SIMITAR: realistic and autoconfigurable traffic generation

1st Anderson dos Santos Paschoalon
dept. name of organization (of Aff.)
State University of Campinas
Campinas, Brazil
anderson.paschoalon@gmail.com

2nd Christian Esteve Rothenberg
dept. name of organization (of Aff.)
State University of Campinas
Campinas, Brazil
chesteve@dca.fee.unicamp.br

***Abstract*—It is a well known fact that the type of traffic on network measurements matters. A realistic Ethernet traffic have a different impact over a device compared to constant traffic (such as generated by Iperf), even with the same avarage bandwidth. A burstier traffic can cause more buffers overflows, increace losses and decrease on the measurement acuracy. The number of flows of a traffic may have an impact of flow-oriented nodes, such as SDN switches and controllers. Therefore, in an scenario where software defined networks are going to play an essential role in the future of the networks, a depper validation of new technologies in order to guarantee the SLAs is crucial. In this scenario we come with an anternative for realist traffic generation. Most of open-source realistic traffic generator tools have the modelling layer coupled to the traffic generator API, wich make it difficult to be update to newer lybraries, or require user programming and manual configuration. We propose a sollution called SIMITAR: SnIffing, ModellIng and TrAffic geneRation, wich separetes the processes of data collection, modelling and traffic generation, is flow-oriented, and is autoconfigurable. It creates and use as imputs compact traces descriptos, XML files that model a network trace. Currently we are able to replicate with acuracy some applications traffics and the flow-level properties from the traffic.**

***Index Terms*—Sniffing, traffic modelling, BIC, AIC, inter-packet times, Wavelet scalling, traffic generator, Burstiness, pcap file, linear regression, Iperf**

I. Introduction

It is already a well-known fact that the type of traffic used on tests impacts in many results, even with the same avarage bandwidth. Tests with constant bit rate traffic generator tools are not enough for a complete validation of new technologies. Realistic Ethernet traffic provides a different and more variable load characteristics on routers [1], even with the same avarage bandwidth. As reason we can for this behavior we have traffic burstiness and many different packet sizes[1]. A realistic burstier traffic can cause more buffer overflows, increase packet losses, cause a network performance degeneration [2] [3] [4], and interfere on the measurement acuracy [5] [6] compared to a constrant traffic. Also the traffic burstiness interferes on the measurement accuracy.

Another key question is how applications will deal with packets. It is a well-known fact that applications have a huge performance degradation processing small packets [7]. A realistic synthetic traffic must not have a single packet-size but must use a distribution [8]. Realistic workload generators are also essential security research [9]. It includes studies of intrusion, anomaly detection, and malicious workloads [9], over firewall middleboxes and softwares. In this case, not just the traffic burstiness is relevant, but the connection behavior and header contents, such as protocols and port numbers.

Emerging technologies such as SDN and NFV are drastically changing computer networks. But enabling technologies such as virtualization still pose challenges on performance, reliability, and security [10], since hardware solutions have a much more predictable behavior. There is a demand for more reliable methods to ensure the Service Layer Agreements, over different types of loads. We can expect an even larger impact over over virtualized middle-boxes should be even larger compared to dedicated hardware devices, due the virtualization-layer extra overhead. Along with that, on Software Defined Networks we have the flow-based operation [11]. Each new flow arriving on an SDN switch demands an extra communication between it and the controller, wich may create a boobleneck. The SDN switches have a flow-oriented operation, since it check each new packet in an flow-match table. Therefore an performance measurement must use a traffic with the same flow properties of an actual Internet Service Provider. Therefore, there is a demand for the study of the impact of a realistic traffic on this new sort of environment. How virtualized network functions, virtualized middle-boxes and SDN switches and cotrollers will behave if stressed with a realistic traffic load in comparison to a constant rate traffic is an important study subject.

II. Open-source Traffic Generators

The open-source community offers a huge variety of workload generators and benchmarking tools [9] [12] [7] [13]. Most of these tools were built for specific purposes and goals, each of them wiht diferent methods of traffic generation and enabling the controll of different features, such as throughput (bits/bytes per second, packets per seconds), packet-sizes, protocols and header customization, payload customization, inter-packet times, packet train periods, start and sending time, and emulation of applications such as Web server/client

[1]Traffic burstness is the intermitent increase and decreace on data transference. Fuatures such as packet-trains periods and inter-packet times affect traffic burstiness

communication, VoIP, HTTP, FTP, p2p applications, and many more. Traffic generators are commonly classified into four different groups, according to its layer of operation [14]: application-level, flow-level, packet-level and multi-level traffic generators.

Application-level traffic generators try to emulate the behavior of network applications, simulating real workloads stochastically and/or responsively [2]. As examples we have Surge [15] able to emulate the behavior of web clients and services.

Flow-level traffic generators are able to configure and reproduce features of flows [14] [16], such as number of flows and flow duration, start times distributions, and temporal (diurnal) traffic volumes [14]. Harpoon [17] is able to extract these parameters from Cisco NetFlow data, collected from live routers.

Packet-level traffic generators is the most common traffic generators class available. They are able to craft packets and inject into the network, and controll features like inter departure times (IDT), packet size(PS), throughput and packets per second [12]. [12] classify them as replay engines, maximum throughput generators, and model based generators. On the other hand, offer the control for just a single flow each time, and if available will require scripting. Traffic Replay engines, such TCPreplay [18], are able to read packet capture files (*pcap* files), and inject copies on a network interface. Model-based generators, such D-ITG [9], TG [19], may have PS and IDT configured by the user. They can configure IDT either by constant values (defining the packet rate or bandwidth) or by stochastic models. Maximum throughput engines like Ostinato and Seagull are tools that the main goal is to perform end-to-end stress testing. Are included in this category generators that are able to craft packets from the link layer, like Brute, KUTE, and pktgen.

Multi-level traffic generators have the porpose of take into account existing interaction among each layer of the network stack, to create network traffic as close as possible from reality. The most relevant open-source tool of this class is Swing [20]. Swing take into account many points of each layer, using an approach based on randomness and responsiveness [20]. That means, it uses packet-sizes and inter packet times stochastic models, but also take into account flow and application layer behavior. Swing is able to model things like [20]:

- User behavior: Number of request and responses, time between requests and responses;
- Request and Responses: number of connections, time between start of connections;
- Connection: Number of request and responses per connection, applications modeling (HTTP, P2P, SMTP), transport protocols TCP/UDP based on the application, response sizes, request sizes, user think between exchanges on a connection;
- Packet: packet size (Maximum transmission unit), IP addresses, (MTU), bit rate and packet arrival distribution,
- Link: link Latency Delay, loss rates.

Swing take as input collected *pcap* files. Botta at al. [14] argues that due its complexity, they are rarely used to conduct experimental researchers.

Application level traffic generators may be realistic from a operation system perspective, but are not able to represent real complex scenarios. Replay engines such as TCPreplay and TCPivo have some limitation. First, they usually do not replicate the same inter-packet times form the original trace, just creates a constant bandwidth traffic. They also requere storage of *pcap* files, wich may become huge for long-term and high-speed traffic capture traces. Also, obtaining good traffic traces sometimes is hard, due privacy issues and limited sources. Many tools support a larger set of protocols and high-performance, such Seagull and Ostinato. Others are also able to control inter-packet times and packet sizes using stochastic models, like D-ITG [9] and MoonGen [21]. But all of them just offer a complet framework to be congfigured, but the customization of the traffic is all up to user to do. So, there is no simple way for the user to create an synthetic realistic traffic scenario.

We also have available a variety of APIs that enable creation of traffic and custom packets, wich include low-level APIs, such as Linux Socket API, Libpcap, Libtins, DPDK, Pcapplusplus, libcrafter, impacket, scapy and many others. These APIs enable a finer controll and customization over each packet, and are used to implement traffic generators. For example, Ostinato and TCPreplay uses Libpcap, and MoonGen uses DPDK. Many of the listed traffic generators provides their own high-level API, such as Ostinato Python API, D-ITG C API, and MoonGen LUA API.

