

Incremental Network Programming for Wireless Sensors

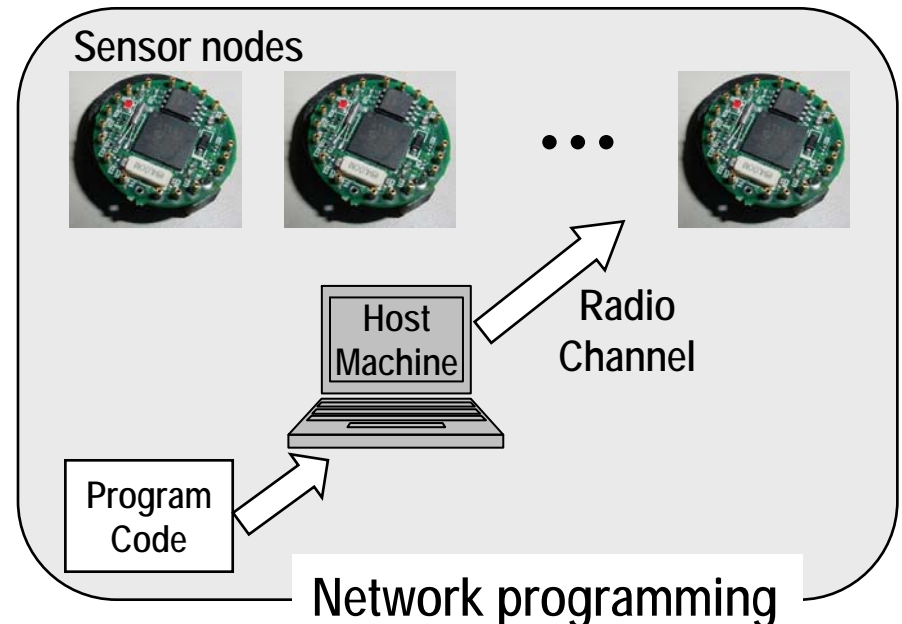
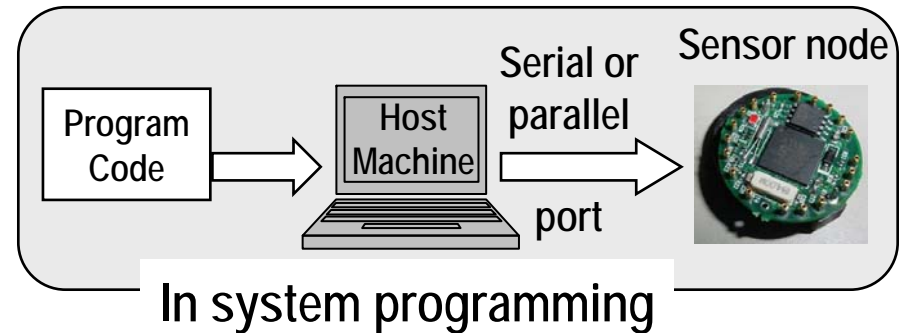
IEEE SECON 2004

Jaein Jeong and David Culler

UC Berkeley, EECS

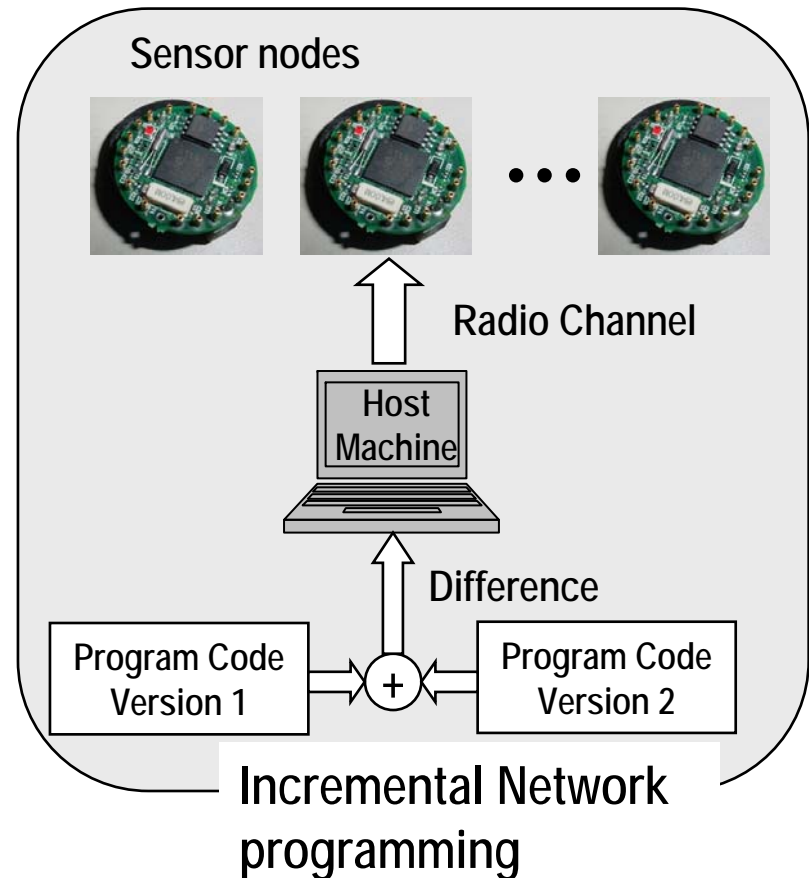
Introduction – *Loading Program to Wireless Sensors*

- In System Programming
 - Most Common.
 - Programming time is in proportion to # nodes.
- Network Programming
 - Sending whole code over radio still takes time.



Introduction - *Incremental Network Programming*

- Program source code is changed in small amounts.
- Reduce programming time by sending the difference.

