



CS649

Sensor Networks

Lecture 25: Reprogramming

Andreas Terzis

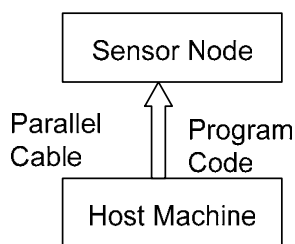
<http://hinrg.cs.jhu.edu/wsn06/>

Outline

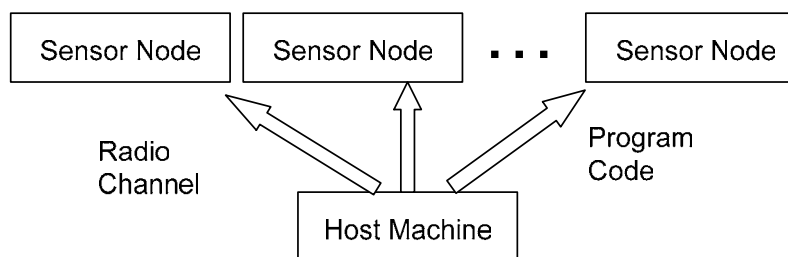
- Problem: Reprogram the network after it is deployed
- Alternatives
 - Ship new binary
 - Ship "script"
- Common issues
 - Reliable broadcast
- Different challenges
 - Execution engine for scripting language
 - Code size

Programming Wireless Sensors

- In-System Programming (ISP)
 - A sensor node is plugged to the serial / parallel port.
 - But, it can program only one sensor node at a time.
- Network Programming
 - Delivers the program code to multiple nodes over the air with a single transmission.
 - Saves the efforts of programming each individual node.



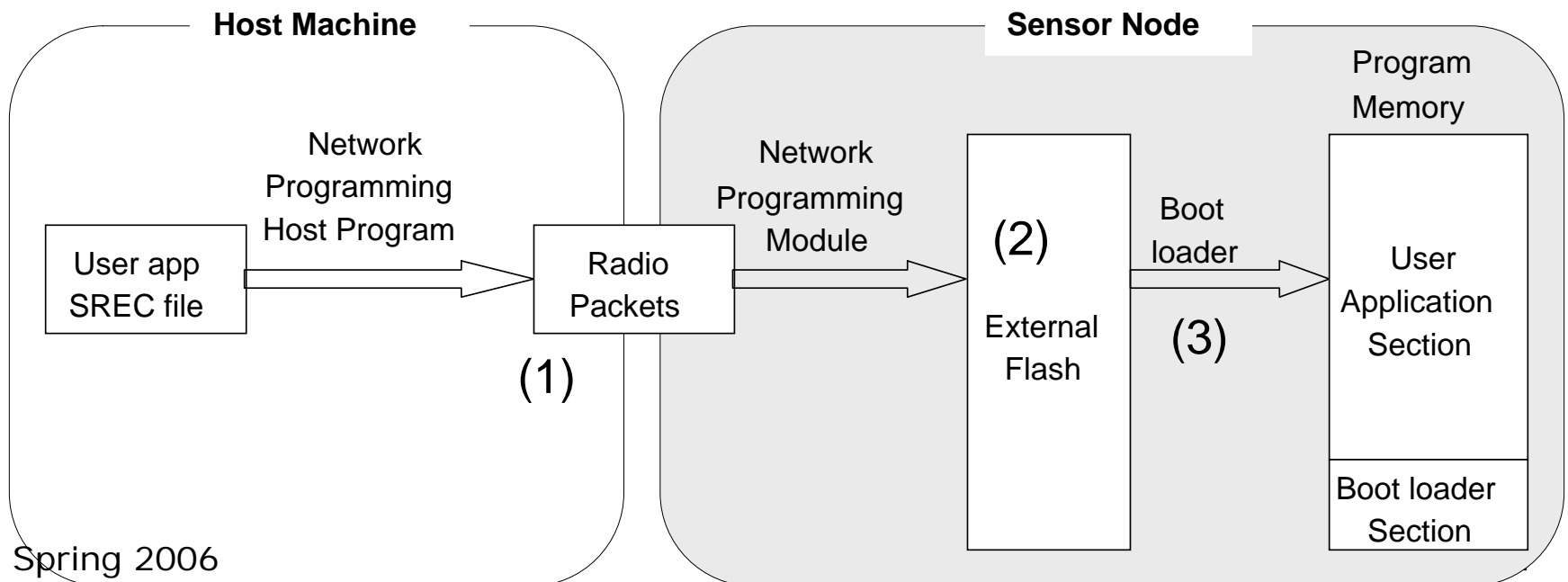
In-system programming



Network programming

Background – Mechanisms of XNP

- (1) Host: sends program code as download msgs.
- (2) Sensor node: stores the msgs in the external flash.
- (3) Sensor node: calls the boot loader. The boot loader copies the program code to the program memory.





