Handout: Using Meta-Code for Building Task-Specific WSNs

Igor Talzi [1], Christian Tschudin [2]
Computer Science Dept, University of Basel
CH-4056 Basel, Switzerland
[1]Igor.Talzi@unibas.ch
[2]Christian.Tschudin@unibas.ch

In this handout we give some idea of how meta-code can be used to build task-specific WSN configurations. All examples are written in Meta-Lang, the assembler-like language of meta-code. We implement a simple data gathering application with the following set of features: spanning-tree based, unique id assigned for each node, 1-min[1] measurements are collected, delivered to the top of the spanning tree and stored into the buffer. Hence, we cover the following layers of the WSN network stack: routing, data and application processing. In this example we assume that MAC-layer is provided. Time synchronization (e.g. LTS spanning-tree based time-sync algorithm) can be easily added with minimal changes needed to the presented capsules.

The first capsule we use to build a classic spanning tree (see Listing 1). This capsule must periodically (10s) flood the network and be executed on each node to reflect the changes in the tree structure.

```
.sys                    # SYSTEM segment
    AUTOUPDATE 0        #   disable autoupdate (capsules of the same version
                        #   will be accepted, capsules of lower versions
                        #   will be declined)
    LIFETIME 10s        #   recognized post-fixes: ms (millisec), s (sec),
                        #   p (packets)
    ID 0x11             #   4-bit ID + 4-bit version number
.bufc                   # DATA segment (allocated inside the capsule)
    from=S              #   "S" is some real network address
    hops=0              #   local variables
.code.init              # CODE segment "init" (executed once)
    inc hops
    push BUFS[0]        #   first we check the ID
    jmpeq ME.ID,l1      #   "ME.*" - this capsule, "CAP.*" - capsule,
                        #   "PACK.*" - packet
    mov BUFS[0],ME.ID   #   store ID and "hops" in the shared memory BUFS
    mov BUFS[1],hops    #   (allocated from the node's memory pool)
    jmp l2

l1: push BUFS[1]        #   check the distance
    jmplet hops,l3
    replace             #   replace the existing capsule

l2: send ME,ALL         #   broadcast itself
    mov from,ME.FROM
    exit

l3: die                 #   if none - kill the capsule
.code.pack              # CODE segment "receive packet" (executed upon
                        # receiving a packet)
    push PACK.DST       #   "PACK.SRC" and "PACK.DST" are fixed, "PACK.FROM"
                        #   and "PACK.TO" change at each hop
    jmpeq S,l4          #   process packets addressed to S

    exit                #   exit point (the capsule stays alive)

l4: send PACK,from      #   send a packet up the spanning tree
    exit
```

Listing 1: Spanning tree construction

[1]This is just a sensing interval; no time-sync is used; measurements are not timestamped.

IPSN'10, April 12–16, 2010, Stockholm, Sweden.
ACM 978-1-60558-955-8/10/04.

The second capsule performs automatic node id assignment based on a measured temperature value (can be humidity, or any other available 16-bit sensor, or a mix) and a simple pseudo-random number generator shown in Listing 2 below:

```
m_w = <choose-initializer>;    /* must not be zero */
m_z = <choose-initializer>;    /* must not be zero */

uint get_random()
{
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;  /* 32-bit result */
}
```

Listing 2: "Multiply-With-Carry" random number generator of G. Marsaglia

This capsule is executed once during the initialization phase on each node (see Listing 3).

```
1  .sys                    # SYSTEM segment
2      AUTOUPDATE 1
3      LIFETIME 10s
4      ID 0x21
5  .code.init              # CODE segment "init"
6      send ME,ALL         #   broadcast itself
7      sense_temp          #   choose initializer for A
8      pop BUFS[0]         #   A = 18000 * (A & 65535) + (A >> 16)
9      push BUFS[0]
10     and 65535
11     mult 18000
12     push BUFS[0]
13     rsh 16
14     add
15     sense_temp          #   choose initializer for B
16     pop BUFS[0]         #   B = 36969 * (B & 65535) + (B >> 16)
17     push BUFS[0]
18     and 65535
19     mult 36969
20     push BUFS[0]
21     rsh 16
22     add
23     lsh 16              #   (B << 16) + A (32-bit result)
24     add
25     die TOP             #   clean up the top part
26     pop BUFC[0]         #   store the new ID in the capsule
27     send ME,S           #   send it up the spanning tree
```