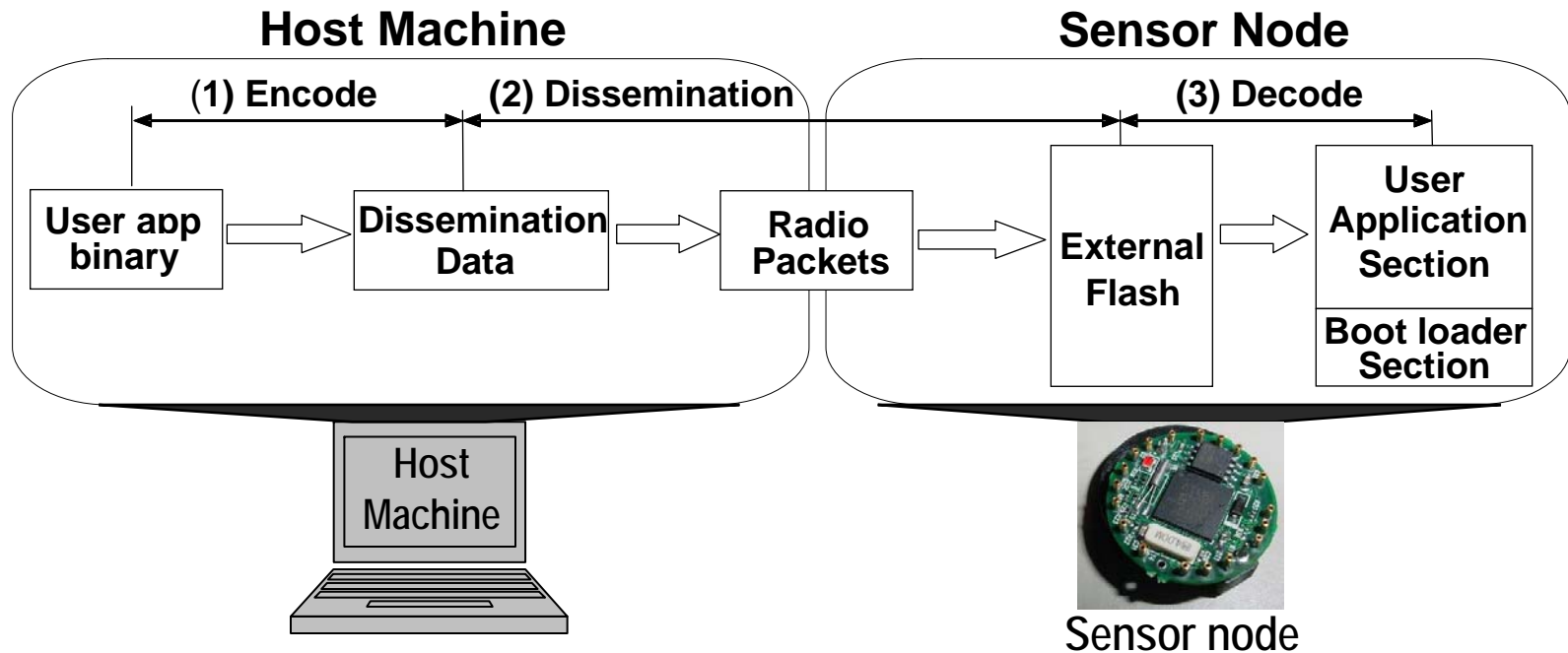


Previous Work

- Single-hop Network Programming: *XNP*
- Multi-hop network programming: *MOAP, Deluge*
 - *Extends the Range*
- Incremental network programming: *Reijers / Kapur*
 - *Reduces the Programming Time*
- Virtual machine programming: *Maté / Trickle*
 - *Small Application Level Code*
- Our incremental network programming approach
 - *Difference Generation using Rsync.*
 - *Platform independent solution.*

Network Programming Steps - *Incremental Network Programming*

- (1) Encoding: Generates the difference.
- (2) Dissemination: Transmits the difference.
- (3) Decoding: Rebuilds the new code.

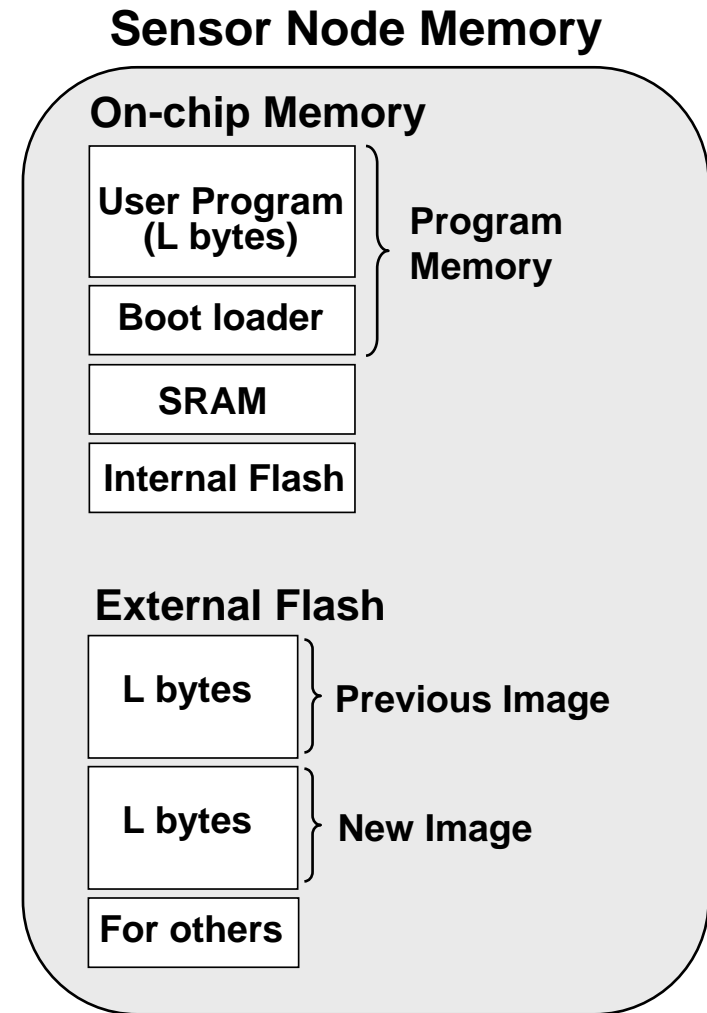


Design Considerations

- Reduce the amount of data transmission.
- Minimize the access to the external flash memory.
- Avoid expensive operations for sensor nodes.

