Contiki Hands-on

Siena, Italy

July 2009

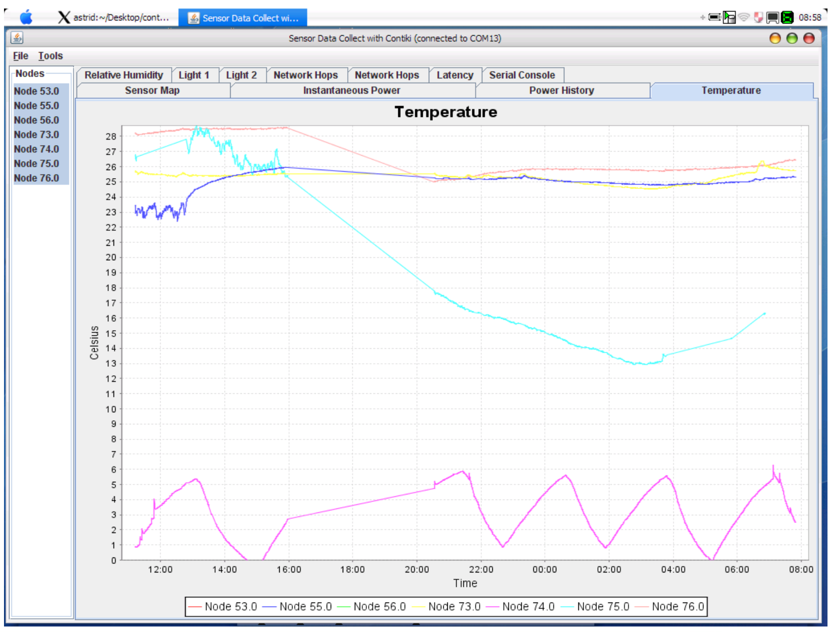
Fredrik Österlind

fros@sics.se

- Swedish Institute of Computer Science
  - Networked Embedded Systems group
- Fredrik Österlind
  - Contiki network simulator Cooja
  - Nightly Contiki tests

Purpose of today

- Contiki introduction
- Get started with Contiki
  - Programming
  - Hardware: Tmote Sky
  - Simulation in Cooja
- Hands-on experience



Contiki project

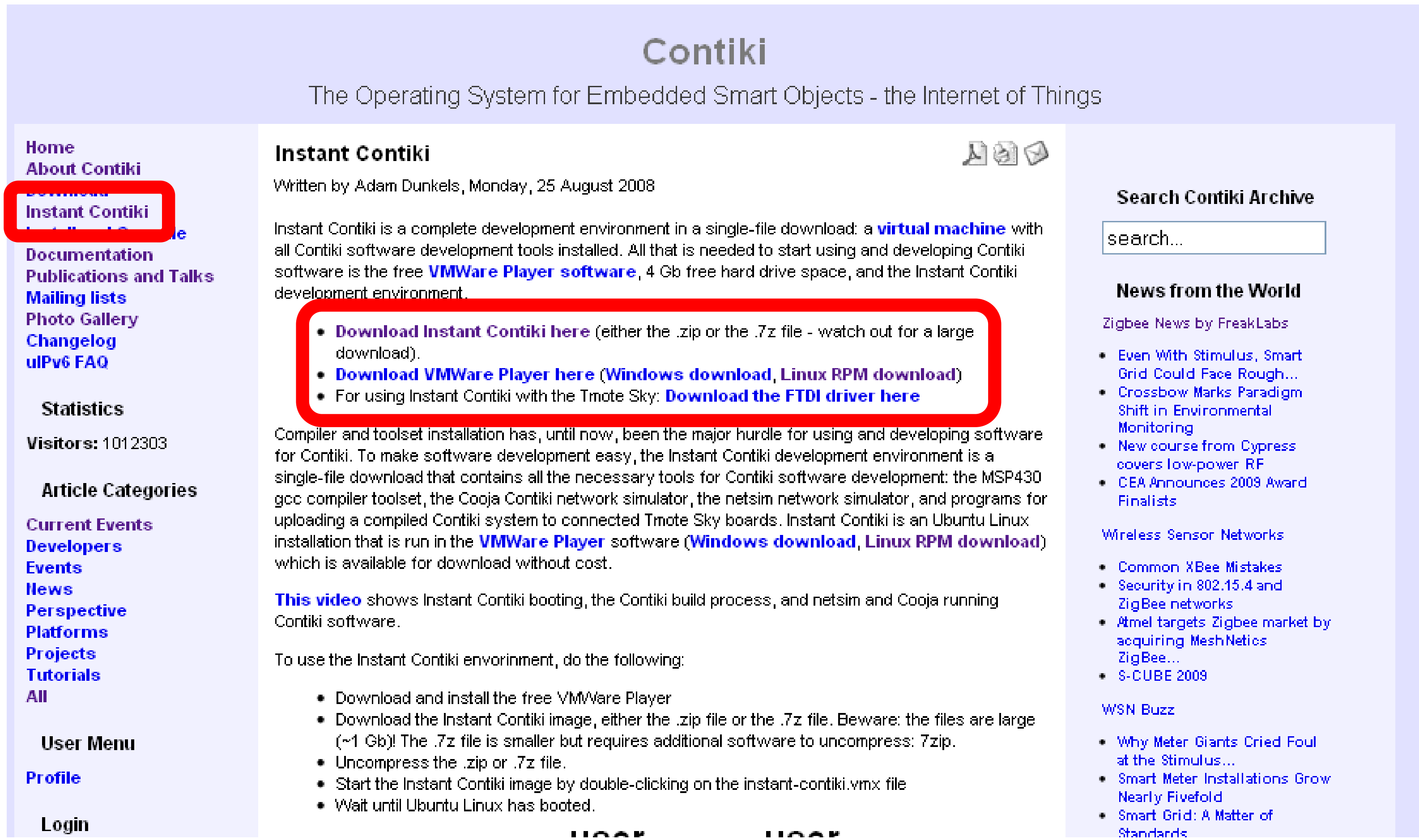
- Core group
  - Adam Dunkels (leader), Oliver Schmidt, Fredrik Österlind, Niclas Finne, Joakim Eriksson, Nicolas Tsiftes, Takahide Matsutsuka
- Developers
  - Zhitao He, Simon Barner, Simon Berg
  - 5+ incoming
- Contributors
  - Thiemo Voigt, Björn Grönvall, Tony Nordström, Matthias Bergvall, Groepaz, Ullrich von Bassewitz, Lawrence Chitty, Fabio Fumi. Matthias Domin, Christian Groessler, Anders Carlsson, Mikael Backlund, James Dessart, Chris Morse, …

Agenda

- 9:00-10:30 Contiki basics
  - Communication, processes, protothreads…
- Coffee break
- 11:00-12:30 Tmote Sky exercises
  - Contiki examples, shell application
  - **Participants need Instant Contiki**
    - Memory stick
- Lunch
- 13:30-15:00 Neighbor discovery exercise
  - Collaborative effort

Installing Contiki

http://www.sics.se/contiki/



Contiki

The Operating System for Embedded Smart Objects - the Internet of Things

- Home
- About Contiki
- Instant Contiki
- Documentation
- Publications and Talks
- Mailing lists
- Photo Gallery
- Changelog
- uIPv6 FAQ

Statistics

Visitors: 1012303

Article Categories

- Current Events
- Developers
- Events
- News
- Perspective
- Platforms
- Projects
- Tutorials
- All

User Menu

- Profile

Login

Instant Contiki

Written by Adam Dunkels, Monday, 25 August 2008

Instant Contiki is a complete development environment in a single-file download: a virtual machine with all Contiki software development tools installed. All that is needed to start using and developing Contiki software is the free VMWare Player software, 4 Gb free hard drive space, and the Instant Contiki development environment.

- Download Instant Contiki here (either the .zip or the .7z file - watch out for a large download).
- Download VMWare Player here (Windows download, Linux RPM download)
- For using Instant Contiki with the Tmote Sky: Download the FTDI driver here

Compiler and toolset installation has, until now, been the major hurdle for using and developing software for Contiki. To make software development easy, the Instant Contiki development environment is a single-file download that contains all the necessary tools for Contiki software development: the MSP430 gcc compiler toolset, the Cooja Contiki network simulator, the netsim network simulator, and programs for uploading a compiled Contiki system to connected Tmote Sky boards. Instant Contiki is an Ubuntu Linux installation that is run in the VMWare Player software (Windows download, Linux RPM download) which is available for download without cost.

This video shows Instant Contiki booting, the Contiki build process, and netsim and Cooja running Contiki software.

To use the Instant Contiki envorinment, do the following:

- Download and install the free VMWare Player
- Download the Instant Contiki image, either the .zip file or the .7z file. Beware: the files are large (~1 Gb)! The .7z file is smaller but requires additional software to uncompress: 7zip.
- Uncompress the .zip or .7z file.
- Start the Instant Contiki image by double-clicking on the instant-contiki.vmx file
- Wait until Ubuntu Linux has booted.

Search Contiki Archive

search...

News from the World

Zigbee News by FreakLabs

- Even With Stimulus, Smart Grid Could Face Rough...
- Crossbow Marks Paradigm Shift in Environmental Monitoring
- New course from Cypress covers low-power RF
- CEA Announces 2009 Award Finalists

Wireless Sensor Networks

- Common XBee Mistakes
- Security in 802.15.4 and ZigBee networks
- Atmel targets Zigbee market by acquiring MeshNetics ZigBee...
- S-CUBE 2009

WSN Buzz

- Why Meter Giants Cried Foul at the Stimulus...
- Smart Meter Installations Grow Nearly Fivefold
- Smart Grid: A Matter of Standards

What is Contiki and where does it come from?

