Dynamic Link Network Emulation: A Model-based Design

Erick Petersen[1,2], Jorge López[1], Natalia Kushik[2], Claude Poletti[1] and Djamal Zeghlache[2]
[1]*Airbus Defence and Space, Issy-Les-Moulineaux, France*
[2]*Télécom SudParis, Institut Polytechnique de Paris, Palaiseau, France*

Keywords: Model-based Design, Dynamic Link Networks, Emulator, Traffic Generation.

Abstract: This paper presents the design and architecture of a network emulator whose links' parameters (such as delay and bandwidth) vary at different time instances. The emulator can thus be used in order to test and evaluate novel solutions for such networks, before their final deployment. To achieve this goal, different existing technologies are carefully combined to emulate link dynamicity, automatic traffic generation, and overall network device emulation. The emulator takes as an input a formal model of the network to emulate and configures all required software to execute live software instances of the desired network components, in the requested topology. We devote our study to the so-called *dynamic link networks*, with potentially asymmetric links. Since emulating asymmetric dynamic links is far from trivial (even with the existing state-of-the-art tools), we provide a detailed design architecture that allows this. As a case study, a satellite network emulation is presented. Experimental results show the precision of our dynamic assignments and the overall flexibility of the proposed solution.

1 INTRODUCTION

As the demand for interactive services, multimedia and network capabilities grows in modern networks, novel software and/or hardware components should be incorporated (Deng et al., 2019). As a consequence, the evaluation and validation process of these newly developed solutions is critical to determine whether they perform well, are reliable and robust before their final deployment in a real network (Alsmadi et al., 2020). However, thorough testing or qualifying (Shan, 2021) the produced software under a wide variety of network characteristics and conditions is a challenging task (Gandhi et al., 2021).

Currently, many of these tests are done through operational, controlled and small-scale networks (physical testbeds) or alternatively software-based testbeds. Ideally, if available, such tests are performed on the original system in order to replicate the conditions in which a service or protocol will be used at the highest level of fidelity (Horneber and Hergenröder, 2014). Unfortunately, while system modeling is not needed, such testbeds are not always desirable or pertinent due to several reasons (Sun and Chai, 2003). For example, there are difficulties in creating various network topologies, generating different traffic scenarios, and testing the implementations under specific conditions (network load or weather conditions that may affect the radio-physical links in specific network technologies such as wireless or satellite communications).

A very well-known alternative method is the use of network simulators (Khan et al., 2012). Through network simulation, researchers can mimic the basic functions of network devices and study specific network-related issues on a single computer or high-end server. However, the adequacy of simulated systems is always in question due to the model abstraction and simplification. At the same time, not simulation but emulation for the related networks can also be a solution (Lai et al., 2019). Network emulation provides the necessary mechanisms to reproduce the behavior of real networks at low infrastructure costs, compared with physical testbeds while achieving better realism than simulations since it allows the interaction with interfaces, protocol stacks and operating systems. Moreover, there is a possibility to perform continuous testing on the final implementation without having to make any changes in the solution once deploying it in a real network. However, the emulation of dynamic link networks, i.e., networks whose link parameters change, complicates the emulation ar-

Petersen, E., López, J., Kushik, N., Poletti, C. and Zeghlache, D.
Dynamic Link Network Emulation: A Model-based Design.
DOI: 10.5220/0011091100003176
In *Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2022)*, pages 536-543
ISBN: 978-989-758-568-5; ISSN: 2184-4895

chitecture. For example, certain radio-frequency links have different up/down bandwidth capacity (Gandhi et al., 2021), large delays (due to distant transmitters), and the links' capacities may change due to external interference, propagation conditions (weather), traffic variations (due to the shared medium), or others. Therefore, it is extremely important to have methods which allow controlling key parts of the emulation over time, such as the generation of traffic or the modification of the link property values (capacity, delay). These are required in order to build a proper emulation environment of interest which is the main focus of this work.

To cope with such requirements, we herein propose a dynamic link network emulation and traffic generation which combines the functional realism and scalability of virtualization and link emulation to create virtual networks that are fast, customizable and portable. The main contributions of this paper are: **i)** a design and architecture of a dynamic link network emulator that meets dynamic link emulation needs by the effective use of software technologies, such as virtualization (containers and virtual machines) and Linux kernel capabilities; **ii)** a dynamic link network emulator that provides a fast and user friendly workflow, from the installation to the configuration of scenarios by using a formal model of the network; **iii)** a use case illustrating the dynamic link capabilities of our emulator – we provide an experimental evaluation, varying different network parameters configured by demand for effectively emulating satellite communications.