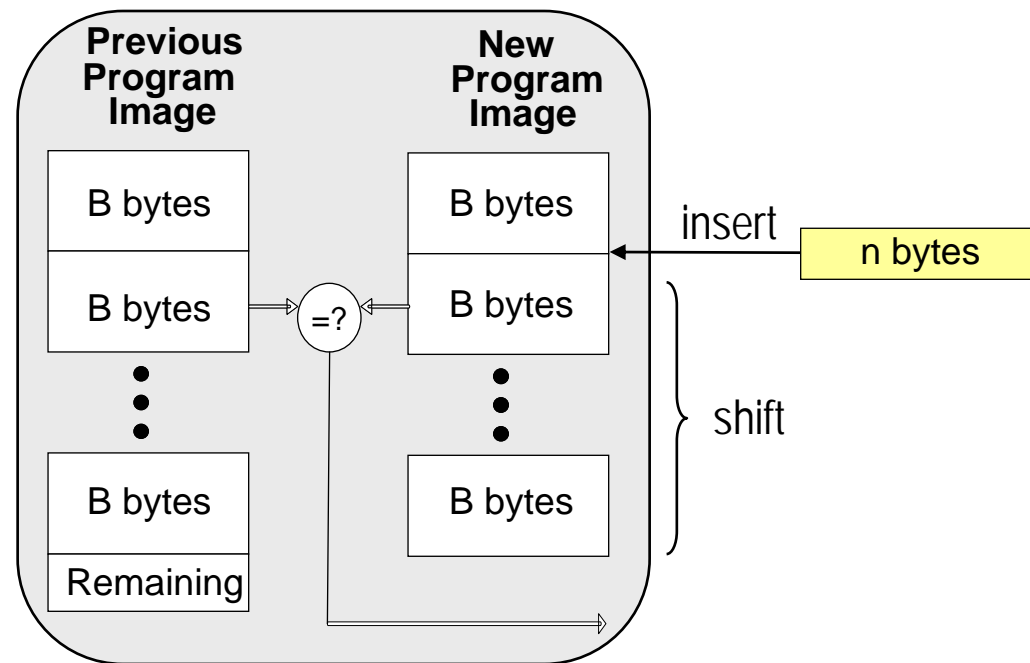
Step 1: Difference Generation (Encoding) – Storage Organization

- Program Memory
 - Running Program
- External Flash Memory
 - Program images for previous / current version



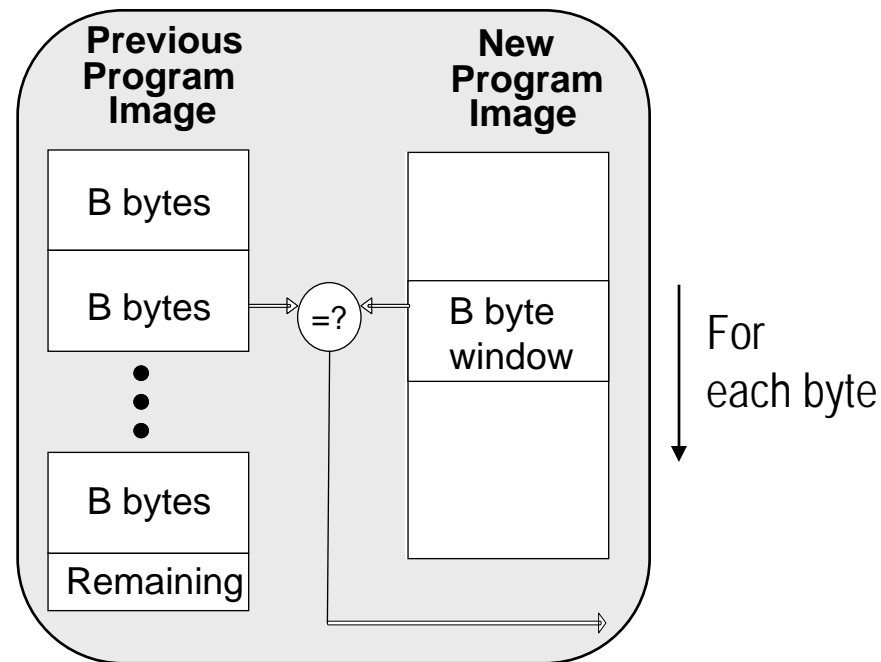
Step 1: Difference Generation (Encoding) - *First Approach: Fixed Block Comparison*

- Comparing at fixed sized blocks:
 - Doesn't work when code is shifted.



