analogous to downcalls while events are analogous to upcalls.

Tasks are used for long running computations that can be preempted by events. A command (or event) can enqueue a task to perform a computation and immediately return. TinyOS schedules tasks on a FIFO basis and runs a task to completion before another task is run; as they do not preempt one-another, they must be short. Otherwise, the task queue can overflow as new tasks are posted.

TinyOS supports high concurrency through split-phase non-blocking execution. No command or event blocks. Instead, completion of long lasting operations (such as sending a byte over a radio) are indicated by the issuing of an event (such as a send byte done). Non-blocking operation allows a form of high concurrency distinct from a threaded model: the operations of many components can be interleaved at a fine granularity. This asynchronous behavior is ultimately powered by hardware interrupts.

TinyOS provides high parallelism and efficiency through a constrained, and somewhat tricky, programming interface. This interface is badly suited to non-expert programmers, such as the biologists and civil engineers we are working with to deploy networks. A simpler programming model, which allows novice programmers to express their desired behavior without worrying about timing and asynchrony, would greatly improve the usefulness of TinyOS sensor networks.

3.3 TinyOS Networking: Active Messages

The top-level TinyOS packet abstraction is an Active Message [33]. The characteristics of this abstraction are important because they define the capabilities of systems built on top of it. AM packets are an unreliable data link protocol; the TinyOS networking stack handles media access control and single hop communication between motes. Higher layer protocols (e.g. network or transport) are built on top of the

AM interface.

AM packets can be sent to a specific mote (addressed with a 16 bit ID) or to a broadcast address (0xffff). TinyOS provides a namespace for up to 256 types of Active Messages, each of which can each be associated with a different software handler. AM types allow multiple network or data protocols to operate concurrently without conflict. The AM layer also provides the abstraction of an 8-bit AM group; this allows logically separate sensor networks to be physically co-present but mutually invisible, even if they run the same application.

3.4 System Requirements

Looking at sensor network challenges and the limitations of TinyOS and its hardware, a set of clear requirements emerge for an effective sensor network programming system. They are:

- Small – it must fit on rene2 and mica hardware (targeting only the cutting edge – mica – would alienate users);

- Expressive – one must be able to write a wide range of applications;

- Concise – applications should be short, to conserve network bandwidth;

- Resilient – applications cannot crash a mote;

- Efficient – energy efficient sensing and communication;

- Tailorable – support efficient specialized operations;

- and Simple – programming an entire network should be in-situ, fast, and mostly autonomous.

4. Maté

Maté is a bytecode interpreter that runs on TinyOS. It is a single TinyOS component that sits on top of several system components, including sensors, the network stack, and non-volatile storage (the "logger"). Code is broken in **capsules** of 24 instructions, each of which is a single byte long; larger programs can be composed of multiple capsules. In addition to bytecodes, capsules contain identifying and version information. Maté has two stacks: an operand stack and a return address stack. Most instructions operate solely on the operand stack, but a few instructions control program flow and several have embedded operands. There are three execution contexts that can run concurrently at instruction granularity. Maté capsules can forward themselves through a network with a single instruction. Maté provides both a built-in ad-hoc routing algorithm (the send instruction) as well as mechanisms for writing new ones (the sendr instruction).

We designed Maté to run on both the mica and the rene2 hardware platforms. This means that Maté and all of its subcomponents must fit in 1KB of RAM and 16 KB of instruction memory. Figure 1 outlines the code and data size breakdowns of the TinyOS components in a Maté mote.

4.1 Architecture and Instruction Set

The communication model in the Maté VM architecture allows a program to send a message as a single instruction; the sent message is (from the caller's perspective) automatically routed to its destination. The arrival of a packet automatically enqueues a task to process it. This approach
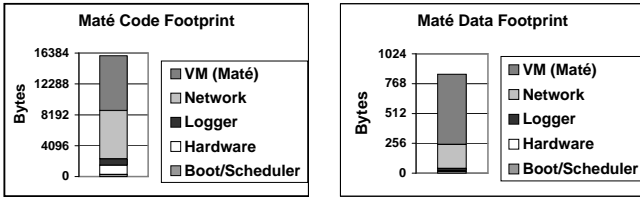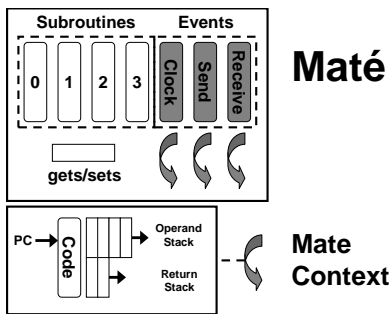
Figure 1: Maté Component Breakdown

Component	Code (bytes)	Data (bytes)
VM (Maté)	7286	603
Network	6410	206
Logger	844	18
Hardware	1232	8
Boot/Scheduler	272	14
Total	16044	849

Figure 2: Maté Architecture and Execution Model: Capsules, Contexts, and Stacks

basic	`00iiiiii`	i = instruction
s-class	`01iiixxx`	i = instruction, x = argument
x-class	`1ixxxxxx`	i = instruction, x = argument

Figure 3: Maté Instruction Formats

has strong similarities to Active Messages [33] and the J-Machine [26]. There are, of course, important differences – for example, instead of reliably routing to processors, it routes through an unreliable multihop wireless network. The tiny amount of RAM also forces motes to have a constrained storage model – Maté cannot buffer messages and tasks freely as the J-Machine can.

