NePA TesT: Network Protocol and Application Testing Toolchain for Community Networks

Luca Baldesi*, Leonardo Maccari*
*Department of Information Engineering and Computer Science, University of Trento, Italy
{luca.baldesi, leonardo.maccari}@unitn.it

***Abstract*—A Community Network (CN) is a bottom-up network infrastructure that interconnects communities of people and represents a promising and successful networking model. The more people join a CN, the higher is the demand for a wider range of community services and the need for scalable network protocols. CN designers and passionates hence require adequate frameworks to develop and test different technical solutions, that must reproduce realistic environments and allow a rapid prototyping and deployment. We propose a full-comprehensive framework that can be used to develop new protocols and applications both in a flexible and portable manner. NePA TesT (Network Protocols and Applications Testing Toolchain) is based on the Mininet emulator enriched with: a library capable of creating random synthetic topologies with the most up to date realistic topology generators, a data set with topologies extracted from real world CNs, and powerful extensions to dynamically run and control the emulated scenarios. This paper will analyse and document each component and show how NePA TesT can be used in two realistic application scenarios.**

I. Introduction

In the past few years the performance of long-range wireless devices has increased dramatically, and their price has lowered. Today, the market offers for less than 100 € outdoor devices that can be used for long range links with a real throughput of hundreds of Mbps. As a consequence those CNs that emerged at the beginning of the 2000s, today, grew up to thousands of nodes. Freifunk in Germany, Wlan-Slovenia, FunkFeuer in Austria, Ninux in Italy, AWMN in Greece and most notably Guifi.net[1] in Spain are examples of mesh networks, mostly realized with wireless links but in some cases with a mixed wireless/fiber technology, that range from a few hundreds of nodes to tens of thousands. CNs grow as they take advantage of the so-called *network effect*: the largest the network, the largest is its growing pace. In fact, the chances for a new person to join a network depends on the availability of active nodes in his own neighborhood, so the larger and denser is the network, the easier is to attract new participants.

Mesh networks that scale to hundreds of nodes must be made of wireless links with directional antennas to minimize cross-link interference. This approach alleviates the challenges related to the physical and data link layer and produces topologies similar to the ones we observe on wired networks. Still, compared to wired networks, the wireless media is dynamic, its capacity is limited, and thus there is a need for specific network protocols that can face scalability, availability and performance challenges. Moreover, the increasing number of users accelerates the demand for a wider range of internal community services, from the classical ones (mailing lists, chat, file sharing) to the more modern and demanding ones (cloud systems, video chat, distributed filesystems). As it happens for the protocols, also the services need specific tuning to scale and exploit the distributed nature or CNs. Therefore, to support the growth of CNs, researchers, designers and passionates (from now on simply "developers") require adequate frameworks to develop and test different technical solutions from network layer up.

While this challenge in the academia is approached with a widespread use of network simulators, this solution does not fit this use case, for two main reasons. The first is that physical and link-layer features can not be modified in COTS hardware, so there is no real reason to simulate them with their whole complexity, the second is that developers want to apply their results to real networks and source code developed for network simulators can not run on real hardware.

This paper describes NePA TesT, a developer framework based on the Mininet emulator created to fill this gap and let developers test protocols and applications for CNs. The Mininet emulator makes it easy to test programs in an emulated environment that will natively run on real Linux platforms, and NePA TesT adds a set of indispensable tools to make the emulation environment as close as possible to reality.

NePA TesT includes a topology parser that allows the easy definition of different network topologies, it implements several state-of-the-art topology generators, from the classical ones to the ones specifically designed for CNs, it allows a fine-tuned configuration of the emulation parameters and an easy monitoring of the host machine resources. Moreover, it includes a database of topologies and other network parameters (such as loss and delay) derived from real networks, which can be extended via the support to the NetJSON format, a description format that is being defined by the communities to exchange information about their networks. NePA TesT thus represents an indispensable toolchain to develop and test software that will be ready to be deployed on real CNs and mesh networks.

NePA TesT is open-source software and it is freely accessible on-line[2].

The paper is organized as follows: Sec. II briefly introduces the state of the art on network emulation, Sec. II-A gives a

[1] http://freifunk.net/, https://wlan-si.net, http://funkfeuer.at/, http://ninux.org, http://www.awmn.net, https://guifi.net

[2] https://ans.disi.unitn.it/redmine/wcn_emulator.git

ISBN 978-3-901882-80-7 © 2016 IFIP

functional overview of the three key components of NePA TesT, Sec. III documents the interfaces that developers can use to write and run emulations, Sec. IV shows the application of NePA TesT to two real world scenarios, and finally Sec. V draws the conclusions and outlines future works.

