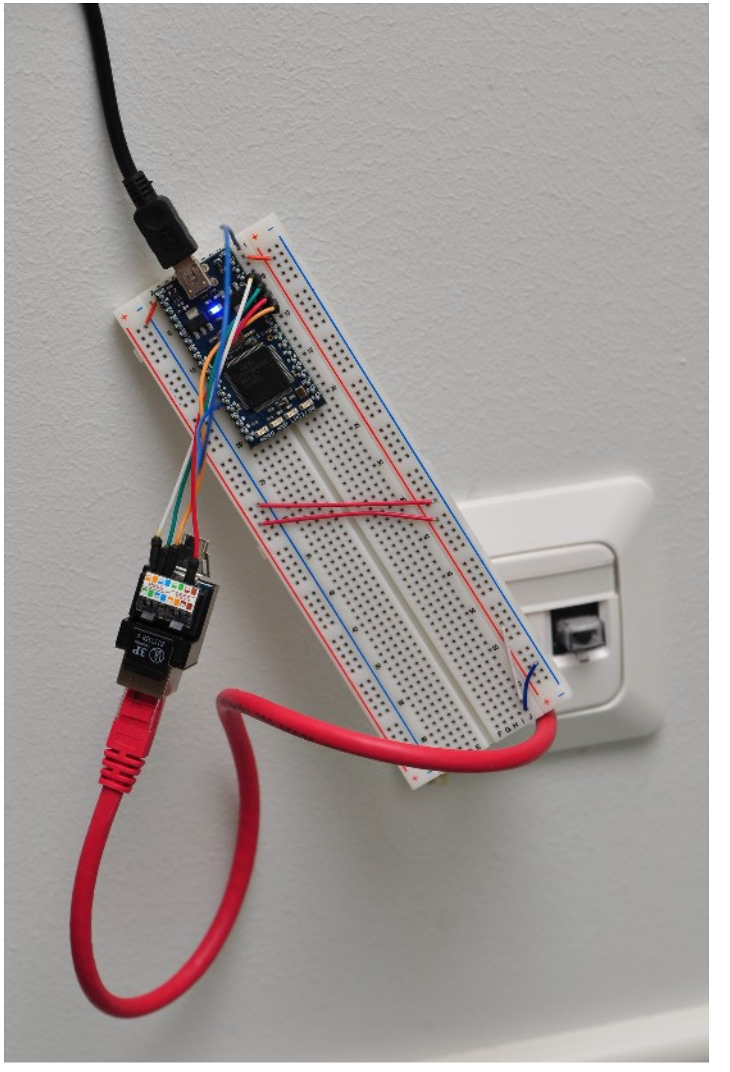
Tiny COAP Sensors

draft-arkko-core-sleepy-sensors

Jari Arkko, Heidi-Maria Rissanen, Salvatore Loreto, Zoltan Turanyi, and Oscar Novo