Contiki

- Contiki – pioneering open source operating system for sensor networks
  - IP networking
  - Hybrid threading model, protothreads
  - Dynamic loading
  - Power profiling
  - Network shell
- Small memory footprint
- Designed for portability
  - 14 platforms, 5 CPUs in current CVS code

Contiki as a tool

- For building systems
  - Programming abstractions
  - Shell
  - Power-saving mechanisms
- For writing papers
  - Power profiling

Contiki as knowledge transfer

- Making research useful
  - Transfer research results to useable C code
  - Promote simplicity and clarity over excessive complexity
  - Example: protothreads
- Put research into perspective
  - Transfer knowledge from practice to research
  - Example: IP for sensor networks

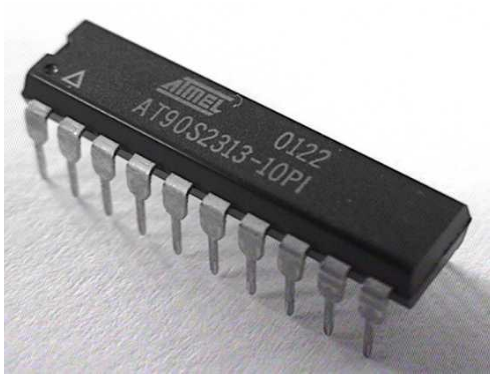
Contiki features

- Contiki: loadable modules [Emnets 2004]
  - SOS: loadable modules [MobiSys 2005]
- Contiki: preemptive threads on top of events [Emnets 2004]
  - TOSThreads: preemptive threads on top of events (TinyOS 2.1.0, 2008)
- Contiki: IP in sensor networks [EWSN 2004]
  - IETF 6lowpan: IP over 802.15.4 (2006)
  - IP for Smart Objects Alliance: IP in sensor networks (2008)
- Contiki: Software-based energy estimation (2007)
  - Quanto (OSDI 2008)

- Network shell
  - Makes interaction easier
- Rime stack
  - Makes network programming easier

Contiki target systems

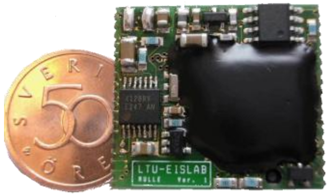
- Small embedded processors with networking
  - – Sensor networks, smart objects, ...
- 98% of all microprocessors go into embedded systems
  - – 50% of all processors are 8-bit
- MSP430, AVR, ARM7, 6502,



SICS

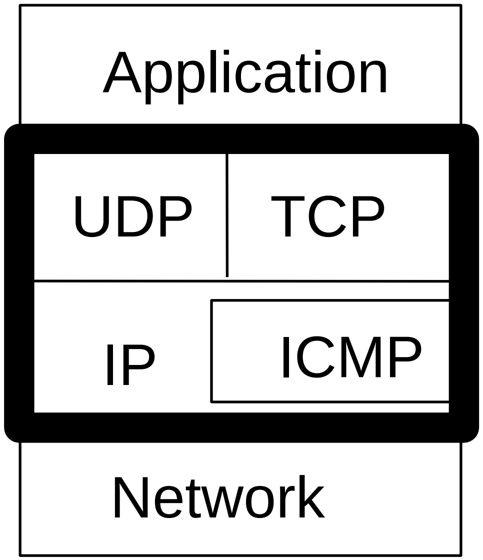
Background: The Arena Project (2000) LTU, Telia, Ericsson, SICS

- Hockey players with wireless sensors
  - Bluetooth sensors, camera on helmet
- Spectators with direct access to sensor readings
- TCP/IP used on the Bluetooth-equipped sensors
  - lwIP stack



- Luleå Hockey lost with 1-4...

Background: uIP – the world's smallest TCP/IP stack (2001)



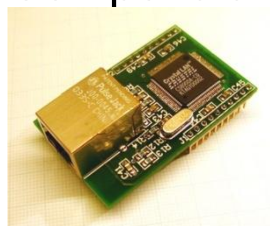
- uIP – micro IP
- Open source
- ~5k code, ~2k RAM
  - Smallest configuration ~3k code, ~128 bytes RAM
- RFC compliant
  - Order of magnitude smaller than previous stacks
- Bottom-up design
  - Single-packet buffer
  - Event-driven API

lwIP and uIP today

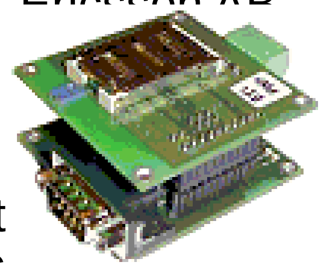
- Very well-known in the embedded community
- Used in products from 100+ companies
- Covered in several books on embedded networking
- Porting uIP in professional magazines
- Recommended by leading professionals
- Competence specifically required in job postings

ABB
Actern
All Set
Altera
Amirix
Analog
Aurion
Axon D
BMW
Campb

Cisco Systems
Compendium Technologies
ratio
olo



Engenium Tech
Ericsson AB



GE Security
(
I
H
Inc
ix

Odense Unive
Denmark

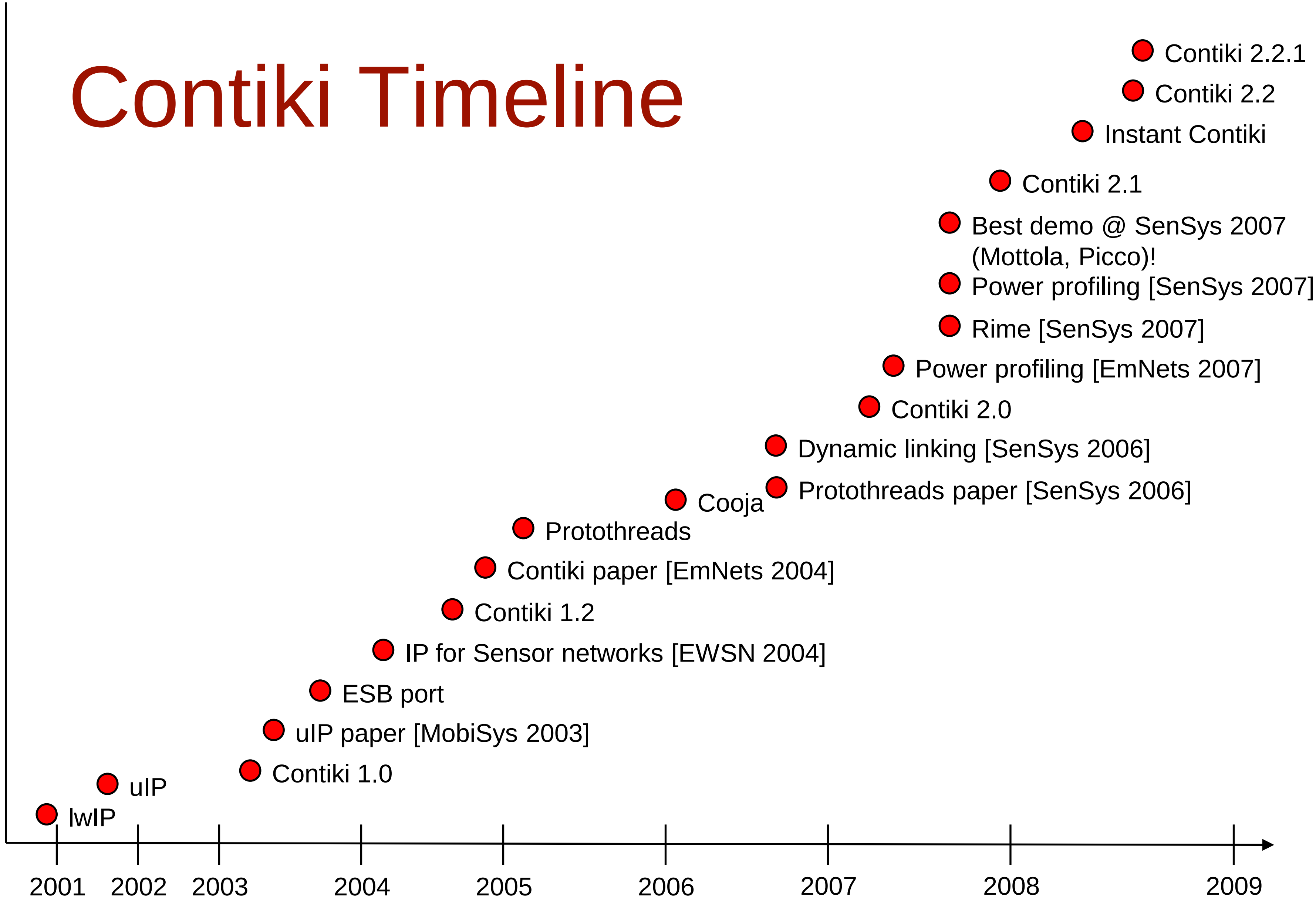
ng,
rs
La
iv
f A
f C
f C
SA

NASA
NBS Tech

gy
Rowley Associates Ltd.

Shift
SYS
Tar
Vol
Wir
Xili

Contiki Timeline

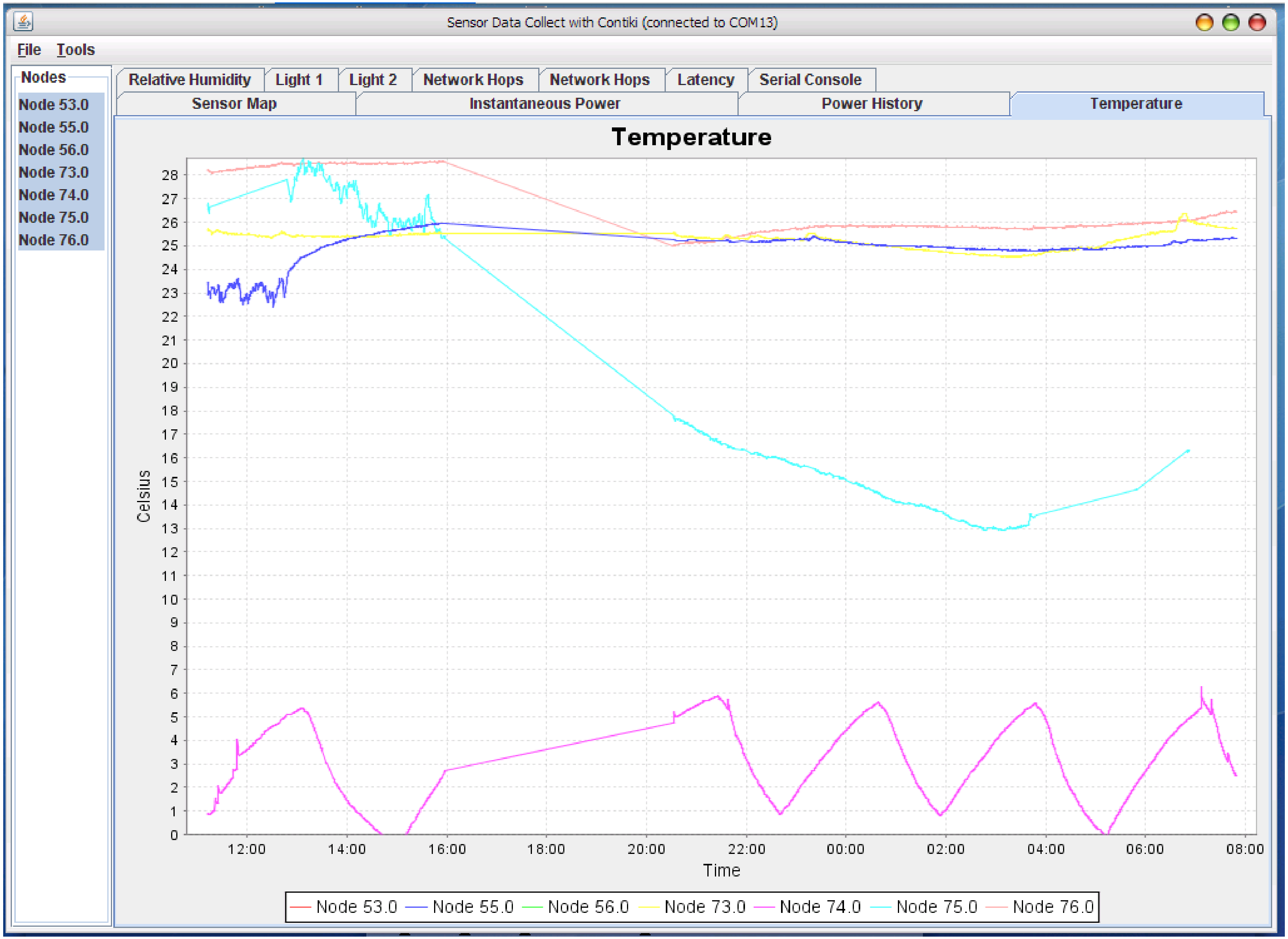


The name Contiki

- The Kon-Tiki raft: sailed across the Pacific Ocean with minimal resources

Demo:

Contiki collect + shell


Sensor Data Collect with Contiki (connected to COM13)
File Tools
Nodes
Node 53.0
Node 55.0
Node 56.0
Node 73.0
Node 74.0
Node 75.0
Node 76.0
Relative Humidity
Light 1
Light 2
Network Hops
Network Hops
Latency
Serial Console
Sensor Map
Instantaneous Power
Power History
Temperature
Temperature
Celsius
Time
12:00
14:00
16:00
18:00
20:00
22:00
00:00
02:00
04:00
06:00
08:00
Node 53.0 Node 55.0 Node 56.0 Node 73.0 Node 74.0 Node 75.0 Node 76.0

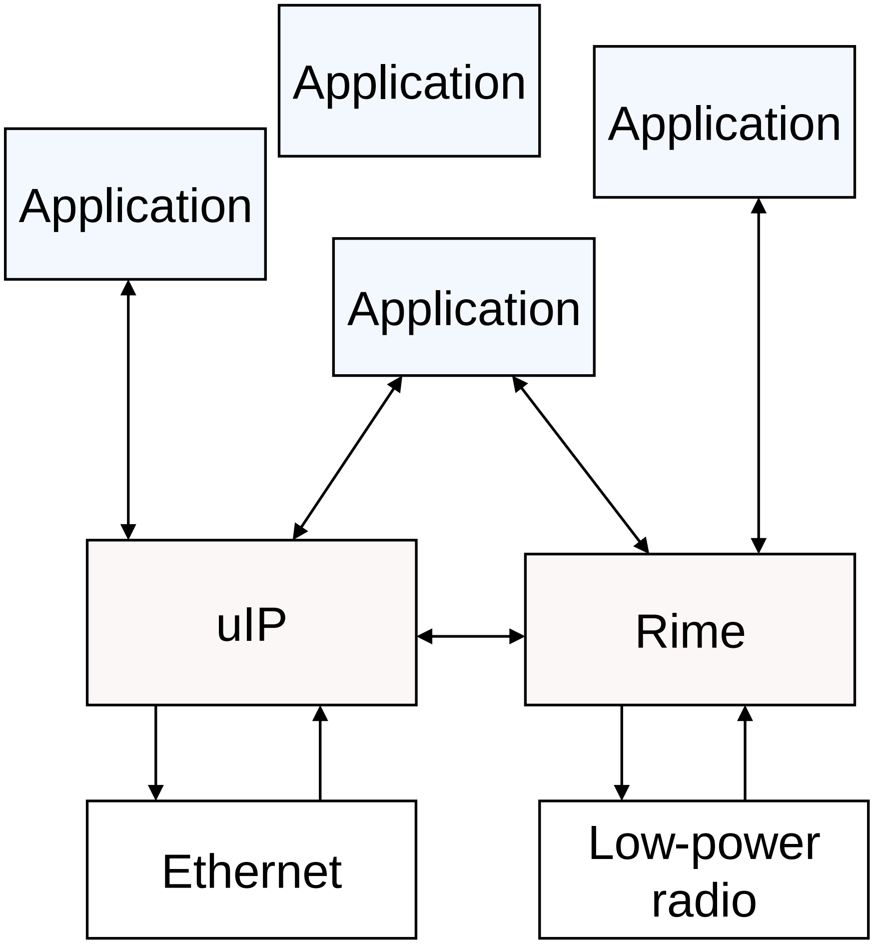
The shell

- Command-line interaction with a network of Contiki nodes
- Unix-style pipelines
- File system interaction: ls, write, read
- Repetition: repeat, randwait
- Network commands: netcmd
- **sense | senseconv**
- **sense | write file | send**
- **repeat 0 20 { randwait 20 { sense | blink | send } } &**
- **collect | binprint &**

Communication in Contiki

Contiki: two communication stacks

- Two communication stacks in Contiki
  - uIP – TCP/IP
  - Rime – low overhead
- Applications can use either or both
  - Or none
- uIP can run over Rime
- Rime can run over uIP



uIP

- Processes open TCP or UDP connections
  - tcp_connect(), tcp_listen(), udp_new()
- **tcpip_event** posted when new connection arrives, new data arrives, connection is closed, etc.
- Reply packet is sent when process returns
- TCP connections periodically polled for data
- UDP packets sent with uip_udp_packet_send()

uIP APIs

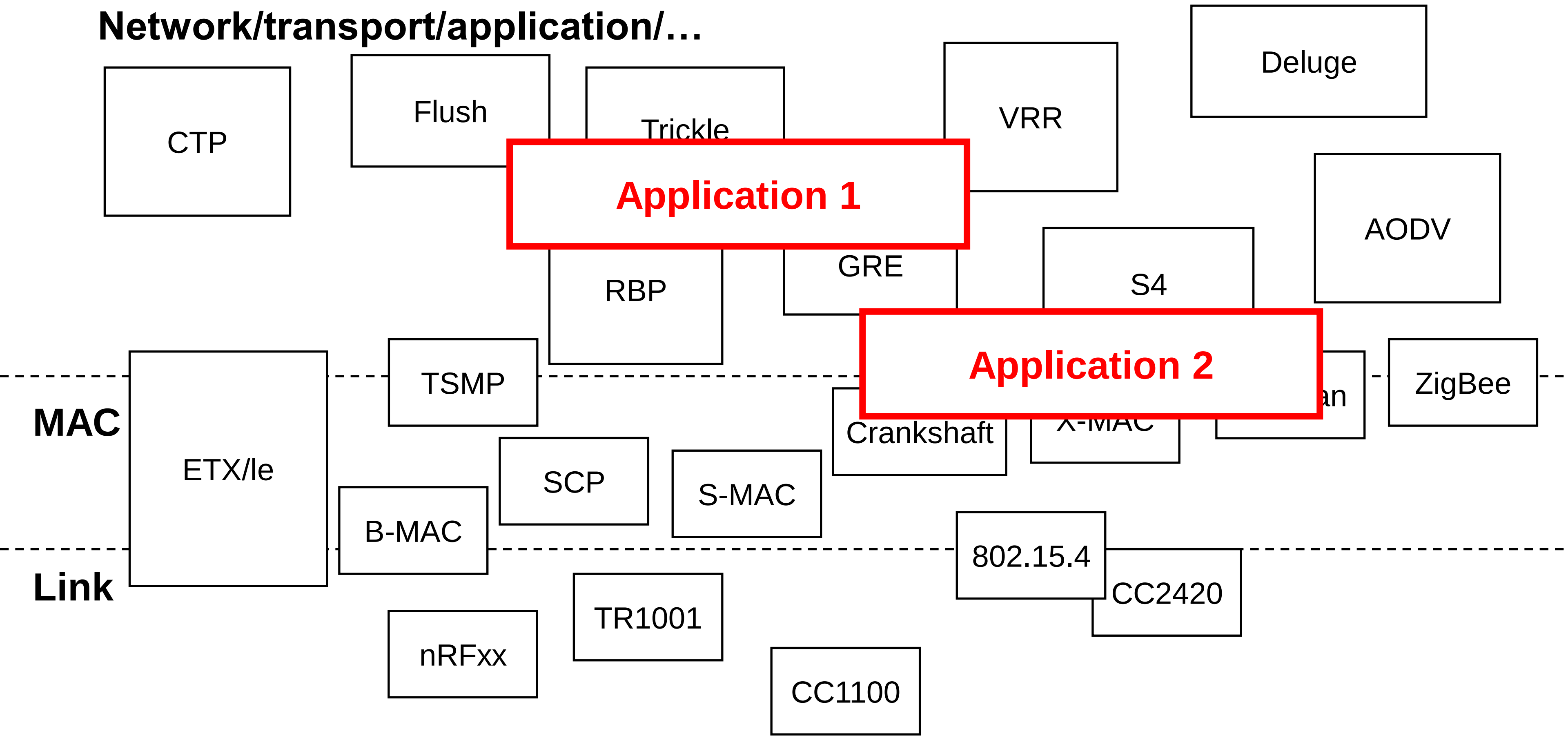
- Two APIs
  - The “raw” uIP event-driven API
  - Protosockets – sockets-like programming based on protothreads
- Event-driven API works well for small programs
  - Explicit state machines
- Protosockets work better for larger programs
  - Sequential code

