Contiki tutorial

Xu Dingxin

2011/07/25

Outline

- **OS**
  - Event driven kernel
  - Process and protothread
- **Service**
  - Timers
- **Communication Stack**
  - Rime

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 process and protothread

- **Kernel is event-based**
  - Single stack, less memory
  - Invoke processes whenever something happens
    - Sensor events, timer expired
  - Process invocation must not block
- **Process functionality is implemented by protothread**
  - The body of process is a protothread
- **Protothreads provide sequential flow of control in Contiki process**
  - No state machine

Contiki Process

```
PROCESS(test_pt1_process, "Test pt1 Process");
PROCESS(test_pt2_process, "Test pt2 Process");
AUTOSTART_PROCESSES(&test_pt1_process);
PROCESS_THREAD(test_pt1_process, ev, data)
{
    static struct etimer et;
    PROCESS_BEGIN();
    process_start(&test_pt2_process,  NULL);
    etimer_set(&et, CLOCK_SECOND);
    while(1) {
        printf ( "in process thread 1\n" );
        etimer_reset(&et);
    }
    PROCESS_END();
}
PROCESS_THREAD(test_pt2_process, ev, data)
{
    static struct etimer et;
    PROCESS_BEGIN();
    etimer_set(&et, CLOCK_SECOND);
    while(1) {
        printf ( "in process thread 2\n" );
        etimer_reset(&et);
    }
    PROCESS_END();
}
```

Contiki Process

```
PROCESS(test_pt1_process, "Test pt1 Process");
PROCESS(test_pt2_process, "Test pt2 Process");
AUTOSTART_PROCESSES(&test_pt1_process);
PROCESS_THREAD(test_pt1_process, ev, data)
{
    static struct etimer et;
    PROCESS_BEGIN();
    process_start(&test_pt2_process,  NULL);
    etimer_set(&et, CLOCK_SECOND);
    while(1) {
        printf ( "in process thread 1\n" );
        etimer_reset(&et);
    }
    PROCESS_END();
}
PROCESS_THREAD(test_pt2_process, ev, data)
{
    static struct etimer et;
    PROCESS_BEGIN();
    etimer_set(&et, CLOCK_SECOND);
    while(1) {
        printf ( "in process thread 2\n" );
        etimer_reset(&et);
    }
    PROCESS_END();
}
```

```
PROCESS(test_pt1_process, "Test pt1 Process"
PROCESS(test_pt2_process, "Test pt2 Process"
AUTOSTART_PROCESSES(&test_pt1_process);
PROCESS_THREAD(test_pt1_process, ev, data)
{
    static struct etimer et;
    PROCESS_BEGIN();
    process_start(&test_pt2_process,  NULL);
    etimer_set(&et, CLOCK_SECOND);
    while(1) {
        printf ( "in process thread 1\n" );
        etimer_reset(&et);
        PROCESS_YIELD();
    }
    PROCESS_END();
}
PROCESS_THREAD(test_pt2_process, ev, data)
{
    static struct etimer et;
    PROCESS_BEGIN();
    etimer_set(&et, CLOCK_SECOND);
    while(1) {
        printf ( "in process thread 2\n" );
        etimer_reset(&et);
        PROCESS_YIELD();
    }
    PROCESS_END();
}
```

Process Implementation: PT_THREAD

- **PROCESS definition**

```
#define PROCESS_THREAD(name, ev, data) \
static PT_THREAD(process_thread_##name(struct pt *process_pt,        \
                                        process_event_t ev,          \
                                        process_data_t data))
```