II. Related Works

To test and develop network protocols and applications, three approaches are typically used. The first one is to use real testbeds, whether in their own lab or accessible on remote interfaces, like Emulab [1], Planetlab [2] and Community-lab [3]. While these systems offer a good flexibility, their complexity is high, the access and their scale is limited, and, often they are far from being close to a production network. A second approach, in order to manage an entire network on a single machine, is to use network simulation. Network simulators like ns-3 [4], OMNeT++ [5] or Opnet [6] are widely used in the academia and implement all the network layers via software, approximating the communication channel with mathematical models. However, this approach has the big limitation that the source code used for the tests can not be re-used as-is in real networks, so that the majority of the proposed techniques remain on their published papers and are hard to reproduce in real world scenarios. A third approach is given by network emulators that allow the testing of deployable code, since they set-up a virtual infrastructure that runs the same operating system of the target environment. NEmu [7] and Naxim [8] take advantage of QEMU [9], a machine virtualizer, to emulate an entire network. However this kind of emulation is much more resource hungry than network simulation and requires powerful hardware to run small-scale emulations. Note that ns-3 supports the interaction with Linux containers (OS-level machine virtualization) however, its configuration is not straightforward and the container overhead in terms of disk usage is not negligible. At the core of NePA TesT stays the Mininet emulation platform which exploits kernel namespaces, the base component of Linux containers. Kernel namespaces can be used to exclusively assign a set of resources of a specific kind (i.e., network resources) to a certain process. Thus, they can be used to emulate processes running on different hosts without the overhead introduced by the execution of several Linux containers.

A. Mininet

Mininet [10] is a lightweight system for easy setup of an emulated network to test deployable code. The Mininet emulated nodes have separated network environments but share the same filesystem, memory and cpu resources, that is far less resource-hungry compared to full host virtualization. This technique enables the networks to scale up to hundreds of emulated hosts on a single laptop. Mininet does not provide an intuitive interface to ease the typical developing and testing workflow, instead it delegates everything to the source code realized by developer himself. Mininet has been created for experimenting with Software Defined Networks (SDN) and there are several works in this direction [11], [12], [13]. We instead developed NePA TesT as a full toolchain which includes all the components needed to realize a realistic emulation of a network. For this purpose we had to patch Mininet[3] in order to support some key features, such as the possibility to assign different delay and loss settings to every network link.

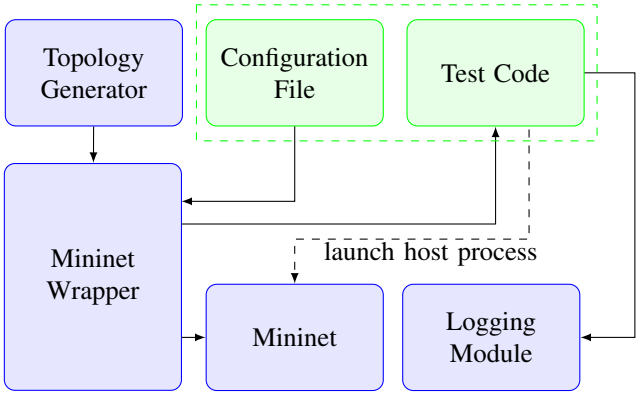

Fig. 1: NePA TesT architecure, the user interface is depicted in green.

NePA TesT enhances the Mininet framework with three new components that allow the realization of complex emulated networks. The three components are a topology generator and parser, a Mininet wrapper that allows the flexible configuration of each emulation scenario and a logging module that helps to monitor the performance of the operating system on which the emulation is run. Fig. 1 depicts the architecture.

B. Topology generator and parser

The performance of network protocols and applications is strongly influenced by the underlying network topology, and in particular by the size, the density, and the degree sequence of the corresponding graph. For this reason, when doing network emulations the choice of a topology that is a close approximation of the network in which the software will run is a key factor. Furthermore one single topology is not sufficient to have statistically sound results, since multiple runs of each test should be performed on similar topologies. Finally, to widen the applicability of the results the parameters that define the topology need to be modifiable (for instance, increasing the number of network hosts). The generation of multiple random but realistic topologies is, in itself, a non-trivial task. NePA TesT comes with a companion library that allows the generation of synthetic networks with three well-known graph generators and two generators derived from data measured on real wireless community networks. The first three generators are wrapper functions around the Barabasi-Albert, Erdős and Watts-Strogatz graph generators present in the NetworkX Python library. These generators produce graphs with well known properties, so they are very useful to test protocols and applications in a controlled environment. NetworkX generators do not have a compatible interface so, for example, the Erdős graph generator takes as input the number of desired nodes and a per-edge probability, while

[3]https://ans.disi.unitn.it/redmine/mininet.git

the Barabasi-Albert generator takes as input the number of nodes and the number of outgoing edges per each node. This means that it is hard to generate graphs with the same density (ratio between edges and nodes), or, in some cases even with a predictable number of edges. We uniformed the interfaces so that the researcher can test the software on synthetic graphs with different properties but with similar densities.

Two more generators are included in NePA TesT using algorithms from Milic and Malek [14] and Cerdà-Alabern [15]. The first algorithm is designed from the observation of several German CNs, and it is based on the geographical placement of nodes with some constraints that produce topologies with similar features to the observed ones. The second one derives from the statistical analysis of large portions of the Guifi community networks.

