Connecting Wireless Sensor Networks to the Internet - a 6lowpan Implementation for TinyOS 2.0

Matúš Harvan

Jacobs University Bremen
Bremen, Germany

Universität Bremen, 25 May 2007

Outline

Wireless Sensor Networks

Definition

A wireless sensor network (WSN) is a **wireless network** consisting of **spatially distributed autonomous devices** using **sensors** to **cooperatively monitor** physical or environmental conditions, such as temperature, sound, vibration, pressure, motion or pollutants.

- Small computers with a wireless interface
- Smart alternatives to dumb RFID tags

Wireless Sensor Networks

Definition

A wireless sensor network (WSN) is a **wireless network** consisting of **spatially distributed autonomous devices** using **sensors** to **cooperatively monitor** physical or environmental conditions, such as temperature, sound, vibration, pressure, motion or pollutants.

- Small computers with a wireless interface
- Smart alternatives to dumb RFID tags

Outline

1. Introduction
   - TelosB and MicaZ Hardware Platforms
   - IEEE 802.15.4 (PHY and MAC layer)
   - TinyOS and nesC
   - 6lowpan
2. Implementation
   - Buffers
   - Fragments
   - 6lowpan for Linux
   - Missing Features
3. Demonstration
4. Conclusion
   - Evaluation

TelosB Hardware Platform

TI MSP430 MCU

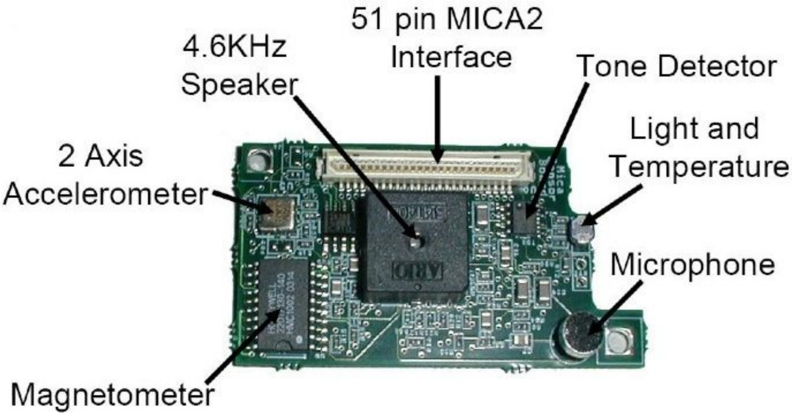
- 16 bit RISC at 8 MHz
- 16 registers
- 10kB RAM
- 48kB Flash
- 16kB EEPROM

MicaZ Hardware Platforms

Atmel AVR ATmega 128

- 8 bit RISC
- 32 registers
- 4kB RAM
- 128kB Flash
- 4kB EEPROM

Mica Sensor Board MTS310



Outline

1. Introduction
   - TelosB and MicaZ Hardware Platforms
   - IEEE 802.15.4 (PHY and MAC layer)
   - TinyOS and nesC
   - 6lowpan
2. Implementation
   - Buffers
   - Fragments
   - 6lowpan for Linux
   - Missing Features
3. Demonstration
4. Conclusion
   - Evaluation

Radio – IEEE 802.15.4

IEEE 802.15.4

- 250 kbps (16 channels, 2.4 GHz ISM band)
- personal area networks (10 meters range)
- PHY and MAC layer covered
- Link encryption (AES) (no key management)
- Full / Reduced function devices

ChipCon CC2420

- popular 802.15.4 air interface
- 128byte TX/RX buffer
- used on the TelosB and MicaZ motes

Radio – IEEE 802.15.4

IEEE 802.15.4

- 250 kbps (16 channels, 2.4 GHz ISM band)
- personal area networks (10 meters range)
- PHY and MAC layer covered
- Link encryption (AES) (no key management)
- Full / Reduced function devices

ChipCon CC2420

- popular 802.15.4 air interface
- 128byte TX/RX buffer
- used on the TelosB and MicaZ motes

Outline

1. Introduction
   - TelosB and MicaZ Hardware Platforms
   - IEEE 802.15.4 (PHY and MAC layer)
   - TinyOS and nesC
   - 6lowpan
2. Implementation
   - Buffers
   - Fragments
   - 6lowpan for Linux
   - Missing Features
3. Demonstration
4. Conclusion
   - Evaluation

