PYSENSE: PYTHON DECORATORS FOR WIRELESS SENSOR MACROPROGRAMMING

Davide Carboni
CRS4, Parco Tecnologico, Pula, Italy

Keywords: WSN, Macroprogramming, Python, Distributed systems.

Abstract: PySense aims at bringing wireless sensor (and "internet of things") macroprogramming to the audience of Python programmers. WSN macroprogramming is an emerging approach where the network is seen as a whole and the programmer focuses only on the application logic. The PySense runtime environment partitions the code and transmits code snippets to the right nodes finding a balance between energy consumption and computing performances.

1 INTRODUCTION

Decentralized computing architectures in wireless sensor programming are emerging in contrast with mainstream WSNs, characterized by a single application gathering and reporting data. One of the main reasons to decentralize computation is to achieve better energy efficiency and to prolong network lifetime. Radio is by far the most energy demanding device on wireless sensor boards (Pottie & Kaiser 2000), and sending a byte over 1 meter long radio channel is much more expensive (in joules) that performing an integer computation. From this simple assertion descends the choice to consider nodes in a WSN not only as simple sensors gathering data and transmitting them back to a central computer, but rather as elements capable of computation in a distributed system. To quantify how energy efficiency is achieved lets consider a simple example.

We assume as large as *1* the normalized energy needed to compute an integer instruction in a mote, and as large as *100* the cost of sending an integer over 1 meter long radio channel.

We can then obtain the energy needed to run a distributed program as follows:

$$E = O + 100 * \sum B_i * d^2_i \quad (1)$$

where E is total energy consumed by mobile nodes during the program execution; B_i is the number of integers transmitted in the i-th transmission; d_i is the distance covered by the i-th transmission; O is the total number of instructions computed by motes. We acknowledge that this model may be considered simplistic but this is not the point of this work. In (Heinzelman et al. 2002) has been proved that node-clustering and application specific data processing can prolong the network lifetime of one order of magnitude. Nevertheless, it shows in Fig. 1 how clustering and data processing on cluster heads can improve the energy balance.

If we consider a simple example of three motes equipped with thermometer and a program to compute the average temperature: in (a) every node sends its temperature and the computation is performed outside the network; in (b) the clusterhead collects data from its nodes and act as a router; in (c) the cluster head perform the computation and sends a smaller amount of data to the central computer.

According to the energy model seen before the energy consumption is 230 in (a), 160 in (b) and 60.03 in (c).

No matter the model chosen for E, our work is to define a programming tool that supports the clustering of motes in order to minimize the energy consumption delegating computation to cluster heads and to motes depending on computational capabilities. In other words, we argue that given an energy consumption model E and an application code C, there exists a partitioning of code $C=\{c_1,c_2,...,c_n\}$ and a set Tx of transmissions $Tx=\{tx_1,tx_2,...,tx_k\}$ which is optimal for E. In this position paper we envision a programming environment called PySense which supports the

Carboni D. (2010).
PYSENSE: PYTHON DECORATORS FOR WIRELESS SENSOR MACROPROGRAMMING.
In *Proceedings of the 5th International Conference on Software and Data Technologies*, pages 165-169
DOI: 10.5220/0003038801650169