2 RELATED WORK

Several works have been devoted to simulation and emulation of networks, to perform experiments on novel or existing protocols and algorithms. Below, we briefly summarize relevant existing solutions.

Ns-3 (Henderson et al., 2008) is a widely used network simulator. Ns-3 simulates network devices by compiling and linking C++ modules; thus it simulates the behavior of real components in a user-level executable program. However, real world network devices are extremely complex (functionally speaking) or cannot be compiled and linked together with Ns-3 to form a single executable program. Therefore, Ns-3 cannot run real-world network devices but only specific ones developed for it. vEmulab (Stoller et al., 2008) is a network emulator with a minimum degree of virtualization, aiming to provide application transparency and to exploit the hierarchy found in real computer networks. Its architecture uses FreeBSD jail namespaces (Kamp and Watson, 2000) in low-end computers to emulate virtual topologies. Similarly, Mininet (Lantz et al., 2010) enables rapid testbeds by using several virtualization features, such as virtual ethernet pairs and processes in Linux container network namespaces. It emulates hosts, switches and controllers which are simple shell processes that are given their network namespace and links between them. However, both still present some limitations including the lack of support for dynamic features such as link emulation, resource management and traffic generation. EstiNet (Wang et al., 2013) is based on network simulation/emulation integration for different kinds of networks. Unlike previous simulators, EstiNet allows not only observation but also configuration through a GUI. It also supports wireless channel modeling. However, since EstiNet is a commercial solution, it cannot be easily extended. Moreover, its features are limited and depend on the EstiNet developers. Thus, the performance fidelity and the expansion to new features are reduced. OpenNet (Chan et al., 2014) also merges simulation and emulation network capabilities by connecting Mininet and Ns-3. However, it inherits the limitations from Ns-3 and Mininet. Additionally, its main focus is on software-defined wireless local area networks (SDWLAN).

More recently, the introduction of lightweight virtualization technologies (e.g., containers) has led to some few container-based emulation tools (Srisawai and Uthayopas, 2018; Farias et al., 2019). SDN Owl (Srisawai and Uthayopas, 2018) is a network emulation tool to create simple SDN testbeds using few computers with Linux OSs. SDN Owl utilizes Ansible to send a set of scripts to properly configure each virtual component. However, it fails to provide scalability and isolation, since experiments with different types of network topologies or resource allocation are not shown. vSDNEmul (Farias et al., 2019) and ContainerNet (Peuster et al., 2018) are network emulators based on Docker container virtualization allowing autonomous and flexible creation of independent network elements, resulting in more realistic emulations. However, these emulators can only create SDN networks. Furthermore, the network descriptions remain rather informal and do not facilitate the verification of the emulator, in order to guarantee that the emulator properly replicates the desired network. A feature comparison is shown in Table 1.

As can be seen and to the best of our knowledge, we are not aware of any works that meet all the features required to properly emulate dynamic link networks in order to qualify novel engineered solutions. This is the main motivation behind this work, which presents the design and architecture of an emulator

Table 1: Comparison of Software-based Network Testbeds.

Name	Open Source	Language	GUI	Emulation Support	Scalability	Portability	Dynamic Links	Automatic Traffic Generation	Formal Description
Ns-3(Henderson et al., 2008)	✓	C++/Python	x	x	+++	x	✓	x	x
Mininet(Kaur et al., 2014)	✓	Python	✓	✓	+	✓	x	x	x
Containernet(Peuster et al., 2018)	✓	Python	✓	✓	++	✓	x	x	x
OMNet++(Varga, 2001)	x	C++	✓	x	+++	x	✓	x	x
Emulab(Stoller et al., 2008)	✓	C	✓	✓	+	✓	x	x	x
OpenNet(Chan et al., 2014)	✓	C++	✓	x	+++	x	✓	x	x
vSDNEmul(Farias et al., 2019)	✓	Python	x	✓	++	✓	x	x	x
EstiNet(Wang et al., 2013)	x	-	✓	✓	++	✓	✓	-	-
SDN Owl(Srisawai and Uthayopas, 2018)	✓	Python	x	✓	++	✓	x	x	x
NetEM(Hemminger et al., 2005)	✓	-	x	✓	-	x	✓	x	x

that meets all the requirements. In order to do so, a model-based engineering approach is well suited, and thus, utilized.