Incremental Network Programming for Wireless Sensors

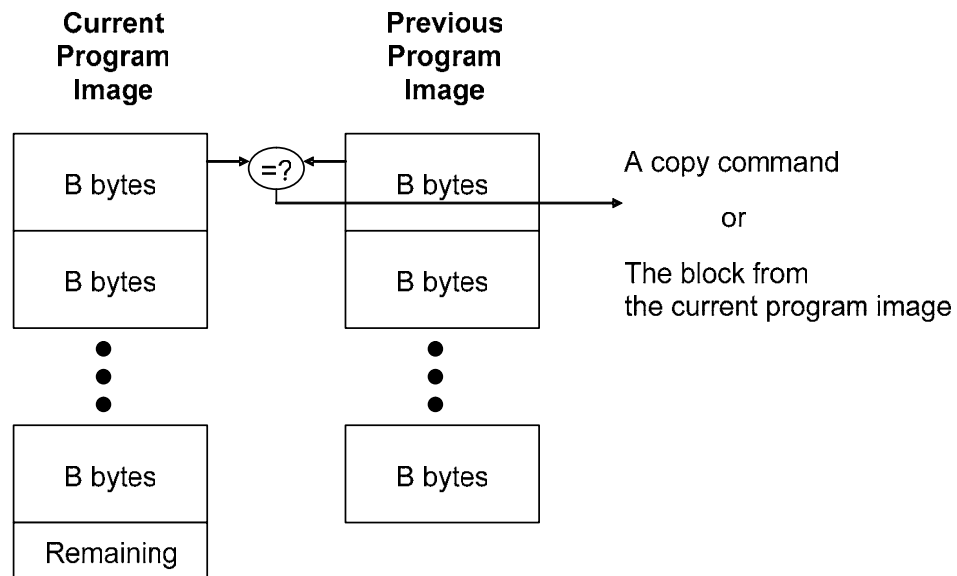
Jaein Jeong
IEEE SECON 2004

High Level Idea

- Usually code changes little between different versions
 - Change constants
 - Modify part of the implementation
 - Modify part of configuration
- Instead of sending the whole binary send only the “diffs”

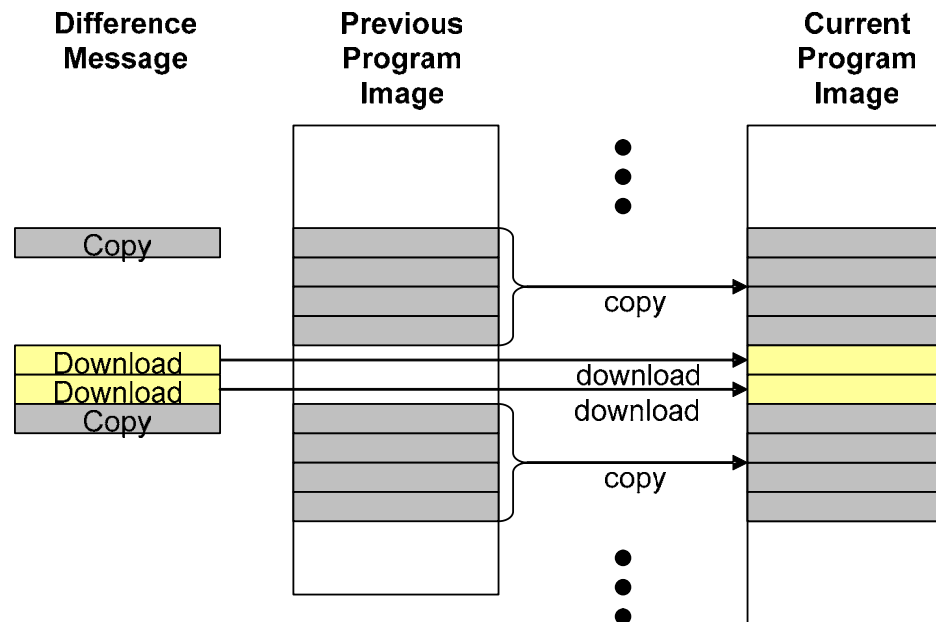
First Approach – Fixed Block Comparison (Difference Generation)

- The host program
 - Generates the difference of the two program versions.
 - Compares the corresponding blocks of the two program images.
 - Sends “copy” for the matching blocks, “download” for the unmatched ones.



