UNICAMP

UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Engenharia Elétrica e de Computação

Anderson dos Santos Paschoalon

SIMITAR: A Tool for Generation of Synthetic and Realistic Network Workload for Benchmarking and Testing

SIMITAR: Uma Ferramente para Geração de Trafego de Rede Sintético e Realistico para Benchmarking e Testes

CAMPINAS

2017

Anderson dos Santos Paschoalon

SIMITAR: A Tool for Generation of Synthetic and Realistic Network Workload for Benchmarking and Testing

SIMITAR: Uma Ferramente para Geração de Trafego de Rede Sintético e Realistico para Benchmarking e Testes

Dissertation presented to the Faculty of Electrical and Computer Engineering of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Electrical Engineering, in the area of Computer Engineering.

Dissertação apresentada à Faculdade de Engenharia Elétrica e Computação da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de Mestre em Engenharia Eletrica, na Àrea de Engenharia de Computação.

Supervisor: Prof. Dr. Christian Rodolfo Esteve Rothenberg

Este exemplar corresponde à versão final da dissertação defendida pelo aluno Anderson dos Santos Paschoalon , e orientada pelo Prof. Dr. Christian Rodolfo Esteve Rothenberg

CAMPINAS
2017

Agência(s) de fomento e nº(s) de processo(s): FUNCAMP

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Rose Meire da Silva - CRB 8/5974

Lachos Peréz, Danny Alex, 1983-

L118d Delivering application-layer traffic optimization services based on public routing data at Internet eXchange points / Danny Alex Lachos Peréz. – Campinas, SP : [s.n.], 2016.

Orientador: Christian Rodolfo Esteve Rothenberg.

Dissertação (mestrado) – Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Roteamento (Administração de redes de computadores). 2. Engenharia de tráfego. 3. Redes de computadores. 4. Banco de dados. I. Esteve Rothenberg, Christian Rodolfo,1982-. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Serviços para otimização do tráfego de aplicações a partir de informações públicas de roteamento nos Pontos de troca de tráfego na internet
Palavras-chave em inglês:
Routing (Computer network management)
Traffic engineering
Computer networks
Database
Área de concentração: Engenharia de Computação
Titulação: Mestre em Engenharia Elétrica
Banca examinadora:
Christian Rodolfo Esteve Rothenberg [Orientador]
Marcos Antonio de Siqueira
Edmundo Roberto Mauro Madeira
Data de defesa: 08-07-2016
Programa de Pós-Graduação: Engenharia Elétrica

COMISSÃO JULGADORA - DISSERTAÇÃO DE MESTRADO

Candidato: Anderson dos Santos Paschoalon RA: 083233
Data da Defesa:
Título da Tese:
"SIMITAR: A Tool for Generation of Synthetic and Realistic Network Workload for Benchmarking and Testing"
"SIMITAR: Uma Ferramente para Geração de Trafego de Rede Sintético e Realistico para Benchmarking e Testes"

Prof. Dr. Christian Rodolfo Esteve Rothenberg (Presidente, FEEC/UNICAMP)
Prof. Dr. Edmundo Roberto Mauro Madeira (IC/UNICAMP) - Membro Titular
Prof. Dr. Marcos Antonio de Siqueira (PADTEC) - Membro Titular

Ata de defesa, com as respectivas assinaturas dos membros da Comissão Julgadora, encontra-se no processo de vida acadêmica do aluno.

Nesssa dedicatoria, gostaria de agradescer a todos que me ajudarem por essa etapa, direta ou indiretamente. Des daqueles que me inspiraram e me motivaram a seguir por esse caminho, aqueles que me ensinaram e me ajudaram durante o processo, e a aqueles cuja simples companhia me deram energia e me motivaram para estar aqui onde estou hoje. A todos, seja os que estão listado abaixo, como aqueles cuja minha memória não me ajudou na escrita desse texto.

Gostaria de agradecer ao meu professor e orientador Christian Esteves Rothemberg, sem o qual, seja pelo ensino, seja pela orientação e apoio durante o projeto, este trabalho não teria saído do papel.

Agradeço também a todos os Intrigers, colegas de grupo e de bancada, Alex, Javier, Nathan, Daniel, Danny, Gyanesh, Rafael, Fabricio e todos os demais. Agradeço a todos os demais colegas de laboratorio do LCA, em especial a Mijail, Suelen, Amadeu, Paul, ...

Agradeço a todos os companheiros e amigos que fiz em todos esses anos de Unicamp

Agradeço a todos os grandes amigos e companheiros da Opus Dei, em especial Padre Fabiano. E também todos meus caros amigos da Igreja Batista Fonte.

Agradeço aos meus companheiros passados e atuais da casa P7: Lucas Zorzetti(Xildo),

Agradeço a minha família, a meu Pai Tirso José Paschoalon por todo sua preocupação e ensino. A minha Mãe Rosangela dos Santos Mota, por todo o seu carinho e amor. E a minha irmã Ariela Paschoalon, pela companhia e afeto.

E por ultimo agradeço a Deus por todos seu dons, proteção aamor

Acknowledgements

First of all, I would like to thank my advisor Prof. Dr. Christian Rothenberg for the trust and for letting me be part of his selected group of students. I would not be able to imagine the undertaking of this research without his innovative ideas, consistent support and continuous encouragement. I would therefore like to express my gratitude for being a great instructor and an even greater friend.

At the State University of Campinas (Unicamp) I have had the opportunity to learn from the best professors. I would like to acknowledge Prof. Dr. Mauricio Magalhães, Prof. Dr. Edson Borin, Prof. Dr. Edmundo Madeira and Prof. Dr. Léo Pini for sharing their knowledge and experience, which served as a constant motivation throughout this work.

I would like to thank the committee members Marcos Antonio de Siqueira, Edmundo Roberto Mauro Madeira and Christian Rothenberg for the recommendations and insightful criticism they offered me in order to improve the final version of my thesis.

I had the good fortune to meet very talented people within the INTRIG and LCA groups. My special thanks go to Mateus, Samuel, Samira, Gyanesh, Alex, Luis, Raphael, Anderson, Hirley, Ramon, Alaelson, Claudio, Talita, Javier, Elias, Rodrigo, Roberto, Mariana, Ranyeri, Aldo, Wallace, Vitor, Raphael V. and Amadeu, Mariana and Lino for their friendship and their support throughout these two years.

I would like to express special gratitude for the financial and technical support received from the Ericsson Innovation Center (Brazil), which allowed me to carry out this research.

“Discipline, sooner or later, will defeat intelligence.”
“A disciplina cedo ou tarde vencerá a inteligência.”
“La disciplina tarde o temprano vencerá a la inteligencia.”

Japanese proverb

Abstract

Application-Layer Traffic Optimization (ALTO) is a recently standardized protocol that provides abstract network topology and cost maps in addition to endpoint information services that can be consumed by applications in order to become network-aware and to take optimized decisions regarding traffic flows. In this work, we propose a public service based on the ALTO specification using public routing information available at the Brazilian Internet eXchange Points (IXPs). Our ALTO server prototype takes the acronym AaaS (ALTO-as-a-Service) and is based on over 2.5GB of real BGP data from the 25 Brazilian IX.br public IXPs. We evaluate our proposal in terms of functional behaviour and performance via proof-of-concept experiments, which point to the potential benefits of applications being able to take smart endpoint selection decisions when consuming the developer-friendly ALTO APIs.

Keywords: Routing (Computer network management); IXPs (Internet exchange points); Computer networks; SDN (software defined networking).

Resumo

Otimização de Tráfego na Camada de Aplicação (ALTO - *Application-Layer Traffic Optimization*) é um protocolo recentemente padronizado que fornece uma topologia da rede e mapa de custos abstratos, além de serviços de informação de endpoints que podem ser consumidos pelos aplicativos, a fim de tornar-se conscientes da rede e tomar decisões otimizadas sobre os Neste trabalho, propomos um serviço público baseado nas especificações ALTO usando informação de roteamento pública disponível nos Pontos de Troca de Tráfego (PTTs) brasileiros. Nosso protótipo de servidor ALTO, representado pela sigla AaaS (ALTO-as-a-Service), é baseado em mais de 2,5 GB de dados BGP reais dos 25 PTTs públicos brasileiros (IX.br). Nossa proposta é avaliada em termos de comportamento funcional e desempenho através de experimentos de prova de conceito que apontam como potencial benefício das aplicações, a capacidade de tomar decisões inteligentes na seleção de endpoint ao consumir as APIs ALTO. **Palavras-chaves**:

Roteamento (Administração de redes de computadores); Engenharia de tráfego; Redes de computadores; Bancos de dados.

List of Figures

List of Tables

Contents

1 Introduction

1.1 ~~State of Affairs and~~ Motivation

Emerging technologies such as SDN and NFV are great promises. If succeeding at large-scale, they would change drastically the development and operation of computer networks. But, especially on NFV, its enabling technologies such as virtualization still pose challenges on performance, reliability, and security [Han *et al.* 2015]. Closed hardware solutions are easier to pass on Service Layer Agreement since they have a much more predictable behavior. Since it is expected virtualization to impact negatively on performance, these VNFs have to keep its degradation as small as possible. Guarantee the Service Layer Agreements on emerging scenarios is now a harder question. There is a demand for more reliable methods to ensure the SLAs, over different types of loads.

It is already a well-known fact that the type of traffic used on performing tests matters. Studies show that a realistic Ethernet traffic provides a different and more variable load characteristics on routers [Sommers e Barford 2004], even with the same mean bandwidth consumption. It indicates that tests with constant bit rate traffic generator tools are not enough for a complete validation of new technologies. There are many reasons for this behavior, which includes burstiness and packet sizes.

A burstier traffic can cause packet more buffer overflows on network [Cai *et al.* 2009] [Field *et al.* 2004] [Kushida e Shibata 2002], degenerating network performance[1], and measurement accuracy [Bartlett e Mirkovic 2015] [Vishwanath e Vahdat 2008]. 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 [Srivastava *et al.* 2014]. A realistic synthetic traffic must not have a single packet-size but must use a distribution [Castro *et al.* 2010].

Realistic workload generators are also essential security research [Botta *et al.* 2012]. Generation of realistic workloads is important in the evaluation of firewall middleboxes. It includes studies of intrusion, anomaly detection, and malicious workloads [Botta *et al.* 2012].

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

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

controller. This may create a bottleneck between the switch and the controller. Also the SDN switches have a flow-oriented operation. Since its operation relies on queries on flow tables, a stress load must have 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 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 relevant subject.

1.2 ~~Open-source Solutions~~ Related Work

The open-source community offers a huge variety of workload generators and benchmarking tools [Botta *et al.* 2012] [Molnár *et al.* 2013] [Srivastava *et al.* 2014] [Kolahi *et al.* 2011]. Most of these tools were built for specific purposes and goals, so each uses different methods of traffic generation, and enable 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, On/Off periodos, start and sending time, and emulation of applications such as Web server/client communication, VoIP, HTTP, FTP, p2p applications, and many more.