NePA TesT also comes with a set of network topologies extracted from the observation of three real wireless community networks. The three networks are the FunkFeuer networks from the cities of Wien and Graz in Austria, and the Ninux network from the city of Rome. These networks have been monitored for a week and the extracted data was analysed and summarised [16]. Together with the simulator a subset of the available topologies (ranging from 126 to 227 nodes) is included to test the protocols and applications on real topologies. As a last option, NePA TesT supports topologies expressed in the NetJSON format. NetJSON is "*a data interchange format based on JavaScript Object Notation (JSON) designed to describe the building blocks of the layer 2 and layer 3 of networks*"[4]. The NetJSON specifications is currently under development, but its main features are already defined, so that an initial support is included in NePA TesT. Already several routing protocols for mesh networks support the automated generation of the network topology (at least the portion they are aware of) in NetJSON natively or through the `netDiff` conversion library (part of the NetJSON reference implementation). It is the case of the widely used OLSRd and Batman-adv routing daemons, implementations of the OLSR [17] and B.A.T.M.A.N. [18] standards respectively. As a reference we mention the effort done by the Freifunk German community, which is building an index of all the (tens) of networks in Germany in a machine-readable form[5]. From the repository of networks, the URL of the topologies exported by some of the communities is available, and it can be translated into NetJSON format with the `netDiff` utility. With time, this will be a source of tens of different network topologies of various size that developers can use to test their protocols and applications.

With all the described options, NePA TesT gives to the developers a powerful platform to design their protocols and test their performance on both synthetic topologies and real topologies, including the topology of a specific target network imported via NetJSON.

[4]See the NetJSON project, www.netjson.org

[5]https://github.com/freifunk/directory.api.freifunk.net/blob/master/directory.json

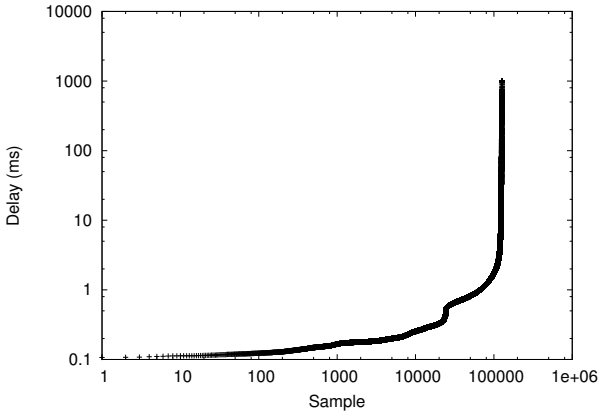

Fig. 2: The whole sample set of the measured delays in the qMp Sants-UPC network (log-log scale).

C. Mininet wrapper

Mininet is a powerful network emulator that exploits the namespace isolation of the Linux kernel to have several independent processes running simultaneously with separated resources, but it is not specifically tailored to emulate CNs. NePA TesT provides an extension to the Mininet class space that allows the developers to plug-in their code in a modular way, without caring of the underlying details of the system. Developers need to extend a specific class and to include their code in a few methods that get called by the emulator.

NePA TesT enriches Mininet with a configuration system based on the use of .ini files, that the user can organize in a hierarchical way, in order to reproduce the typical network emulation and simulation workflow, which includes repeating the emulation with different parameters, different topologies and different random seeds. Each test inherits some configuration parameters from a common parent class needed to configure the underlying network. Among the important configuration parameters, we mention the file expressing the topology, and, for each edge of the network, the maximum bit rate, loss, and delay, the last two values being expressed as a fixed parameter or as a histogram from which to randomly extract each single sample. The real topologies shipped with NePA TesT include in their links the ETX value (a metric widely used in mesh networks that estimates the probability of correctly delivering a packet on a link). From the available data-sets and literature [15] we have also included a default distribution for the delay value, obtained from tests on real networks. The delay has been measured on the qMp Sants-UPC community network, a portion of the Guifi network made of about 60 nodes and configured with directional antennas and 802.11n radios. For several weeks on every link in the network the RTT has been measured with the `ping` shell command. We were able to access such data and we approximated the delay as half of the RTT (which, considering that 802.11 needs to acknowledge data packets, it is a reasonable approximation at least for small unicast packets even on links that have asymmetric performance).

Figs. 2 and 3 report the samples of the distribution included into NePA TesT and the frequency of each delay value. The

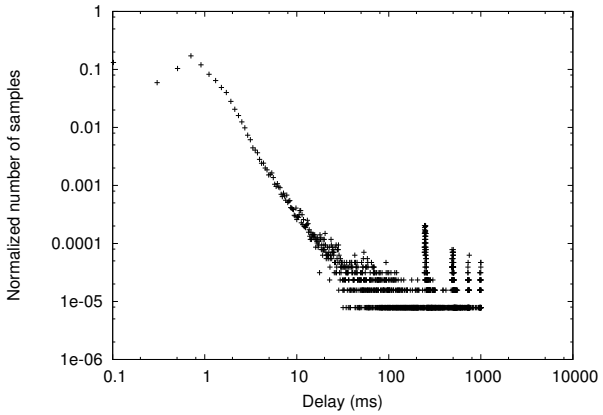

Fig. 3: The frequency of the delay samples, binned with 5000 intervals on the x axis (log-log scale).

