pyDTNsim Documentation

Release 0.1.0

Robert Wiewel

Feb 11, 2019

User Guide

1 Contents **3**

1.1 Installation . . . 3
1.2 Getting started with pyDTNsim . . . 5
1.3 Examples . . . 12
1.4 Simulation Scenario Setup . . . 12
1.5 Routing Mechanisms . . . 12
1.6 Simulation Monitoring . . . 12
1.7 Simulation Data Processing . . . 12
1.8 pydtnsim package . . . 12
1.9 Architecture Overview . . . 14
1.10 Development Guide . . . 14

2 Indices and tables **15**

pyDTNsim provides a simulation environment that allows the simulation of arbitrary Delay Tolerant Networking (DTN) scenarios on a packet level. *pyDTNsim* provides users with the ability to evaluate the performance of various routing approaches and to detect possibly occurring overload situations. Currently, the focus lies on deterministic Contact-Graph-based routing approaches, but there might be other approaches available in the future. The clear modularization allows users to easily implement routing approaches on their own.

CHAPTER 1

Contents

1.1 Installation

Depending on the intended use case, the installation procedure for the library differs. If the library modules should only be invoked by custom scripts, installing `pyDTNsim` via the Python package index PyPi is sufficient. If the simulation modules have to be altered, downloading the source code and installing `pyDTNsim` as editable python module is required.

1.1.1 Module Dependencies

The following modules are used in various contexts of `pyDTNsim`. However, some are only necessary for the development of the module and not for running simulations.

Python Module	License	Purpose	Dev?[1]
networkx	BSD	Library allows the export of networkx `DiGraph` objects for additional graph analyses.	
tqdm	MIT	Used for displaying the simulation progress (i.e. elapsed simulated seconds).	
jsonschema	MIT	Validation of JSON schemes in the loaded topology files.	
pytest	MIT	Used for running the modules unit tests. Only executed in CI, not integrated otherwise.	x
sphinx	BSD	Generation of this documentation. Not invoked by the module.	x
sphinx_rtd_theme	MIT	Theme for `sphinx`.	x
pylint	GPL	Tool for detecting source code issues. Only executed in CI, not integrated otherwise.	x
pydocstyle	MIT	Tool for validating docstrings in source code.	x
termcolor	MIT	Provides colorful shell output when testing the examples of the module.	x

pyDTNsim uses new Python features, in particular

[1] Modules with 'Dev?' checked are only relevant in the development context of this module.

- Data Classes (3.7),
- Formatted String Literals (3.6) and
- Insertion-Order Preservation of items added to a `dict` object. (3.7).

Therefore, **Python 3.7+** is currently required for using this library.

Note: It is planned to establish compatibility with older versions (especially 2.7) in the future.

1.1.2 PyPi Installation

The latest version of *pyDTNsim* can be installed with `pip3`:

```
$ pip3 install pydtnsim
```

Thats it, `pip3` will download the module from PyPi and install it locally. Check if the module was installed correctly by invoking a Python shell and importing the module:

```
> import pydtnsim
```

If no error occurs, the installation was successful. Continue with the with the section *Getting started with pyDTNsim*.

1.1.3 Source Code Installation

Alternatively, the module can be made available in the local Python instance by downloading it from Github and then installing it as editable package.

This more advanced installation is necessary when the library module (or parts of it) have to be altered. For example, this is the case if contributing to the module is intended.

Archive Download and Extraction

The source can be downloaded as .zip or .tar.gz archive.

In Linux, the download and extraction of the files can usually also be achieved using the utilities `wget` and `tar`:

```
$ wget https://github.com/ducktec/pydtnsim/archive/master.tar.gz
$ tar -xzf master.tar.gz
$ cd pydtnsim-master/
```

Git Clone

If the version-control system `git` is installed, the project can also be cloned:

```
$ git clone https://github.com/ducktec/pydtnsim.git
$ cd pydtnsim/
```

Module Installation

Warning: Please store the `pydtnsim/` source code folder in an appropriate (long-term) directory on your local device. As we are installing the module as editable, the Python environment will continuously reference the files directly instead of copying them to hidden internal folders. Moving the directory around after the installation will likely result in broken references and errors!

As next step, the module can be made available in the Python environment:

```
$ pip install -e "."
```

`pip3` installs the module as editable (achieved with the parameter `-e`) and tries to satisfy all core dependencies (see above *dependency table*).

