SIMITAR: realistic and autoconfigurable traffic generator

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 have a different impact compared to constant traffic tools like Iperf, even with the same bandwidth. A burstier traffic may cause buffers overflows when a constant traffic does not, 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. Today 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, that have a separated moddelling framework from the traffic generator, which is flow-oriented, and is autoconfigurable. It creates and use as imputs compact traces descriptos, XML files that describes all features of the traffics. Currently we are able to replicate with acuracy**

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

I. Introduction

On network benchmarking and testing, it is known that type of traffic used on performing tests matters [1]. Studies show that a realistic Ethernet traffic provides a different and more variable load characteristics on routers [2], even with the same mean bandwidth consumption. It indicates that tests with constant bit rate traffic generator tools such as Iperf and Ostinato are esseintial but not ehough to guarantee the SLAs for newer technologies. T

A burstier traffic can cause packet more buffer overflows on network [3] [4] [5], degenerating network performance[1]. Also, a burstier and realistic traffic impacts not just on performance, but on measurement accuracy as well [6] [7]. Along with that, buffers quees and software applications have performance degradation processing small-sized packets. Realistic workload generators are also essential security research [8], and are crucial on the evaluation of firewall middleboxes. It includes studies of intrusion, anomaly detection, and malicious workloads [8]. Since on traditional hardware-based types of *middleboxes*, the impact of realistic traffic is not negligible; we can expect that its impact over virtualized middle-boxes should be even larger, due the extra overhead of a virtualization layer. 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. 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.

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

II. Literature Review

III. System Architecture and Modelling

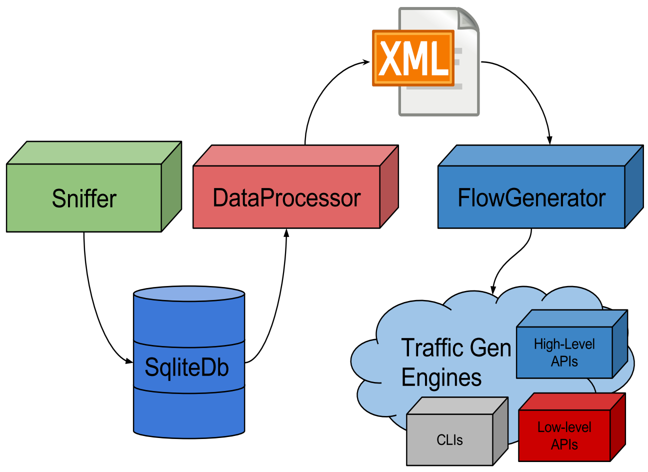


Fig. 1: Architecture of SIMITAR

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 [9], through header fields matches (Link Protocol, Network Protocol, Network Source Address, Network Destination Address, Transport Protocol, Transport Source Port, Transport Destination Port). It can work over a *pcap* file or over an Ethernet interface.

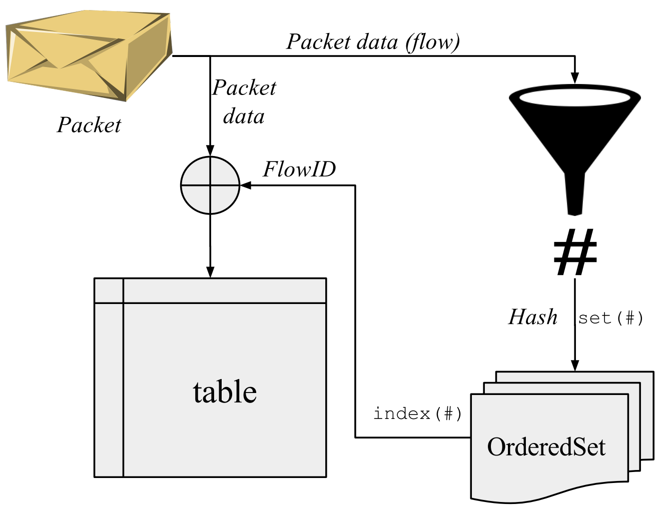


Fig. 2: SIMITAR's sniffer hash-based flow classification

The second version was implemented using Python. This version used Pyshark[2] as sniffer library. The flow classification is made by a class called FlowId. It has class we developed called OrderedSet. A set is a list of elements with no repetition but do not keep track of the insertion order. Our OrderedSet does. Also, it make use of a 64 bits hash function of the family FNV[3]. The listed header fields are inputs for a hash function, its value is set on the ordered set, and its order is returned. This value is set as a packet flowID.