Protosockets: example

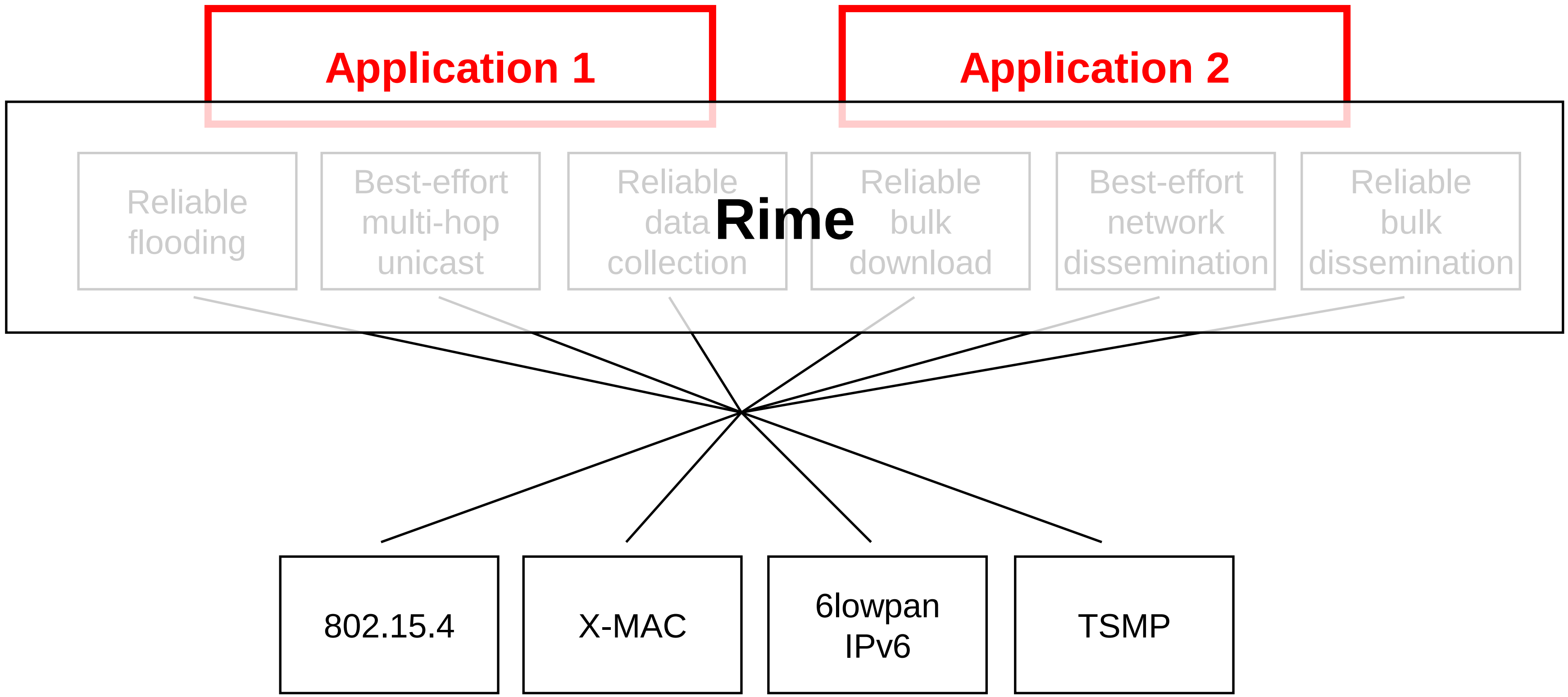
```
int smtp_protothread(struct psock *s)
{
  PSOCK_BEGIN(s);

  PSOCK_READTO(s, '\n');

  if(strncmp(inputbuffer, “220”, 3) != 0) {
    PSOCK_CLOSE(s);
    PSOCK_EXIT(s);
  }

  PSOCK_SEND(s, “HELO ”, 5);
  PSOCK_SEND(s, hostname, strlen(hostname));
  PSOCK_SEND(s, “\r\n”, 2);

  PSOCK_READTO(s, '\n');

  if(inputbuffer[0] != '2') {
    PSOCK_CLOSE(s);
    PSOCK_EXIT(s);
  }
…
…
```

Rime

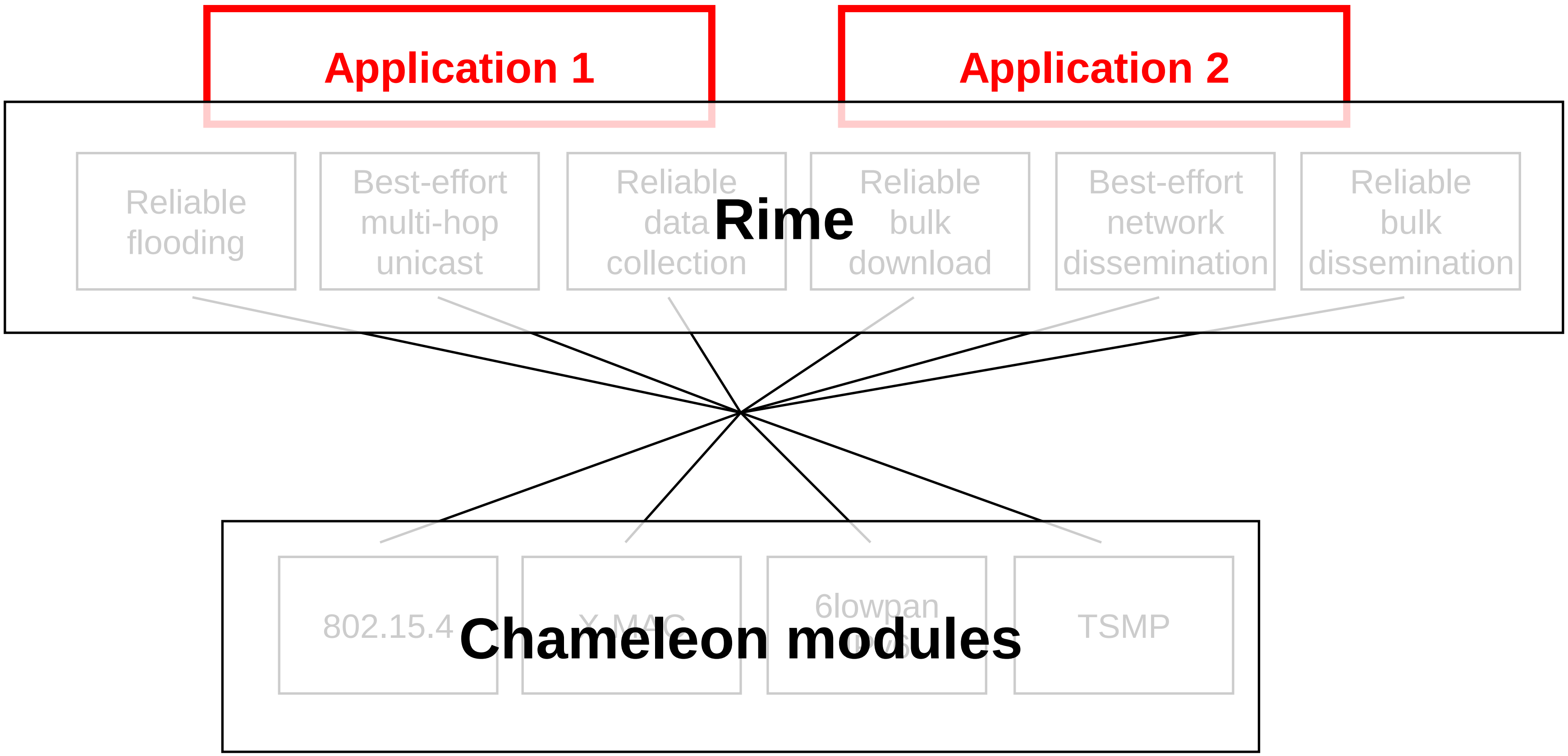
Sensor communication programming (before Contiki/Rime)



Rime: “sockets” for sensor networks

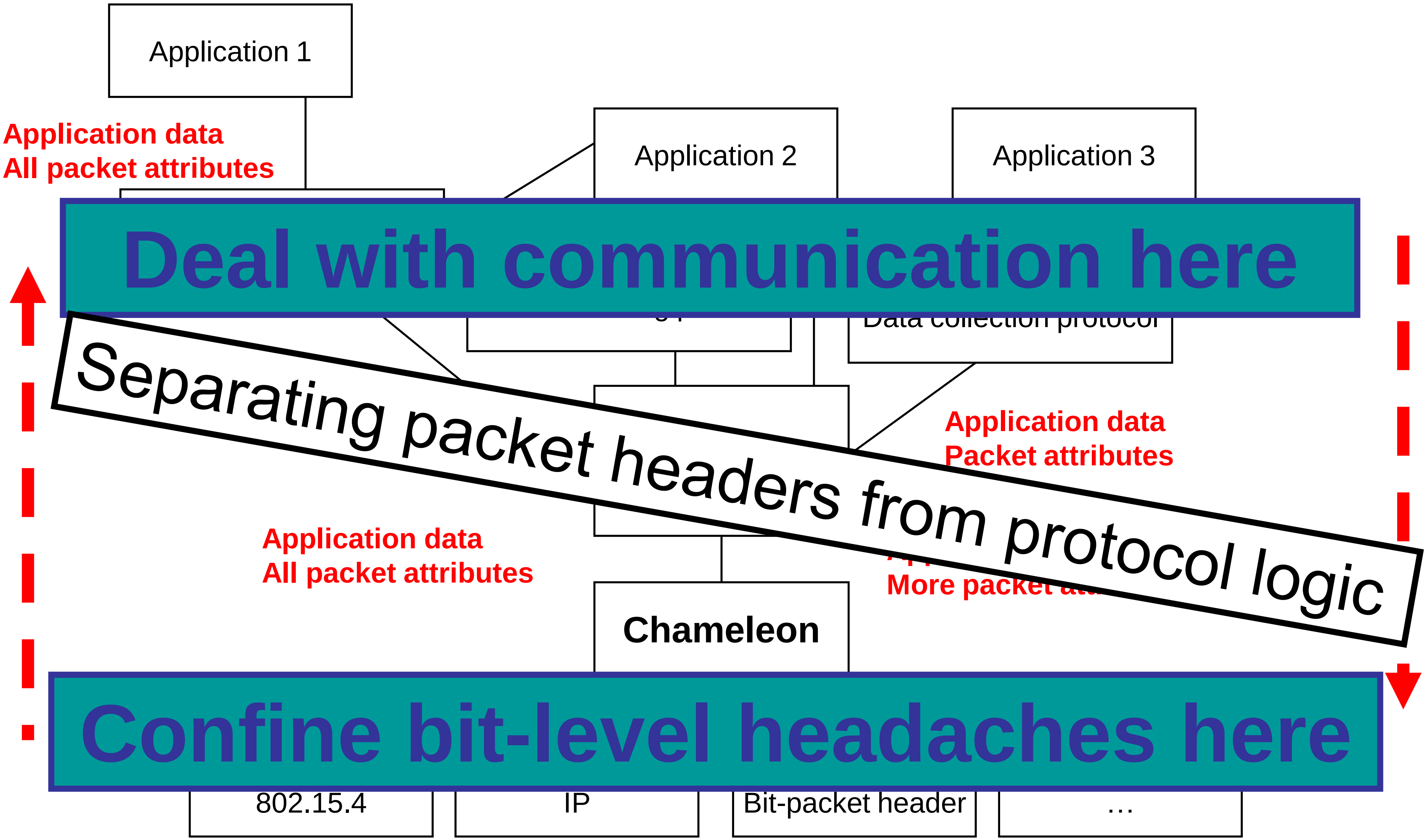


Chameleon: adapting to underlying MAC layers, link layers, protocols



Chameleon / Rime

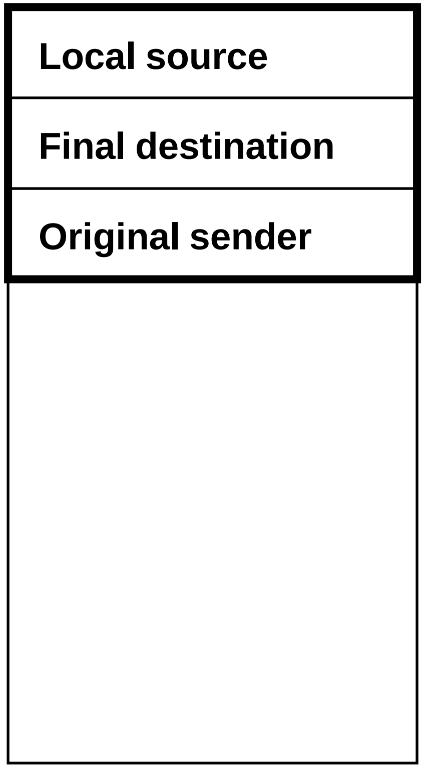
- Separating packet headers from protocol logic
- Rime: a set of communication primitives
  - Lightweight layering: primitives built in terms of each other
  - Compose simple abstractions to more complex ones
- Chameleon modules
  - Header construction/parsing done separate from communication stack