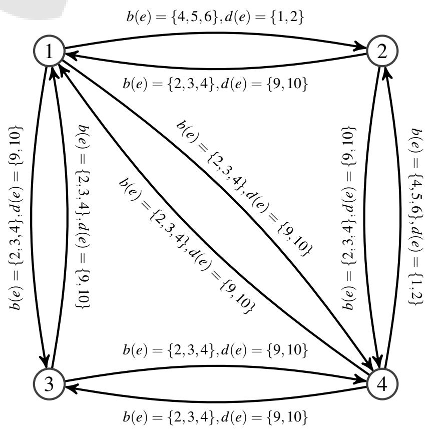
3 BACKGROUND

Dynamic Link Networks. As introduced in (Petersen et al., 2020), a static network is a computer network where each link has a set of parameters that do not change, for example bandwidth (capacity) or delay. Differently from static networks, the parameters of the links may change in dynamic link networks (in the scope of our current work, we assume the network topology does not change); such change can be the consequence of the physical medium (e.g., in wireless / radio frequency networks) or due to logical changes (e.g., rate limiting the capacity of a given link).

Static networks can be modeled as (directed) weighted graphs $(V, E, p_1, \dots, p_k)$, where V is a set of nodes, $E \subseteq V \times V$ is a set of directed edges, and p_i is a link parameter function $p_i : E \to \mathbb{N}$, for $i \in \{1, \dots, k\}$; without loss of generality, we assume that the parameter functions map to non-negative integers (denoted by $\mathbb{N}$) or related values can be encoded with them. Similarly, dynamic link networks can be modeled as such graphs, however, p_i maps an edge to a non-empty set of integer values, i.e., $p_i : E \to 2^{\mathbb{N}} \setminus \emptyset$, where $2^{\mathbb{N}}$ denotes the power-set of $\mathbb{N}$. An example dynamic network is depicted in Fig. 1, and its model $\mathcal{N} = (V, E, p_1(e), p_2(e))$, where:

$$
\begin{aligned}
V &= \{1,2,3,4\} \\
E &= \{(1,2),(2,1),(1,3),(3,1),(1,4), \\
&\quad (4,1),(2,4),(4,2),(3,4),(4,3)\} \\
p_1(e) = b((s,d)) &= \begin{cases} \{4,5,6\}, & \text{if } d = 2 \\ \{2,3,4\}, & \text{otherwise} \end{cases} \\
p_2(e) = d((s,d)) &= \begin{cases} \{1,2\}, & \text{if } d = 2 \\ \{9,10\}, & \text{otherwise} \end{cases}.
\end{aligned}
$$

Semantically, this model represents a dynamic link network in which the link's available bandwidth can vary according to the function b (for *bandwidth*), and the link's delay can vary according to the function d (for *delay*).



$b(e) = \{4,5,6\}, d(e) = \{1,2\}$
1
2
$b(e) = \{2,3,4\}, d(e) = \{9,10\}$
$b(e) = \{2,3,4\}, d(e) = \{9,10\}$
$b(e) = \{2,3,4\}, d(e) = \{9,10\}$
$b(e) = \{2,3,4\}, d(e) = \{9,10\}$
$b(e) = \{2,3,4\}, d(e) = \{9,10\}$
$b(e) = \{2,3,4\}, d(e) = \{9,10\}$
$b(e) = \{4,5,6\}, d(e) = \{1,2\}$
$b(e) = \{2,3,4\}, d(e) = \{9,10\}$
3
4
$b(e) = \{2,3,4\}, d(e) = \{9,10\}$

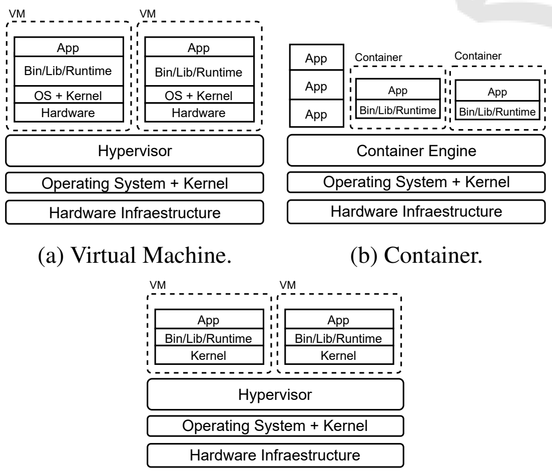
Figure 1: Example dynamic link network.