B. *SQLite database*

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

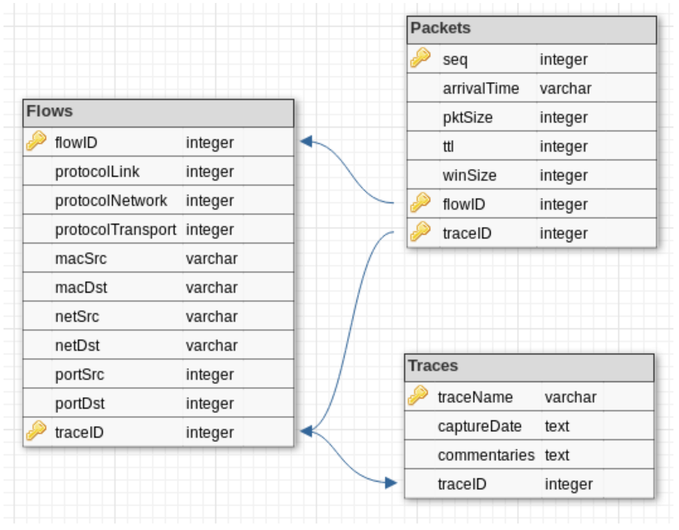
C. *Trace Analyzer*

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 an original traffic trace trough a set of flows, each of them described by a set of parameters, like

[2]https://pypi.python.org/pypi/pyshark

[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



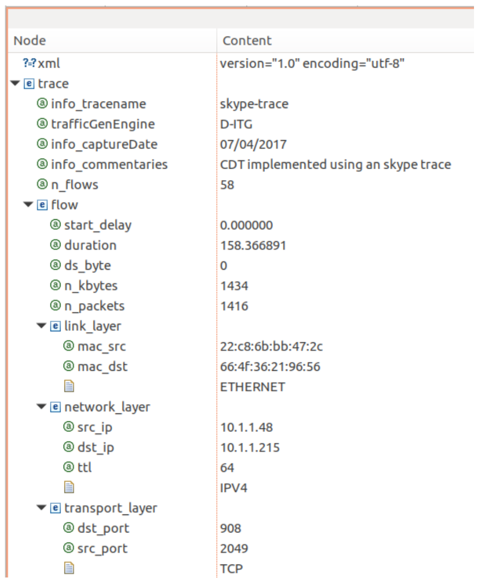
Packets	
seq	integer
arrivalTime	varchar
pktSize	integer
ttl	integer
winSize	integer
flowID	integer
traceID	integer

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

Traces	
traceName	varchar
captureDate	text
commentaries	text
traceID	integer

Fig. 3: SIMITAR's SQLite database relational model

header information and analytical models. Therefore, the Trace Analyze learns the features from raw data of traces (stored in the SQLite database) and write them in 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.



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. On the left, we present a dissected flow, and on the right a set of flows.

1) Flow features: Some features are unique per flow and can be 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.

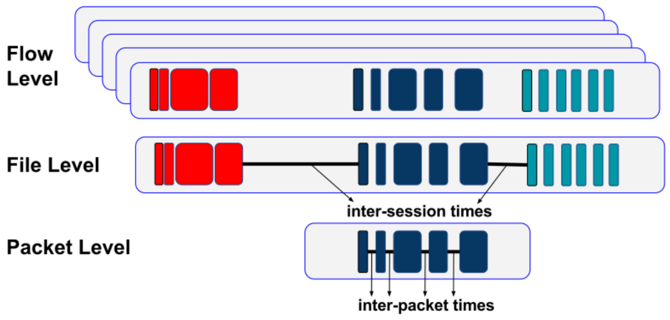


Fig. 5: The schema of the modified version of the Harpoon algorithm we adopt on SCIMITAR.

