Hindawi Publishing Corporation
International Journal of Distributed Sensor Networks
Volume 2013, Article ID 797354, 12 pages
http://dx.doi.org/10.1155/2013/797354

Research Article

Pymote: High Level Python Library for Event-Based Simulation and Evaluation of Distributed Algorithms

Damir Arbula and Kristijan Lenac

Department of Computer Engineering, Faculty of Engineering, University of Rijeka, 51000 Rijeka, Croatia

Correspondence should be addressed to Damir Arbula; damir.arbula@riteh.hr

Received 4 November 2012; Accepted 19 December 2012

Academic Editor: Long Cheng

In recent years we have witnessed strong development and widespread use of powerful wirelessly connected platforms, thus the set of the related problems that need to be solved by distributed algorithms is growing rapidly. Some of them present large obstacles in harnessing the full potential of this new technology, so there is an imminent need for a fast and easy evaluation of new ideas and approaches. Simulation is a fundamental part of distributed algorithm design and evaluation process. In this paper, we present a library for event-based simulation and evaluation of distributed algorithms. This library provides a set of simple but powerful tools with a goal to ease virtual setup of a complex system such as a distributed network of communicating entities and to define, simulate, and analyze its behavior. In order to reduce a huge problem space inherent in such systems, our library is using a high level of abstraction. This is made possible by a strict and complete definition of the distributed computing environment. The library is implemented in Python whose simple and expressive syntax provides a possibility of minimal implementations and a mild learning curve. In addition to executing automated simulations or larger experiments, the library fully supports interactive mode along with a step-by-step execution, which can be a very powerful combination.

1. Introduction

The evaluation of distributed algorithms calls for adequate simulation and comparison with existing state-of-the-art solutions. Although it seems that this task is straightforward, in practice there are a few issues that need to be handled correctly. Existing simulation environments, like OMNeT++, usually require definition of large set of low level parameters (transmitter frequency, communication protocol, etc.) to simulate the behavior of a system as close to implementation as possible. There is certainly nothing wrong with that approach but when a problem is defined in a more generic way (i.e., anchor free localization of wireless sensor nodes with ranging capabilities) selection of some of those parameters is not problem related, thus they are highly arbitrary. This can lead to a simulation of a more specific case than needed.

Here we present a high level Python library for event-based simulation of distributed algorithms in wireless ad hoc networks. The library allows the user to make implementation of their ideas using Python—a popular, easy to learn, full featured, object oriented programming language. Functionalities provided by the library are implemented without additional layer of abstraction, thus harnessing full power of Python's native highly expressive syntax. Using the library, users can quickly and accurately define and simulate their algorithms.

The library particularly focuses on

(1) fast and easy implementation of ideas and approaches at algorithm level without any specification overhead using formally defined distributed computing environment;

(2) support for two different workflows and their seamless combination: (1) *interactive* control and step by step execution of simulation with easy introspection and modification of all objects in the runtime environment, and (2) *fully automated* creation and modification of simulation environment and running multiple experiments in a clean and minimal way using simple Python scripts;

(3) promoting open source reproducible research, thus encouraging its reuse and reevaluation trough comparisons with new ideas based on easily customized criteria.

Since the library is built upon a formally defined distributed computing environment, implementation of specific algorithm is a straightforward process. This process is additionally alleviated by using interactive console and the native Python debugger in which all objects are directly accessible for introspection and modification.

Python is a modular language, so advanced usage or extension of basic functionalities is easily available by writing additional modules and inheriting from the library core classes. These extensions are encouraged through the open source developing workflow.

In the next section, a brief analysis of currently available simulators and libraries is given. In Section 3, we formally define distributed computing environment giving theoretic foundation and basic principles on which this library is based. This is followed by a short discussion on platform selection, description of some implementation details and possible ways of extending the library beyond its current functionalities. Finally, in Section 5, there is an example of definition, simulation, and analysis of one of the popular localization algorithms using the library in interactive and automated workflow.

2. Related Work

A large number of simulators have been proposed in literature in which algorithms for wireless ad hoc networks can be implemented and studied. These simulators have different design goals and largely vary in the level of complexity and included features. They support different hardware and communication layers assumptions, focus on different distributed networks implementations and environments, and come with a different set of tools for modeling, analysis, and visualization. Classical algorithms include NS-2, OMNeT++, J-Sim, TOSSIM, and others.

NS-2 [1] is a discrete event simulator for general network simulation. It is probably the most widely used network simulator for research. NS-2 provides substantial support for simulation of TCP, routing, and multicast protocols over wired and wireless networks. It was originally targeted to IP networks but extensions [2, 3] for wireless sensor networks have been proposed in the past. NS-2 allows for a detailed simulation tracing and comes with the simulation tool called NAM (network animator) for later playback. Due to a very detailed packet level simulation, the NS-2 simulator is not suitable for simulation of very large networks made of thousands of nodes. NS-2 has many different distributions and extensions and a large number of network simulations have been performed with them. However NS-2 has a steep learning curve and requires considerable effort to repeat the simulations and compare with the obtained results. The core of the simulator and most of the network protocol models are written in C++, while OTcl is used for the definition and configuration of the simulation environment. It is available under an open source license.

Wireless sensor network simulation can be performed using Mannasim framework which extends NS-2 by introducing new modules for design, development, and analysis of different WSN applications. The Mannasim framework provides standardized structures for common sensor, cluster heads, and access point nodes on top of NS-2. In a simulation these three types of nodes run different algorithms which are implemented directly in C++.