If all development dependencies[1] shall be installed, the `[dev]` specifier has to be added to the installation command:

```
$ pip install -e ".[dev]"
```

Check if the module was installed correctly by invoking a Python shell and importing the module:

```
> import pydtnsim
```

If no error occurs, the installation was successful. Continue ith the section *Getting started with pyDTNsim*.

Tip: A recommendation in general, but a must in the context of upstream development is the use of the environment wrappers pyenv (for the version of the Python interpreter) and pipenv (for the Pip package environment). The configuration files for *pipenv* are provided (`Pipfile` and `Pipfile.lock`) and contain also the requirements for *pyenv*.

In combination, the wrappers ensure a sound and deterministic development environment for all developers by installing clearly specified versions and encapsulating them from the other python (package) installations on the systems. An article outlining the benefits of the wrappers can be found at[2].

1.2 Getting started with pyDTNsim

With *pyDTNsim* installed, we can start with running simulations using the library. In this section, a hands-on introduction into the features and the instrumentation procedure of the library module will be provided.

The goal of this introduction is to simulate the simple intermittently connected network topology as depicted below and to generate key characteristics of this simulation run.

1.2.1 Simulation Scenario

The following network topology shall be simulated for 1000 seconds:

The annotations at the arrows represent available contact between the two (physical) network nodes that an arrow is connecting in mathematical interval notation in seconds. All contacts are considered to allow for a transmission of data with 100 KBps.

[2] https://hackernoon.com/reaching-python-development-nirvana-bb5692adf30c?gi=26b62f02bc0b

Note: The propagation delays are considered neglible in this scenario. This is in line with the current configuration of *pyDTNsim* which is not supporting (individual or global) delays at the moment.

The nodes `A` and `C` are representing *"active"* endpoints which are both continuously inserting packets of 100 KB with a data generation rate of 10 KBps, addressed at node `A` and `C` respectively. The data generation will continue throughout the entire simulation period.

Node `B` is functioning as intermediary node that is solely forwarding packets received from `A` and `B`. It is neither the destination of any packets nor is it injecting any packets.

Contact Graph Routing (CGR) will be used as routing mechanism. See *Routing Mechanisms* for more details on provided mechanisms and their implementation.

The characteristics that should be acquired with the simulation run are

- the overall **average delivery time** of all delivered packets during the simulation run,
- the **number of packets** enqueued into the **limbo** (i.e., packets that could not be scheduled for transmission with CGR) and
- a **histogram** of the **average delivery time** of all delivered packets throughout the simulation run.

1.2.2 Creating a simulation script

As *pyDTNsim* is a library module, we have to create a simulation script ourselves to leverage the invoke the module's functionality.

Just create a new python script file with your favorite editor or type

```
touch dtn_simulation.py
vim dtn_simulation.py
```

With the script created, we can now start to import the libraries components. We start with creating a `pydtnsim.simulator.Simulator` object. This object represents the event-oriented simulation environment that keeps track of the simulations components and is later invoked for the actual simulation run. Details about the abstract concept of the simulation environment can be found in *Architecture Overview*.

The `Simulator` can then be used to perform a simulation using it's member function `Simulator.run_simulation()`. For now, it is sufficient to provide this function with the simulation duration in milliseconds. It will then run a simulation from 0 ms to that provided parameter.

The following code snippet shows the most basic simulation script using the `Simulator` class. Please add this snippet to your script file.

```
from pydtnsim import Simulator

def main():
    """Simulate basic scenario."""
    # Create simulation environment
    simulator = Simulator()

    # Run the simulation for 1000 seconds (1000000 ms)
    simulator.run_simulation(1000000)

if __name__ == "__main__":
    main()
```

As no nodes or contacts were added to the `Simulator` object, nothing has to be simulated. When running the script, the output is as follows:

```
> python3 dtn_simulation.py
  Running simulation  for 1000000 ms ...
  Simulation completed!
  Simulation Results:
  - total number of packets generated: 0
  - total number of packets enqueued in limbos: 0
  - total number of packets enqueued in contacts: 0
```

Hooray, that was the first "successful" pyDTNsim simulation run! We didn't actually simulate any network but we can change that by adding simulation elements in the next step.

But first, let's have a look at the output provided by the `Simulator` object: besides the message that the simulation was completed the output also provides some simple statistics about the performed simulation run in lines *(5-7)*. In our case, no packets were generated and subsequently, no packets remained in limbos or contacts at the end of the simulation run.