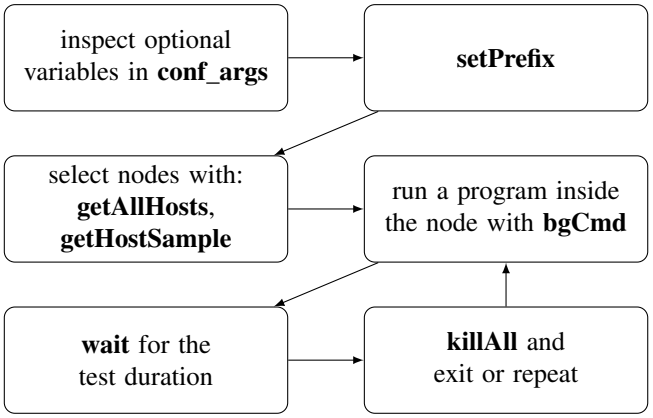

Fig. 4: Typical flow of the **runTest** function of a test class.

majority of the delays falls below one ms with a few samples larger than 10 ms. These values are compiled in order to be used by the `netem` Linux command, once compiled, the data file needs to be saved in a system library where `netem` can find it, and then it can be used by Mininet. The data file express the delay distribution as a function of two parameters, mean and sigma that represent the location and scale parameters of the distribution. NePA TesT includes the data file for the distribution in Fig. 3 and accepts mean and sigma values that can be used by the developer to stretch and shift the given distribution, as shown in Sec. III-C.

D. The logging Module

The logging module is a simple but effective way to keep under control the host on which the emulation is performed. One of the huge differences between an emulator and a simulator is that the first one runs in real time, and thus, if the emulation is not properly dimensioned, the host system becomes a bottleneck and influences the results. This happens for instance when the CPU or memory of the system can not cope with the number of the emulated applications, and some applications starve for the lack of resources. While a network simulator solves this problem with an internal clock that can be slowed down as much as necessary, in emulation there is only one system clock and CPU overload influences the results of the emulation.

The logging module can be enabled easily from the developers in order to keep track of some key performance indicators, such as the CPU load and the memory and swap occupation, so that each phase of the emulation can be monitored and the developer can easily understand when the host machine is potentially influencing the emulation results.

III. NePA TesT Interface

Each test in NePA TesT is defined by a triple formed by a Python class, a network graph and a configuration stanza in a configuration file. The configuration file name and the stanza are passed as command line arguments, all the other parameters are normally included in the configuration variables, including the network graph file, but can be also overridden with command line arguments, as shown in List. 1 and detailed in Sec. III-D.

List. 1: Invocation of NePA TesT using the configuration stanza OLSRTest defined in conf/olsr.ini.

```
python wcn_emulator -f conf/olsr.ini -t\
    OLSRTest
```

A. Test code

For each experiment a developer wants to realize, he has to add a Python class derived from the `MininetTest` class which has the following basic methods:

- **runTest**: method invoked by NePA TesT to start the test;
- **getAllHosts**: get a list of hosts in the network;
- **getHostSample**: get a random sample of hosts of size *n*;
- **bgCmd**: make a host run a specific command, i.e. a network protocol or an application;
- **sendSig**: send a signal to a specific host;
- **killAll**: terminate all running processes launched by the hosts;
- **setPrefix**: set the default folder for storing experiment data and logs;
- **wait**: sleep for a given period of time and optionally log host machine resources.

The **runTest** function must be overloaded and is used to pass the control from the main of NePA TesT to the specific test code. Fig. 4 shows a typical sequence of calls in a custom **runTest** function.

B. Network graph

The network graph definition can be given in two formats, the first is the well known edge-file format, in which each line represents an edge of the network, in the simple form "`Src Dst ETX`" where the ETX value is to be interpreted as the ETX metric used by several wireless routing protocols. The second is the mentioned NetJSON format, which we do not report for brevity.

C. The Configuration File

A typical configuration file is an .ini file similar to the one in List. 2, in which the configuration for the OLSRd daemon (the open-source implementation of the Optimized Link State Routing protocol) is sketched.