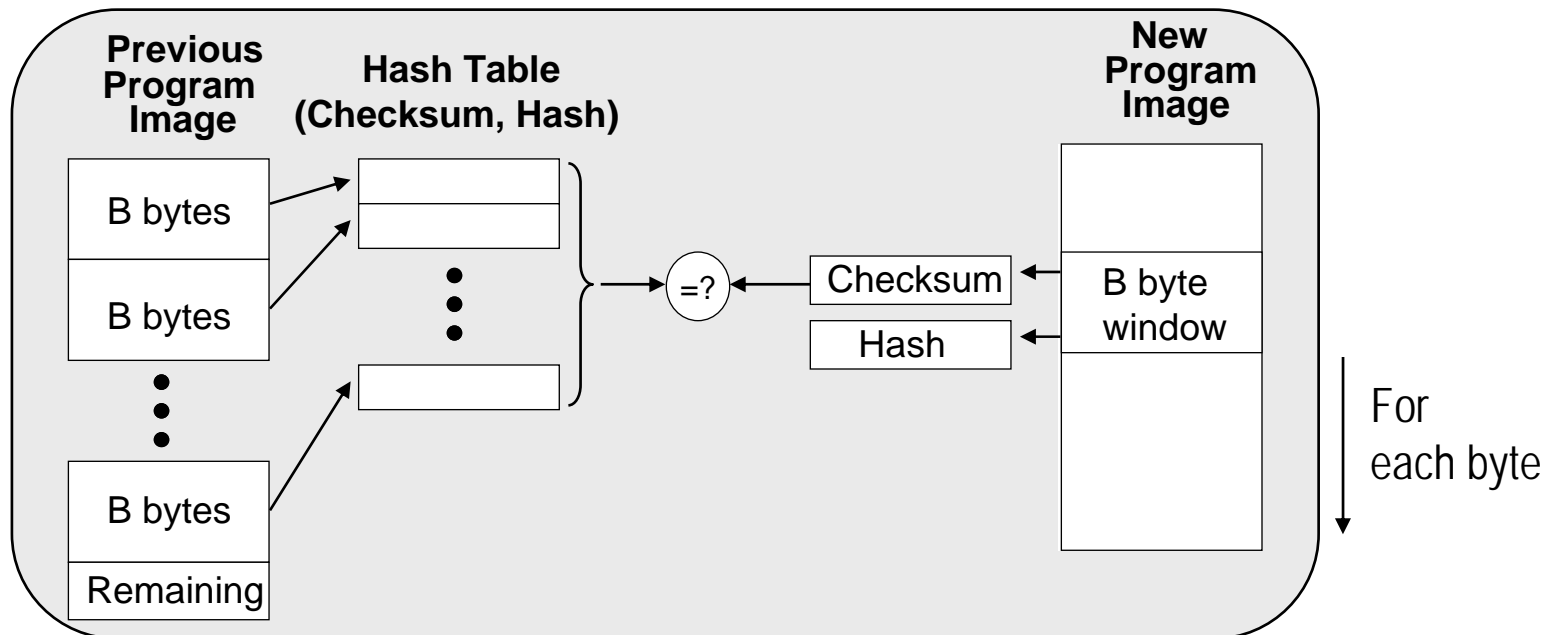
Step 1: Difference Generation (Encoding) - *First Approach: Fixed Block Comparison*

- Comparing at every byte:
 - Finds shared blocks with high cost.
- Need an efficient way of finding shared blocks.



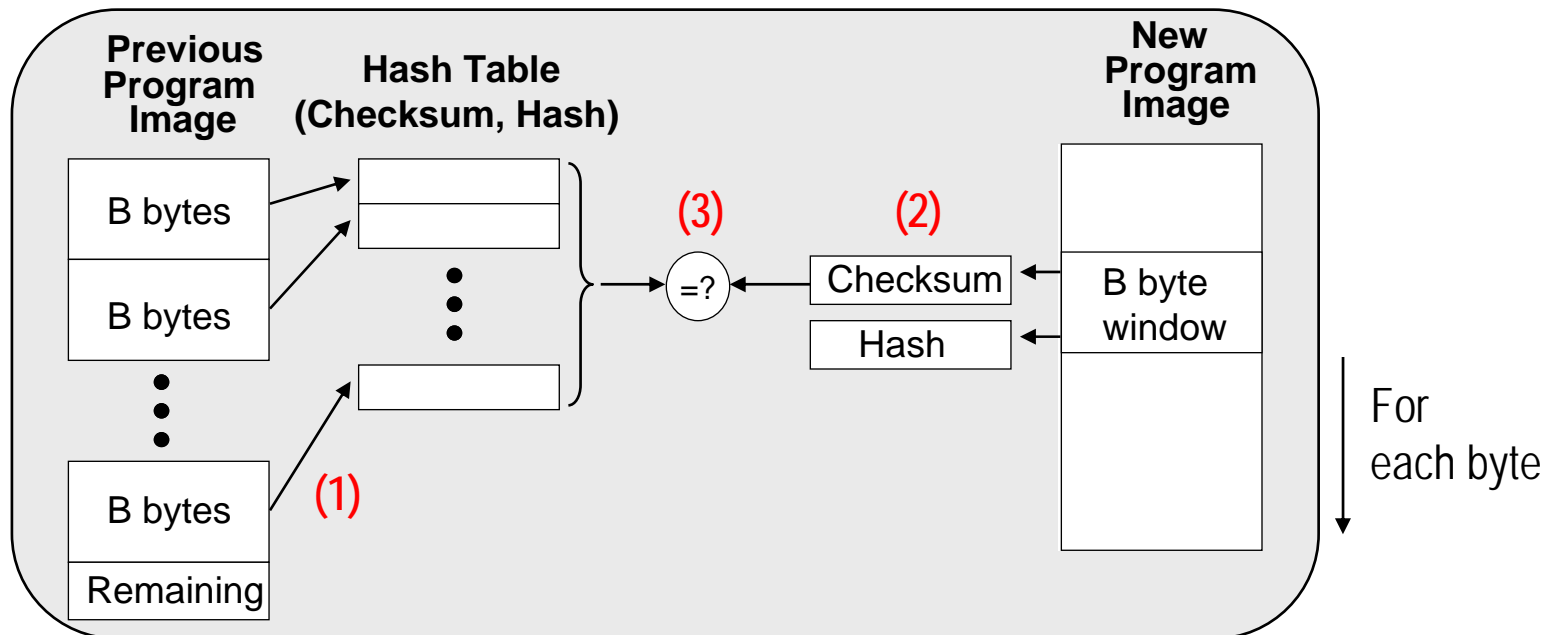
Step 1: Difference Generation - *How to Find Shared Blocks Efficiently?*

