# Case Study and Comparison of SimPy 3 and OMNeT++ Simulation

Vaclav Oujezsky and Tomas Horvath
Department of Telecommunications
Brno University of Technology
Brno, Czech Republic
Email: vaclav.oujezsky@phd.feec.vutbr.cz

***Abstract*—This article presents a case study and comparison of simulations carried out by SimPy 3.0 of Python module and OMNeT++ tool to explore the performance. The target of this comparison is in particular configuration of network for future steps what tools would be good to use. Two simple models of a network were created with Python and C++ language.**

***Keywords*—comparison; case study; Python; simulation; Simpy 3**

## I. Introduction

Simulations of network behavior can be modeled in software, and simulation tools exist for this purpose, e.g.: OMNeT++ [1], SimPy module [2] of standard Python, NS2, NS3 [3] and others. In this article, we describe how we created and tested two chosen programs from those listed above and then, on the basis of these results, chose one for our further work.

We are concentrating on network research and network behavior. The criterion for determining which modeling tool to select were as follows: open source license, scalability, performance, the user opinion of the use, which mean what modeling software is more acceptable to use (meaning factor) and networking.

The authors in [4] presented the experimental comparison of different technologies in the context of processing speeds in different Python version. They also compared the results from Python with the C programming language.

The comparison of network simulators is presented by Atta ur Rehman Khan et al. [5]. The authors evaluated the performance of state of the art simulators from the routing protocol point of view. A simulator based on Python language has not been compared in this article.

The Python language is very intuitive and an example of network simulations using this language is presented in [6]. Techniques, as is: distributions in networking, M/M/1 queue, switch port, traffic shaping are presented by examples and source codes. After considering all conditions, we chose Python with the modules SimPy 3 and OMNeT++.

## II. Equipment

Python 3.4.3 version, SimPy 3.0.8, OMNeT++ 4.6 IDE (Integrated Development Environment) were used for our investigation. A computer with Intel(R) Core(TM) i5-3340M CPU @ 2.70GHz, 4 GB RAM, and Windows 7 OS $F\times 64$. PyCharm Educational Edition 1.0.1. for Python programming is used.

Research described in this paper was financed by the National Sustainability Program under grant LO1401. For the research, infrastructure of the SIX Center was used.

Any module to improve the performance of the source code, for example “Parakeet runtime compiler” [7], has not been used. Such modules can speed up the processing. At the end of development, the processing speed of our program was tested in Python 3.4.3 with the `cProfile` module and in OMNeT++ with the included tools. The source code of the simulations we created are stored on GitHub [8] for further improvement.

### *A. SimPy 3*

SimPy 3 is the newest release of SimPy [2] which is a “Process-based Discrete-event Simulation Framework” based on standard Python. Events in SimPy are based on Python generators and can also be used for asynchronous networking or to implement simulated and real multi-agent communication systems.

We used the Windows operating system (OS), and the following commands describe the process of installing SimPy for this OS. SimPy can be installed on Windows in the same way as SimPy 2, that is: by running the following commands in the Windows CMD (Command Line):

```
  C:\>pip install simpy
```

or

```
  C:\>easy_install simpy
```

The `pip` and `easy_install` commands are provided by from Python installation. SimPy 3 and SimPy 2 have some discrepancies in coding conventions; for information on this, reference the section of source [2] titled “Porting from SimPy 2 to 3”.

### *B. OMNeT++*

OMNeT++ is a powerful “Discrete Event Simulator”. As of writing, the latest release is 5.0b2. This tool is available for download at the OMNeT++ web page [1]. OMNeT++ is installed by using the commands specified below, which are provided by the file `mingwenv.bat` file in the download.

```
/omnetpp-4.6$ . setenv
/omnetpp-4.6$ ./configure
/omnetpp-4.6$ make
/omnetpp-4.6$ omnetpp
```

The distribution of OMNeT++ includes samples and libraries which can help to easily create a variety of network status simulations. The common library "INET" includes elements such as routers, switches, servers and so on.