Ericsson Research



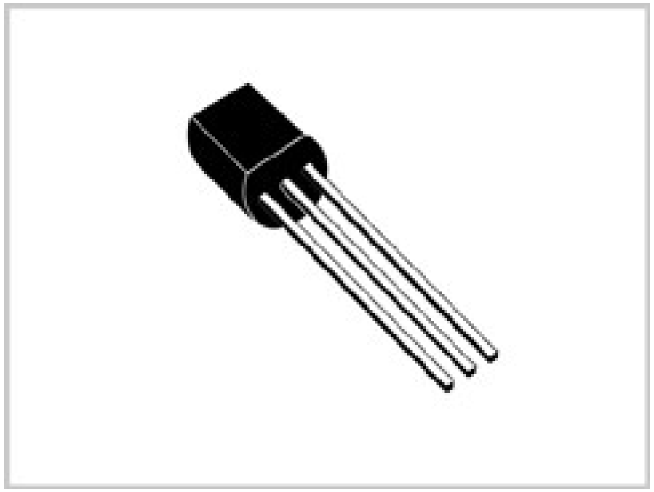
Outline

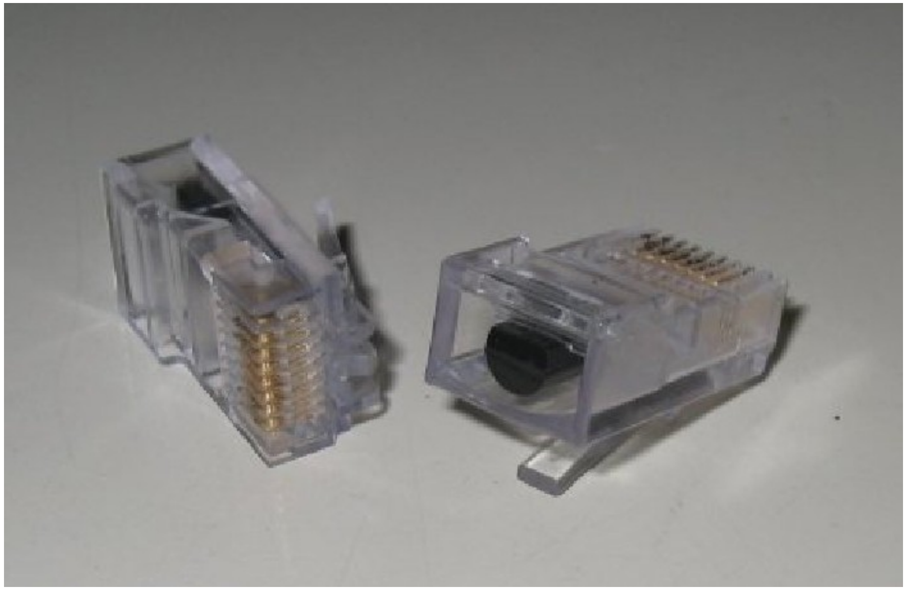
1. Motivation
2. Implementation highlights
3. Major architectural and design choices
4. Reflections on COAP
5. Implementation techniques

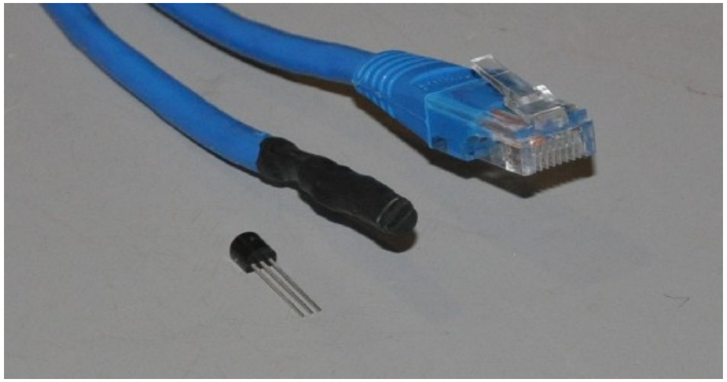
Legacy, Non-IP Technology

Can we do the same on IP?

YES we can!







Motivation

The goal was to create IP(v6) based sensors with

1. Natural support for *sleeping* nodes
2. Build something so simple that it could be re-implemented later with *gates* (not CPUs)
3. Communication models that fit the problem at hand
4. Good design from user perspective

Non-Goals

This is NOT

1. A general purpose implementation of COAP or any other protocol; we only implement what is actually needed in the application context
2. An implementation for general purpose computers
3. RFC compliance exercise. It works. 'nuff said.

Highlights from the Implementation

- Consists of 48 lines of assembler code
- Ethernet, IPv6, UDP, COAP, XML, and app
- Multicast, checksums, msg and device IDs
- Approaches theoretical minimum power usage
- No configuration needed

Look for packets to ff02::fe00:1 in the IETF wired network!

Making Small Implementions: Problem 1 – Sleeping Nodes

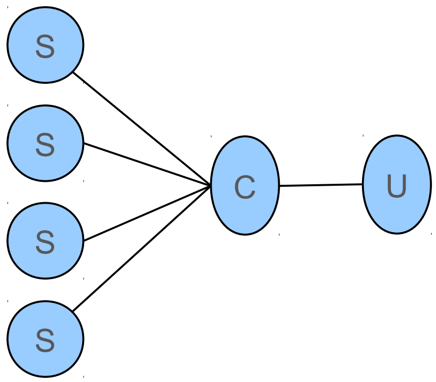
The device should ideally sleep as much as possible

The fundamental issue is having to wait for responses

- Asking for an address from DHCP, waiting for a prefix from RA, waiting for DAD responses, waiting for COAP/HTTP requests, or waiting for COAP registrations

The communication model is wrong!

Do this instead:



1. Sensors multicast their readings
2. A cache node collects the messages
3. Other nodes access the cache at any time

Power Savings Comparison

Lets assume periodic messages once per minute. On a 10Mbit/s interface sending one message takes 100 us, i.e., ratio of sleep vs. awake is 600000x

A node that wakes up for one second every minute to listen has a ratio of only 60x

10000x difference!!!