- **PT_THREAD definition**

```
#define PT_THREAD(name_args)  char name_args
```

- **So…**

```
static char process_thread_##name(….)
```

- **Just a common function, no magic**

Inside a Process: 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);
}
```

Protothread implementation: C-switch

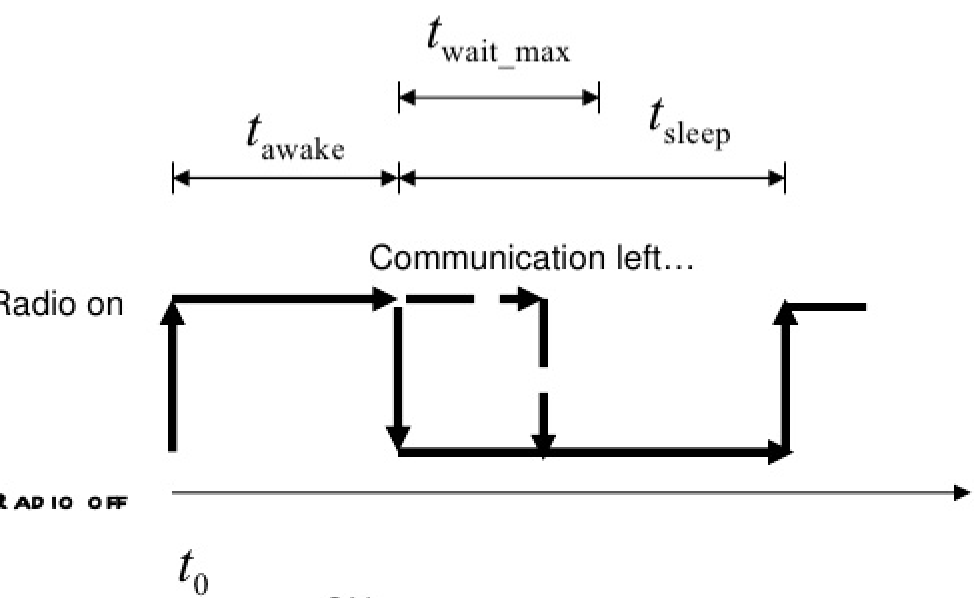
```
int a_protothread(struct pt
*pt) {
  PT_BEGIN(pt);


  PT_WAIT_UNTIL(pt,
condition1);


  if(something) {


    PT_WAIT_UNTIL(pt,
condition2);


  }

  PT_END(pt);
}
```

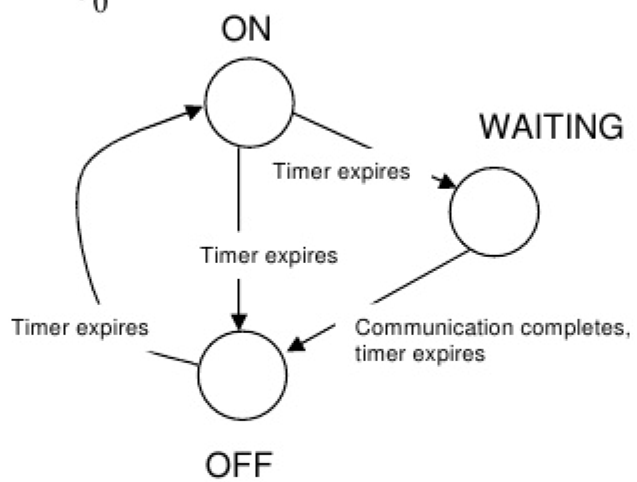
```
int a_protothread(struct pt *pt)
{
  switch(pt->lc) { case 0:


  pt->lc = 5; case 5:
  if(!condition1) return 0;

  if(something) {


    pt->lc = 10; case 10:
    if(!condition2) return 0;

  }

  } return 1;
}
```

Line numbers

Why protothread?

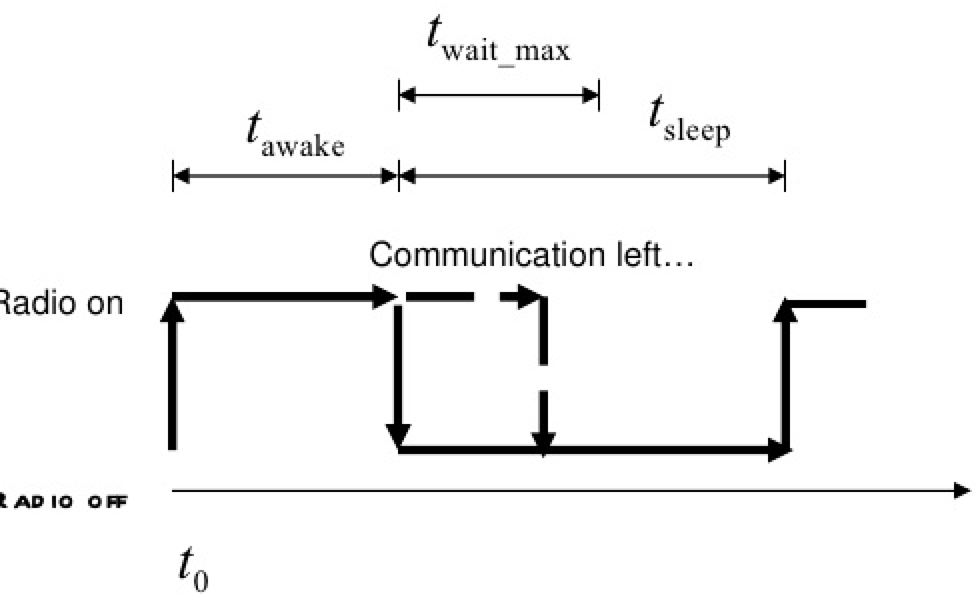
Event-driven state machine implementation: messy






