EE 579: Wireless and Mobile Networks Design & Laboratory

Lecture 7

Amitabha Ghosh
Department of Electrical Engineering
USC, Spring 2014

Lecture notes and course design based upon prior semesters taught by Bhaskar Krishnamachari and Murali Annavaram.

Outline

- Administrative Stuff
- Contiki Internals – Research Papers
- Hands-on with Tmote Sky

Project Pitch (10% of Grade)

- Presentations in class on March 11, 2014
- 15 min max for each team (2 to 3 students)
- Make slides
- Choose a name for your project
- Email Suvil and me the name of your project and the names of your team members by March 7, 2014
- Reports due (upload in Blackboard) by Tuesday, March 25, 2014 midnight (after Spring break 17-21st March)

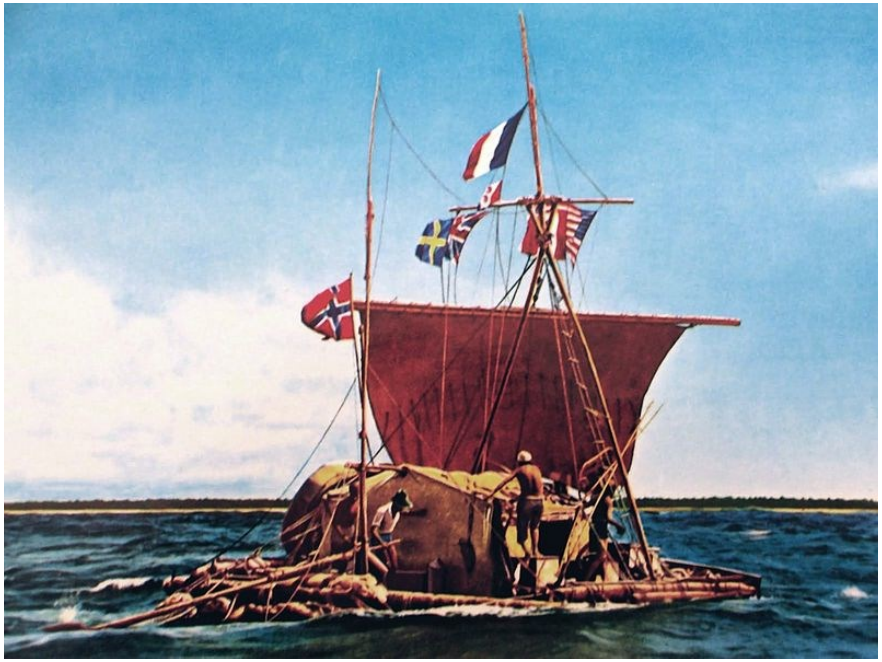
Project Pitch (10% of Grade)

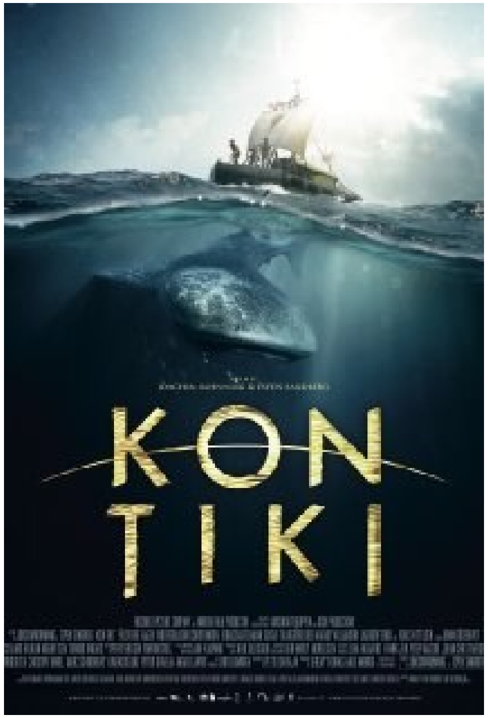
- Report Format
  - Max 5 pages (including text, images, tables, bibliography, etc.) on A4/Letter size page, single column, single spacing, at least 10 point font, Times New Roman font

- Report Content
  - Summary of your proposal
  - Motivation / applications in real life; state-of-the-art
  - Preliminary design (pics, tables, etc.), if any
  - Evaluation / experimentation / demo plan
  - Task breakdown and milestones (who will do what and by when)

The Name Contiki

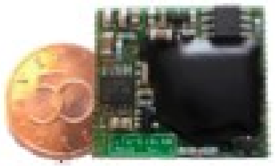
- The Kon-Tiki raft
  - Used by Norwegian explorer and writer Thor Heyerdahl in his 1947 expedition across the Pacific Ocean from South America to the Polynesian islands with minimal resources
  - Named after the Inca sun god, Viracocha, whose old name was "Kon-Tiki"





Background: The Arena Project (2000)

- Ice hockey players with wireless sensors
  - Bluetooth sensors, camera on helmet (802.11)
  - M16C CPU with 20 KB RAM and 100 KB flash ROM