Virtualization & Software Technologies Involved. Nowadays, virtualization technologies (Giallorenzo et al., 2021) have been used for the design of virtual networks and furthermore, to conduct necessary experiments (emulations). The reason is that virtualization allows creating controlled environments (virtual instances) and reproduce their real underlying network architectures, as well as creating artificial conditions, that can be hard to reproduce in real life. Below we present the virtualization technologies, utilized in our work, namely Virtual Machines, Containers, and Unikernels.

Virtual Machines (VMs) are created and managed by a hypervisor (Wulf et al., 2021), which allows to fully emulate different types of devices (e.g., routers, cellphones, etc.) with their own complete and isolated operating system (OS) or CPU architecture.

Containers are isolated environments (user-spaces instances) for processes; they are created by utilizing the kernel features of a given operating system (Sultan et al., 2019). In contrast to VMs, containers do not get their own virtualized hardware but use the hardware of the host system. Therefore, not having to emulate hardware and boot a complete operating system enables containers to be more efficient than VMs but dependent on the kernel functions of the host OS.

Unikernels are single-address-space machine images constructed by using library operating systems (Watada et al., 2019). The approach consists of packaging a given application (e.g., a router) with the minimal kernel functions and drivers required to run as a sealed and immutable executable directly on the hardware (or sometimes on a hypervisor).



(a) Virtual Machine. (b) Container.

(c) Unikernel.

Figure 2: Virtualization Architectures: VM (a), Container (b), Unikernel (c).

In Figure 2, we depict the architectural differences between the virtualization technologies. Although, the minimality of unikernels seems to be a promising emulation solution between the flexibility of VMs and the performance of containers, it requires applications to be written specifically for them. Thus, we do not consider this option in our work. At the same time, plenty of emulators as introduced above rely just on one single virtualization technology to emulate homogeneous networks. In contrast, we believe that both virtualization technologies (VMs and containers) can be used simultaneously (hybrid approach), to emulate heterogeneous nodes and thus replicate other types of networks with dynamic capabilities.

4 EMULATOR ARCHITECTURE

Our emulation platform design and architecture is based on well-known state-of-the-art technologies presented above, such as virtualization (VMs and containers) and Linux kernel features (namespaces or cgroups). When combined efficiently, these technologies provide excellent capabilities for the emulation of a diverse set of network topologies alongside the dynamic links and interconnected network devices. The emulation platform architecture, shown in Figure 3, consists of several independent, flexible and configurable components. We describe each of these components in detail in the following paragraphs.

The **Emulator Manager** is the main component and the central processing unit. It has a single instance per physical machine and it is composed of several independent modules in charge of the management, deployment and verification of the emulator components for a given network description (input for the emulation). In addition, it is responsible for providing, within the same physical host, the containers or VMs required for each emulated device as well as their own emulated network specifications.

The **Input/Output Processing Module** fulfills several tasks. First, since our emulation platform relies on state-of-the-art virtualization (or container-based) solutions, it is in charge of creating and maintaining a network model that later is used by other modules to implement the necessary infrastructure elements for each emulation. To achieve this, we utilize a formal network description (specification) in terms of first order logic formulas verified throughout the emulation by a satisfiability modulo theories solver. Indeed, the network topology can be verified using model checking strategies before its actual implementation, as well as at run-time, to assure that certain properties of interest hold for the static network instances (Petersen et al., 2020). The module is also in charge of parsing and verifying the file to generate