Maté instructions hide the asynchrony (and race conditions) of TinyOS programming. For example, when the `send` instruction is issued, Maté calls a command in the ad-hoc routing component to send a packet. Maté suspends the context until a message send complete event is received, at which point it resumes execution. By doing this, Maté does not need to manage message buffers – the capsule will not resume until the network component is done with the buffer. Similarly, when the `sense` instruction is issued, Maté requests data from the sensor TinyOS component and suspends the context until the component returns data with an event. This synchronous model makes application level programming much simpler and far less prone to bugs than dealing with asynchronous event notifications. Additionally, Maté efficiently uses the resources provided by TinyOS; during a split-phase operation, Maté does nothing on behalf of the calling context, allowing TinyOS to put the CPU to sleep or use it freely.

Maté is a stack-based architecture [19]. We chose this to allow a concise instruction set; most instructions do not have to specify operands, as they exist on the operand stack [17]. There are three classes of Maté instructions: basic, s-class, and x-class. Figure 3 shows the instruction formats for each class. Basic instructions include such operations as arithmetic, halting, and activating the LEDs on a mote. s-class instructions access in-memory structures used by the messaging system, such message headers or messaging layer state; they can only be executed within the message send and receive contexts. The two x-class instructions are `pushc` (push constant) and `blez` (branch on less than or equal to zero). Both the s-class and x-class instructions have an operand embedded in the instruction: s-class have a 3-bit index argument and x-class have a 6-bit unsigned value argument.

Eight instructions (usr0-7) are reserved for users to define. By default, they are no-ops. However, as a class of sensor network applications might require some specific processing outside the capabilities of Maté, such as a complex data filter, these user instructions allow efficient domain-specific instructions. Maté has been structured to make implementing these instructions easy. One can therefore build a specially tailored version of Maté with efficient support for common complex operations.

Maté's three execution contexts, illustrated in Figure 2, correspond to three events: clock timers, message receptions and message send requests. Inheriting from languages such as FORTH [25], each context has two stacks, an operand stack and a return address stack. The former is used for all instructions handling data, while the latter is used for subroutine calls. The operand stack has a maximum depth of 16 while the call stack has a maximum depth of 8. We have found this more than adequate for programs we have written. The clock operand stack persists across executions – if one invocation left a sensor reading on the top of the stack, it is available in the next invocation. This is an easy way to implement internal clock timers, as Section 4.3 demonstrates. When a clock capsule is installed, the value zero is pushed onto its operand stack. The receive and send contexts expect the message received or the data to send to be on the top of their stacks when they begin execution, so these stacks do not persist across invocations.

There are three operands types: values, sensor readings, and messages. Some instructions can only operate on certain types. For example, the `putled` instruction expects a value on the top of the operand stack. However, many instructions are polymorphic. For example, the `add` instruction can be used to add any combination of the types, with different results. Adding messages results in appending the data in the second operand onto the first operand. Adding a value to a message appends the value to the message data payload. Adding a sensor reading to a value results in a sensor reading of the same type increased by the value, while adding two sensor readings of different types (e.g. light and humidity) just returns the first operand. Sensor readings can be turned into values with the `cast` instruction.

There is a single shared variable among the three contexts – a one word heap. It can be accessed with the `sets` and `gets` instructions. This allows the separate contexts to communicate shared state (e.g. timers). Our experience so far has shown this to be adequate but it is unclear whether a

```
      pushc 1      # Push one onto operand stack
      add          # Add the one to the stored counter
      copy         # Copy the new counter value
      pushc 7
      and          # Take bottom three bits of copy
      putled       # Set the LEDs to these three bits
      halt
```

Figure 4: Maté `cnt_to_leds` − **Shows the bottom 3 bits of a counter on mote LEDs**

heap this size will be feasible in the long term. Its size could be easily increased by having **sets** and **gets** specify an additional operand to state the address within the heap. As we have yet to find a situation where this is necessary, we have decided against it for now.

4.2 Code Capsules and Execution

Maté programs are broken up into capsules of up to 24 instructions. This limit allows a capsule to fit into a single TinyOS packet. By making capsule reception atomic, Maté does not need to buffer partial capsules, which conserves RAM. Every code capsule includes type and version information. Maté defines four types of code capsules: message send capsules, message receive capsules, timer capsules, and subroutine capsules. Subroutine capsules allow programs to be more complex than what fits in a single capsule. Applications invoke and return from subroutines using the `call` and `return` instructions. There are names for up to 2^{15} subroutines; to keep Maté's RAM requirements small, its current implementation has only four.

Maté begins execution in response to an event – a timer going off, a packet being received, or a packet being sent. Each of these events has a capsule and an execution context. Control jumps to the first instruction of the capsule and executes until it reaches the `halt` instruction. These three contexts can run concurrently. Each instruction is executed as a TinyOS task, which allows their execution to interleave at an instruction granularity. Additionally, underlying TinyOS components can operate concurrently with Maté instruction processing. When a subroutine is called, the return address (capsule, instruction number) is pushed onto a return address stack and control jumps to the first instruction of the subroutine. When a subroutine returns, it pops an address off the return stack and jumps to the appropriate instruction.