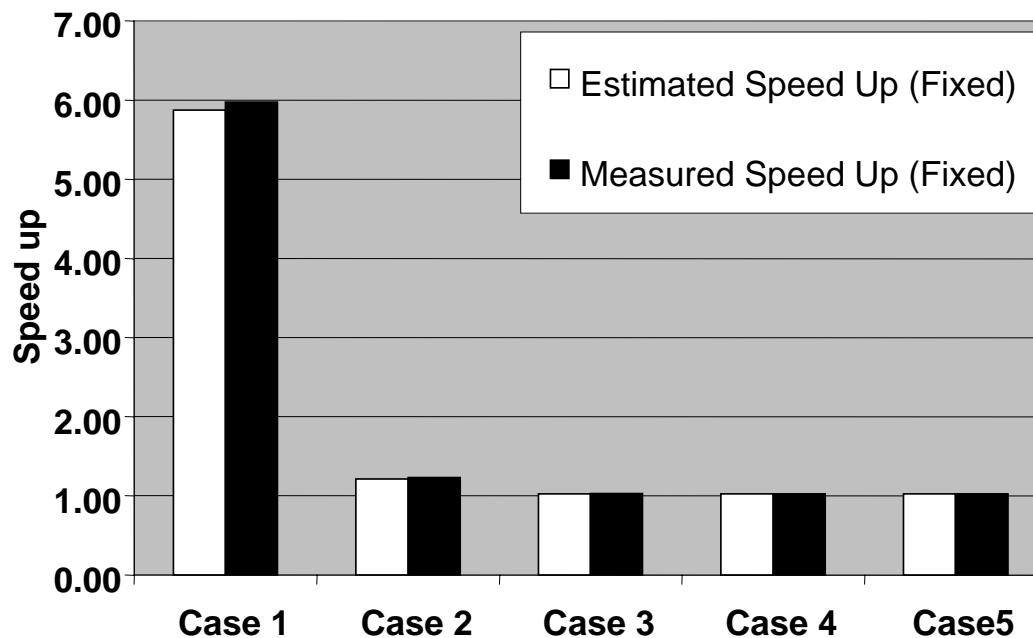
First Approach – Fixed Block Comparison (Storage Organization and Code Rebuild)

- The sensor node
 - Keeps new and previous sections in the external flash.
 - For “copy”, copies the records from the prev section to new section.
 - For “download”, stores the code bytes in the new section.



Results – Fixed Block Comparison

- Compare the transmission time (T) with the non-incremental delivery time (T_{xnp}).
- There is almost no speed up except when constant in the source code was modified (case 1).



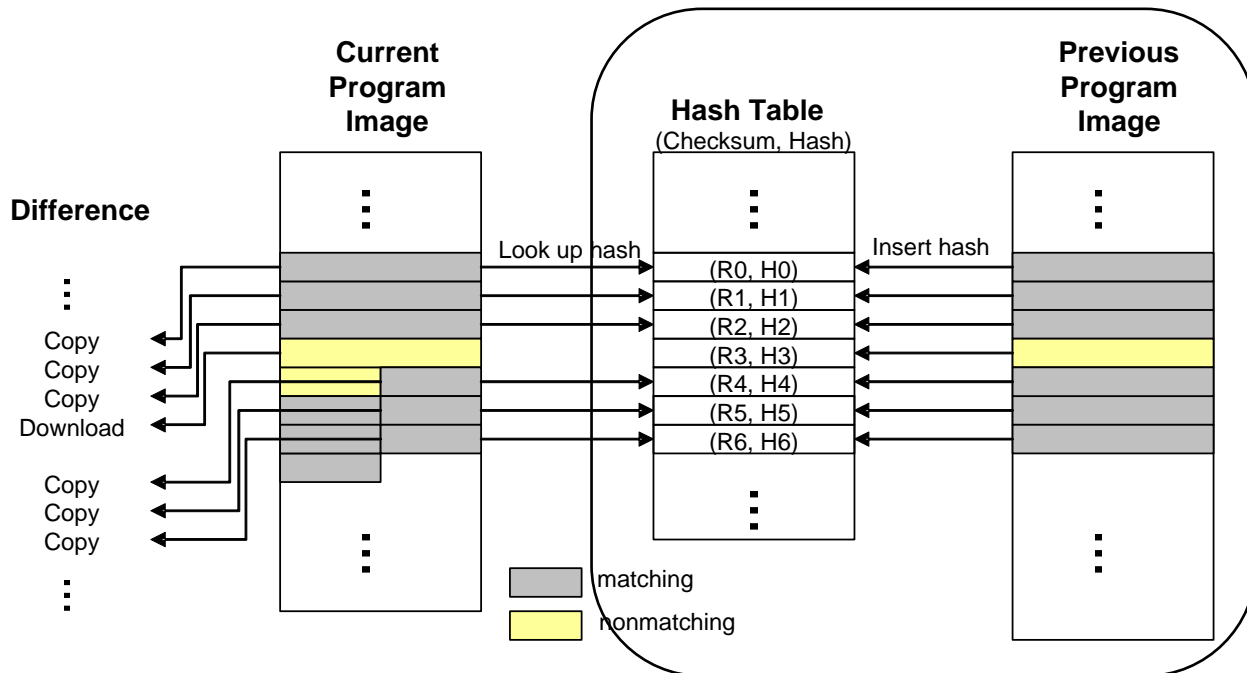
Optimizing Difference Generation using Rsync algorithm

- Need an algorithm that can capture the shared data even when the program is shifted.
- Use Rsync algorithm to compare the two program images at an arbitrary byte position.
- Rsync was originally created to transfer the incremental update of arbitrary binary file over a low bandwidth Internet connection.

Difference Generation using Rsync algorithm (Diff Generation)

(1) Build the hash table for the previous image.