Application 1
Application 2
Application 3
Application data
All packet attributes
Deal with communication here
Separating packet headers from protocol logic
Application data
Packet attributes
Application data
All packet attributes
Chameleon
Confine bit-level headaches here
802.15.4
IP
Bit-packet header
…

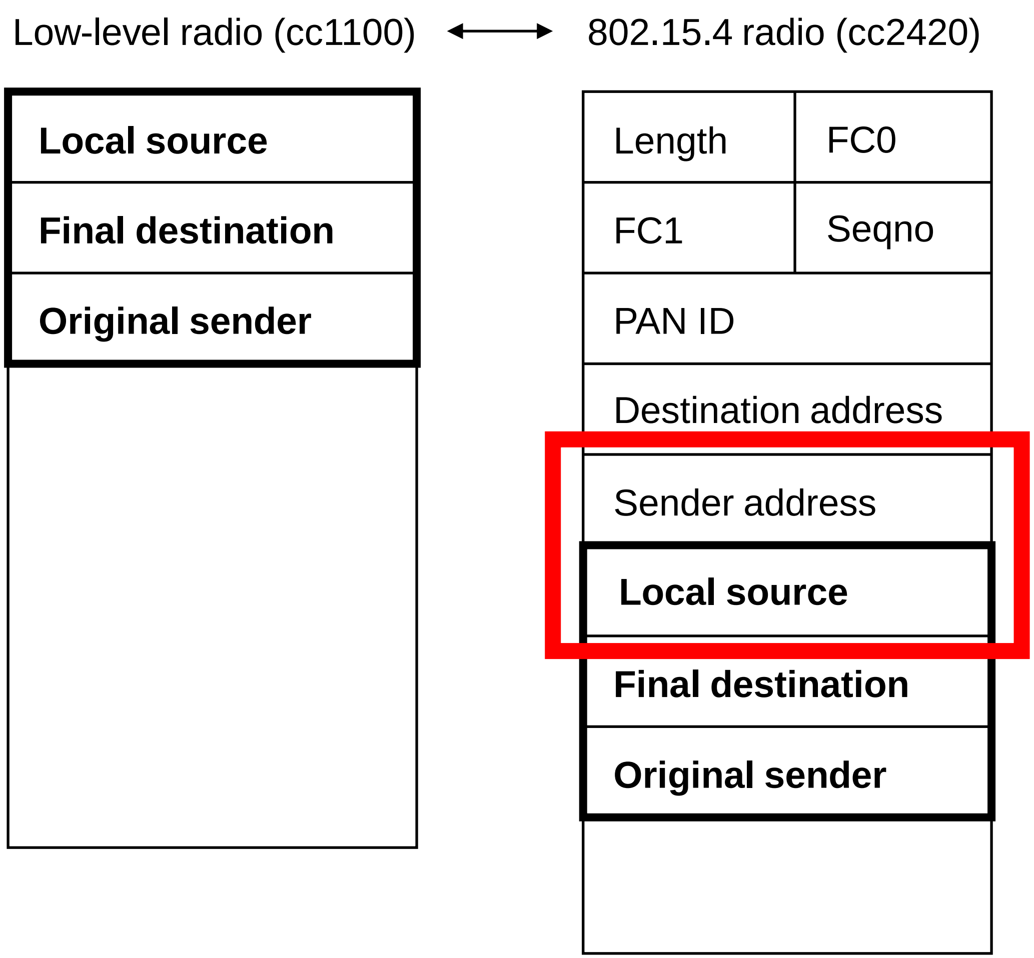
A Communication Protocol with Different Radios

Low-level radio (cc1100)

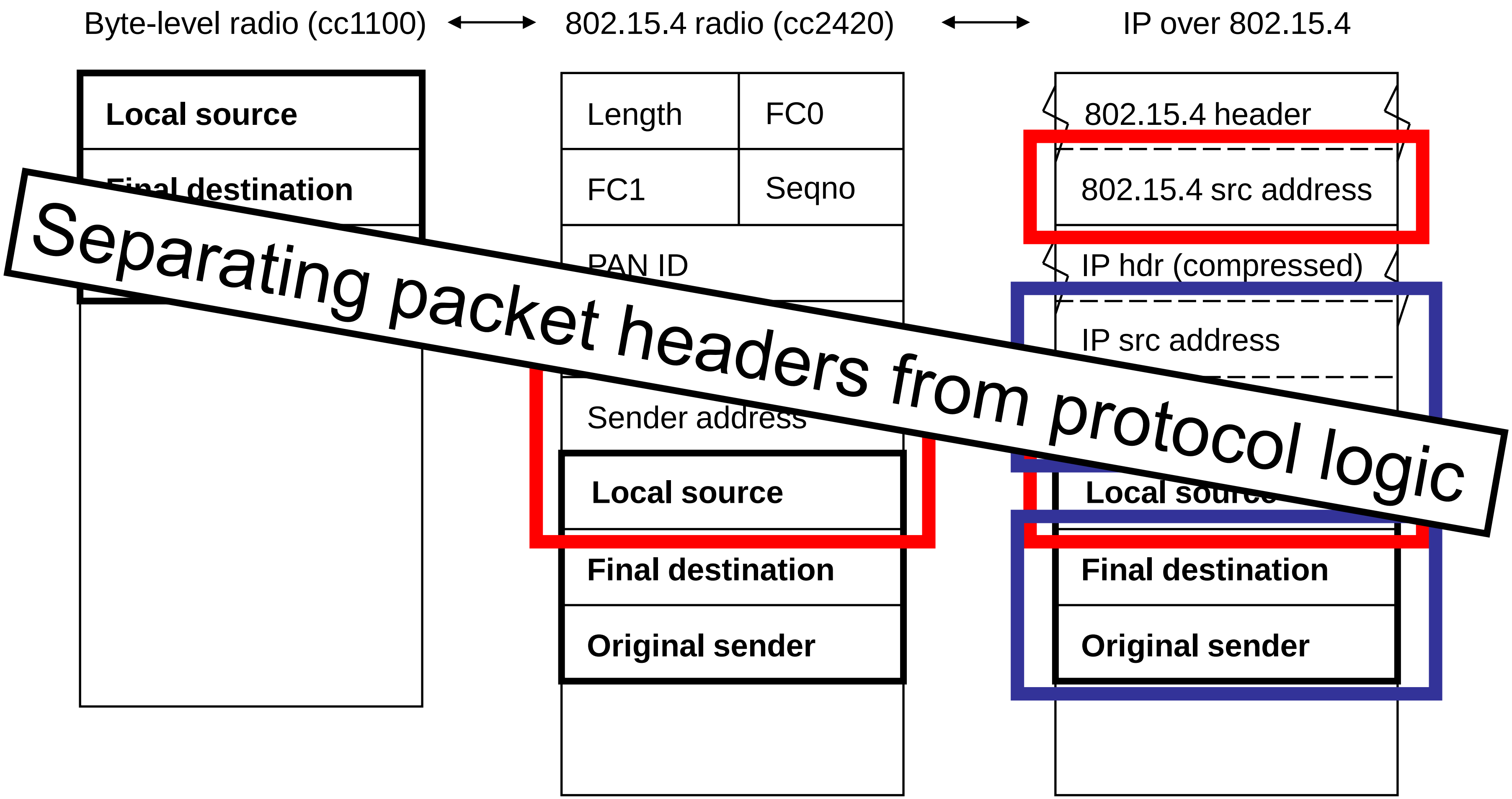


- No extra headers added by radio

Header Fields hold the Same Information



Chameleon: Efficient Header Compression



Rime – a lightweight, layered communications stack

- A set of communication abstractions (in increasing complexity):
  - Single-hop broadcast (broadcast)
  - Single-hop unicast (unicast)
  - Reliable single-hop unicast (runicast)
  - Best-effort multi-hop unicast (multihop)
  - Hop-by-hop reliable multi-hop unicast (rmh)
  - Best-effort multi-hop flooding (netflood)
  - Reliable multi-hop flooding (trickle)

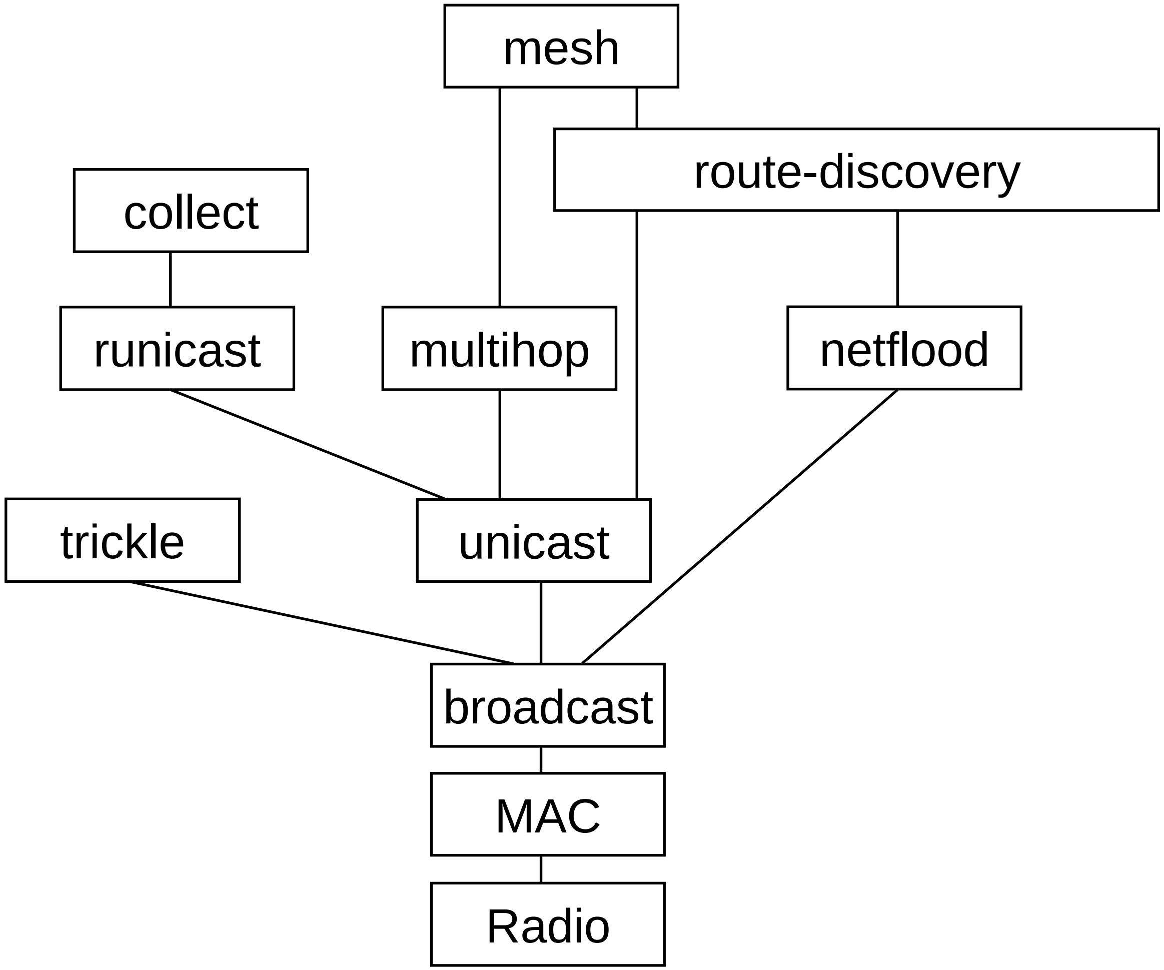
Rime – a lightweight, layered communications stack