TinyOS

- embedded operating system for WSN motes
- written in the *nesC* language
- event-driven architecture
- no kernel/user space differentiation
- single shared stack
- static memory allocation only (no malloc/free)
- no process or memory management
- components statically linked together

nesC: Programming Language for Embedded Systems

- Programming language:
  - a dialect/extension of C
  - static memory allocation only (no malloc/free)
  - whole-program analysis, efficient optimization
  - race condition detection
- Implementation:
  - pre-processor – output is a C-program, that is compiled using gcc for the specific platform
  - statically linking functions
- For more details, see [3]

Outline

1. Introduction
   - TelosB and MicaZ Hardware Platforms
   - IEEE 802.15.4 (PHY and MAC layer)
   - TinyOS and nesC
   - 6lowpan
2. Implementation
   - Buffers
   - Fragments
   - 6lowpan for Linux
   - Missing Features
3. Demonstration
4. Conclusion
   - Evaluation

6lowpan – IPv6 over 802.15.4

- IETF working group (IPv6 over low-power wireless personal area networks)
- 6lowpan header/dispatch value before the IP header

layer 2 header (802.15.4)
optional Mesh Addressing Header (6lowpan)
optional Broadcast Header (6lowpan)
optional Fragmentation Header (6lowpan)
IPv6 header (6lowpan-compressed)
layer 4 header (i.e. 6lowpan compressed UDP header)
layer 4/application payload

Table: 802.15.4 frame with 6lowpan payload

6lowpan – Details

- header compression
  - IPv6 and UDP headers can ideally be compressed from $40 + 8$ to $2 + 4$ bytes
  - no prior communication for context estabilishment necessary
- fragmentation below the IP layer
  - IPv6 requires a minimum MTU of 1280 bytes, but 802.15.4 can at best provide 102 bytes
  - Fragmentation Header
- mesh networking support
  - Mesh Addressing Header and Broadcast Header
  - routing algorithms and further details out of scope of the 6lowpan working group

Outline

Design principles

- run on the TelosB and MicaZ motes
  - fit into 4KB of RAM
- easily readable and maintainable code preferred over optimizing to squeeze into the least possible amount of memory

Modules and interfaces

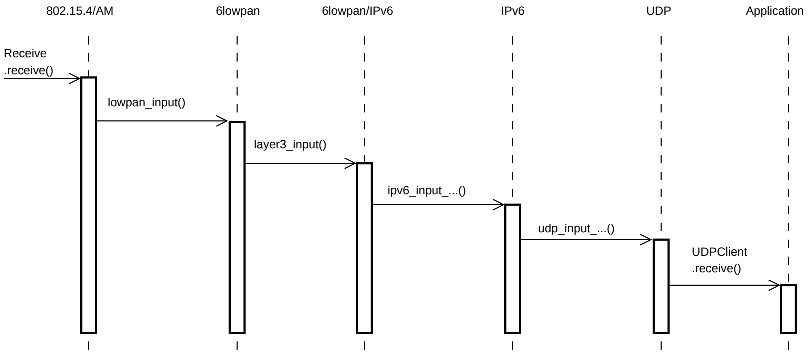
- `IPC.nc` – configuration, used by the appliciation
- `IPP.nc` – module with the implementation
- `UDPClient.nc` – interface used by the application
- `IP.h` – included by the application
- `IP_internal.h` – used only by `IPC` and `IPP`

UDPClient interface

UDPClient

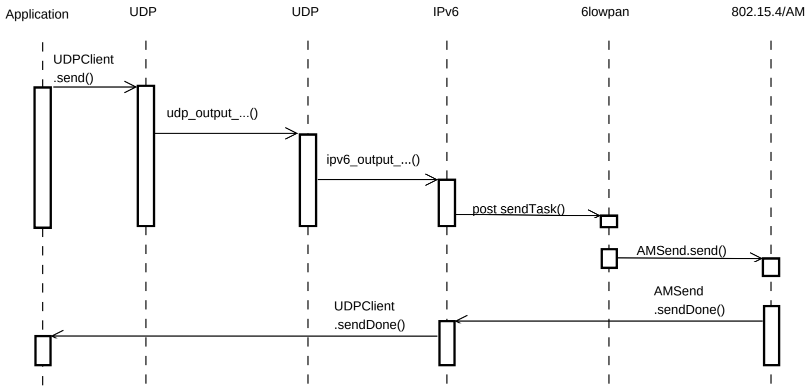
```
interface UDPClient {
    command error_t listen( uint16_t port );

    command error_t connect(const ip6_addr_t *addr, const uint16_t port);

    command error_t sendTo(const ip6_addr_t *addr, uint16_t port,
                           const uint8_t *buf, uint16_t len);
    command error_t send(const uint8_t *buf, uint16_t len);
    event   void    sendDone(error_t result, void* buf);

    event   void    receive(const ip6_addr_t *addr, uint16_t port,
                            uint8_t *buf, uint16_t len);
}
```

