Python Opportunistic Network Simulator (PONS) Manual

Lars Baumgaertner

2025-07-02

Table of Contents

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.

2.2.1. 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

2.2.2. 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.

2.2.3. 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)
```

2.2.4. 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)
```

2.2.5. 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.

2.2.6. 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** 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'.

2.2.7. 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.

2.3. 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
2025-07-20 13:44:01,177 - pons.simulation - INFO - Initializing simulation:
{'movement_logger': False, 'peers_logger': False, 'event_logging': False}
2025-07-20 13:44:01,177 - pons.simulation - INFO - ENV LOG_FILE found! Activating
event logging using log file: tutorial.log
2025-07-20 13:44:01,177 - pons.simulation - INFO - == running simulation for 86400
seconds ==

Progress: |██████████████████████████████████████████████████| 100.0% Complete



2025-07-20 13:44:19,077 - pons.simulation - INFO - simulation finished
2025-07-20 13:44:19,079 - pons.simulation - INFO - simulated 86401 seconds in 17.90
seconds (4827.05 x real time)
2025-07-20 13:44:19,079 - pons.simulation - INFO - real: 17.899320, sim: 86401 rate:
4827.05 steps/s
{
    "tx": 81628,
    "rx": 81264,
    "drop": 364,
```

```
    "loss": 0
}
{
    "created": 2880,
    "delivered": 2561,
    "dropped": 0,
    "started": 81718,
    "relayed": 81264,
    "removed": 22055,
    "aborted": 364,
    "dups": 58933,
    "latency_avg": 873.8002361692731,
    "delivery_prob": 0.8892361111111111,
    "hops_avg": 2.1144084342053886,
    "overhead_ratio": 30.731354939476766
}
```

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.

2.4. Running the Example

To run the example, save the code above in a file called `pons-tutorial.py` and execute it in your terminal:

```
$ LOG_LEVEL=info LOG_FILE=/tmp/tutorial.log python3 pons-tutorial.py
```

You can also use the `ponsanim` tool to visualize the simulation results, which will create an mp4 video of the simulation. Make sure you have `imageio[ffmpeg]` installed for video encoding.

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.

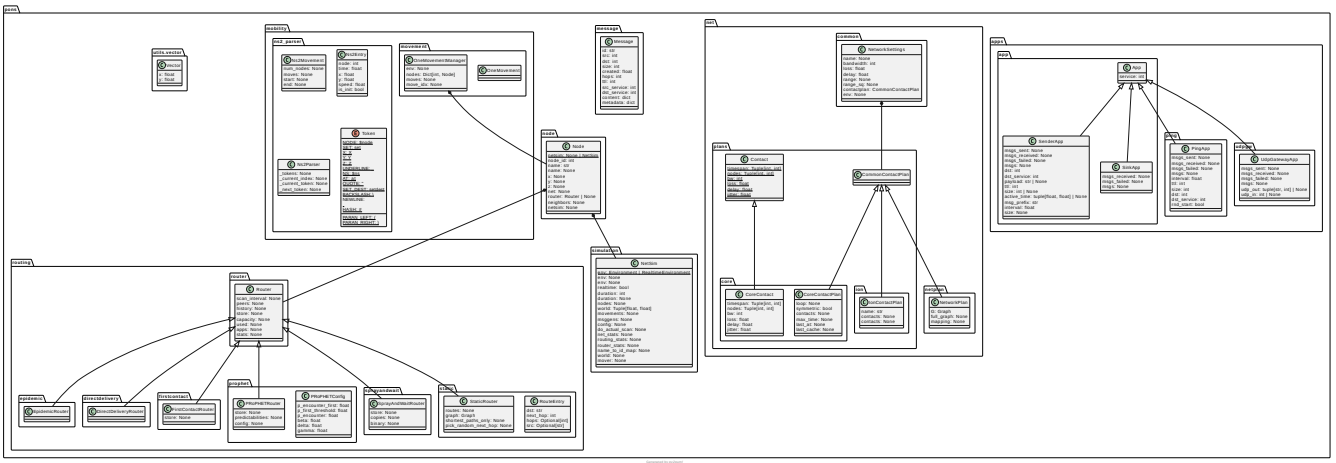


Figure 1. Full Architecture Diagram of PONS

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. More details can be found in the Chapter 5.
- **Mobility**: Manages the (optional) movement of nodes within the network, simulating realistic mobility patterns. More details can be found in the Section 2.2.5.
- **Routing**: Implements various routing protocols to facilitate message delivery across the network. More details can be found in the Chapter 7.
- **Applications**: Simulates applications running on nodes, generating and processing messages. More details can be found in the Chapter 8.

Each node is designed with three layers:

- **Network Layer**: Handles the network interface and connectivity, including the transmission of messages over the network. This layer is fixed for each node and can be configured with different network settings, such as range, bandwidth, delay, and loss or contact plans.
- **Routing Layer**: Implements the routing protocol, managing message storage, forwarding, and delivery. Each node can have exactly one routing algorithm instance, which is responsible for the routing logic. PONS provides several built-in routing algorithms, such as Epidemic, Spray and Wait, and others. The routing layer interacts with the network layer to send and receive messages.
- **Application Layer**: Represents the application logic, generating messages and processing incoming messages. Each node can have multiple applications, which can be used to simulate different behaviors and interactions. Applications can be used to send messages, receive messages, and perform other tasks related to the simulation.

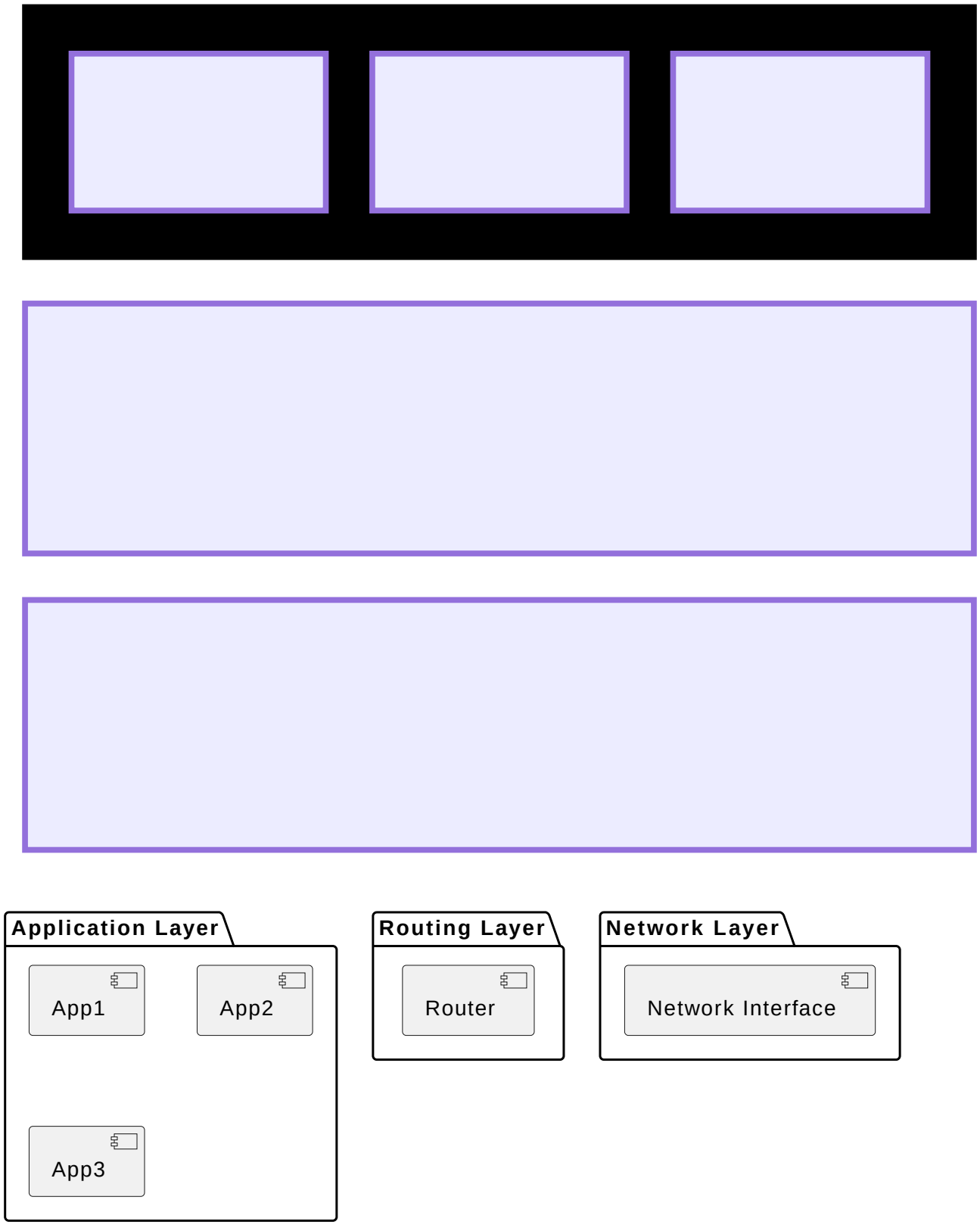


Figure 2. Three layer architecture of a PONS node