List. 2: An example configuration file

```
[OLSRTest]
testModule = olsr
testClass = OLSRTest
times = 5
graphDefinition = data/test.edges
shortestPathComputation = false
link_mean_delay = <mean>ms
link_delay_sd = <sigma>ms
link_delay_distribution = mydist
link_delay_pfifo = True

[OLSRConvergence:OLSRTest]
duration = 35
HelloInterval = 1
TcInterval = 2
startLog = 1s
stopLog = 35s
```

The file is a text file divided in stanzas. In the OLSRTest configuration stanza some generic parameters valid for all the OLSR emulations are included, plus details of the emulated network. In detail:

- `testModule`: the name of the python module where the emulation code is contained;
- `testClass`: the class that must be loaded to perform the test;
- `times`: the number of times the emulation has to be run with different random seeds;
- `graphDefinition`: the network graph that has to be used to run the scenario;
- `shortetPathComputation`: if set to true, compute the shortest path between any couple of nodes and populate the nodes routing tables. In the example it is set to false, since OLSR performs this task;
- `link_delay_distribution`: the name of the file that contains the parametric distribution of the link delays;
- `link_mean_delay`: the mean of the delay distribution;
- `link_delay_sd`: the sigma of the delay distribution;
- `link_delay_pfifo`: the default behaviour of the output queue. With `true` the packets are not reordered, independently of the delay they are assigned, while with `false` packets may be sent on the output interface in a different order than they were pushed in the interface.

The interested reader can refer to the documentation of the `netem` application for the precise meaning of the delay parameters.

The OLSRConvergence stanza inherits from the general OLSRTest configuration stanza and contains specific configuration parameters that are passed in a Python data structure to the OLSRTest python class. The OLSRTest code uses the parameters to configure the OLSR daemon and run the emulation.

D. Command Line Parameters

The `wcn_emulator` binary accepts a few command line parameters, among which the most relevant ones are:

- `-f`: the name of the configuration file to load;
- `-t`: the name of the configuration stanza to parse;
- `-o`: a comma separated string of configuration parameters that are passed to the emulator and override the ones in the configuration file (this option is useful to launch the emulator in a batch script changing some relevant emulation parameters);
- `-g`: the topology to use. A shortcut to override the `graphDefinition` parameter;
- `-s`: a random seed to initialize the random number generators. Needed to have repeatable results.

The topology generator is a separate library that can be used to generate the wanted topologies. The most relevant command line parameters are the following ones:

- `-t`: the kind of topology to generate, detailed below;
- `-g`: the number of graphs to generate;
- `-n`: the number of nodes (n);
- `-e`: the (average) number of edges per node. If not specified it is set to $\frac{\ln(n)}{2n}$ which is the value that makes a symmetric Erdős graph connected with high probability.

The kind of graph to be generated can be:

- `CN`: the community network generator algorithm from Cerdà-Alabern;
- `MM`: the mesh network generator algorithm from Milic and Malek;
- `RE`: a regular graph;
- `WS`: a Watts-Strogatz graph;
- `PL`: a Power-Law graph (with the Barabási-Albert algorithm);
- `ER`: an Erdős graph.

All the original algorithms are tailored to accept the number of nodes and number of edges as input parameter and are preconfigured with sane defaults for the missing ones when needed.

IV. Protocols and Application Testing

In this section we report the results we have with two different testing scenarios, the first one explores the performance of the OLSR routing protocol while the second one explores the performance of PeerStreamer[6], a peer-to-peer video streaming platform. We use these examples to show how NePA TesT can be beneficial for the design of protocols and applications and to outline some of its specific features.

A. OLSR

OLSR is a link-state routing protocol widely used in wireless mesh networks, it instructs nodes to periodically send control messages (HELLO and TC packets in OLSR

[6]http://peerstreamer.org

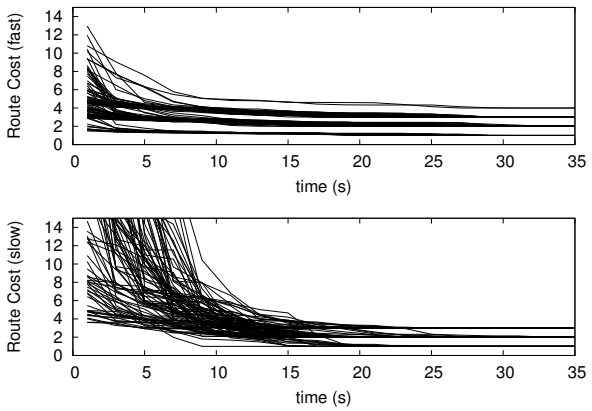

Fig. 5: The cost of 100 random routes in a 50 nodes Erdős graph, with two sets of timers for the HELLO and TC generation (1 s, 2 s for "fast", 5 s and 10 s for "slow")