Receiving a UDP packet



- each network layer and protocol handled by a separate function

Sending a UDP packet



- again separate functions
- task for sending packets
- queue of outgoing packets

Sending – Why a task and queuing?

- may have to determine destination's link-layer address (Neighbor Discovery)
  - before sending the packet
    – need to know where to send the packet
  - before HC1-encoding the IPv6 header
- fragmentation may be needed
- receive frames while fragments are being sent

Outline

Buffers – Design goals

- accommodate for both
  - short unfragmented packet – up to 102 bytes
  - large fragmented packet – up to 1280 bytes
- higher-layer (UDP) payload buffer provided by the application
- need to prepend headers

Buffers – Design

- all headers together are certainly not larger than an unfragmented packet
- use the same buffer for headers as for the unfragmented packet payload

size	header
	6lowpan optional headers
5 — 19	mesh addressing
2	broadcast
4 — 5	fragmentation
	layer 3 header
41	IPv6 (uncompressed)
3 — 41	IPv6 (HC1-compressed)
	layer 4 headers
8	UDP (uncompressed)
4 — 9	UDP (HC_UDP-compressed)
8	ICMP
24	TCP

Buffers - representing a packet

lowpan_pkt_t

```
typedef struct _lowpan_pkt_t {
    /* buffers */
    uint8_t *app_data;        /* buffer for application data */
    uint16_t  app_data_len;   /* how much data is in the buffer */
    uint8_t *app_data_begin;  /* start of the data in the buffer */
    uint8_t app_data_dealloc; /* shall IPC deallocate the app_data buffer? */

    uint8_t header[LINK_DATA_MTU]; /* buffer for the header (tx)
                                    * or unfragmented 802.15.4 frame (rx) */
    uint16_t  header_len;     /* how much data is in the buffer */
    uint8_t *header_begin;    /* start of the data in the buffer */

    /* fragmentation (tx) */
    uint16_t dgram_tag;
    uint16_t dgram_size;
    uint8_t dgram_offset;     /* offset where next fragment starts (tx) */

    /* IP addresses */
    ip6_addr_t ip_src_addr;
    ip6_addr_t ip_dst_addr;

    /* 802.15.4 addresses */
    hw_addr_t hw_src_addr;
    hw_addr_t hw_dst_addr;

    uint8_t notify_num;       /* num of UDPClient + 1
                               * 0 for no sendDone notification */
    struct _lowpan_pkt_t *next;
} lowpan_pkt_t;
```

Buffers – changing the owner

- `app_data_dealloc` allows to change the "owner" of the `app_data` buffer
- used when replying to an ICMP echo request to prevent copying of data
- might be useful for `UDPClient` as well

Buffers - receiving a packet

- one global buffer for received packet
- unfragmented packets fit into `header` buffer
- fragmented packets in `app_data` buffer (provided by fragment reassembly)
- no concurrency – processing of the received packet cannot be interrupted by receiving another packet
  (until control is returned back to TinyOS)

Buffers - sending a packet

- `SendPktPool`
  - pool of `lowpan_pkt_t` packets for sending
  - compile-time configurable size – allows to make use of extra memory on the TelosB mote
- outgoing packets queued, queue processed by `sendTask`

Outline

Fragmentation

- dealt with in `sendTask`

Fragment reassembly

- reassembled into an `app_data` buffer
- pool of `app_data` buffers – `AppDataPool`
  - compile-time configurable size
  - size determines how many packets can be reassembled concurrently
- when reassembly completed, `app_data` buffer moved into the global `lowpan_pkt_t` for receiving

