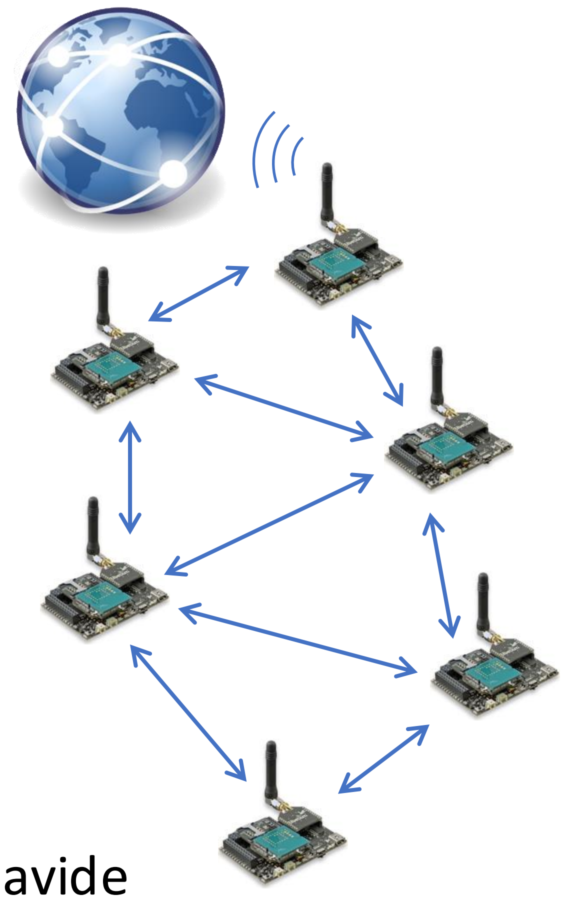
Low-power Wireless Networking for the Internet of Things



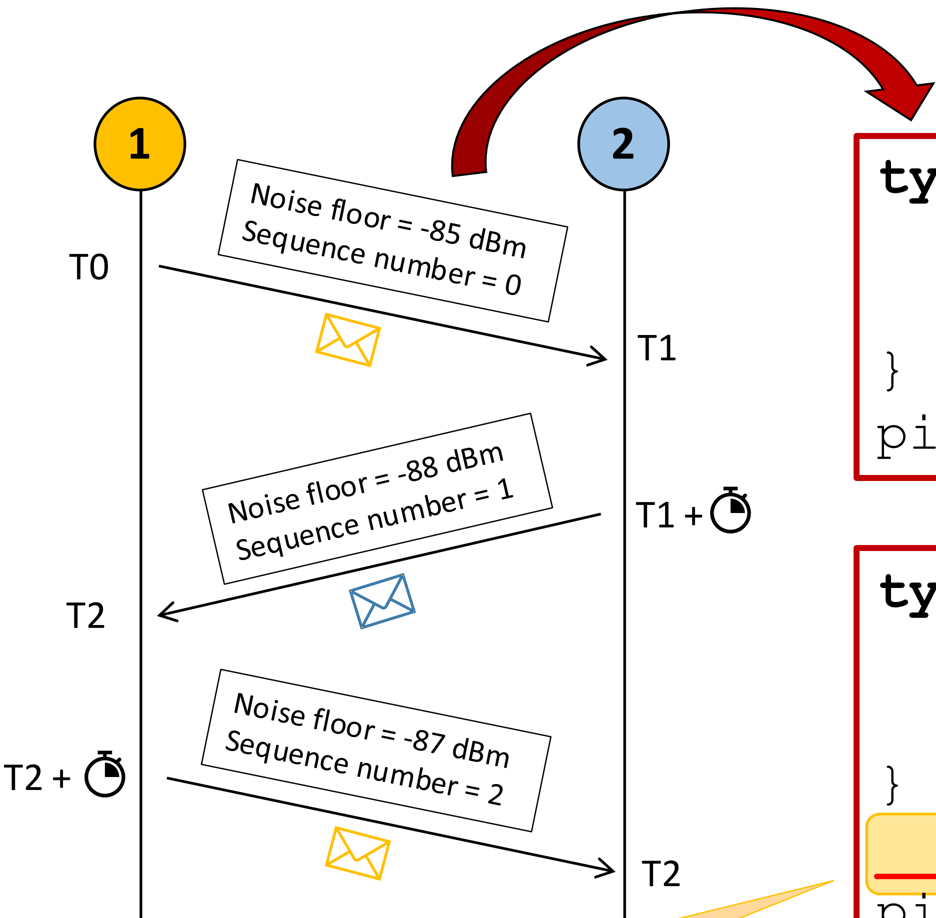
Lab4: Data serialization for radio communication