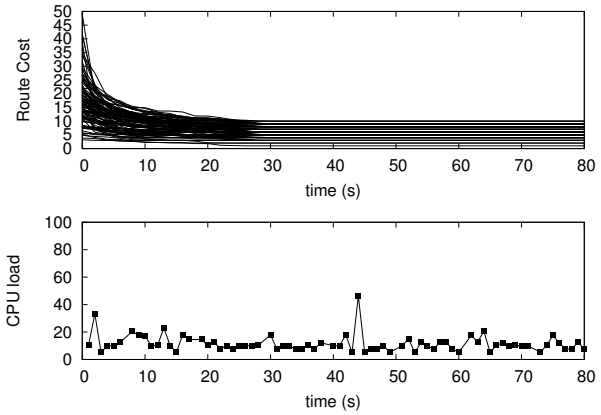

Fig. 6: The cost of 100 random routes in the Ninux topology with 112 nodes and 136 edges ("fast" configuration) and the CPU load during the emulation

terminology) that are used by the other nodes to reconstruct the whole network topology as a weighted graph. Each link has a quality parameter ranging from one (best quality) to infinite (broken link) that depends on the rate of received and lost HELLO messages from the neighbors. Each node uses the Dijkstra algorithm to compute the shortest path and set the next hop to any destination. We modified the open source OLSRd routing daemon in order to periodically dump the routing table of every node, once the emulation is over, we parse all the dumped routing tables and we check when the routes stabilize. A route stabilizes when its cost (the sum of the quality of all the links that compose it) stops fluctuating. For simplicity in this test we set to zero the loss in any link, so that all the links will eventually reach a quality that equals 1 and all the route cost will converge to some integer value. The convergence speed is influenced by many factors, among which the timers used for the generation of the control messages.

Fig. 5 reports the cost for a random set of 100 routes computed on an Erdős graph with 50 nodes and 175 edges. The figure reports results for two different runs (labeled with "fast" and "slow" in the figure) in which the timer of the HELLO and TC messages were set to 1 s and 2 s and 5 s and 10 s respectively. It is clear that there is a dependency from the timers, not much in the time that it takes to stabilize all the routes (in both cases the routes stabilize around 30 s) but in the speed of convergence. In the fast case, at second 5 the costs are already clearly grouped around the values they will finally take. In the slow case, this become visible only after second 20. A faster convergence is achieved at the cost of a much higher overhead in terms of control messages, so this example shows how NePA TesT can be used by developers to get the needed insights to be able to fine-tune the OLSR parameters for their specific needs.

We use the OLSR protocol to showcase another important feature of NePA TesT, that is the capability of monitoring the system resources, which is essential to ensure the realism of the results. Fig. 6 reports the cost of 100 random routes measured on the topology of a real network (the Ninux network topology) on a 4-core 2.5GHz Intel processor with 8G of RAM. The CPU load stays below 50% during the whole emulation, with some fluctuations due to other applications running on the host. The convergence time is similar to the values reported in Fig. 5 (timers are configured as in the "fast" case), and routes stabilize after about 30 seconds. Fig. 7 instead reports the same values for a 100 nodes, 400 edges Erdős graph. The CPU is always running at 100% and the routes never converge. In fact every change in the weight of a link triggers the recomputation of Dijkstra's algorithm for all the OLSRd daemon instances, for every destination. Since OLSRd is a single-process software, the time spent in the shortest path computation will delay the generation of control messages, which will be interpreted from neighbor nodes as a packet loss. In practice, the lack of sufficient CPU resources will produce the fictitious fluctuations of link quality values. If link quality fluctuates, in turn continuous recomputations of the routing tables are triggered, which produces an even higher load on the CPU and ignites a cascading effect. Fig. 7 shows that even if initially the route cost seems to converge, some routes never actually reach a steady state. Note that under such circumstances the network may still deliver traffic, but its behaviour diverges from a real network. It is also interesting to note that since the complexity of the Dijkstra's algorithm depends both on the number of nodes and the number of edges, the Ninux topology, even if it has a larger number of nodes reaches a steady state, while the Erdős network, being much denser never reaches a steady state. Summing up, it is extremely hard to tell in advance which network may or may not saturate the system, and NePA TesT helps the developer to monitor the system parameters to filter out wrong measures that can pollute the final results.

B. Live Video Streaming

Video streaming is a popular service and given the cooperative nature of the WCN it is likely that some users will eventually broadcast a video streaming content, such as video conferences or live events. We use NePA TesT to assess the performance and tune the parameters of Peerstreamer [19], an open-source live streaming platform capable of distributing

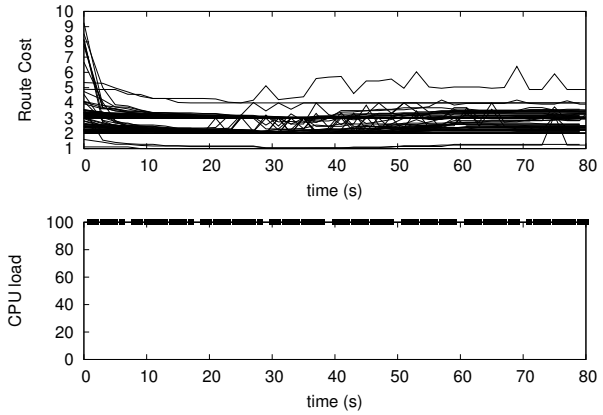

Fig. 7: The cost of 100 random routes in an Erdős graph with 100 nodes and 400 edges ("fast" configuration) and the CPU load during the emulation