2) Inter Packet Times: To represent inter packet times, we adopt a simplified version of the Harpoon traffic model. A deep explanation of the original model can be found at [10] and [2]. Here, we will explain our simplified version, which is illustrated at figure 5. Harpoon uses a more conventional definition of each level, based on the measurement of SYN and ACK TCP flags. It use TCP flags (SYN) to classify packets in different levels, named there 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 (as Harpoon does) 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 transmitted downloading a file, of UDP packets of a 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. And the flow level refers to the conjunct of flow classified by the Sniffer.

Flow-layer: The Trace Analyzer loads the flow arrival times from the database, and calculates the inter-packet times within the flow context.

Session-layer: At the session layer, we use a deterministic approach for evaluating times between files and the actual file transference, the OFF and ON times of a packet train model, respectivaly. We choose a deterministic model because in this way we can express diurnal behavior [10]. 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. We did this, first because ON times can be arbitrary smalls, and they could be incompatible with acceptable ON periods for traffic generators. Second, because in the case of just one packet, the ON time would be zero. OFF times, on the other hand, are defined by the constant `m_session_cut_time`. If the time between packets of the same flow is larger than this, we consider it belonging to a different file, so it is a Session OFF time. In this case, we use the same value of the constant *Request and Response timeout* of Swing [1] (30 seconds). The control of ON/OFF periods is made by the `NetworkFlow` class.

File-layer: Here 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 `m_session_cut_time`. 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 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) [11] [12]. We will enter in deeper details on this methodology on the chapter **??**. The methodology of selection is presented in the figure 6

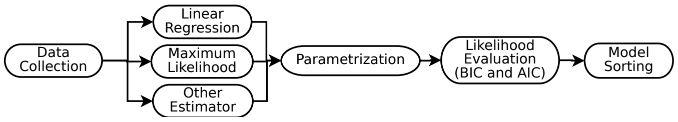


Fig. 6: Diagram of parameterization and model selection for inter-packet times and inter-file times.

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 [13] [11] [14], 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 [14].

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 do 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 [13] [14], 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) Compact Trace Descriptor: An example of the final result of all the methods is presented in the XML code down below. It illustrates a flow of a *Compact Trace Descriptor*(CDT) file.