Even if we assume that it takes ten times more to wake up and process the packet than the actual line speed is, we still get a 1000x difference

Making Small Implementions: Problem 2 – Broadcast Storms

Have to avoid everyone receiving everything

IPv6 multicast can solve this problem nicely:

1. Use multicast, not broadcast (duh!)
2. Sensor-class specific multicast groups
   - Only those that want to know need to receive the packets
   - Similar to solicited node multicast address trick in ND
   - Using FF02::1:FEXX:XXXX in the prototype, XXXXXX = 1 for temperature sensors
3. Randomized sleep duration

Making Small Implementions: Problem 3 – Address Configuration

How do we get an address without having to stay awake?

The solution:

1. Use IPv6 link-local source addresses
   - No need to wait for RAs or remember prefixes
2. Use MAC-address -based generation of these addresses
3. Do not employ DAD
   - Not quite according to the RFC... but works better

Making Small Implementions: Problem 4 – Zero Configuration

How do we avoid having to configure these tiny devices?

The solution:

1. Sensor IDs are burned into the hardware at factory
2. Sensors use multicast, no need to know any specific destination addresses
3. All configuration that might be needed (e.g., sensor X is at room Y) happens at the gateway/cache node

Making Small Implementions: Problem 5 – Checksum

Checksum code is bloat

```
u16 udp_sum_calc(u16 len_udp, u16 src_addr[],u16 dest_addr[], BOOL padding, u16 buff[])
{
u16 prot_udp=17;
u16 padd=0;
u16 word16;
u32 sum;

        // Find out if the length of data is even or odd number. If odd,
        // add a padding byte = 0 at the end of packet
        if (padding&1==1){
                padd=1;
                buff[len_udp]=0;
        }

        //initialize sum to zero
        sum=0;

        // make 16 bit words out of every two adjacent 8 bit words and
        // calculate the sum of all 16 vit words
        for (i=0;i<len_udp+padd;i=i+2){
                word16 =((buff[i]<<8)&0xFF00)+(buff[i+1]&0xFF);
                sum = sum + (unsigned long)word16;
        }
        // add the UDP pseudo header which contains the IP source and destinationn addresses
        for (i=0;i<4;i=i+2){
                word16 =((src_addr[i]<<8)&0xFF00)+(src_addr[i+1]&0xFF);
                sum=sum+word16;
        }
        for (i=0;i<4;i=i+2){
                word16 =((dest_addr[i]<<8)&0xFF00)+(dest_addr[i+1]&0xFF);
                sum=sum+word16;
        }
        // the protocol number and the length of the UDP packet
        sum = sum + prot_udp + len_udp;

        // keep only the last 16 bits of the 32 bit calculated sum and add the carries
        while (sum>>16)
                sum = (sum & 0xFFFF)+(sum >> 16);

        // Take the one's complement of sum
        sum = ~sum;

return ((u16) sum);
}
```