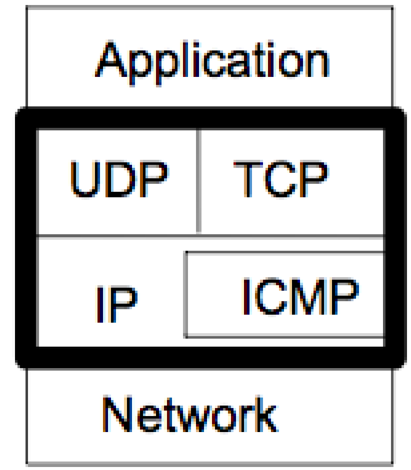


- Spectators with access to sensor readings, enriching viewing experience
  - Transmitted with UDP/IP over an ad hoc Bluetooth connection
  - lwIP (lightweight IP) stack – developed in SICS for resource-constrained devices

- Lulea Hockey lost 1-4 to Brynas Hockey
  - Technology worked, but players did not like to have a breathing rate sensor in their noses

Background: uIP (2001)

- uIP (micro IP) – world's smallest open source TCP/IP stack compatible with 8/16 bit micro-controllers
  - Developed by Adam Dunkels at SICS



- ~5K code, ~2K RAM
  - Smallest configuration: ~3K code, ~128 bytes RAM
- Unusual design choices to reduce resource usage
  - Only one packet buffer, used in a half-duplex way (tx, rx in turn)
  - Connection management using an array
- RFC and industry compliant
  - Cisco, Atmel, and SICS released uIPv6 (2008)

lwIP and uIP Today

- Very well-known in the embedded community
- Used in products from 100+ companies
- Covered in several books on embedded systems
- Porting uIP in professional magazines
- Competence specifically required in job postings
- Companies: GE Security, Cisco Systems, Pumpkin, …

Contiki OS (2002)

- Contiki – pioneering open source operating system for sensor networks
  - IP networking
  - Hybrid threading model, protothreads
  - Dynamic loading
  - Power profiling – measure network power consumption
  - Network shell – makes interaction easier
  - Rime stack – makes network programming easier
  - Multitasking using C language
  - Highly portable – 14 platforms, 5 CPUs

- Small memory footprint targeted for small embedded processors with networking
  - 50% of all processors are 8-bit, e.g., MSP430, AVR, ARM7, 6502, ...

Original Research Paper

[1] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt, "Contiki – a Lightweight and Flexible Operating System for Tiny Networked Sensors," *IEEE Local Computer Networks (LCN)*, pp. 455-462, November 2004.

Design Features

- Downloading code at run-time
- Portability
- A hybrid of multi-threaded and event-driven system model
- Next: Comparisons with existing system and Contiki

Downloading Code at Run-Time

- Large-scale sensor networks
  - Download program code, fix bugs in operational networks
- Reduce the number of bytes and transfer time
  - Not feasible to physically collect and reprogram all sensor devices
- Most embedded OS require a complete binary image of the entire system built and downloaded into each device
  - OS, system libraries, actual applications
- Contiki can load/unload individual applications or services at run-time
  - Smaller than entire system image; less energy and transfer time

Portability

- Increasing number of different sensor device platforms
- Unifying characteristic of today's platform is the CPU architecture
  - Memory model without segmentation or memory protection
  - Program code stored in reprogrammable ROM, data in RAM
- Contiki provides CPU multiplexing and support for loadable programs and services
  - Other abstractions are better implemented as libraries or services due to application specific nature of sensor networks
  - Provides mechanisms for dynamic service management

Problems with Existing Models

- Multi-threaded model
  - Often consumes large amounts of memory
  - Each thread must have its own stack
  - Hard to know a priori how much stack space a thread needs
  - Stack must be over-provisioned
  - Stack memory allocated and reserved during thread creation
  - Requires locking mechanisms for thread concurrency

- Event-driven system
  - Essentially a state-driven programming model
  - Processes are implemented as event handlers
  - All processes share the same stack (event handlers cannot block)
  - Locking mechanisms not needed
  - Not all programs can be expressed as state machines (e.g., crypto); states hard to maintain for programmers

Contiki: A Hybrid Model

- Benefits of both event-driven systems and pre-emptible threads
- An event-driven kernel
- Pre-emptive multi-threading implemented as an application library

Preemptive Multi-Threading

- Implemented as a library on top of event-based kernel
- Library is optionally linked with applications that explicitly require a multi-threaded model of operation
- Two parts to the library:
  - Platform independent part, interfacing with event kernel
  - Platform specific part, implementing stack switching and pre-emption primitives (using timer interrupt)
- Each thread needs a separate stack, and executes on its own stack until yielded or pre-empted

Preemptive Multi-Threading

- Four function APIs for multi-threading library that can be called from a running thread
  - mt_yield()
  - mt_post()
  - mt_wait()
  - mt_exit()

- Two functions to set up and run a thread
  - mt_start()
  - mt_exec()

Other Embedded OS

Next, we will compare a few embedded operating systems with Contiki, and see how Contiki overcomes some of the drawbacks of earlier OS

Other Embedded OS

- TinyOS – earliest (ASPLOS 2000) open source OS targeting wireless sensor networks
  - Component-based OS from UC Berkeley
  - Programmed using NesC language
  - Built around a lightweight event scheduler as a set of cooperating tasks and processes
  - Statically linked with the kernel to a complete system image
  - Modifying the system is not possible after linking

- Contiki: provides a dynamic structure
  - Allowing programs and drivers to be replaced during run-time and without relinking

Other Embedded OS

- Mate [ASPLOS 2002]
  - A tiny virtual machine for sensor networks (UC Berkeley)
  - VM code can be downloaded at runtime
  - Built-in multi-hop routing
- MagnetOS [SIGOPS 2002]
  - Uses Java VM to distribute applications across the network
  - VM code can be made smaller than native code – reduce energy consumption during transporting
  - But increased energy spent in interpreting code
- Contiki: programs use native code
  - Can be used for all types of programs, including low level device drivers without loss of execution efficiency