Some traffic generator tools offer support emulation of single application workloads. But this does not correspond to real complex scenarios. Other tools work as packet replay engines, such as TCPreplay and TCPivo. Although in that way is possible to produce a realistic workload at high rates, it comes with some issues. First, the storage space required becomes huge for long-term and high-speed traffic capture traces. Also, obtaining good traffic traces sometimes is hard, due privacy issues and fewer good 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 [Botta *et al.* 2012] and MoonGen. 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. Also, many of the listed traffic generators provides their own API, such as Ostinato Python API, D-ITG C API, and MoonGen LUA API.

Therefore 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 [Bartlett e Mirkovic 2015] [Leland *et al.* 1994]. 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, there are some sollutions on the open source community. 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 [Vishwanath e Vahdat 2009]. But they have their issues as well. Harpoon does not configure parameters at packet level [Sommers *et al.* 2004] and is not supported by newer Linux kernels, what may be a huge problem with setup and configuration. Swing [Vishwanath e Vahdat 2009] aims to generate realistic traffic, but focous on background traffic, and high throughputs it is not a goal off the application [Vishwanath e Vahdat 2009] [Bartlett e Mirkovic 2015]. 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 [Bartlett e Mirkovic 2015], 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.

1.3 *SIMITAR*: SnIffing, ModellIng and TrAffic geneRation

In this section, we are going to present our solution discussed in this text. SIMITAR is a traffic generator able to learn features of a real traffic automatically, and reproduce a synthetic traffic similar to the original. It records a model for the traffic in an XML file we call Compact Trace Descriptor.

But first we will formalize our research targets, and then define a requirement list for SIMITAR, that means, what we want it to do. The main research targets are:

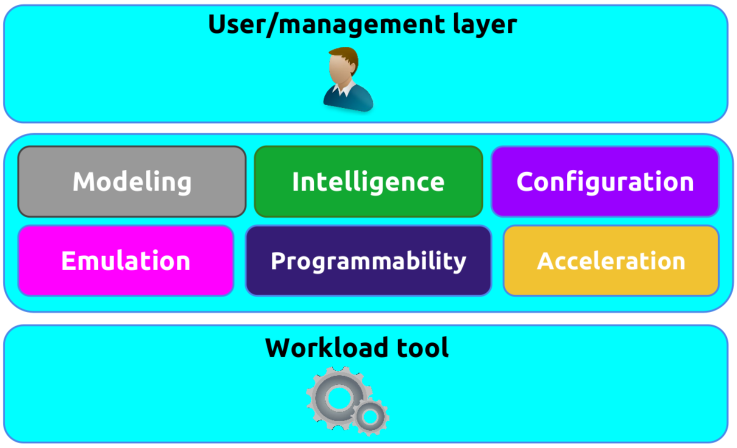


Figure 1 – Architecture conceptual idea: a toll to automatize many tasks on traffic modelling and generation.

- Survey open-source Ethernet workload tools, addressing different features of each one. In this step, we want to evaluate the existing solution ready to use for network researchers and developers, and what can be used and integrated into our project as part of our solution;
- Define what a realistic Ethernet traffic is, and a set of metrics to measure the realism and the similarity between an original and a synthetic traffic.
- Research the main apporoaches and methods found of the literature to create model Ethernet traffic, and for realistic traffic generation.
- Create a general method for modeling and parameterization of Ethernet traffic, aiming to mimic any traffic provided as input;
- Create a software able of *learning* metrics from real network traffic, and based on the that reproduce it workload with similar (but not equal) characteristics, *avoiding* the storage of *pcap* files.

We call our tool *SIMITAR*, an acronym for *SnIffing, ModellIng and TrAffic geneRation*. This acronym summarizes its operation. Now, we are going to define a requirement list for our proposed software.

- It must be able to extract data from a real traffic and store in a database. It must be able to obtain data from real-time traffics and from *pcap* files;
- It must have a flow-based abstract model for traffic generation, not attached to any specific technology. Then, its model should be parameterized using this stored data;
- Human readable: it must produce a human-readable file as output that describes our traffic using our abstract model. We call this file a Compact Trace Descriptor;

- Easy expansion: the traffic modeling and generation must be decoupled. Ideally, it must able to use as traffic generator engine any library or traffic generator tool;
- Easy to use: It must be simple to use. It has to take as input a Compact Trace Descriptor, just as a traffic replay engine (such as TCPreplay) would take a *pcap* file;
- Traffic programmability: It must have what we call *traffic generation progamability*. The compact trace descriptor must be simple and easy to read. In that way, the user may want to create his custom traffic, in a platform agnostic way. The traffic programmability must be flow-oriented;
- The traffic modeling and generation must be flow-oriented. Each flow must be modeled and generated separately.

On the figure 1 we abstract our concepts we stated in a layer diagram model. Or tool must work as an interface for the user automatizing traffic modeling, configuration, and emulation. Also should provide a service intelligence, programmability, and acceleration[2].

Our primary goal is to offer simple configuration, with realism, at a reasonable speed. To do this, we are introducing the concept of programmability for traffic generation. With SIMITAR, the user may program its synthetic traffic in a platform agnostic way. The intermediate layer of the figure 1 summarize, the goal of the project in an illustrative way.

Instead of storing a huge file *pcap* with a size of many *gigas*, we create a lightweight set of models and parameters that describe this same trace. This tool will store the generalized set of parameters in a human and machine-readable XML file we call *Compact Trace Descriptor* (*CTD*). Then, we use this data as input for a traffic generator engine. We do so using APIs or creating new processes in a controlled manner. So will be possible to control packet parameters and flow's behavior in an automatized and self-configuring way.

Using a component methodology, we decouple the traffic generation, from the data collection and parameterization process. Building it in this non-monolithic way, enable a simple port for different traffic generators engines, what make our tools easy for updating with newer technologies. A project guideline is to reuse as much code and components as possible from the open-source community.

[2] Packet acceleration is a concept introduced by DPDK [DPDK – Data Plane Development Kit 2016], which means kernel by-pass. Packet acceleration optimizes the packet processing, and therefore traffic generation, enabling higher throughput rates. But, in the current version, this feature is not implemented, but its methodology is discussed in the section of future works in the chapter **??**.

1.4 Document Overview

In this introductory chapter, we presented an abstract of state of the art, a problem statement, and proposed an approach for research and requirements for development.

In the chapter section 2, we go more in-depth on some subjects mentioned here. First, we present an extensive survey on open-source traffic generator tools. We summarize the benefits, and features supported by each one. After, we offer a brief review of essential topics on realistic traffic generation. We are defining some central concepts we are going to use in this work, such as self-similarity, and heavy-tailed functions. Then, we discuss some techniques of validation of traffic generator tools and some practical examples. We also will analyze some related works.

Chapter 3 introduces the methodology used in our project. We describe *SIMITAR* low-level requirements and define an architecture and their algorithms. We also present its classes design and explain how *SIMITAR* we can expand to any traffic generator engine or library, with an API, CLI interface. We use the Iperf as an example. We explain its operation and suggest some use cases.

In the chapter **??** we go deep in some subjects pointed in the previous chapter. We present how the modeling process works, using a defined data set (which we are going to use in the rest of the work). We also show some evaluation methods to check the modeling quality. We also describe our used and developed algorithms.

In the chapter **??**, we define a set of metrics based on previous tests on validation of traffic generators found in the literature. Here, we focus on packet, flow, and scaling metrics. we test *SIMITAR* in an emulated SDN testbed with Mininet, using OpenDayLight as controller [The OpenDayLight Platform 2017].

Finally, on chapter Conclusion and Future Work **??**, we summarized our work and highlighted future actions to improve *SIMITAR* on realism and performance.

2 Literature Review

In this chapter, we will review some topics from the literature that serve as a base for this whole work. We will define what is a traffic generator tool, and introduce a taxonomy that distinguishes them. After, we will present a survey of many available open-source and free traffic generators. We introduce each tool and then present a feature comparison between each of them. As far as we know, this is the extensible features comparison of traffic generators available so far in the literature. The first two sections are devoted to this topic.

In the third section we will overview network traffic modeling and realistic traffic generation. We will discuss stochastic models that used to describe the Internet traffic. Based on that we will discuss features a realistic traffic should have. We can count self-similarity and burstiness, header fields, packet size and flow features.

In the following section, we will make a discussion on validation of traffic generator tools. First, presenting the main approaches of validation made on literature. Then, we will discuss some validation study cases of related works, always introducing new concepts when needed.

2.1 Traffic generator tools

A traffic generator tool is a system capable of creating and injecting packets into a computer network in a controlled way to generate a synthetic traffic [Molnár *et al.* 2013]. there is a huge variety of traffic generation tools described on the literature [Molnár *et al.* 2013] [Botta *et al.* 2012] and available in the open-source community[1]. This is a cause of many different approaches and methodologies available to network traffic generator's developers [Molnár *et al.* 2013].

New network generators development is directly related with to the current needs of network environment, applications sets, and purpose of use [Molnár *et al.* 2013]. According to the context, you may want to send as many packets as possible through an interface to equipment stress . So, in this case, a maximum throughput traffic generator is more interesting. In others situations, you may want a stochastically realist profile. Or even a traffic generator that reflect a special scenario such as a specific application; for example a client and server exchanging data [Barford e Crovella 1998]. In this case, the target would be to stress a server. Another possibility a huge variety of protocols available, or even support of new protocols.

Alongside with the traffic generators, there are many APIs available. There are low-

[1] http://www.icir.org/models/trafficgenerators.html

level APIs which enables direct packet injection, such as *libpcap* [TCPDUMP/LIBPCAP public repository], *libtins* [libtins: packet crafting and sniffing library], DPDK [DPDK – Data Plane Development Kit 2016] and the GNU C socket library [Sockets]. And there are high-level APIs, provided by traffic generator tools, such as D-ITG [Botta *et al.* 2012], Ostinato [Ostinato Network Traffic Generator and Analyzer 2016] and MoonGen [Emmerich *et al.* 2015]. In this case they enable an easy custom traffic generation and statistics. Also, there are implementations of hardware-based traffic generators using NetFPGAs[2].

In the literature there are many classifications of traffic generators [Varet 2014] [Wu *et al.* 2015] [Molnár *et al.* 2013] [Botta *et al.* 2010]. Some tools may have some features compatible with one or more classes. But classify them it is an efficient way to distinguish the difference between these tools. We will give two different taxonomies:

- According to its abstraction level [Botta *et al.* 2010];
- According to its implementation.