Matteo Trobinger (matteo.trobinger@unitn.it)

Credits for some slides to:

Timofei Istomin, Pablo Corbalán, Enrico Soprana, Davide Vecchia

Struct type definition for the ping-pong message




```
typedef struct ping_pong_msg {
  uint16_t sequence_number;
  int16_t noise_floor;
}
ping_pong_msg_t;
```

```
typedef struct ping_pong_msg {
  uint16_t sequence_number;
  int16_t noise_floor;
}
__attribute__((packed))
ping_pong_msg_t;
```

What's that? **A hack to avoid implementing a serialization framework**

Serialization: the process of converting an object into a format that can be readily transported

Problems of C structures

There are three problems of C structures preventing their direct use in network messages:

1. **Byte order —** There are big-endian and little-endian platforms
2. **Unaligned access —** CPU can directly operate with a multi-byte number *only* if it is placed at the word boundary in memory
3. **Structure padding —** Compiler inserts holes between structure fields to optimize access. We don’t want to send holes (and they are architecture-dependent)

Byte order

Least significant byte (LSB)

Let's take number 0x89ABCDEF (hexadecimal).

In memory the number might be stored differently:

addr+0	EF
addr+1	CD
addr+2	AB
addr+3	89

LSB first

Little-endian
(e.g. x86)

MSB first

Big-endian
(some ARMs)

addr+0	89
addr+1	AB
addr+2	CD
addr+3	EF

In a network, computers have to agree!

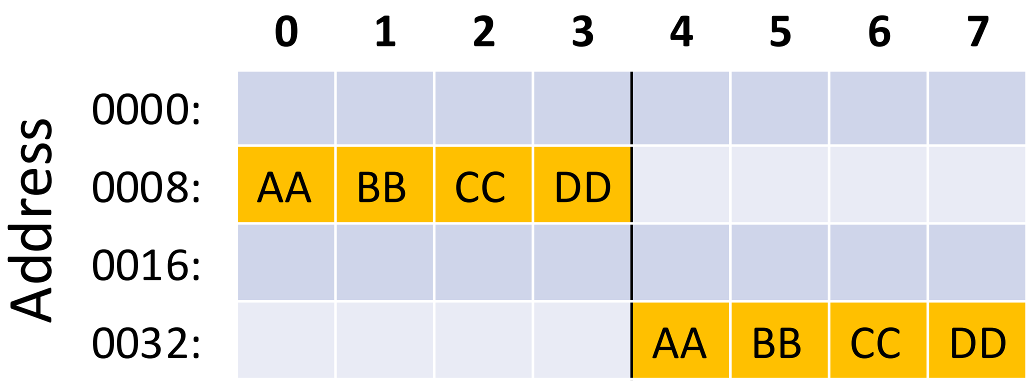
- Numbers must be converted into **network byte order**, which is **big endian** [RFC 1700].

Otherwise, e.g., a big-endian sensor says the temperature is 100°C and an x86 server thinks it's 1 677 721 600°C.