The default mode of operation for PONS is non-interactive, meaning that the simulation runs without user intervention. Therefore, it should be much faster than realtime. However, PONS also supports an interactive mode, where everything is running in realtime and external applications can interact with the simulation. This mode is useful for live demonstrators and testing client applications in various scenarios, but it is not recommended for large-scale simulations due to performance limitations.

PONS provides built-in logging capabilities to record simulation events, messages, and routing decisions. This data can be analyzed to evaluate the performance of routing protocols, application behavior, and network dynamics. The logs can be processed using various tools and scripts to extract meaningful insights and statistics from the simulation results. To activate it set `LOG_FILE` environment variable to a file path. Furthermore, by setting `LOG_LEVEL` environment variable to `DEBUG`, `INFO`, `WARNING`, `ERROR`, or `CRITICAL`, you can control the verbosity of the internal logging output. The default is `INFO`.

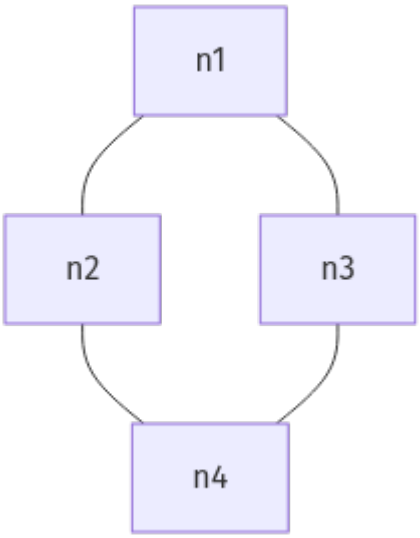
In addition to the core components, PONS provides several tools and utilities to facilitate the simulation process, including:

- **netedit**: A graphical editor for creating and modifying network topologies, mobility patterns, and routing configurations. It allows users to visualize and edit the simulation environment interactively.
- **ponsanim**: A tool for visualizing the simulation results, including node movements, message exchanges, and routing behavior. It provides a graphical representation of the simulation, making it easier to analyze and understand the results.
- **scenariorunner**: A command-line tool for running PONS simulations with predefined scenarios. It allows users to quickly set up and execute simulations with specific parameters, making it easier to test different configurations and compare results.

Chapter 4. Topology

There are multiple ways to position nodes in PONS. They can either be set manually or loaded from an existing graph structure. Alternatively, if you use an external movement model such as **ONE** or **ns2**, these will set node positions accordingly.

For the rest of this section, we will look at different ways to create the following topology:



4.1. Manual Node Positions

Each node has `.x`, `.y` and `.z` values as floats that can be manipulated after creating the node.