Other Embedded OS

- SensorWare [MobiSys 2003]
  - Abstract scripting language for programming sensors, but target platforms not as resource constrained as motes
- EmStar [USENIX 2004]
  - Designed for less resource constrained systems
- Mantis [WSNA 2003]
  - Traditional pre-emptive multi-threaded model
  - Reserved thread stack space and locking mechanisms
- Contiki: hybrid of multi-threaded and event-based model
  - Reduces the number of kernel-provided abstractions
  - Libraries provide them which have full access to hardware

System Overview

- A running Contiki system consists of:
  - Kernel
  - Libraries
  - Program loader
  - Processes (an application program or a service)

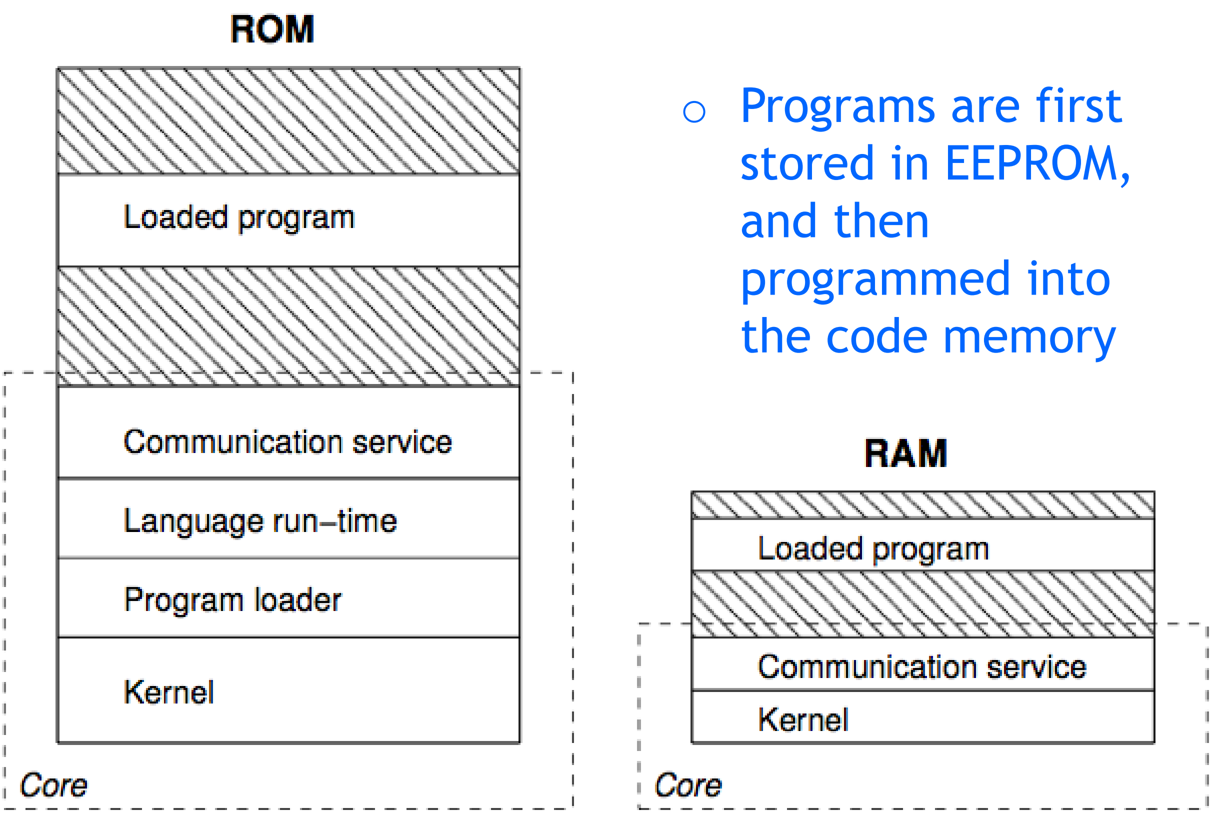
System Overview

- A running Contiki system consists of:
  - Kernel
  - Libraries
  - Program loader
  - Processes (an application program or a service)

- Process – defined by an event handler and an optional poll handler function
  - Process state is held in private memory; kernel keeps track of a pointer to the state
  - Share the same address space and do not run in different protection domains
  - Can be replaced at run-time
  - Interprocess communication is done by posting events

System Overview

- Contiki system is partitioned into two parts at compile time, and is specific to the deployment
  - Core
  - Loaded programs



- Core is compiled into a single binary image, and is stored in the devices prior to deployment
- Core is generally not modified after deployment
- Programs are first stored in EEPROM, and then programmed into the code memory

Kernel Architecture

- Consists of a lightweight event scheduler
  - Dispatches events to running processes and periodically calls processes' polling handlers
  - Program execution is triggered by events, or by polling mechanism
  - Event handlers may use internal mechanism to achieve preemption

Kernel Architecture

- Consists of a lightweight event scheduler
  - Dispatches events to running processes and periodically calls processes' polling handlers
  - Program execution is triggered by events, or by polling mechanism
  - Event handlers may use internal mechanism to achieve preemption
- Supports two kinds of events
  - Asynchronous - deferred procedure calls, enqueued and dispatched to the target process sometime later
  - Synchronous - immediately schedules the target process