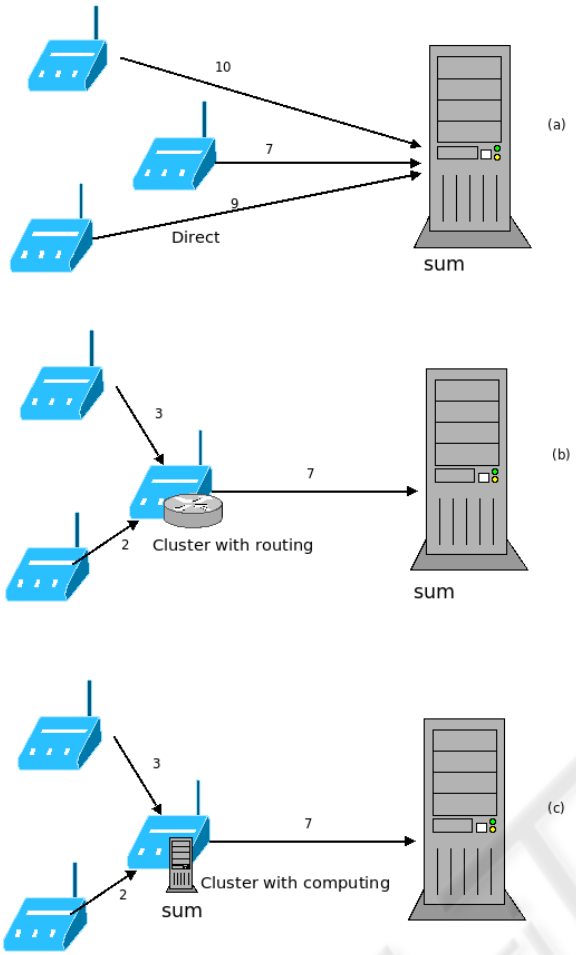

Figure 1: In (a) every node sends its temperature and the computation is performed outside the network; in (b) the cluster head collects data from its nodes and act as a router; in (c) the cluster head perform the computation and sends a smaller amount of data to the central computer. The number over arrows is the distance covered by a transmission.

programmer in partitioning the code among nodes in order to find the optimal compromise between computation and communications.

2 THE PYSENSE SYSTEM

The idea is to base the language on Python because is a development language widely adopted in PC applications and that have proven to be very appreciated for its balance of functional, procedural and object oriented constructs.
The elements of the system are:

1. The PySense language and API, which is a language hosted by Python decorators. PySense programs are written in Python and the idea is to use the Python decorator to add some semantics to classes and functions constructs. In this way no new keywords are added to the language so existing code is not broken.
2. The Base Runtime Environment (BRE) is a computer with unlimited energy and unbounded computing capabilities (i.e. memory, CPU, storage, threading).
3. The Remote Runtime Environment (RRE), is the execution environment running on board of motes and able to receive and *deploy* Python expression or bytecode at runtime.

The code partitioning is the strategy that allows the BRE to analyse the code and according to the network topology to split the program into pieces to be deployed either on the BRE or on one or more RREs. Our approach recalls the aspect-oriented programming in which cross-cutting concerns are kept aside from business logic. The concerns about code partitioning and migration on motes are implemented with meta-programming inspecting the code and modifying it at runtime. Few classes are defined in the API: *Cluster*, *Region*, and the class decorator *Mote*.

Region describes a region in the space. A *Region* can be defined either from a cartesian/polar bounding box or from a graph path. On *Region* instances one can apply the set algebra operations as specified in Python. For instance, the expression:

```
Region((0,0,100,100)) |
Region("/floors/3/312")
```

defines the union ('|') between the bounding box *(0,0,100,100)* and the room *312* on third floor of the building root ('/'). While the expression:

```
R.items(M)
```

returns the set of items (motes) defined by class *M* located in the region *R*. The details of the location mechanism are beyond the scope of this paper. For sake of clarity, we assume that the position of motes is either known in advance by the BRE or dynamically retrieved by some system not covered here.

The Cluster class is the super class for all clusters. Given C a subclass of Cluster it is possible to create a cluster with the expression:

```
C([m1,m2,....,mk])
```

where m_i are instances of classes decorated with *@mote* (see later). The expression:

```
C(R.items(M))
```

is the cluster of motes located inside region *R*. How clusters are composed and which mote is elected as

cluster head is beyond the scope of this paper and will be investigated in future works. We want just point the fact that a cluster may elect both a cluster router and a cluster computer which may be running on different motes. The former is chosen according to the network topology and distribution of motes in space, while the latter is chosen according to memory and CPU capabilities.

As previously mentioned, the PySense language is hosted in Python decorators. Python decorators are special purpose functions that are invoked in the code when classes or functions are defined. The syntax of decorators is:

```
@<func decorator>
def f(x):
    <some code here>

@<class decorator>
class C:
    <some code here>
```

respectively for decorating a function *f* and for decorating a class *C*. The use of decorators is some how similar to aspect programming (Elrad et al. 2001) and allows to keep aside all the code needed to inspect the application logic and to modify it according to the partitioning needed for macroprogramming.