2.1.1 ~~Classes of traffic generator tools~~

The most common traffic generator taxonomy in the literature is about its abstraction level [Botta *et al.* 2010]. There are four groups: Application-level, flow-level, packet-level and multi-level traffic generators. We present a diagram ilustrating its concept at figure 2.

2.1.1.1 According to its abstraction level

Application-level traffic generators: they try to emulate the behavior of network applications, simulating real workloads stochastically and/or responsively [3]. As examples we have Surge [Barford e Crovella 1998] able to emulate the behavior of web clients and services. The behavior of such applications is, in general, based on request-response exchanges models.

Flow-level traffic generators: they are able to configure and reproduce features of flows [Botta *et al.* 2010] [Varet 2014], such as flow duration, start times distributions, and temporal (diurnal) traffic volumes [Botta *et al.* 2010]. Harpoon [Sommers *et al.* 2004] is able to extract these parameters from Cisco NetFlow data, collected from live routers.

Packet-level traffic generators: most traffic generators available fall in this ~~class~~. 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 [Molnár *et al.* 2013]. [Molnár *et al.* 2013] classify them as replay engines, maximum throughput generators, and

[2] http://netfpga.org/site/#/

[3] 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

model based generators. Traffic Replay engines, such TCPreplay [Tcpreplay home], are able to read packet capture files (*pcap* files), and inject copies on a network interface. Model-based generators, such D-ITG [Botta *et al.* 2012], TG [Traffic Generator 2011], 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: this is a more recent class of network traffic generator, and its proposal is to take into account existing interaction among each layer of the network stack, to create background network traffic as close as possible from reality. One of the most relevant tools we can found is Swing [Vishwanath e Vahdat 2009]. Swing take into account many points of each layer, using an approach based on randomness and responsiveness [Vishwanath e Vahdat 2009]. That means, it uses PS and IDT stochastic models, but also take into account flow and application layer behavior. Swing is able to model things like [Vishwanath e Vahdat 2009]:

- 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. [Botta *et al.* 2010] argues that due its complexity, they are rarely used to conduct experimental researchers. Also, we can justify due its overheads, since the performance of such complex traffic generator is limited [Bartlett e Mirkovic 2015]. The authors Vishwanat et al. [Vishwanath e Vahdat 2009] cite that they have generated traces of about of 200 Mbps, about the same result found by [Bartlett e Mirkovic 2015]. In the other hand, D-ITG reached 9808 Mbps [Srivastava *et al.* 2014] in a link of 10 Gbps, but operating as a maximum throughput generator.

2.1.1.2 According to its implementation

Software-only traffic generators: Implementations of traffic generators completely independent of its running hardware platform. This comprehends most of traffic generator tolls.

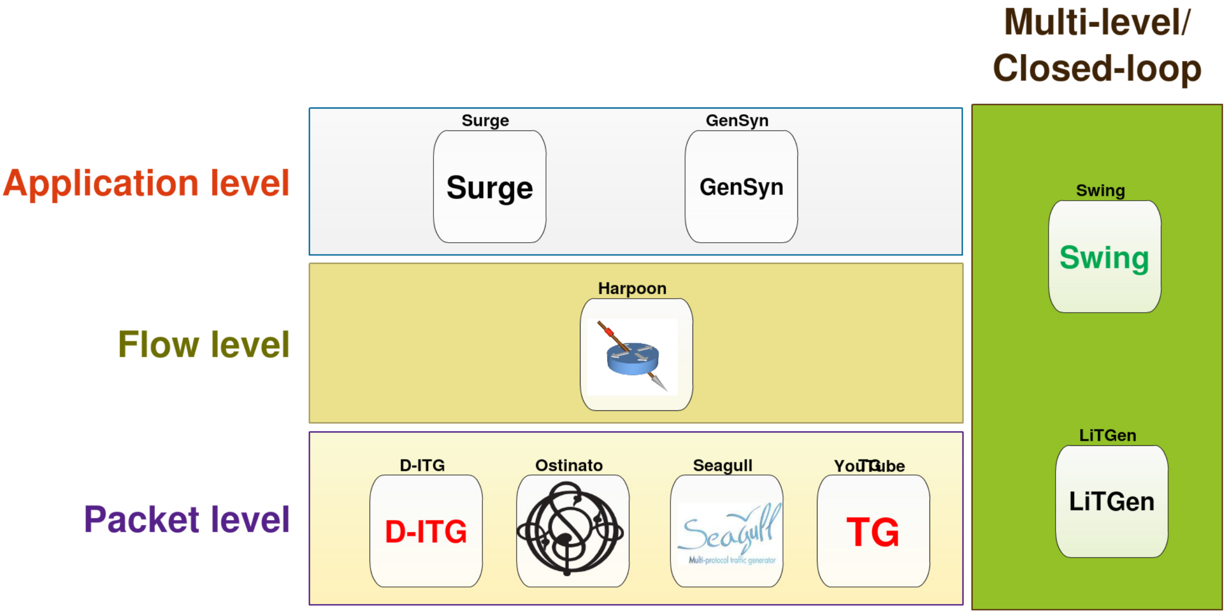


Figure 2 – Diagram representing different traffic generators, according to its abstraction layer.

Software and hardware-dependent traffic generators: Traffic generators implemented in software, but dependent of underlying hardware. The most preeminent examples of this class are implemented over DPDK [DPDK – Data Plane Development Kit 2016]. DPDK works directly on the NIC interface, avoiding overheads of the Operational System. As cited on its official website, this approach permits huge precision and speed in the timing of packets, since it is able to send and receive packets within less than 80 clock cycles.

Hardware traffic generators: This open-source traffic generators implementations are implemented in hardware description language (VHDL/Verilog), and work on NetFPGAs. Some examples of implementations are: PacketGenerator [NetFPGA], Caliper [PreciseTrafGen], and OSNT Packet Generator [OSNT Traffic Generator].

Table 1 – Review of features of some open-source traffic generators

Traffic Generator	**Operating System**	**Protocols supported**	**Stochastic distribution**	**Interface**	**Operation Level**
GenSyn	Java virtual machine	TCP, UDP, IPv4	(*user model*, *responsible*)	GUI	application-level
Harpoon	FreeBSD 5.1-5.4, Linux 2.2-2.6, MacOS X 10.2-10.4, and Solaris 8-10	TCP, UDP, IPv4, IPv6	(*flow-level model, based on a input trace*)	CLI	flow-level
D-ITG	Linux, Windows, Linux Familiar, Montavista, Snapgear	IPv4, IPv6, ICMP, TCP, UDP, DCCP, SCTP	Constant, uniform, exponential, pareto, cauchy, normal, poisson, gamma, on/off, *pcap*	CLI, Script, API	packet-level
Ostinato	Linux, Windows, FreeBDS	Ethernet/802.3/LLC SNAP; VLAN (with QinQ); ARP, IPv4, IPv6, IP Tunnelling (6over4, 4over6, 4over4, 6over6);TCP, UDP, ICMPv4, ICMPv6, IGMP, MLD; HTTP, SIP, RTSP, NNTP etc... *extensible*	constant	GUI, CLI, Script, API	packet-level
Seagull	Linux, Windows	IPv4, IPv6, UDP, TCP, SCTP, SSL/TLS and SS7/TCAP. *extensible*	constant, poisson, *responsible*	CLI, API	packet-level
PackETH	Linux, MacOS, Windows	ethernet II, ethernet 802.3, 802.1q, QinQ, ARP, IPv4, IPv6, UDP, TCP, ICMP, ICMPv6, IGMP	constant	CLI, GUI	packet-level
Iperf	Windows, Linux, Android, MacOS X, FreeBSD, OpenBSD, NetBSD, VxWorks, Solaris	IPv4, IPv4, UDP, TCP, SCTP	constant	CLI	packet-level

Table 2 – Review of features of some open-source traffic generators

Traffic Generator	**Operating System**	**Protocols supported**	**Stochastic distribution**	**Interface**	**Operation Level**
Swing	Linux	IPv4, TCP, UDP, HTTP, NAPSTER, NNTP and SMTP	*capture trace*	CLI	closed-loop and multilevel
BRUTE	Linux	IPv4, IPv6	constant, poisson, trimoda	CLI	packet-level
SourcesOnOff	Linux	IPv4, TCP, UDP	on/off (Weibull, Pareto, Exponential and Gaussian)	CLI	packet-level
TG	Linux, FreeBSD, Solaris SunOS	IPv4, TCP, UDP	Constant, uniform, exponential, on/off	CLI	packet-level
Mgen	Linux(Unix), Windows	IPv4, IPv6, UDP, TCP, SINK	Constant, exponential, on/off	CLI, Script	packet-level
KUTE	Linux 2.6	UDP	constant	kernel module	packet-level
RUDE & CRUDE	Linux, Solaris SunOS, and FreeBSD	IPv4, UDP	constant	CLI	packet-level
NetSpec	Linux	IPv4,UDP, TCP	uniform, Normal, log-normal, exponential, Poisson, geometric, Pareto, gamma	Script	packet-level
Nping	Windows, Linux, Mac OS X	TCP, UDP, ICMP, IPv4, IPv6, ARP	constant	CLI	packet-level
TCPreplay	Linux	*pcap*	constant, *pcap*	CLI	packet-level (*replay engine*)
TCPivo	Linux	*pcap*	constant, *pcap*	CLI	packet-level (*replay engine*)
NetFPGA PacketGenerator	Linux	*pcap*	constant	CLI	packet-level (*hardware-based*)

Table 3 – Review of features of some open-source traffic generators

Traffic Generator	**Operating System**	**Protocols supported**	**Stochastic distribution**	**Interface**	**Operation Level**
NetFPGA Caliper	Linux	*pcap*	constant	CLI	packet-level (*hardware-based*)
NetFPGA OSNT	Linux	*pcap*	constant	CLI	packet-level (*hardware-based*)
MoonGen	Linux	IPv4, IPv6, IPsec, ICMP, UDP, TCP	constant, poisson	Script API (Lua)	packet-level (*hardware-dependent*)
Dpdk Pktgen	Linux	IPv4, IPv6, ARP, ICMP, TCP, UDP, *pcap*	constant, *pcap*	CLI, Script API (Lua)	packet-level (*hardware-dependent*)
Dpdk NFPA	Linux	*pcap*	constant, *pcap*	CLI, Web	packet-level (*hardware-dependent*)
LegoTG	Linux	[4]	[5]	CLI, Script	packet-level
LiTGen	*missingInfo*	*missingInfo*	*wifi model*	*missingInfo*	closed-loop and multilevel
gen_send/ gen_recv	Solaris, FreeBSD, AIX4.1, Linux	UDP	constant	CLI	packet-level
mxtraf	Linux	TCP, UDP, IPv4	constant	GUI, script	packet-level
Jigs Traffic Generator (JTG)	Linux	TCP, UDP, IPv4, IPv6	constant	CLI	packet-level
SURGE	Linux	TCP, IPv4	*application model*	CLI	application-level
Httperf	Linux	TCP, IPv4	*application model*	CLI	aplication-level
VoIP Traffic Generator	Linux	UDP, IPv4	*application model*	CLI	application-level

2.2 ~~Overview of N~~etwork Traffic Modeling and Realistic Traffic Generation

Along with the huge diversity of traffic generator tools, there are many traffic generation approaches, depending on the main purpose of the traffic generator. Maximum throughput traffic generators have in mind the idea of sending as much traffic through an interface as it can. But the generation of a realistic workload is a need for validation of many aspects of a new infrastructure. First, to generate a realist traffic, we need to define the complexity that our synthetic traffic must have, compared to real ones. Depending on our purpose, we may focus on one or more aspects of the traffic. For example, protocols, header customization, packet-level features (inter-departure, packet-size) and flow level features (number of flows and flow modeling).