Kernel Architecture

- Consists of a lightweight event scheduler
  - Dispatches events to running processes and periodically calls processes' polling handlers
  - Program execution is triggered by events, or by polling mechanism
  - Event handlers may use internal mechanism to achieve preemption
- Supports two kinds of events
  - Asynchronous - deferred procedure calls, enqueued and dispatched to the target process sometime later
  - Synchronous - immediately schedules the target process
- Provides a polling mechanism
  - Scheduling high priority events in-between asynchronous events
  - Used by processes operating near the hardware; uses a single shared stack for all process execution

Services

- A service is a process that implements functionality that can be used by other processes – form of shared library, e.g.,
  - Communication protocol stacks
  - Sensor device drivers
  - Sensor data handling algorithms

- Dynamically replaced at run-time and must be dynamically linked; special mechanisms, internal state

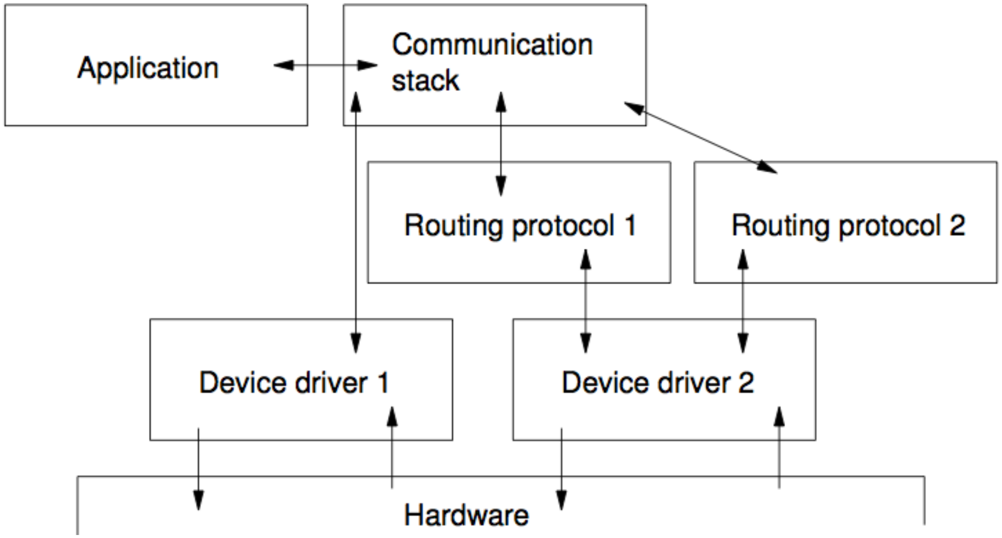
- Application programs use a stub library to communicate with the service
  - Catches the process ID

Libraries

- Kernel only provides the most basic CPU multiplexing and event handling features
- Rest of the system is implemented as libraries that are optionally linked with programs
  - Implemented as services and dynamically replaced at run-time
- Linking in three different ways:
  - Statically linked with libraries that are part of the core (often-used parts), e.g., `memcpy()`
  - Statically linked with libraries that are part of loadable programs (rarely used parts), e.g., `atoi()`
  - Programs can call services implementing a specific library

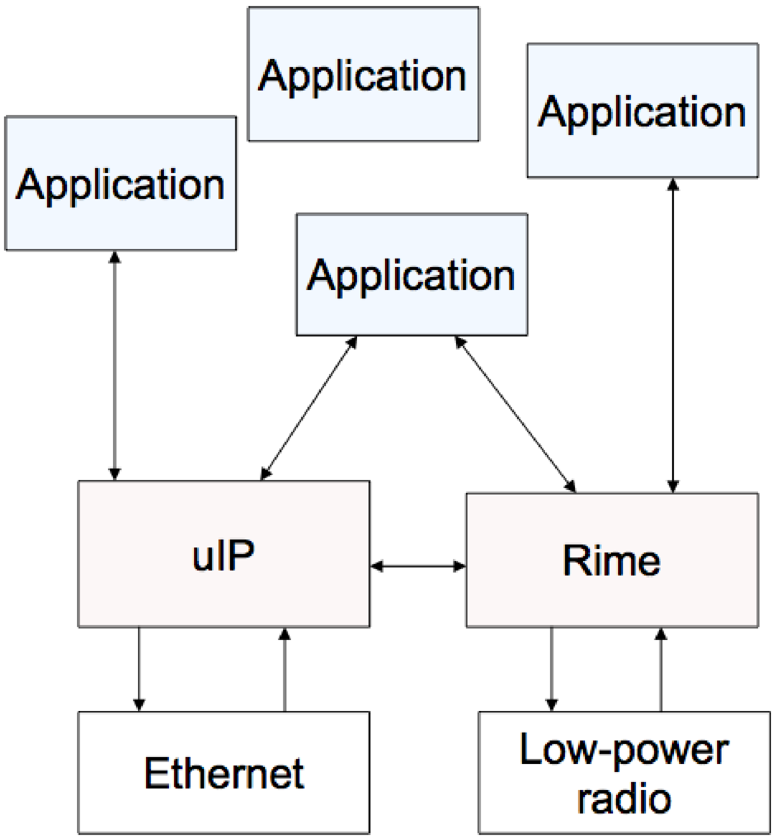
Communication Support

- Implemented as a service to enable run-time replacement
- Being a service, multiple communication stacks can be loaded simultaneously
  - Evaluate and compare different stacks
- Processes headers and posts a synchronous event
- Application program acts on the packet contents