```
n1 = pons.Node(1, "n1",
  net=[pons.NetworkSettings("WIFI_40m", range=40)],
  router=pons.routing.EpidemicRouter(capacity=CAPACITY)
)
n1.x = 50
n1.y = 50

n2 = pons.Node(1, "n2",
  net=[pons.NetworkSettings("WIFI_40m", range=40)],
  router=pons.routing.EpidemicRouter(capacity=CAPACITY)
)
n2.x = 25
n2.y = 75

n3 = pons.Node(1, "n3",
  net=[pons.NetworkSettings("WIFI_40m", range=40)],
  router=pons.routing.EpidemicRouter(capacity=CAPACITY)
)
n3.x = 75
n3.y = 75

n4 = pons.Node(1, "n4",
```

```
  net=[pons.NetworkSettings("WIFI_40m", range=40)],
  router=pons.routing.EpidemicRouter(capacity=CAPACITY)
)
n4.x = 50
n4.y = 100
```

We now created all 4 nodes with an epidemic router and a WLAN network with 40 meters of range. Thus, n1 can barely reach n2 and n3 but not n4. Alternatively, a contact plan could be used to provide fixed connections, see the following examples on how to build such topologies.

4.2. NetworkX Graph

If actual positions are not as important as the general network graph, it is very convenient to use [networkx](https://networkx.org/) for modelling the network.

```
topo = nx.graph.Graph()
topo.add_node(1)
topo.add_node(2)
topo.add_node(3)
topo.add_node(4)
topo.add_edge(1, 2)
topo.add_edge(1, 3)
topo.add_edge(2, 4)
topo.add_edge(3, 4)

plan = pons.net.NetworkPlan(topo)
net = pons.NetworkSettings("networkplan", range=0, contactplan=plan)

nodes = pons.generate_nodes_from_graph(
    topo,
    router=pons.routing.EpidemicRouter(capacity=CAPACITY),
    net=[net],
)
```

We can use the `NetworkPlan` class to derive a contact plan for all graph edges. This can be supplied to our `NetworkSettings`, a `range` is not needed anymore as all links are fixed through the network contact plan.

Similar to the `generate_nodes` helper from the [tutorial](example.md), there is a helper to generate nodes from a given graph object that we can use (`generate_nodes_from_graph`).

4.2.1. Generating random topologies

The build in functions of `networkx` make it pretty easy to quickly generate random graphs that can be used as network topologies.