The packet receive and clock capsules execute in response to external events; in contrast, the packet send capsule executes in response to the `sendr` instruction. As `sendr` will probably execute a number of Maté instructions in addition to sending a packet, it can be a lengthy operation. Therefore, when `sendr` is issued, Maté copies the message buffer onto the send context operand stack and schedules the send context to run. Once the message has been copied, the calling context can resume execution. The send context executes concurrently to the calling context, preparing a packet and later sending it. This frees up the calling context to handle subsequent events – in the case of the receive context, this is very important.

The constrained addressing modes of Maté instructions ensure a context cannot access the state of a separate context. Every push and pop on the operand and return value stack has bound checks to prevent overrun and underrun. As there is only a single shared variable, heap addressing is not a

```
      pushc 1      # Push one on the operand stack
      sense        # Read sensor 1 (light)
      copy         # Copy the sensor reading
      gets         # Get previous sent reading

      inv          # Invert previous reading
      add          # Current - previous sent value
      pushc 32
      add

      blez  17     # If curr < (prev-32) jump to send
      copy         # Copy the sensor reading
      inv          # Invert the current
      gets         # Get the previous reading

      add          # Previous - current
      pushc 32
      add
      blez  17     # If (curr+32) > prev jump to send

      halt
      copy         # PC 17 -- jump-to point from above
      sets         # Set shared var to current reading
      pushm        # Push a message onto operand stack

      clear        # Clear out the message payload
      add          # Add the reading to message payload
      send         # Send the message
      halt
```

Figure 5: Maté Program to Read Light Data and Send a Packet on Reading Change

problem. Unrecognized instructions result in simple no-ops. All bounds are always checked – the only way two contexts can share state is through `gets` and `sets`. Nefarious capsules can at worst clog a network with packets – even in this case, a newer capsule will inevitably be heard. By providing such a constrained execution environment and providing high-level abstractions to services such as the network layer, Maté ensures that it is resilient to buggy or malicious capsules.

4.3 Simple Maté Programs

The Maté program in Figure 4 maintains a counter that increments on each clock tick. The bottom three bits of the counter are displayed on the three mote LEDs. The counter is kept as a value which persists at the top of the stack across invocations. This program could alternatively been implemented by using `gets` and `sets` to modify the shared variable. This code recreates one of the simplest TinyOS applications, `cnt_to_leds`, implemented in seven bytes.

The Maté program in Figure 5 reads the light sensor on every clock tick. If the sensor value differs from the last sent value by more than a given amount (32 in this example), the program sends the data using Maté's built-in ad-hoc routing system. This program is 24 bytes long, fitting in a single capsule.

4.4 Code Infection

A capsule sent in a packet contains a type (subroutines 0-3, clock, receive, send) and a version number. If Maté receives a more recent version of a capsule than the one of the specified type currently being used, Maté installs it. A capsule can be transmitted to other motes using the `forw` instruction, which broadcasts the issuing capsule for network neighbors to install. These motes will then issue `forw` when they execute the capsule, forwarding the capsule to their local neighbors. Because of the version information, motes

Simple	Downcall	Quick Split	Long Split
and	rand	sense	sendr
call	putled	log	send
swap	pots	logr	forwo
blez	son		

Figure 6: Maté Instruction Overhead Classes

with the new code will ignore older capsules they receive. Over time, the new code will disseminate through the logical network like a virus – all one needs to do is install it on a single mote and execute the capsule. Correspondingly, for a mote to be able to run a different version of the program with no threat of reprogramming, it must be in a logically separate network. Versioning is implemented as a 32 bit counter – this allows a single network to last for a very long time (centuries) even with very rapid reprogramming rates (once every few seconds).

A capsule can also forward other installed capsules with the `forwo` (forward other) instruction. This is useful if the desired program is composed of several capsules; a temporary clock capsule that forwards every capsule can be installed, then as each component capsule is installed it will be forwarded. Once the entire network has installed all of these capsules, the clock capsule can be replaced with a program to drive the application.

5. EVALUATION

To test the expressiveness, behavior and performance of Maté, we implemented an ad-hoc routing algorithm, measured its rate of instruction issue, quantified its CPU overhead, and measured network infection rates with different capsule forwarding probabilities.

5.1 BLESS – BeaconLESS ad-hoc routing

Ad-hoc networking is a critical system issue in sensor networks. The transient nature of sensor networks means packet routing must be adaptive; effectively collecting data from a network requires an ad-hoc routing algorithm. BLESS is an ad-hoc routing protocol included in the standard TinyOS release, implemented in 600 lines of C. We have re-implemented a slightly simpler version[4] of BLESS in Maté to demonstrate that Maté is expressive enough to provide similar functionality to native TinyOS code. Additionally, BLESS can now be dynamically installed on a network, demonstrating that Maté transforms sensor networks into active networks.

BLESS includes routing information in every packet and transmits everything on an AM broadcast. All messages are forwarded to the root of a tree, which can be connected to a PC for processing or storage. By snooping traffic, motes can hear packets sent to other motes to find a suitable routing tree parent.

Every BLESS packet contains three fields: the address of the source mote, the address of the destination mote, and the hopcount of the source. The hopcount of the tree root is zero. Motes try to minimize their hopcount in the tree. A maximum hopcount of 16 prevents disconnected graphs from wastefully sending packets in perpetual cycles. If a parent is not heard for a given interval, a mote chooses a new parent.

