Python Opportunistic Network Simulator (PONS) Manual

Lars Baumgaertner

2025-07-02

Table of Contents

1. Introduction . . . 1
2. PONS in 10 Minutes . . . 2
   2.1. Installation . . . 2
   2.2. Preparing the Simulation . . . 2
   2.3. Creating The Nodes . . . 3
      2.3.1. Network Setup . . . 3
      2.3.2. DTN Routing . . . 3
      2.3.3. Putting it all together . . . 3
   2.4. Mobility . . . 3
      2.4.1. Sending Messages . . . 4
      2.4.2. Simulation Setup . . . 4
   2.5. Everything combined . . . 5
3. Architecture . . . 8
4. Network . . . 10
5. Mobility . . . 11
6. Routing . . . 12
7. Applications . . . 13
8. Logging . . . 14
9. Analyzing Results . . . 15
10. Tools . . . 16
   10.1. netedit . . . 16
   10.2. ponsanim . . . 16
   10.3. scenariorunner . . . 16

Chapter 1. Introduction

PONS is a Python-based simulator for opportunistic networks, designed to facilitate the study and development of delay-tolerant networking protocols and applications. It provides a flexible environment for simulating various network topologies, mobility patterns, and routing algorithms.

Chapter 2. PONS in 10 Minutes

In this example, we will learn how to install PONS, create a bunch of nodes, connect them via a pseudo-WLAN and how them walk around randomly.

2.1. Installation

PONS is available via pip in the `pons-dtn` package. If you want mp4 support for `ponsanim`, the event log visualizer, you also need to install `opencv-python` and optionally `ffmpeg` for your OS.

The recommended setup procedure for a typical PONS project looks as following:

```
$ mkdir my_netsim
$ cd my_netsim
$ python3 -m venv .venv
$ . .venv/bin/activate
$ pip3 install pons-dtn opencv-python
```

2.2. Preparing the Simulation

For each simulation we should add some imports and define constants for various settings.

```
import random
import json

import pons
import pons.routing

# A random seed, so we get reproducible results
RANDOM_SEED = 42
# The simulation time in seconds
SIM_TIME = 3600 * 24
# For our pseudo-WLAN we need a maximum communication range in meters
NET_RANGE = 50
# The number of nodes we want to simulate
NUM_NODES = 10
# Our virtual world should be 1000m x 1000m
WORLD_SIZE = (1000, 1000)
# Each node should have a store capacity of 10000 bytes
CAPACITY = 10000

random.seed(RANDOM_SEED)
```

A world size is only needed, if a movement model is used, for fixed topologies with contact windows the world size can be set to `NONE`. If the capacity is set to 0, the storage capacity of each node is unlimited.

2.3. Creating The Nodes

Nodes in PONS have a few core components:

- One or more network interfaces
- A DTN router
- A name and unique node number

2.3.1. Network Setup

Each node can have multiple network interfaces but for our example we only need a single one simulating a primitive distance-based WLAN.

```
net = pons.NetworkSettings("WIFI_50m", range=NET_RANGE)
```

We configure the network with the chosen identifier "WIFI_50m" and our preset range. Settings for *bandwidth* (default: 54MBit/s), *delay*, *loss* or an optional *contactplan* are left deactivated respectively defaults.

2.3.2. DTN Routing

PONS comes preconfigured with various DTN routing algorithms, all derived from the base **Router** class. For the purpose of this tutorial we are going to use standard epidemic routing with a limited store capacity.

```
epidemic = pons.routing.EpidemicRouter(capacity=CAPACITY)
```

2.3.3. Putting it all together

In PONS, we have a helper function to generate a large number of nodes from a few template parameters and automatically name them and provide them with their own copies of network and router objects.

```
nodes = pons.generate_nodes(NUM_NODES, net=[net], router=epidemic)
```

2.4. Mobility

Mobility in PONS is optional, but for this example we want to generate movements to randomly chosen waypoints using the internal movement generator.

```
moves = pons.generate_randomwaypoint_movement(
    SIM_TIME, NUM_NODES, WORLD_SIZE[0], WORLD_SIZE[1], max_pause=60.0
)
```

Here, we generate for all nodes and the whole simulation time movements with a maximum pause time for moving nodes of 60s. More parameters can be configured as described in the mobility section of this user guide.