Another simulation platform which is widely used in the global scientific community is OMNeT++ [4, 5]. It is an extensible, modular, component-based C++ simulation library and framework, used primarily for building network simulators. OMNeT++ offers extensive simulation library that includes support for input/output, statistics, data collection, graphical presentation of simulation data, random number generators, and data structures. Domain-specific functionality such as support for sensor networks, wireless ad hoc networks, Internet protocols, performance modeling, and so forth are provided by model frameworks and developed as independent projects. Extensions for real-time simulation, network emulation, alternative programming languages, database integration, and other functions exist. For the simulation of wireless sensor networks one popular extension is Castalia [6]. Its development was motivated by the desire to provide a realistic channel/radio modeling. However, for the development of high level algorithm and studying of its behavior Castalia is time consuming because one has to account for all the details related to specific low level modeling.

J-Sim [7] is a general purpose simulator written in Java according to the component-based software paradigm. Components are loosely coupled as each component can be designed, implemented, and tested independently. On the top of the autonomous component architecture, a generalized packet switched network model defines the generic structure of a node and the generic network components, both of which can then be used as base classes to implement protocols across various layers. J-Sim was initially designed for wired network simulation, but a wireless extension exists which proposes an implementation of the IEEE 802.11 MAC together with a set of network and protocol components which facilitates the simulation of wireless networks. It supports real-time process-driven simulation.

TOSSIM [8] is a platform-specific simulation library which simulates TinyOS [9] motes at the bit level. It allows emulation of system components and modeling of different network topologies thus providing a realistic setting for measurement of the communication costs of algorithms. TOSSIM is an open source discrete event simulator which directly compiles code written for TinyOS to an executable file that can be run on standard PC equipment. It can run simulations with a few thousand virtual TinyOS nodes. It ships with the graphical user interface TinyViz that can visualize and interact with running simulations.

For a survey and comparison of these and other simulation platforms the reader can refer to [10–12].

One classification [13] divides simulators into three major categories based on the level of complexity:

(1) algorithm level,

(2) packet level, and

(3) instruction level.

Algorithm level simulators focus on the logic, data structure, and presentation of algorithms. These simulators do not consider detailed communication models and, most commonly, they rely on some form of a graph data structure to illustrate the communication between nodes. Packet level simulators implement the data link and physical layers in a typical OSI network stack. Hence, it is common for this type of simulators to include implementations of 802.11b or newer MAC protocols and radio models that account for propagation, fading, collision, noise, and wave diffraction. Instruction level simulators model the CPU execution at the level of instructions or even cycles. They are often regarded as emulators.

According to this classification, Pymote is an algorithm level simulator. In comparison to the abovementioned widely used simulators, Pymote does not provide packet level and instruction level simulation. Instead, it uses abstract models of the communicating entities and the environment thus enabling researcher to focus on a general principles not influenced by large amount of implementation details. This fact makes it considerably easier to learn and more straightforward to use. An additional benefit of such decision is scaling of the simulation environment in a way that can accommodate large networks. Pymote is focused on the design and evaluation of algorithms while also providing tools for quick definition of different network structures. In the remaining part of this section we describe some of the algorithm level simulators that were proposed in the literature.

AlgoSensim [14] is a framework used to simulate distributed algorithms. It focuses on network specific algorithms like localization, distributed routing, flooding, and so forth. It is written in Java and uses XML files for configuration. The framework was published as open source in 2006 but has remained in alpha release since.

Shawn's [15] primary design goals are to simulate the effect caused by a phenomenon, not the phenomenon itself, to improve scalability, and to support free choice of the implementation model. Instead of performing a complete simulation of the MAC layer including radio propagation properties such as attenuation, collision, fading, and multipath propagation, Shawn simulates the effects of a MAC layer for the application like packet loss, corruption, and delay. In this way, while producing similar effects on the application layer, a performance gain is obtained with a more efficient implementation. This additionally enables Shawn to support large-scale network simulation. Shawn is written in C++.

NetTopo [13] is an integrated framework for simulation and visualization of wireless sensor networks written in Java. Its design is also algorithm-oriented with the goal of rapid prototyping of algorithms; however, it derives its motivation by the need to study applications which can run partially in a simulation environment and partially in a physical wireless sensor network testbed. NetTopo supports the simulation of large scale networks and provides a graphical user interface to drive the simulation.

Sinalgo [16] is another simulation framework for wireless networks written in Java which does not simulate the different layers of the ISO network stack and focuses instead on algorithm layer abstraction. It offers a message passing view of the network and can simulate very large networks. Sinalgo comes with a set of available models for node mobility, connectivity, initial distribution, interference, and transmission which a user can extend with his own if necessary. The simulation is usually started from the available graphical user interface, but for long-lasting well-defined simulations it can also be started in batch mode.

One important difference of these algorithm level simulators with Pymote is the programming language and environment. Pymote leverages Python's strengths like ease of learning and use and faster development than C++ and Java, making the solution well suited for rapid prototyping. Further difference is that Pymote naturally supports both interactive and programmed simulation modes enabling and actually fostering quick, often intertwined definition and simulation phases. By using Python's introspection power, that is, its ability to inspect objects at runtime, determine information about them, and make that information available to the user, Pymote is able to interactively explore and manage all the entities involved in the simulation.

3. Distributed Computing Environment

To design a proper distributed algorithm, the environment in which it performs must be strictly defined. Distributed environment and restrictions to problem space used in specifications for making Pymote library are taken from [17]. Main principles governing algorithm operation in this environment are described below.

Distributed computing environment is composed from a set of *computational entities* $\mathscr{E}$ (in our case wireless nodes) and *messages* they interchange. A node $x \in \mathscr{E}$ has the capability to store data in finite local memory M_x consisting from a number of defined registers. One of them, with special function, is the *status* register that can take values from a finite set of states $\mathcal{S}$. Other parts of the node x are *CPU* and *communication*.