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 Stack

- 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.
- Return packet is sent when process returns
- TCP connections periodically polled for data
- UDP packets sent with `uip_udp_packet_send()`

uIP Stack APIs

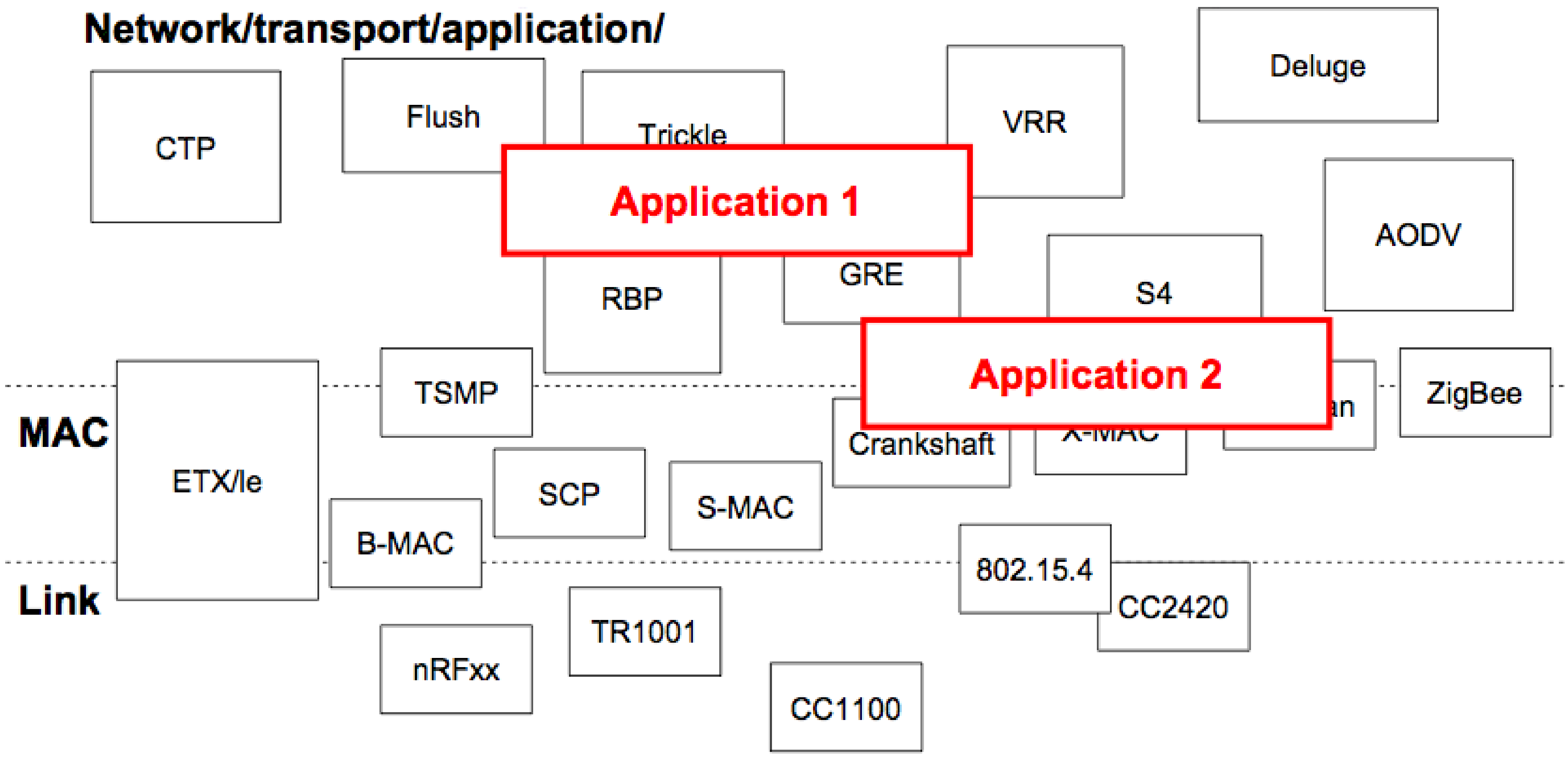
- Two APIs
  - The “raw” uIP event-driven API
  - Protosockets – sockets-like programming based on protothreads
- Event-driven API works well for small programs
  - Explicit state machines
- Protosockets work better for larger programs
  - Sequential code