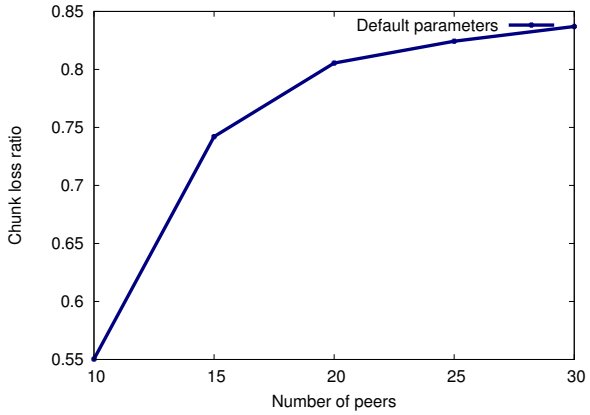

Fig. 8: Average loss ratio distributing a live content with Peerstreamer and the Ninux topology using the default parameter values.

a media content from a simple file or generated with a USB camera. Peerstreamer has been designed to address the challenges of live streaming on the Internet, since the structure of a WCN is completely different we take advantage from the wide range of tuning parameters available on Peerstreamer to tailor its behavior accordingly. We have already conducted an initial evaluation campaign of Peertreamer in the WCN context using both the Community-lab testbed and our lightweight emulation environment [20].

Peerstreamer is a peer-to-peer application operating on a mesh overlay. Peers continuously send each other gossiping messages in order to discover their neighbours and form a connected overlay. On top of that, a special peer called source injects periodically in the overlay small "chunks" of the video stream. If the source has enough network resources, multiple copies of the video can be injected in parallel, to have a more robust delivery. Peers exchange with their neighbours the received chunks in order to obtain a best-effort distribution of the whole content. The number of copies of each chunk injected by the source is a configurable parameter that depends on the specific network conditions.

Peerstreamer has been developed for live content, as such, the receiving ratio of the chunks and the receiving delay are equally important. In our experiment we assume that chunks received four seconds after their generation time are no more useful and must be discarded.

To get a taste of the potential of NePA TesT, we test Peerstreamer using a topology from the real world, namely Ninux, composed of 112 nodes and 136 edges. Our tests last for six minutes and we analyze the data in the last five minutes in order to avoid transient behaviors. We stream the "Big Buck Bunny"[7] short movie encoded at $300\,\mathrm{kbit\,s^{-1}}$. We use the default link delay distribution[8], and configure link loss according to the corresponding ETX value. All the code we use for our experimentation is freely available on-line[9]. We

[7]https://peach.blender.org/

[8]https://ans.disi.unitn.it/redmine/attachments/download/64/qmp_delay_m8.0658_s55.7166.dist

[9]https://ans.disi.unitn.it/redmine/nepatest_peerstreamer.git

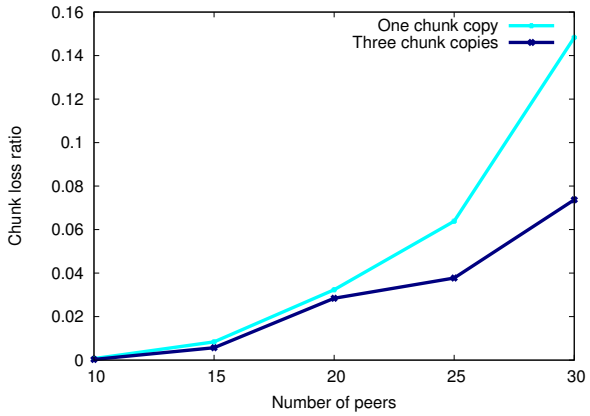

Fig. 9: Average loss ratio distributing a live content with Peerstreamer on the Ninux topology after adjusting the parameters.

try Peerstreamer with different numbers of peers scattered randomly in the topology, setting the neighbour number to ten and we repeat each scenario for ten times.

Fig. 8 shows the performance of Peerstreamer using the default (very restrictive) parameter values. Even with a small number of peers Peerstreamer cannot be used as-is in the Ninux topology for live streaming, in fact a loss of 55% prevents any type of streaming service. However, we showed in a previous work [20] that a suitable set of configuration values leads to excellent performance of Peerstreamer in WCNs. For the sake of brevity, here we only present the dependency on one single parameter; Fig. 9 shows the loss rate when sending one or three initial copies of each chunk, leaving away all the combinations of the other parameters. The loss ratio resulted after the optimization confirms that NePA TesT is a valuable instrument to assess and tune the performance of high-level applications in the context of WCNs.

V. Conclusions and Future Works

We presented NePA TesT, a set of instruments that can be used to perform tests in realistic environments and emulate the behaviour of large networks. NePA TesT wraps the Mininet

emulator, complements some of its missing features, and completes it with a set of tools that help the developer to design and test protocols and applications that will then be ready to be deployed on real networks.