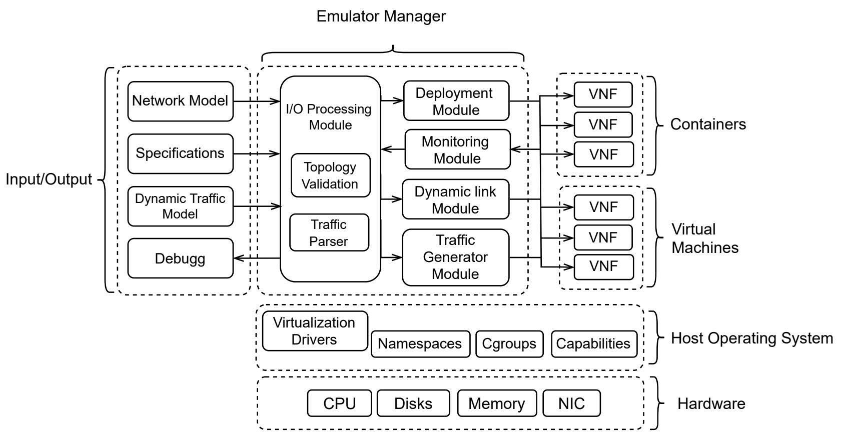


Figure 3: Emulation Platform Architecture.

dynamic traffic scenarios between the components of an emulated network as well as the debugging output of the platform. An example of a network description is given in Listing 1 (for the network in Figure 1).

```
(declare-datatypes()
    ((Edge (mk-edge (src Int) (dst Int)))))
(declare-fun bandwidth (Edge) Int)
(declare-fun delay (Edge) Int)
;; Node storage omitted on purpose
;; to reduce the space, see edge storage
(declare-const edges (Array Int Edge))
(declare-const edges_size Int)
(assert (= (store edges 1 (mk-edge 1 2)) edges))
;; Edge storage omitted on purpose
;; to reduce the space, see first and last edge
(assert (= (store edges 10 (mk-edge 4 2)) edges))
(assert (= edges_size 10))
(assert (forall ((x Int)) (=>
      (and (> x 0) (<= x edges_size))
      (and
        (=> (= (dst (select edges x)) 2)
        ;; ite not used on purpose
            (and
                (>= (bandwidth (select edges x)) 4)
                (<= (bandwidth (select edges x)) 6)
                (>= (delay (select edges x)) 1)
                (<= (delay (select edges x)) 2)
            )
        )
        (=> (not (= (dst (select edges x)) 2))
            (and
                (>= (bandwidth (select edges x)) 2)
                (<= (bandwidth (select edges x)) 4)
                (>= (delay (select edges x)) 9)
                (<= (delay (select edges x)) 10)
            )
      ) ) ) ) ) ;; closing parentheses
```

Listing 1: Example of a Network description (SMT-LIB).

The **Deployment Module** is in charge of converting the previously generated network model into running instances of emulated network devices. In order to achieve this, the module makes use of the `docker` engine for the management and support of containers and `libvirt` for different virtualization technologies such as `KVM`, `VMware`, `LXC`, and `virtualbox`. At the first step, it takes the input specification and creates the required nodes with their corresponding images and properties. Each emulated node is deployed by means of a VM or a container attached to its own namespace and acts according to the software or service running inside of it (as requested by the input specification). For example, if it is desired to run a virtual switch as a Docker container, the Deployment Module creates the proper container and executes the corresponding Virtual Network Function (VNF) via a docker image (e.g., Open vSwitch). Therefore, each node has an independent view of the system resources such as process IDs, user names, file systems and network interfaces while still running on the same hardware. It can also hold several individual (virtual) network interfaces, along with its associated data, including ARP caches, routing tables and independent TCP/IP stack functions.