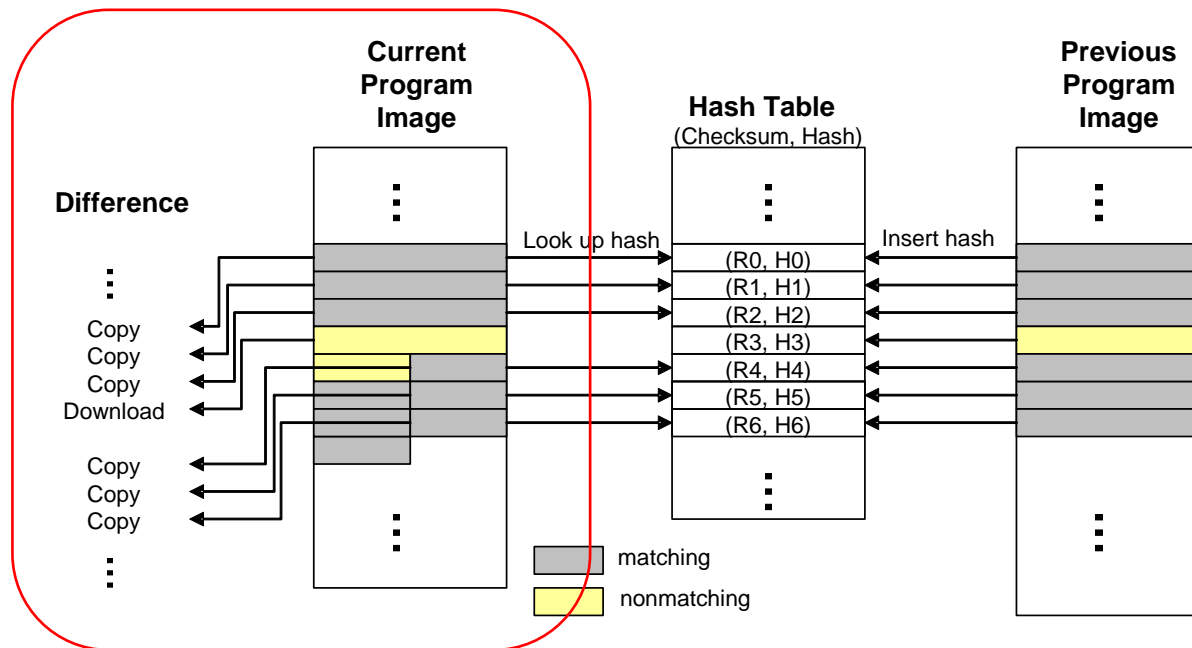
- Calculate the checksum pair (checksum, hash) for each block.
- The checksum is for the fast match.
- The hash is for more accurate match.



Difference Generation using Rsync algorithm (Diff Generation)

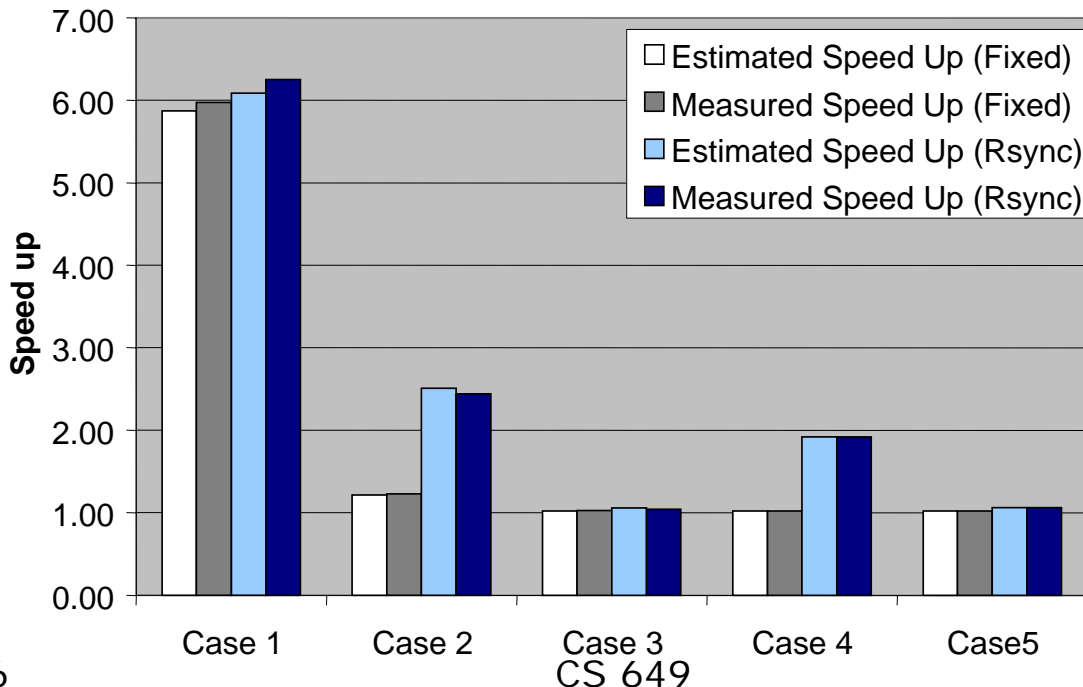
(2) Scan the current image and find the matching block.