We have shown that NePA TesT supports the emulation of networks of considerable size on a standard PC, it can be used to perform tests on different topologies with sane presets, and allows the monitoring of the host machine, which is a key feature to ensure the consistency of the results (a task that is often neglected in simulation-based results). We have shown also how NePA TesT can be used both at the network and application layer.

The current limitation of NePA TesT is its impossibility of emulating lower layers, e.g. physical and MAC layer, so that the network behaves like a wired switched network. In the future we plan to address this issue with the integration of specific code for wireless link simulation. The ns-3 network simulator in fact can be used to create Linux `tap` devices that communicate with each other simulating a wireless channel, while the `mac80211_hwsim` Linux module allows natively the usage of the Linux wireless drivers on emulated devices. The interaction of these pieces of software and their integration with NePA TesT (including an easy configuration scheme to represent link-layer properties of links) will be carried out in the future.

VI. Acknowledgements

This work was partially financed by the University of Trento with the grant for the project *Wireless Community Networks: A Novel Techno-Legal Approach*.

References

[1] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 255–270, 2002.

[2] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: an overlay testbed for broad-coverage services," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 3, pp. 3–12, 2003.

[3] A. Neumann, I. Vilata, X. Leon, P. Garcia, L. Navarro, and E. Lopez, "Community-lab: Architecture of a community networking testbed for the future internet," in *Wireless and Mobile Computing, Networking and Communications (WiMob), 2012 IEEE 8th International Conference on*, Oct 2012, pp. 620–627.

[4] "The network simulator - ns-3." [Online]. Available: https://www.nsnam.org/

[5] A. Varga *et al.*, "The OMNeT++ discrete event simulation system," in *Proceedings of the European simulation multiconference (ESM2001)*, vol. 9, no. S 185. sn, 2001, p. 65.

[6] "Opnet modeler." [Online]. Available: www.opnet.com/solutions/network_rd/modeler.html

[7] V. Autefage and D. Magoni, "Network emulator: a network virtualization testbed for overlay experimentations," in *Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), 2012 IEEE 17th International Workshop on*. IEEE, 2012, pp. 266–270.

[8] K. Nakajima, S. Kurebayashi, Y. Fukutsuka, T. Hieda, I. Taniguchi, H. Tomiyama, and H. Takada, "Naxim: A fast and retargetable network-on-chip simulator with qemu and systemc," *International Journal of Networking and Computing*, vol. 3, no. 2, pp. 217–227, 2013.

[9] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[10] B. Lantz, B. Heller, and N. McKeown, "A Network in a Laptop: Rapid Prototyping for Software-defined Networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: ACM, 2010, pp. 19:1–19:6. [Online]. Available: http://doi.acm.org/10.1145/1868447.1868466

[11] B. Lantz and B. O'Connor, "A Mininet-based Virtual Testbed for Distributed SDN Development," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. ACM, 2015, pp. 365–366.

[12] R. de Oliveira, A. Shinoda, C. Schweitzer, and L. Rodrigues Prete, "Using Mininet for emulation and prototyping Software-Defined Networks," in *Communications and Computing (COLCOM), 2014 IEEE Colombian Conference on*, June 2014, pp. 1–6.

[13] K. Kaur, J. Singh, and N. S. Ghumman, "Mininet as Software Defined Networking Testing Platform," in *International Conference on Communication, Computing & Systems (ICCCS*, 2014.

[14] B. Milic and M. Malek, "NPART - Node Placement Algorithm for Realistic Topologies in Wireless Multihop Network Simulation," in *Int. Conf. on Simulation Tools and Techniques (SIMUTOOLS)*, 2009.

[15] L. Cerda-Alabern, "On the topology characterization of Guifi.net," in *2012 IEEE 8th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, Oct. 2012, pp. 389–396.

[16] L. Maccari and R. Lo Cigno, "A week in the life of three large Wireless Community Networks," *Ad Hoc Networks*, vol. 24, Part B, no. 0, pp. 175 – 190, 2015.

[17] T. Clausen and P. Jacquet, "Optimized link state routing protocol (OLSR)," Tech. Rep., 2003.

[18] D. Johnson, N. Ntlatlapa, and C. Aichele, "Simple pragmatic approach to mesh routing using BATMAN," 2008.

[19] R. Birke, E. Leonardi, M. Mellia, A. Bakay, T. Szemethy, C. Kiraly, R. L. Cigno, F. Mathieu, L. Muscariello, S. Niccolini *et al.*, "Architecture of a network-aware P2P-TV application: the NAPA-WINE approach," *Communications Magazine, IEEE*, vol. 49, no. 6, pp. 154–163, 2011.

[20] L. Baldesi, L. Maccari, and R. Lo Cigno, "Improving P2P streaming in Wireless Community Networks," *Computer Networks*, 2015.