D. Flow Generator

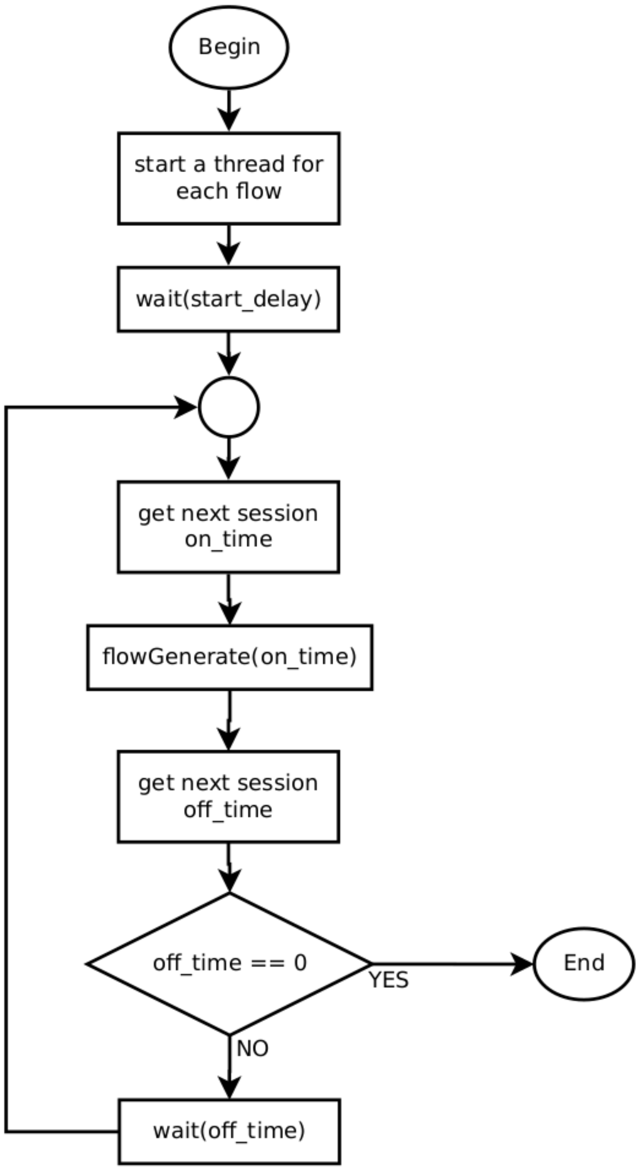


Fig. 7: Simplified-harpoon emission algorithm

The Flow Generator handles the data on the CTD file, and use to retrieve parameters for traffic generation. It crafts and controls each flow in a separated thread.

We first implemented it using the D-ITG API. But our Flow Generator can use any traffic generator with API, CLI or script interface as well. To easily expand this component, we use the design pattern factory. If the user wants to introduce support for a new traffic generator, he just has to expand the class `DummyFlow` and add the support on the method `make_flow()` of the class `NetworkFlow`. We will discuss this deeper in the next section.

This component works in two different layers according to the traffic generators classification introduced in the chapter **??**: at the Flow level, and packet level, as presented in the figure **??**.

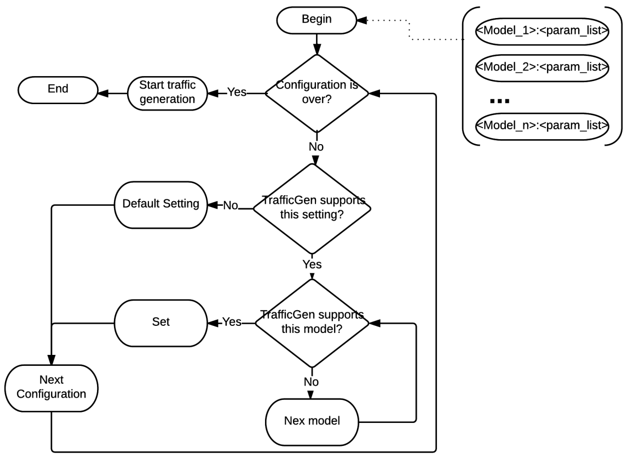


Fig. 8: Traffic engine configuration model

1) Flow-level Traffic Generation Operation: At the flow level, it controls each flow using the algorithm at figure **??**. This algorithm basically handles the our model defined at the figure 5.

It starts a *thread* for each flow in the *Compact Trace Descriptor*, then the thread sleeps its `start_delay`. It is the time when the first flow packet was captured in the original capture. it then calls the underlying traffic generation method, passing to it the file ON time, number of packets and number of bytes to be sent (file size). Then it sleeps the next Session-OFF time from the stack until it ends. When the list of ON/OFF times is over, the thread ends. The thread scheduling/joining is implemented by the class `NetworkTrace` and its sleep/wake process by the class `DummyFlow`.

E. Network Traffic Generator

A network traffic generator software that should provide its API or script interface for the *FlowGenerator* component.

Since `DITGsend()` is thread safe, no mutex is need. It would be needed if a lower level API, such Libtins or Libpcap was used. In that case, these APIs would permit control precisely each packet sent through the interface, but a mutex would be necessary. An scriptable tool, such as Iperf can be used as well, using `fork()` or `popen()`.

As we claimed before, this tools is simple of being expanded for almost any traffic generation API, since its modeling framework is not coupled to the traffic generation. The figure 9 illustrates it well. It presents three possibilities of different traffic generation engines, all being configured and scheduled in a platform-agnostic way.