- Two level checksums (Idea of Rsync algorithm)
 - Finds a matching code block quickly with high accuracy.
 - Checksum (1st level): *Fast but not accurate (32-bit).*
 - Hash (2nd level): *Not fast but very accurate (128-bit).*



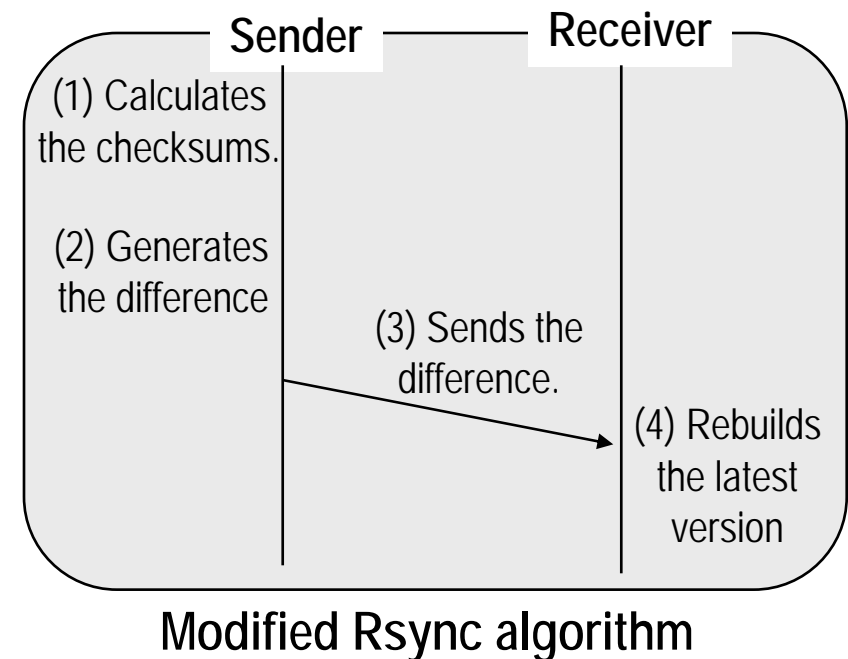
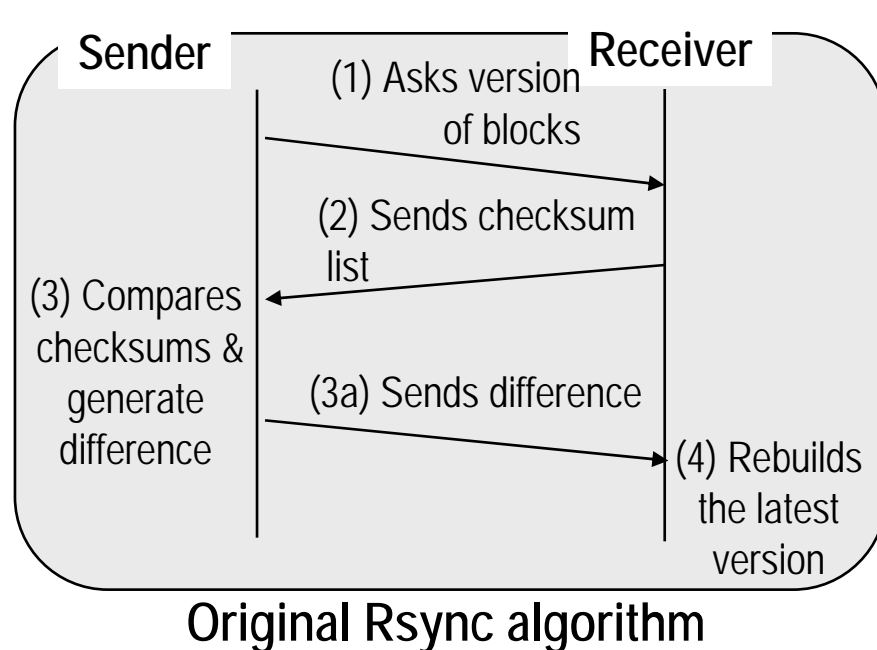
Step 1: Difference Generation - Using Rsync Algorithm

- (1) Build hash table for previous image.
- (2) For the window of new image, calculate checksum.
- (3) Lookup hash table.
 - For a matching checksum, calculate hash.
 - Otherwise, move to the next byte and repeat (2).



