VNE-Sim Tool User Guide

Nashila Jahan
Kamila Bekshentayeva
Soroush Haeri
Ana Gonzalez

Simon Fraser University
20/09/2019

VNE-Sim is a discrete event simulator written in C++ 11. It is publicly available under the terms of the MIT License. This tool may be used to simulate virtualization algorithms and compare their performance using various topologies of data center networks. CMake build system is used to compile all required packages before the test cases are run for various scenarios. Furthermore, being written in C++, VNE-Sim offers good memory management and enables batch simulations using any scripting language. A step by step procedure is provided to enable effective use of the tool.

1. Gathering Pre-Requisites

Follow the next steps to install VNE-sim and required packages:
Download the package from website: https://bitbucket.org/shaeri/vne-sim
and unzip it in your home directory. VNE-Sim code may be also downloaded using a command line:

```
git clone https://bitbucket.org/shaeri/vne-sim
```

The root directory of vne-sim should be */home/username/vne-sim*.

Install the following pre-requisites available via Linux distribution:

- CMake
- Boost libraries
- GSL: GNU scientific library
- GLPK: GNU linear programming kit.

For Ubuntu, use specific version or name of the library:

```
sudo apt-get install <name of the library>
```

For MacOS, use the following commands:
to install Homebrew, a free and open-source software package management system that simplifies the installation of missing packages:

```
ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

then to install the prerequisite libraries:

```
brew install <name of the library>
```

example: `brew install boost`

Other required libraries will be installed as part of *VNE-Sim* package are:

- Fast Network Simulation Setup (FNSS): used to create data center topologies
- Boston University Representative Topology Generator (BRITE): for generating network topologies together with FNSS
- Hiberlite: for saving simulation results
- Adevs: for modeling virtual network embedding processes

- SQLite3: used for handling simulation results exported automatically as SQLite databases.

CMake relies on scripts to search for the listed libraries and applies the required modifications after downloading these libraries. A C++11 compiler is required to compile the code.

2. Compiling Source Code

To compile the source code, change directory to */home/username/vne-sim* created in the previous step and type the following commands in the terminal window:

```
mkdir build && cd build
cmake .. -DWITH_FNSS_SUPPORT=off
make
```

This will create a *build* directory under *vne-sim* directory and install all required libraries after successful compilation. At the beginning, all required modules are checked:

```
-- The CXX compiler identification is GNU 4.9.4
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- ADEVS DOES NOT EXIST
-- HIBERLITE DOES NOT EXIST...
-- Found Git: /usr/bin/git (found version "1.9.1")
-- Boost version: 1.56.0
```

After initiating the build for the first time, CMake attempts to download and patch some of the dependencies (FNSS, Hiberlite, ADEVS). Samples of errors if this step fails or gets interrupted are:

```
make[2]: *** No rule to make target `../external-libs/hiberlite/
libhiberlite.a', needed by `lib/libcore.so'.  Stop.
```

In this case, following directories need to be deleted in the *vne-sim* root directory:

- external-libs/adevs
- external-libs/hiberlite

When installing Hiberlite, Github may request for your name and email address.
Also delete all CMake generated files under the *vne-sim/build* directory. Then, attempt to build the code again from scratch. If successfully compiled, the display should show:

```
[ 98%] Building CXX object src/network-file-generator/test/CMakeFiles/nfg-
test.dir/network-file-generator-test.cc.o
Linking CXX executable ../../../bin/nfg-test
[ 98%] Built target nfg-test
Scanning dependencies of target vineyard-test
```

```
[100%] Building CXX object src/Vineyard/test/CMakeFiles/vineyard-
test.dir/vy-test.cc.o
Linking CXX executable ../../../bin/vineyard-test
[100%] Built target vineyard-test
```

3. Modifying Configuration File

All configurations and runtime parameters required for *VNE-sim* execution are stored in an XML file *configurations.xml*, which resides in the *root* folder of *VNE-sim*. It is important to set correct run-time paths pointing to *VNE-sim* root directory in the *configurations.xml*. The GLPK files *<glpk></glpk>* are part of the code. Hence, change only the beginning of the paths to point to the *VNE-sim* directory.

Next, download the latest network files *network_files.zip*, save them outside the *VNE-sim* folder in the *home* directory, and unzip them:

http://www2.ensc.sfu.ca/~ljilja/cnl/projects/VNE-Sim/vne-sim-web/index.html#network

The network files are already available and hence they do not need to be generated again. Ignore all the configurations within the tag:

```
<NetworkFileGenerator></NetworkFileGenerator>
```

Set the correct path within the tags pointing to the network files you have unzipped:

```
<SubstrateNetwork></SubstrateNetwork>
```

and

```
<VirtualNetRequest></VirtualNetRequest>
```

NOTE: The directory listed in configurations.xml file refers to a set of network files for mean arrival rate of 100 (1 request per 100 units of time or 10 Erlangs). This directory name should be changed to arrival rate of 12 because network files network_files.zip downloaded from the website only have a subset of files for traffic with mean arrival rate of 12 (12.5) (8 requests per 100 units of time or 80 Erlangs)

Change:

```
<dir>reqs-100-1000-nodesMin-3-nodesMax-10-grid-25</dir>
```

to

```
<dir>reqs-12-1000-nodesMin-3-nodesMax-10-grid-25</dir>
```

These two paths in configurations.xml file need to be changed to point to correct network files:

```
<SubstrateNetwork>
<path>/home/username/network_files/SN</path>
```

```
<filename>vy_substrate_net_n_50_outergrid_25_inner_grid_25.txt</filename>
</SubstrateNetwork>
<VirtualNetRequest>
<path>/home/username/network_files/VNRs</path>
<dir>reqs-12-1000-nodesMin-3-nodesMax-10-grid-25</dir>
<reqfileExtension>.txt</reqfileExtension>
</VirtualNetRequest>
```

The configuration within the <vineyard></vineyard> tag should be correctly set and point to the existing files. The directory and filenames should match with actual directory and filenames in the SN and VNRs directories within the network files network_files.zip used in the test simulation. The modified configurations file is listed in the Appendix A1.

4. Executing the Test Case

The source code of the test cases is placed in various packages under the test folder. All package/test/testname.cc (e.g., vineyard/test/vineyard-test.cc) files are compiled into executables. For the first test cases, only use this "experiment-test" located at:

```
src/experiments/test/expriments-test.cc
```

It is compiled into an executable under:

```
build/bin/experiments-test
```

These test cases have been reported in References [1]-[5] listed at the end of this Guide. The experiments and tests are written using the Boost library unit test framework. The documentation describing the framework is available at:

http://www.boost.org/doc/libs/1_53_0/libs/test/doc/html/utf/user-guide/runtime-config/run-by-name.html

Open the file *src/experiments/test/expriments-test.cc*. After the *#include* and using clauses, review the section:

```
BOOST_AUTO_TEST_SUITE (AlgorithmExperiments)
BOOST_AUTO_TEST_CASE(ARRIVAL_RATE_TESTS)
```

"AlgorithmExperiments" is the name of test suite and *"ARRIVAL_RATE_TESTS"* is the name of test case that we wish to run. Some minor modifications are required in *experiments-test.cc* file because it was originally written for a complete set of network files. However, we shall only use a subset of these network files. Now, modify original experiments-test.cc file, within the scope of:

```
BOOST_AUTO_TEST_CASE(ARRIVAL_RATE_TESTS)
{
}
```

Change:

```
std::string vnr_dirs[] =
{"reqs-12-1000-nodesMin-3-nodesMax-10-grid-25", "reqs-14-1000-nodesMin-3-
nodesMax-10-grid-25","reqs-16-1000-nodesMin-3-nodesMax-10-grid-25", "reqs-
20-1000-nodesMin-3-nodesMax-10-grid-25","reqs-25-1000-nodesMin-3-nodesMax-
```

```
10-grid-25","reqs-33-1000-nodesMin-3-nodesMax-10-grid-25" "reqs-50-1000-
nodesMin-3-nodesMax-10-grid-25", "reqs-100-1000-nodesMin-3-nodesMax-10-
grid-25”};
```

to:

```
std::string vnr_dirs[] = {"reqs-12-1000-nodesMin-3-nodesMax-10-grid-25”};
```

and all the for loops:

```
for (int j = 0; j < 8; j++)
                { … }
```

to:

```
for (int j = 0; j < 1; j++)
                { … }
```

After we have modified experiments-test.cc, the entire code needs to be recompiled using **make** command as instructed in Section 2.

Finally, to run the first test, type in the terminal:

```
./experiments-test --run_test=AlgorithmExperiments/ARRIVAL_RATE_TEST
```

5. Extracting Simulation Results

After successful completion of the test run, unprocessed data generated by discrete events that occur during simulations are saved as SQLite databases in VNE-sim directory. These data may be processed using SQL queries or other statistical analysis tools. The shell script (that can be found on the website) may be used to extract key performance data from database files generated through the test.

Save *export_overal_avg_data.sh* outside of vne-sim directory. In terminal, change directory to where the script is saved and type below command to run the script:

```
./export_overal_avg_data.sh
```

Based on which test cases you are running in step 4, you may not have all required database tables mentioned in the script. You will see error messages for missing database files. Same time CSV files with key performance statistics will be generated from existing database files in VNE-sim directory.

Note that we are using only one arrival rate of 12 for test simulation. This can be easily extended to a range of arrival rates by using for loop:

```
for
arrivalRate in 12 14 16 20 25 33 50 100
do
sqlite3vne-sim/mcvne_bfs_mcf_reqs-$arrivalRate-1000-nodesMin-3-nodesMax-10-
grid-25.db < query.sql
```

The table below lists average performance data for various algorithms:

Algorithm	Acceptance Ratio	Average Revenue	Average Cost	Average Node Util-n	Average Link Util-n	Average Proc. Time
mcvne_mcf_mcf	0.593	99.52720777	116.488989	0.845251659	0.150005423	47.407665
mcvne_bfs_mcf	0.59775	99.96272148	117.6633616	0.846695419	0.151604332	1.298086342
mcvne_bfs_bfs	0.59275	100.0233351	116.828891	0.845651886	0.157358927	1.191999576
grc_mcf	0.59725	99.60965667	129.7124461	0.844490922	0.184736977	0.184035019
grc_bfs	0.59725	99.39737272	129.1837899	0.840446143	0.193236378	0.018504416
dvine_mcf	0.58125	97.84037373	133.1728131	0.805088215	0.192580523	0.622051665
rvine_mcf	0.59125	97.93744992	134.3247153	0.819640308	0.197085074	0.608317071

6. Network File Generation using BRITE Handler

This Section describes various parameters used for generating network files for Substrate Network (SN) and Virtual Network Requests (VNRs).

Both SN and VNRs are generated using BRITE [6] library with RT Waxman algorithm. Substrate graph is composed of 50 nodes where each node is randomly connected to a maximum of 5 other nodes. The substrate graph generated for the simulation scenarios consists of 221 edges. Each node of the substrate network is randomly placed on a 25 by 25 grid as:

```
<nodePlacement>1</nodePlacement>
<numNeighbors>5</numNeighbors>
<innerGridSize>25</innerGridSize>
<outerGridSize>25</outerGridSize>
<RTWaxman>
        <growthType>2</growthType>
        <alpha>0.5</alpha>
        <beta>0.2</beta>
</RTWaxman>
```

For each VNR, number of nodes are uniformly distributed between 3 and 10 and each virtual host is connected to maximum of 3 neighbor hosts. The CPU requirements of the virtual nodes are uniformly distributed between 2 and 20 units while the bandwidth requirements of the virtual links are uniformly distributed between 0 and 50 units, or 1 and 10 (as used below). Duration of each simulation scenario is 50,000 time units. The listed code shows the network file parameters for simulating VNRs:

```
<TotalTime>50000</TotalTime>
<VNTopologyType>Waxman</VNTopologyType>
<VNRLinkSplittingRate>0.1</VNRLinkSplittingRate>
<VNRNumNodesDist>0</VNRNumNodesDist>
<VNRNumNodesDistParam1>3</VNRNumNodesDistParam1>
<VNRNumNodesDistParam2>10</VNRNumNodesDistParam2>
<VNRNumNodesDistParam3>-1</VNRNumNodesDistParam3>
<VNRDurationDist>1</VNRDurationDist>
<VNRDurationDistParam1>1000</VNRDurationDistParam1>
<VNRDurationDistParam2>-1</VNRDurationDistParam2>
<VNRDurationDistParam3>-1</VNRDurationDistParam3>
<VNRArrivalDist>2</VNRArrivalDist>
```

```
<VNRArrivalDistParam1>12.5</VNRArrivalDistParam1>
<VNRArrivalDistParam2>-1</VNRArrivalDistParam2>
<VNRArrivalDistParam3>-1</VNRArrivalDistParam3>
<VNRMaxDistanceDist>1</VNRMaxDistanceDist>
<VNRMaxDistanceDistParam1>15</VNRMaxDistanceDistParam1>
<VNRMaxDistanceDistParam2>25</VNRMaxDistanceDistParam2>
<VNRMaxDistanceDistParam3>-1</VNRMaxDistanceDistParam3>
<VNCPUDist>0</VNCPUDist>
<VNCPUDistParam1>2</VNCPUDistParam1>
<VNCPUDistParam2>20</VNCPUDistParam2>
<VNCPUDistParam3>-1</VNCPUDistParam3>
<VLBWDist>0</VLBWDist>
<VLBWDistParam1>1</VLBWDistParam1>
<VLBWDistParam2>10</VLBWDistParam2>
<VLBWDistParam3>-1</VLBWDistParam3>
<VLDelayDist>0</VLDelayDist>
<VLDelayDistParam1>50</VLDelayDistParam1>
<VLDelayDistParam2>100</VLDelayDistParam2>
<VLDelayDistParam3>-1</VLDelayDistParam3
```

Note that you may change any of the above parameters in configurations.xml file to generate various sizes and scales of SN and VNRs.

6.1 Generating VNR files for varying traffic load

Sample network files only include subset of VNRs, we need to generate VNR network files for a range of arrival rates. VNRs arrive according to a Poisson process with a mean arrival rate of λ requests per unit time. Their lifetimes are exponentially distributed with a mean 1/μ yielding to a VNR traffic of λ × 1/μ Erlangs. For simulation scenarios, we assume 1/μ = 1, 000. Therefore, for 8 requests per 100-unit time, arrival rate parameter is 100/8 = 12.5 and VNR traffic is equivalent to $\frac{8}{100}$ * 1000 = 80 Erlangs.

From this point onwards, we shall require Network File Generator module of the tool. This part was commented out before in step 3. Include all the configurations in configurations.xml file within the tag:

```
<NetworkFileGenerator></NetworkFileGenerator>
```

Then modify VNRArrivalDistParam1 to a range of values (12, 14, 16, 20, 25, 33, 50, 100) and generate VNR files for each value.

```
<VNRArrivalDistParam1>12</VNRArrivalDistParam1>
```

Save the files in a directory outside of vne-sim directory as before -

```
<path>/home/username/network_files/VNRs</path>
```

After modifying configuration file, the tool must be recompiled using make command under *vne-sim/build* listed in step 2 to include NetworkFileGenerator module and related test cases. Next, change directory to *vne-sim/build/bin* to run VNRGenerator test. Source code for the test is in *vne-sim/src/network-file-generator/test directory*. The executable is in *build/bin/* folder after recompiling the tool. To generate VNR files, type in terminal:

```
./nfg-test –run_test=FileGenerator/VNRGenerator
```

6.2 Generating SN files with BRITE

To generate substrate network file with BRITE parameters (same or modified) listed above type:

```
./nfg-test –run_test=FileGenerator/SubstrateNetworkGenerator
```

Save substrate network files with a directory path-