```
enum {ON, WAITING, OFF} state;

void eventhandler() {
  if(state == ON) {
    if(expired(timer)) {
      timer = t_sleep;
      if(!comm_complete()) {
        state = WAITING;
        wait_timer = t_wait_max;
      } else {
        radio_off();
        state = OFF;
      }
    }
  } else if(state == WAITING) {
    if(comm_complete() ||
       expired(wait_timer)) {
      state = OFF;
      radio_off();
    }
  } else if(state == OFF) {
    if(expired(timer)) {
      radio_on();
      state = ON;
      timer = t_awake;
    }
  }
}
```

Prototthreads-based implementation is shorter




```
int protothread(struct pt *pt) {
  PT_BEGIN(pt);
  while(1) {
    radio_on();
    timer = t_awake;
    PT_WAIT_UNTIL(pt, expired(timer));
    timer = t_sleep;
    if(!comm_complete()) {
      wait_timer = t_wait_max;
      PT_WAIT_UNTIL(pt, comm_complete()
                    || expired(wait_timer));
    }
    radio off();
    PT_WAIT_UNTIL(pt, expired(timer));
  }
  PT_END(pt);
}
```

- Code shorter than the event-driven version
- Code uses structured programming (if and while statements)
- Mechanism evident from the code

Outline

