Poster Abstract: 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

ABSTRACT

High-level programming abstractions (i.e. "middleware") primarily aim at: abstracting from heterogeneity and hardware complexity, simplification of (re-)programming, specifying system behavior in a post-hoc fashion. So far in the Wireless Sensor Networks (WSNs) domain the focus has been put mainly on the first two tasks and the application level of the last one. The meta-code approach we propose offers mechanisms to support low-level programming (e.g. time-sync, routing schemes, etc) on pre-deployed networks still requiring minimal knowledge of network characteristics. Its sole task is to provide tools to finely tune existing infrastructures with an "assembler-level" granularity in order to create task-specific highly optimized configurations. The meta-code is being designed taking into account specific requirements of WSNs, namely limited computational resources and power consumption.

Categories and Subject Descriptors: C.2.1 [Computer-Communication Networks]: Network Architecture and Design; D.3.4 [Programming Languages]: Processors

General Terms: Design, Languages, Performance

Keywords: Wireless Sensor Networks, Virtual Machines, Optimization, Code Compression

1. INTRODUCTION

Middleware and novel programming paradigms normally come as a second front after corresponding low-level protocols within a certain domain have been designed, developed and evaluated. In case of WSNs those things started happening at the same time: Maté [1], Dynamic VM [2] and Contiki VM [3] (reprogramming), Agilla [4] (mobile agents), TinyDB [5] and SwissQM [6] (queries). This might be the reason why the solutions mentioned above lack a number of important features: changeability, scalability, efficiency, etc.

By introducing the new approach called *meta-code* we try to fill the gap in which high-level programming abstractions can be used to build network architectures (full network stacks, or separate layers) in a post-deployment manner. Meta-code provides control (including version control), deployment and execution of a code without requiring any pre-configuration of the network. Meta-code can be used for building full network stacks from scratch, or to create a complementary layer to an existing one.

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

Our motivation behind creating the meta-code is to make feasible designing system-level protocols and deploying them over the existing network infrastructure. Eventually, it will allow us to build auto-configurable (can react to changes in topology, sensor attachments, etc) and self-documenting (can report changes) network systems. Another stimulus would be having a system which is able to take proactive actions (further operations are based on recognition and analysis of patterns from the past). More inspiration was taken from the works on mobile codes and chemical computing [7].

2. SYSTEM OVERVIEW

In the heart of the meta-code framework (see Figure 1) lies a stack-based extensible Virtual Machine (VM) which must be installed on each node.

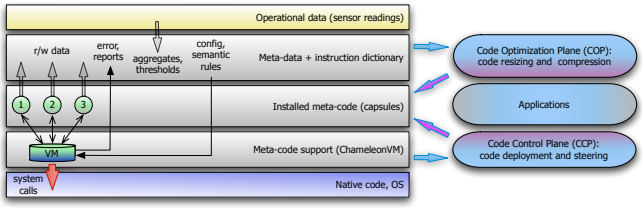

Figure 1: Meta-code framework

VM (see Figure 2) is OS-independent and type-free, although it uses system calls to perform low-level operation (forward a packet, write a memory, etc). VM has a dynamically extensible Instruction Set Architecture (ISA). VM's programs (meta-code) are delivered to a node and executed there in a form of *capsules* which can migrate from node to node. Capsule's code is versioned. Capsules support split/merge operations; they are isolated from each other but can communicate via shared memory, though.

2.1 Meta-data Management

Meta-data layer is a fundamental part of the system which serves as an exchange media between various layers (VM, capsules) and a semantics holder for meta-code. Meta-data can hold any data types: network configuration (topology, faults, etc), software versioning, instruction set dictionary, user-defined structures. Instruction Set Dictionary (ISD) allows to dynamically change ISA by defining new or removing obsolete instructions. Possible version conflicts are managed through dictionary versioning. As a result code can be optimized (compressed) at a bit-level. Nodes exchange and

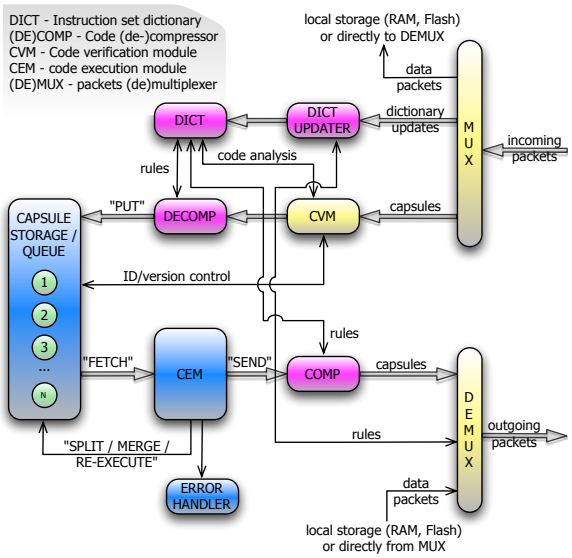

Figure 2: Node-level support for meta-code

propagate this compressed code which allows to better utilize low-bandwidth wireless links.