The complete Maté BLESS code, the packet format and the frame format are included in Appendix B.

---

[4]The native TinyOS version maintains a table of possible parents to use; the Maté version only keeps track of one.

```
gets      # Get the counter
pushm     # Get a buffer
clear     # Clear the buffer
add       # Append the count

send      # Send the packet
pushc 0
sets      # Clear counter to 0
gets      # Jump-to point

pushc 1
add
sets      # Increment counter
pushc 0

blez 7    # Always branch
```

Average. loop executions: 8472
Maté IPS: 7 + ((loops / 5) * 6):   10173

Figure 7: Maté Simple Loop and IPS Calculation

Operation	Maté Clock Cycles	Native Clock Cycles	Cost
Simple: and	469	14	33.5:1
Downcall: rand	435	45	9.5:1
Quick Split: sense	1342	396	3.4:1
Long Split: sendr	$685 + \approx 20,000$	$\approx 20,000$	1.03:1

Figure 8: Maté Bytecodes vs. Native Code

5.2 Instruction Issue Rate

Some instructions (such as sending a packet) take much more time than others. Figure 6 contains a rough breakdown of the cost classes of Maté instructions. Some instructions (such as `add`) merely manipulate the operand stack with minimal additional processing. A second class of instructions (such as `putled`) call commands on components below Maté. TinyOS commands return quickly, but calling a function obviously has a CPU cost. The last two classes of instructions perform split-phase TinyOS operations; these instructions include the cost of a TinyOS event handler plus the latency between the command its corresponding event. For `sense`, this is very short, on the order of 200 microseconds. For `send` or `forw`, however, this involves a packet being sent, which takes roughly 50 milliseconds on the 10 Kbit radio.

To determine the cost of issuing an instruction in Maté, we wrote a program that executes a tight loop (six instructions) for five seconds. The code that clears the previous count and sends a packet containing the count is seven instructions. Maté does not normally have sole control of the processor when running. For example, the TinyOS network stack handles interrupts at 20KHz to find packet start symbols, then at 10KHz when reading or writing data. To measure Maté's raw performance, we turned the TinyOS radio off when running the Maté timing loop. From the loop counts we calculated the number of Maté instructions executed per second.

Every instruction in the loop was a simple one (e.g. add, branch, push constant). Therefore, the IPS we calculated executing the loop measures the approximate Maté overhead imposed on every instruction as opposed to overhead from calling components or split-phase operations. The average Maté IPS (instructions per second) was just over 10,000.

To precisely quantify the overhead Maté places over native code, we compiled a few small operations to native mote code and compared the cost of the native and Maté implementations. Obviously, some Maté instructions impose a higher overhead than others; those that encapsulate high-

Application	Binary		Maté	
	Size(bytes)	Install Time	Capsules	Instructions
sens_to_rfm	5394	≈79s	1	6
gdi-comm	7130	≈104s	1	19
bless_test	7381	≈108s	7	108

Figure 9: Application Installation Costs

level TinyOS constructs are efficient, while those that perform simple arithmetic operations are not nearly as much so. The comparative instruction costs are in Figure 8. We selected one instruction from each of the cost classes of Figure 6: **and**, **rand**, **sense**, and **send**. A Maté implementation's native instruction count is 33.5 to 1 for a logical and on two words, while 1.03 to 1 for a packet send. A logical and operation takes more cycles than random because it involves popping two operands and pushing one; in contrast, random pushes a single operand.

Approximately one-third of the Maté overhead is due to every instruction being executed in a separate TinyOS task, which requires an enqueue and dequeue operation. We ran the loop code for determining IPS except that every Maté task executed three instructions instead of one. The average IPS jumped to 12,754, which quantifies the task operations to be roughly 35% of Maté's overhead. This indicates a tradeoff between Maté concurrency and performance that could be adjusted for different applications.

5.3 Energy

Maté's computational overhead poses an energy overhead, as Maté must execute the additional instructions for interpretation. However, the concise representation of programs represents a savings in energy over full binary uploads; programs can be contained in a handful of packets instead of hundreds. Given the overhead data from the previous section, we can compute the energy overhead of Maté execution data over native code. Figure 9 gives a comparison of the size of different TinyOS programs in binary code and capsules. The application bless_test's Maté version is much larger than the others because instead of representing co-operation between a few subsystems, it implements a new one.

Uploading and installing an 8KB binary programs requires the mote to be fully active for roughly two minutes; this cost scales linearly with size [31]. By comparing binary programs with their equivalents in Maté, we can calculate the relative energy costs of installation. Given the energy cost of an execution and the energy cost of installation in the two systems, we can calculate at what point each of the two approaches is preferable. For a small number of executions, Maté is preferable; the energy cost of the CPU overhead is tiny in comparison to savings of only having to be awake to receive a single packet. For a large number of executions, native code is preferable; the savings on each execution overcomes the cost of installation.