```
topo1 = nx.erdos_renyi_graph(NUM_NODES, 0.3)
```

```
topo2 = nx.watts_strogatz_graph(NUM_NODES, 2, 0.3)
```

They can then just be used as the manually created one in the previous example.

4.2.2. GraphML

Building complex topologies manually can be cumbersome, therefore, PONS supports of loading graphml files with node information. These can be generated by external tools. Graphml is very powerful and depending on the tools can have additional extra information. If using [netedit`](tools/netedit.md) to generate the graphml file one can also set the node **ID**, **name** and coordinates (**x** and **y**).

Loading a graphml file as a contact plan and creating the nodes from it is pretty straight forward:

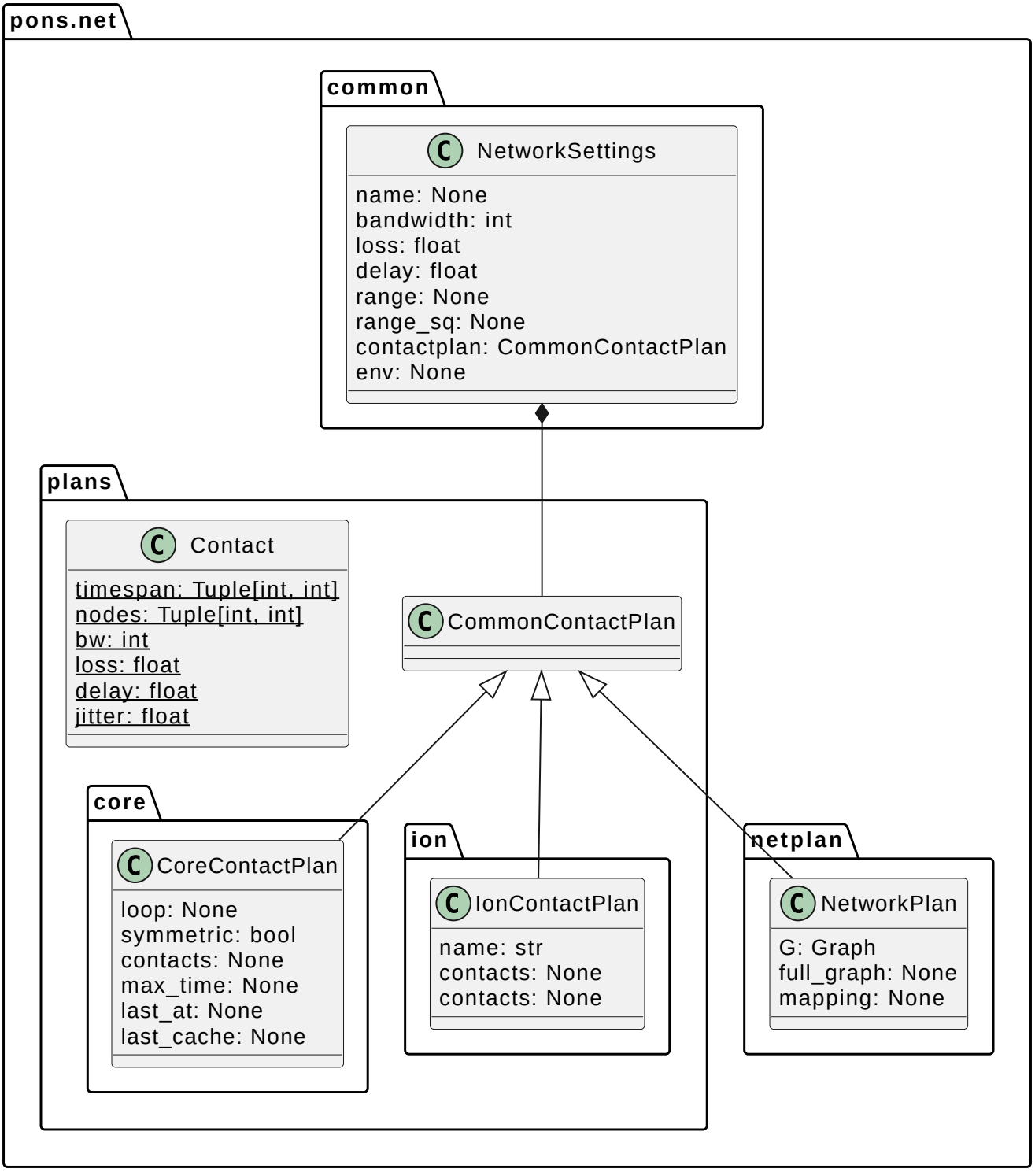
```
plan = pons.net.NetworkPlan.from_graphml("4n-diamond.graphml")

print(plan.nodes())
print(plan.connections())

net = pons.NetworkSettings("networkplan", range=0, contactplan=plan)