Indeed, we have a large variety open-source tools available for custom traffic generation. But reproducing a realistic traffic scenario is a hard question. Selecting the right framework, a good traffic model, and the right configuration, is by itself a complex research project [5] [22]. But usually this is not the goal of the project, so the researcher or the them have scarce time available to work over it. Since it is usually is not the main goal of the project, but just a mean many times a simplistic and irrealistic sollution is opt because of the limited resources such as time and manpower. Reproducig a realistic traffic usung these tools is a manual process and demands implementation of scripts or programs leveraging human (and scarce) expertise on network traffic characteristics and experimental evaluation. Since it usually is not the goal of the research, but a mean.

For this gap tools like Swing and Harpoon uses capture traces to set intern parameters, enabling an easier configuration and provide a realistc traffic generaion. Also, Swing uses complex multi-levels models which are able to provide a high degree of realism [20]. But they have their issues as well. Harpoon does not configure parameters at packet level [1] and is not supported by newer Linux kernels, what may be a huge problem with setup and configuration. Swing [20] aims

[2]Responsiveness refers to the property of capacity of response over time of the workload tool. That means, the behavior of the output may change according to what packets have arrived in the network interface

to generate realistic traffic, but focous on background traffic, and high throughputs it is not a goal off the application [20] [5]. Due the fact that its traffic generation engine is coupled to its modeling framework, you can't opt to use a newer/faster packet generation library. The only way of replacing the traffic engine is changing and recompiling the original code. This is clearly a hard task [5], and an error prone activity. Again, we fall in the same problem a complex task that usually is not the goal of the project.

Another issue is the large variety of tools, and different methods of configuration and limitations. To create a custom traffic, a user must read large manuals, and custom-design scripts. One of our bigger proposals is create a tool able to automatically do this processes. So The user may design his custom traffic by creating his own Compact Trace Descriptor, and create a traffic using many different tools, like Ostinato, D-ITG, Iperf, but not caring about how to proper configure each of them.

III. Architecture and Modelling

In this section we are going to present the SIMITAR architecture and operation. SIMITAR is composed of four components, as presented in the figure 1: the *Sniffer*, a *SQLite database*, the *TraceAnalyzer*, and the *FlowGenerator*. We explain each part down below.

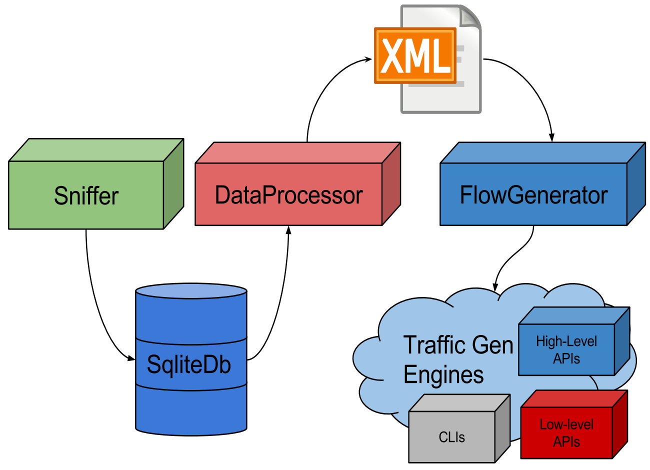


Fig. 1: Architecture of SIMITAR

A. Sniffer

This component collects network traffic data and classifies it into flows, storing stores in an SQLite database. It defines each flow by the same criteria used by SDN switches [11], through header fields matches. It uses:

- Link Protocol
- Network Protocol
- Network Source and Destination Address
- Transport Protocol
- Transport Source and Destination Port

It has a data structure we developed called *OrderedSet*. A set is a list of elements with no repetition but does not keep track

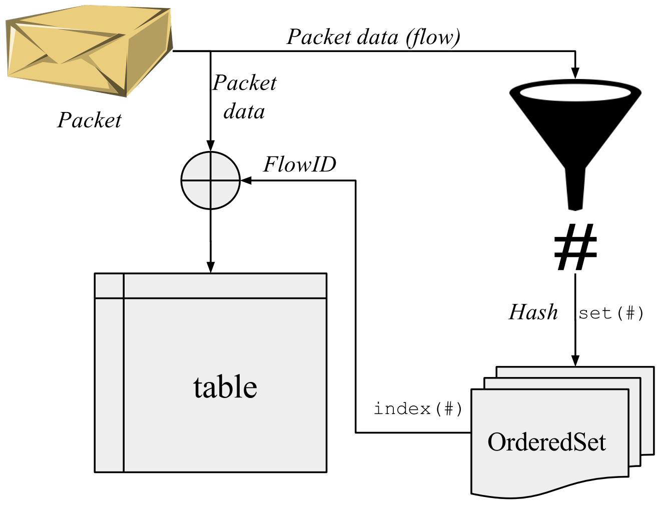
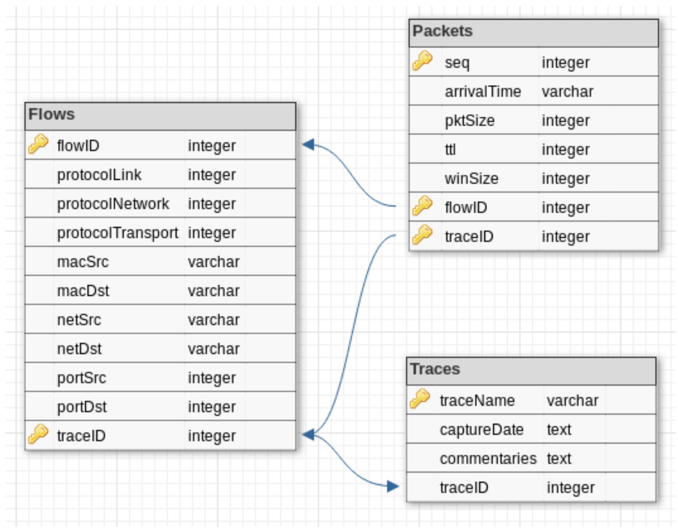


Fig. 2: SIMITAR's sniffer hash-based flow classification

of the insertion order. Our OrderedSet does. Also, it makes use of a 64 bits hash function of the family FNV[3]. The listed header fields are inputs for a hash function, and its value is set on the ordered set which returns its order (index on the *OrderedSet*). The index value is chosen as a packet *flowID*.

B. SQLite database



Flows	
flowID	integer
protocolLink	integer
protocolNetwork	integer
protocolTransport	integer
macSrc	varchar
macDst	varchar
netSrc	varchar
netDst	varchar
portSrc	integer
portDst	integer
traceID	integer

Packets	
seq	integer
arrivalTime	varchar
pktSize	integer
ttl	integer
winSize	integer
flowID	integer
traceID	integer

Traces	
traceName	varchar
captureDate	text
commentaries	text
traceID	integer

Fig. 3: SIMITAR's SQLite database relational model

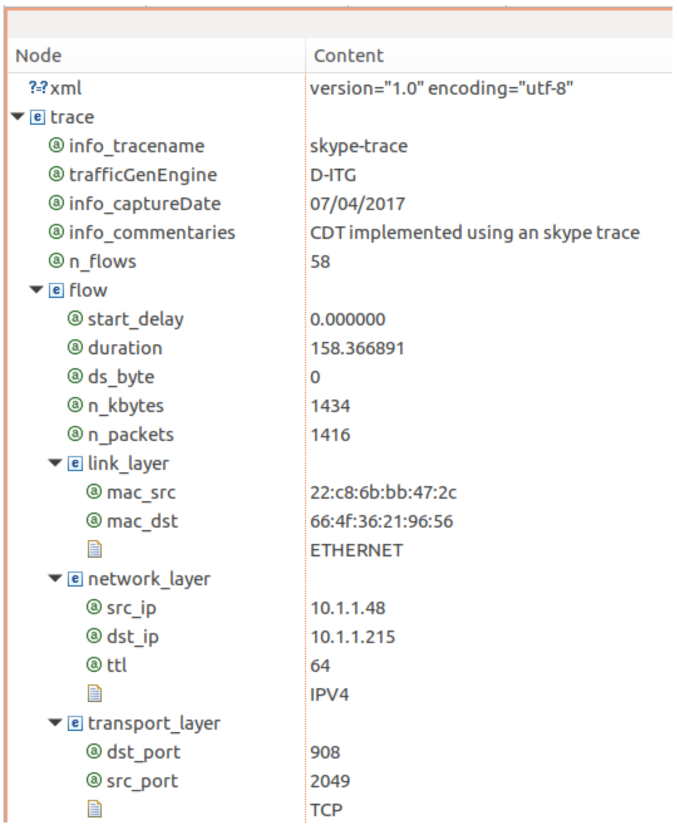
The database stores the collected raw data from the traces for further analysis. The *Sniffer* and the *TraceAnalyzer* components uses the database. We choose an SQLite database, because according to its specifications it is simple and well-suitable for an amount of data smaller than terabytes [4]. In the figure 3 we show the relational model of our database, which stores a set of features extracted from packets, along with the *flowID* calculated by the sniffer component.