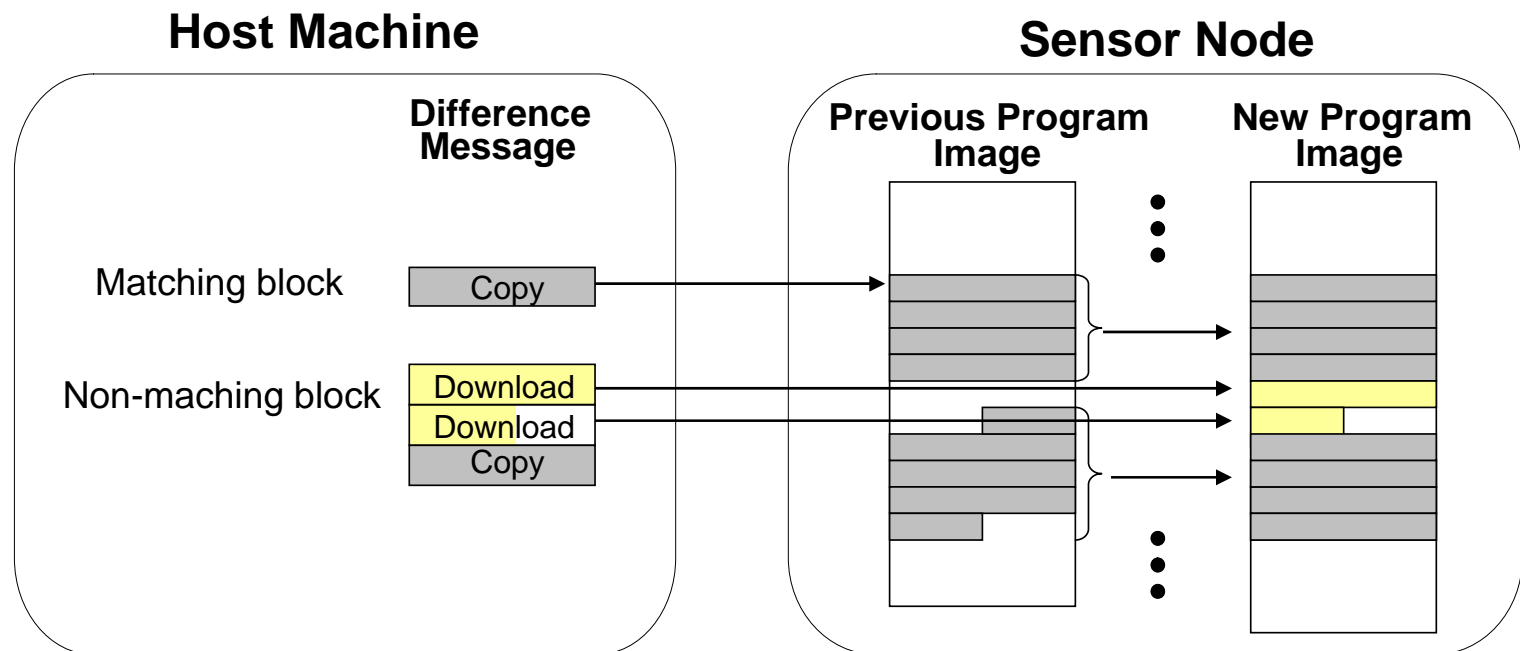
Step 2: Dissemination - *Modified Rsync Protocol for Wireless Sensors*

- Modified Rsync protocol for resource constrained sensor nodes.



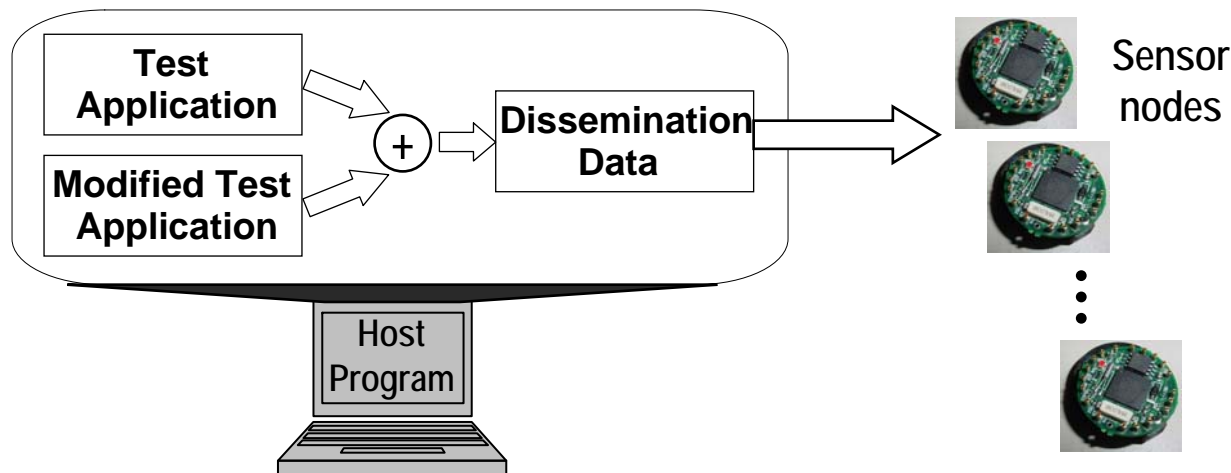
Step 3: Decoding - *Program Rebuild during Code Delivery*

- Host sends difference as a sequence of messages.
- Sensor rebuilds new image using the diff.
- Optimizing Flash Memory Access
 - Copy blocks are aligned to flash memory record boundary.



Experiment Setup

- Test Platforms: MICA2 / MICA2DOT
- Test Applications
 - Simple network programmable app: *XnpBlink* and *XnpCount*.
- Test Steps
 - Test app code and modified code are given to host program.
 - Compare the xmit time with that of XNP (non-incremental).



Experiment Setup – Test Cases

- Case 1: Changing a constant (*XnpBlink*)

```
command result_t StdControl.start() {  
    return call Timer.start(TIMER_REPEAT, 1000);  
}
```

- Case 2: Modifying implementation file (*XnpCount*)

```
event result_t Xnp.NPX_DOWNLOAD_DONE (uint16_t wProgramID,  
    uint8_t bRet, uint16_t wEENofP){  
    if (bRet == TRUE)  
        call CntControl.start();  
    else  
        call CntControl.stop();  
    return SUCCESS;  
}
```

- Case 3: Major change (*XnpBlink* → *XnpCount*)