2.4.1. Sending Messages

Messages or bundles should be sent either through an event generator or simulated apps running on top of the node routers. In this example, we will be using a **single** node message event generator.

```
msggenconfig = {
    "type": "single",
    "interval": 30,
    "src": (0, NUM_NODES),
    "dst": (0, NUM_NODES),
    "size": 100,
    "id": "M",
    "ttl": 3600,
}
```

Each simulation can contain multiple message generators. They can be either of type **single** or **bulk**, meaning at a given interval either a randomly chosen source node or all source nodes send a message. Here, we want a single node to be the source in 30s intervals. The source should be randomly chosen between **all** nodes and the destination can also be any node in the range. The message should have a fixed size of 100 bytes, a time-to-live of 3600s and the messages should be prefixed with 'M'.

2.4.2. Simulation Setup

A `NetSim` is used to create our little simulation world. There are various logging options one can set, which either log to **stdout** or to an event log file.

```
config = {"movement_logger": False, "peers_logger": False, "event_logging": False}

netsim = pons.NetSim(
    SIM_TIME,
    nodes,
    world_size=WORLD_SIZE,
    movements=moves,
    config=config,
    msggens=[msggenconfig],
)

netsim.setup()

netsim.run()

# The low level network statistics including the node discovery messages
```

```
print(json.dumps(netsim.net_stats, indent=4))

# The actual DTN statistics from the routers
print(json.dumps(netsim.routing_stats, indent=4))
```

Prior to running the simulation (`netsim.run()`), we have to setup everything using `netsim.setup()`. Here, all routers get initialized, initial node positions and network maps are loaded/calculated, and other important preparations.

Afterwards, we print some low level statistics from the network and routing layers.

2.5. Everything combined

All pieces combined, we get a small DTN network simulation as follows:

```
import random
import json

import pons
import pons.routing

# A random seed, so we get reproducible results
RANDOM_SEED = 42
# The simulation time in seconds
SIM_TIME = 3600 * 24
# For our pseudo-WLAN we need a maximum communication range in meters
NET_RANGE = 50
# The number of nodes we want to simulate
NUM_NODES = 10
# Our virtual world should be 1000m x 1000m
WORLD_SIZE = (1000, 1000)
# Each node should have a store capacity of 10000 bytes
CAPACITY = 10000

random.seed(RANDOM_SEED)

net = pons.NetworkSettings("WIFI_50m", range=NET_RANGE)
epidemic = pons.routing.EpidemicRouter(capacity=CAPACITY)

nodes = pons.generate_nodes(NUM_NODES, net=[net], router=epidemic)

moves = pons.generate_randomwaypoint_movement(
    SIM_TIME, NUM_NODES, WORLD_SIZE[0], WORLD_SIZE[1], max_pause=60.0
)

msggenconfig = {
    "type": "single",
    "interval": 30,
    "src": (0, NUM_NODES),
```

```
    "dst": (0, NUM_NODES),
    "size": 100,
    "id": "M",
    "ttl": 3600,
}

config = {"movement_logger": False, "peers_logger": False, "event_logging": False}

netsim = pons.NetSim(
    SIM_TIME,
    nodes,
    world_size=WORLD_SIZE,
    movements=moves,
    config=config,
    msggens=[msggenconfig],
)

netsim.setup()

netsim.run()

# The low level network statistics including the node discovery messages
print(json.dumps(netsim.net_stats, indent=4))

# The actual DTN statistics from the routers
print(json.dumps(netsim.routing_stats, indent=4))
```