[3]The collision probability of a good 64 bits hash function in a table with 10000 items is about of $2.71e - 12$.

[4]https://www.sqlite.org/whentouse.html

C. Trace Analyzer

This module is the core of our project. It creates a trace model via the analysis of the collected data. We define here a Compact Trace Descriptor (CTD) as a human and machine-readable file, which describes a traffic trace through a set of flows, each of them represented by a set of parameters, such as header information and analytical models. The Trace Analyze has the task to learn these features from raw traces data (stored in the SQLite database) and generate an XML file. In the figure 4 we show a directory diagram of a CDT file. It has many of many flow fields, and each one contains each parameter estimated. Now we will describe how we construct it.



Node	Content
?-? xml	version="1.0" encoding="utf-8"
trace	
info_tracename	skype-trace
trafficGenEngine	D-ITG
info_captureDate	07/04/2017
info_commentaries	CDT implemented using an skype trace
n_flows	58
flow	
start_delay	0.000000
duration	158.366891
ds_byte	0
n_kbytes	1434
n_packets	1416
link_layer	
mac_src	22:c8:6b:bb:47:2c
mac_dst	66:4f:36:21:96:56
	ETHERNET
network_layer	
src_ip	10.1.1.48
dst_ip	10.1.1.215
ttl	64
	IPV4
transport_layer	
dst_port	908
src_port	2049
	TCP

Fig. 4: Directory diagram of the schema of a Compact Trace Descriptor (CDT) file.

1) Flow features: Some unique per-flow features are directly measured from the data. They are:

- Flow-level properties like duration of flow, start delay, number of packets per flow, number of KBytes per flow;
- Header fields, like protocols, QoS fields, ports, and addresses.

Each one of these parameters is unique per flow. Other features like PSD (packet size distribution) e IPT (Inter-packet time), have a more complex behavior. To represent these characteristics, we will use sets of stochastic-based models.

2) Inter Packet Times: To represent inter-packet times, we adopt a simplified version of the Harpoon's traffic model. A deep explanation of the original model can be found at [17] and [1]. Here, we will explain our simplified version, which is illustrated at figure **??**.

Harpoon uses a definition of each level, based on the measurement of SYN and ACK TCP flags. It uses TCP flags (SYN) to classify packets in different levels, named their file, session, and user level. We choose to estimate these values, based on inter-packet times only. The distinction is made based on the time delay between packets.

In our algorithm simplified version, we define three different layers of data transference to model and control: file, session, and flow. For SIMITAR, a file is just a sequence of consecutive packets transmitted continuously, without large interruption of traffic. It can be, for example, packets sent downloading a file, packets from a UDP connection or a single ICMP echo packet. The session-layer refers to a sequence of multiple files transmitted between a source and a destination, belonging to the same flow. The flow level refers to the conjunct of flows, as classified by the Sniffer. Now, we explain SIMITAR operation on each layer.

In the **flow layer** the *TraceAnalyzer* loads the flow arrival times from the database and calculates the inter-packet times within the flow context.

At the **session layer**, we use a deterministic approach for evaluating file transference time and times between files: ON/OFF times sequence of packet trains. We choose a deterministic model because in this way we can express diurnal behavior [1]. We develop an algorithm called *calcOnOff* responsible for estimating these times. It also determines the number of packets and bytes transferred by each file. Since the ON times will serve as input for actual traffic generators, we defined a minimum acceptable time for on periods equals to 100 ms. ON times can be arbitrary smalls, and they could be incompatible with acceptable ON periods for traffic generators. Also in the case of just one packet, the ON time would be zero. So setting a minimum acceptable time to solve these issues. The OFF times, on the other hand, are defined by the constant `session_cut_time` [5]. If the time between two packets of the same flow is larger than `session_cut_time`, we consider them belonging to a different file, so this time is a session OFF time. In this case, we use the same value of the constant *Request and Response timeout* of Swing [20] for the `session_cut_time`: 30 seconds. The control of ON/OFF periods in the traffic generation is made by the *Flow Generator* component [6].

In the **file layer**, we model the inter-packet times at the file level. To estimate inter-packet times within files, we select all inter-packet times smaller than `session_cut_time`[7]. All files within the same flow are considered to follow the same model. We delegate the control of the inter-packet times to the underlying workload engine tool. We ordered them, from the best to the worst. Currently, we are using eight different

[5]In the code it is called `DataProcessor::m_session_cut_time`
[6]This control is made by the class `NetworkFlow`
[7]In the code it is called`DataProcessor::m_session_cut_time`

stochastic functions parameterizations. They are Weibull(linear regression), Normal(mean/standard deviation calculation), Exponential(mean and linear regression estimation), Pareto(linear regression and maximum likelihood), Cauchy(linear regression) and Constant(mean calculation). From those, Weibull, Pareto, and Cauchy are heavy-tailed functions, and therefore self-similar processes. But if the flow has less than 30 packets, just the constant model is evaluated. It is because numerical methods gave poor results if the data sample used is small. We sort these models according to the Akaike Information Criterion (AIC) as default [16] [23]. We will enter into deeper details on this methodology on the chapter **??**. The methodology of selection is presented in the figure **??**, and all constants and modes of operation can be changed by command line options.

3) Packet Sizes: Our approach for the packet size is much simpler. Since the majority of packet size distribution found on real measurements are bi-modal [8] [16] [24], we first sort all packet sizes of flow in two modes. We define a packet size mode cut value of 750 bytes, same value adopted by [24].

We know how much packets each mode has, and then we fit a model for it. We use three stochastic models: constant, exponential and normal. Since self-similarity does not make sense for packet-sizes, we prefer to use just the simpler models. When there is no packet for a model, we set a flag NO_MODEL, and when there is just a single packet we just use the constant model. Then calculate the BIC and AIC for each, but we decide to set the constant model as the first.

As is possible to see in many works [8] [24], since the standard deviation of each mode tends to be small, constant fittings use to give good approximations. Also, it is computationally cheaper for the traffic generated than the other models, since no calculation is a need for each packet sent. Since both AIC and BIC criteria always will select the constant model as the worst, we decide to ignore this.

4) Packet Sizes: Our approach for the packet size is much simpler. Since the majority of packet size distribution found on real measurements are bi-modal [8] [16] [24], we first sort all packet sizes of flow in two modes. We define a packet size mode cut value of 750 bytes, same value adopted by [24].

We know how much packets each mode has, and then we fit a model for it. We use three stochastic models: constant, exponential and normal. Since self-similarity does not make sense for packet-sizes, we prefer to use just the simpler models. When there is no packet for a model, we set a flag NO_MODEL, and when there is just a single packet we just use the constant model. Then calculate the BIC and AIC for each, but we decide to set the constant model as the first.

As is possible to see in many works [8] [24], since the standard deviation of each mode tends to be small, constant fittings use to give good approximations. Also, it is computationally cheaper for the traffic generated than the other models, since no calculation is a need for each packet sent. Since both AIC and BIC criteria always will select the constant model as the worst, we decide to ignore this.