Node behavior is reactive. It acts only when it detects one of two possible events: (1) arrival of message, and (2) spontaneous impulse—usually used in random or defined nodes at algorithm initiation. The action that node performs is a result of its state (in status register) and the event, which in our case is the arrival of message as follows:

$$\text{status} \times \text{event} \longrightarrow \text{action}. \tag{1}$$

Distributed algorithm is defined as a set of rules that associate all possible combinations of states and events with specific actions.

The system has homogenous behavior if all nodes run the same algorithm which simplifies its development and analysis. Since every nonhomogenous behavior can be made homogeneous, as described in [17], all nodes in our library are running the same algorithm.

Table 1: Message structure.

Field name	Data type	Description
Source	Node	Sender node instance
Nexthop	Node	Neighboring node instance that is next hop in path to destination
Destination	Node	Destination node instance
Header	String	Message header defining function and structure of sent data
Data	Dict	Any data

There are three types of actions: storing and processing data, message transmission, and changing the value of status register. All other actions, such as measuring some phenomena or relation between nodes, can be thought of as a combination of message transmission, storing, and processing data.

Message transmission between nodes is defined as a transfer of finite sequence of bits. Simple but generic structure and description of fields in messages used in our library is presented in Table 1.

Every node x can send a message to a set of other nodes $N_{\text{out}}(x) \in \mathscr{E}$ and receive it from set $N_{\text{in}}(x) \in \mathscr{E}$. Inherent property of every node is *local orientation* which means that a node can distinguish between its neighbors by their unique ID, so for example it can send a message to a specific neighbor without sending it to the other neighbors. This is utilized through `destination` field in the message structure.

In practice, sending and receiving messages are complex operations liable to failures and spanning through several communication layers with unknown, possibly infinite duration. When designing distributed algorithms, it is very important to make their performance invariant to communication failures and delays.

Up to this point, all network properties have been defined for some general case. Special properties are called *restrictions* because algorithms designed under assumption of this properties are restricted and cannot be applied to a more general, unrestricted, cases.

Current restrictions in our library are as follows.

Bidirectional Links. For every node x, a set of outgoing and incoming links are identical $N_{\text{out}}(x) = N_{\text{in}}(x)$, for all $x \in \mathscr{E}$ and the nodes that belong to this set are called *neighbors* of x.

Connectivity. In cooperative distributed algorithms it is very important that every node can communicate to all other nodes, directly or through intermediate nodes.

Total Reliability. Every sent message eventually, in unknown but finite time, will be received with its content uncorrupted and without any failure occurring.

These restrictions are strong but sensible. They try to limit problem space to an application layer only and remove unwanted overspecification by focusing on a generic case solutions.

Node's memory content and information held in it represent its *local knowledge*. If at least one node in a set $W \subseteq \mathscr{E}$ has information p, it is defined as *implicit knowledge*. If every node in a set W has the same information then it is *explicit knowledge*. Notion of knowledge is very important since the sole purpose of (distributed) algorithm is improving local knowledge of nodes and implicit and explicit knowledge of the network in general.

4. Implementation

The Pymote library is based on goals described in Section 1 and formalism of the distributed computing environment stated in Section 3. We proceed with discussion on platform selection and brief description of the implementation of basic library functionalities. We conclude with a description of some of directions that can be taken to extend them.

4.1. Why Python? Python language with included libraries and tools was selected after research and analysis of a number of existing platforms and frameworks. The selection of Python allows to fulfil the following requirements:

(i) easy to learn and well documented;

(ii) full featured object oriented language. Partial or restricted solutions (i.e., MATLAB) are bound to have a limit regarding supported object oriented features;

(iii) simple and highly expressive. As such it keeps the code clean and minimal, making its usage straightforward;

(iv) support for interactive mode since this kind of workflow is especially suitable for experimentation and analysis. *IPython* [18] is a Python interactive console that provides all major functionalities needed for interactive scientific computing (Figure 1);

(v) introspection features, namely, easy programmatic inspection of all defined object's properties;

(vi) rich scientific functions library and strong support for scientific computing and calculations. Although Python is a general purpose language, in this field it is very competitive. *NumPy* and *SciPy* [19] are the fundamental packages for scientific computing in Python. They add significant power to the interactive Python session by exposing the user to high-level commands and classes needed for all kinds of scientific data manipulation. In addition to these, for plotting and general graphical representation of data, *matplotlib* is a perfect choice;

(vii) promote open source reproducible research: Python is a platform that already promotes similar ideas and that grows with the community of its users and developers.

After long and exhaustive usage, we have concluded that Python completely fulfills the requirements.

4.2. Core Classes. The implementation philosophy followed DRY (Do not Repeat Yourself) and KISS (Keep It Simple Stupid) principles. DRY principle states our intention to make a library that will not rediscover and rewrite already existing

```
Python 2.7.1 (r271:86832, Nov 27 2010, 18:30:46) [MSC v.1500 32 bit (Intel)]
Type "copyright", "credits" or "license" for more information.

IPython 0.13.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

Welcome to pylab, a matplotlib-based Python environment [backend: Qt4Agg].
For more information, type 'help(pylab)'.
In [1]:
```

Figure 1: IPython interactive console.