2.2 Meta-code Execution

Meta-code can be used either for performing system-level tasks (track changes in the network, process on-demand requests, create reports for the upper layers, etc) or to develop user applications (e.g. data gathering). Dynamically extensible and configurable dictionary-based ISA used in the VM allows meta-code to change its behavior without changing the actual code resided on the node. This also makes possible code polymorphism when one instruction can be used to perform different actions (on different nodes and/or at different time points).

3. SHOW CASES

The following show cases are supposed to give some idea of how meta-code can be used. In the first example we show how to build a classical spanning tree using meta-code.

Show Case 1: Building a spanning tree

```
.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[1]
    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[2]
    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
```

[1]"S" is some real address.

[2]"ME.*" - this capsule, "CAP.*" - capsule, "PACK.*" - packet.

```
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 from       #   send a packet up the spanning tree
    exit
```

The second show case demonstrates how to count the number of nodes in the network; the result is known at the top of the tree. The algorithm assumes that we already have an established tree topology in the network (see the example above). In addition, for this example we will have to change the spanning tree building capsule: instead of checking PACK.DST we check CAP.DST in order to receive and forward incoming capsules (not packets).

Show Case 2: Count the nodes

```
.sys                # SYSTEM segment
    AUTOUPDATE 1
    LIFETIME 10s
    ID 0x21
.code.init          # CODE segment "init"
    send ME,ALL     #   broadcast itself
    die TOP         #   clean the code located above
    send ME,S       #   send it up the spanning tree
    die
```

The last capsule is executed locally on the sink node S; it calculates all incoming "counting" capsules:

```
.sys                # SYSTEM segment
    AUTOUPDATE 1
    LIFETIME 10s
    ID 0x31
.code.cap           # CODE segment "receive capsule"
    push CAP.ID     #   count "marked" capsules only
    jmpeq 0x21,l1
    exit

l1: inc BUFS[0]
    exit
```

Another examples (present in the hand-out) include creating a MANET-like route discovery scheme and an automatic ID-assignment protocol.

4. IMPLEMENTATION AND STATUS

The meta-code framework is being implemented and tested under ContikiOS [8] (with some initial tries under TinyOS-v2). The reason for choosing ContikiOS was its features of dynamic memory allocation and module linking. In our near plans is to obtain initial results on simulation, come up with more complex proof-of-concept show cases (e.g. a lightweight skew-balance time sync protocol).

5. REFERENCES

[1] Philip Levis and David Culler. Máte: A Tiny Virtual Machine for Sensor Networks. 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X), 2002.

[2] Rahul Balani and Chih-Chieh Han and Ram Kumar Rengaswamy and Ilias Tsigkogiannis and Mani Srivastava. Multi-level Software Reconfiguration for Sensor Networks. Seoul, South Korea, October 2006. ACM Conference on Embedded Systems Software (EMSOFT).

[3] Adam Dunkels. *Programming Memory-Constrained Networked Embedded Systems, Stockholm, Sweden.* PhD thesis, Swedish Institute of Computer Science, 2007.

[4] C.-L. Fok and G.-C. Roman and C. Lu. Agilla: A Mobile Agent Middleware for Self-Adaptive Wireless Sensor Networks. pages 1–26, July 2009.

[5] *TinyDB.* http://telegraph.cs.berkeley.edu/tinydb/.

[6] René Müller and Gustavo Alonso and Donald Kossmann. SwissQM: Next Generation Data Processing in Sensor Networks. Asilomar, CA, USA, January 2007. 3rd Biennial Conference on Innovative Data Systems Research.

[7] Christian Tschudin. Fraglets - a Metabolistic Execution Model for Communication Protocols. Menlo Park, USA, July 2003. 2nd Annual Symposium on Autonomous Intelligent Networks and Systems (AINS).

[8] *ContikiOS.* http://www.sics.se/contiki/.