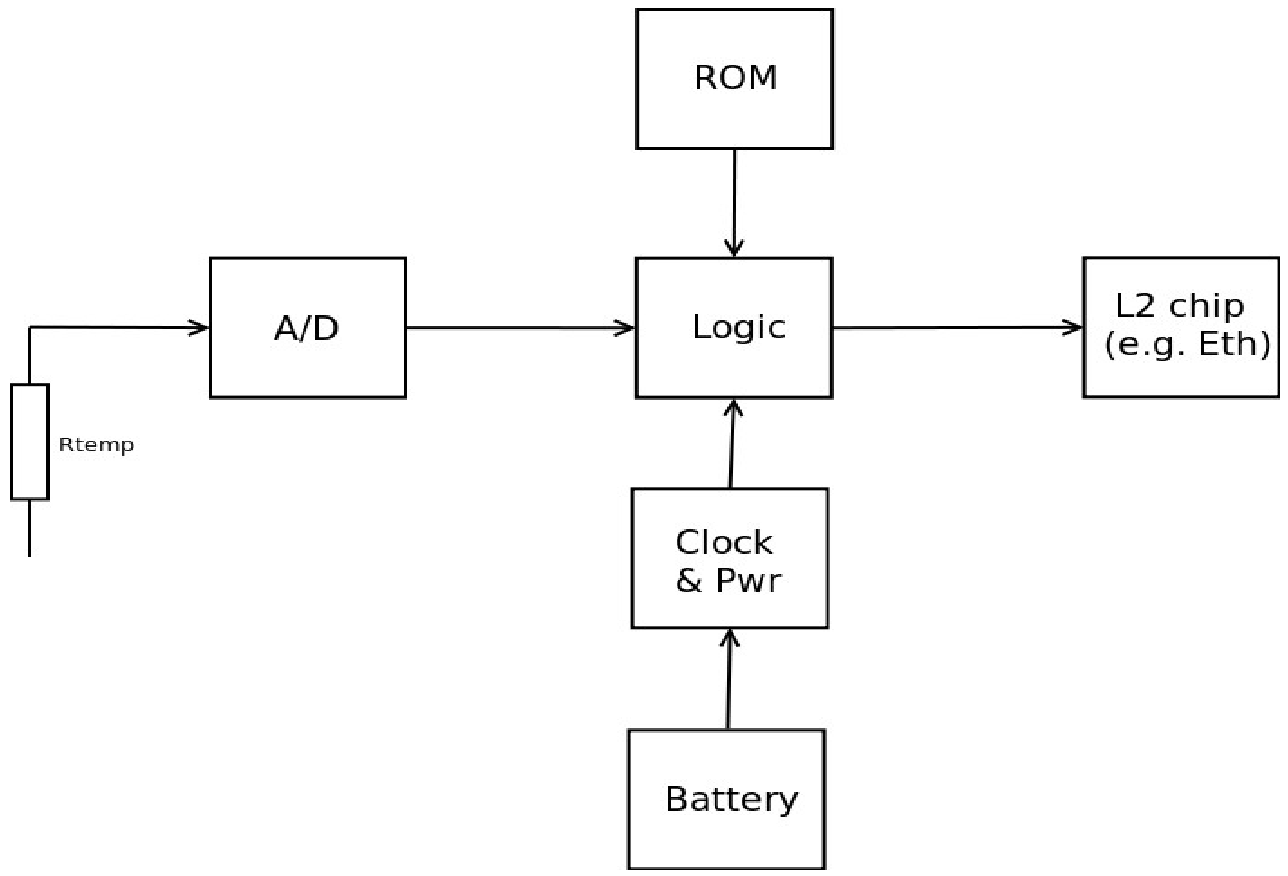
Fortunately 1s complement checksums are commutative and transitive

Change a word from 0 to x and you only need to recalculate:

sum = ~(x+~sum);

We can use precomputation + recalculation

Draft Schema for HW Implementation



Reflections on COAP

Some detailed issues discussed in the draft & CORE WG

But there are also fundamental concerns

- The lightweight nature of COAP is more about small changes to syntax and behavior (TCP=>UDP) than about eliminating reasons behind complexity and power usage
- Communication models are the key here
- COAP can (perhaps) be used in sleepy nodes, but it requires great care
- COAP observer spec to be revised

Communication Models: 1. Send-Only

```
      User or                    Sensor
   Intermediary                 (Client)
     (Server)                      .
        |                          .
        |                          .
        |                       wake-up
        |                          |
        |      NON/POST            |
        |      content             |
        +<-------------------------|
        |                          |
        |                     power-down
        |                          .
        |      NON/RSP             .
        |----------------/         .
        |                          .
```

Communication Modes: 2. Server

```
  User or                    Sensor
Intermediary               (Server)
  (Client)
     |                          |
     |      CON/GET             |
     +------------------------->|
     |                          |
     |      ACK/RSP             |
     |      content             |
     |<-------------------------+
     |                          |
```

More About the Implementation

48 lines:

- 25% initialization of A/D converter
- 25% binary-to-decimal conversion
- 25% checksum calculation
- 25% other

Also requires a 160 byte message template

- Pre-filled and pre-computed as far as possible
- Needs to be copied from ROM to RAM
- No RAM necessary across invokations

Other

- Assumes a real-time clock for COAP message IDs

Implementation Techniques

- Selecting the right communication model
- Employing the freedom that the protocol allows for the sender to choose optional protocol features and behaviours
- Selecting a single stack (IPv6)
- Building only the necessary stack components around a fixed application
- Monolithic implementation (not layered)
  - For instance, the message template has everything from Ethernet header to XML
- Message templates

Observations About the Implementation Techniques

- The same implementation technique would have worked for JSON, XML, binary formats
- Binary format would have saved ~10 instructions, ~40% message length
- Compressed formats would be extremely complex
- Negotiation would have negated any simplifications
- Logic-based implementation would be feasible, but decimal formats make it too complex (hex OK though)

Final Piece of Advice

We are protocol engineers and like to tinker with protocol designs, lighter-weight versions of protocols, enhancements that improve efficiency

Lets resist that temptation!

Better implementations are often the answer

ERICSSON