```
<path>/home/username/network_files/SN</path>
```

Now a complete set of files is generated using BRITE handler of VNE-Sim tool which may be used to run tests and produce results as explained in steps 4 and 5.

7. Datacenter Network (DCN) Topology Generation using FNSS Handler

Various network topologies such as Two-Tier, Three-Tier, FatTree, BCube and Diamond may be used to create substrate networks using Fast Network Simulation Setup (FNSS) [12]. Then, these SN graphs may be used together with the VNR graphs produced in step 6 to compare performances over various topologies as described in [2].

First, FNSS handler needs to be selected in configuration.xml file:

```
<NetworkFileGenerator>
  <!-- Select network file generator: BRITE or FNSS-->
  <Handler>FNSS</Handler>
```

Then under */home/username/vne-sim* directory open file CMakeLists.txt and include FNSS option (previously commented out) -

```
option (WITH_FNSS_SUPPORT "WITH FNSS SUPPORT" ON)
```

correct the paths for the FNSS patches in the following way:

```
/home/username/vne-sim/cmake/patches/fnss.fix_quantity_clang.patch
/home/username/vne-sim/cmake/patches/fnss.fix_macos_sed.patch
```

(Make sure that the patches exist under VNE-sim/cmake folder).

After the reorganization of folders in the 0.8.2 version of FNSS, user may encounter the following errors:

Error: cpp/src/quantity.cpp: No such file or directory
Action: Copy "cpp" folder from https://github.com/fnss to FNSS directory in external libraries.

Fatal error: fnss/cpp/include/topology.h: No such file or directory #include "fnss/cpp/include/topology.h"
Action: Rename the folder "src" to "include" and make sure this folder is under cpp (not cpp/cpp).

Make[2]: *** No rule to make target '../external-libs/fnss/cpp/lib/libfnss.a', needed by 'lib/libnfg.so'.
Action: Download C++ zip file from http://fnss.github.io, and unzip it to a convenient location. In terminal cd to the cpp folder in the unzipped file and type "make". This will generate new folders, including "lib", which contains libfnss.a. Copy the folder into ../external-libs/fnss/cpp/

Next to recompile the source code with FNSS handler, change directory to /home/username/vne-sim/build, remove everything from build directory and type in the terminal window:

```
cmake .. -DWITH_FNSS_SUPPORT=on
make
```

VNE-Sim tool already had three DCN topologies implemented with FNSS handler. They are BCube, FatTree and TwoTier topologies. You may select any of these topologies in configurations.xml file and generate SN files as in step 6.