Fragment reassembly

- two options for keeping track of received fragments
  1. bitmap – 160 bits = 20 bytes would do
  2. linked list
- 6lowpan draft requires treating overlapping fragments differently if offset or length differ
- need a linked list to determine if offset or length differ or the fragment is just a duplicate
- linked list items also managed by a pool
- if full size of 802.15.4 frames is used, 15 fragments are sufficient for a 1280-byte packet

Link-layer

- TinyOS 2.0 does not have a proper 802.15.4 stack
- TinyOS notion of networking: *Active Messages*
- Active Message header same as 802.15.4 data frame header
    - additional 1-byte `AM Type` field in the 802.15.4 payload

```
+-------------------+---------+-----------------------------+--------------+
| 802.15.4 Header   | AM type |            data             | 802.15.4 CRC |
+-------------------+---------+-----------------------------+--------------+
```

- solution: sending 6lowpan payload as Active Message payload

IPv6 Addresses

- one global IPv6 address (prefix currently hardcoded)
- one link-local IPv6 address
- interface identifier computed from Active Message address (802.15.4 short address) of the mote
- the application cannot change the IPv6 addresses (for now)

Outline

serial_tun daemon – 6lowpan for Linux

- allows a Linux PC to use a mote as an 802.15.4 interface
  - the mote runs the BaseStationCC2420 application to forward frames between the USB and the radio interface
- 6lowpan en- and decapsulation
  - the Linux kernel does not speak 6lowpan
- tun interface
  - proper network interface, `ifconfig`-supported
  - packets further handled by the Linux kernel

Outline

Missing features

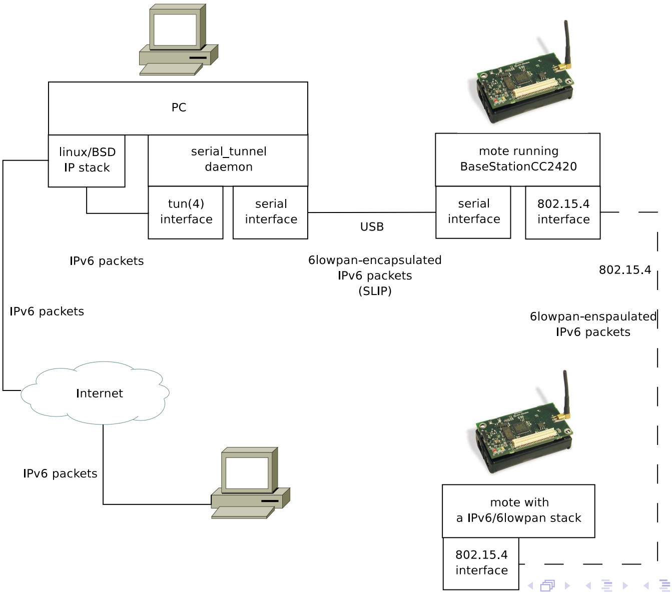
- proper 802.15.4 stack – tunneling as Active Message payload
- HC1 encoding – non-zero Traffic Class and Flow Label
- HC_UDP encoding – compressed UDP port numbers
- fragmentation PC $\longrightarrow$ mote not yet 100% reliable
  - some fragments not received by the mote
  - workaround with sleeping before sending a subsequent fragment

Missing features

- Neighbor Discovery
  - using link-layer broadcast instead
  - unclear which parts actually needed, under discussion in 6lowpan and RSN groups
- IPv6 extension header, IPv6 fragmentation
- sending ICMP error message

Outline

6lowpan demonstration



6lowpan – Demonstration

- ping – IPv6
    - unfragmented
    - fragmented
- cli – telnet over IPv6/UDP
    - control the leds
    - control the sounder
    - request UDP data to be sent back
        - unfragmented
        - fragmented

Outline

Evaluation

- 21900 bytes of ROM, 2906 bytes of RAM
- ping works
- UDP works
- fragmentation works
- tested against Linux `ping6` and `nc6`
- robust – ping can run for several hours
- the design allows to easily add
  - replying to Neighbor Solicitations
  - sending Neighbor Solicitations before HC1-encoding or sending a packet
  - TCP protocol

References

K. Römer and F. Mattern.
The Design Space of Wireless Sensor Networks
IEEE Wireless Communications 11(6), December 2004.

J. Polastre, R. Szewczyk and David Culler.
Telos: Enabling Ultra-Low Power Wireless Research
IEEE IPSN, April 2005.

D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer and D. Culler.
The nesC Language: A Holistic Approach to Networked Embedded Systems
ACM PLDI, June 2003.