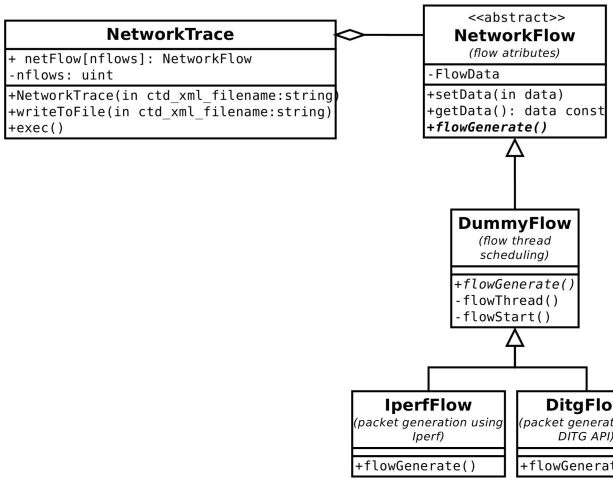


Fig. 9: Class hierarchy of NetworkTrace and NetworkFlow, which enables the abstraction of the traffic generation model of the traffic generation engine.

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 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.

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) [10], and the Wavelet multiresolution analysis. On every case, the 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.

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

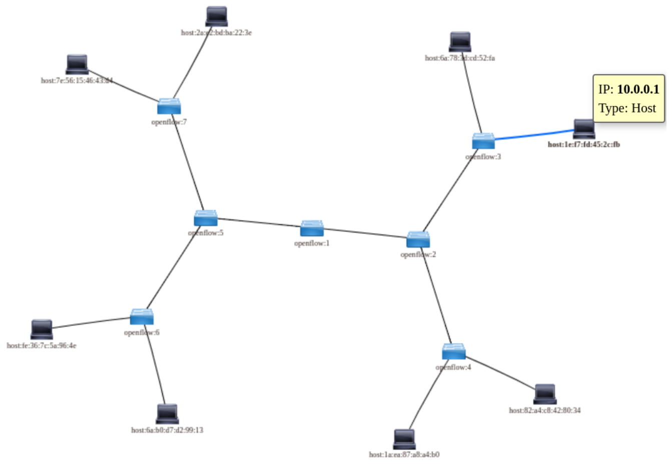


Fig. 10: Tree SDN topology emulated by mininet, and controlled by OpenDayLight Beryllium

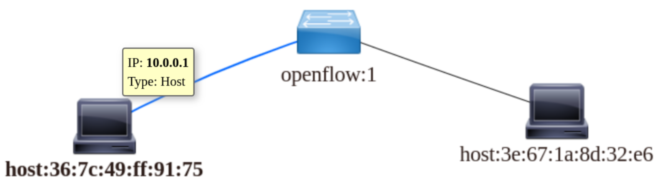


Fig. 11: Single hop SDN topology emulated by mininet, and controlled by OpenDayLight Beryllium

We use as testbeds: a tree topology (figure 10), similar to tests performed by Swing [1] [7] [6], 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[5]: 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)[6], as tagged at the GitHub repository. SIMITAR already have two functional traffic generator engines: Iperf and libtins C++ API.

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.

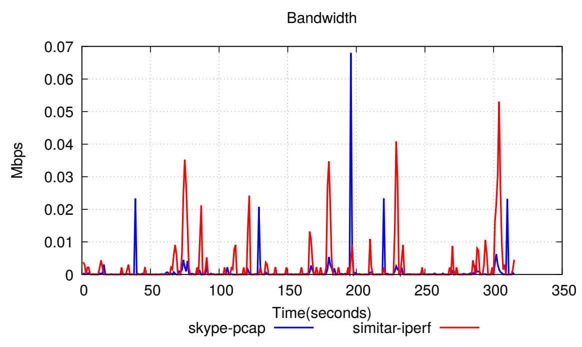
Libtins enable the creation and emission of arbitrary packets and do not require the establishment of a connection. Thus SIMITAR does not need to operate in server mode on the destination. The packet customization capability is vast, and enable a full usage of our model parameters. Control inter-packet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

B. Results

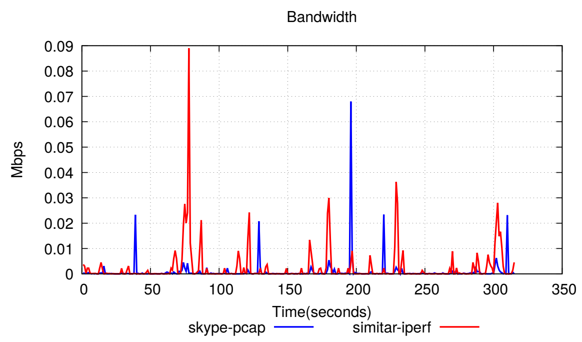
We display our results in the figures 13 to 15, and in the table II, where the original and the synthetic traffics are compared. As we can see in the figure 12, the generated traffics are not identical regarding bandwidth, however both presents fractal-like shape. The Hurst exponent of inter-packet times 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 distribution14, 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.