```
<SNTopologyType>DCNFatTree</SNTopologyType>
```

This section describes implementing a new ThreeTier DCN topology in this tool. Implementation of this DCN topology in VNE-Sim is achieved by modifying of the network-file-generator and experiment_parameters packages located in the core directory. In the header file fnss-handler.h, the structure for the three-tier DCN topology is defined based on parameters required by FNSS three-tier class. This class may be found under the external libraries/fnss/fnss/topologies folder. Three-tier DCN parameters include number of core switches (n_core), aggregation switches (n_aggregation), edge switches (n_edge) per aggregation switch, and hosts (n_hosts) per edge switch. Instances for other parameters such as CPU distance, virtual link bandwidth and delay distances are also defined for this topology. After implementing the structure and parameters, six new templates for the network are created. The first one establishes the default name format of the topology file while the second returns the CPU, bandwidth, and delay distance parameters. A third template initializes the FNSS python script to generate the three-tier and a fourth one retrieves the values of the attributes established in the configurations.xml file. The last two templates are used to initialize the structure and parameters. Implemented code is shown in Appendix A1.

We also modified the header experiment-parameters.h and source code experiment-parameters.cc as presented in Appendix A2. In the header file, the internal structure of three-tier DCN topology is included by configuring access method as well as declaring variables for parameters n core, n aggregation, n edge, n host, and core bandwidth multiplier. Variables created are initialized in the source code based on settings provided in configurations.xml file.

References

[1] S. Haeri and Lj. Trajkovic, "Virtual network embedding via Monte-Carlo tree search," IEEE Transactions on Cybernetics, vol. 47, no. 2, pp. 1–12, Feb. 2017.

[2] H. Ben Yedder, Q. Ding, U. Zakia, Z. Li, S. Haeri, and Lj. Trajkovic, "Comparison of virtualization algorithms and topologies for data center networks," The 26th International Conference on Computer Communications and Networks (ICCCN 2017), 2nd Workshop on Network Security Analytics and Automation (NSAA), Vancouver, Canada, Aug. 2017.

[3] S. Haeri and Lj. Trajkovic, "VNE-Sim: a virtual network embedding simulator," SIMUTOOLS, Prague, Czech Republic, Aug. 2016, pp. 112–117.

[4] S. Haeri, Q. Ding, Z. Li, and Lj. Trajkovic, "Global resource capacity algorithm with path splitting for virtual network embedding," IEEE Int. Symp. Circuits and Systems, Montreal, Canada, May 2016, pp. 666–669.

[5] S. Haeri and Lj. Trajkovic, "Virtual network embeddings in data center networks," IEEE Int. Symp. Circuits and Systems, Montreal, Canada, May 2016, pp. 874–877.

[6] (2018, Aug.) Boston University Representative Internet Topology Generator. [Online]. Available: http://www.cs.bu.edu/brite/.

[7] (2018, Aug.) SQLite: Small. Fast. Reliable. Choose any three. [Online]. Available: https://www.sqlite.org/.

[8] (2018, Aug.) CMake Build System. [Online]. Available: https://cmake.org/.

[9] (2018, Aug.) Boost C++ Libraries. [Online]. Available: http://www.boost.org/.

[10] (2018, Aug.) GSL-GNU Scientific Library. [Online]. Available: https://www.gnu.org/software/gsl/.

[11] (2018, Aug.) GLPK-GNU Linear Programming Kit. [Online]. Available: http://www.gnu.org/software/glpk/.

[12] (2018, Aug.) Fast Network Simulation Setup. [Online]. Available: http://fnss.github.io/.

[13] (2018, Aug.) Hiberlite Library. [Online]. Available: https://github.com/paulftw/hiberlite/.

A. Appendix

A1. FNSS-Handler.h Code Sections

Listing 1: Structure Constructor (struct Parameters)

```
struct DCNThreeTier

    {
     DCNThreeTier ();
     int n_core;
     int n_aggregation;
     int n_edges;
     int n_hosts;
     int coreBWMultiplier;
    } threetier;
```

Listing 2: Topology Parameters

```
. . .
private:
int numHosts = 0;
int numSwitches = 0;
Topology_Type _Topology_Type;
Parameters params;
. . .
std::shared_ptr<Network<A, B>> getNetwork_DCNThreeTier
(Distribution cpu_dist, double cpu_param1, double cpu_param2, double cpu_param3,
Distribution bw_dist, double bw_param1, double bw_param2, double bw_param3,
Distribution delay_dist, double delay_param1, double delay_param2, double delay_param3);
};
```

Listing 3: Default Name Format of Topology File (getPreferredFileName())

```
template <typename A, typename B>
std::string FNSSHandler<A,B>::getPreferredFileName ()
{
stringstream ss;
 . . .
else if (_Topology_Type == Topology_Type::DCNThreeTier)
     ss << "_nCore_" << params.threetier.n_core << "_nAggregation_" << params.threetier.n_aggregation << "_nEdges_" <<
params.threetier.n_edges <<
      "_nHosts_" << params.threetier.n_hosts << "_coreMultiplier_" << params.threetier.coreBWMultiplier <<
      "_nHosts_" << numHosts << "_nSwitches_" << numSwitches;
else
ss << "_k_" << params.fattree.k << "_coreMultiplier_" << params.fattree.coreBWMultiplier <<
"_nHosts_" << numHosts << "_nSwitches_" << numSwitches;
return ss.str();
}
```

Listing 4: Substrate Network Parameters (getNetwork)

```
template <typename A, typename B>

std::shared_ptr<Network<A, B>> FNSSHandler<A,B>::getNetwork

(Topology_Type tt, int n, Distribution cpu_dist, double cpu_param1, double cpu_param2, double cpu_param3,

Distribution bw_dist, double bw_param1, double bw_param2, double bw_param3,

Distribution delay_dist, double delay_param1, double delay_param2, double delay_param3)