- A set of communication abstractions (continued)
  - Hop-by-hop reliable data collection tree routing (collect)
  - Hop-by-hop reliable mesh routing (mesh)
  - Best-effort route discovery (route-disovery)
  - Single-hop reliable bulk transfer (rudolph0)
  - Multi-hop reliable bulk transfer (rudolph1)

Rime – layers reduce complexity

- Each module is fairly simple
  - Compiled code between 114 and 598 bytes
- Complexity handled through layering
  - Modules are implemented in terms of each other
- Not a fully modular framework
  - Full modularity typically gets very complex
  - Instead, Rime uses strict layering

Rime map (partial)



Rime – the name

- **Rime frost** composed of many thin layers of ice
- **Syllable rime** – last part of a syllable
  - Communication formed by putting many together

Rime channels

- All communication in Rime is identified by a 16-bit channel
- Communicating nodes must agree on what modules to use on a certain channel
  - Example
    - unicast <-> unicast on channel 155
    - netflood <-> netflood on channel 130
- Channel numbers < 128 are reserved by the system
  - Used by the shell, other system apps

Rime programming model

- Callbacks
  - Called when packet arrives, times out, error condition, …
- Connections must be opened before use
  - Arguments: module structure, channel, callbacks

Rime example: send message to all neighbors

```
void recv(struct broadcast_conn *c) {   /* Called when a */
  printf("Message received\n");          /* message is    */
}                                        /* received.     */
```

Receive data

```
struct broadcast_callbacks cb = {recv};/* Callback */
struct broadcast_conn c;               /* Connection */
```

```
void setup_sending_a_message_to_all_neighbors(void) {
  broadcast_open(&c, 128, &cb);        /* Channel 128 */
}
```

Open connection

```
void send_message_to_neighbors(char *msg, int len) {
  packetbuf_copyfrom(msg, len);        /* Setup packetbuf */
  broadcast_send(&c);                  /* Send message  */
}
```

Send data

Rime example: send message to entire network

```
void recv(struct trickle_conn *c) {  /* Called when a*/
  printf("Message received\n");       /* message is    */
}                                     /* received.     */

struct trickle_callbacks cb = {recv}; /* Callbacks */
struct trickle_conn c;                /* Connection    */

void setup_sending_a_message_to_network(void) {
  trickle_open(&c, CLOCK_SECOND, 129, &cb);   /* Channel 129 */
}

void send_message_to_network(char *msg, int len) {
  packetbuf_copyfrom(msg, len);     /* Setup packetbuf */
  trickle_send(&c);                 /* Send message */
}
```

Rime example: send message to node somewhere in the network

```
void recv(struct mesh_conn *c, rimeaddr_t *from) {
  printf("Message received\n");
}

struct mesh_callbacks cb = {recv, NULL, NULL};
struct mesh_conn c;

void setup_sending_a_message_to_node(void) {
  mesh_open(&c, 130, &cb);
}

void send_message_to_node(rimeaddr_t *node, char *msg, int len) {
  packetbuf_copyfrom(msg, len);
  mesh_send(&c, node);
}
```

Structure of the Rime stack

Application	apps/shell, …
Rime + Chameleon	
Power-saving MAC	core/net/mac/
Radio	CC2420

Summary so far

- History
- Communication: uIP and Rime
  - uIP: TCP/IP
  - Rime: low-power, routing
  - Can run on top of each other

Programming in Contiki

Overview

- Hello, world!
- Contiki processes and protothreads
- Timers in Contiki
- The Contiki build system
- Developing software with Contiki
  - Native port
- Directory structure
- Coding and naming standards

Hello, world!

```
/* Declare the process */
PROCESS(hello_world_process, “Hello world”);

/* Make the process start when the module is loaded */
AUTOSTART_PROCESSES(&hello_world_process);

/* Define the process code */
PROCESS_THREAD(hello_world_process, ev, data) {
  PROCESS_BEGIN();          /* Must always come first */
  printf(“Hello, world!\n”);  /* Initialization code goes here */
  while(1) { /* Loop for ever */
    PROCESS_WAIT_EVENT(); /* Wait for something to happen */
  }
  PROCESS_END();            /* Must always come last */
}
```

Contiki processes and protothreads

Contiki processes

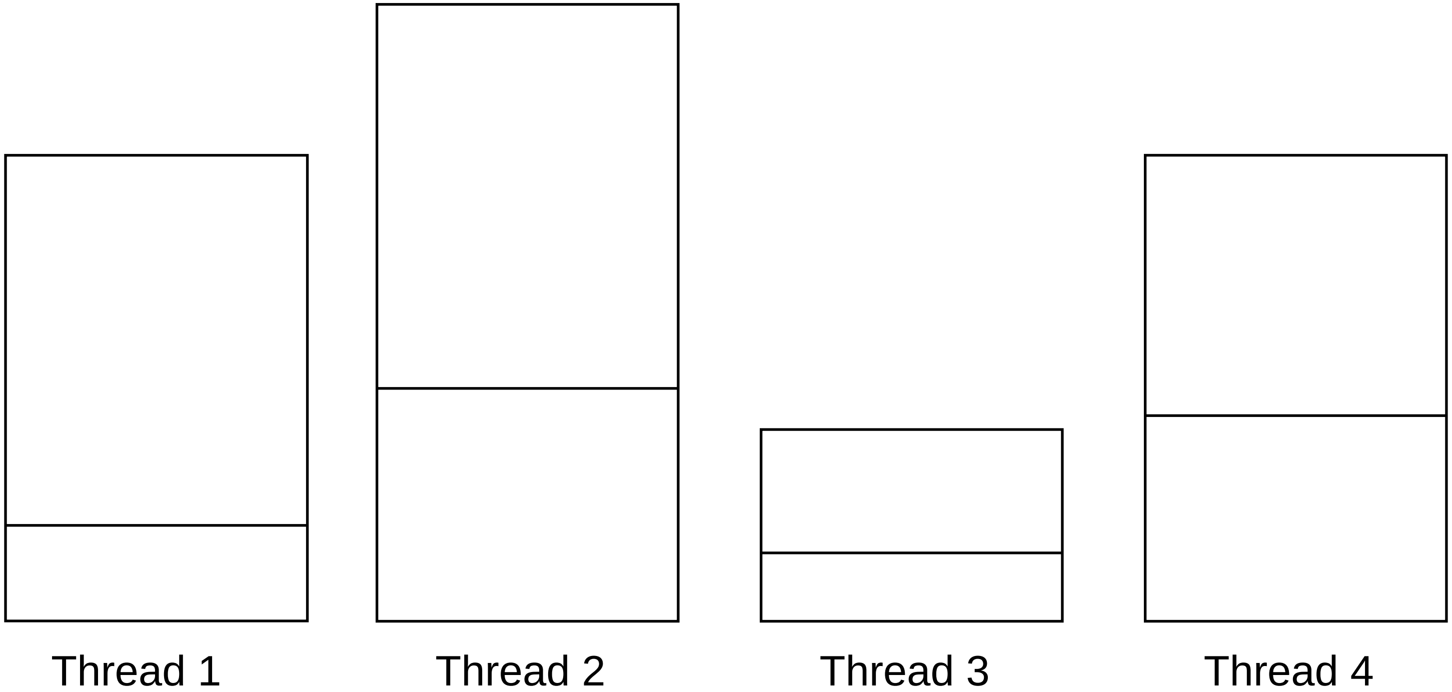
- The Contiki kernel is event-based
  - Invokes processes whenever something happens
    - Sensor events, processes starting, exiting
  - Process invocations must not block
- Protothreads provide sequential flow of control in Contiki processes

The event-driven Contiki kernel

- Event-driven vs multithreaded
  - Event-driven requires less memory
  - Multithreading requires per-thread stacks