This gives great flexibility and capabilities to the emulator, it can execute any real software as in the real physical systems. At the last step, the module creates the links between the nodes to complete the emulation topology. The links are emulated with Linux virtual networking devices; `TUN/TAP` devices are used to provide packet reception and transmission for user space processes (applications or services) running inside each node. They can be seen as simple Point-to-Point or Ethernet devices, which, instead of receiving (and transmitting, correspondingly) packets from a physical medium, read (and write, correspondingly) them from a user space process. `veth` (virtual Ethernet) devices are used for combining the network facilities of the Linux kernel to connect different virtual networking components together. `veth` are built as pairs of connected virtual Ethernet interfaces and can be thought of as a virtual "patch" cable. Thus, packets transmitted on one device in the pair are immediately received on the other device and when either device is down the link state of the pair is down too.

```
{ "name": "test_traffic_scenario",
 "initialTime": 0,
 "timeUnitInSec": 1e-3,
 "bandwidthUnitInBits": 1e3,
    "flowSequence": [{
          "time": 4000,
          "flows": [{
                "flow_id": 1,
                "requiredBandwidth": 2,
                "protocol": "tcp",
                "source_ip": "192.168.1.2",
                "source_port": 4000,
                "destination_ip": "192.168.1.3",
                "destination_port": 4000
           }]
           },
                {
                "time": 10000,
                "flows": [{
                "flow_id": 1,
                "requiredBandwidth": 4,
                "protocol": "tcp",
                "source_ip": "192.168.1.2",
                "source_port": 8000,
                "destination_ip": "192.168.1.3",
                "destination_port": 8000
                },{
                "flow_id": 2,
                "requiredBandwidth": 3,
                "protocol": "tcp",
                "source_ip": "192.168.1.2",
                "source_port": 8080,
                "destination_ip": "192.168.1.3",
                "destination_port": 8080
                }]
       }]
}
```

Listing 2: Example of a Dynamic Traffic Description.

The **Dynamic Link Module** is in charge of establishing and modifying the dynamic properties of the links (between the nodes) during the emulation's execution time. An asymmetric link between two nodes, as shown in Figure 4, is emulated by a set of nesting queues; in the simplest case - two queues.

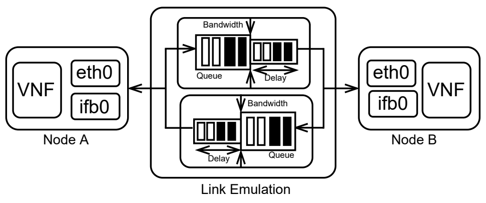


Figure 4: Asymmetric Link emulation model.

At the first step, packets are queued or dropped depending on the size of the first queue. This queue is drained at a rate corresponding to the link's bandwidth. Once outside, packets are staged in a delay line for a specific time (propagation delay of the link) in the second queue and then finally injected into the network stack. This module uses the Linux Advanced traffic control `tc`, to control and set these properties by using filtering rules (classes) to map data (at the data link or the network layer) to queuing disciplines (`qdisc`) in an egress network interface. Note that since `tc` can be used only on egress, Intermediate Functional Block devices (IFB) are created to allow queuing disciplines on the incoming traffic and thus use the same technique.

The **Traffic Generation Module** is in charge of converting the dynamic description of the traffic (see an example of it in Listing 2), into a timed sequence of network packets. This sequence is then introduced into the deployed nodes during the emulation. For the generation of network packets, the module uses `nmap` at each node, particularly `nping`, allowing to generate traffic with headers from different protocols. This is achieved by using `virsh` commands utilizing `libvirt` for VMs or by passing `execute` commands through the Docker daemon (for containers). It is important to keep in mind that `nping` can be replaced for any other software to generate traffic. Additionally, multiple instances of the same or different traffic generators can be executed inside each emulated node.

Finally, the **Monitoring Module** retrieves and collects information from the nodes and their links. This information can be used for example, to verify that the emulation process is executed correctly. Additionally, for our case study, this information is useful to change the bandwidth on demand (in fact this is known as demand assigned multiple access or *DAMA* which further serves as a case study).

5 EVALUATION

Use Case – Satellite Communications. It is certainly difficult to assess novel software systems in satellite networks, mainly due to the complex network dynamics, models and mechanisms that are involved in such networks. Particularly, the difficulties are related to the asymmetric bandwidth capacities of satellite links (due to the use of different radio-frequencies and modulations) and the highly dynamic behaviour of the mechanisms to make an efficient use of the scarce and costly transmission resources. Usually, these mechanisms tend to dynamically allocate bandwidth depending on internal factors but, more importantly external factors (e.g., traffic crossing the network) can also be crucial.

We focus on the DAMA mechanism for dynamic link bandwidth assignation. As its name suggests, the idea is to assign physical resources to nodes which need to transmit traffic, according to their demands. The physical resources are translated to bandwidth. In real settings, in order to assign a given bandwidth, the DAMA module may assign certain time slots and frequencies to a particular node (station). We abstract from this physical level and consider that the bandwidth to physical resource assignation is done by a proper translation module. Therefore, we are interested in the logical view of the algorithm, i.e., only the bandwidth assignations and no deeper. It is important to note that in DAMA, nodes have minimal and maximal bandwidth capacities assigned. Likewise, the satellite transponder has a maximal bandwidth capacity that it can handle.