D. Flow Generator

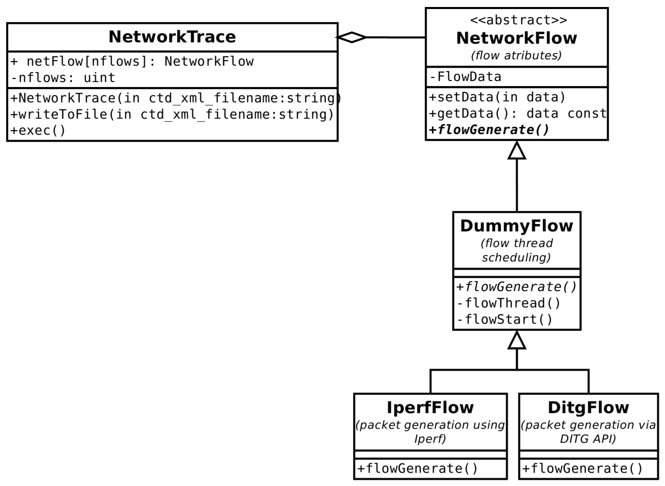


Fig. 5: Class hierarchy of NetworkTrace and NetworkFlow, which enables the abstraction of the traffic generation model of the traffic generation engine.

The Flow Generator handles the data on the *Compact Trace Descriptor* file, and use to retrieve parameters for traffic generation. It crafts and controls each flow in a separated thread. We already implemented this component using Iperf traffic generator and Libtins C++ API, and it must follow the calss hierarcy as presented in the figure 5. This component was designed using the design pattern Factory, to simplify its expansion and support[8].

This component and therefore our traffic generator is a multi-layer workload generator according to the typing introduced on chapter **??** [9]. At the flow-level, SIMITAR controls each flow using the algorithm at figure 6. This algorithm handles our model defined at the figure **??**. This procedure is independent of the underlying packet crafting tool used. It starts a *thread* for each flow in the *Compact Trace Descriptor*, then the thread sleeps for `start_delay` seconds. This is the arrival time of the first flow packet. Passed this time, it then calls the underlying traffic generation tool(defined as command line argument), and pass to it the flowID, file ON time, number of packets and number of bytes to be sent (file size), and network interface. Then it sleeps the next session OFF time.

[8]If the user wants to introduce support for a new traffic generator, he just has to expand the class `DummyFlow`, creating a new derived class (figure 5). On our current implementation we already have implemented `IperfFlow` and `TinsFlow`, and `DitgFlow` is under validation process. This new class has the requirement of not have any new attribute. The support for this new class must be given on the factory class `NetworkFlowFactory`. For closed loop packet-crafters (that means, tools which must establish a connection between the source and the destination), two methods must be implemented: `flowGenerate()` and `server()`. `flowGenerate()` is responsible for sending a *file*, as defined on de figure **??**, and `server()` to receive n files. Open loop packet-crafters, such as we implemented using Libtins (just sent the packets, do not establishes a connection), the server do not need to be implemented.

[9]Since it works both at packet and flow level, but do not work at application level.

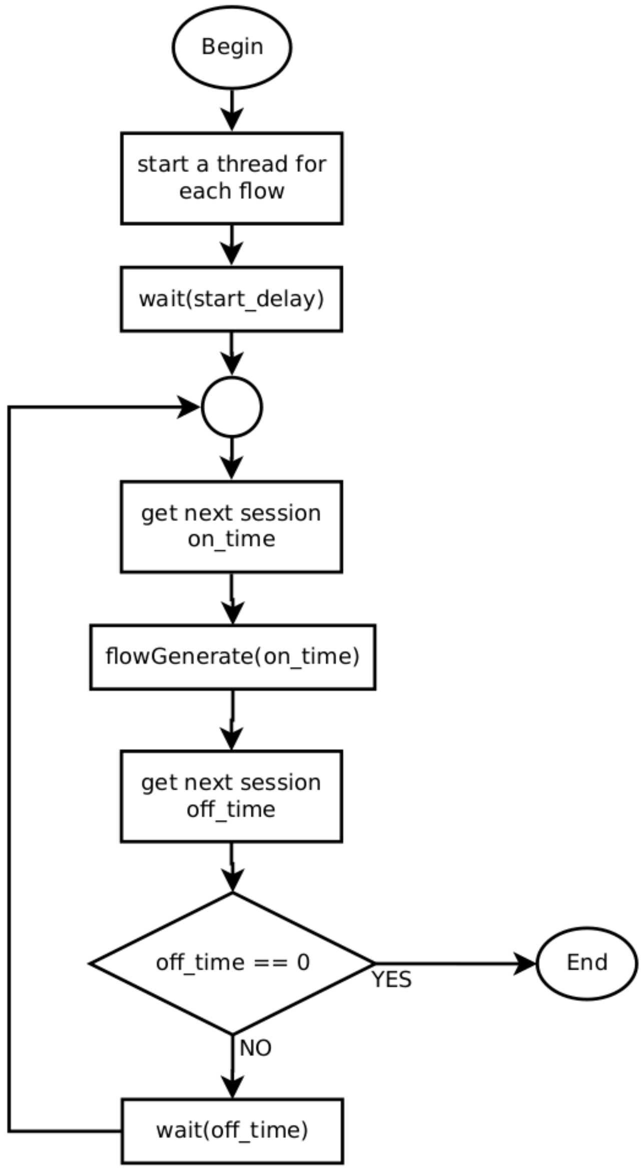


Fig. 6: Simplified-harpoon emission algorithm

When the list of ON/OFF times from the flow is over, the thread ends.

At the packet level, the job of our tool is to configure the underlying engine, to send a *file*, as defined in the figure **??**. This method must use the available parameters, attributes and *getters* to configure and generate a thread-safe traffic using the method `flowGenerate()`. The *file* configuration must follow the in the figure 7.

IV. Validation

A. Methodology

As proof of concepts for our tool, we are going to use Mininet's emulated testbeds. We automate all tests using scripts, so our experiments are are fully reproducible. They are

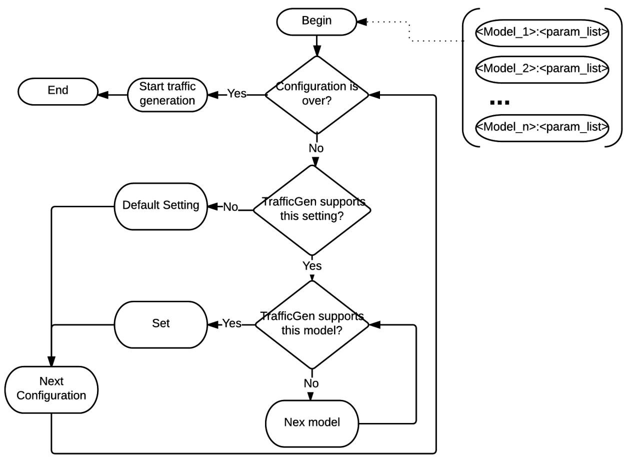


Fig. 7: Traffic engine configuration model

responsible for running all the proposed tests in this chapter and perform the calculations. It includes:

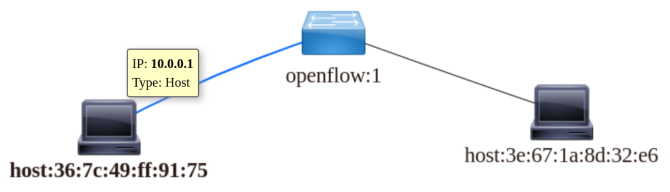
- Build the topology;
- Run the SIMITAR traffic generator;
- Collect the packeckets as *pcap* files, and stract data from it;
- Perform the proposed proof of concept analysis;
- Plot the data.

Each test is organized as a Python packege, responsible for tigger all the applications and procedures. The parameters for each simulation can be configured by a *config.py* file. The files *README* on each packege, provide a tutorial to run correctely each test. With all tools instaled, less then fifteen minutes is enought for run each test again. A complete specification of our scenario we show at the table I, including the hardware specificatins and the software versions.