Instead of using the included elements in "INET", we decided to create own elements according to the "OMNeT++ User Manual Version 4.6" [9].

## III. Scenario

A simple scenario was created for testing. It models a network with two clients. One client generates messages (packets) and the second client consumes these messages, as shown in Fig. 1. A buffer which processes messages through a queue is located between the two clients.

The idea is that the second client has limited resources to forward incoming traffic, varying with the propagation link delay, and the simulation shows the frequency with which the buffer drops messages.

The buffer has a size limit of 100 messages (units) per one period (simulation time) and it is defined in the class `Switch()` with the Simpy module container. One packet is represented by one message in OMNeT++ or one number in SimPy. The messages are generated and transmitted from Client 1 to Client 2 in each period step. We ran tests with 300, 3,000, and 30,000 messages to compare the behavior of the simulation at different message volumes.

## IV. Configuration

### A. SimPy configuration

A simulation in SimPy consists of imported modules, and its resources and processes are defined by parameters. SimPy defines three types of resources. These resources can be used to synchronize processes.

- **resource** – expresses shared resources with priorities and preemption.
- **container** – expresses resource that shares homogeneous objects among processes.
- **store** – expresses shared resources capable of storing an unlimited amount of objects.

```
                        (Switch)
+-----------+              |              +-----------+
|           |              |              |           |
|  Client2  | <<------+Buffer -------  |  Client1  |
|           |            Queue            |           |
+-----------+              |              +-----------+
 Process RX                |               Process TX
```

Fig. 1: The basic idea and schema of the simulation

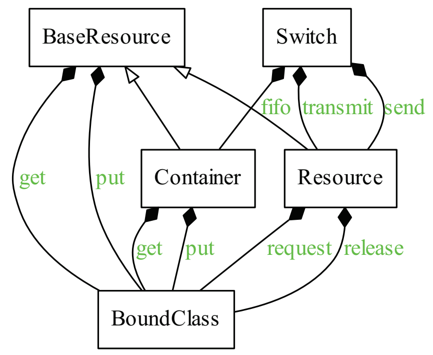

Fig. 2: "Pyreverse [10]" UML diagram

Location #(2) in Code 1 defines the class `Switch()`. This class contains defined types of SimPy resources: `transmit: Resource`, `send: Resource`, and `fifo: Container`. Its UML diagram, and its interactions with SimPy classes, are shown in Fig. 2. At location #(3) in Code 1, the function `drop()` is defined. This function represents a message being dropped due to a full buffer. At #(4) and #(5), the `rx()` (receive) and `tx()` (transmit) functions are defined, and the functions at #(6) and #(7) are generators.

Code 1: Abbreviated notation of SimPy coding

```
import simpy # (1)

class Switch(): # (2)
def __init__(self, env):
def monitor_fifo(self, env):

def drop(env, switch): # (3)
def rx(num, env, switch): # (4)
def tx(num, env, switch): # (5)
def tx_generator(env, switch): # (6)
def rx_generator(env, switch): # (7)

def main():
env = simpy.Environment() # (8)
switch = Switch(env) # (9)
user_proc = env.process(rx_generator
\(env, switch)) # (10)
env.process(tx_generator(env, \
switch)) # (11)
env.run(until=rx_proc) # (12)
```

At #(8 – 12), the `main()` function is defined: it defines the "environment" in which the simulation is run. First, it creates an instance of the "Environment" class and passes this new instance into the constructor for the "Switch" class. Next, transmitting and receiving generators are created and added to the “environment”.

They need to be started and added to the “environment” via `env.process()`. Finally, the simulation starts by calling `env.run()` with passing the time of simulation duration. This time is adjusted by duration of `rx()` generator.

## B. *OMNeT++ configuration*

As was mentioned above, the scenario in OMNeT++ is created by C++ definitions of modules. An OMNeT++ model consists of modules which are hierarchically nested, as shown in Fig. 3.

Modules communicate by passing messages. There is no fixed limit on how deeply modules may be nested.