Listing 3: Automatic node id-assignment

The following example shows how program size can be reduced by making calls to macro-instructions (this is supposed to replace a part of the original code on lines 7-24 in Listing 3):

```
    mov BUFS[1],18000 # compute A
    comp_coef
    mov BUFS[1],36969 # compute B
    comp_coef
    lsh 16            # (B << 16) + A (32-bit result)
    add
```

The macro-instruction "`comp_coef`" shown above is stored in the on-board instruction dictionary and defines the following sequence of simpler operations:

```
    sense_temp        # choose initializer for A (or B)
```

```
2       pop BUFS[0]        # A = 18000 * (A & 65535) + (A >> 16) or
3       push BUFS[0] 16    # B = 36969 * (B & 65535) + (B >> 16)
4       rsh
5       push BUFS[0] BUFS[1] 65535
6       and
7       mult
8       add
```

In the code above lines 3 and 5 show that macro-instructions can be nested.

The next two capsules are responsible for taking and collecting measurements. The algorithm assumes that we already have an established tree topology in the network. The capsule periodically initiates a measurement on each node, accumulates a buffer of 10 measurements and sends it back up to the top of the spanning tree (see Listing 4). The sink node does not execute this capsule.

```
1  .sys                    # SYSTEM segment
2      AUTOUPDATE 1
3      LIFETIME 0          #   live forever
4      ID 0x31
5  .code.init
6      push 10
7  .code.cap               # CODE segment "receive capsule"
8      dec
9  l1: ifeq 0,l2           #   check the counter
10     sense               #   "sense" calls "sense_temp" on this node
11                         #   which measures temperature
12     pop BUFC            #   append to BUFC
13     delay 60s           #   sleep 1 min
14     jmp l1
15
16 l2: push 10             #   start loop again
17     die TOP
18     send ME,S           #   send it up
19
20     exit
```

Listing 4: Sense and send measurements to the sink

Line 10 above is an example of using code polymorphism.

The last capsule resides on the sink node, receives measurements from different nodes and stores them into the buffer (see Listing 5).

```
.sys                    # SYSTEM segment
    AUTOUPDATE 1
    LIFETIME 10s
    ID 0x41
.code.cap               # CODE segment "receive capsule"
    push CAP.ID         #   count "our" capsules only
    jmpeq 0x31,l1
    exit

l1: push CAP.BUFC
    pop SHMEM           #   append to shared memory:

    exit                #   [ID1,10 values], [ID2,10 values], ...
```

Listing 5: Collect and store measurements in the sink's buffer

Typically sensor network applications work based on principles shown above (spanning tree, periodical sampling, etc). But what if we need to make two nodes communicate to each other? In this case the following MANET-like route discovery scheme may be useful:

```
.sys                   # SYSTEM segment (a part of the capsule's header)
    AUTOUPDATE 1       #   enable autoupdate (only capsules of lower
                       #   versions will be updated)
    LIFETIME 10s       #   recognized post-fixes: ms (milliseconds),
                       #   s (seconds), p (packets)
    ID 1234            #   4-bit ID + 4-bit version number
.bufc                  # DATA segment (in-capsule data buffer)
    from=A             #   these variables are stored inside the capsule
    to
.code.init             # CODE segment "init" (executed once)
    send ME,ALL        #   broadcast itself
    mov from,ME.FROM
.code.pack             # CODE segment "receive packet" (executed multiple
                       # times upon receiving a packet)
    push PACK.LABEL
    jmpeq ME.Id,l3
    exit

l3: push PACK.SRC
    push PACK.DST
    jmpeq A,C,l1       #   process packets from C to A (some real addresses)

    jmpeq C,A,l2       #   process packets from A to C (some real addresses)

    exit               #   exit point (the capsule stays alive)

l1: send PACK,from
    mov to,PACK.FROM
    exit

l2: send PACK,to
    exit
```

Listing 6: MANET-like route discovery

The examples above can be further improved and customized to meet the requirements of a specific application.