The first decorator is *@mote* and is applied to classes. A class *M* decorated with *@mote* is inspected to find *getter* and *setter* methods. We adopt the convention that a getter is a "protocol" to read from a remote sensor while a setter is a "protocol" to write on a remote effector. Thus, the class:

```
@mote
class M:
    def getX(self):pass
    def setY(self,y):pass
```

becomes in the BRE a proxy class to real motes equipped with a sensor named X.
An invocation:

```
m.getX()
```

will result in a message:

```
TO m.addr READ x
```

where *m.addr* is the network address of the mote associated to instance *m*. While the invocation:

```
m.setY(2)
```

is translated in a message:

```
TO m.addr WRITE y 2
```

Invoking an action on a mote (e.g. a rotation of an engine) may require an unpredictable amount of energy. This possibility is given but the programmer must carefully consider the power consumption on his own.

Differently from normal Python classes, *@mote* classes are limited in the number of instances that they can create. If we define a class as follows:

```
@mote
class M:
    def getX(self):pass
    def getY(self):pass
    def setZ(self,value):pass
```

That means that only motes equipped at least with sensors *x* and *y* and with actuator *z* can be associated with *M*. The number of such motes is the maximum number of *M* instances that the BRE can create with *M()*. Any further invocation of *M()* will cause an exception to be thrown. A *@mote* class can have also pure computational methods (to be distinguished from getter/setter). Some decorators can be used for these methods. In the example below:

```
@mote
class M:
    def getX(self):pass

    @onboard
    def f(self,args):<some code here>

    @onbase
    def g(self,args):<some code>

    @auto
    def h(self,args):<some code>
```

The method *f* is expected to run on RREs after a lazy deployment: the bytecode of *f* is sent to a mote only when *f* is called on that mote for the first time. Moreover, a mote required to compute *f* can receive the bytecode of *f* not necessarily from the BRE, but if available, from a closer mote or from its cluster head.

The method *g* is simply executed on the BRE and this does not pose any issue. The code of method *h* is treated in a way discussed below. When moving code from BRE to RRE three different approaches are in principle possible:

1. The names that cannot be resolved on RRE are transmitted to from BRE to RRE. This implies the movement of bytecode for each unresolved name (e.g. functions, classes, globals) and given that each piece of code can depend on other pieces this may cause the movement of an entire graph of code. This approach is highly demanding in terms of memory for the recipient RRE and the initial deployment is also costly in terms of bandwidth.
2. Another approach is to not solve locally on

the RRE but instead to replace every reference to an unknown name with a remote call from the RRE to the BRE. This approach is lightweight in terms of memory but may require a lot of messages from RRE back to BRE.

3. The last approach is to prevent code deployment on the RRE if one or more names are remotely unknown and in that case the BRE launches an exception causing the programmer to reconsider the design of the code.

In our design, for *@onboard* methods the third approach is always used, while for *@auto* the system determines automatically which is the best strategy according to network topology and motes capabilities.

When on BRE is invoked a code *f(x,y,...)* deployed on a RRE, a message:

```
TO <addr> CALL f  x,y,...
```

is produced and sent to the mote with address addr. The decorators *@onboard*, *@onbase* and *@auto* can be also used to decorate methods of *Cluster* instances. In this case the semantics is the same than before, except that an unknown mote (unknown at design time) is elected cluster head and delegated to perform remote computation.

As mentioned by (Newton et al. 2007) no paper in this area can elude the example of the spatial average temperature.

```
@mote
class M:
    def getTemperature(self):pass

class C(Cluster):
    @onboard
    def average(self):
        return sum(
[m.getTemperature() for m in
self.motes]) / len(self.motes))

C(region.items(M)).average()
```

The simple program above defines the class *M* as a mote with temperature sensor, then defines a cluster *C* and engages it to compute the average method on the cluster head. Every *m.getTemperature* invocation is translated as a message:

```
TO <m.addr> READ temperature
```

sent from the cluster head. The variable *region* is a generic *Region* instance previously created. In the following section are described some simple programs similar to those described for Regiment (Newton et al. 2007).