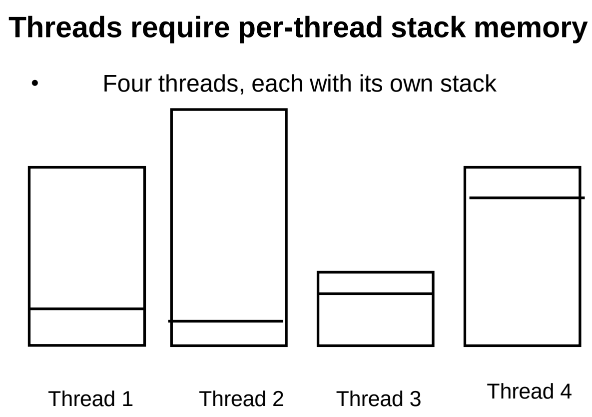
Threads require per-thread stack memory

- Four threads, each with its own stack

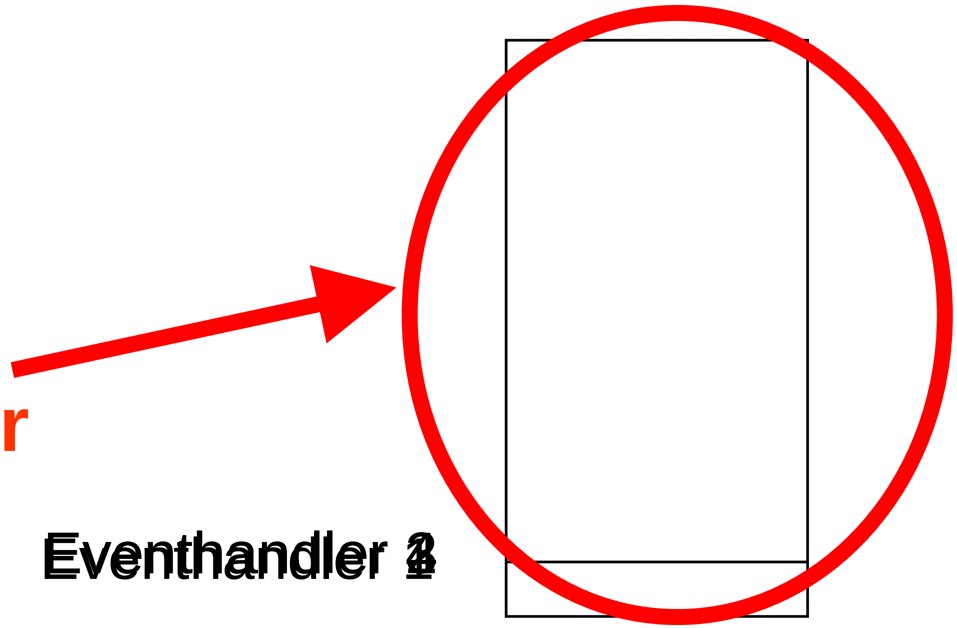


Events require one stack

- Four event handlers, one stack

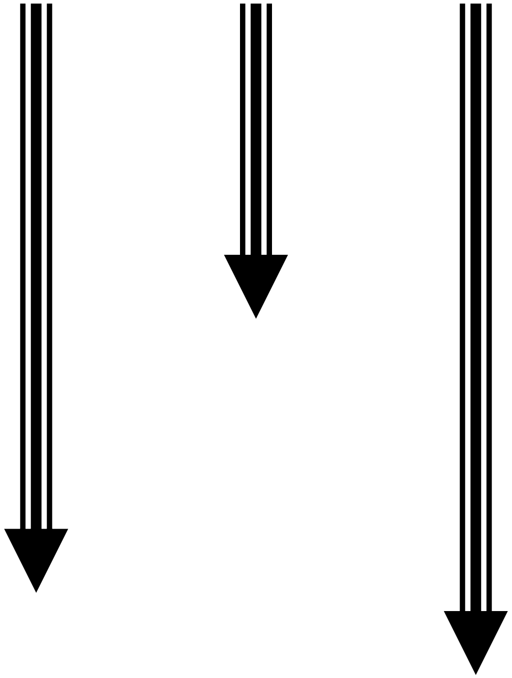


Stack is reused for every event handler

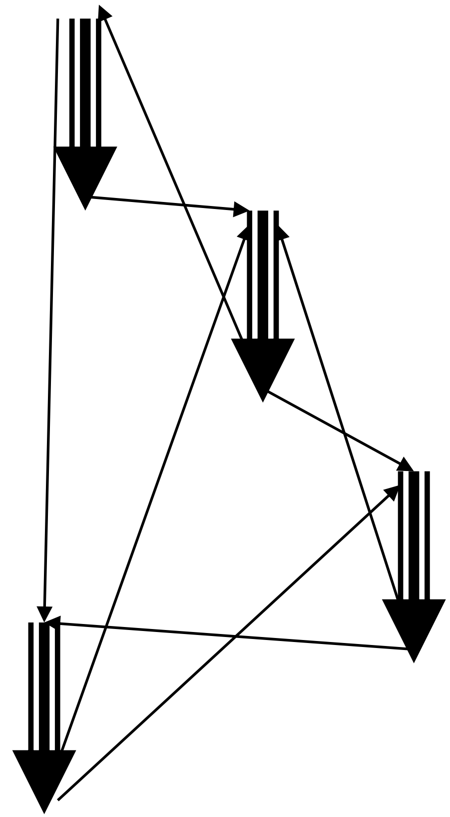


The problem with events: code flow

Threads: sequential code flow



Events: unstructured code flow



Very much like programming with GOTOs

Contiki: Combining event-driven and threads

- Event-based kernel
  - Low memory usage
  - Single stack
- Multi-threading as a library (mt_*)
  - For those applications that needs it
  - One extra thread, one extra stack
- The first system in the sensor network community to do this

However...

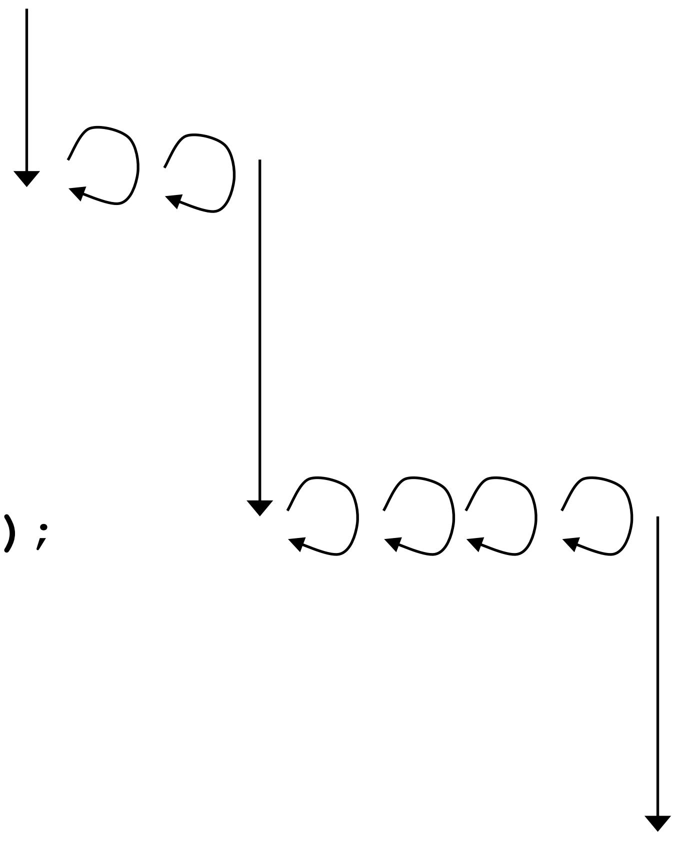
- Threads still require stack memory
- Unused stack space wastes memory
  - 200 bytes out of 2048 bytes is a lot!
- A multi-threading library quite difficult to port
  - Requires use of assembly language
  - Hardware specific
  - Platform specific
  - Compiler specific

Protothreads:
A new programming abstraction

- A design point between events and threads
- Programming primitive: conditional blocking wait
  - PT_WAIT_UNTIL(*condition*)
- Single stack
  - Low memory usage, just like events
- Sequential flow of control
  - No explicit state machine, just like threads
  - Programming language helps us: **if** and **while**

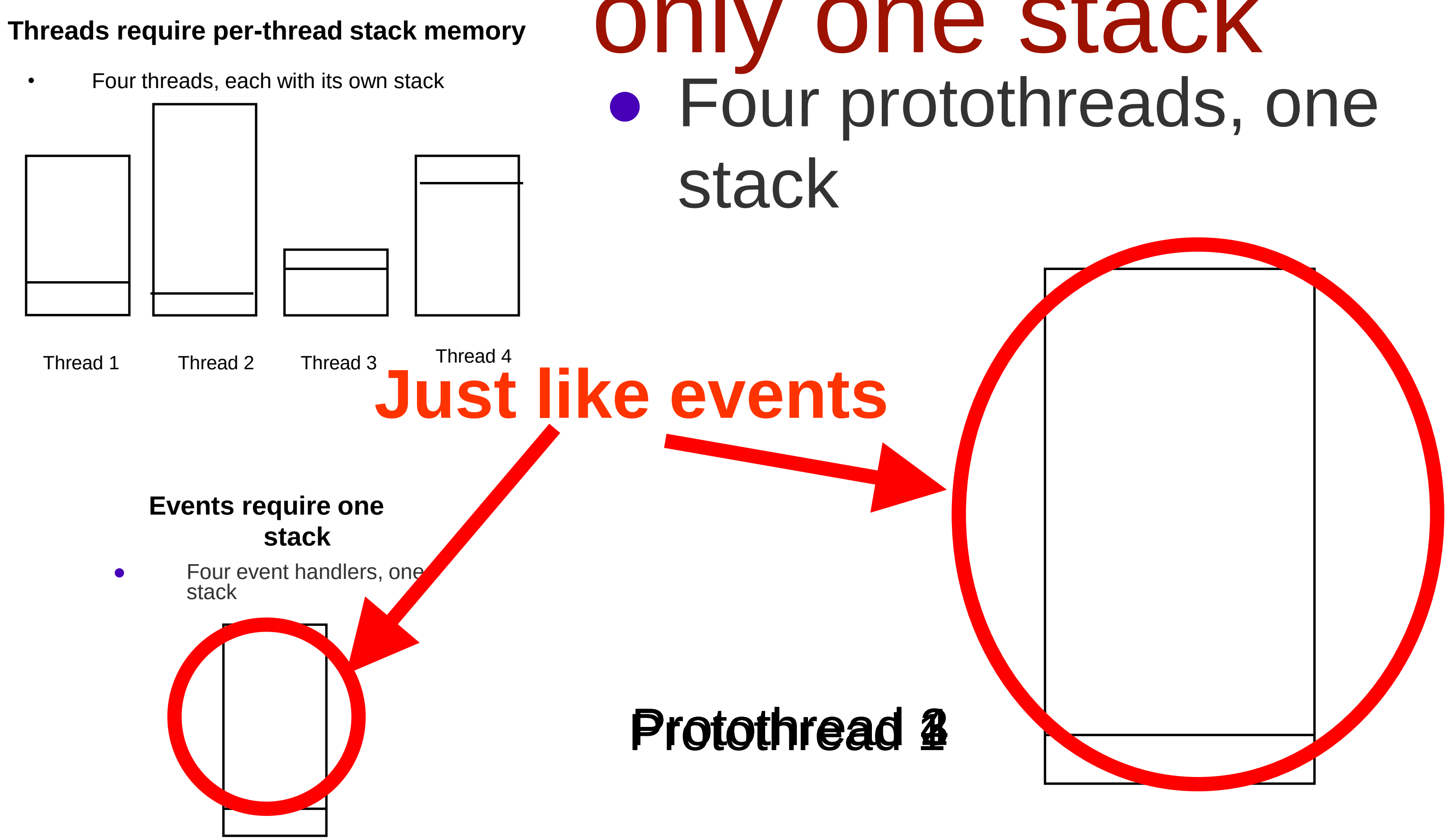
An example protothread

```
int a_protothread(struct pt *pt) {
  PT_BEGIN(pt);

   /* … */

  PT_WAIT_UNTIL(pt, condition1);

   /* … */

  if(something) {

     /* … */

    PT_WAIT_UNTIL(pt, condition2);

     /* … */

  }

  PT_END(pt);
}
```