As presented, there is a huge amount of open-source traffic generators available. Each of them with many different sets of features available. But, on the generation of realistic workload, the set of possibilities become much more restrict. On the other hand, there are many works on characterization, modeling, and simulation of different types of network workload [Botta *et al.* 2012].

As stated by Botta et al. [Botta *et al.* 2012], a synthetic network workload generation over real networks should be able to: (1) Capture real traces complexity over different scenarios; (2) Be able to custom change some specific properties generated traffic ; (3) Return measure indicators of performance experienced by the workload.

To generate realistic traffic workloads, two main approaches exist in the literature [Botta *et al.* 2012]. First, we have the trace-based generation, where replay engines generate the traffic. As examples, we have TCPreplay, TCPivo, and others. The other approach is an analytical model-based generation. In this case, the packet generation process depends on analytical and stochastic models. As examples of tools, we have Swing, D-ITG, TG, MGEN RUDE/CRUDE, Seagull, and many others. These tools control traffic features using stochastic and analytical models and/or enable header customization.

The advantage of the replay engine is its simplicity. There is no need for stochastic modeling of features. It just needs to know how to read the packet from a file, called *pcap*, and replicate it on the internet interface. Most of the traffic features such as packet sizes, inter-departure, and packet headers will be realistic. But, its major issue is the storage space required to generate traffic without replication. In fact, depending on the throughput required, just some seconds of traffic replication will cost GB's of hard disk space. If it is necessary to generate traffic for a longer time, or good *pcaps* traces are not available, analytical are necessary.

Classical models for network traffic generation were the same used in telephone traffic, such as pure Poisson or Poisson-related, like Markov and Poisson-batch [Leland *et al.*

1994]. They are able to describe the randomness of an Ethernet link but cannot capture the presence of "burstiness" in a long-term time scale, such as traffic "spikes" on long-range "ripples" [Leland *et al.* 1994]. In fact, as we can see at [Leland *et al.* 1994] the nature of the Ethernet traffic is self-similar. It has a fractal-like shape since characteristics seen in a small time scale should appear on a long-scale as well. This is most of the time referred as long-range dependence or degree of long-range dependence (LRD). One way to identify if a process is self-similar is checking its Hurst parameter, or Hurst exponent H, as a measure of the "burstiness" and LRD. In fact, a random process is self-similar and LRD if $0.5 < H < 1$ [Rongcai e Shuo 2010]. Furthermore, some later studies advocate the use of more advanced multiscaling models (multifractal), addressed by investigations that state multifractal characteristics.

Another desired characteristic is a high variability, in mathematical terms, an infinite variance. Process with such characteristic is said to be heavy-tailed [Varet 2014]. In practical terms, that means a sudden discontinuous change can always occur. Heavy tail means that a stochastic distribution is not exponentially bounded [Varet 2014]. This means that a value far from the mean does not have a negligible probability of occurrence. We can express self-similar and heavy-tailed processes using heavy-tailed stochastic distributions, such as Pareto and Weibull. As a reference for these stochastic distributions, a list in the table 4. In the last column, we indicate if the distribution is or not heavy-tailed.

We call these concepts of High variability and Self-similarity Noah and Joseph Effects [Willinger *et al.* 1997]. We can see at [Willinger *et al.* 1997] that a superposition of many ON/OFF sources (or packet trains) using ON and OFF that obey the Noah Effect (heavy-tailed probabilistic functions), also obey the Joseph effect. That means, it is self-similar and we can use to describe an Ethernet traffic. As we can see, some works on the literature on synthetic traffic uses this principle, like sourcesOnOff [Varet 2014], or have to support models like D-ITG [Botta *et al.* 2012].

An accurate replication of a workload should be able to control packet headers such as QoS fields, protocols, ports, addresses, and so on. Traffic generators provide support for these features, more frequently in a limitted way. Most offer support just common protocols, such as TCP, UDP, and IPv4. On the other hands, there are some which provide a huge variety of support and control over packet headers like PackETH [PACKETH 2015] and D-ITG. Other tools are even able to enable you to extend this feature and develop support to new protocols. For example, Ostinato and Seagull permit you to define your own customized protocol.

An important desirable feature that should be controlled is the packet size distribution since each flow may have its own packet distribution. As we can see in many works, the packet size of a trace may result in a huge impact in a trace throughput [Rongcai e Shuo 2010] [Kolahi *et al.* 2011]; small packets cause a huge overhead in the packet throughput. In the table 6 there are a survey of results found by two different works [Srivastava *et al.* 2014] [Kolahi *et al.* 2011] about the impact of different packet sizes in the throughput rate.Therefore, it is an

Table 4 – Probability density function (PDF) and Cumulative distribution function (CDF) of some random variables, and if this stochastic distribution has or not self-similarity property. Some functions used to express these distributions are defined at the table 5

Distribution	PDF Equation	CDF Equation	Parameters	Heavy-tailed
Poisson	$f[k] = \frac{e^{-\lambda}\lambda^k}{k!}$	$F[k] = \frac{\Gamma(\lfloor k+1 \rfloor, \lambda)}{\lfloor k \rfloor !}$	$\lambda > 0$ (mean, variance)	no
Binomial	$f[k] = \binom{n}{k} p^k (1-p)^{n-k}$	$F[k] = I_{1-p}(n-k, 1+k)$	$n > 0$ (trials) $p > 0$ (success)	no
Normal	$f(t) = \frac{1}{\sqrt{2\sigma^2\pi}} e^{\frac{(t-\mu)^2}{2\sigma^2}}$	$F(t) = \frac{1}{2}[1 + \mathrm{erf}(\frac{t-\mu}{\sigma\sqrt{2}})]$	μ (mean) $\sigma > 0$ (std.dev)	no
Exponential	$f(t) = \begin{cases} \lambda e^{-\lambda t}; & t \geq 0 \\ 0; & t < 0 \end{cases}$	$F(t) = 1 - e^{-\lambda t}$	$\lambda > 0$ (rate)	no
Pareto	$f(t) = \begin{cases} \frac{\alpha t_m^\alpha}{t^{\alpha+1}}; & t \geq t_m \\ 0; & t < t_m \end{cases}$	$F(t) = \begin{cases} 1 - (\frac{t_m}{t})^\alpha; & t \geq t_m \\ 0; & t < t_m \end{cases}$	$\alpha > 0$ (shape) $t_m > 0$ (scale)	yes
Cauchy	$f(t) = \frac{1}{\pi\gamma}[\frac{\gamma^2}{(t-t_0)^2+\gamma^2}]$	$F(t) = \frac{1}{\pi}\arctan(\frac{t-t_0}{\gamma}) + \frac{1}{2}$	$\gamma > 0$ (scale) $t_0 > 0$ (location)	yes
Weibull	$f(t) = \begin{cases} \frac{\alpha}{\beta^\alpha} t^{\alpha-1} e^{(t/\beta)^\alpha}; & t \geq 0 \\ 0; & t < 0 \end{cases}$	$F(t) = \begin{cases} 1 - e^{-(t/\beta)^\alpha}; & t \geq 0 \\ 0; & t < 0 \end{cases}$	$\alpha > 0$ (shape) $\beta > 0$ (scale)	yes
Gamma	$f(t) = \frac{\beta^\alpha}{\Gamma(\alpha)} t^{\alpha-1} e^{-\beta t}$	$F(t) = 1 - \frac{1}{\Gamma(\alpha)}\Gamma(\alpha, \beta x)$	$\alpha > 0$ (shape) $\beta > 0$ (rate)	no
Beta	$f(t) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)}$	$F(t) = I_x(\alpha, \beta)$	$\alpha > 0$ (shape) $\beta > 0$ (shape)	no
Log-normal	$f(t) = \frac{1}{t\sigma\sqrt{2\pi}} e^{-\frac{(\ln(x)-\mu)^2}{2\sigma^2}}$	$F(t) = \frac{1}{2} + \frac{1}{2}\mathrm{erf}[\frac{\ln(x)-\mu}{\sqrt{2}\sigma}]$	μ (location) $\sigma > 0$ (shape)	yes
Chi-squared	$f(t) = \frac{1}{2^{\frac{k}{2}}\Gamma(\frac{k}{2})} t^{\frac{k}{2}-1} e^{-\frac{t}{2}}$	$F(t) = \frac{1}{\Gamma(\frac{k}{2})}\gamma(\frac{k}{2}, \frac{x}{2})$	$k \in \mathbb{N}_{>0}$	no

important factor we must taken into account in the evaluation of new proposals, and in the implementation of network workloads generators. On packet size distribution characterization, we can find many works in the literature. For example, [Castro *et al.* 2010] analyses many packet sizes distributions of many packet traces, in many environments. Some general results are that 90% of UDP packets are smaller than 500 bytes, and most packets transmitted using TCP have 40 bytes (acknowledgment) and 1500 bytes (Maximum Transmission Unit, MTU) [Castro *et al.* 2010]. For UDP traffic, we have similar results, since mostly they are bimodal as well [Ostrowsky *et al.* 2007]. Ostrowsky et al. [Ostrowsky *et al.* 2007] found that, on UDP traces the modes of two regions are 120 and 1350 bytes, with a cut-off value of 750 bytes. They also find that roughly UDP packets constitute 20% of the total number of packets on captures.