Experiment Setup – Test Cases

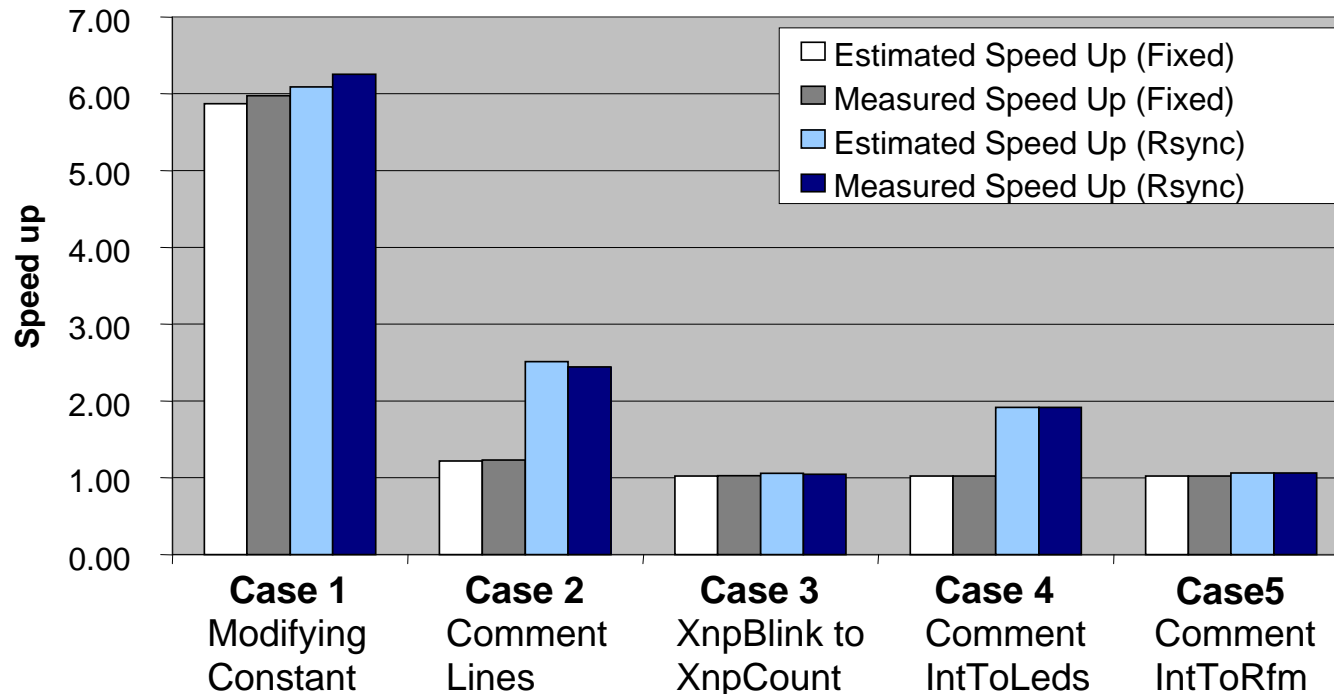
- Case 4: Modifying configuration file (*XnpCount*)
 - Commented out IntToLeds component.
- Case 5: Modifying configuration file (*XnpCount*)
 - Commented out IntToRfm component.

```
configuration XnpCount { }
implementation {
  components Main, Counter, /*IntToLeds,*/
    /*IntToRfm,*/ TimerC, XnpCountM, XnpC;

  // Main.StdControl -> IntToLeds.StdControl;
  // IntToLeds <- Counter.IntOutput;
  // Main.StdControl -> IntToRfm.StdControl;
  // Counter.IntOutput -> IntToRfm;
}
```

Results

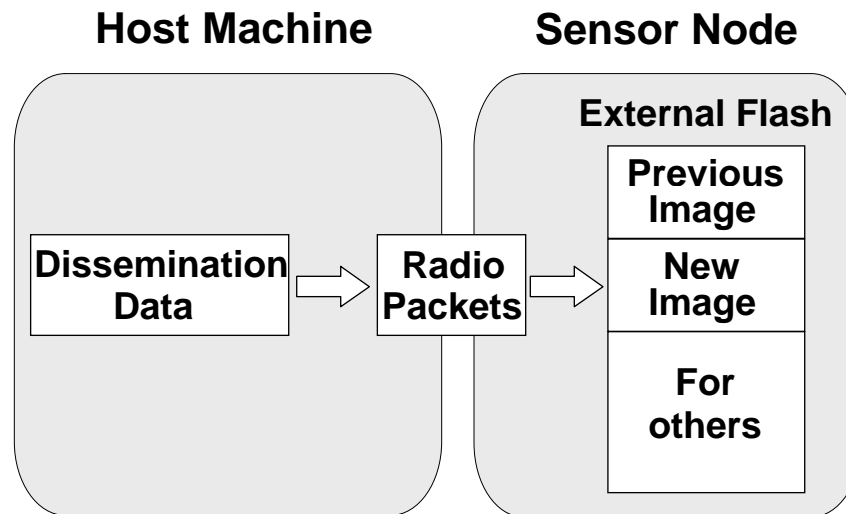
- Fixed Block Comparison: Almost no speedup except for case 1.
- Using Rsync algorithm:
 - Speed up of 2 to 2.5 for a small change (case 2 and 4).
 - Still limited speed up for big changes (case 3 and 5).



Problems of Rebuild during Delivery – *Difference Delivery Still Not Optimal*

Difference is decoded without being stored.

- (1) Size of copy msg is limited to bound running time.
- (2) Inefficient handling of missing copy msg.