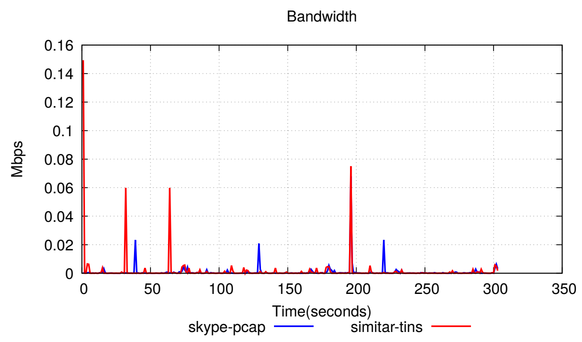
The best results achieved by SIMITAR were on the flow distribution characterization. On every experiment made, the cumulative distribution of flows was almost identical. The



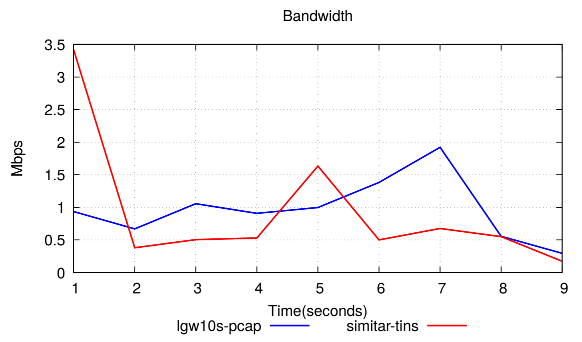
(a) *Iperf, single-hop, skype-pcap*



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



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



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

Fig. 12: Traces bandwidth.

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

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

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

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 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:

$$t = \frac{2^j}{100}[s] \tag{1}$$

In the first case (figure 15a), 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 [1] measured the same periodicity pattern. But on this trace, we observe some periodicity at 11 and 13 time-scales (20 and 80 seconds).

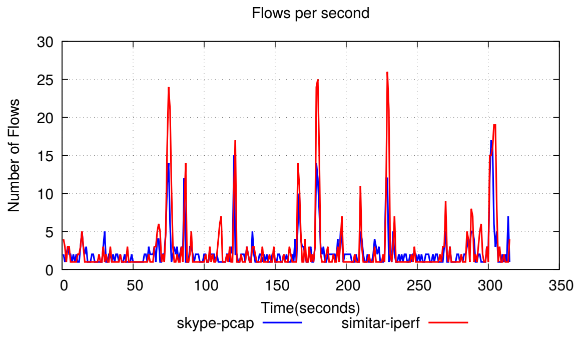
In the second case, on a tree topology on small time scales, the same white-noise behavior is identified. We identify a similar behavior on the energy behavior on larger time-scales, but with a larger gap between the curves. Packet collision on switches requiring retransmission packets explains this 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 12c and 12d that than have much higher peaks compared to the original *skype-pcap*. As we see in the figure 12c, 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.

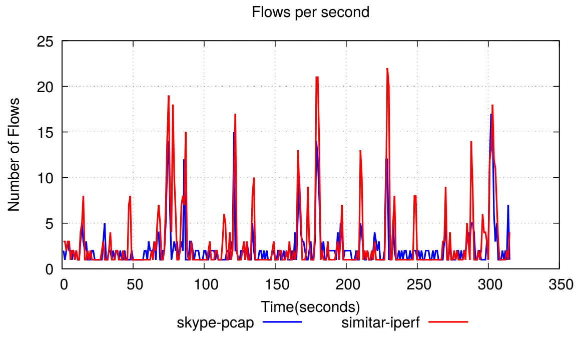
The current implementation still has problems on accurately reproduce *pcaps* with high bandwidth values and a more substantial number of flows. The primary limitation is the time required to generate the trace descriptor. The procedure is still mono-thread, and the linear regression procedure is not 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.