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

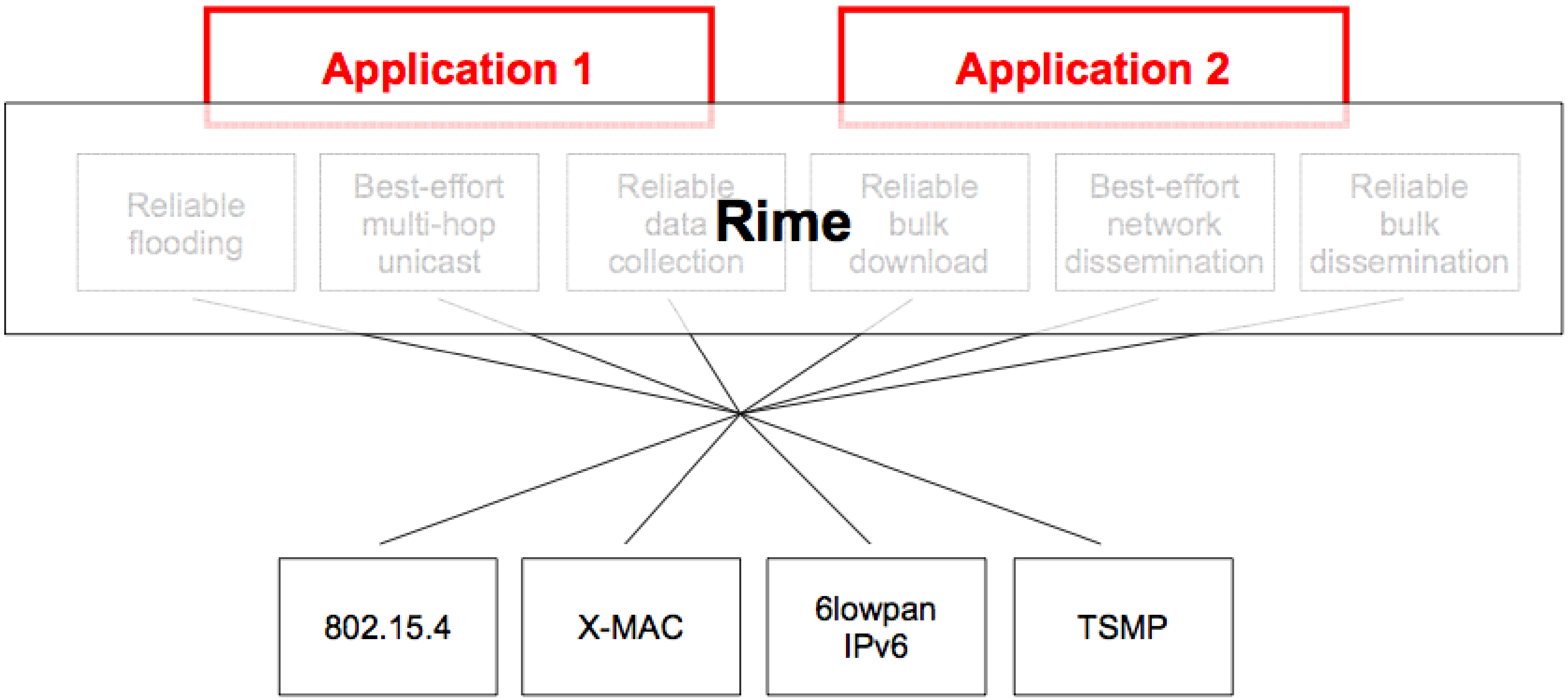
Rime: "Sockets" for Sensor Networks

Before Contiki/Rime



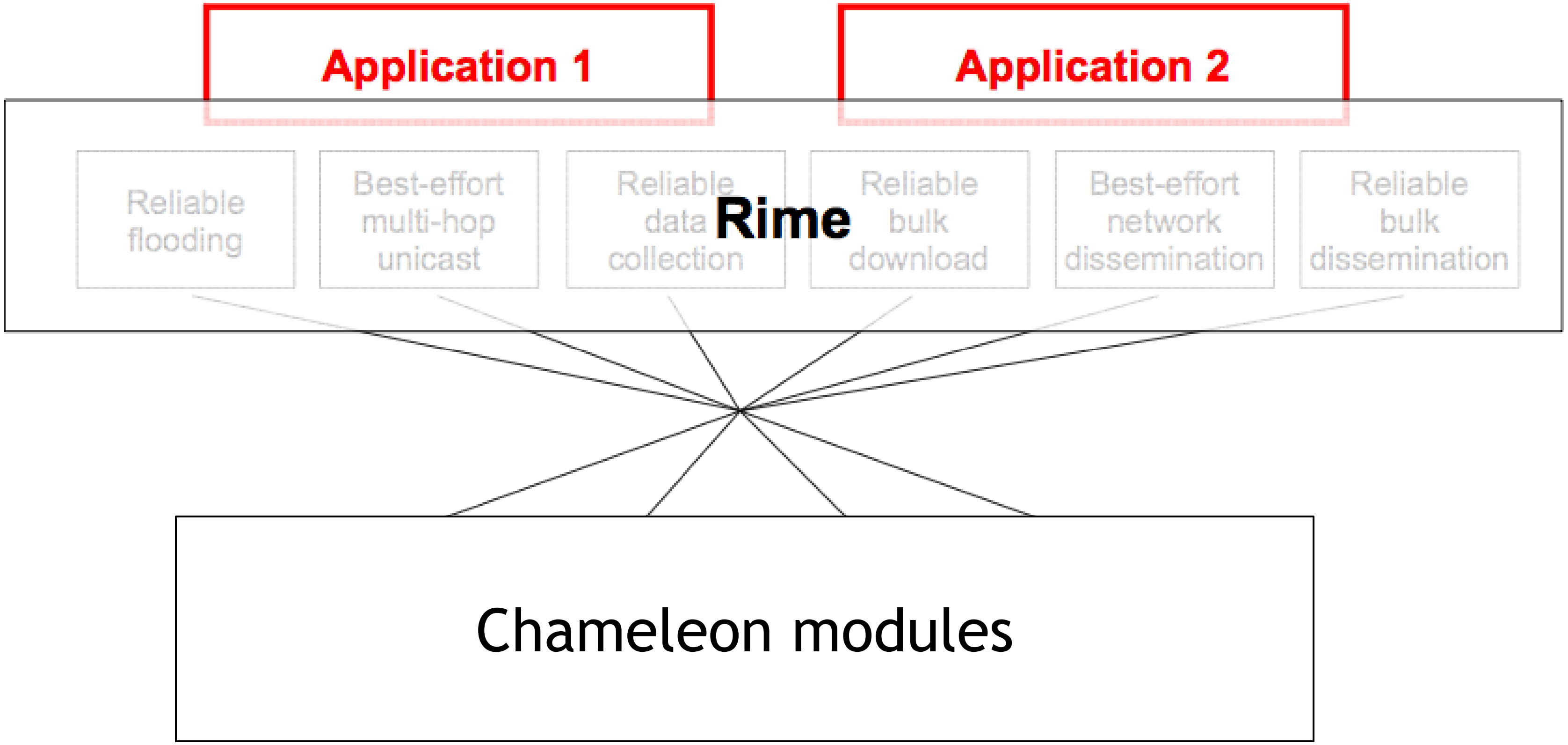
Rime: “Sockets” for Sensor Networks

With Rime



Rime: “Sockets” for Sensor Networks

With Rime



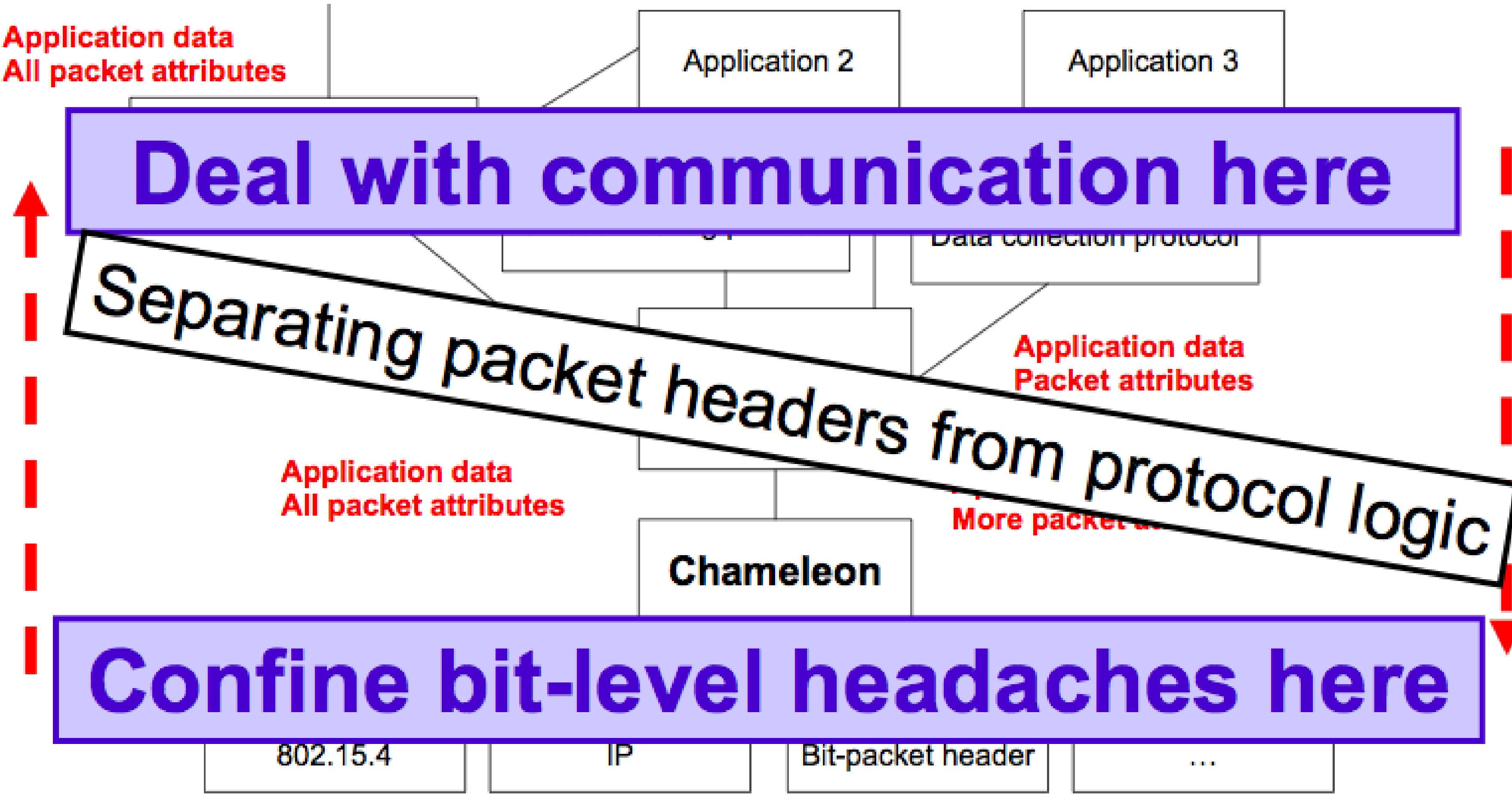
Chameleon / Rime

- 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

Chameleon / Rime



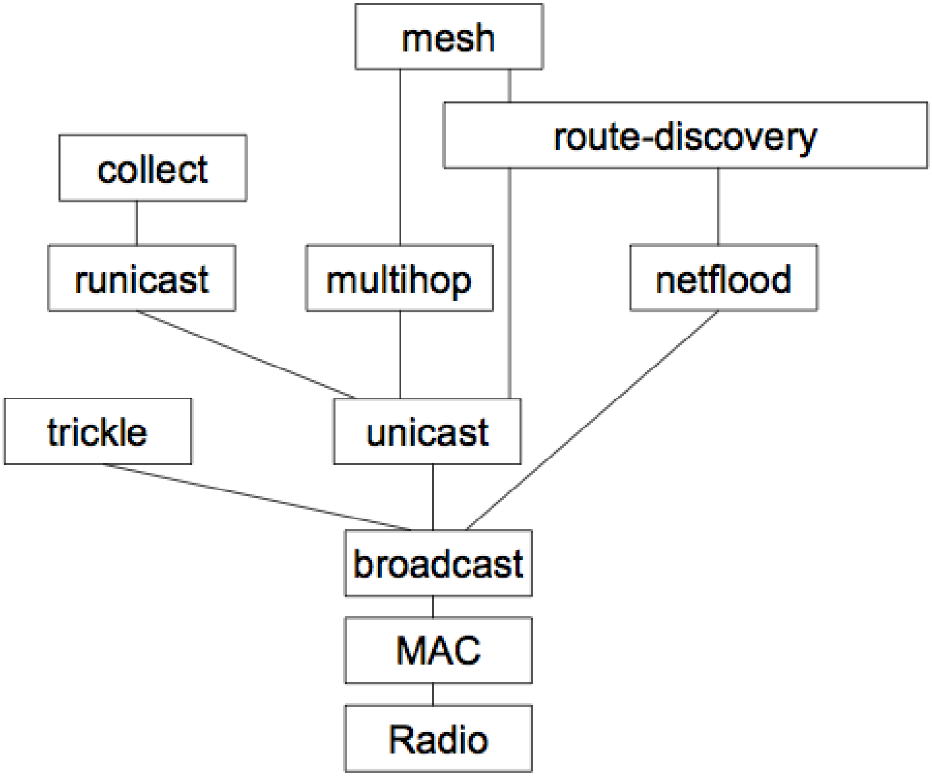
Rime: Lightweight and Layered

- 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 (multicast)
  - Hop-by-hop reliable multi-hop unicast (rmh)
  - Best-effort multi-hop flooding (netflood)
  - Reliable multi-hop flooding (trickle)
  - Hop-by-hop reliable data collection tree routine (collect)
  - Hop-by-hop reliable mesh routing (mesh)
  - Best-effort route discovery (route-discovery)
  - Single-hop reliable bulk transfer (rudolph0)
  - Multi-hop reliable bulk transfer (rudolph1)

Rime: Lightweight and Layered

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

http://contiki.sourceforge.net/docs/2.6/examples.html



An Example

- We will go through an example Contiki program step-by-step to see the structure of the code and different data structures used