As could be expected, router capture traces, are mostly bimodal, since most of the traffic in backbones is TCP. But, the size of the each mode may change depending on the appli-

Table 5 – Definitions of some functions used by PDFs and CDFs

Function	Definition
Regularized Incomplete beta function	$I_x(a,b) = \frac{B(x\|a,b)}{B(a,b)}$
Incomplete beta function	$B(x\|a,b) = \int_0^x t^{a-1}(1-t)^{(b-1)}\mathrm{d}t$
Beta function	$B(x\|a,b) = \int_0^1 t^{a-1}(1-t)^{(b-1)}\mathrm{d}t$
Error function	$\mathrm{erf}(x) = \frac{1}{\sqrt{\pi}}\int_x^{-x} e^{-t^2}\mathrm{d}t$
Lower incomplete Gamma function	$\gamma(s,x) = x^s\Gamma(s)e^{-x}\sum_{k=0}^{\infty}\frac{x^k}{\Gamma(s+k+1)}$

Table 6 – Two different studies evaluating the impact of packet size on the throughput. Both compare many available open-source tools on different testbeds. In all cases, small packet sizes penalize the throughput. Bigger packet sizes achieve a higher throughput.

Article and setup	Traffic Generators		
	Toll	Maximum bit-rate at small packet sizes	Maximum bit-rate at big packet sizes
Comparative study of various Traffic Generator Tools [Srivastava et al. *2014] ;* setup: Linux (Centos 6.2, Kernel version 2.6.32), Inter(R) Xeon(R) CPU with 2.96GHz, RAM of 64GB , NIC Mellanox Technologies MT25418 [ConnectXVPI PCIe 2.0 2.5GT/s - IB DDR] 10 Bbps. Protocol: TCP	PackETH	150 @(64 bytes)	1745 @(1408 bytes)
	Ostinato	135 @(64 bytes)	2850 @(1408 bytes)
	D-ITG	62 @(64 bytes)	1950 @(1408 bytes), 9808 @(1460 bytes, 12 threads)
	Iperf	*	8450 @(1460 bytes, 12 threads)
Performance Monitoring of Various Network Traffic Generators [Kolahi et al. *2011];* Inter(R) Pentium 4(R), CPU with 3.0GHz, RAM 1GB, NIC Intel Pro/100 Adapter (100Mbps), Hard Drivers Seagate Barracuda 7200 series with 20BG. Protocol:TCP	Iperf	46.0 @(128 bytes)	93.1 @(1408 bytes)
	Netperf	46.0 @(128 bytes)	89.9 @(1408 bytes)
	D-ITG	38.1 @(128 bytes)	83.1 @(1408 bytes)
	IP Traffic	61.0 @(128 bytes)	76.7 @(1408 bytes)

cation. For example, a www usage tends to have a mode close to the MTU higher if compared to an FTP capture. So, for packet workload generation, these results suggest the importance of controlling the packet size behavior for each flow [Castro *et al.* 2010].

On flow level traffic generation, some packet-level traffic generators permit the control of flow generation, mostly by manually controlling headers parameters through an API or via scripting. In terms of automatic flow configuration, an example is Harpoon [Sommers *et al.* 2004] which can to automatic configure its flows, using as input NetFlow Cisco traffic traces to automatically setting parameters. Harpoon deals with flow modeling in three different levels: file level, session level, and user level, not dealing with packet level at all. In the file level, Harpoon model two parameters: the size of files being transferred, and the time interval between consecutive file requests, called inter-file request time. The middle level is the session level, that consist of sequences files transfer between two distinct IP addresses. The session level has three components: the IP spatial distribution, the second is the inter-session start times and the third is the session duration. The last level is the user level. In Harpoon, "users" are divided on "TCP" and "UDP" users. Which conduct consecutive session using these protocols. This level has two components: the user ON time, and the number of active users. By modeling the number of users, harpoon can reproduce temporal(diurnal) traffic volumes, which is an interesting feature, once it is common on Internet traffic.

A feature that may be highly desirable for realistic traffic generation is operating in closed-loop like Swing [Vishwanath e Vahdat 2009]. This means when the changes its behavior at run time according to the observation made in real-time, with involves modification of the traffic to be generated [Botta *et al.* 2012]. These modifications involve changes on parameters of statistical distributions of inter-departure time (IDT) and packet size (PS).

Closed-loop multi-layer and application layer traffic generators also models application and user behavior, like the number of request/response exchanges per connection, response sizes, request sizes, user think time between connections, the number of RREs, RREs think time, etc [Vishwanath e Vahdat 2009].

Finally, on newly arrived internet scenarios, due its increasing complexity of networks and the rise of many new technologies, such as SDN [Kreutz *et al.* 2015] and NFV [Han *et al.* 2015]; which include the replacement of old and reliable technologies by new and not as strongly validated ones. In this sense that a point to point validation is not enough anymore, an arriving requirement is the distributed workload generator, which includes a logically centralized module such as a controller [Botta *et al.* 2012], or an orchestrator [Bartlett e Mirkovic 2015], able to control and synchronize and deploy workloads of many different hots. As examples, D-ITG API gives this possibility via daemons, and LegoTG [Bartlett e Mirkovic 2015] Framework implements a complete and easily extensible orchestrator. Due new scenarios where virtualization network functions and hardware is overcoming hardware, a logically centralized control and orchestration of synthetic workloads is a feature that easy the work [Bartlett e Mirkovic 2015].

2.3 Validation of Traffic Generator Tools

There are many possible validation technics for traffic generator tools found in the literature. Any of them have their own importance and application. Magyesi and SzabÃ³ [Molnár *et al.* 2013] presents an abstract of the main metrics validation metrics. The authors classify them in four categories: packet based metrics, flow based metrics, scaling characteristics and QoS/QoE related metrics. As suggested by other works [Vishwanath e Vahdat 2009], we state here that an synthetic traffic can be considered realistic, metrics of these four classes are close to measured on a real scenario.

Here we present a short review of each of these validation techniques. After that, we will present seven different study cases on validation of related traffic generators. They are Swing [Vishwanath e Vahdat 2009], Harpoon [Sommers *et al.* 2004], D-ITG [Botta *et al.* 2012], sourcesOnOff [Varet 2014], MoonGen [Emmerich *et al.* 2015], LegoTG [Bartlett e Mirkovic 2015] and NFPA [Csikor *et al.* 2015].

2.3.1 Packet Based Metrics

Packet based are the most simple and more used metrics in the validation of traffic generators [Molnár *et al.* 2013]. The most relevant packet based metrics are throughput [Botta *et al.* 2010] [Srivastava *et al.* 2014] [Kolahi *et al.* 2011] [Emmerich *et al.* 2015] (bytes and packets), packet size distribution [Castro *et al.* 2010] and inter packet time distribution (inter-arrival and inter-departure) [Varet 2014] [Botta *et al.* 2012].

2.3.2 Flow Based Metrics

Flow based metrics are becoming more important since newer network elements, like SDN devices, are able of execute flow-based operations [Molnár *et al.* 2013] [Kreutz *et al.* 2015]. Magyesi and SzabÃ³ [Molnár *et al.* 2013] consider the most important flow metrics the flow size distribution and the flow volume. The flow volume are proportional to the number of flow instances and flow-based device should run simultaneously. And the flow sizes defines how much time each instance will run.

2.3.3 Scaling Characteristics

Second order characteristics such as burstiness and long-range dependence are responsible for the complex nature of the internet traffic [Molnár *et al.* 2013]. Due its non-stationary nature, traditional methods fail on extract useful information [Molnár *et al.* 2013]. The first analysis made in that way were focus on the estimation of the Hurst exponent [Leland *et al.* 1994]. They demonstrated the self-similar nature of the ethernet traffic. As explained before, a self-similar traffic should an Hust exponent H, such as $0.5 < H < 1$. Over the years, wavalet based analysis have become an efficient way of reveal correlations, bursts and scaling nature of the ethernet traffic [Molnár *et al.* 2013]. Many works found on the literature have used wavelet based analysis [Vishwanath e Vahdat 2009] [Huang *et al.* 2001] [Abry e Veitch 1998].

Huang et al. [Huang *et al.* 2001] and Abry and Veitch [Abry e Veitch 1998] offers an extensible explanation of wavelet-based scaling analysis (WSA) or wavelet multi-resolution energy analysis (WMA). Here, is presented a brief summary of the main information presented by these two works, which you should refer to, for further details.

First, consider a time series $X_{0,k}$ for $k = 0, 1, ...2^n$:

$$\{X_{0,k}\} = \{X_{0,0}, X_{0,1}, ..., X_{0,2^n}\} \tag{2.1}$$

Supose then that we coaser X_0 in another time-serie X_1 with half of the original resolution, but using$\sqrt{2}$ as normalization factor:

$$X_{1,k} = \frac{1}{\sqrt{2}}(X_{0,2k} + X_{0,2k+1}) \tag{2.2}$$

If we take the differences, instead of the avarages, evaluate the so called *details*.

$$d_{1,k} = \frac{1}{\sqrt{2}}(X_{0,2k} - X_{0,2k+1}) \tag{2.3}$$

We can continue repeating this process, writing more coarse time series X_2 from X_1, until we reach X_n. Therefore, we will get a collection of *details*:

$$\{d_{j,k}\} = \{d_{1,0}, d_{1,1}, ..., d_{1,2^{n/2}}, ..., d_{n,0}\} \tag{2.4}$$

This collection of details $d_{j,k}$ are called Discrete Haar Wavelet Transform. Using the *details* we can calculate the energy function E_j, for each scale j, using:

$$E_j = \frac{1}{N_j} \sum_{k=0}^{N_j - 1} |d_{j,k}|^2; \qquad j = 1, 2, ..., n \tag{2.5}$$

were N_j is the number of coefficients at scale j. If we plot $\log(E_j)$ as a function o the scale j, we will obtain an wavelet multiresolution energy plot.

On energy wavelet multiresolution energy plots, we can capture three main different behavior, according to the scale. On **periodic time series**, the Energy values will be small. In fact, on perfectly periodic scales j, the values of the energy function E_j will be zero. So periodicity will be sensed if the value of the energy function decrease. Perfect **white noise time series** will maintain the same value of the energy function. So an approximately constant values for the energy function E_j indicates white noise behavior (wich can be represented by a Poisson process [Grigoriu 2004]). On **self-similar time series**, the energy function plot $\log(E_j)$ will grow approximately linearly with the scale j.

Some recent works suggest the use of multi-fractal models, instead of the self-similar models (also called monofractal) [Molnár *et al.* 2013] [Ostrowsky *et al.* 2007]. Since there is a lack on multiscaling analysis on the literature on validation of traffic generators, this approach will not be used in this work.