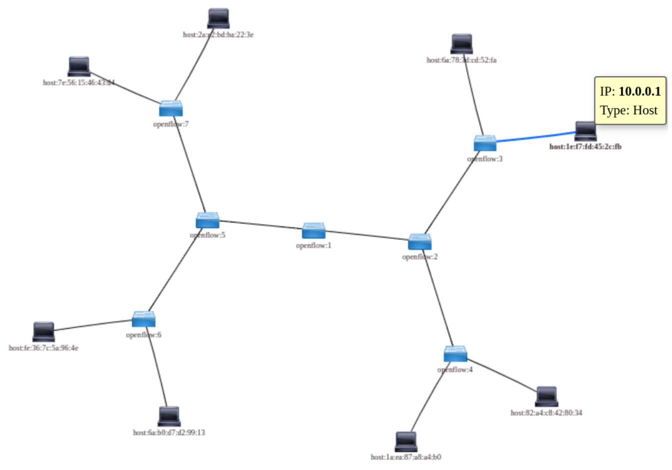
TABLE I: Experiments specification table

Processor	Intel(R) Core(TM) i7-4770, 8 cores, CPU @ 3.40GHz
RAM	15.5 GB
HD	1000 GB
Linux	4.8.0-59-generic
Ubuntu	Ubuntu 16.10 (yakkety)
SIMITAR	v0.4.2 (Eulemur rubriventer)
Mininet	2.3.0d1
Iperf	iperf version 2.0.9 (1 June 2016) pthreads
Libtins	3.4-2
OpenDayLight	0.4.0-Beryllium
Octave	4.0.3
Pyshark	0.3.6.2
Wireshark	2.2.6+g32dac6a-2ubuntu0.16.10
Tcudump	4.9.0
libpcap	1.7.4

For each test, we generate a set of plots to compare the original and synthetic trace. Two of them we use for mere visual comparison: flows per second and bandwidth. To compare the realism quality of the generated traffic, we plot the flows cumulative distribution function (CDF) [17], and the Wavelet multiresolution analysis. On every case, the



(a) Single hop SDN topology



(b) Tree SDN topology

Fig. 8: Mininet SDN topologies. Controller OpenDayLight Beryllium.)

more closer the plots are, the more similar the traffics are according to each perspective. Also, we wrote in the table II a complation of each traffic statistics.

The flow's cumulative distribution measures each new flow identified the trace. It is a measure similarity of the traffic at the flow-level. The wavelet multiresolution analysis is capable of capture traffic scaling characteristics and is a measure of similarity at the packet-level. If the value decreases, a periodicity on that time scale exists. With white-noise features, the traffic will remain constant. If the traffic has self-similar characteristics on a particular time scale, its value will increase linearly.

We use as testbeds: a tree topology (figure 8b), similar to tests performed by Swing [20] [6] [5], and a one-hop connection of two hosts. Both topologies are SDN networks and have OpenDayLight Beryllium as the controller.

For generating the traffic on the host *h1* with IPv4 address 10.0.0.1. The traffic is captured from the host interface with TCPdump in a *pcap* format. We organized the project directory tree as follows[10]: the software is at *SIMITAR*. All validation tests and project prototypes we aved at *Tests* directory. The software documentation is at *Docs* directory.

We use SIMITAR v0.4.2 (Eulemur rubriventer)[11], as tagged at the GitHub repository. SIMITAR already have two functional traffic generator engines: Iperf and libtins C++ API.

[10] https://github.com/AndersonPaschoalon/ProjetoMestrado

[11] We label the tags of SIMITAR control version on GitHub as lemurs species names (https://en.wikipedia.org/wiki/List_of_lemur_species)

For schedule of the timing of traffic generated by each flow, we implemented three methodologies: `usleep()` C function,`select()` C function and *pooling*. Here we use `usleep()` . We implemented the class `IperfFlow`, responsible for generate the traffic of each flow, using `popen()` and `pclose()` to instantiate Iperf processes, responsible for generating the traffic. Traffic customization on Iperf has many limitations. It cannot assign arbitrary IP addresses as source and destination since it must establish a connection between the source and destination. For the transport layer, it just supports TCP and UDP protocol, and constant bandwidth traffic. On the other hand, it enables customization of transmission time, number of packets, windows size, TTL, packet sizes, payload and many other features. Since Iperf has to establish a communication, SIMITAR must operate in the client mode on the source, and server mode on the destinations.

Libtins enable the creation and emission of arbitrary packets and do not require the establishment of a connection. Thus SIMITAR does not need to operate in server mode on the destination. The packet customization capability is vast, and enable a full usage of our model parameters. Control interpacket times stays for future work.

B. Results

We display our results in the figures 10 to 12, and in the table II, where the original and the synthetic traffics are compared. As we can see in the figure 9, the generated traffics are not identical regarding bandwidth, however both presents fractal-like shape. The Hurst exponent of inter-packet times in every case has an error smaller than 10% compared to the original in every case. This result indicates that in fact, the fractal-level of each synthetic traffic is indeed similar to the original.

The plot of flows per second seems much more accurate visually since most of the peaks match. Indeed, no visual lag between the plots. We can analyze it precisely observing the cumulative flow distribution11, where the results were almost identical on every plot. However, when SIMITAR is replicating the traffic of *lgw10s-pcap* the number of flows per second decreases. It happens because of the methodology of traffic generation of the class *TinsFlow* since it sends the packets of each stream as fast as possible. So each flow occurrence is restricted to smaller intervals. This behavior can be observed as well in the bandwidth plot. It is much larger in the first seconds and small at the end.

The best results achieved by SIMITAR were on the flow distribution characterization. On every experiment made, the cumulative distribution of flows was almost identical. The small imprecisions on the plots are expected and should result from threads and process concurrence of resources and imprecision on the sleep/wake signaling on the traffic generation side. Imprecisions on packet capture buffer may count as well since the operating system did the packet timing. This was our most significant achievement in our research. This result means that our method of flow scheduling and

TABLE II: Sumary of results comparing the original traces (italic) and te traffic generated by SIMITAR, with the description of the scenario.

	skype-pcap	skype, one-hop, iperf	skype, tree, iperf	skype, one-hop, libtins	*lgw10s-pcap*	lgw10s, one-hop, libtins
Hurst Exponent	0.601	0.618	0.598	0.691	0.723	0.738
Data bit rate (kbps)	7	19	19	12	7252	6790
Average packet rate (packets/s)	3	4	5	6	2483	2440
Average packet size (bytes)	260,89	549,05	481,14	224,68	365,00	347,85
Number of packets	1071	1428	1604	2127	24 k	24 k
Number of flows	167	350	325	162	3350	3264

independent traffic generation were effective and efficient on replicating the original traffic at the flow-level. The actual number of flows was much more significant when SIMITAR used Iperf and about the same amount but little small when used libtins. This discrepancy happens with Iperf because it establishes additional connections to control the signaling and traffic statistics. So, this more substantial number of flows comes from accounting, not just the traffic connections, but also with the signaling as well. With libtins, the number of flows is small, because, if it fails to create a new traffic flow, this flows generation execution is aborted.

On Wavelet multiresolution analysis of inter-packet times, the results have changed more in each case. The time resolution chosen was ten milliseconds, and it is represented in $\log_2$ scale. The actual time pf each time-scale j value is given by the equation:

$$t = \frac{2^j}{100}[s] \qquad (1)$$

In the first case (figure 12a), SIMITAR using Iperf in a single-hop scenario, on small time scales the energy level of the synthetic traffic remained almost constant, which indicates white-noise characteristics. The original skype traffic increased linearly, an indication of fractal shape. The synthetic trace started to increase at the time scale 5-6 (300-600 milliseconds). After this scale, the error between the curves become very small. One possible reason for this behavior is the fact that the constant which regulate the minimum burst time (`DataProcessor::m_min_on_time`) is set to 100 milliseconds. For small time scales the traffic become similar to white noise since Iperf just emits traffic with constant bandwidth. For larger scales, it captures the same fractals patterns of the original trace. It also captures a periodicity pattern at the time-scale of 9 seconds. The authors of [20] measured the same periodicity pattern. But on this trace, we observe some periodicity at 11 and 13 time-scales (20 and 80 seconds).

In the second case, on a tree topology on small time scales, the same white-noise behavior is identified. We identify a similar behavior on the energy behavior on larger time-scales, but with a larger gap between the curves. Packet collision on switches requiring retransmission packets explains this behavior. In fact, as we can see in the table II, two hundred more packets are leaving the client on the tree topology compared to the one-hop.