- Calculate the checksum for the block at each byte.
- Calculate the hash only if the checksum matches.
- Moves to the next byte and recalculates the checksum if the block doesn't match.



Results – Using Rsync

- The performance got better than Fixed Block Comparison.
- Speed up of 2 to 2.5 for a small change.
 - 2.5 for adding a few lines in implementation file (case 2).
 - 2.0 for commenting out IntToLeds in the configuration file (case 4).
- Still limited speed up for big changes.
 - Major change (case 3).
 - Commenting out IntToRfm (case 5).





"Maté: A Tiny Virtual Machine for Sensor Networks.

Phil Levis and David Culler
Appeared in ASPLOS 2002

Motivation

- TinyOS programming complex
- Application flexibility needed
- Binary reprogramming takes ~2 minutes
 - Significant energy cost
 - Can lose motes

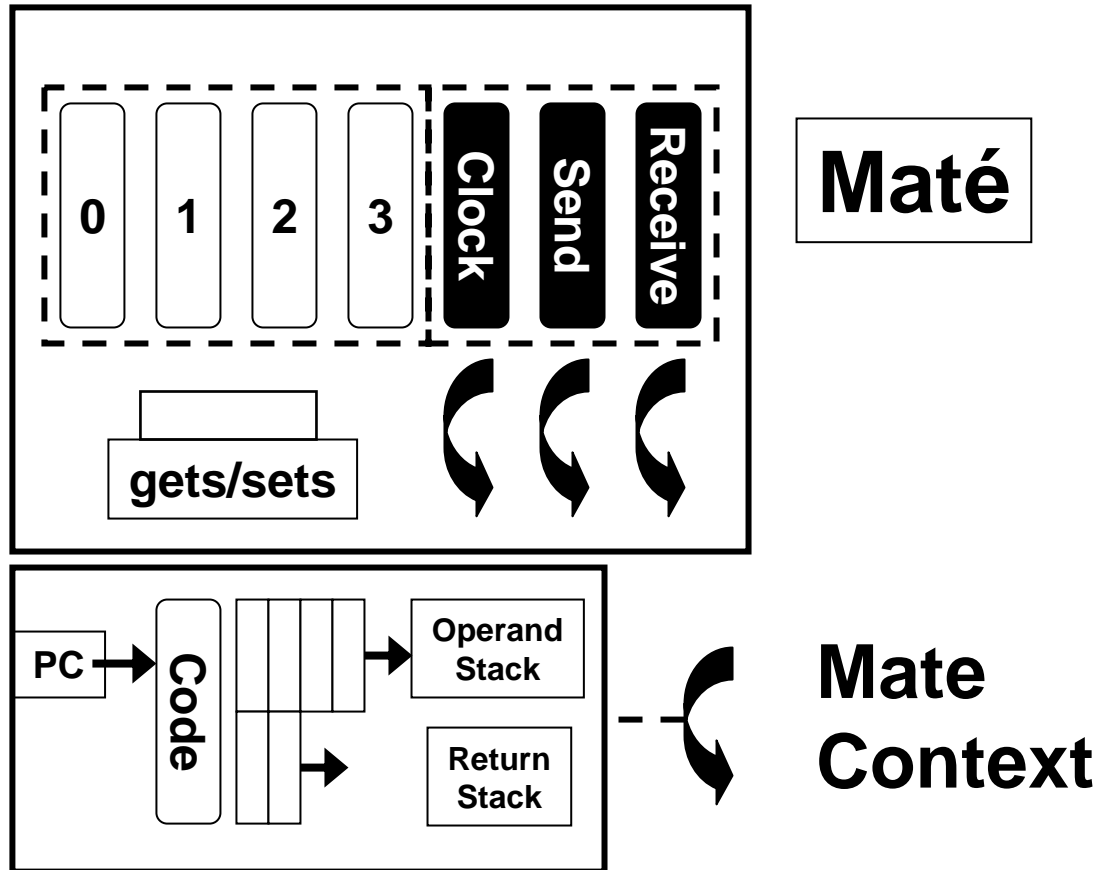
Maté Overview

- TinyOS component
- 7286 bytes code, 603 bytes RAM
- Three concurrent execution contexts
- Stack-based bytecode interpreter
- Code broken into 24 instruction capsules
- Self-forwarding code
- Rapid reprogramming
- Message receive and send contexts