{

_Topology_Type = tt;
 . . .
else if (tt == Topology_Type::DCNThreeTier)

return getNetwork_DCNThreeTier (cpu_dist, cpu_param1, cpu_param2, cpu_param3, bw_dist, bw_param1, bw_param2, bw_param3,

delay_dist, delay_param1, delay_param2, delay_param3);
```

```
...
```

Listing 5: Substrate Network Generation Using Python Script (getNetwork DCNThreeTier)

```
template<typename A, typename B>
std::shared_ptr<Network<A, B>> FNSSHandler<A,B>::getNetwork_DCNThreeTier
(Distribution cpu_dist, double cpu_param1, double cpu_param2, double cpu_param3,
Distribution bw_dist, double bw_param1, double bw_param2, double bw_param3,
Distribution delay_dist, double delay_param1, double delay_param2, double delay_param3)
{
std::stringstream pythonScript;
Py_Initialize();
pythonScript << "import fnss\n";
//pythonScript << "import networkx as nx\n";
pythonScript << "topology = " << "fnss.three_tier_topology(n_core=" << params.threetier.n_core <<
", n_aggregation" << params.threetier.n_aggregation << ", n_edge= " << params.threetier.n_edges <<
", n_hosts=" << params.twotier.n_hosts << ")\n";
pythonScript << "fnss.write_topology(topology, '.datacenter_topology.xml')\n";
PyRun_SimpleString(pythonScript.str().c_str());
Py_Finalize();
fnss::Topology t = fnss::Parser::parseTopology(".datacenter_topology.xml");
std::set<std::pair <std::string, std::string> > edges = t.getAllEdges();
std::set<std::string> nodes = t.getAllNodes();
assert (nodes.size() > 0 && edges.size() > 0);
std::shared_ptr<Network<A, B>> net (new Network<A, B>());
std::map<std::string, int> fnssNodeIdToVNESimNodeId;
for(set<string>::iterator it = nodes.begin(); it != nodes.end(); it++)
{
fnss::Node fnssNode = t.getNode(*it);
std::shared_ptr<A> n = nullptr;
// If the node is a host create it with a cpu capacity
if (fnssNode.getProperty("type").compare("host") == 0)
{
numHosts++;
double node_cpu = RNG::Instance()->sampleDistribution<double,double,double,double>
(cpu_dist, std::tuple<double,double,double> (cpu_param1, cpu_param2, cpu_param3)) ;
n.reset (new A (node_cpu, 0, 0));
}
else
{
numSwitches++;
n.reset (new A (0,0,0));
}
fnssNodeIdToVNESimNodeId[*it] = n->getId();
net->addNode (n);
}
for(set<pair <string, string> >::iterator it = edges.begin(); it != edges.end(); it++)
{
double link_bw = RNG::Instance()->sampleDistribution<double,double,double,double>
(bw_dist, std::tuple<double,double,double> (bw_param1, bw_param2, bw_param3));
double link_delay = RNG::Instance()->sampleDistribution<double,double,double,double>
(delay_dist, std::tuple<double,double,double> (delay_param1, delay_param2, delay_param3));
std::shared_ptr<B> l = nullptr;
int nodeFromId = fnssNodeIdToVNESimNodeId[(*it).first];
fnss::Node fnssNodeFrom = t.getNode ((*it).first);
int nodeToId = fnssNodeIdToVNESimNodeId[(*it).second];
fnss::Node fnssNodeTo = t.getNode ((*it).second);
if ((fnssNodeFrom.getProperty("type").compare("host") == 0 || fnssNodeTo.getProperty ("type").compare("host") == 0 ))
{
l.reset (new B (link_bw, link_delay, nodeFromId, nodeToId));
}
else
l.reset (new B (params.twotier.coreBWMultiplier * link_bw, link_delay, nodeFromId, nodeToId));
net->addLink (l);
}
this->pt.put ("n_switches", numSwitches);
this->pt.put ("n_hosts", numHosts);
this->pt.put ("n_links", net->getNumLinks());
return net;
}
```

Listing 6: Retrieval of Values for DCN Three-Tier Parameters from \con_guration.xml" File