We assume that the behaviour of user traffic is not predictable in advance, i.e., we do not deal with a seasonal behavior of the network traffic. Thus, dynamic bandwidth assignations of the emulation system are required. That is, the passing traffic modifies the configuration of the emulation and consequently the resulting modifications will have an effect on the traffic going through the emulator. The goal of the developed emulator is to reproduce the conditions described above.

Experimental Settings & Results. For the link's bandwidth assignation we configured it by demand, that is the link's bandwidth gets assigned to whatever the node requires up to a maximal allocation (as in satellite communications). The results are presented in Figure 5; as can be seen, the measured bandwidth (with the `bmw-ng` utility) varies considerably w.r.t. the demand.

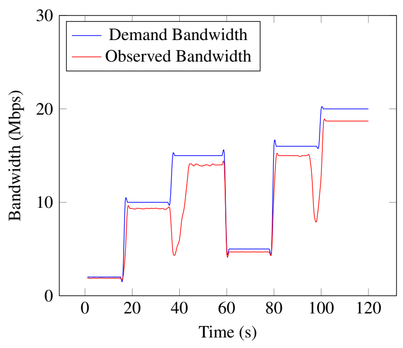


Figure 5: Varying emulated bandwidth by demand.

For the link's delay emulation, we have varied the desired delay between 5ms and 100ms. The assignation was made randomly by setting the delay queue with a *normal* distribution (using the `tc` utility). The results are presented in Figure 6; as can be seen, the delays have been measured (with `nping`) and the histogram clearly shows a normal distribution, as expected. It is important to note that the measured delay can be higher than 100ms since the processing time of the equipment plays a role in the delay, as in the real system.

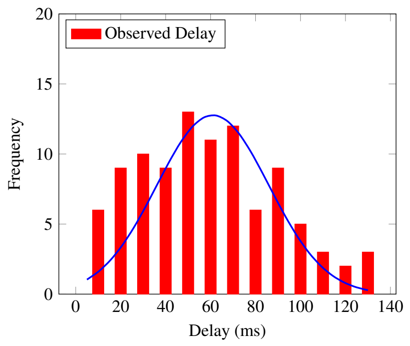


Figure 6: Emulated delay histogram.

In this section, we have showcased how both link parameters of interest (bandwidth and delay) can be dynamically (and automatically) assigned in our emulator; it can properly emulate a complex DAMA scheme for satellite communications. Moreover, it is capable of producing distinct traffic configurations to thoroughly test novel engineered solutions. Additionally, the emulator's design allows to easily interact with real life software components.

6 CONCLUSION

In this paper, we have showcased the design and architecture for a dynamic link network emulator. The emulator is flexible and can emulate any existing software; additionally, it can dynamically change the link parameter values. Although the emulator is very flexible, we note that the emulator cannot handle mobile nodes, i.e., networks whose topology may change (such as mobile ad-hoc networks).

As for future work, we plan to incorporate more features into the architecture so that it becomes more controllable and realistic, for example, we consider incorporating link state scenarios to qualify the solutions under different conditions (degraded links, weather conditions, etc.). Additionally, we envision incorporating mobility patterns in links to emulate mobile networks. Finally, we plan to investigate various model-based simulation strategies for dynamic link networks, to efficiently (faster than the emulation) obtain static network snapshots with certain desired properties; this should allow the emulator to provide interesting states/configurations to assess novel network management solutions.

REFERENCES

Alsmadi, I., Zarrad, A., and Yassine, A. (2020). Mutation testing to validate networks protocols. In *2020 IEEE International Systems Conference (SysCon)*, pages 1–8, Montreal, QC, Canada. IEEE.

Chan, M.-C., Chen, C., Huang, J.-X., Kuo, T., Yen, L.-H., and Tseng, C.-C. (2014). Opennet: A simulator for software-defined wireless local area network. In *2014 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 3332–3336. IEEE.

Deng, B., Jiang, C., Yao, H., Guo, S., and Zhao, S. (2019). The next generation heterogeneous satellite communication networks: Integration of resource management and deep reinforcement learning. *IEEE Wireless Communications*, 27(2):105–111.