As an example, we consider the application currently deployed on Great Duck Island. This application spends most of its time in a deep sleep mode that has a power draw of roughly 50 μA. Every eight seconds, the application wakes up, reads several sensors, and sends a packet containing the sensor data. Given the CPU overhead of Maté, the duty cycle of the GDI application and the energy cost of installing a native image, a Maté version running for five days or less will save energy over a binary version. After running for about six days, the energy cost of Maté's CPU overhead grows to
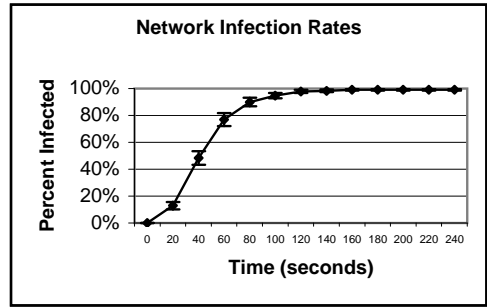

Figure 10: Percentage of Motes Running New Program Over Time

	Network			
New	12.5%	25%	50%	100%
12.5%	23 ± 8	28 ± 15	45 ± 24	361 ± 252
25%	10 ± 4	10 ± 4	19 ± 10	425 ± 280
50%	7 ± 3	7 ± 2	21 ± 10	226 ± 199
100%	8 ± 2	12 ± 5	14 ± 4	400 ± 339

Figure 11: Time to Complete Infection (seconds)

be greater than the cost of installing a binary version of the GDI program.

These relative energy costs suggest a clear use for Maté in energy constrained domains. While the interpretation overhead makes implementing complex applications entirely in Maté wasteful, infrequent invocations have a tiny energy cost; using capsules to reconfigure the native execution of an application (modifying sample rates, which sensors to sample, etc.) provides greatly improved flexibility at a much lower energy cost than installing a new application. Instead of building a new RPC-like mechanism for every application to control its configuration, applications can use Maté capsules as a general RPC engine.

5.4 Network Infection

To measure network infection rates, we deployed a 42 node network as a 3 by 14 grid. The radio transmission radius formed a 3 hop network; depending on the mote, cells varied from 15 to 30 motes in size.

Figures 10 and 11 contain the data we collected on network infection behavior. Figure 10 shows the rate at which motes in a network starting running a new program. In this experiment, configured every mote to run its clock capsule every twenty seconds. We introduced a mote to the network that ran a newer self-forwarding clock capsule. Every twenty seconds, we recorded how many motes were running the new program (the new program changed the LED pattern a mote displayed). We ran this experiment ten times, and averaged the results. The curve does not converge at one hundred percent because often one mote in the network (the same mote every time) would not reprogram for a long time, or at all – it was very resistant to a new viral capsule. We later inspected this mote and found it had a loose antenna; when re-soldered, it reprogrammed similarly to other motes.

Figure 11 shows the time a healthy network took to be fully reprogrammed. In this experiment, capsules had varying probabilistic forwarding rates. For each trial, we configured the network to have a clock capsule that ran once a second and forwarded itself with a certain probability. We

then introduced a mote into the network that had a new clock capsule that self-forwarded with another probability. We measured the elapsed time between the mote being introduced and the entire network being infected with the new program. We performed each infection six times.

The percentages in Figure 11 are the probabilities that a capsule would forward itself when run. There are two values: the forwarding probability of the running network, and the forwarding probability of the introduced capsule. The columns are the forwarding rates of the running network, while the rows are the forwarding rates of the introduced capsule. Each entry in the table shows the mean time to infection and the standard deviation between the infection runs. We chose a probabilistic scheme to prevent mote synchronization, which could artificially inflate network contention.

Increasing the forwarding rate of a capsule can increase the rate a program infects the network, but with diminishing returns. For example, increasing the forwarding probability from one eighth to one fourth often halved the time to network infection but increasing it further did not have nearly as notable an effect. The drastically higher time to infection for networks always forwarding their capsules is due to network congestion. The 10 kilobit radio can support roughly twenty packets per second after backoff, encoding, etc. As the maximum cell size of the network was approximately 30, capsule forwarding once per second resulted in the network going well past its saturation point.

6. DISCUSSION

The presence of an interpreter for dynamically loaded code qualitatively changes the behavior and usage model of wireless sensor networks. We discuss three ramifications of this change: phased programming of the network as a whole, interactions between static and dynamic layers in capsule forwarding and system architecture directions.

6.1 Phased Execution, Agility, and Active Sensors

The ease of installing new code in Maté means that programs which transition through several states can be written as a series of capsules. For example, to have a sense-report cycle in a network, one could first write a capsule that sensed the environment, placing this data in non-volatile storage with the `log` instruction. Once the data acquisition is complete, one could inject a new program that reads in stored data entries (with the `logr` instruction) then sends them over the network to be collected. The lightweight nature of capsules also makes them excellent candidates as a mechanism for experimenting with sensor network application agility [27]; the Great Duck Island use case is an example of this.

Currently, Maté executes capsules in response to only three types of events. One could imagine extending Maté to have contexts and capsules associated with a much richer set of activating primitives. Active networks ran code in response to only network events; the possibility of running easily installable code in response to such things as sensor thresholds, signal detection, or real-world actuation expands this idea from an active network node into an active sensor.

6.2 Capsule Forwarding

The results from our network reprogramming experiments establish that application control of the propagation rate is undesirable. There are obviously more efficient possibilities; one would be to tag whether capsules should forward or not. If a capsule is tagged, Maté could broadcast the capsule at a rate appropriate for the network density, effectively adapting to a forwarding rate that the network could sustain. This would not necessarily slow down the rate of programming – in a dense network, more motes would hear a forwarded capsule than in a sparse one. This issue gets at a fundamental limitation in the current TinyOS design; motes can actuate a network (send packets), but there is no mechanism to sense how busy the network is. If TinyOS included such a mechanism, Maté could provide mechanisms such as message merge capsules, which are called to ask an application to aggregate data buffers when it is sending data more rapidly than the network can handle.