- **compound module:** this module contains submodules.
- **simple module:** this module is created by C++ algorithms.

In our scenario two simple modules and one compound module were created, as is shown in Fig. 4. The modules for “Client 1” and “Client 2” consist of `client1.cc` and `client2.cc` files.

The “Switch” consists of a simple module “iSwitch”, “buffer” and “bucket”, as is shown in Fig. 5. These modules are also created by their own `*.cc` files.

All modules are connected with gates and connections and which are defined in a node “NED” file. All these together are the “runtime parameters”:

- **defining parameters** – node’s “NED” file.
- **reading parameters** – node’s C++ source code.
- **setting parameters** – omnetpp.ini file.

In the Code. 2 and Code. 3 below there are parts from the “NED” and C ++ source files of the “switch” node. The code of “iswitch” which is a part of the “Switch” mentioned above includes class `iSwitch` as the `cSimpleModule` and the handling of messages is ensured by “finite state machine” code called `FSM_switch`.

In this scenario “Client 1” generates messages until fall short of the defined parameters are reached. Those generated messages pass through the compound module “switch”, where the “buffer” is implemented. The processing of this “buffer” is controlled by self messages and messages from “Client 2”. The code of the “buffer” is is again created as **`cSimpleModule`**.

```
+-------------------------------------------------------+
| +----------------------------+   System module        |
| |       Compound module      |                        |
| |+-----------+  +-----------+|         +-----------+  |
| ||  Simple   |  |  Simple   || Message |  Simple   |  |
| ||  Module   |  |  Module   ||<------->|  Module   |  |
| |+-----------+  +-----------+|         +-----------+  |
| +----------------------------+                        |
+-------------------------------------------------------+
```

Fig. 3: The basic composition of modules in OMNeT++

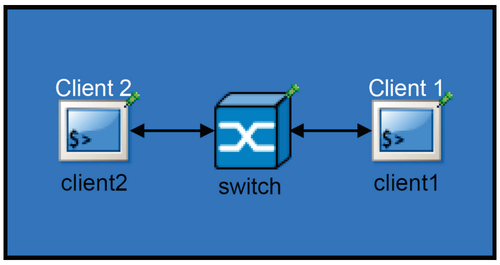

Fig. 4: Scenario in OMNeT++

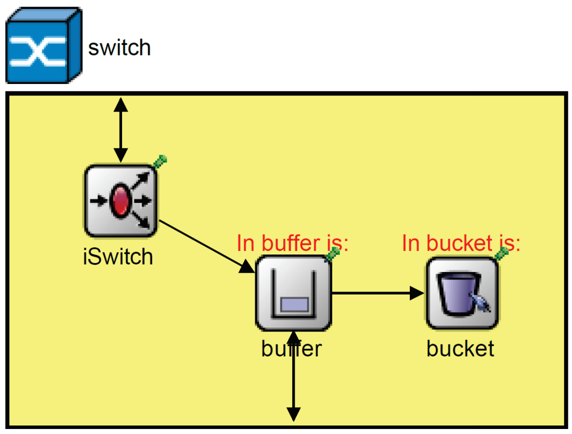

Fig. 5: Created compound module “switch”

The handling of messages is controlled by the itself message and it is waiting for messages arriving from the port `iSwitch`. If the buffer is empty, the messages are pop-ed to the queue, in other case they are sent to the module “bucket”.

Code 2: “NED” part of the iSwitch and switch module

```
simple iSwitch
{  parameters:
gates:
output dim2ToClient1;
output dim2ToClient2;
input in;
}

module switch
{ connections:
iSwitch.dim2ToClient1 --> tx1;
iSwitch.in <-- rx;
iSwitch.dim2ToClient2 --> \
    buffer.portISwitch;
buffer.out --> tx2;
rx2 --> buffer.in;
buffer.toBucket --> bucket.rx;
}
```

This process is also combined by function which is waiting for such confirmation that “Client 2” has processed the previous message. Until then, the queue process adds incoming messages from “Client 1” to the queue. Once it receives confirmation from the “Client 2”, removes a message from the queue and sends it.