nodes = pons.generate_nodes_from_graph(
  plan.G,
  router=pons.routing.EpidemicRouter(),
  contactplan=plan
)
```

Chapter 5. Network



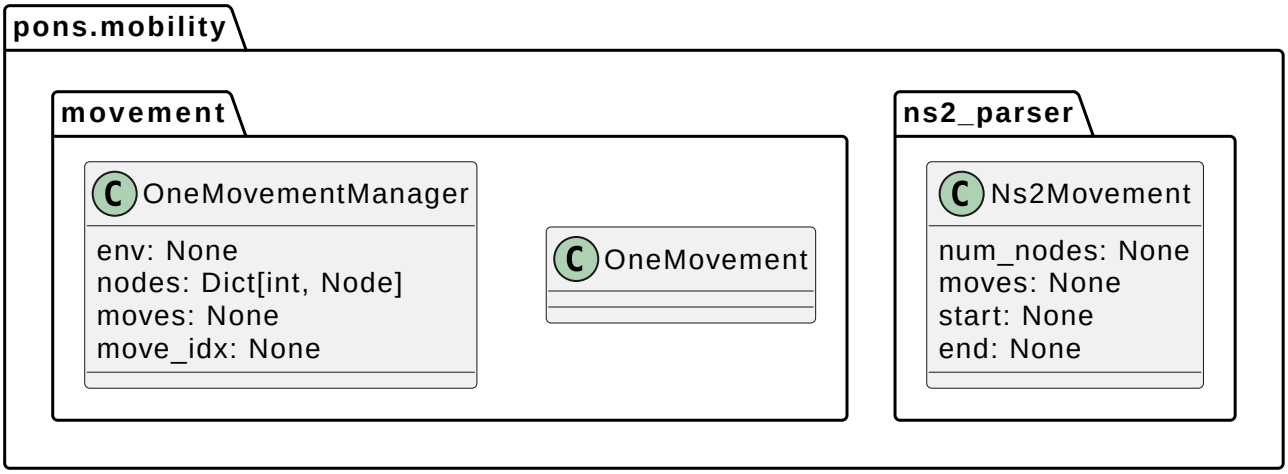
Generated by *py2puml*

PONS supports various different types of contact plans. All have in common that they allow one to get a list of active contacts at a given time and return the potential transmission time needed for a transfer. There are three main types of contact plans:

- **Netplan**: A contact plan derived from a networkx graph, describing the topology. This is useful for scenarios where large parts of the network topology is known in advance and does not change during the simulation. It can be merged with another more dynamic contact plan to add additional contacts or modify existing ones.

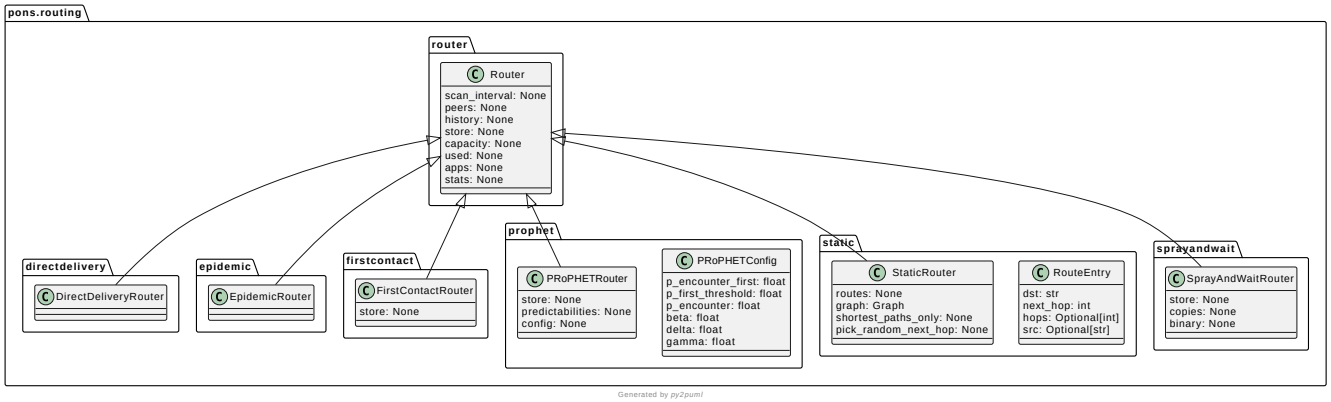
- **Core Contact Plan**: Based on the format of the *Core Contact Plan Manager*. It defines a set of contacts between nodes based on a fixed schedule, but with additional features like bandwidth limitations or delays and jitter. It defines a set of contacts between nodes that are based on a fixed schedule.
- **Ion Contact Plan**: Similar to the **Core Contact Plan**, but based on the format of the *ION DTN* for its contact plans.

Chapter 6. Mobility



Generated by *py2puml*

Chapter 7. Routing



Chapter 8. Applications

Chapter 9. Logging

Chapter 10. Analyzing Results

Chapter 11. Tools

11.1. netedit

11.2. ponsanim

11.3. scenariorunner

11.4. plot-contacts