Additionally, the `setgrp` instruction allows Maté to construct several logical networks on top of a single physical network by changing a mote's AM group ID. This raises the question of whether several logically separate networks running different applications could share common infrastructure, such as an ad-hoc routing layer. The current TinyOS AM group mechanism prevents such a system from being implemented.

6.3 Architectural Directions

Motes currently do not have the traditional user/kernel boundary enforced by hardware protection mechanisms; a badly written application component can cause all of TinyOS to fail. Maté's interface solves this problem – a program cannot, among other things, disable interrupts or write to arbitrary memory locations. The need for user-land is supplanted by VM-land, which can provide the same guarantees to applications. Maté contexts are also much smaller than hardware contexts; allocating multiple C stacks is nigh impossible on the `rene2` without putting harsh limits on call depth.

Given their size, it is no surprise that motes do not have hardware support for virtual memory. Maté could provide functionality equivalent of many of the benefits a virtual memory system brings. As it is a virtual machine, Maté can provide a virtual address space. Swapping to backing store for suspended programs can be provided by using a mote's non-volatile storage. Motes could enter low-power sleep states with RAM powered down and restore an execution context on waking up, even possibly running under a different version of Maté.

Currently, Maté has eight instructions reserved for user specific operations. As all of the other instructions, these user instructions are part of the Maté binary image; they must be defined when Maté is installed, and cannot be changed. While subroutines allow some execution flexibility, Maté's overhead means that any non-trivial mathematical operation is infeasible. Being able to load binary code for user instructions in a manner similar to subroutines would greatly improve Maté; the fact that motes are Harvard architectures prevents this from being effectively and safely implemented. For Maté to be efficiently tailorable in-situ, motes should have a unified data and instruction memory, or at least the capability to execute instructions out of data memory.

7. CONCLUSION

For sensor networks to be widely adopted, they must be easy to use. We have defined a set of system requirements for ease of sensor network programming and presented a tiny virtual machine, Maté, which meets these requirements. Maté is small and expressive, has concise programs that are resilient to failure, provides efficient network and sensor access, can be tailored to specific domains, and can quickly self-program a network. The effectiveness of Maté as an execution model suggests that virtual machines are a promising way to provide protective hardware abstractions to application code in sensor networks, fulfilling the traditional role of an operating system. Clearly, given the autonomous nature of viral reprogramming, significant and possibly novel security measures must be taken to protect a network. While an application composed of a static organization of components might evolve slowly, Maté makes the network dynamic, flexible and easily reconfigurable. This suggests Maté can participate in the management of networks in addition to being a platform for application development.

Our future work on Maté focuses on two areas: application-specific virtual machines and the user-land abstraction. We are currently looking at application-specific Maté flavors for identified domain requirements. In addition to being a top-level interface for mote programming, the VM can also sit below other TinyOS components as a computational engine in certain domains. For example, we have developed a version named Bombilla to provide an execution engine to the TeleTiny system, a miniature query processor and data aggregation system for SQL-like queries on a network [21]. In its current state, Maté is only an architecture and bytecodes; the next step is to develop higher level languages and programming models for application development, providing a user-land programming environment distinct from TinyOS.

Acknowledgments

8. REFERENCES

[1] Smart buildings admit their faults. Lab Notes: Research from the College of Engineering, UC Berkeley. http://coe.berkeley.edu/labnotes/1101.smartbuildings.html, 2001.

[2] Small Times: Big News in Small Tech. http://www.smalltimes.com

[3] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, 1997.

[4] David Culler, Anurag Sah, Klaus Schauser, Thorsten von Eicken and John Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[5] Lloyd I. Dickman. Small Virtual Machines: A Survey. In *Proceedings of the Workshop on Virtual Computer Systems*, 1973.

[6] Wolfgang Emmerich, Cecilia Mascolo, and Anthony Finkelstein. Implementing Incremental Code Migration with XML. In *Proceedings of the 22nd International Conference on Software Engineering*, 2000.

[7] Deborah Estrin, Ramesh Govindan, John Heidemann, and Satish Kumar. Next Century Challenges: Scalable Coordination in Sensor Networks. In *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking*, 1999.

[8] Virginie Galtier, Kevin L. Mills, Yannick Carlinet, Stefan Leigh, and Andrew Rukhin. Expressing Meaningful Processing Requirements Among Heterogeneous Nodes in an Active Network. In *Proceedings of the Second International Workshop on Software and Performance*, 2000.

[9] John Heidemann, Fabio Silva, Shalermek Intanagonwiwat, Ramesh Govindan (USC/ISI), Deborah Estrin, Deepak Ganesan (UCLA). Building Efficient Wireless Sensor Networks with Low-Level Naming. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, 2001.

[10] Jason Hill and David Culler. A wireless embedded sensor architecture for system-level optimization. Intel Research IRB-TR-02-00N, 2002.

[11] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System Architecture Directions for Networked Sensors. In the *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[12] Victor Hsu, Joseph M. Kahn, and Kristofer Pister. Wireless Communications for Smart Dust. Electronics Research Laboratory Technical Memorandum Number M98/2, 1998.