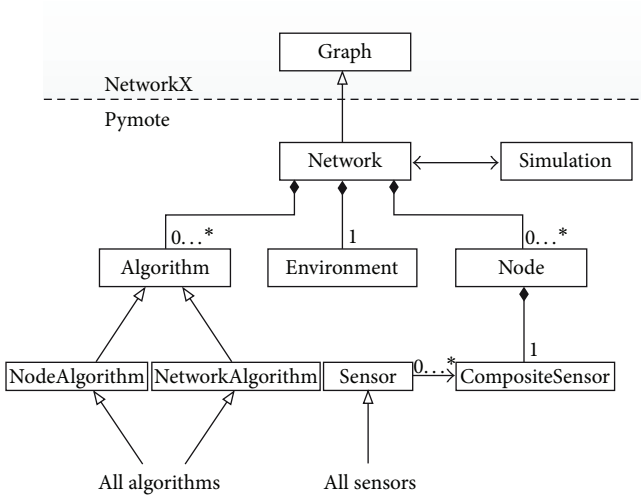

Figure 2: Simplified class diagram.

functionalities but one that will build on top of the existing code. KISS principle reflects the need to simplify library usage and library implementation. To adhere with the DRY principle, Pymote library is placed in such a position that it does not repeat work that has been already done. Since the network is identified as an instance of graph, the only sensible choice was to find a package on top of which this library should be built.

NetworkX [20] is a package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. Its graph definition is as general as possible allowing for easy extension. It has a huge base of graph-related functions and methods already implemented. Pymote library extends from NetworkX and defines few core classes that we describe in this section. Simplified class diagram describing relations between core classes is presented in Figure 2.

Network class is a subclass of the NetworkX's Graph class and as such it implements all methods provided by this class. In addition, all graph-related algorithms implemented in this package are available directly in Pymote. Based on node positions (stored inside network), environment, communication range and channel type, Network is managing creation and removal of links, namely, edges of the underlying Graph. A list of algorithm instances that should be executed is placed inside Network instance. Other responsibilities of Network class include communication or passing messages between neighboring nodes and managing state of algorithms that are executed on a network.

Node class represents a wireless node. Every node has an id to satisfy local orientation property. Additionally, it has a set of memory fields (outbox, inbox, status, and general purpose memory) and communication range property that is used as an argument to a Network class method that is managing communication links. In order to simulate node's knowledge in a more realistic way, a node does not know anything that it would not know in real deployment, for example, the node does not know its position inside the network. If this information should be part of node's knowledge (i.e., node is an anchor), the user could ensure this in experiment setup by equipping the node with an adequate sensor, as demonstrated in Section 5.2.

Instances of *Simulation* class are used to control step-by-step execution of algorithms defined in the network. Simulation is using methods defined in Algorithm class to execute them either on a every node or on a network level, as presented in a sequence diagram in Figure 3. During execution, network and its nodes' states are changing but the simulation instance does not store any data in the process. All data before, during, and after simulation is held inside the network instance which is important as all needed data can be stored simply by serializing and storing the network instance.

Algorithm class represents an executable code that should run inside a network. Currently there are two main subclasses of Algorithm class: NetworkAlgorithm and NodeAlgorithm.

NodeAlgorithm is a classic distributed algorithm that runs in every node. Its start is triggered spontaneously either in randomly chosen nodes or defined ones. Every action after the algorithm initiation is a result of incoming message and state in which the node is in.

There is a notable difference between local termination where the node knows it is done with all actions in the current algorithm and a global termination in which the node knows that all other nodes are also done. Only global termination can be a simple and clear signal to advance further to the next algorithm. In our library the simulation instance (which is used to execute algorithms on a network) is responsible for detecting when a certain algorithm has terminated. It can be detected by the absence of messages (local termination) or by a node returning a clear signal that every node has terminated its execution (global termination).