1.2.3 Adding simulation elements

In order to not just simulate empty scenarios, we now have to add (active and passive) simulation elements to the simulation environment. In particular, two elements have to be represented in the environment: physical network nodes (e.g., `SimpleCGRNode`) and contacts in between those nodes (`Contact`).

Note: Both classes/objects referenced in the paragraph above are exemplary. Depending on the simulated routing mechanisms, the instantiated network node objects have to vary (and might even have to be implemented oneself for novel routing concepts). An opportunistic routing approach might have differing processing requirements both in terms of gathered knowledge at the physical nodes and the forwarding behavior during node contacts.

Warning: With the development focus of this simulation environment having been CGR, the generalization in terms of applicable routing mechanisms has not been fully implemented in this area of the application. For now, only CGR implementations are provided and no abstract parent class exists for the easy adoption with other routing approaches. This improvement will likely be implemented in the near future.

For the simulation elements including their helper classes, we need to add the following imports:

```
from pydtnsim import ContactPlan, ContactGraph, Contact
from pydtnsim.nodes import SimpleCGRNode
from pydtnsim.routing import cgr_basic
from pydtnsim.packet_generators import ContinuousPacketGenerator
```

The imported objects will be explained in the following paragraphs.

Network Topology

As the network node need to be aware about the network topology since we are using CGR (Contact Graph Routing) as routing mechanism, we have to provide such information during the instantiation.

We provide the topology knowledge as a `ContactGraph` object. This object represents the topology as a time-invariant graph and can be easily generated from a `ContactPlan` object. This object holds the same information as

the ContactGraph, but is easier to understand and modify for humans. More details on the reasoning behind the ContactPlan and the ContactGraph in the context of CGR is provided in *Contact Graph Routing*.

As outlined before, we first create the ContactPlan (line 2). The parameters provided during the instantiation are the default data rate in bits per millisecond (10 bits per millisecond, i.e. 10 KBps) and the default propagation delay in milliseconds (50 ms).

Warning: The propagation delay is currently not factored in when simulating networks. The interface for providing such information is already implemented, but the simulation logic is not implemented yet. This is future work, so for now, the propagation delay is always neglected.

```
1 # Generate empty contact plan
2 contact_plan = ContactPlan(10, 50)
3
4 # Add the contacts
5 contact_plan.add_contact('node_a', 'node_b', 0, 100000)
6 contact_plan.add_contact('node_a', 'node_b', 500000, 750000)
7 contact_plan.add_contact('node_b', 'node_c', 0, 200000)
8 contact_plan.add_contact('node_b', 'node_c', 350000, 400000)
9 contact_plan.add_contact('node_b', 'node_c', 950000, 990000)
```

In lines 5 to 9, the contacts based on our previously outlined scenario are added to the contact_plan object using ContactPlan.add_contact(). The parameters are

- the source node,
- the destination node,
- the start time in milliseconds and
- the end time in milliseconds.

As no additional optional parameters for the data rate and the delay were provided, the default values of the contact_plan object are used.

Finally, we can simply convert the filled ContactPlan object into a ContactGraph object:

```
# Convert contact plan to contact graph
contact_graph = ContactGraph(contact_plan)
```

Contacts

The contacts available in between network nodes throughout the simulation are simulated using the Contact object. These objects are an integral part of the simulation environment as they are one of two active **generator** elements that drive the simulation (and generate events, hence the name). Contacts are activated upon their contact start time and then perform handover operations from one node to another during their time active. At the end of a handover of a packet to another node, the routing mechanism on that other node is called to determine the future forwarding.

With the information about the contacts already being available in the contact_plan object of the previous step, we can iterate over that information to generate our Contact objects:

```
1 # Generate contact objects and register them
2 for planned_contact in contact_plan.get_contacts():
3     # Create a Contact simulation object based on the ContactPlan
4     # information
5     contact = Contact(planned_contact.from_time, planned_contact.to_time,
```

(continues on next page)

(continued from previous page)

```
6                       planned_contact.datarate, planned_contact.from_node,
7                       planned_contact.to_node, planned_contact.delay)
8       # Register the contact as a generator object in the simulation
9       # environment
10      simulator.register_contact(contact)
```

In addition to the instantiation of the contacts in lines 5 to 7, in line 10, the respective contact also has to be registered with the simulation environment. This is to allow the simulation environment to call the contact upon its start time.