[13] Chalermek Intanagonwiwat, Deborah Estrin, Ramesh Govindan, and John Heidemann. Impact of Network Density on Data Aggregation in Wireless Sensor Networks. Technical Report 01-750, University of Southern California Computer Science Department, 2001.

[14] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking*, 2000.

[15] Java 2 Platform Micro Edition (J2ME) Technology for Creating Mobile Devices White Paper. http://java.sun.com/j2me/docs/

[16] Java Card Technology. http://java.sun.com/products/javacard/

[17] Richard K. Johnsson, John D. Wick. An Overview of the Mesa Processor Architecture. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, 1982.

[18] Joseph M. Kahn, Randy H. Katz, Kristofer Pister. Next Century Challenges: Mobile Networking for "Smart Dust." In *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking*, 1999.

[19] Philip J. Koopman, Jr. Modern Stack Computer Architecture. In *System Design and Network Architecture Conference*, 1990.

[20] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification, Second Edition. Addison-Wesley, 1999.

[21] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In submission.

[22] Samuel R. Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously Adaptive Continuous Queries over Streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2002.

[23] Alan Mainwaring, private communication, 2002.

[24] Dejan S. Milojicic, Fred Douglas, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process Migration. In *ACM Computing Surveys*, Volume 32, Issue 3, September 2000.

[25] Charles H. Moore and Geoffrey C. Leach. FORTH - A Language for Interactive Computing. 1970.

[26] Michael Noakes, Deborah Wallach, and William J. Dally. The J-Machine Multicomputer: An Architectural Evaluation. In *Proceedings of the 20th International Symposium on Computer Architecture*, 1993.

[27] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, 1997.

[28] Charles E. Perkins, Elizabeth M. Belding-Royer, and Samir Das. Ad Hoc On Demand Distance Vector (AODV) Routing. IETF Internet draft, draft-ietf-manet-aodv-09.txt, November 2001 (Work in Progress).

[29] Adrian Perrig, Robert Szewczyk, Victor Wen, David Culler and J.D. Tygar. SPINS: Security Protocols for Sensor Networks. In *The Proceedings of Mobicom 2001*, 2001.

[30] Phillip Stanley-Marbell and Liviu Iftode. Scylla: A Smart Virtual Machine for Mobile Embedded Systems. In *Proceedings of The 3rd IEEE Workshop on Mobile Computing Systems and Applications*, 2000.

[31] Robert Szewczyk, private communcation, 2002.

[32] David L. Tennenhouse and David J. Wetherall. Towards an Active Network Architecture. In *Computer Communication Review*, Vol. 26, No. 2, April 1996.

[33] Thorsten von Eicken, David Culler, Seth Goldstein, and Klaus Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, 1992.

[34] David J. Wetherall, John V. Guttag and David L. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. *IEEE OPENARCH '98*, 1998.

[35] Alexander L. Wolf, Dennis Heimbigner, John Knight, Premkumar Devanbu, Michael Gertz, and Antonio Carzaniga. Bend, Don't Break: Using Reconfiguration to Achieve Survivability. *Third Information Survivability Workshop*, October 2000.

APPENDIX

The complete Maté instruction set follows, as well as our Maté implementation of the ad-hoc routing protocol BLESS.

A. Maté ISA

MESSAGE HEADER		ROUTING DATA FRAME	
Destination address		Parent address	
Source address		Hopcount	Parent heard?
Hopcount			

Figure 12: BLESS Message Header and Frame Layouts

```
16  OPcast    0x10   00010000   push(const($0))
17  OPpushm   0x11   00010001   push(message)
18  OPmovm    0x12   00010010   push(pull entry off $0)
19  OPclear   0x13   00010011   clear($0), don't pop it
20  OPson     0x14   00010100   turn sounder on
21  OPsoff    0x15   00010101   turn sounder off
22  OPnot     0x16   00010110   push(~$0)
23  OPlog     0x17   00010111   logwrite($0) onto stable storage
24  OPlogr    0x18   00011000   read(line $1 into msg $0)
25  OPlogr2   0x19   00011001   read(line $0 into msg $1)
26  OPsets    0x1a   00011010   set shared variable to $0
27  OPgets    0x1b   00011011   push(shared variable)
28  OPrand    0x1c   00011100   push 16 bit random number onto stack
29  OPeq      0x1d   00011100   if $0 == $1, push 1 else 0
30  OPneq     0x1e   00011101   if $0 != $1, push 1 else 0
31  OPcall    0x1f   00011111   call $0
32  OPswap    0x20   00100000   swap $0 and $1

46  OPforw    0x2e   00101110   forward this code capsule
47  OPforwo   0x2f   00101111   forward capsule $0

48  OPusr0    0x30   00110000   user instruction 0
49  OPusr1    0x31   00110001   user instruction 1
50  OPusr2    0x32   00110010   user instruction 2
51  OPusr3    0x33   00110011   user instruction 3
52  OPusr4    0x34   00110100   user instruction 4
53  OPusr5    0x35   00110101   user instruction 5
54  OPusr6    0x36   00110110   user instruction 6
55  OPusr7    0x37   00110111   user instruction 7

58  OPsetgrp  0x3a   00111010   set group id to $0
59  OPpot     0x3b   00111011   push(potentiometer setting)
60  OPpots    0x3c   00111100   set radio potentiometer to $0
61  OPclockc  0x3d   00111101   set clock counter with $0
62  OPclockf  0x3e   00111110   set clock freq with $0 (0-7)
63  OPret     0x3f   00111111   return from subroutine

SCLASS
64  OPgetms   0x40-47  01000xxx   push(short xxx from msg header)
72  OPgetmb   0x48-4f  01001xxx   push(byte xxx from msg header)
80  OPgetfs   0x50-57  01010xxx   push(short xxx from frame)
88  OPgetfb   0x58-5f  01011xxx   push(byte xxx from frame)
96  OPsetms   0x60-67  01100xxx   short xxx of msg header = $0
102 OPsetmb   0x68-6f  01101xxx   byte xxx of msg header = $0
108 OPsetfs   0x70-77  01110xxx   short xxx of frame = $0
114 OPsetfb   0x78-7f  01111xxx   byte xxx of frame = $0

XCLASS
128 OPpushc   0x80-bf  10xxxxxx   push(xxxxx)  (unsigned)
192 OPblez    0xC0-ff  11xxxxxx   if ($0 <= 0) jump xxxxxx
```