```
template <typename A, typename B>
FNSSHandler<A,B>::Parameters::DCNThreeTier::DCNThreeTier () :
n_core(ConfigManager::Instance()->getConfig<int>("NetworkFileGenerator.FNSSHandler.DCNThreeTier.n_core")),
n_aggregation(ConfigManager::Instance()->getConfig<int>("NetworkFileGenerator.FNSSHandler.DCNThreeTier.n_aggregation")),
n_edges(ConfigManager::Instance()->getConfig<int>("NetworkFileGenerator.FNSSHandler.DCNThreeTier.n_edges")),
n_hosts(ConfigManager::Instance()->getConfig<int>("NetworkFileGenerator.FNSSHandler.DCNThreeTier.n_hosts")),
coreBWMultiplier (ConfigManager::Instance()->getConfig<int>("NetworkFileGenerator.FNSSHandler.DCNThreeTier.coreBWMultiplier"))
{
}
```

Listing 7: DCN Three-Tier Structure Initialization

```
template<typename A, typename B>
FNSSHandler<A,B>::Parameters::Parameters () :
...
   threetier(DCNThreeTier())
{
}
```

Listing 8: DCN Three-Tier Parameters Initialization

```
template <typename A, typename B>
FNSSHandler<A,B>::FNSSHandler () :
ExternalLibHandler<A,B> (),
params(Parameters())
{
. . .
this->pt.put ("DCNThreeTier.n_core", params.threetier.n_core);
    this->pt.put ("DCNThreeTier.n_aggregation", params.threetier.n_aggregation);
    this->pt.put ("DCNThreeTier.n_edges", params.threetier.n_edges);
    this->pt.put ("DCNThreeTier.n_hosts", params.threetier.n_hosts);
this->pt.put ("DCNThreeTier.coreBWMultiplier", params.threetier.coreBWMultiplier);
}
```

A2. Experiment Parameters Code Sections

Listing 9: Hiberlite Access to DCN Three-Tier Parameters (experiment-parameters.h)

```
class ExperimentParameters
{
friend class hiberlite::access;
template<class Archive>
void hibernate(Archive & ar)
{
. . .
ar & HIBERLITE_NVP (sn_three_tier_core);
ar & HIBERLITE_NVP (sn_three_tier_aggregation);
ar & HIBERLITE_NVP (sn_three_tier_edge);
ar & HIBERLITE_NVP (sn_three_tier_host);
ar & HIBERLITE_NVP (sn_three_tier_core_bw_multiplier);
ar & HIBERLITE_NVP (sn_fat_tree_k);
ar & HIBERLITE_NVP (sn_fat_tree_core_bw_multiplier);
}
```

Listing 10: DCN Three-Tier Topology Parameters Variables De_nition (experiment-parameters.h)

```
class ExperimentParameters
. . .
private :
 . . .
int sn_three_tier_core;
int sn_three_tier_aggregation;
int sn_three_tier_edge;
int sn_three_tier_host;
int sn_three_tier_core_bw_multiplier;
int sn_fat_tree_k;
```

```
int sn_fat_tree_core_bw_multiplier;
}
```

Listing 11: DCN Three-Tier Topology Parameters Variables Initialization (experiment-parameters.cc)

```
void ExperimentParameters::setSNNetParams (boost::property_tree::ptree &pt, Topology_Type tt)
{
 . . .
else if (tt == Topology_Type::DCNThreeTier)
{
sn_dcn_n_switches = pt.get<int> ("n_switches");
sn_dcn_n_hosts = pt.get<int> ("n_hosts");
sn_dcn_n_link = pt.get<int>("n_links");
sn_three_tier_core = pt.get<int>("DCNThreeTier.n_core");
sn_three_tier_edge = pt.get<int>("DCNThreeTier.n_aggregation");
sn_three_tier_edge = pt.get<int>("DCNThreeTier.n_edges");
sn_three_tier_host = pt.get<int>("DCNThreeTier.n_hosts");sn_three_tier_core_bw_multiplier =
pt.get<int>("DCNThreeTier.coreBWMultiplier");
}
```