- **Kernel**
  - Event driven kernel
  - Process
  - **Service**
  - Timers
  - **Communication Stack**

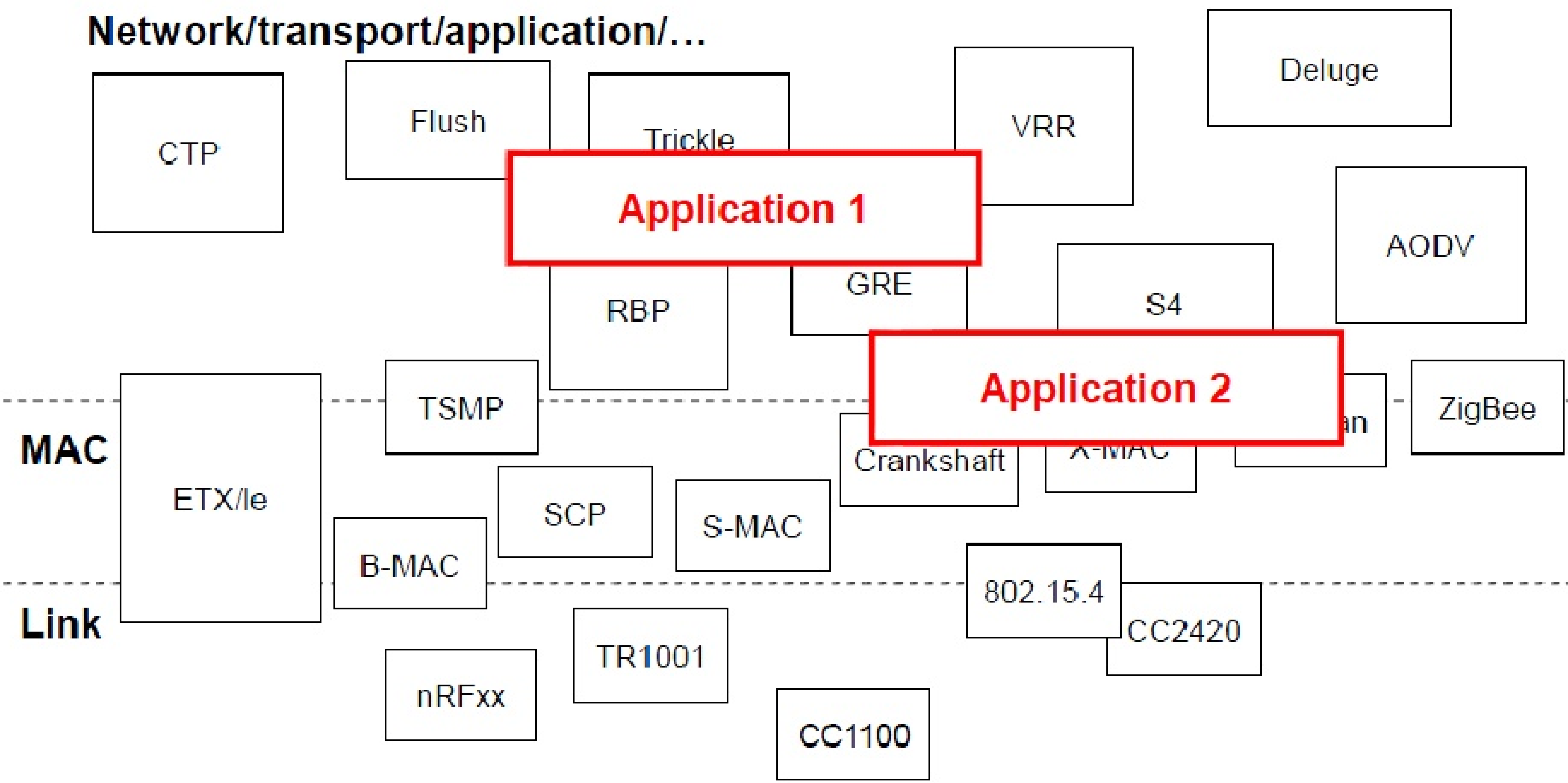
Four types of Timers

- 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

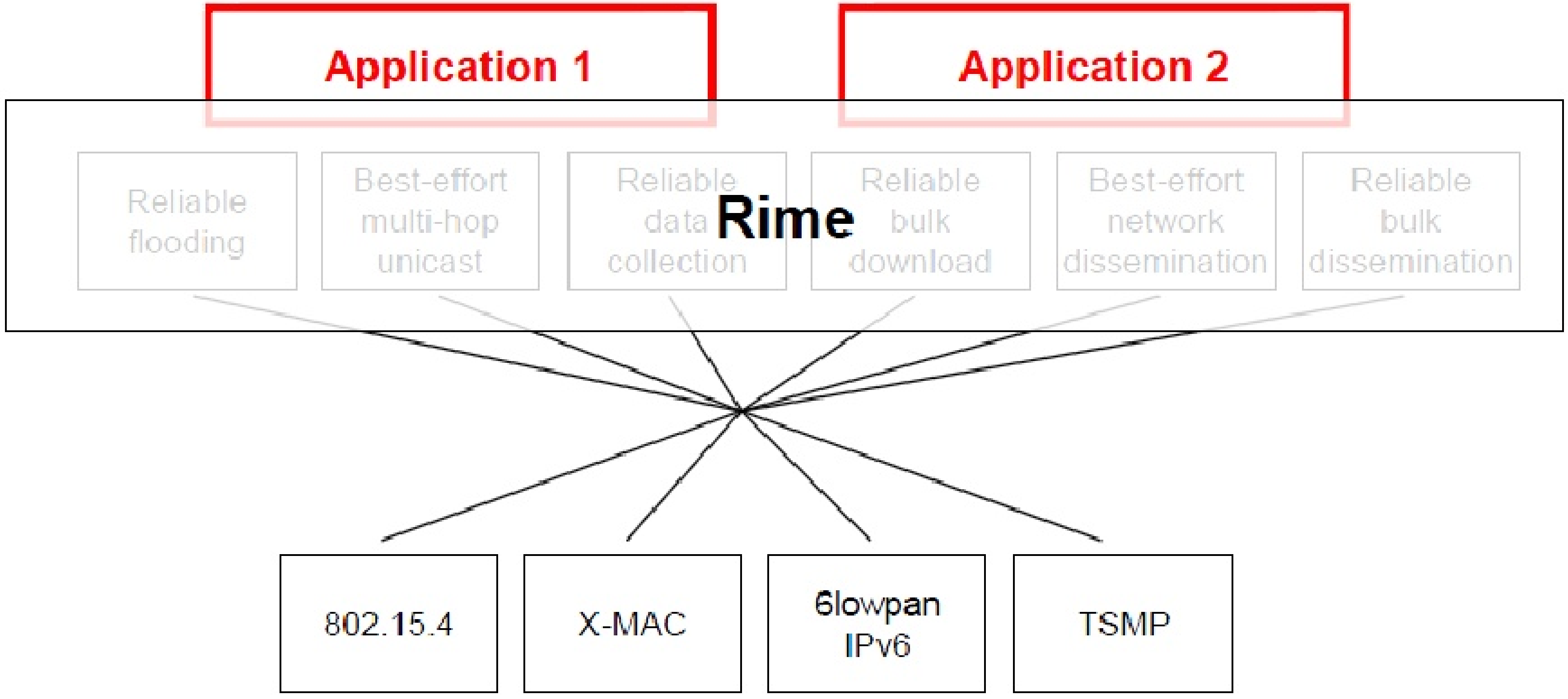
Outline

- **Kernel**
  - Event driven kernel
  - Process
  - **Service**
  - Timers
  - **Communication Stack**