On the last two plots, where we use libtins as packet crafter, the energy level is much higher, and the curves are much less correlated. SIMITAR are not modeling inter-packet with libtins, and are sending packets as fast as possible. We may observe on the figures 9c and 9d that than have much higher peaks compared to the original *skype-pcap*. As we see in the figure 9c, it just captured some periodicity characteristics at the time larger than 11-12 (20-40 seconds). SIMITAR determines larger periods between packets with session OFF times, and the session cut time (`DataProcessor::m_session_cut_time`) is 30 seconds, that explain this behavior.

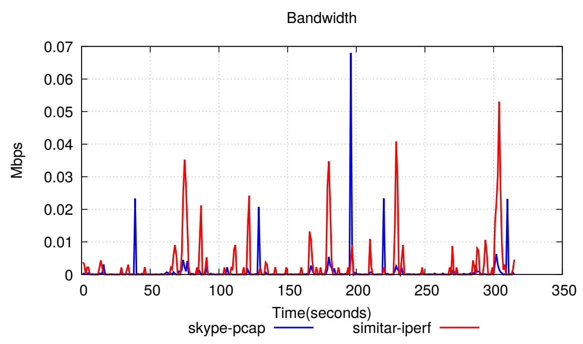
The current implementation still has problems on accurately reproduce *pcaps* with high bandwidth values and a more substantial number of flows. The primary limitation is the time required to generate the trace descriptor. The procedure is still mono-thread, and the linear regression procedure is not optimized. Although creating a trace descriptor of a small pcap file is fast, the time for processing large pcap file with thousands of flows is still prohibitively high, spending dozens of hours.

Using the first 10 seconds of the trace *bigFlows-pcap*, on 10 seconds of operation, Iperf generated much fewer packets than the expected. It is an expected behavior since the operating system couldn't handle so much newer processes (one per flow) in such short time. On the other hand, the same drawback wasn't observed using libtins as traffic generator tool, since being a low-level API makes it much computationally cheaper. Some others unexpected behavior must have a more in-depth investigation, such as libtins generating more packets than expected for *skype-pcap*, but not for *lgw10s-pcap*, and why the creation of some flows fail, and how to fix it if possible.

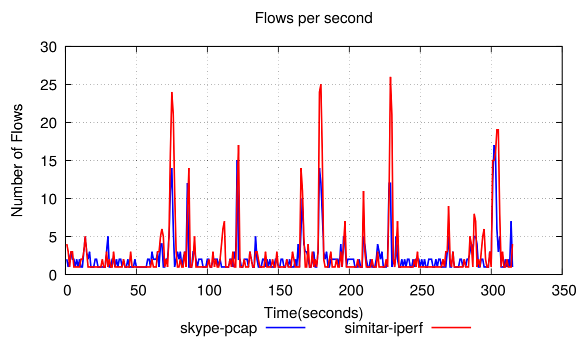
Another promising possible investigation is what traffic each tool can better represent. In terms of number of flows and packets, bandwidth and for larger *pcaps*, *libtins* is a best option. But Iperf has presented a better performance replicating scaling characteristics of applications.

V. Conclusions

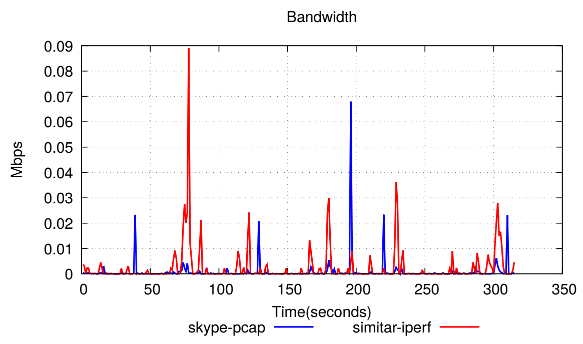
SIMITAR was designed to work at flow-level and packet level. At the flow-level, our methodology can achieve great results. In fact, The cumulative distribution of flows is almost identical on each case. From the perspective of benchmark of a middle-box such as an SND switch, this is a great result, since its performance depends on the number of registered



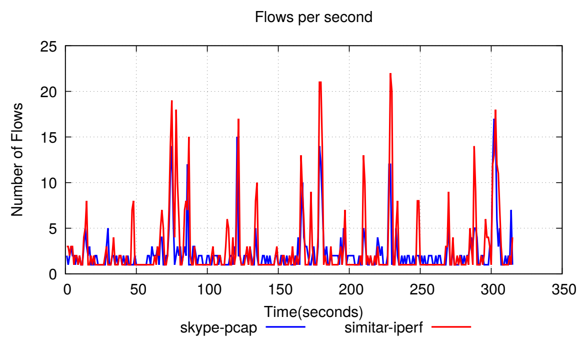
(a) *Iperf, single-hop, skype-pcap*



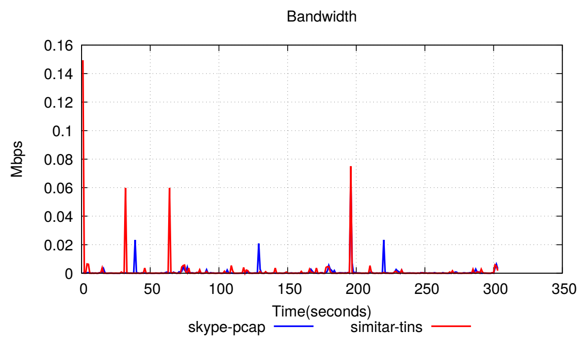
(a) *Iperf, single-hop, skype-pcap*



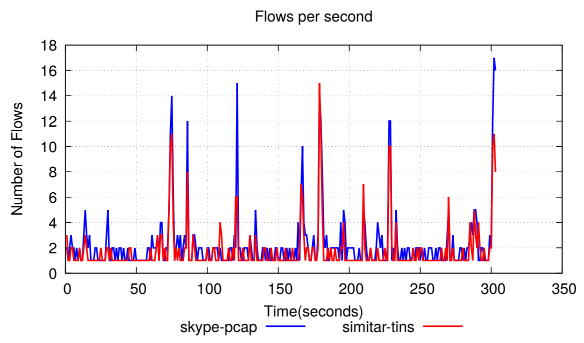
(b) *Iperf, tree, skype-pcap*



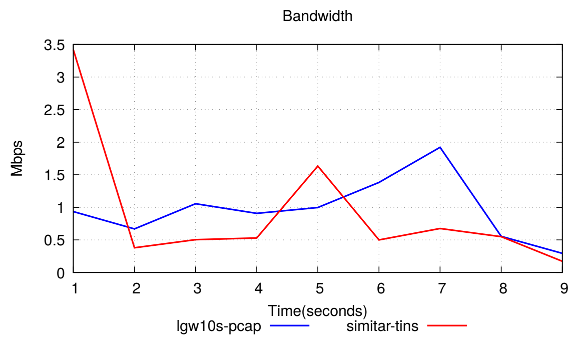
(b) *Iperf, tree, skype-pcap*



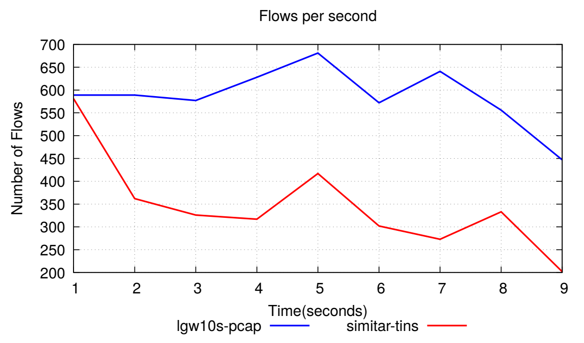
(c) *libtins, single-hop, skype-pcap*



(c) *libtins, single-hop, skype-pcap*



(d) *libtins, single-hop, langw10s-pcap*



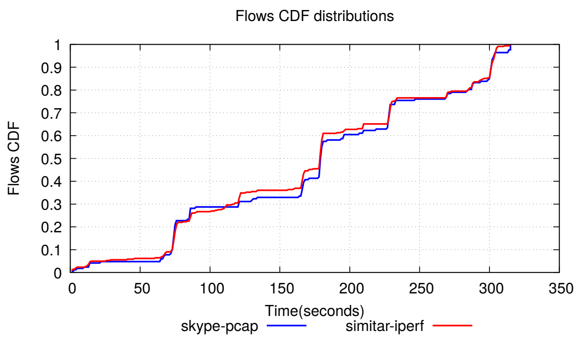
(d) *libtins, single-hop, langw10s-pcap*

Fig. 9: Traces bandwidth.