When running the simulation, we get some status output and a progress bar, while the simulation is executing. In the end, we get some statistics about simulation.

```
$ python3 pons-tutorial.py
initialize simulation:  {'movement_logger': False, 'peers_logger': False,
'event_logging': False}
-> start movement manager
{0: <pons.node.Node object at 0x102df0150>, 1: <pons.node.Node object at 0x10568fb90>,
2: <pons.node.Node object at 0x1056c1090>, 3: <pons.node.Node object at 0x1056c1410>,
4: <pons.node.Node object at 0x1056c1790>, 5: <pons.node.Node object at 0x1056c1b50>,
6: <pons.node.Node object at 0x1056c1ed0>, 7: <pons.node.Node object at 0x1056c2250>,
8: <pons.node.Node object at 0x1056c25d0>, 9: <pons.node.Node object at 0x1056c2910>}
== running simulation for 86400 seconds ==
global number of unique contact plans:  0

start message generator
Progress: |                                                                                | 100.0% Complete

simulation finished
simulated 86401 seconds in 6.51 seconds (13275.43 x real time)
real: 6.508340, sim: 86401 rate: 13275.43 steps/s
{
    "tx": 126232,
```

```
    "rx": 124604,
    "drop": 1628,
    "loss": 0
}
{
    "created": 2880,
    "delivered": 2595,
    "dropped": 0,
    "started": 88668,
    "relayed": 88648,
    "removed": 0,
    "aborted": 0,
    "dups": 66044,
    "latency_avg": 777.5032203836079,
    "delivery_prob": 0.9010416666666666,
    "hops_avg": 2.154527938342967,
    "overhead_ratio": 33.161078998073215
}
```


The statistics are useful when evaluating, e.g., different routing algorithms or buffer priorities. Note, the network rx/tx stats are much higher than the routing ones. This is due to the fact, that each node does a peer discovery scan (default: every 2s) using the nodes network simulation, thus, generating some extra non-dtn network traffic.

Chapter 3. Architecture

PONS consists of several components that work together to simulate opportunistic networks. The architecture is modular, allowing for easy extension and customization. An overview of architecture can be found in Figure 1.

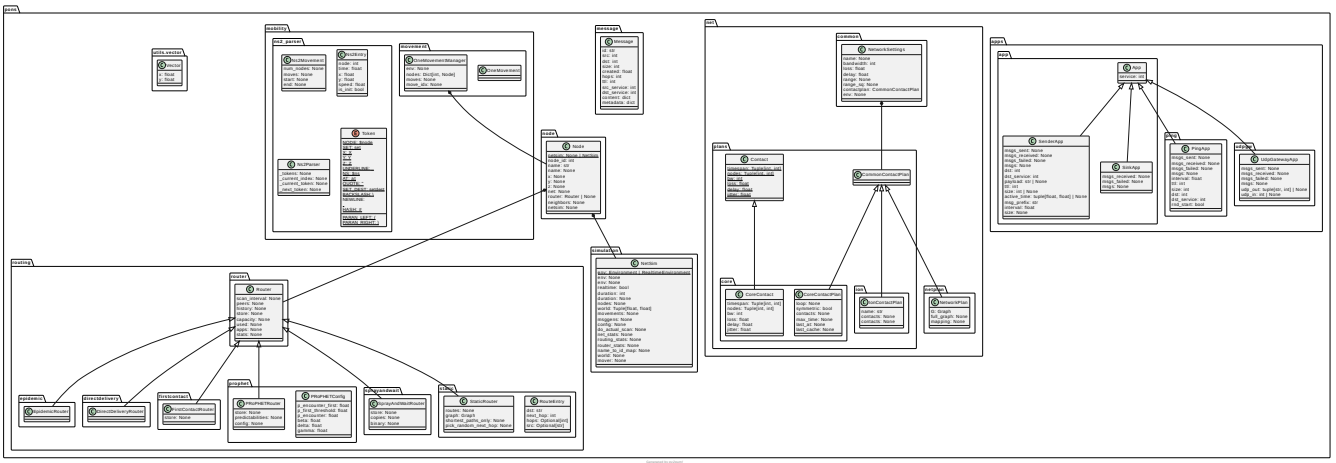


Figure 1. Full Architecture Diagram of PONS

The main components of a PONS simulation include:

- **Networks**: Represents the simulated network environment, including links, and their properties. Links can be static, range-based, requiring a mobility model, or contact-plan-based, allowing for different types of connectivity between nodes.
- **Mobility**: Manages the movement of nodes within the network, simulating realistic mobility patterns.
- **Routing**: Implements various routing protocols to facilitate message delivery across the network.
- **Applications**: Simulates applications running on nodes, generating and processing messages

Each node is designed with three layers:

- **Network Layer**: Handles the network interface and connectivity, including the transmission of messages over the network.
- **Routing Layer**: Implements the routing protocol, managing message storage, forwarding, and delivery.
- **Application Layer**: Represents the application logic, generating messages and processing incoming messages.