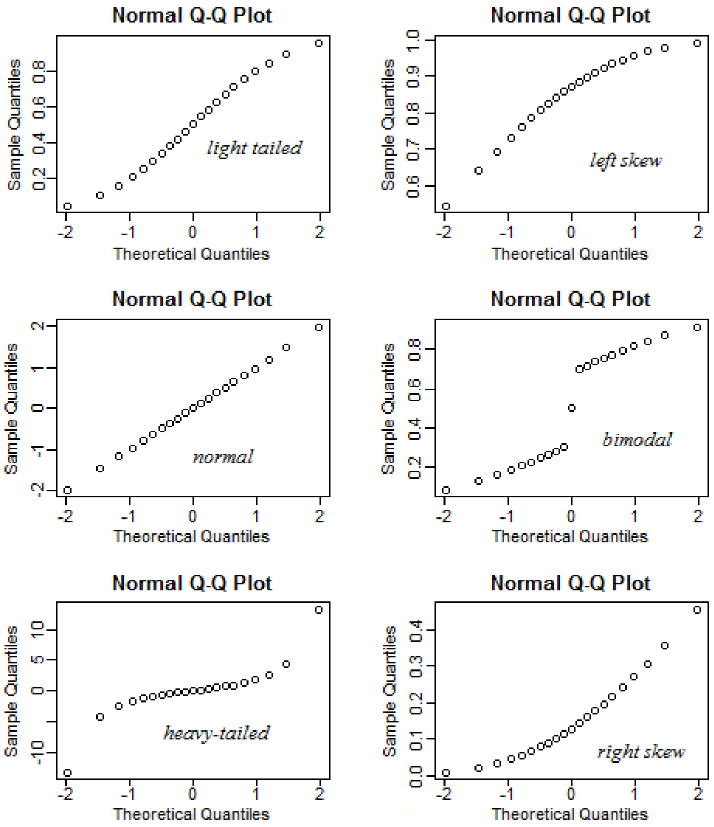


Figure 3 – How information about may be extracted from QQplots. Source: "How to interpret a QQ plot" [Online]. Available: https://stats.stackexchange.com/questions/101274/how-to-interpret-a-qq-plot [Accessed 09 June 2017]

Another way to analyze scalling characteristics is through QQplots. QQplot is a visual method to compare sample data with a specific stochastic distribution. It orders the sample data values from the smallest to largest, then plots it against the expected value given by the probability distribution function. The data sample values appear along the y-axis and the expected values along the x-axis. The more linear, the more the data is likely to be expressed by this specific stochastic distribution.

Dependendin on the type of plot, some behavios of the empirical dataset, compared to the theoretical may be observed. In the figure 3 is presented a sumary.

2.3.4 QoS/QoE Related Metrics

For the point of view of a traffic generation, is interesting that the QoS and QoE metrics present similar values to the ones found on real scenarios. As stated by Magyesi and SzabÃ³ [Molnár *et al.* 2013], important QoS/QoE metrics on validation of workload tools are: Round trip Time values (RTT), avarage queue whaiting time and queue size. Still on queue size, self-similar traffic consumes router buffers faster than Poisson traffic [Cevizci *et al.* 2006].

2.3.5 Study cases

2.3.5.1 Swing

Swing [Vishwanath e Vahdat 2009] is at the present time, one of the main references of realistic traffic generation. The authors extracted bidirectional metrics from a network link of synthetic traces. Their goals were to get realism, responsiveness, and randomness. They define realism as a trace that reflects the following characteristics of the original:

- Packet inter-arrival rate and burstiness across many time scales;
- Packet size distributions;
- Flow characteristics as arrival rate and length distributions;
- Destination IPs and port distributions.

The traffic generator uses a structural model the account interactions between many layers of the network stack. Each layer has many control variables, which is randomly generated by a stochastic process. They begin the parameterization, classifying tcpdump [Tcpdump & Libpcap] *pcap* files with the data, they are able to estimate parameters.

They validate the results using public available traffic traces, from Mawi [MAWI Working Group Traffic Archive] and CAIDA [CAIDA Center for Applied Internet Data Analysis]. On the paper, the author focuses on these validation metrics:

- Comparison of estimated parameters of the original and swing generated traces;
- Comparison of aggregate and per-application bandwidth and packets per seconds ;
- QoS metrics such as two-way delay and loss rates;
- Scaling analysis, via Energy multiresolution energy analysis.

To the vast majority of the results, both original and swing traces results were close from each other. Thus, Swing was able to match aggregate and burstiness metrics, per byte and per packet, across many time scales.

2.3.5.2 Harpoon

Harpoon [Sommers e Barford 2004] [Sommers *et al.* 2004] is a traffic generator able to generate representative traffic at IP *flow level*. It is able to generate TCP and IP with the same byte, packet, temporal and spatial characteristics measured at routers. Also, Harpoon is a self-configurable tool, since it automatically extracts parameters from network traces. It estimates some parameters from original traffic trace: file sizes, inter-connection times, source and destination IP addresses, and the number of active sessions.

As proof of concept [Sommers e Barford 2004], the authors compared statistics from original and harpoon's generated traces. The two main types of comparisons: diurnal throughput, and for stochastic variable CDF and frequency distributions. Diurnal throughput refers to the mean bandwidth variation within a day period. In a usual network, during the day the bandwidth consumption is larger, and at night smaller. Also, they compared:

- CDF of bytes transferred per 10 minutes
- CDF of packets transferred per 10 minutes
- CDF of inter-connection time
- CDF of file size
- CDF of flow arrivals per 1 hour
- Destination IP address frequency

At the end, they showed the differences on throughput evaluation of a Cisco 6509 switch/router using Harpoon and a constant rate traffic generator. Harpoon was proven able to give close CDFs, frequency and diurnal throughput plots compared to the original traces. Also, the results demonstrated that Harpoon provide a more variable load on routers, compared to a constant rate traffic. It indicates the importance of using realistic traffic traces on the evaluation of equipment and technologies.

2.3.5.3 D-ITG

D-ITG [Botta *et al.* 2012] is a network traffic generator, with many configurable features. The tool provides a platform that meets many emerging requirements for a realistic traffic generation. For example, multi-platform, support of many protocols, distributed operation, sending/receiving flow scalability, generation models, and analytical model based generation high bit/packet rate. You can see different analytical and models and protocols supported by D-ITG at table 1.

We will focus on the evaluation of realism on analytical model parameterization. It is a synthetic replication of a LAN party of eight players of Age of Mythology [6]. They have captured traffic flows during the party. Then, they modeled its packet size and inter-packet time distributions. They show that the synthetic traffic and the analytical model have similar curves of packet size and inter-packet time, thus it can approximate the empirical data. Also, the mean and the standard deviation of the bit rate of the empirical and synthetic data are similar.

2.3.5.4 sourcesOnOff

Varet et al. [Varet 2014] creates an application implemented in C, called SourcesOnOff. It models the activity interval of packet trains using probabilistic distributions. To choose the best stochastic models, the authors have captured traffic traces using TCPdump. Then the developed tool is able to figure out what distribution(Weibull, Pareto, Exponential, Gaussian, etc) fits better the original traffic traces. It uses the Bayesian Information Criterion (BIC) for distance assessment. It tests the smaller BIC for each distribution, and selects it as the best choice. It ensures good correlation between the original and generated traces and self-similarity.

The validation methods used on sourcesOnOff are:

- Visual comparison between On time and Off time of the original trace and the stochastic fitting;
- QQplots, which aim to evaluate the correlation between inter-trains duration of real and generated traffic;
- Measurement of Autocorrelation[7] of the measured throughput of the real and synthetic traffic;
- Hurst exponent computation of the real and the synthetic trace;

[6] https://www.ageofempires.com/games/aom/

[7] The autocorrelation functions measures the correlation between data samples y_t and y_{t+k}, where $k = 0, ..., K$, and the data sample $\{y\}$ is generated by an stochastic process.

According to [Box *et al.* 1994], the autocorrelation for a lag k is:

$$r_k = \frac{c_k}{c_0} \tag{2.6}$$

where

$$c_k = \frac{1}{T-1} \sum_{t=1}^{T-k} (y_t - \bar{y})(y_{t+k} - \bar{y}) \tag{2.7}$$

and c_0 is the sample variance of the time series.

The results pointed to a good stochastic fitting, but better for On time values. On the other hand, the correlation value of the QQplot was bigger on the Off time values (99.8% versus 97.9%). In the real and synthetic traces, the autocorrelation of the throughput remained between an upper limit of 5%. Finally, the ratio between the evaluated Hurst exponent always remained smaller than 12%.

2.3.5.5 MoonGen

MoonGen [Emmerich *et al.* 2015] is a high-speed scriptable paper capable of sature 10 GnE link with 64 bytes packets, using a single CPU core. The authors have built it over DPDK and LuaJit, enabling the user to have high flexibility on the crafting of each packet, through Lua scripts. It has multi-core support and runs on commodity server hardware. It is able to test latency with sub-microsecond precision and accuracy, using hardware timestamping of modern NICs cards. The Lua scripting API enable the implementation and high customization along with high-speed. This includes the controlling of packet sizes and control of inter-departure times.

The authors evaluated this traffic generator focused on throughput metrics, rather than others. Also, they have small packet sizes (64 bytes to 256), since the per-packet costs dominate. In their work, they were able to surpass 15 Gbit/s with an XL710 40 GbE NIC. Also, they achieve throughput values close to the line rate with packets of 128 bytes, and 2 CPU cores.

2.3.5.6 LegoTG

Bartlett et al. [Bartlett e Mirkovic 2015] implements a modular framework for composing custom traffic generation, called LegoTG. As argued by the authors (and by this present work), automation of many aspects of traffic generation is a desirable feature. The process of how to generate a proper background traffic may become a research by itself. Traffic generators available today offer a single model and a restricted set of features into a single code base, makes customization difficult.

The main purpose of their experiment is to show how LegoTG is able to generate background traffic, in a simple way. Also, it shows how a realistic background traffic can influence on research conclusions. The experiment chosen is one of the use cases proposed for Swing [Vishwanath e Vahdat 2008], and its evaluate the error on bandwidth estimation of different measurement tools. It shows that LegoTG is able eble to provide easy and custrom traffic generation.

3 System architecture and Methods

In this chapter, we will present our tool which aims to fill the gaps addressed in the chapter 1. It is called SIMITAR (SnIffing, ModellIng and TrAffic geneRation) and aims to generate a platform-agnostic traffic: it can use as traffic generator engine almost any traffic generator tools and programming API. It just requires a programming or command line interface. SIMITAR creates a model in XML based on actual traces (*pcaps* or real-time captures), we call Compact Trace Descriptor (CTD).

We abstract its whole operation cycle in the figure 4. Our tool, from live captures or *pcap* file collects raw data from Ethernet traffic. It then breaks these data into different flows and uses this data to generate a set of parameters for our traffic model, using a set of algorithms. Finally, SIMITAR provides these parameters to a traffic generator engine and controls its packets injection.