B. BLESS

B.1 Clock Capsule

```
pushc 1
add
copy       # Keeping a counter -- copy its value
pushc 4    # Timer is every 4 clock ticks

inv
add
blez 11    # If the timer hasn't expired, skip flush check (jumps after call)
pop        # Parent flush check code - get rid of old timer

pushc 0    # Reset timer
pushc 3
call       # Call subroutine 3 -- check for parent flush
pushc 1    # Jump-to point for branch above

sense
pushm
clear
add        # Read light sensor, put value in message

sendr      # Send message with send capsule
pushc 1
pushc 3
pushc 3

shiftl
or         # Create bit pattern 0x9
putled     # 0x9 toggles red LED
halt
```

B.2 Message Send

```
0   OPhalt    0x00   00000000
1   OPreset   0x01   00000001   clear stack
2   OPand     0x02   00000010   push($0 & $1)
3   OPor      0x03   00000011   push($0 | $1)
4   OPshiftr  0x04   00000100   push($0 >> $1) (signed)
5   OPshiftl  0x05   00000101   push($0 << $1) (signed)
6   OPadd     0x06   00000110   push($0 + $1) -- depends on types
8   OPputled  0x08   00001000   $0 used as 2-bit cmd + 3-bit oprnd
9   OPid      0x09   00001001   push(moteID)
10  OPinv     0x0a   00001010   push(-$0)
11  OPcopy    0x0b   00010000   copy $0 on top of stack
12  OPpop     0x0c   00001100   (pop $0)
13  OPsense   0x0d   00001101   push(sensor($0))
14  OPsend    0x0e   00001110   send($0)
15  OPsendr   0x0f   00001111   send($0) with capsule 5
```

```
pushc 0
not
getfs 0
neq       # If our parent != 0xffff (no parent), we won't branch

blez 16   # If our parent is 0xffff (no parent), we'll skip sending
getfs 0   # Get short 0 of frame -- parent addr
setms 0   # Set short 0 of header -- destination addr
id

setms 1   # Set source field (short 1) of header to our addr
getfb 2   # Get our hopcount
setmb 4   # Set hopcount field in header
pushc 0

not       # Create AM broadcast addr (0xffff)
sendr     # Send packet on broadcast
pushc 4
putled    # Turn on green led

halt      # Jump-to point of above branch
```

B.3   Message Receive

```
pushc 0
call
pushc 1
call      # Check if we should make sender our new parent

pushc 2
call      # If it's our current parent, flag parent heard
getms 0   # Get desination addr of packet
id

eq
blez 13   # Branch to halt if we're not the destination
pushc 2
putled    # Turn on green LED

sendr     # Send packet on AM broadcast
halt
```

B.4   Subroutine 0 – If no parent, set hopcount to 64

```
getfs 0
pushc 0
not
eq

blez 9     # If parent != 0xffff (no parent), jump to ret
pushc 1
pushc 6
shiftl

setfb 2    # Set hopcount to 64
ret
```

B.5   Subroutine 1 – Change Parent?

```
getfb 2
getmb 4
inv
add

blez 11     # If msg hopcount greater or equal to our hopcount, jump to ret
getms 1
setfs 0     # Set our parent addr to source addr of packet
getmb 4

pushc 1
add
setfb 2     # Set our hopcount to hopcount of source + 1
ret
```

B.6   Subroutine 2 – Flag Parent Heard?

```
getms 1
getfs 0
eq
blez 10     # If not our parent, jump to ret

pushc 1
setfb 3     # Set heard-parent flag of frame to 1
getmb 4
pushc 1

add
setfb 2     # Set our hopcount to parent hopcount + 1
ret
```

B.7   Subroutine 3 – Flush Parent?

```
getfb 3
pushc 0
eq
blez 7    # If we have heard our parent, skip parent clear

pushc 0
not
setfs 0   # Clear parent
pushc 0

setfb 3   # Set parent not-heard
pushc 1
pushc 4
shiftl

inv       # Create -16
getfb 4
add
blez  19  # skip if hopcount <= 16

pushc 0
inv
setfs 0   # Set parent to 0xffff (no parent) -- too far from root
ret
```