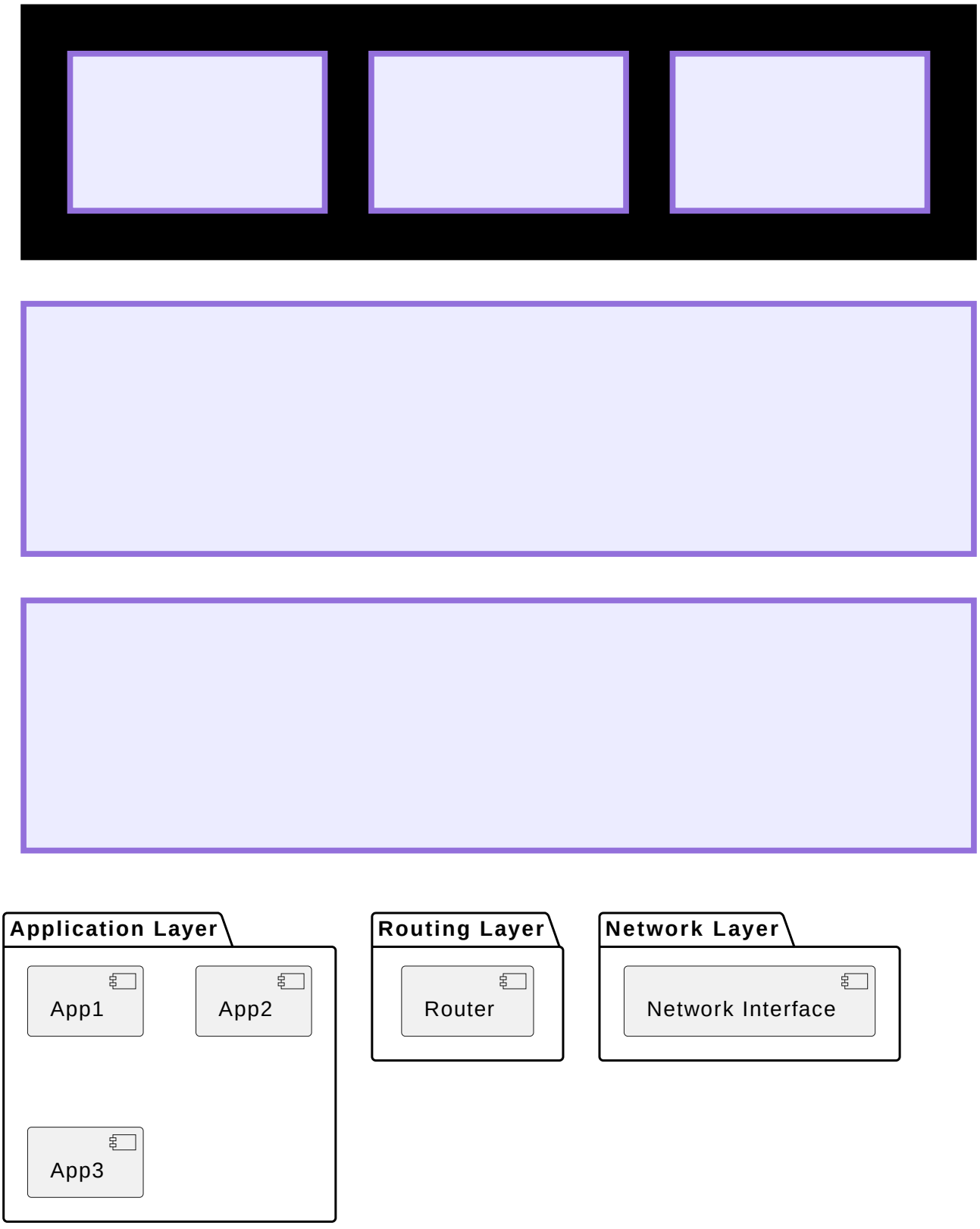
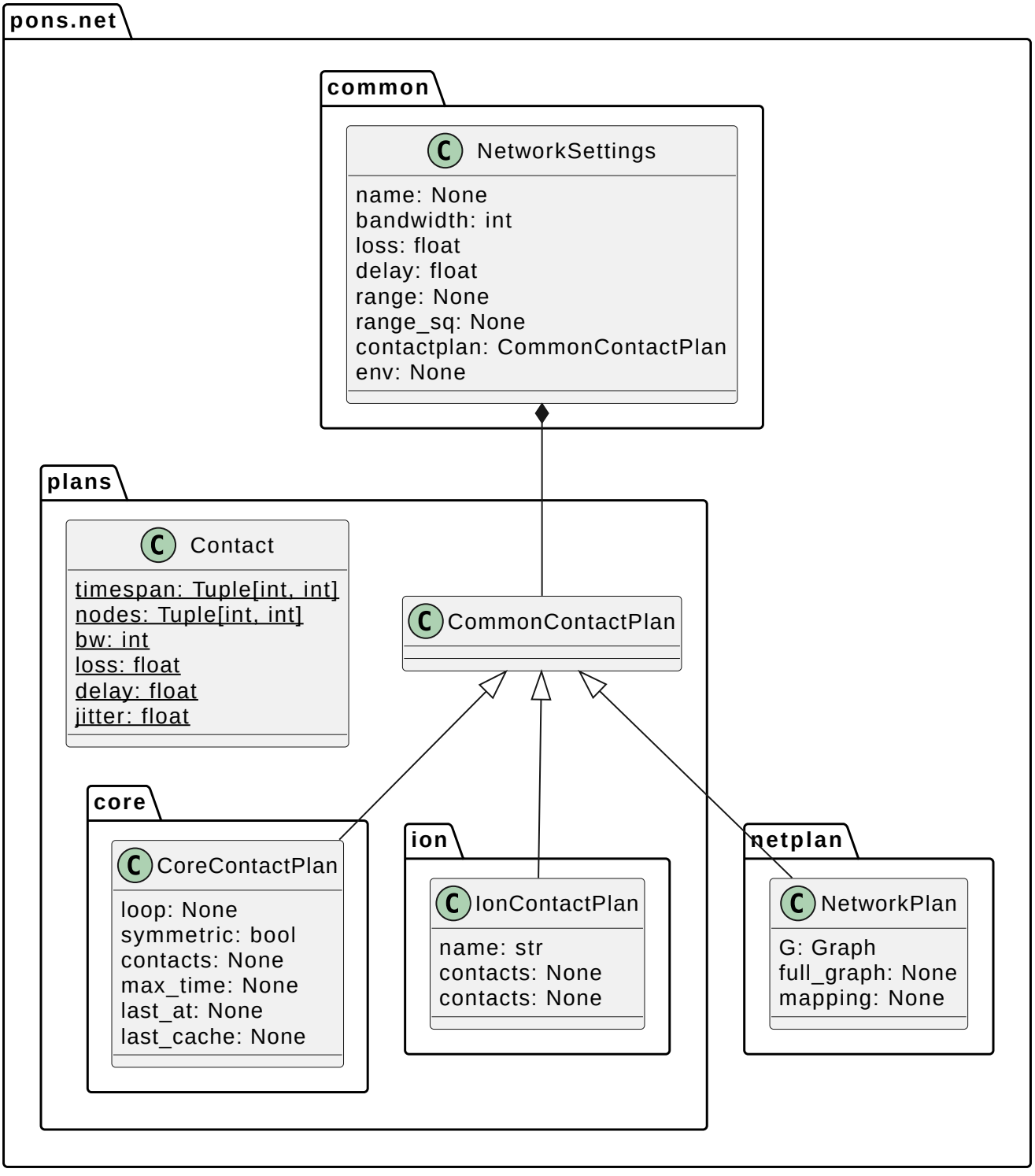