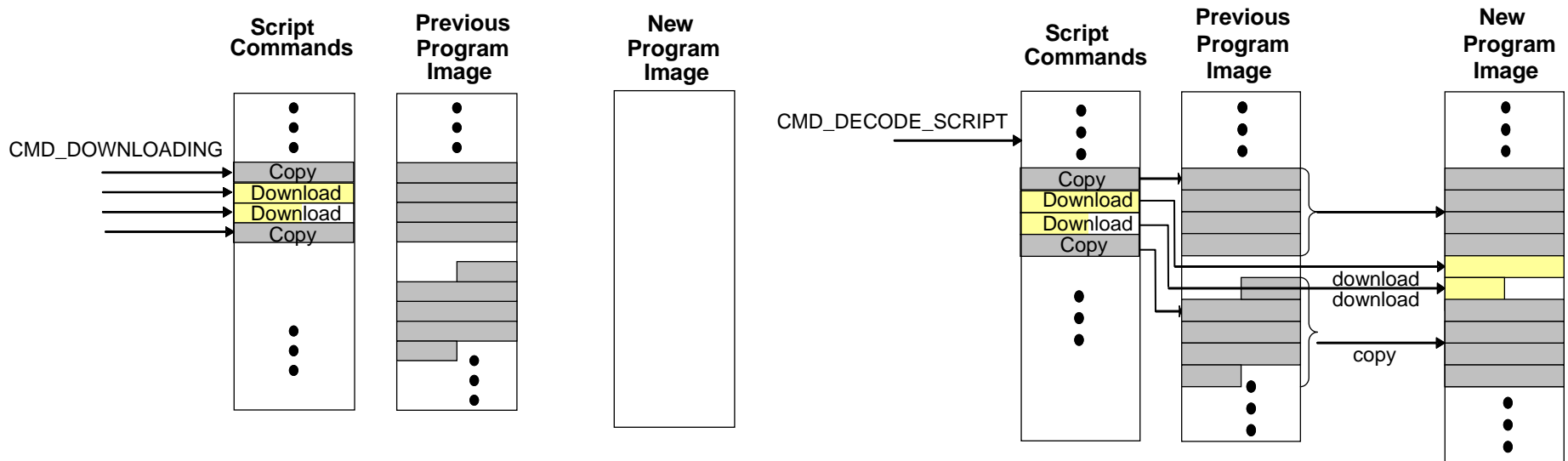


Optimizing Difference Delivery

Solution: Separate difference delivery and decoding.

(1) Stores the difference script in the first step

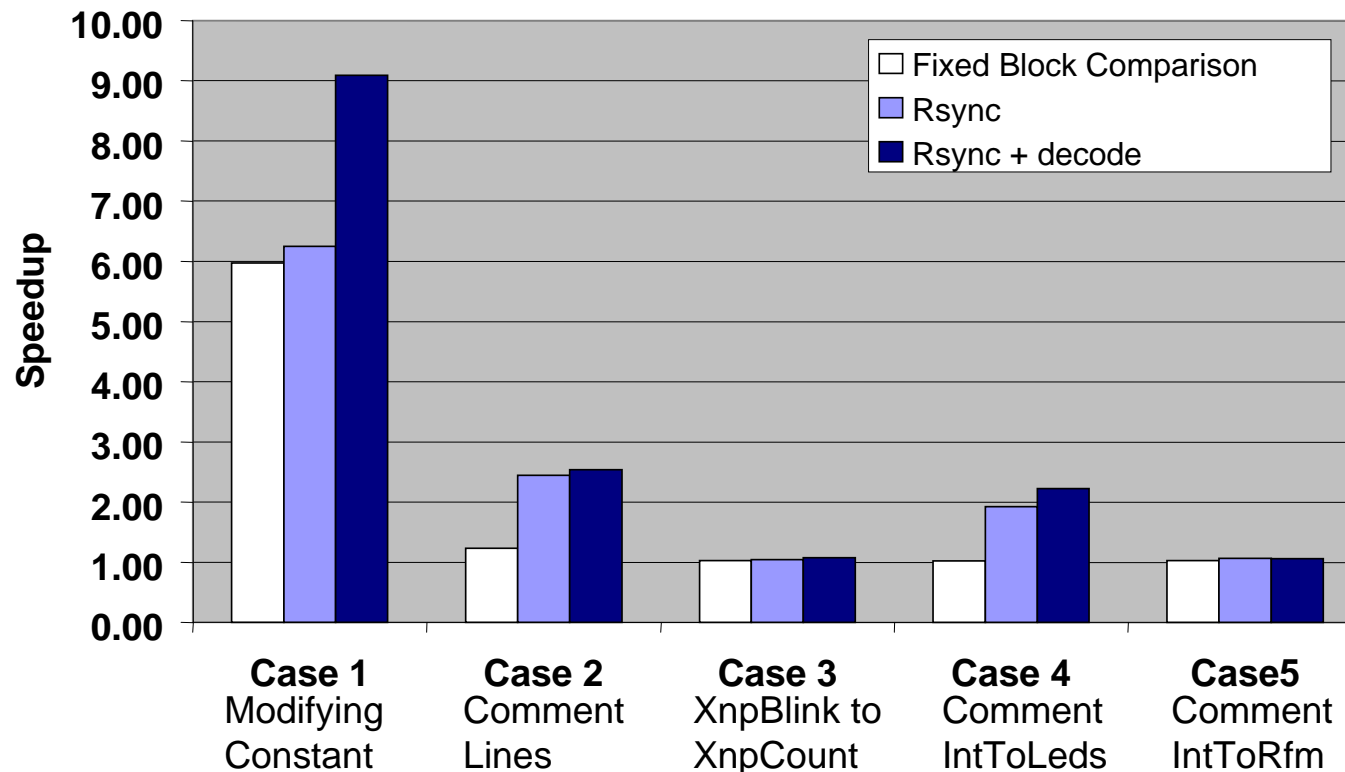
(2) Rebuilds the program after receiving decode command.



Results -

Using Rsync with separate decode

- The performance of using Rsync algorithm with separate decode command is similar to just using Rsync.
- But, the performance for changing a constant has improved (speed up of 9.1).



Conclusion

- Faster network reprogramming using incremental update.
- Platform independent solution using Rsync algorithm.
- Speed-up over non-incremental delivery
 - 2 – 2.5 for changing a few lines.
 - 9.1 for changing a constant.
- Future Work
 - Extension for multi-hop delivery.