Memory alignment

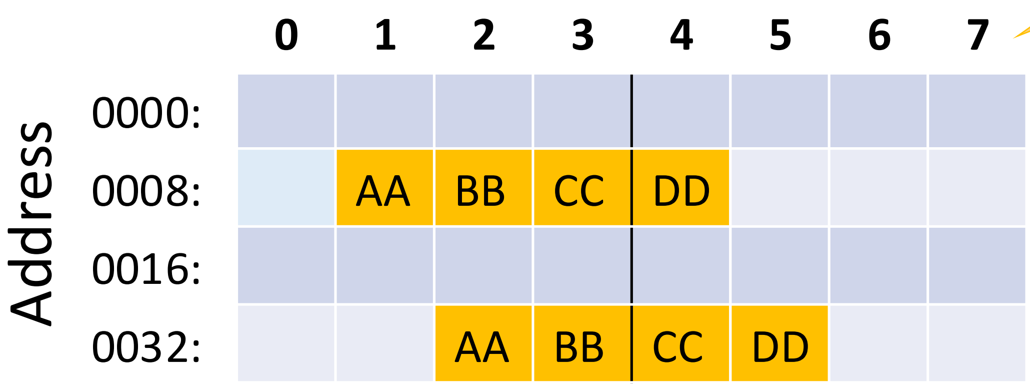
Most CPUs can work with multi-byte numbers *only* when they are placed at the *word boundary*

- 32-bit CPUs have words of 4 bytes



Properly aligned values 0xDDCCBBAA

Examples of unaligned values (errors)



Compiler *always* aligns variables properly, but <u>if you receive a number in a payload, it can be shifted arbitrarily</u>

Unaligned access

```
uint16_t val;
memcpy(&val, packetbuf_dataptr(), 2);
printf(“%d”, val);
```

OK!

```
uint16_t* val = (uint16_t*)packetbuf_dataptr();
printf(“%d”, *val);
```

Error!

- **When you receive anything, copy it first to a variable, then use!**
- **To send a value, copy it to the payload from a variable!**

Structure padding

Because of the memory alignment requirement, the compiler sometimes has to insert holes in structures

```
struct {
  uint8_t  a;
  uint32_t b;
  uint16_t c;
}
```

Size 7?

32-bit CPU

```
struct {
  uint8_t  a;
  // 3-byte hole
  uint32_t b;
  uint16_t c;
  // 2-byte hole
}
```

Size 12

Packed structs

There's a way to tell the compiler to disable padding for a structure

```
struct {
  uint8_t  a;
  // 3-byte hole
  uint32_t b;
  uint16_t c;
  // 2-byte hole
}
__attribute__((packed))
```

Size 7

Is it enough? It might be, if you care more about *simplicity* rather than *efficiency* and *portability*.

Why packed structs are not that good

- **They do not solve the endianness problem**
  - It's the programmer's job to take care of it

- **The code that the compiler generates to work with packed structs is less efficient**
  - Because the struct members are not properly aligned

Packed structs are inefficient

```
struct foo {int a;} x;
```

```
x.a=0x1234;
```

Compiled to

```
mov   #0x1234, @x
```

```
struct bar{int a;}
__attribute__((packed)) y;
```

```
y.a=0x1234;
```

Compiled to

```
mov    #y, r15
mov.b #0x34, @r15
mov.b #0x12, 1(r15)
```

Serialization (marshaling)

The correct way: for *every structure* to be sent over the network, define 2 functions

1. To *serialize* the structure to a byte buffer
2. To *deserialize* a byte buffer into the structure

Example: assume you need to send/receive a message containing the following 3 fields:

- **a**: 1 byte, **b**: 2 bytes, **c**: 4 bytes

First thing to do: define (*document*) the message format, e.g.:

<table>
<tr><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th></tr>
<tr><td>a</td><td colspan="2">b</td><td colspan="4">c</td></tr>
</table>

<u>Specify also the network byte order</u>! We'll use **big-endian** here (and our platform might be little-endian)

Serialization

0	1	2	3	4	5	6
a	b		c			

Values to TX:
a = 0x01
b = 0x2345
c = 0x6789ABCD

```
//pl is a pointer to the
//payload portion of the
//packetbuf (uint8_t*)

pl[0] = a;

pl[1] = b >> 8;
pl[2] = b;

pl[3] = c >> 24;
pl[4] = c >> 16;
pl[5] = c >> 8;
pl[6] = c;
```

pl[.] is a 8-bit variable!