Code 3: C++ source code part of the iSwitch

```
void iSwitch::initialize() {
fsm.setName("fsm");
}
```

This queue is FIFO (First In First Out) and it is created from the OMNeT++ simulation class library `cQueue queue;`. Messages are only sent to the bucket when the “buffer” exceeds its configured length limit `queue.length()`.

In order to keep parity with the SimPy simulation, the OMNeT simulation adds an additional transmission delay (default of 2s) between “switch” sending a message and the “Client 2” receiving it. This transmission delay is defined by configuration `delay = default(2s)` in the “NED” file of the scenario.

## V. Comparison of simulations

Execution and completion of time of simulation was observed and, with SimPy, the real time of the processes. In both cases of simulations statistics were created. Statistics were assembled for both simulations.

Code 4: Statistic in OMNeT++

```
// buffer.cc definition
simsignal_t queuelenghtSignal =
cComponent::registerSignal
                        ("gueuelenght");
//in queue process queue.pop()
emit(queuelenghtSignal,queue.length());

// switch ``NED'' definition
@signal[queuelenght](type="long");
@statistic[queuelenght]
            (title="queue length vector";
record=vector;
interpolationmode=sample-hold);
```

We searched for variations in behavior with different volumes of messages. A graphical illustration of the “buffer” in the SimPy simulation was created using the `plot()` function from the Python `matplotlib` library.

In OMNeT++, the simulation was configured to emit signals for statistics-gathering, and the configuration is shown in Code. 4. The results are compared in Fig. 6 and Fig. 7. In both graphs we can see results of the simulation.

Buffer size is plotted on the y-axis, and the total simulation time is on the x-axis. In this case, 100 messages are sent through the queue process. If the limit was exceeded, messages were dropped to the “bucket”. In OMNeT++ this is achieved with the statement `delete msg;` in the C++ definition, and in SimPy with `env.process(drop(self))`. The simulation is run from the command line: in SimPy, through the Python command prompt, and in OMNeT++, with a command to `mingwenv.bat`, as shown below.

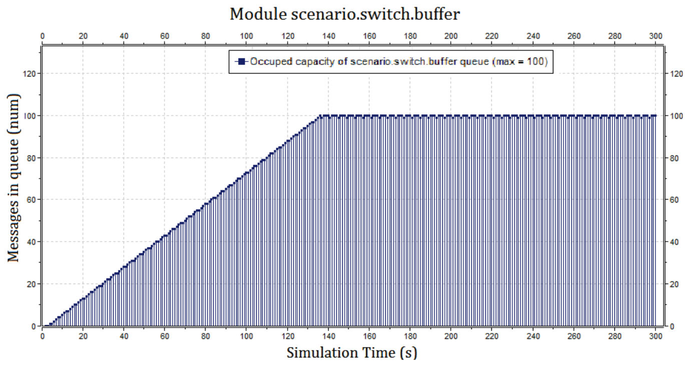

Fig. 6: Graph of buffer in OMNeT++

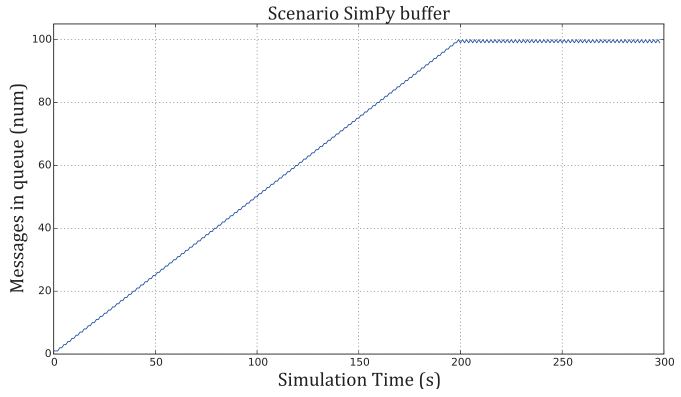

Fig. 7: Graph of buffer in SimPy

```
/omnetpp-4.6/samples/NeT$
NeT.exe -u Cmdenv
```