Network Nodes

With the topology information available in the correct format, we can add the network nodes. For all three nodes, we will use `SimpleCGRNode` as representation in the simulation environment. Again, we can use the *contact_plan* object that we instantiated and filled earlier to gather the relevant information for the instantiation:

```
1  # Generate network node objects and register them
2  for planned_node in contact_plan.get_nodes():
3      # Generate contact list of node
4      contact_list = contact_plan.get_outbound_contacts_of_node(planned_node)
5      # Create a dict that maps the contact identifiers to Contact simulation
6      # objects
7      contact_dict = simulator.get_contact_dict(contact_list)
8      # Create a node simulation object
9      SimpleCGRNode(planned_node, contact_dict, cgr_basic.cgr, contact_graph,
10                   simulator, [])
```

In line 2, we get a list of all network nodes in the topology using `ContactPlan.get_nodes()`. We then iterate over this list and create the individual nodes:

1. We first have to get a list of all outbound nodes that the individual network node has. This is done calling `ContactPlan.get_outbound_contacts_of_node()`.
2. This list is then used to get a `contact_dict` from the `Simulator` object. As this object is aware of all contacts and their registered instantiations, it can map the textual list entries of the outbound contacts from the previous step to actual `Contact` objects. So by calling `Simulator.get_contact_dict()`, we get a `dict` mapping a contacts identifier to the instantiation in the simulation context.
3. With the `contact_dict` available, we can instantiate the actual network node object as `SimpleCGRNode` with the following parameters:
   - the node identifier (e.g. `node_a`),
   - the `contact_dict`,
   - the routing mechanism's main function `cgr_basic.cgr`,
   - the topology information (in this case as `ContactGraph` object) and
   - the `Simulator` object.

Packet Generators

Finally, we have to add packet generators that are injecting packets into the simulated network. Without them, regardless of the specified topology, no packets would be forwarded and thus, no non-trivial network behavior would be simulated.

Currently, there two different packet generators provided in the simulation environment, the `BatchPacketGenerator` and the `ContinuousPacketGenerator`. Both are children of the parent `BasePacketGenerator`.

The injection behavior of the two generators is depicted in the following figure:

The `BatchPacketGenerator` injects a specified number of packets at specified points in time whereas the `ContinuousPacketGenerator` injects packets continuously throughout the simulation period with a defined generation data rate.

Depending on the scenario, one of them might be used for the simulation conducted. Alternatively, an own generator can be implemented based on the `BasePacketGenerator`.

For our scenario, we will use the `ContinuousPacketGenerator` to inject packets for the routes `node_a -> node_c` and `node_c -> node_a` at a specified data generation rate of 10 KBps and with a packet size of 100 KB:

```
1  # Generate packet generator 1 and register them
2  generator1 = ContinuousPacketGenerator(
3      10,           # Data Generation Rate: 10 Bytes per ms
4      100000,       # Packet Size: 100 KB
5      ['node_a'],   # From 'node_a'
6      ['node_c'],   # To 'node_c'
7      0,            # Start injection at simulation time 0s
8      1000000)      # End injection at simulation end (1000s)
9
10 # Generate packet generator 2 and register them
11 generator2 = ContinuousPacketGenerator(
12     10,           # Data Generation Rate: 10 Bytes per ms
13     100000,       # Packet Size: 100 KB
14     ['node_c'],   # From 'node_c'
15     ['node_a'],   # To 'node_a'
16     0,            # Start injection at simulation time 0s
17     1000000)      # End injection at simulation end (1000s)
18
19 # Register the generators as a generator objects in the simulation
20 # environment
21 simulator.register_generator(generator1)
22 simulator.register_generator(generator2)
```

Warning: The implementation of the packet generator configuration is currently requiring the instantiation of two distinct generators to accomplish the bidirectional injection of packets between `node_a <-> node_c`. This will be changed in a future release (likely `v0.3.0`).

Warning: Also, the registration procedure for the generators is currently inconsistent with the other simulation elements. Therefore, a consistent registration procedure will be established in a future release as well.

Two generators have to be instantiated, one for the injection of packets traveling from *node_a* to *node_c* and one for the reverse direction from *node_c* to *node_a*.