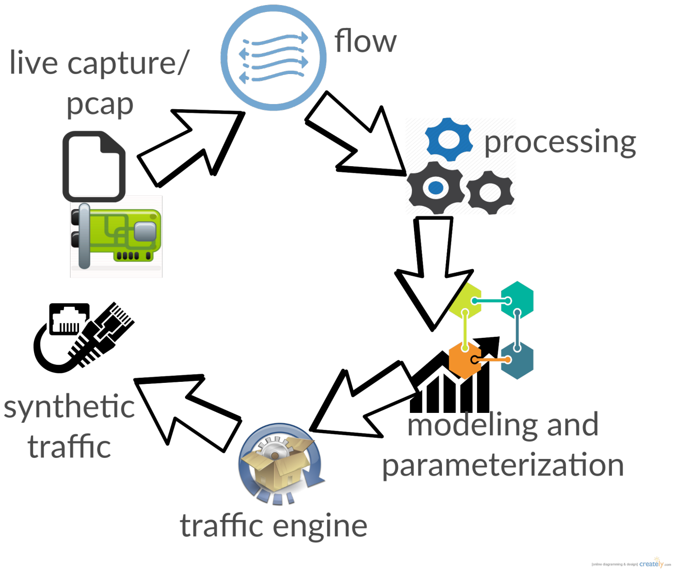


Figure 4 – This figure represents an operation cycle of SIMITAR, emphasizing each main step: sniffing, flow classification, data storing, data processing and fitting, model parameterization, and synthetic traffic generation.

3.1 SIMITAR Architecture

To meet the requirements presented in the chapter 1, we will define our solution, and how it works. SIMITAR architecture is presented in the figure 5. It is composed of four components: a *Sniffer*, a *SQLite database*, a *TraceAnalyzer*, a *FlowGenerator*; and a *Network Traffic Generator*, as subsystem. We describe each part below.

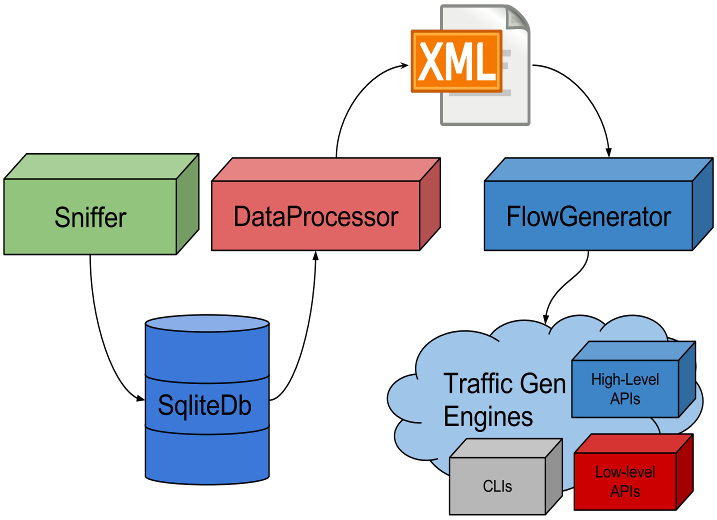


Figure 5 – Architecture of SIMITAR

3.1.1 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 [Kreutz *et al.* 2015], through header fields matches. It uses:

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

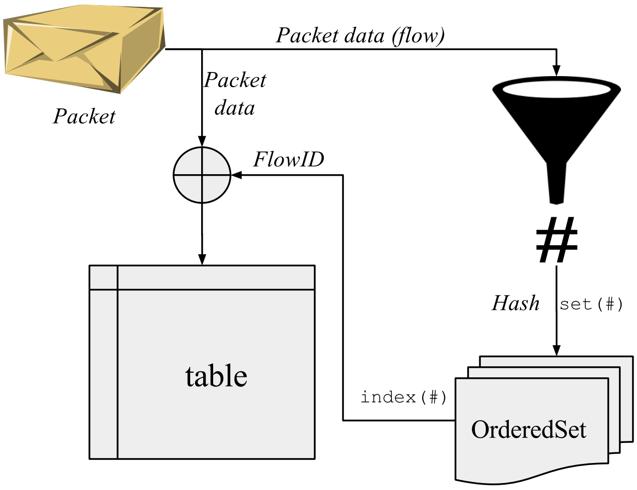


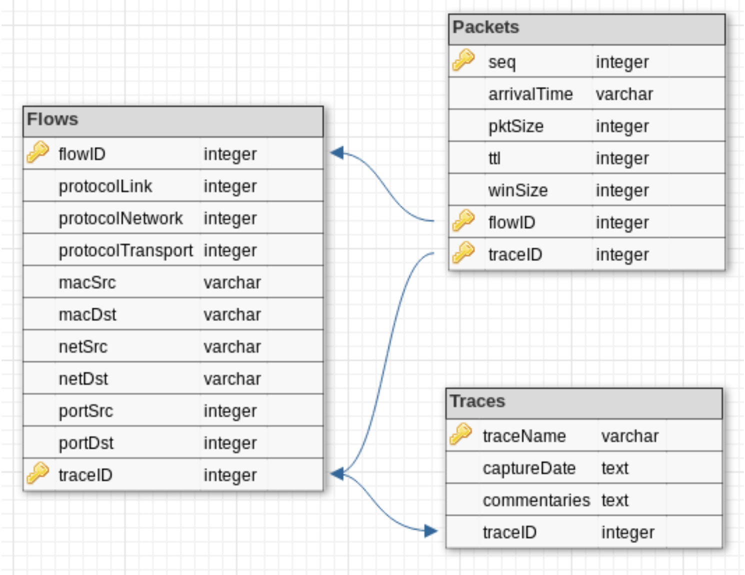
Figure 6 – SIMITAR's sniffer hash-based flow classification

We implemented the first version of this component in Shell Script (Bash). Tsahrk[1] was used to extract header fields, and Awk to match the flows, and Sed/Awk to create the SQLite queries. This version was too slow to operate in real time on Ethernet interfaces. On the other hand, this approach was fast to implement and enable the implementation of the other components. The second and current version was made using Python. This version used Pyshark [2] as sniffer library.

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

As future improvements for this component, we propose a more efficient implementation in C++ and data visualization for the collected data. In that way, we may optimize the packet processing. We discuss this in deeper details in the chapter **??**.

3.1.2 SQLite database



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

Figure 7 – SIMITAR's SQLite database relational model

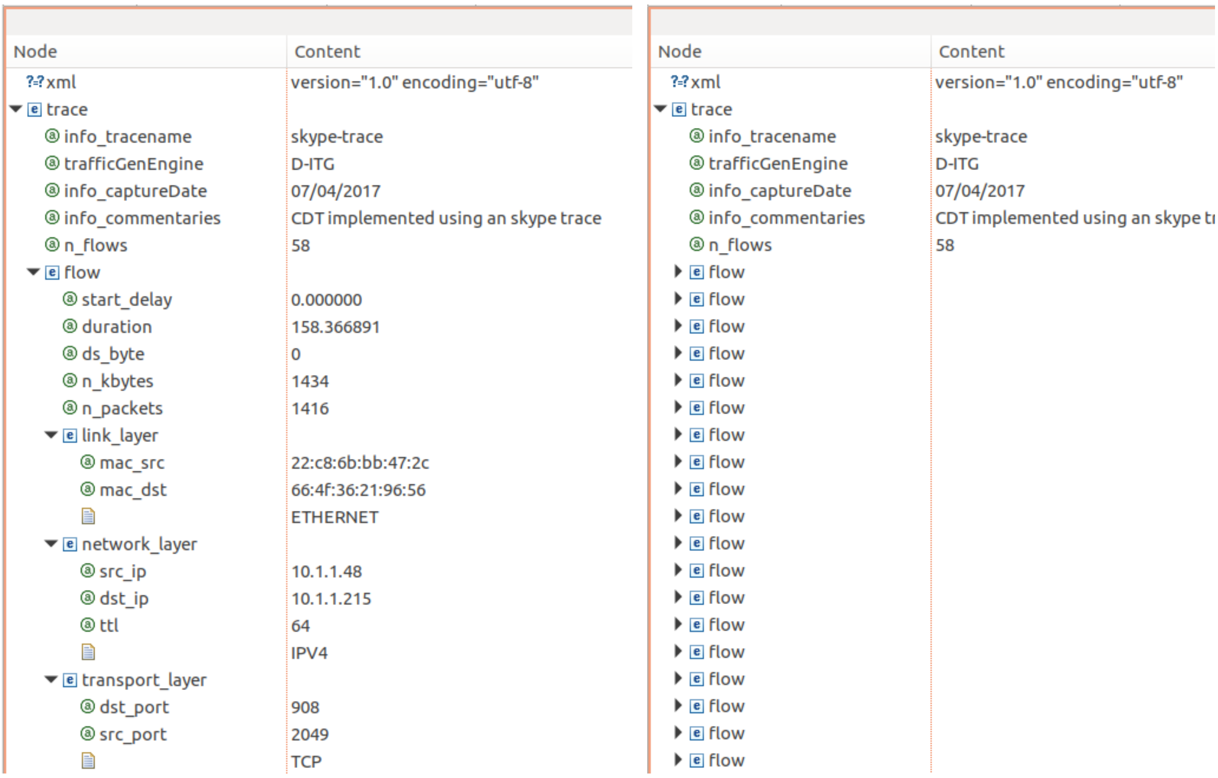
The database stores the collected raw data from the traces for further analysis. The *Sniffer* and the *TraceAnalyzer* components uses the database. We choose an SQLite database, because according to its specifications[4], it fits well our purposes. It is simple and well-suitable for an amount of data smaller than terabytes. In the figure 7 we present the relational model

[1] https://www.wireshark.org/docs/man-pages/tshark.html

[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



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

(a)

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 t
n_flows	58
flow	
flow	
flow	
flow	
flow	
flow	
flow	
flow	
flow	
flow	
flow	
flow	
flow	
flow	
flow	
flow	
flow	
flow	

(b)

Figure 8 – 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.

of our database, which stores a set of features extracted from packets, along with the flowID calculated by the sniffer component.

3.1.3 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 8 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.

3.1.3.1 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.

3.1.3.2 Inter Packet Times

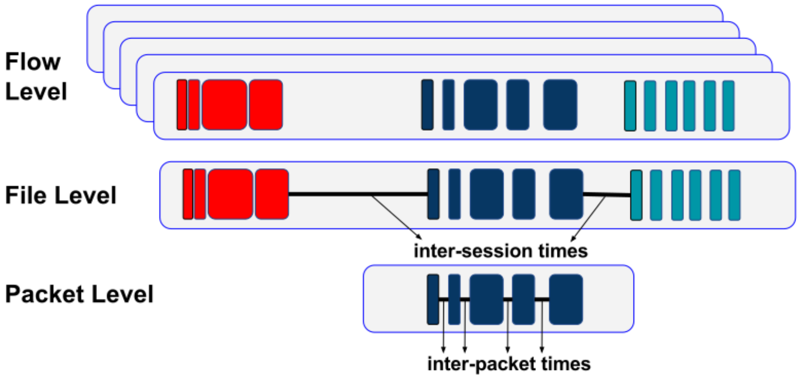


Figure 9 – The schema of the modified version of the Harpoon algorithm we adopt on SCIMITAR.