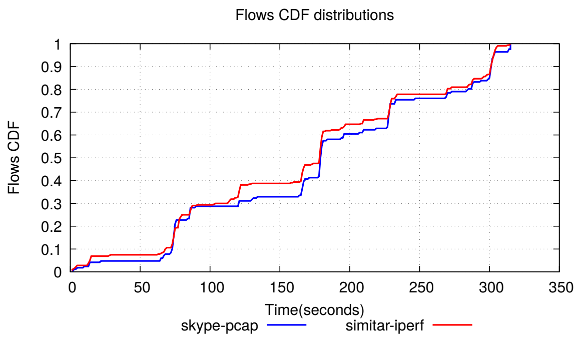
Fig. 10: Flow per seconds

flows. However, because of packets exchanged by signaling

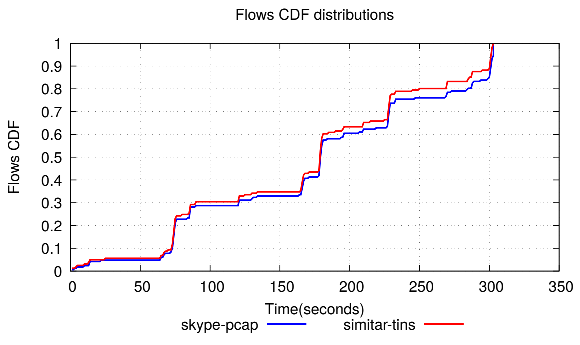
connection, the traffic generated by Iperf, even following the



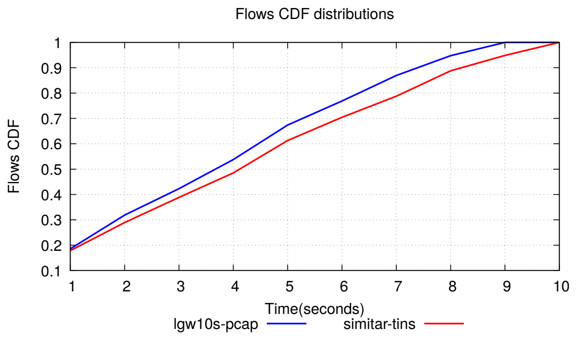
(a) *Iperf, single-hop, skype-pcap*



(b) *Iperf, tree, skype-pcap*

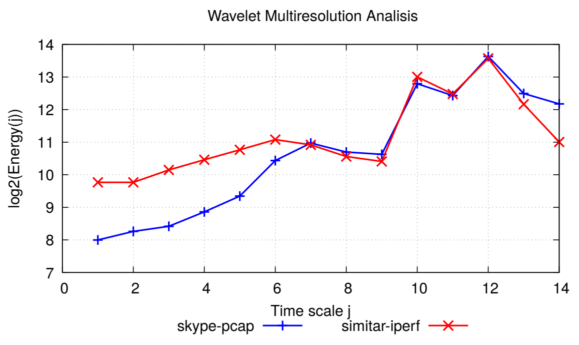


(c) *libtins, single-hop, skype-pcap*

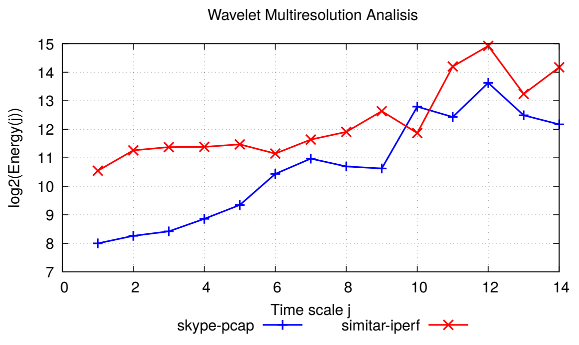


(d) *libtins, single-hop, langw10s-pcap*

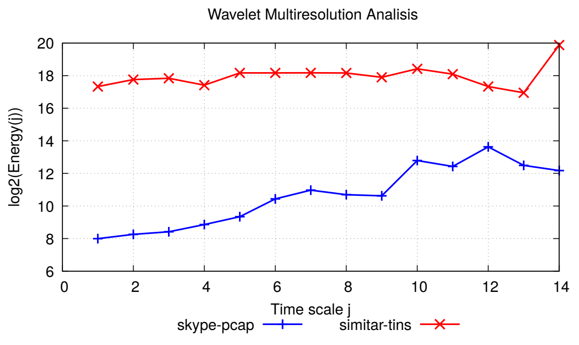
Fig. 11: Flows cumulative distributions.



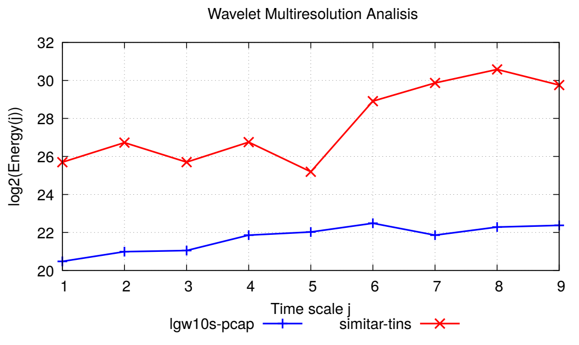
(a) *Iperf, single-hop, skype-pcap*



(b) *Iperf, tree, skype-pcap*



(c) *libtins, single-hop, skype-pcap*



(d) *libtins, single-hop, langw10s-pcap*

Fig. 12: Wavelet multiresolution energy analysis.

same cumulative flow distribution, had created more streams then expected.

At packet level, the current results with Iperf replicate with high accuracy the scaling characteristics of the original traffic, and the number of generated packets are not far than the expected. However, the packet size replication and therefore the bandwidth is not accurate. Also, *libtins* as traffic generator tool still is limited in this aspect.

Once the current implementation of the Flow Generator still does not uses the whole set of parameters from the Compact Trace Descriptor, and many optimizations yet to be made, this is a satisfactory result that validates and proves the potential of our proposed methodology. Even we designing SIMITAR to prioritizing modularity over finner control, when we compare these current results with the more consolidated realistic traffic generator available, they are closer. At the flow-lever our results are at least as good as the achieved by Harpoon [17] and Swing [20]. At the scaling characteristics, on lightweight traces, they are already comparable in quality.

++++++++++++++++++++++++++++++++

VI. ¿¿¿¿¿ Temp Introduction ¡¡¡¡¡

On network benchmarking and testing, it is known that the traffic properties interfere on the network component performance [2] [1] and measurament acuracy [20]. Studies show that a realistic Ethernet traffic provides a different and more variable load characteristics on routers [1], even with the same mean bandwidth consumption. A burstier traffic can cause packet more buffer overflows on network [2] [3] [4], degenerating network performance[12]. Also, a burstier and realistic traffic impacts not just on performance, but on measurement accuracy as well [5] [6]. Along with that, buffers quees and software applications have performance degradation processing small-sized packets. All this indicates that tests with constant bit rate traffic generator tools such as Iperf and Ostinato are important but not enough to guarantee the SLAs for new technologies.

Realistic workload generators are also essential security research [9], and are crucial on the evaluation of firewall middleboxes. It includes studies of intrusion, anomaly detection, and malicious workloads [9]. Another critical point is the flow-oriented operation of SDN networks. Each new flow arriving on an SDN switch demands an extra communication load between it and the controller. This may create a bottleneck between the switch and the controller. We also have a flow-oriented operation on SDN switches. Since its operation relies on queries on flow tables, a stress load must have the same flow properties of an actual ISP scenario.