G. Montenegro, N. Kushalnagar, J. Hui and D. Culler.
Transmission of IPv6 Packets over IEEE 802.14.4 Networks
Internet-Draft draft-ietf-6lowpan-format-13 (work in progress), April 2007.

Questions?

Implementation: http://www.eecs.iu-bremen.de/users/harvan/files/6lowpan.tar.gz

Backup slides

Applications

- Environmental monitoring
- Seismic detection
- Disaster situation monitoring and recovery
- Health and medical monitoring
- Inventory tracking and logistics
- Smart spaces (home/office scenarios)
- Military surveillance

Why connect WSNs to the Internet?

- Internet Protocol (IP)
  - ubiquitous
  - de-facto standard
  - already deployed
  - plethora of applications available

Design Goals

- cheap
  - ideally less than 1 Euro
- many
  - lots of devices, economies of scale
- robust
  - unattended operation (no repair)
- small
  - importance depends on the circumstances
- low-power
  - difficult/impossible to replace batteries

Design Goals

- cheap
  - ideally less than 1 Euro
- many
  - lots of devices, economies of scale
- robust
  - unattended operation (no repair)
- small
  - importance depends on the circumstances
- low-power
  - difficult/impossible to replace batteries

Design Goals

- cheap
  - ideally less than 1 Euro
- many
  - lots of devices, economies of scale
- robust
  - unattended operation (no repair)
- small
  - importance depends on the circumstances
- low-power
  - difficult/impossible to replace batteries

Design Goals

- cheap
  - ideally less than 1 Euro
- many
  - lots of devices, economies of scale
- robust
  - unattended operation (no repair)
- small
  - importance depends on the circumstances
- low-power
  - difficult/impossible to replace batteries

Design Goals

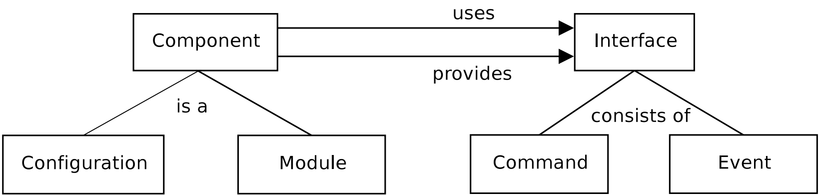
- cheap
  - ideally less than 1 Euro
- many
  - lots of devices, economies of scale
- robust
  - unattended operation (no repair)
- small
  - importance depends on the circumstances
- low-power
  - difficult/impossible to replace batteries

TinyOS – Functionality

- hardware abstraction
- access to sensors
- access to actuators
- scheduler (tasks, hardware interrupts)
- timer
- radio interface
- Active Messages (networking)
- storage (using flash memory on the motes)
- ...

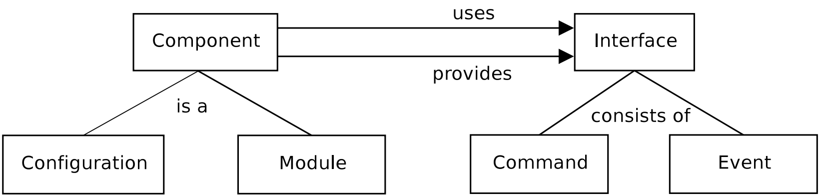
nesC – Components and Interfaces

- a nesC application consists of components
  - *modules* – implement interfaces
  - *configurations* – connect modules together via their interfaces (*wiring*)
- components provide and use *interfaces*
  - *commands* – can be called by other modules
  - *events* – signaled by other modules



nesC – Components and Interfaces

- a nesC application consists of *components*
  - *modules* – implement interfaces
  - *configurations* – connect modules together via their interfaces (*wiring*)
- components provide and use *interfaces*
  - *commands* – can be called by other modules
  - *events* – signaled by other modules



NesC — Concurrency — Tasks

Define a Task

```
task void task_name() { ... }
```

Post a Task

```
post task_name();
```

- posting a task – the task is placed on an internal task queue which is processed in FIFO order
- a task runs to completion before the next task is run, i.e. tasks do not preempt each other
- tasks can be preempted by hardware events