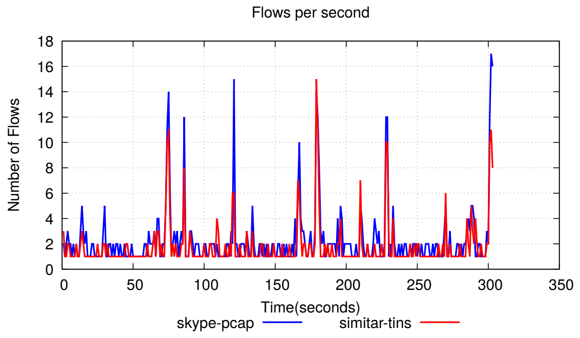
Another promising possible investigation is what traffic each tool can better represent. In terms of number of flows and packets, bandwidth and for larger *pcaps*, *libtins* is a best option. But Iperf has presented a better performance replicating scaling characteristics of applications.



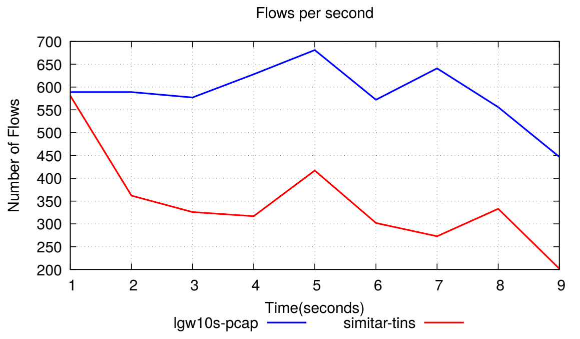
(a) *Iperf, single-hop, skype-pcap*



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

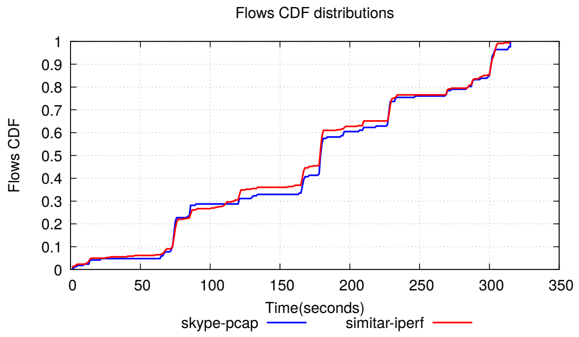


(c) *libtins, single-hop, skype-pcap*

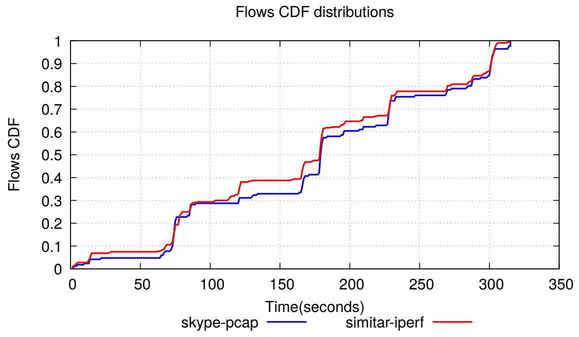


(d) *libtins, single-hop, langw10s-pcap*

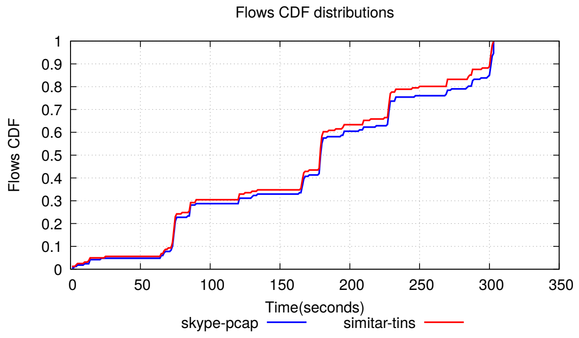
Fig. 13: Flow per seconds



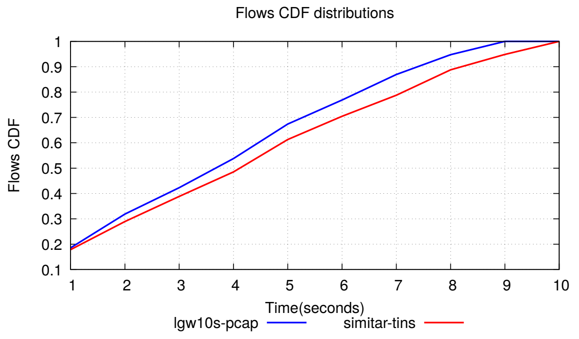
(a) *Iperf, single-hop, skype-pcap*



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



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

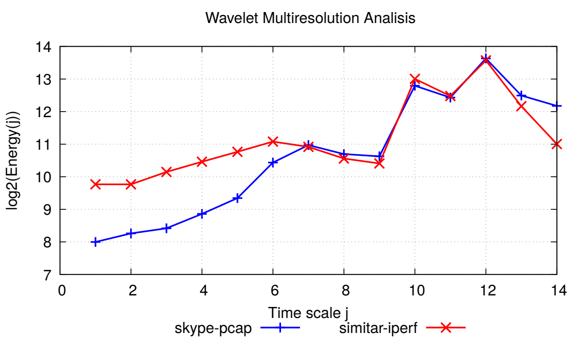


(d) *libtins, single-hop, langw10s-pcap*

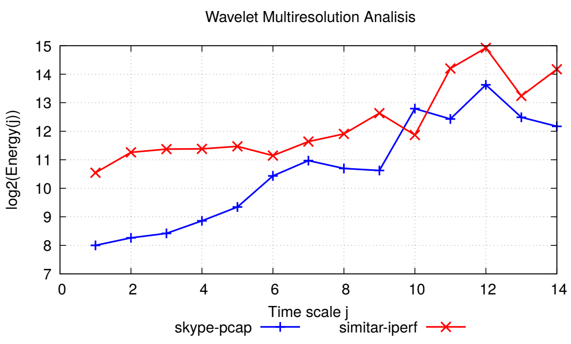
Fig. 14: Flows cumulative distributions.

V. Conclusions

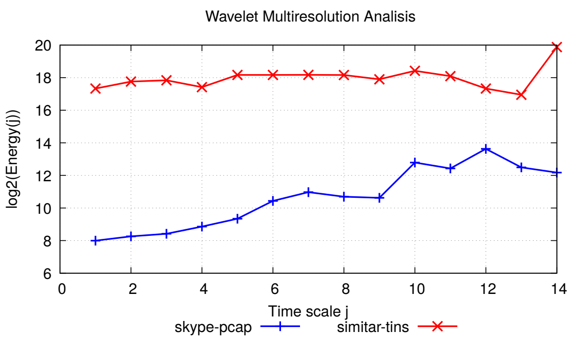
SIMITAR was designed to work at flow-level and packet level. At the flow-level, our methodology can achieve great



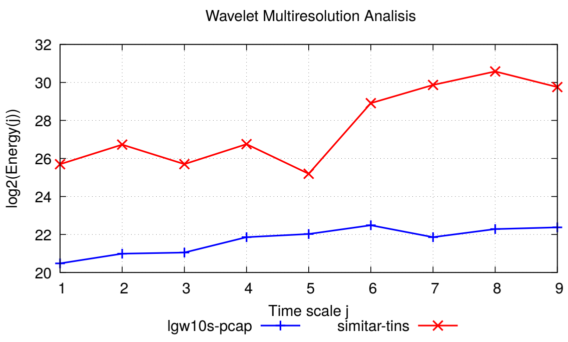
(a) *Iperf, single-hop, skype-pcap*



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



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



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

Fig. 15: Wavelet multiresolution energy analysis.

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 flows. However, because of packets exchanged by signaling connection, the traffic generated by Iperf, even following the 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 [10] and Swing [1]. At the scaling characteristics, on lightweight traces, they are already comparable in quality.

VI. Conclusion

References

[1] 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.

[2] 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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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

[8] 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

[9] 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.

[10] 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

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

[12] 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

[13] 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.

[14] 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.