The two instantiations in lines 2 to 8 and 11 to 17 are provided with several parameters:

- the data generation rate,
- the packet size,

- a `list` of (`string`) node identifiers identifying all source nodes that the generator should inject packets with the given parameters and data rate,
- a `list` of (`string`) node identifiers identifying all destination nodes that the generator should address packets to with the given parameters and data rate,
- the injection start time (in ms absolute to the simulation start time) and
- the injection end time (in ms absolute to the simulation start time).

With `list`'s used for identifying source and destination nodes, the the generator injects for every element of the source node list packets with the given characteristics and rates to all elements of the destination node list. This injection scheme is also depicted in the following diagram:

With the two generators instantiated an configured, we have to register them with the simulation environment. This is done in lines 21 and 22.

We now have all simulation elements in place and can run the simulation again. If you don't want to copy all code snippets from this documentation, you can also download the file created up to this point of the tutorial at `this link`.

If we run this extended script, we get the following output:

```
1 > python3 dtn_simulation_elements.py
2   Running simulation  for 1000000 ms ...
3   Simulation completed!
4   Simulation Results:
5   - total number of packets generated: 198
6   - total number of packets enqueued in limbos: 165
7   - total number of packets enqueued in contacts: 0
```

You can see that the generators properly generated packets and injected them into the network. The number of generated packets seems about right: with the configuration provided, the generators inject a packet to the (single) destination every 10 seconds. With 1000 seconds being simulated, this results in 99 injected packets per generator and 198 in total. As the simulation ends when 1000s is reached (excluding the termination value), the 100th packet of each generator that would be due at time 1000s is not added.

Also, we can see in line 6, that only a fraction of the injected packets is actually forwarded and 165 of the 198 packets are enqueued in one of the nodes' limbos. A limbo is a queue that holds packets that cannot be forwarded to their destination nodes based on the available topology information. Every node has a limbo that is used for such packets.

Note: The number of "discarded" packets can be directly attributed to the selected topology, the contact times and the nodes configuration (including the injection rates) and does not represent an (programming) error.

As our contact plan has the same validity period as our simulation duration, no packets should remain scheduled in contacts after the simulation end. This is the case as can be seen in line 7.

Warning: If the validity period of topology information exceeds the simulated period (e.g., a 1 hour simulation is conducted with a contact plan containing the computed contacts for 48 hours), packets can remain enqueued in *future* (i.e., beyond the simulation end time) contacts and will appear in the simulation results summary (in our example in line 7).

In this section, we successfully simulated our specified simulation scenario. We even got some bits of information about what happened during the simulation (e.g., that a large number of packets was at some point enqueued into a node's limbo).

However, usually when running a network simulation (especially in the academic context), more detailed analyses and key values are required. The next section will cope with the monitoring interface of pyDTNsim that allows for an extraction of arbitrary such values.

1.2.4 Monitoring of the Simulation

1.2.5 Running a Simulation

1.2.6 Post Processing and Evaluation

1.3 Examples

1.4 Simulation Scenario Setup

1.5 Routing Mechanisms

With `pyDTNsim`, the behavior of various routing mechanisms can be simulated. The currently implemented mechanisms are outlined in the next chapters:

1.5.1 Contact Graph Routing

1.6 Simulation Monitoring

1.7 Simulation Data Processing

1.8 pydtnsim package

1.8.1 Subpackages

pydtnsim.backend package

Submodules

pydtnsim.backend.qsim module

Module contents

pydtnsim.monitors package

Submodules

pydtnsim.monitors.base_monitor module

pydtnsim.monitors.monitor_notifier module

Module contents

pydtnsim.nodes package

Submodules

pydtnsim.nodes.simple_cgr_node module

Module contents

pydtnsim.packet_generators package

Submodules

pydtnsim.packet_generators.base_packet_generator module

pydtnsim.packet_generators.batch_packet_generator module

pydtnsim.packet_generators.continuous_packet_generator module

Module contents

pydtnsim.routing package

Submodules

pydtnsim.routing.cgr_anchor module

pydtnsim.routing.cgr_basic module

pydtnsim.routing.cgr_utils module

pydtnsim.routing.dijkstra module

pydtnsim.routing.scgr module

Module contents

1.8.2 Submodules

1.8.3 pydtnsim.contact module

1.8.4 pydtnsim.contact_graph module

1.8.5 pydtnsim.contact_plan module

1.8.6 pydtnsim.packet module

1.8.7 pydtnsim.simulator module

1.8.8 Module contents

1.9 Architecture Overview

1.10 Development Guide

CHAPTER 2

Indices and tables

- genindex
- modindex
- search
- glossary