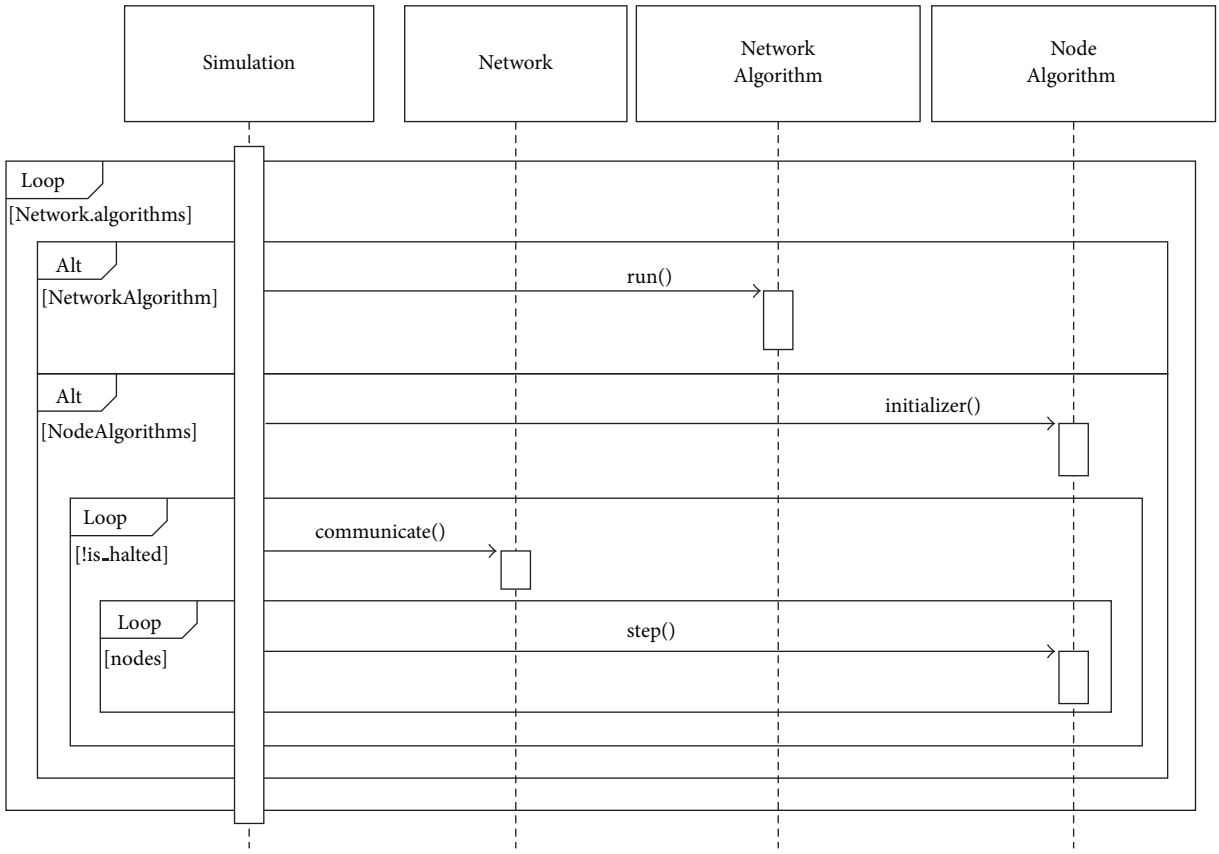

Figure 3: Algorithm sequence diagram.

To simplify the definition and running of distributed algorithms with known behavior, output, and cost, it is allowed for them to use their centralized network version represented by NetworkAlgorithm class. Algorithms that are subclass of NetworkAlgorithm have the ability to improve nodes local knowledge by inserting data directly into their memory. For example, network version of distributed spanning tree algorithm should have the ability to write parent-child relationship directly into every nodes memory. These algorithms are usually helper algorithms that could insert prerequisite knowledge data for algorithms under test. Algorithms under test should always be defined as a proper distributed algorithm or, namely, NodeAlgorithm.

4.3. Extending Base Functionalities. As we have stated in the introduction section, Pymote is primarily focused on the simulation of high level algorithms. Communicating entities should not affect the outcome of the high level algorithm and this is ensured by introducing restrictions such as total reliability, described in Section 3.

Even though they should not affect the outcome, entities and their lower level behavior can affect the algorithm performance, that is, battery usage, reliability of the communication links, and so forth. If the user wants to simulate and inspect lower level behavior or use specific communication protocols it certainly can be done.

For example, low level simulation of battery usage or communication reliability currently can be handled by overriding the `communicate` method of the Network class to simulate and track power consumption or other custom behaviors.

Another example is radio channel simulation which can be done by writing custom ChannelType class and then passing it to network instance. Library is currently implementing two simple ChannelType subclasses: Udg-unit disc graph and SquareDisc in which probability of connection between nodes is equal to $1 - d^2/r^2$ where d is the true distance between nodes and r is the communication range.

The simplest example is the routing protocol which can be implemented as a native NodeAlgorithm with the task of maintaining routing table in the node memory. Data in routing table can be used to appropriately fill `nexthop` field that is already present in the message structure (Table 1) and used in `communicate` method when sending message to nonneighbor node.

All these types of extensions can quickly be incorporated into future versions of the main library. Using collaborative open source workflow, this kind of development is strongly encouraged.

5. Usage

Pymote library supports usage in two distinct workflows. One of the most popular workflows in scientific community

```
class FloodingUpdate (NodeAlgorithm):
   required_params = ('dataKey',)
   default_params = {}

   def initializer (self):
      for node in self.network.nodes():
         if self.initiator_condition(node):
            msg = Message(destination=node,header='initialize')
            self.network.outbox.insert(0,msg)
         node.status = 'FLOODING'

   def flooding(self, node, message):
      if message.header=='initialize':
         node.send(Message(header='Flood',
                           data=self.initiator_data (node)))
      if message.header=='Flood':
         updated_data = self.handle_flood_message (node,message)
         if updated_data:
            node.send (Message(header='Flood',
                               data=updated_data))

   STATUS = {'FLOODING': flooding,}
```

Listing 1: Generic flooding protocol. Function initializer is special function that is issuing spontaneous impulses in the form of *initialize* messages at the beginning of the algorithm execution.

```
class DVHop(FloodingUpdate):

   def initiator_condition(self, node):
      node.memory[self.truePositionKey] = node.compositeSensor.read(node).get('TruePos',None)
      return node.memory[self.truePositionKey] is not None

   def initiator_data(self, node):
      return{node: concatenate((node.memory[self.truePositionKey][:2],[1]))}

   def handle_flood_message(self, node, message):
      if not node.memory.has_key(self.dataKey):
            node.memory[self.dataKey] = {}
      updated_data = {}
      for landmark, landmark_data in message.data.items():
         if landmark==node: continue
         # update only if data is new or new hopcount is smaller
         if not node.memory[self.dataKey].has_key(landmark) or
               landmark_data[2]<node.memory[self.dataKey][landmark][2]:
            node.memory[self.dataKey][landmark] = array(landmark_data)
            # increase hopcount
            landmark_data[2] += 1
            updated_data[landmark] = landmark_data
      # landmarks should recalculate hopsize
      if node.memory[self.truePositionKey] is not None:
         self.recalculate_hopsize(node)
         return updated_data
```

Listing 2: APS DV-hop 1st phase: DVHop. Function *recalculate_hopsize* is omitted. Note that during *initiator_condition* node reads TruePosSensor if it has one.

```
class Trilaterate(FloodingUpdate):

    def initiator_condition(self, node):
        return node.memory[self.truePositionKey] is not None

    def initiator_data(self, node):
        return node.memory[self.hopsizeKey]

    def handle_flood_message(self, node, message):
        if node.memory.has_key(self.hopsizeKey):
            return None
        node.memory[self.hopsizeKey] = message.data
        self.estimate_position(node)
        return node.memory[self.hopsizeKey]
```