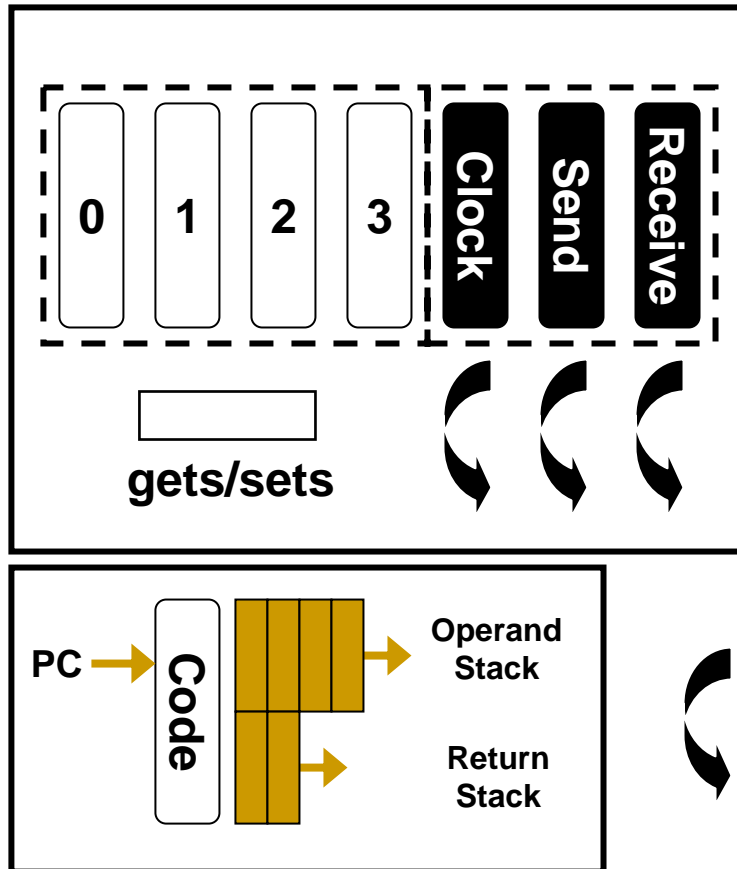
Maté Overview, Continued

- Three execution contexts
 - Clock, Receive, Send
- Seven code capsules
 - Clock, Receive, Send, Subroutines 0-3
- One word heap
 - `gets/sets` instructions
- Two-stack architecture
 - Operand stack, return address stack

Maté Architecture



Maté Instructions



Maté

**Mate
Context**

Maté Instructions

- Two-stack architecture
- One byte per instruction
- Three classes: basic, s-type, x-type
 - basic: data, arithmetic, communication, sensing
 - s-type: used in send/receive contexts
 - x-type: embedded operands

basic	00iiiiiii	i = instruction
s-type	01iiixxx	x = argument
x-type	1ixxxxxxx	

Code Snippet: cnt_to_leds

```
gets      # Push heap variable on stack
pushhc 1  # Push 1 on stack
add       # Pop twice, add, push result
copy     # Copy top of stack
sets     # Pop, set heap
pushhc 7  # Push 0x0007 onto stack
and      # Take bottom 3 bits of value
putled   # Pop, set LEDs to bit pattern
halt     #
```

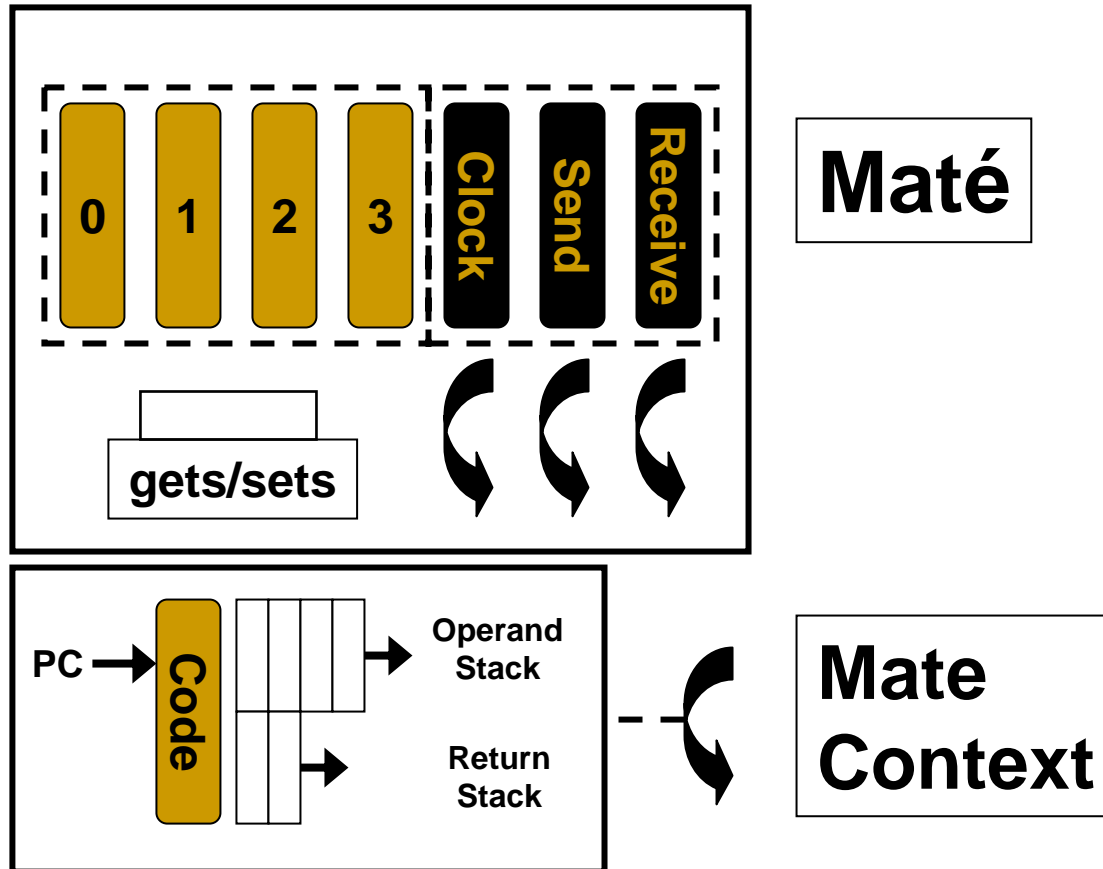
cnt_to_leds, Binary

```
gets          # 0x1b
pushc 1       # 0xc1
add           # 0x06
copy         # 0x0b
sets         # 0x1a
pushc 7       # 0xc7
and          # 0x02
putled       # 0x08
halt         # 0x00
```