3 SAMPLE PROGRAMS

The first program measures the CO2 concentration and sends back data to base runtime. The result variable *values* is scoped in the BRE.

```
@mote
class CO2Sense:
    def getConc(self):pass

values=[c.getConc() for c in
region.items(CO2Sense)]
```

In the example above, all motes are inquired by the BRE. In the example that follows a cluster is first composed and data are collected by the cluster head and then sent to the BRE.

```
@mote
class CO2Sense:
    def getConc(self):pass
class CO2Cluster(Cluster)
    def collect(self):
        return [m.getConc() for m in
self.motes]

values=CO2Cluster(region.items(CO2Sense
)).collect()
```

The next program behaves as the previous one, but only values beyond a threshold TH are sent to the BRE.

```
class CO2(Cluster):
    @onboard
    def collect(self):
        filter(lambda x:x>TH ,
[m.getConc() for m in self.motes])

values=[CO2Cluster(region.items(CO2Sens
e)).collect()]
```

4 RELATED WORKS

This position paper is located in the field of distributed computing and in particular in the field of WSN programming. WSNs are distributed systems equipped with limited computational capabilities, radio communication stack and very constrained energy resources. As assumed in the

previous sections, the computation is less energy demanding than communication and thus computation and communication activity must be organized together.

Many authors have proposed languages and middleware for WSN programming, most of them use nesC (Gay et al. 2003) as C dialect for node level programming with execution of native code in the motes, while other systems like Maté (Levis & Culler 2002) use a virtual machine approach.

For the systems cited above high-level, global behaviour must be expressed in terms of complex, local actions taken at each node. To address this issue different network-oriented programming models have been proposed. One of the most assessed strategies is to consider the whole WSN as a distributed database to be queried with SQL expressions (Madden et al. 2005) or XML expressions. This approaches are not well-suited for developers who wish to implement specific behaviour at a lower level than the query interface. Another promising approach is the so-called macro-programming. In Regiment (Newton et al. 2007) a global defined logic is automatically de-globalized into pieces of code at node level. The Regiment syntax is inspired by ML(Milner 1997) while PySense is based on Python. Another macro-programming language based on Python is Kairos (Gummadi et al. 2005) in which the programmer is presented with three constructs: reading and writing variables at nodes, iterating through the one-hop neighbours of a node, and naming and addressing arbitrary nodes. Kairos language is an extension of the Python language while PySense is a hosted language on top of Python decorators and relies on the runtime deployment of code on mobile Python interpreters. Kairos authors have apparently abandoned the project and have recently proposed Pleiades(Kothari et al. 2007) , an extension of C language with dynamic code partitioning and migration.

5 CONCLUSIONS AND FUTURE WORK

We consider useful developing a new tool in this area given the huge potential of WSNs as technology and the easiness to learn Python as programming language even for application domain experts.

At the time of writing, PySense is still incomplete to be used in real world applications and its development is ongoing. The BRE is almost completely implemented while the set of RRE is at the moment emulated by a network server that emulates the reception of requests, the transmissions of responses, and the computation of migrated code.

The real RRE is under development and based on the Pymite VM developed in the Python-on-a-chip project (python-on-a-chip). Thus, the current commitment for the PySense project is now to fulfil the design depicted in the paper with a complete implementation to be tested with real world applications.

REFERENCES

python-on-a-chip - Project Hosting on Google Code. http://code.google.com/p/python-on-a-chip/

Elrad, T., Filman, R.E. & Bader, A., 2001. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10), 29–32.

Gay, D. et al., 2003. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. pag. 11.

Gummadi, R., Gnawali, O. & Govindan, R., 2005. Macro-programming wireless sensor networks using kairos. *Lecture Notes in Computer Science*, 3560, 126–140.

Heinzelman, W.B. et al., 2002. An application-specific protocol architecture for wireless microsensor networks. *IEEE Transactions on wireless communications*, 1(4), 660–670.

Kothari, N. et al., 2007. Reliable and efficient programming abstractions for wireless sensor networks. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. pag. 210.

Levis, P. & Culler, D., 2002. Mate: A tiny virtual machine for sensor networks. *ACM SIGARCH Computer Architecture News*, 30(5), 95.

Madden, S.R. et al., 2005. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems (TODS)*, 30(1), 173.

Milner, R., 1997. *The definition of standard ML*, MIT Press.

Newton, R., Morrisett, G. & Welsh, M., 2007. The regiment macroprogramming system. In *Proceedings of the 6th international conference on Information processing in sensor networks*. pag. 498.

Pottie, G.J. & Kaiser, W.J., 2000. Wireless integrated network sensors. *Communications of the ACM*, 43(5), 51–58.

Younis, O., Krunz, M. & Ramasubramanian, S., 2006. Node clustering in wireless sensor networks: Recent developments and deployment challenges. IEEE Network, 20(3), 20–25.