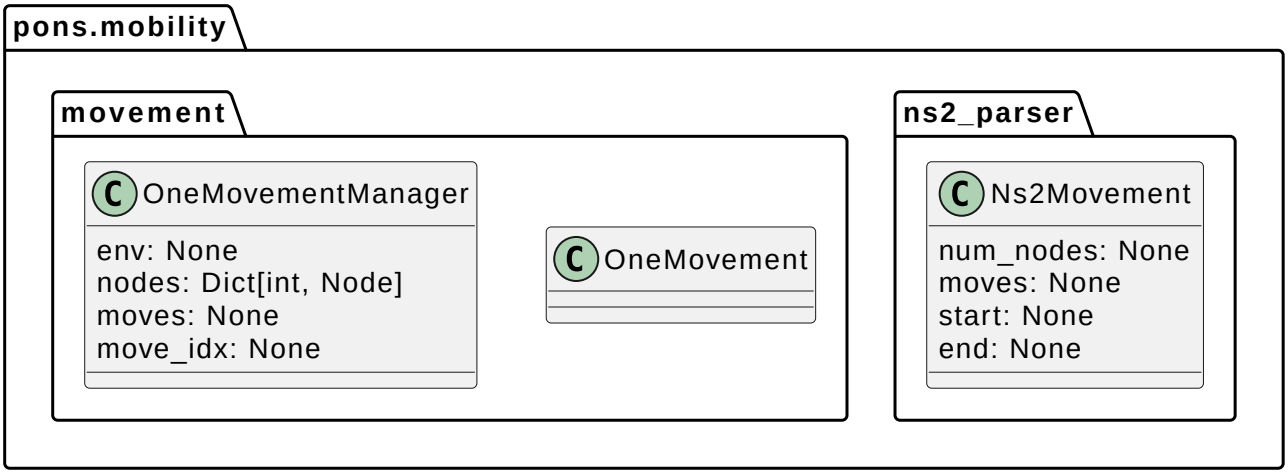
Figure 2. Three layer architecture of a PONS node