Alongside with that the introduction of virtualization via techinologies such as NFV still pose challenges on performance, reliability, and security [10]. Closed hardware solutions are easier to pass on Service Layer Agreement since they have a much more predictable behavior. We can expect that its impact mentioned before over virtualized middle-boxes should be even larger, due the extra overhead of a virtualization layer. Therefore, there is a demand for the study of the impact of a realistic traffic on this new sort of environment. How VNFs and virtualized middle-boxes and SDN testbeds will behave if stressed with a realistic traffic load in comparison to a constant rate traffic is a important subject on testing and benchmarking.

The open-source community offers a huge variety of workload generators and benchmarking tools [9] [12] [7] [13]. Most of these tools were built for specific purposes and goals, so each uses different methods of traffic generation. Some traffic generator tools offer support emulation of single application workloads. But this does not correspond to real complex scenarios, with a large number of distributed hosts in an Internet Service Provider (ISP) or even a Local Area Network (LAN). Many tools support a larger set of protocols and high-performance, such Seagull and Ostinato. Others are also able to control inter-packet times and packet sizes using stochastic models, like D-ITG [9] and MoonGen. Since there is so many features and tools, selecting a good configuration is by itself a research project. How to use each parameter to simulate a specific scenario is a hard question [5] [22]. It is a manual process and demands implementation of scripts or programs leveraging human (and scarce) expertise on network traffic patterns and experimental evaluation.

Some tools like Swing and Harpoon uses capture traces to set intern parameters, enabling an easier configuration. Also, Swing uses complex multi-levels models which are able to provide a high degree of realism [20]. But they have their issues as well. Harpoon does not configure parameters at packet level [17] and is not supported by newer Linux kernels, what may be a huge problem with setup and configuration. Swing [20] aims to generate realistic background traffic, but do not reach high throughputs [20] [5]. Due the fact that its traffic generation engine is coupled to its modeling framework, you can't opt to use a newer/faster packet generation library. The only way of replacing the traffic engine is changing and recompiling the original code. And this is a hard task.

Since synthetic traffic traces generation is mature in academia, the creation of custom network workloads through the configuration of high-throughput open-source is an affordable task. But it is not often available in an automatic way, and most of the times is a challenging question [5], and still, requires expert knowledge. It requires time, study, and is vulnerable to human mistakes and lack of validation. Such work may require weeks to complete a realistic reproduction of a single scenario, so most of the time it is just not done. We argue that widely (i.e. affordable) existing approaches can be regarded as simplistic often point solutions to more general cases.

Also, just choosing which workload generator tool may fit better for the user needs is not a simple question. Tools like D-ITG[13] provide support to many different stochastic functions, Ostinato[14] provides a larger support for protocols, and a higher throughput for each thread [7], and others like Seagull[15] are

[12]Features such as packet-trains periods and inter-packet times affect traffic burstiness

[13]http://traffic.comics.unina.it/software/ITG/

[14]http://ostinato.org/

[15]http://gull.sourceforge.net/doc/WP_Seagull_Open_Source_tool_for_IMS_testing.pdf

responsive.

References

[1] J. Sommers and P. Barford, "Self-configuring network traffic generation," in *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '04. New York, NY, USA: ACM, 2004, pp. 68–81. [Online]. Available: http://doi.acm.org/10.1145/1028788.1028798

[2] Y. Cai, Y. Liu, W. Gong, and T. Wolf, "Impact of arrival burstiness on queue length: An infinitesimal perturbation analysis," in *Proceedings of the 48h IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference*, Dec 2009, pp. 7068–7073.

[3] A. J. Field, U. Harder, and P. G. Harrison, "Measurement and modelling of self-similar traffic in computer networks," *IEE Proceedings - Communications*, vol. 151, no. 4, pp. 355–363, Aug 2004.

[4] T. Kushida and Y. Shibata, "Empirical study of inter-arrival packet times and packet losses," in *Proceedings 22nd International Conference on Distributed Computing Systems Workshops*, 2002, pp. 233–238.

[5] G. Bartlett and J. Mirkovic, "Expressing different traffic models using the legotg framework," in *2015 IEEE 35th International Conference on Distributed Computing Systems Workshops*, June 2015, pp. 56–63.

[6] K. V. Vishwanath and A. Vahdat, "Evaluating distributed systems: Does background traffic matter?" in *USENIX 2008 Annual Technical Conference*, ser. ATC'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 227–240. [Online]. Available: http://dl.acm.org.ez88.periodicos.capes.gov.br/citation.cfm?id=1404014.1404031

[7] S. Srivastava, S. Anmulwar, A. M. Sapkal, T. Batra, A. K. Gupta, and V. Kumar, "Comparative study of various traffic generator tools," in *Engineering and Computational Sciences (RAECS), 2014 Recent Advances in*, March 2014, pp. 1–6.

[8] E. Castro, A. Kumar, M. S. Alencar, and I. E.Fonseca, "A packet distribution traffic model for computer networks," in *Proceedings of the International Telecommunications Symposium – ITS2010*, September 2010.

[9] A. Botta, A. Dainotti, and A. Pescap, "A tool for the generation of realistic network workload for emerging networking scenarios," *Computer Networks*, vol. 56, no. 15, pp. 3531 – 3547, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128612000928

[10] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *Communications Magazine, IEEE*, vol. 53, no. 2, pp. 90–97, Feb 2015.

[11] D. Kreutz, F. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, Jan 2015.

[12] S. Molnar, P. Megyesi, and G. Szabo, "How to validate traffic generators?" in *2013 IEEE International Conference on Communications Workshops (ICC)*, June 2013, pp. 1340–1344.

[13] S. S. Kolahi, S. Narayan, D. D. T. Nguyen, and Y. Sunarto, "Performance monitoring of various network traffic generators," in *Computer Modelling and Simulation (UKSim), 2011 UkSim 13th International Conference on*, March 2011, pp. 501–506.

[14] A. Botta, A. Dainotti, and A. Pescape, "Do you trust your software-based traffic generator?" *IEEE Communications Magazine*, vol. 48, no. 9, pp. 158–165, Sept 2010.

[15] P. Barford and M. Crovella, "Generating representative web workloads for network and server performance evaluation," *SIGMETRICS Perform. Eval. Rev.*, vol. 26, no. 1, pp. 151–160, Jun. 1998. [Online]. Available: http://doi.acm.org/10.1145/277858.277897

[16] N. L. Antoine Varet, "Realistic network traffic profile generation: Theory and practice," *Computer and Information Science*, vol. 7, no. 2, 2014.

[17] J. Sommers, H. Kim, and P. Barford, "Harpoon: A flow-level traffic generator for router and network tests," *SIGMETRICS Perform. Eval. Rev.*, vol. 32, no. 1, pp. 392–392, Jun. 2004. [Online]. Available: http://doi.acm.org/10.1145/1012888.1005733

[18] "Tcpreplay home," http://tcpreplay.appneta.com/, [Online; accessed May 14th, 2016].

[19] "Traffic generator," http://www.postel.org/tg/, 2011, [Online; accessed May 14th, 2016].

[20] K. V. Vishwanath and A. Vahdat, "Swing: Realistic and responsive network traffic generation," *IEEE/ACM Transactions on Networking*, vol. 17, no. 3, pp. 712–725, June 2009.

[21] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*, ser. IMC '15. New York, NY, USA: ACM, 2015, pp. 275–287. [Online]. Available: http://doi.acm.org/10.1145/2815675.2815692

[22] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson, "On the self-similar nature of ethernet traffic (extended version)," *IEEE/ACM Transactions on Networking*, vol. 2, no. 1, pp. 1–15, Feb 1994.

[23] Y. Yang, "Can the strengths of aic and bic be shared? a conflict between model indentification and regression estimation," *Biometrika*, vol. 92, no. 4, p. 937, 2005. [Online]. Available: +http://dx.doi.org/10.1093/biomet/92.4.937

[24] L. O. Ostrowsky, N. L. S. da Fonseca, and C. A. V. Melo, "A traffic model for udp flows," in *2007 IEEE International Conference on Communications*, June 2007, pp. 217–222.