Listing 3: APS DV-hop 2nd phase: trilaterate. Function *estimate_position* is omitted.

```
# create network with degree 9
In [1]: netgen = NetworkGenerator(n_min=100,n_max=300,degree=9)
In [2]: net = netgen.generate()
# select landmarks by placing TruePosSensor on them
In [3]: for node in net.nodes()[:10]:
   ....:     node.compositeSensor = CompositeSensor(('TruePosSensor'))
# import algorithms and pass them to network with their parameters
In [4]: from pymote.algorithms.niculescu2003.dvhop import DVHop
In [5]: from pymote.algorithms.niculescu2003.trilaterate import
      Trilaterate
In [6]: net.algorithms = \
   ....:     ((DVHop,
   ....:           {'dataKey': 'dvData',
   ....:            'truePositionKey':'landmarkPos',
   ....:            'hopsizeKey':'hopsize',
   ....:           }),
   ....:      (Trilaterate,
   ....:          {'dataKey': 'dvData',
   ....:           'positionKey':'dvHop',
   ....:           'truePositionKey':'landmarkPos',
   ....:           'hopsizeKey':'hopsize',
   ....:          }),
   ....:     )
In [7]: sim = Simulation(net)
In [8]: sim.run()
INFO     [simulation.py]: Simulation has finished.
# save network with all relevant data on disk
In [9]: write_npickle(net,'net.gz')
```

Listing 4: Interactive session for preliminary algorithm simulations. First we set up a random network in a default environment. Since in the original article presenting the algorithm there is network with 200 randomly distributed nodes and average degree 9 we set up NetworkGenerator with similar settings to get the appropriate instance of Network. After that, algorithms implemented in the last subsection are instantiated and placed in this new network. Since algorithms require some nodes to know their position (landmarks) we fit first 10 nodes with the appropriate sensor using simple for loop. Network is now ready so after instantiation of new Simulation we can run it. At the end by serializing and storing network object all relevant data is preserved to be analyzed later.

```
In [10]: sim.reset() # first reset network algorithm state and nodes memory
In [11]: sim.run(1) # run 1. step of the first algorithm
In [12]: landmark_node = net.nodes()[0]
In [13]: landmark_node.inbox # landmark receiving initializer message
Out[13]:
[
------ Message ------
     source = None
destination = <Node id=1>
     header = 'initialize'
id(message) = 0x908e9f0>]
In [14]: sim.run(1) # run another step
In [15]: landmark_node.outbox # landmark prepared a broadcast message
Out[15]:
[
------ Message ------
     source = <Node id=1>
destination = Broadcasted
     header = 'Flood'
id(message) = 0x90a3d30>]
In [16]: landmark_node.outbox[0].data
Out[16]: {<Node id=1>: array([201.2419, 141.9482, 1.])}
# check if position being sent in message is one being read by TruePosSensor
In [17]: landmark_node.memory['landmarkPos']
Out[17]: array([201.2419, 141.9482])
# another check by directly inspecting network
In [18]: net.pos[node]
Out[18]: array([201.2419, 141.9482])
```

Listing 5: Example of step-by-step simulation and inspection of nodes and network data. In the first step of the DVHop algorithm spontaneous impulses are issued to landmark nodes in the from of special *initialize* messages. As a consequence in the second step landmark node is broadcasting *Flood* message with its true position and hopcount set to 1. After that we check if this is the same data being read by TruePositionSensor and at the end if that really is the landmark's true position by inspecting network data.

```
netgen = NetworkGenerator(degree=9, n_min=100, n_max=300)
For lm_pct in [5, 10, 20, 33]:
   for net_count in range(100):
      net = netgen.generate()
      for node in net.nodes()[:int(lm_pct*len(net.nodes())/100)]:
         node.compositeSensor = CompositeSensor(('TruePosSensor'))
      net.algorithms = ALGORITHMS
      sim = Simulation(net)
      sim.run()
      write_npickle(net, '%d-%d.gz' % (net_count,lm_pct))
```

Listing 6: Simple automated experiment. For each of the defined landmarks percentages (5%, 10%, 20%, 33%) 100 different networks consisting of 100 to 300 nodes with average degree 9 are created. Each of them is fitted with described algorithms which are then executed in a simulation. At the end of each simulation network data is stored in appropriately named compressed file. Definition of variable ALGORITHMS is omitted for brevity.

is using an *interactive console*. This workflow enables free form experimentation, runtime introspection, direct access, and modification of all objects. Another popular workflow is performing a batch of prepared and *automated experiments*.

In this section examples of using the library in both ways are presented. Specifically, we shall try to implement and simulate Ad hoc Positioning System (APS) with DV-hop propagation—a popular algorithm for node localization proposed by Niculescu and Nath [21].

5.1. Algorithm Definition. In Pymote library, algorithms are defined as direct Python implementations. In our example,

```
# inspect non landmark nodes and list their estimated positions
In [19]: for node in net.nodes()[10:]:
    ....:   print node.memory.get('dvHop', 'Notlocalized.')
[466.04   579.44]
[243.25   80.75]
[386.79   69.09]
[254.92   122.66]
[216.36   80.27]
Not localized.
[431.08   102.28]
[140.36   119.35]
etc...
In [20]: estimated = {}
In [21]: for node in net.nodes():
    ....:   if node.memory.has_key('dvHop'):
    ....:     estimated[node] = node.memory['dvHop']
In [22]: get_rms(net.pos,[estimated])
Out[22]: 32.19781563265385
In [23]: show_localized(net,[estimated],show_labels=False)
```

Listing 7: Analyzing nodes memory after simulation. In dvHop key in memory of all successfully localized nodes, there are estimated coordinates data we can inspect. Using library provided, utilities such as `get_rms` function, root mean square error of estimated positions can be quickly estimated. Finally, errors in estimated positions are visualized graphically. Result of the last command is presented on Figure 5.