Farias, F. N., Junior, A. d. O., da Costa, L. B., Pinheiro, B. A., and Abelém, A. J. (2019). vsdnemul: A software-defined network emulator based on container virtualization. *arXiv preprint arXiv:1908.10980*.

Gandhi, S., Singh, R. K., et al. (2021). Design and development of dynamic satellite link emulator with experimental validation. In *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pages 1–6. IEEE.

Giallorenzo, S., Mauro, J., Poulsen, M. G., and Siroky, F. (2021). Virtualization costs: benchmarking containers and virtual machines against bare-metal. *SN Computer Science*, 2(5):1–20.

Hemminger, S. et al. (2005). Network emulation with netem. In *Linux conf au*, pages 18–23.

Henderson, T. R., Lacage, M., Riley, G. F., Dowell, C., and Kopena, J. (2008). Network simulations with the ns-3 simulator. *SIGCOMM demonstration*, 14(14):527.

Horneber, J. and Hergenröder, A. (2014). A survey on testbeds and experimentation environments for wireless sensor networks. *IEEE Communications Surveys Tutorials*, 16(4):1820–1838.

Kamp, P.-H. and Watson, R. N. (2000). Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, volume 43, page 116.

Kaur, K., Singh, J., and Ghumman, N. S. (2014). Mininet as software defined networking testing platform. In *International Conference on Communication, Computing & Systems (ICCCS)*, pages 139–42.

Khan, A. R., Bilal, S. M., and Othman, M. (2012). A performance comparison of open source network simulators for wireless networks. In *2012 IEEE International Conference on Control System, Computing and Engineering*, pages 34–38, Penang, Malaysia. IEEE.

Lai, J., Tian, J., Jiang, D., Sun, J., and Zhang, K. (2019). Network emulation as a service (neaas): Towards a cloud-based network emulation platform. In Song, H. and Jiang, D., editors, *Simulation Tools and Techniques*, pages 508–517, Cham. Springer International Publishing.

Lantz, B., Heller, B., and McKeown, N. (2010). A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 1–6. ACM.

Petersen, E., López, J., Kushik, N., Poletti, C., and Zeghlache, D. (2020). On using smt-solvers for modeling and verifying dynamic network emulators: (work in progress). In *19th IEEE International Symposium on Network Computing and Applications, NCA 2020, Cambridge, MA, USA, November 24-27, 2020*, pages 1–3. IEEE.

Peuster, M., Kampmeyer, J., and Karl, H. (2018). Containernet 2.0: A rapid prototyping platform for hybrid service function chains. In *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, pages 335–337. IEEE.

Shan, Q. (2021). Testing methods of computer software. In *2020 International Conference on Data Processing Techniques and Applications for Cyber-Physical Systems*, pages 231–237. Springer.

Srisawai, S. and Uthayopas, P. (2018). Rapid building of software-based sdn testbed using sdn owl. In *2018 22nd International Computer Science and Engineering Conference (ICSEC)*, pages 1–4. IEEE.

Stoller, M. H. R. R. L., Duerig, J., Guruprasad, S., Stack, T., Webb, K., and Lepreau, J. (2008). Large-scale virtualization in the emulab network testbed. In *USENIX Annual Technical Conference, Boston, MA*.

Sultan, S., Ahmad, I., and Dimitriou, T. (2019). Container security: Issues, challenges, and the road ahead. *IEEE Access*, 7:52976–52996.

Sun, Z. and Chai, W. K. (2003). Satellite emulator for ip networking based on linux. In *21st International Communications Satellite Systems Conference and Exhibit*, page 2393.

Varga, A. (2001). Discrete event simulation system. In *Proc. of the European Simulation Multiconference (ESM'2001)*, pages 1–7.

Wang, S.-Y., Chou, C.-L., and Yang, C.-M. (2013). Estinet openflow network simulator and emulator. *IEEE Communications Magazine*, 51(9):110–117.

Watada, J., Roy, A., Kadikar, R., Pham, H., and Xu, B. (2019). Emerging trends, techniques and open issues of containerization: A review. *IEEE Access*, 7:152443–152472.

Wulf, C., Willig, M., and Göhringer, D. (2021). A survey on hypervisor-based virtualization of embedded reconfigurable systems. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pages 249–256.