Sending a Message

```
pushc 1    # Light is sensor 1
sense      # Push light reading on stack
pushm      # Push message buffer on stack
clear      # Clear message buffer
add        # Append reading to buffer
send       # Send message using built-in
halt       # ad-hoc routing system
```

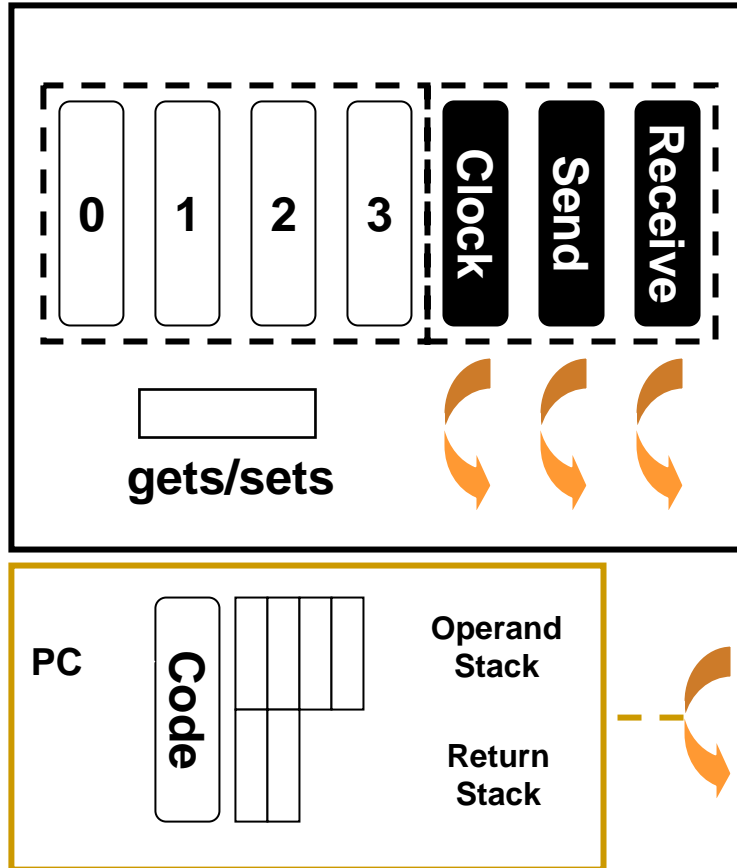
Maté Capsules



Maté Capsules

- Hold up to 24 instructions
- Fit in a single TinyOS AM packet
 - Installation is atomic
- Four types: send, receive, clock, subroutine
- Context-specific: send, receive, clock
- Called: subroutines 0-3
- Version information

Maté Contexts



Maté

**Mate
Context**

Contexts

- Each context associated with a capsule
- Executed in response to event
 - external: clock, receive
 - internal: send (in response to `sendr`)
- Execution model
 - preemptive: clock
 - non-preemptive: send, receive
- Every instruction executed as TinyOS task

Viral Code

- Every capsule has version information
- Maté installs newer capsules it hears on network
- Motes can forward their capsules (local broadcast)
 - `forw`
 - `forwo`