APS with DV-hop propagation algorithm is based on the following simple idea. Every node in a network should estimate its distance to as many as possible landmark (anchor, beacon) nodes. Landmark nodes are nodes that a priori know their absolute position. Using the estimated distance and a landmark coordinates, the node should be able to estimate its position using trilateration.

The algorithm has two distinct phases as follows.

(1) All landmark nodes are flooding the network with information about their coordinates. As these messages are propagating hop by hop through the network, every node is increasing hopcounter and maintaining minimum received hopcount from each of the landmarks. During this phase landmarks are calculating average hopsize by dividing known distances from other landmarks with respective hopcounts.

(2) After the first phase is done, every landmark makes another (controlled) flooding with information of new estimated hopsize. After the first reception of hopsize from nearest landmark, the node forwards it and drops all other incoming messages. Then it makes distance estimation and finally estimates its position.

These phases are implemented as two separate instances of NodeAlgorithm subclass. Since both phases are using the same flooding protocol to share information we can define a generic algorithm class for flooding. In Listing 1 there is the implementation of base flooding protocol.

In general, NodeAlgorithm is structured as a set of functions that are corresponding to all possible statuses. Each function should be able to handle all types of incoming messages (defined by header) in a sequence of *if* conditions. Since this is a simple protocol with nodes remaining only in one status "FLOODING" and with only one type of message "Flood", the implementation is neat.

Note that in this algorithm function calls (hooks) occur in three places. These are *initiator_condition*, *initiator_data* and *handle_flood_message*. By subclassing this protocol and implementing missing hooks we can make algorithms for both phases, as presented in Listings 2 and 3.

This behavior is in accordance with established DRY principle. Even better, every other algorithm using flooding protocol can reuse this distributed algorithm solely by implementing its own hooks. Thus, object oriented paradigm native to Python can be exploited even in the distributed algorithm implementation.

5.2. Simulation. The best way to initially test and, if needed, debug newly defined algorithms is done by using interactive console. In Listing 4 one example of such interactive session is presented.

Another advantage of running simulation in an interactive console is that on every exception the user is back at the console. There, the user can issue a `%debug` command to enter a debug console and run commands, inspect, and modify all data in every frame of the program stack that raised the exception.

In addition to continuous execution, algorithms can be executed step by step. This is very useful when we want to analyze algorithm in depth and inspect or modify objects during every step of the simulation. Furthermore, it can be handy in a situation when the algorithm does not execute as expected but the error is not explicit enough to raise an exception. There is an example of such execution in Listing 5.

The second type of workflow is an automated experiment. The automated experiment is defined as a simple Python

script that is generating networks, running simulations on them, and eventually storing network data on disk to be analyzed later. Example of one such script is given in Listing 6.

5.3. Analyze Data. Objects can be inspected and data can be analyzed at any moment. For example, during the network setup phase or after the network is loaded from disk it is convenient to visualize network topology. It can be done by calling show method defined on a network object: `net.show()`. Results are displayed in a new window like the one in Figure 4.

Interactive console can be used to analyze experiment data in more detail as presented in Listing 7 and Figure 5, or to return to investigate execution of different parts of specific experiment.

As a complement to console based workflows, Pymote includes a graphical user interface for step-by-step algorithm simulation and visualization. Interface screenshot is presented in Figure 6. Through this interface, users can

(i) save and load network files from disk;

(ii) run and control simulation step by step with immediate visual feedback, such as messages passing, node status, and so forth;

(iii) inspect different objects including network, nodes, and messages simply by clicking on them;

(iv) customize display of network topology, (edges, labels, etc.);

(v) visualize of custom in-node memory structures such as trees;

(vi) pan and zoom control; and

(vii) save visualized data in many different image formats.

6. Conclusion

In this paper, we have presented Pymote, the library that provides support for simulation and analysis of distributed algorithms built on top of comprehensive Python environment. Pymote is designed to allow rapid interactive testing of new algorithms, their analysis and visualization while minimizing developers time. It supports both interactive algorithm simulation and automation of experiments and provides visualization tools for both. It has been deliberately kept simple, easy to use, and extensible.

We plan to continue developing Pymote environment in several different directions as follows:

(i) further development of graphical user interface functionalities for network setup and data analysis,

(ii) implementation of some of the state of the art algorithms from the set of different fields and related problems, and

(iii) adding support for web based version of setup and simulation execution.

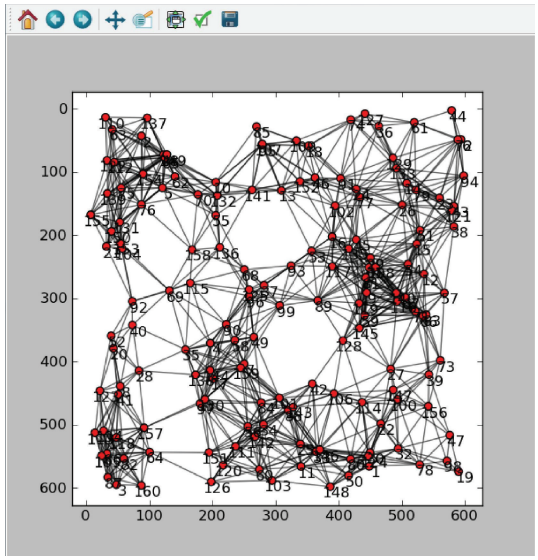

Figure 4: Network topology display.