- This example program opens a UDP broadcast connection and sends one packet every second

An Example

```
#include "contiki.h"
#include "contiki-net.h"
```

/* All Contiki programs must have a process */

```
PROCESS(example_program_process, "Example process");
```

/* To make the program send a packet every second, we use an event timer */

```
static struct etimer timer;
```

```
struct etimer {
    struct timer timer;
    struct etimer *next
    struct process *p; }
```

/* Implement the process. It is run whenever an event occurs, and the parameters "ev" and "data" will be set to the event type and any data that may be passed
*/

```
PROCESS_THREAD(example_program_process, ev, data){
```

An Example

/* Declare the UDP connection. This must be declared static, otherwise the contents may be destroyed, because the process runs as protothreads, which do not support stack variables
*/

```
static struct uip_udp_conn *c;
```

/* Start the process */

```
PROCESS_BEGIN();
```

```
struct uip_udp_conn {
    uip_ipaddr_t ripaddr;
    uint16_t lport;
    uint16_t rport;
    uint8_t ttl;
    uip_udp_appstate_t
              appstate;
}
```

/* Create the UDP connection to port 4321. We don't want to attach any special data to the connection, so pass it a NULL
*/

```
c = udp_broadcast_new(UIP_HTONS(4321), NULL);
```

An Example

```
/* Loop forever */
while(1) {
  /* Set a timer that wakes up once every second */
    etimer_set(&timer, CLOCK_SECOND);
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&timer));

  /* To send a UDP packet, we must call upon the uIP TCP/IP  stack to call us
(Hollywood principle: "Don't call us, we'll call you.") Use the function
tcpip_poll_udp() to tell uIP to call us, and then wait for the uIP event to come
  */
    tcpip_poll_udp(c);
    PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);
    uip_send("Hello", 5);
}
PROCESS_END();
}
```

An Example

```
#include "contiki.h"
#include "contiki-net.h"

PROCESS(example_program_process, "Example process");
static struct etimer timer;

PROCESS_THREAD(example_program_process, ev, data){
        static struct uip_udp_conn *c;
        PROCESS_BEGIN();
        c = udp_broadcast_new(UIP_HTONS(4321), NULL);
        while(1) {
                etimer_set(&timer, CLOCK_SECOND);
                PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&timer));
                tcpip_poll_udp(c);
                PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);
                uip_send("Hello", 5);
        }
        PROCESS_END();
}
```