The Graphical User Interface (GUI) `Tkenv` of OMNeT++ was not used. Instead of this GUI we used the command-line user interface `Cmdenv`. Each simulation was repeated ten times for the chosen number of messages. The simulation was run with 300, 3,000; 30,000; 300,000 and 3,000,000 messages. In case of OMNeT++, there are more events than there are messages generated by “Client 1”, because confirmation messages and self-messages are also generated by the “Client 2” and “buffer”. The final result of the testing is shown below:

```
SimPy
Duration: 0.0443761 seconds 300 units
Duration: 0.524 seconds 3000 units
Duration: 4.729 seconds 30000 units
Duration: 46.634 seconds 300000 units
Duration: 477.372 seconds 3000000 units
```

```
   OMNeT++
   ** Event #1477 T=301
Elapsed: 0.002s (0m 00s)
   ** Event #14752 T=3001
Elapsed: 0.016s (0m 00s)
   ** Event #147502 T=30001
Elapsed: 0.150s (0m 00s)
   ** Event #1475002 T=300001
Elapsed: 1.495s (0m 01s)
   ** Event #14750002 T=3000001
Elapsed: 14.972s (0m 14s)
```

## VI. Conclusion

The article presented the case study and a comparison of the OMNeT++ and SimPy applications for the network simulations. We programmed our own models of simple networks with Python and C++ languages. The complete source code used in this article is saved on the GitHub server to view it.

It was assumed that compiled C ++ applications would have a faster execution time and response, because the source code is not translated and compiled during the start up of the program. We also assumed that due to the increased overhead of the Python language, SimPy would tend to take longer to finish a simulation. This also proved incorrect.

Of course, the complexity of the task must be taken into account. On the other hand, it does not always require solving complex tasks.

In both cases, the execution time of the simulation is in each step increased by one order. The time difference between the two simulations was also one order: for example, when SimPy was generating 300 messages, processing time increased by 0.0443761 seconds per 300 messages, and in OMNeT++ 0.002 seconds per 300 messages. The simulation time of messages dropping by the buffers is also a little bit different. It is due to the demand of more precise settings of timing. But intended overall intention to simulate the mentioned scenario and observe the time processing and ambitiousness of used programming languages was met.

Using the `cProfile` tool, we observed that the most time-consuming part of SimPy simulation is the object generator's `send` method. The total time used by this method was 0.03355 seconds per 300 units.

In the simulation we created, OMNeT++ displayed faster processing times than SimPy; however, preparing the simulation was far simpler in SimPy, hence it is scripting language. Its advantages you can see in [11].

There are also advantages and disadvantages of the C++ and Python languages which must be considered, and most importantly, the needs and experience of individual users will differ broadly.

## References

[1] OpenSim Ltd., OMNeT++: Discrete Event Simulator, ©2001–2016.
[2] Team SimPy., Documentation for SimPy, ©2002–2016.
[3] NS-3 Consortium, NS-3, ©2011–2016.
[4] L. Jun and L. Ling, ”Comparative research on Python speed optimization strategies,” International Conference on Intelligent Computing and Integrated Systems (ICISS), pp. 57–59, October 2010.
[5] A.R. Khan, S.M. Bilal, M. Othman, ”A Performance Comparison of Network Simulators for Wireless Networks,” Cornell University Library, pp. 1–6, July 2013.
[6] Grotto Networking, ”Basic Network Simulations and Beyond in Python,” 2016.
[7] Parakeet a runtime compiler for numerical Python, ©2016.
[8] GitHub, ©2016.
[9] A. Varga, ”OMNeT++ User Guide version 4.6,” ©2014.
[10] E. Anclin, ”Pyreverse : UML Diagrams for Python,” Logilab, ©2016.
[11] C.H. Parker, ”Why We Choose Python,” Sixfeetup, February 2013.