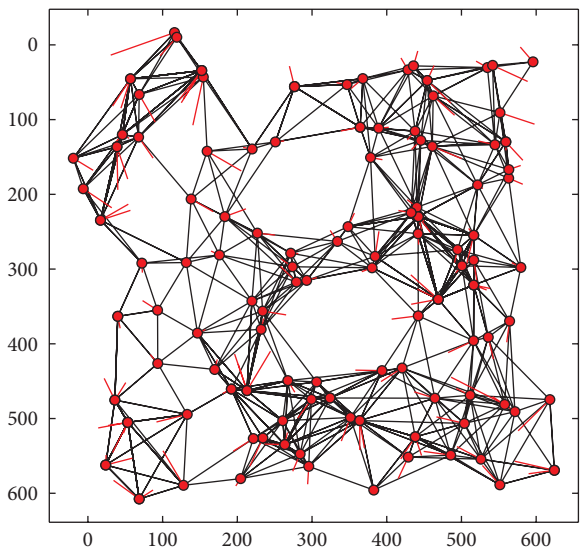

Figure 5: Localization error display.

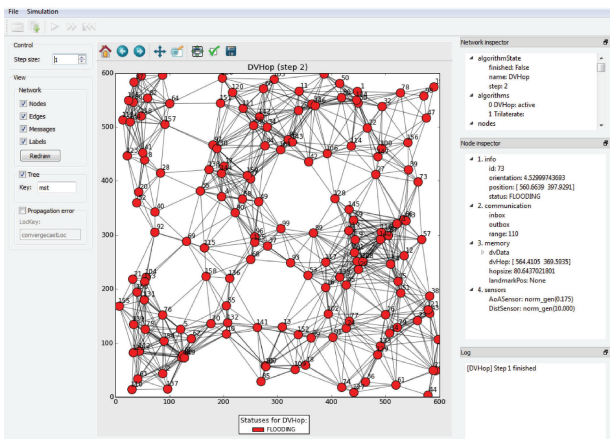

Figure 6: Graphical user interface.

The source code along with the documentation is available as an open source project at https://github.com/darbula/pymote. We hope to have enticed you to take a look at it, use it, or even participate in further development.

References

[1] ns-2, http://www.isi.edu/nsnam/ns/.

[2] I. T. Downard, "Simulating sensor networks in ns-2," Tech. Rep., Naval Research Laboratory, 2004.

[3] Mannasim simulator, http://www.mannasim.dcc.ufmg.br.

[4] Omnet++ simulation system, http://www.omnetpp.org/.

[5] A. Varga, "e omnet++ discrete event simulation system," in *Proceedings of the European Simulation Multi-conference (ESM '01)*, Prague, Czech Republic, June 2001.

[6] A. Boulis, "Castalia: revealing pitfalls in designing distributed algorithms in wsn," in *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems (SenSys '07)*, 2007.

[7] J-Sim, https://sites.google.com/site/jsimofficial/.

[8] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: accurate and scalable simulation of entire TinyOS applications," in *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys '03)*, November 2003.

[9] P. Levis, S. Madden, J. Polastre et al., "Tinyos: an operating system for sensor networks," in *Ambient Intelligence*, W. Weber, J. Rabaey, and E. Aarts, Eds., pp. 115–148, Springer, Berlin, Germany, 2005, http://dx.doi.org/10.1007/3-540-27139-2_7.

[10] H. Sundani, H. Li, V. K. Devabhaktuni, M. Alam, and P. Bhattacharya, "Wireless sensor network simulators: a survey and comparisons," *International Journal of Computer Networks*, vol. 2, no. 6, pp. 249–265, 2011.

[11] E. Egea-Lopez, J. Vales-Alonso, A. S. Martinez-Sala, P. Pavon-Marino, and J. Garcia-Haro, "Simulation tools for wireless sensor networks," in *Proceedings of the Summer Simulation Multiconference (SPECTS '05)*, 2005.

[12] S. Mehta, N. Sulatan, and K. S. Kwak, "Network and system simulation tools for next generation networks: a case study," in *Modelling, Simulation and Identification*, A. Mohamed, Ed., InTech, 2010.

[13] L. Shu, M. Hauswirth, H. C. Chao, M. Chen, and Y. Zhang, "Nettopo: a framework of simulation and visualization for wireless sensor networks," *Ad Hoc Networks*, vol. 9, pp. 799–820, 2011.

[14] Algosensim, http://tcs.unige.ch/doku.php/code/algosensim/overview.

[15] A. Kroeller, D. Pfisterer, C. Buschmann, S. Fekete, and S. Fischer, "Shawn: a new approach to simulating wireless sensor networks," in *Proceedings of the Design, Analysis, and Simulation of Distributed Systems (DASD '05)*, San Diego, Calif, USA, 2005, https://github.com/itm/shawn.

[16] Sinalgo, http://dcg.ethz.ch/projects/sinalgo/.

[17] N. Santoro, *Design and Analysis of Distributed Algorithms*, Parallel and Distributed Computing, John Wiley & Sons, 2007.

[18] F. Pérez and B. E. Granger, "Ipython: a system for interactive scientific computing, computing," *Science and Engineering*, vol. 9, no. 3, pp. 21–29, 2007.

[19] E. Jones, T. Oliphant, P. Peterson et al., "SciPy: open source scientific tools for Python," 2001, http://www.scipy.org/.

[20] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using Net-workX," in *Proceedings of the 7th Python inScience Conference (SciPy '08)*, Pasadena, Calif, USA, August 2008.

[21] D. Niculescu and B. Nath, "DV based positioning in ad hoc networks," *Telecommunication Systems*, vol. 22, no. 1–4, pp. 267–280, 2003.