//**pl** is a pointer to the
//payload portion of the
//packetbuf (uint8_t*)

pl[0] = 0x**01**;

pl[1] = ~~0x00~~**23**;
pl[2] = ~~0x23~~**45**;

pl[3] = ~~0x000000~~**67**;
pl[4] = ~~0x000067~~**89**;
pl[5] = ~~0x006789~~**AB**;
pl[6] = ~~0x6789AB~~**CD**;

0	1	2	3	4	5	6
01	23	45	67	89	AB	CD

Deserialization

0	1	2	3	4	5	6
01	23	45	67	89	AB	CD

```
//pl is a pointer to the
//received payload
//(uint8_t*)

a = pl[0];

b  = (uint16_t)pl[1] << 8;
b |= (uint16_t)pl[2];

c  = (uint32_t)pl[3] << 24;
c |= (uint32_t)pl[4] << 16;
c |= (uint32_t)pl[5] << 8;
c |= (uint32_t)pl[6];
```

```
//pl is a pointer to the
//received payload
//(uint8_t*)

a = 01;

b  = 0x2300;
b |= 0x0045;

c  = 0x67000000;
c |= 0x00890000;
c |= 0x0000AB00;
c |= 0x000000CD;
```

After bitwise OR
b = 0x2345

After *all*
bitwise OR
c = 0x6789ABCD

Serialization (marshaling)

Instead of hard-coding indexes, it is convenient to use a **progressing counter** (`idx`)

Serialize

```
uint16_t idx = 0;

pl[idx++] = a;

pl[idx++] = b >> 8;
pl[idx++] = b;

pl[idx++] = c >> 24;
pl[idx++] = c >> 16;
pl[idx++] = c >> 8;
pl[idx++] = c;
```

Deserialize

```
uint16_t idx = 0;

a = pl[idx++];

b  = (uint16_t)pl[idx++] << 8;
b |= (uint16_t)pl[idx++];

c  = (uint32_t)pl[idx++] << 24;
c |= (uint32_t)pl[idx++] << 16;
c |= (uint32_t)pl[idx++] << 8;
c |= (uint32_t)pl[idx++];
```

Alert: Request access to CLOVES!

How to get access:

Fill in the form on Moodle **<u>with your unitn.it account</u>** to obtain your testbed credentials.

Deadline:

Registration **must be completed** by **October 26th at 23:59**

Code Templates

Download and unzip the provided code

- `$ unzip Lab4-exercise.zip`

Go to the code directory

- `$ cd Lab4-exercise/chain-template`

To compile:

- `$ make` or `make TARGET=SKY`

When everything seems to work, test it in Cooja:

- `$ cooja chain.csc &`

Discover who's around, forward messages to neighbors!

Neighbor discovery:

- Periodically send beacons (*in broadcast*) to announce the node's presence
- Upon receiving a beacon, store the sender address in a neighbor table

Message forwarding:

- On button press, send (*in unicast*) a packet to a random neighbor (the packet format is specified in the next slide!)
- When a unicast packet arrives, increment the hop count and forward the packet to a random neighbor

The template implements everything except for message (de)serialization

Finished? Try to improve the current protocol!
If you want, discuss your ideas/solution with me ☺

Unicast message format

At the node, increment this value **by one** at every button press!!!

octet	**0**	**1**	**2**	**3**	**4**	**…**
field	**seqn**		**hops**	**string**		
size	2 octets		1 octet	variable length		

Network byte order: big-endian