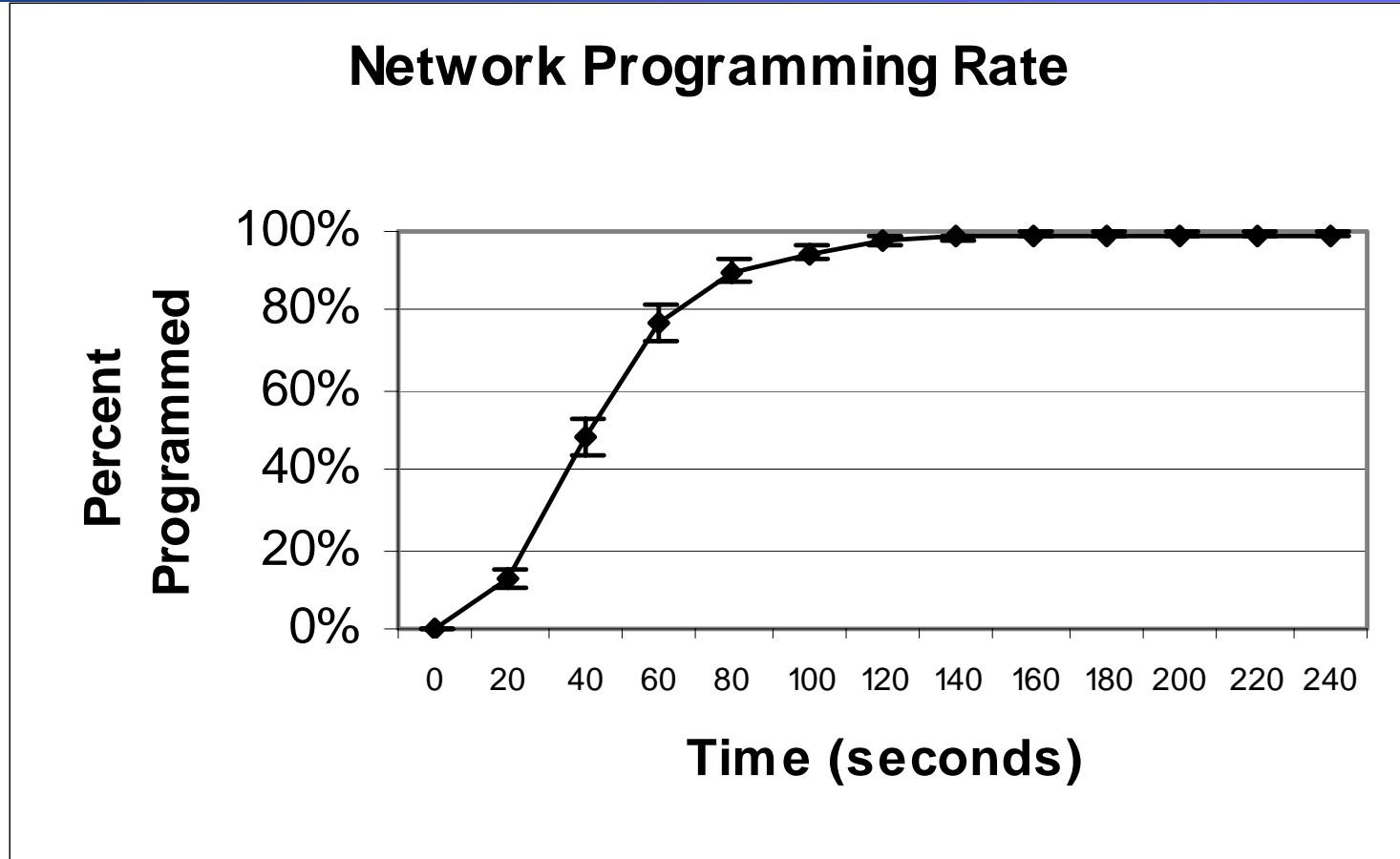
Forwarding: cnt_to_leds

```
gets      # Push heap variable on stack
pushc 1   # Push 1 on stack
add       # Pop twice, add, push result
copy      # Copy top of stack
sets     # Pop, set heap
pushc 7   # Push 0x0007 onto stack
and       # Take bottom 3 bits of value
putled   # Pop, set LEDs to bit pattern
forw    # Forward capsule
halt     #
```

Evaluation

- Code Propagation
- Execution Rate
- 42 motes: 3x14 grid
- 3 hop network
 - largest cell 30 motes
 - smallest cell 15 motes

Code Propagation



Code Propagation, Continued

(seconds)

	Network: 1/8		Network: 1/4	
New	Mean	Std. Dev.	Mean	Std. Dev
1/8	23	8	28	15
1/4	10	4	10	4
1/2	7	3	7	2
1/1	8	2	12	5

	Network: 1/2		Network: 1/1	
New	Mean	Std. Dev.	Mean	Std. Dev
1/8	45	24	361	252
1/4	19	10	425	280
1/2	21	10	226	199
1	14	4	400	339

Maté Instruction Issue Rate

- ~10,000 instructions per second
- Task operations are 1/3 of Maté overhead

Energy Consumption

- Compare with binary reprogramming
- Maté imposes a CPU overhead
- Maté provides a reprogramming time savings
- Energy tradeoff

Case Study: GDI

- Great Duck Island application
- Simple sense and send loop
- Runs every 8 seconds – low duty cycle
- 19 Maté instructions, 8K binary code
- Energy tradeoff: if you run GDI application for less than 6 days, Maté saves energy