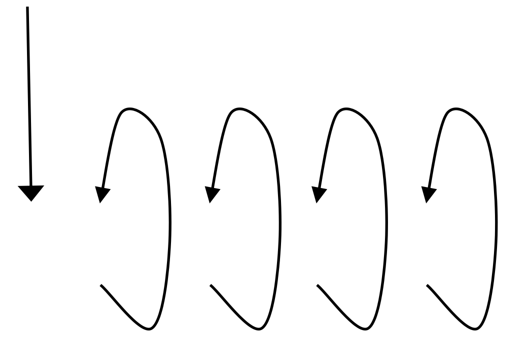
Protothreads require only one stack

- Four protothreads, one stack



Contiki processes are protothreads

```
PROCESS_THREAD(hello_world_process, ev, data) {
  PROCESS_BEGIN();
  printf("Hello, world!\n");
  while(1) {
    PROCESS_WAIT_EVENT();
  }
  PROCESS_END();
}
```



Limitations of the protothread implementation

- Automatic variables not stored across a blocking wait
  - Compiler does produce a warning
  - Workaround: use static local variables instead
- Constraints on the use of switch() constructs in programs
  - No warning produced by the compiler
  - Workaround: don’t use switches

Timers in Contiki

Four types of timers

- struct timer
  - Passive timer, only keeps track of its expiration time
- struct etimer
  - Active timer, sends an event when it expires
- struct ctimer
  - Active timer, calls a function when it expires
  - Used by Rime
- struct rtimer
  - Real-time timer, calls a function at an exact time

Using etimers in processes

```
PROCESS_THREAD(hello_world_process, ev, data) {
  static struct etimer et;        /* Must be static      */
  PROCESS_BEGIN();                /* since processes are */
                                  /* protothreads        */
  while(1) {
    /* Using a struct timer would not work, since
       the process would not be invoked when it expires. */
    etimer_set(&et, CLOCK_SECOND);
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
  }
  PROCESS_END();
}
```

The Contiki build system

The Contiki build system

- Purpose 1: easy to recompile applications for different platforms
- Purpose 2: keep application code out of the Contiki directories
- Only need to change the make command to build for different platforms
- Ideally, no changes needed to the programs
  - In practice, not all ports support everything
  - Particularly low-level hardware stuff

Example: building hello world

- cd examples/hello-world

- make TARGET=native hello-world — Build monolithic system for native
- ./hello-world.native — Run entire Contiki system + app

- make TARGET=sky hello-world — Build monolithic system image
- make TARGET=sky hello-world.u — Build & upload system image
- make TARGET=sky hello-world.ce — Build loadable module

- make TARGET=esb — Monolithic system image for ESB
- make TARGET=esb hello-world.u — Build & upload image
- make TARGET=esb hello-world.ce — Build loadable module

make TARGET=

- TARGET=name of a directory under platform/
- make TARGET=xxx savetarget
  - Remembers the TARGET
  - Example: make TARGET=sky savetarget

Developing software with Contiki

Developing software with Contiki

- The first rule of software development with Contiki:
  - Don’t develop in the target system!
    - Unless you are writing really, really low-level code
  - Do as much as possible in simulation first
    - Contiki helps you a lot
    - Native, netsim, minimal-net ports, Cooja
  - Much easier and less tedious to debug code in simulation
    - Also allows you to see global behavior

Developing software with Contiki

- The second rule of software development with Contiki:
  - Keep your code in a separate project directory
  - Helps keep application code separate from the Contiki source code
    - If you need to change the Contiki source code, make a separate copy of the file in the project directory
    - Local files override Contiki files
  - Simple Makefile in project directory

Makefile in project directory

CONTIKI = (path to Contiki directory)

all: name-of-project-file-without-extension

include $(CONTIKI)/Makefile.include

examples/hello-world/Makefile

```
CONTIKI = ../..
all: hello-world
include $(CONTIKI)/Makefile.include
```

Suggested work plan for developing Contiki code

1. Play around with the target hardware
   - Purpose: get to know the hardware
   - Write small programs that blink LEDs, produce output
2. Put the target hardware away
3. Develop the application logic in simulation
   - Test the application in simulation
4. Recompile for the target hardware
   - Test application on the target hardware
   - Fix bugs and finish up on target hardware

Case study: rewrite xmac.c

- cd examples
- mkdir new-xmac
- cd new-xmac
- cp ../sky-shell/* .
- cp ../../core/net/mac/xmac.c .
- make sky-shell.upload
- make login [run application]
- [edit xmac.c, repeat make sky-shell.upload]

Contiki directory structure

apps/ – architecture independent applications
    One subdirectory per application
core/ – system source code
    Subdirectories for different parts of the system
cpu/ – CPU-specific code
    One subdirectory per CPU
doc/ – documentation
examples/ – example project directories
    Subdirectories with project
platform/ – platform-specific code
    One subdirectory per platform
tools/ – software for building Contiki, sending files

In summary

- Processes and protothreads
  – Process run when events are posted
- Timers: etimers for processes
- The build system: app code separate
- Developing software for Contiki: don't develop in the target system

Getting code into Contiki

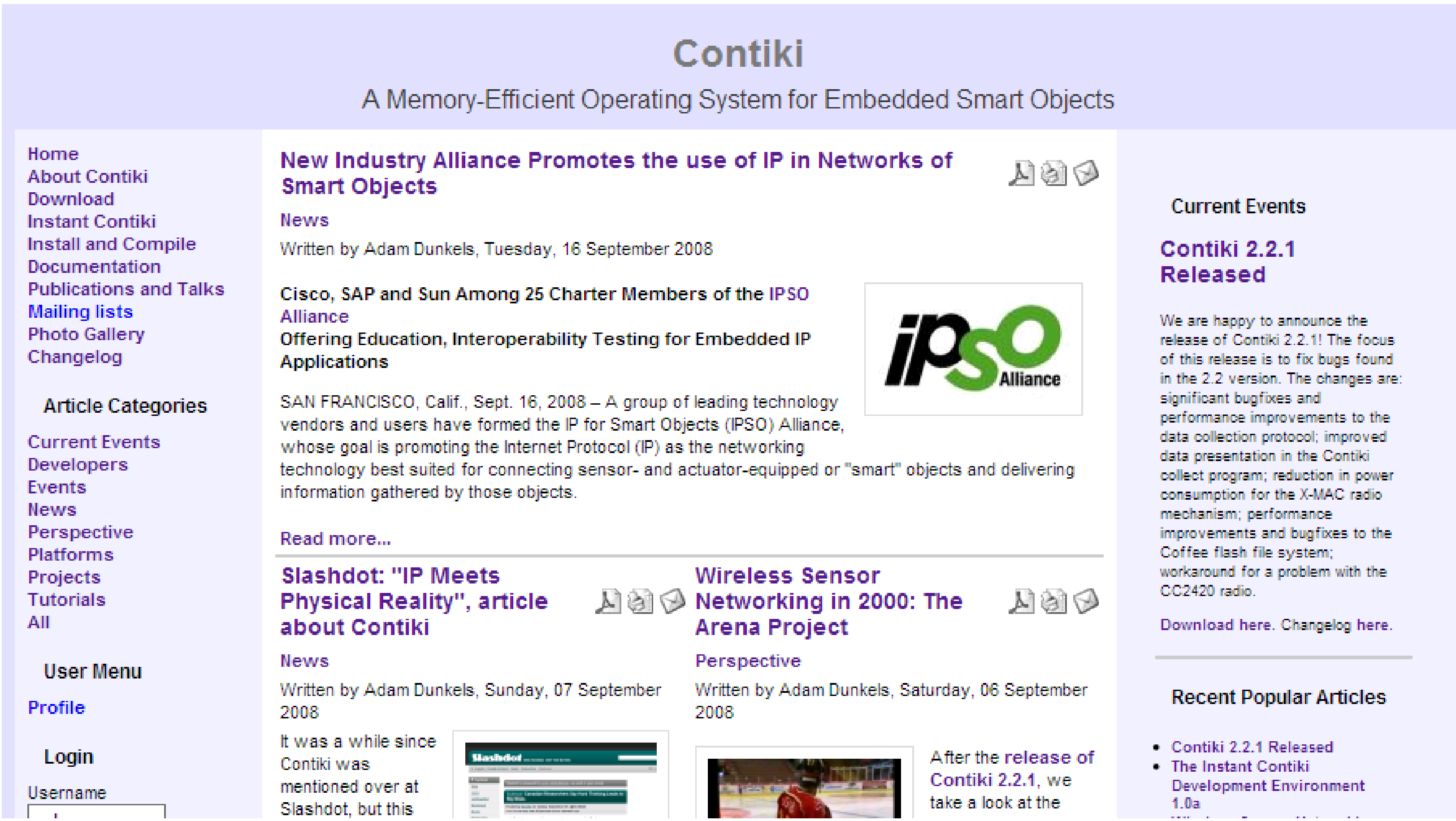
- We are very happy to receive bugfixes, comments!
- Code must
  - Work
  - Be of high quality
  - Follow the naming style
  - Follow the code style

Conclusions

- Contiki – a pioneering OS for sensor networks
  - Makes research useful
  - Simplicity and clarity instead of excessive complexity
- Communication with uIP and Rime
- Programming with protothreads and processes
- Help Contiki by writing great papers
  - Publish at high-impact venues!

Thank you

http://www.sics.se/contiki/



Coding and naming standards

Coding and naming standard

- Important for keeping the project consistent
- When writing code that might end up in Contiki, use the Contiki coding standard from the start
  - Harder to change the look afterwards

Names in Contiki

- In Contiki, all names are prefixed with the module name
  - process_start(), rime_init(), clock_time(), memb_alloc(), list_add()
- Prefix makes it possible to mentally locate all function calls
- All code must abide by this
  - There are a few exceptions in the current code, but those are to be removed in the future

What to avoid

- noCamelCase()
- No_Capital_Letters()
  - #define EXCEPT_FOR_MACROS()

File names

- Contiki uses hyphenated-file-names rather than underscores_in_file_names

Writing a MAC protocol

- core/net/mac/nullmac.c
  - Simplest MAC protocol (always on)
- core/net/mac/lpp.c
  - Simple power-saving MAC protocol [IPSN 2007]
- core/net/mac/xmac.c
  - More complex power-saving MAC protocol [SenSys 2006]

Writing a radio driver

- core/dev/cc2420.c

Two ways to make a process run

- Post an event
  - process_post(process_ptr, eventno, ptr);
    - Process will be invoked later
  - process_post_synch(process_ptr, eventno, ptr);
    - Process will be invoked now
  - Must **not** be called from an interrupt (device driver)
- Poll the process
  - process_poll(process_ptr);
  - Sends a PROCESS_EVENT_POLL event to the process
  - Can be called from an interrupt