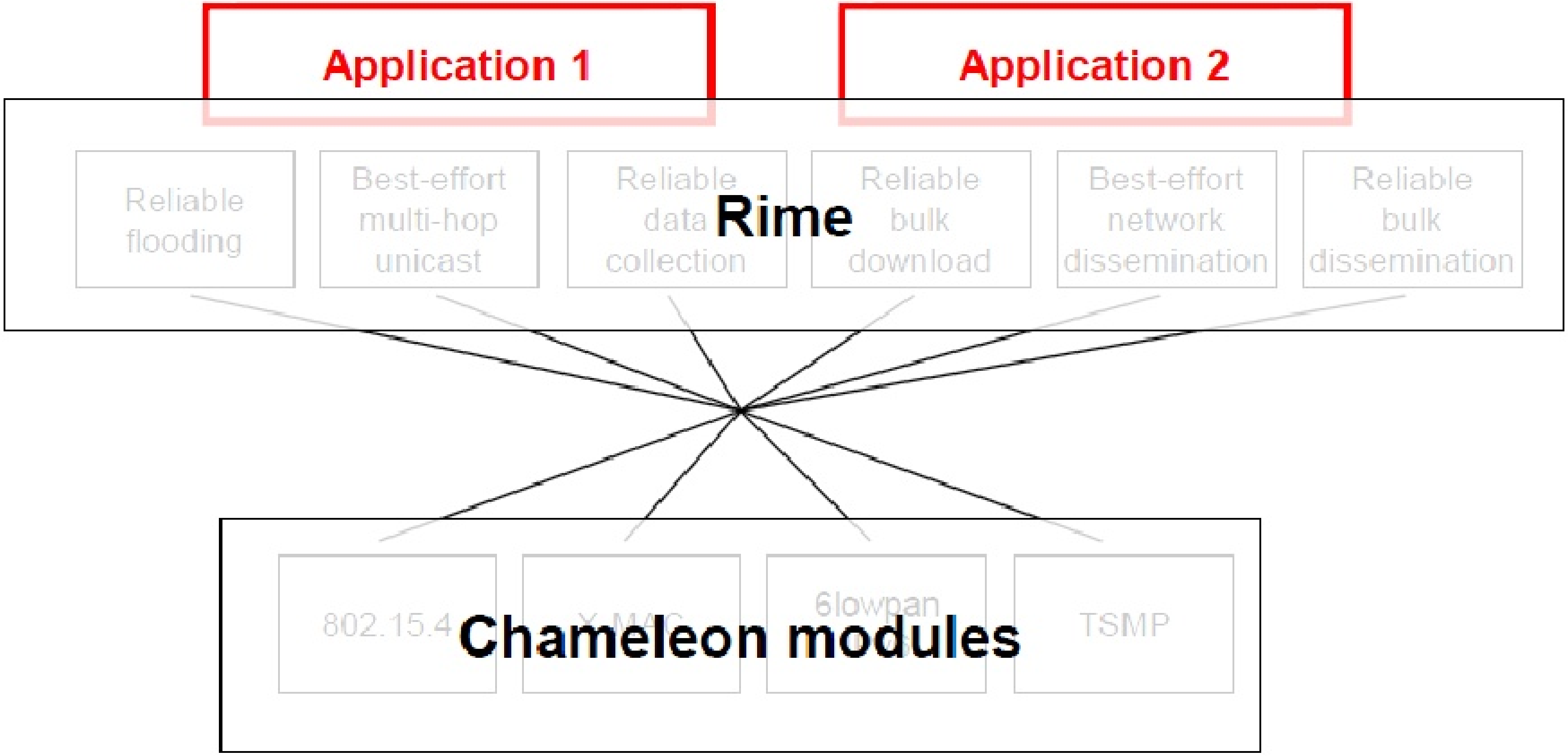
Communication programming: before Rime



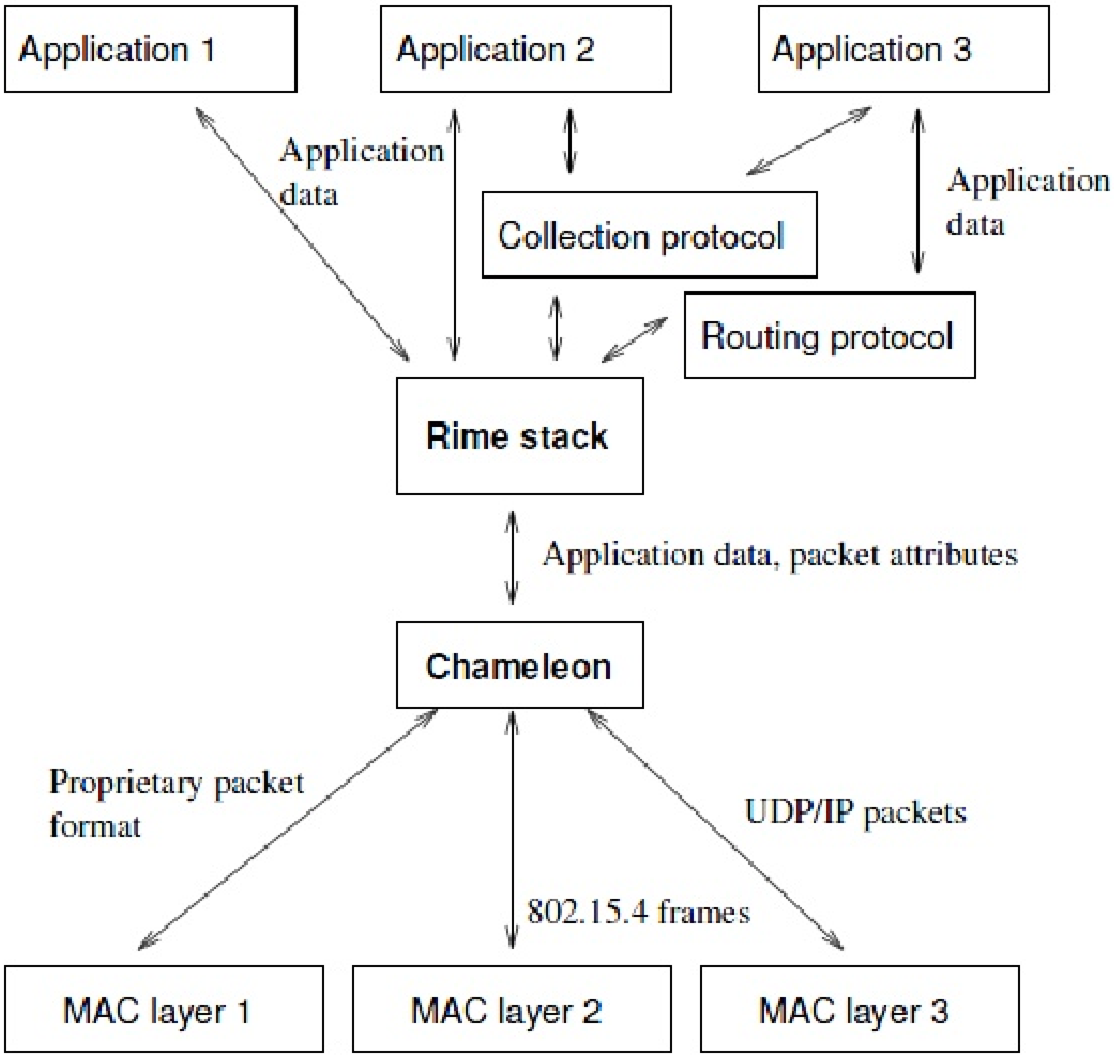
Rime: 'Sockets' for WSN



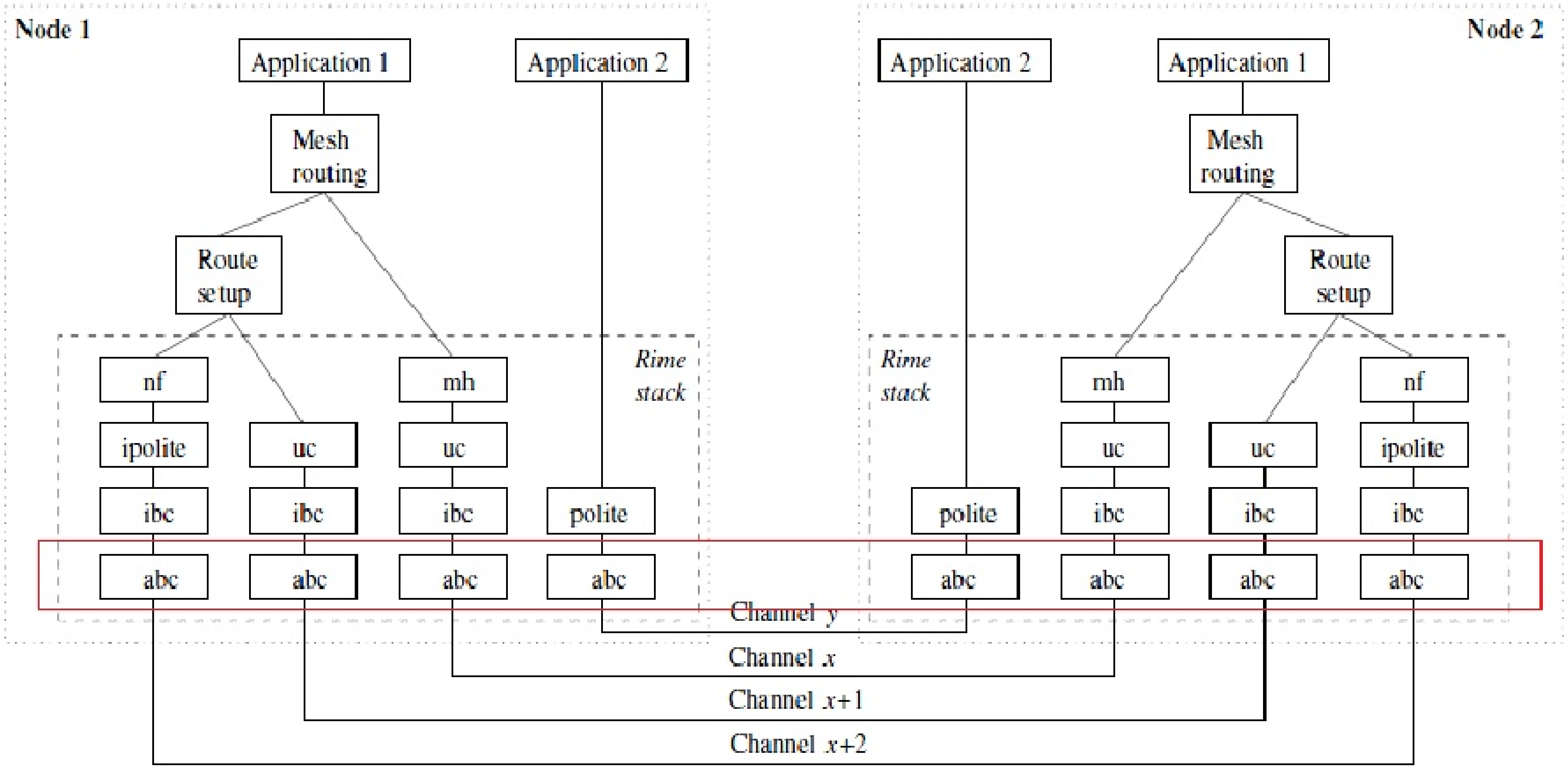
Chameleon: Adapting to underlying layers(link, MAC)



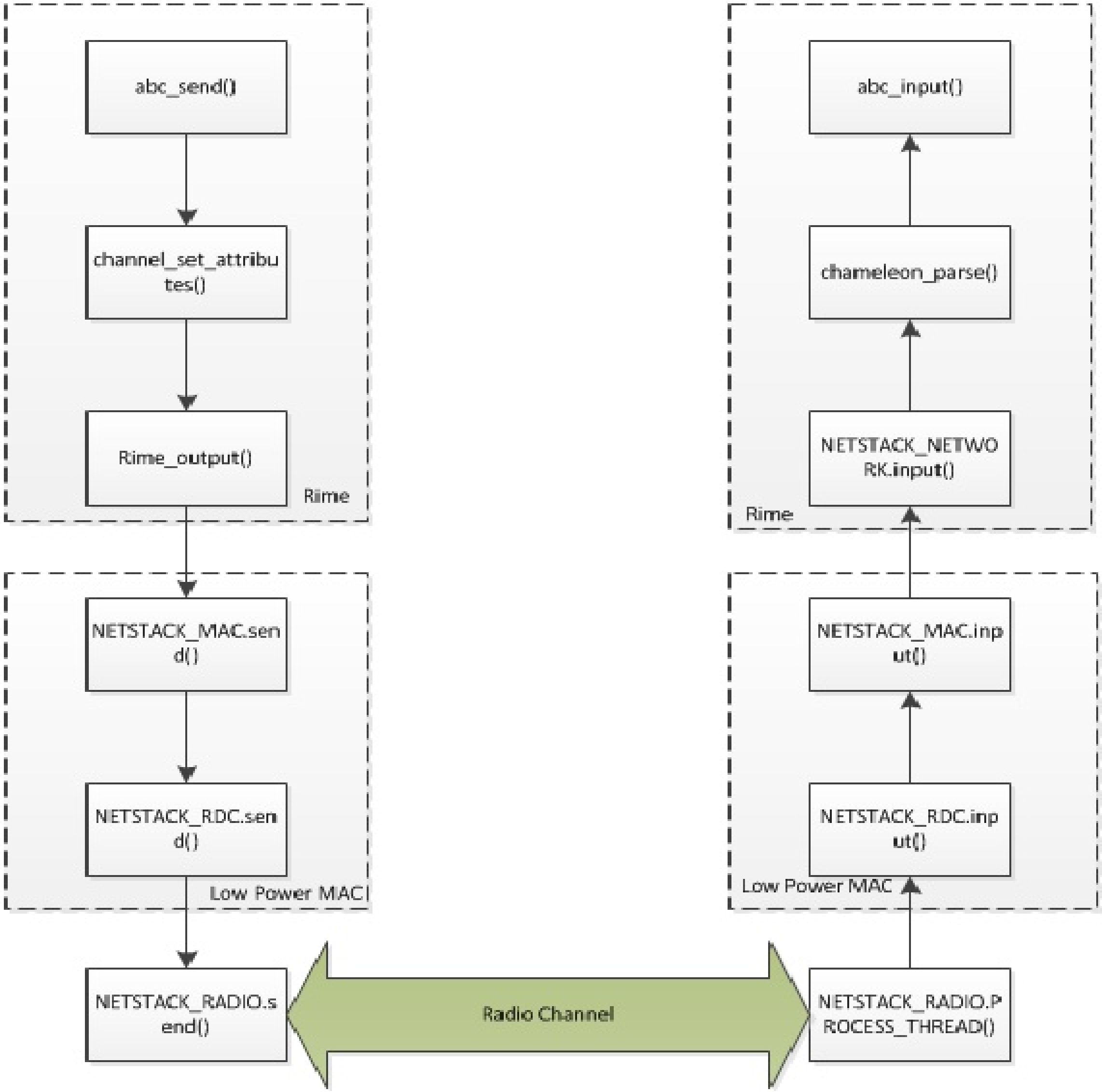
Rime/Chameleon:the whole picture



How to communicate with Rime



Underlying Rime



Programming with Rime: example

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

Step1: Message Handler

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

Step2: Connection definition

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

Step3: open the connection

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

Step4: send the message

That’s All

- Q&A