Chapter 4. Network



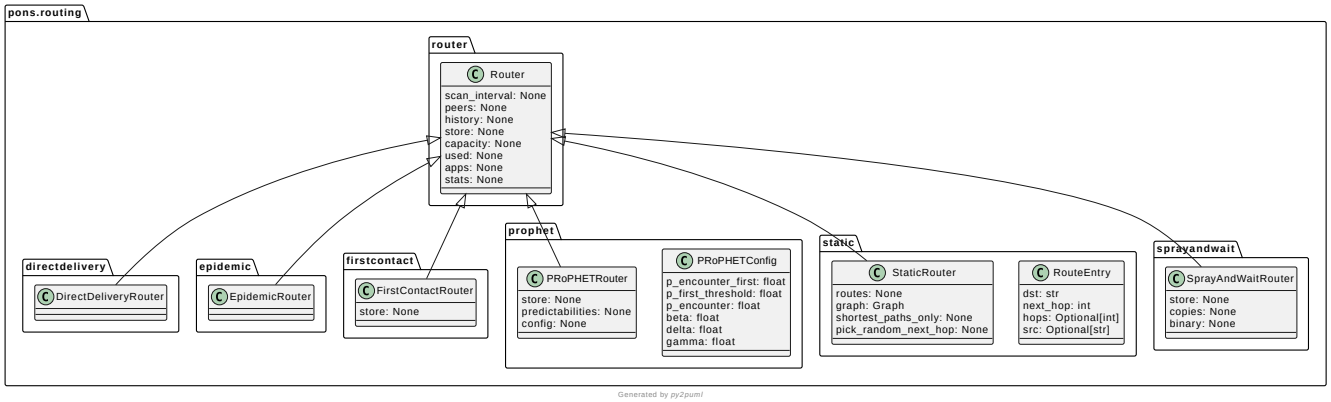
Generated by *py2puml*

Chapter 5. Mobility



Generated by *py2puml*

Chapter 6. Routing



Chapter 7. Applications

Chapter 8. Logging

Chapter 9. Analyzing Results

Chapter 10. Tools

10.1. netedit

10.2. ponsanim

10.3. scenariorunner