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 [Sommers *et al.* 2004] and [Sommers e Barford 2004]. Here, we will explain our simplified version, which is illustrated at figure 9.

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 [Sommers *et al.* 2004]. 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 [Vishwanath e Vahdat 2009] 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 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 [Varet 2014] [Yang 2005]. We will enter into deeper details on this methodology on the chapter **??**. The methodology of selection is presented in the figure 10, and all constants and modes of operation can be changed by command line options.

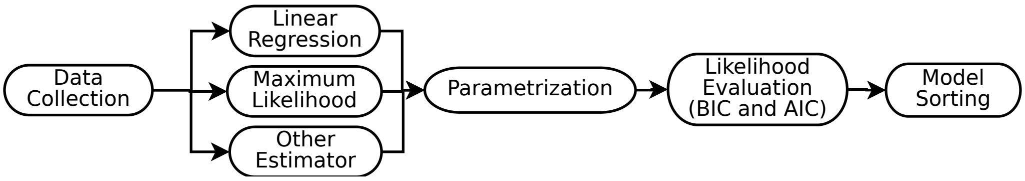


Figure 10 – Diagram of parameterization and model selection for inter-packet times and inter-file times.

[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`

3.1.3.3 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 [Castro *et al.* 2010] [Varet 2014] [Ostrowsky *et al.* 2007], 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 [Ostrowsky *et al.* 2007].

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 [Castro *et al.* 2010] [Ostrowsky *et al.* 2007], 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.

3.1.3.4 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. The traffic models for inter-packet times are grouped by the tag `inter_packet_times`, and the packet trains by the tag `session_times`. All the times are in seconds, and `"inf"` represents infinity. The protocol of each layer is stored as data by each tag.

```xml
<flow start_delay="0.144400" duration="317.744333" ds_byte="0" n_kbytes="40"
 ↪  n_packets="344">
    <link_layer mac_src="64:1c:67:69:51:bb"
     ↪  mac_dst="70:62:b8:9b:3e:d1">ETHERNET</link_layer>
    <network_layer src_ip="192.168.1.1" dst_ip="192.168.1.2"
     ↪  ttl="64">IPV4</network_layer>
    <transport_layer dst_port="2128" src_port="53">UDP</transport_layer>
    <application_layer>DNS</application_layer>
    <inter_packet_times>
        <stochastic_model name="pareto-ml" aic="-1165.310696" bic="-1157.646931"
         ↪  param1="0.405085202535192" param2="0.002272655895996"/>
        <stochastic_model name="pareto-lr" aic="-454.049749" bic="-446.385984"
         ↪  param1="0.061065000000000" param2="0.002272655895996"/>
        <stochastic_model name="weibull" aic="-246.882037" bic="-239.218273"
         ↪  param1="0.120355000000000" param2="0.001629000000000"/>
        <stochastic_model name="exponential-me" aic="486.370061" bic="494.033826"
         ↪  param1="1.340057495455104" param2="0.000000000000000"/>
        <stochastic_model name="normal" aic="1629.370900" bic="1637.034665"
         ↪  param1="0.746236637899171" param2="2.626808289821357"/>
```

```
            <stochastic_model name="exponential-lr" aic="3166.816047" bic="3174.479812"
             ↪   param1="0.009752000000000" param2="0.000000000000000"/>
            <stochastic_model name="cauchy" aic="31737.418442" bic="31745.082207"
             ↪   param1="0.000000000000194" param2="-3152.827055696396656"/>
            <stochastic_model name="constant" aic="inf" bic="inf"
             ↪   param1="0.746236637899171" param2="0.000000000000000"/>
        </inter_packet_times>
        <session_times on_times="29.22199798,73.40390396,151.84077454"
         ↪   off_times="30.85738373,32.42027283" n_packets="19,103,222"
         ↪   n_bytes="2272,12399,26689"/>
        <packet_sizes n_packets="344" n_kbytes="40">
            <ps_mode1 n_packets="344" n_kbytes="40">
                <stochastic_model name="constant" aic="inf" bic="inf"
                 ↪   param1="120.232558" param2="0.000000"/>
                <stochastic_model name="normal" aic="2926.106952" bic="2933.788235"
                 ↪   param1="120.232558" param2="16.941453"/>
                <stochastic_model name="exponential-me" aic="3987.126362"
                 ↪   bic="3994.807645" param1="0.008317" param2="0.000000"/>
            </ps_mode1>
            <ps_mode2 n_packets="0" n_kbytes="0">
                <stochastic_model name="no-model-selected" aic="inf" bic="inf"
                 ↪   param1="0.000000" param2="0.000000"/>
            </ps_mode2>
        </packet_sizes>
    </flow>
```

3.1.4 Flow Generator

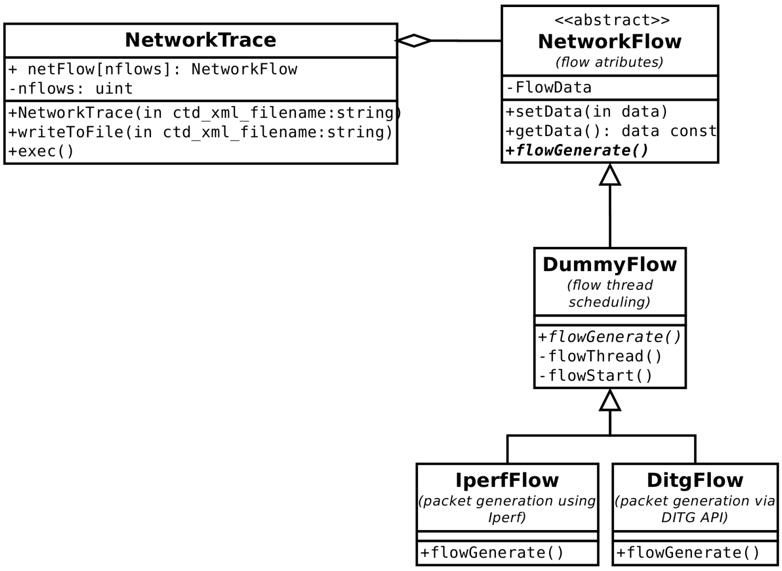


Figure 11 – Class hierarchy of NetworkTrace and NetworkFlow, which enables the abstraction of the traffic generation model of the traffic generation engine.

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++

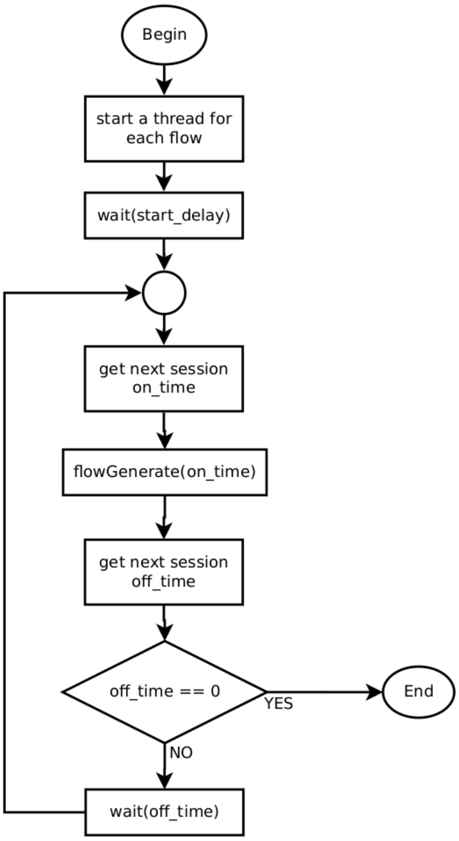


Figure 12 – Simplified-harpoon emission algorithm

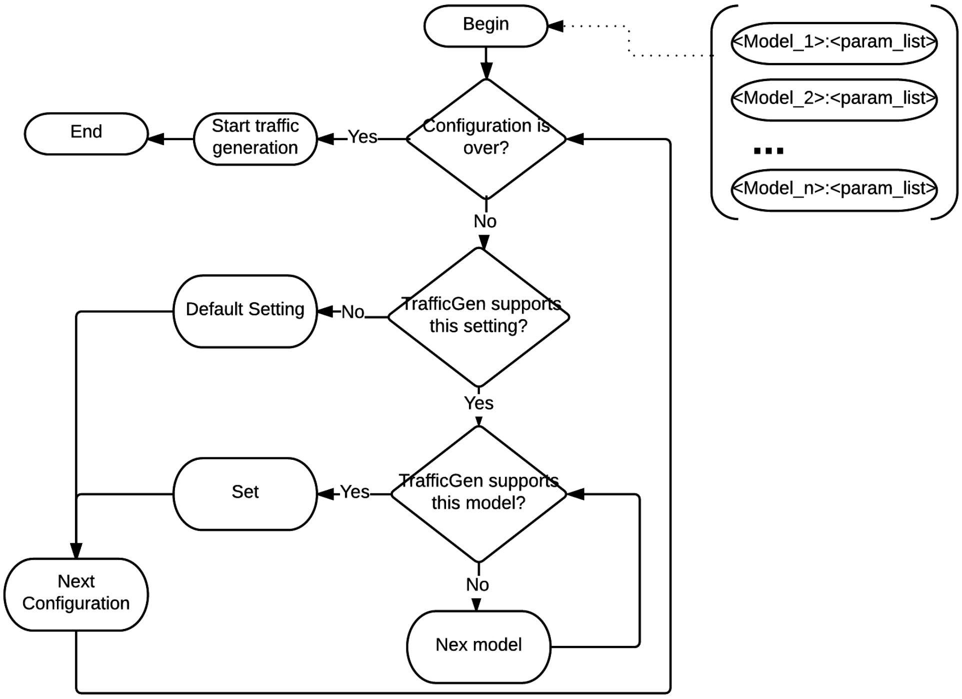


Figure 13 – Traffic engine configuration model

API, and it must follow the calss hierarcy as presented in the figure 11. 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 2 [9]. At the flow-level, SIMITAR controls each flow using the algorithm at figure 12. This algorithm handles our model defined at the figure 9. 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. 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 9. 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 13. Down below we present a simple code of how the D-ITG API can be used to generate packet-level traffic. A more complex configuration is possible, but it serves to illustrate the procedure. We call this concept *Flow Generation Programming*. Its API documentation is available at on D-ITG website [10].

```
// This implementation is just a simplified version to illustrate the procedure.
void DitgFlow::flowGenerate(const counter& flowId, const time_sec& onTime, const uint&
 ↪  npackets, const uint& nbytes,  const string& netInterface)
{
    // create command to generate the traffic
    std::string strCommand;
    std::string localhost = getNetworkSrcAddr();
    strCommand += " -t " + std::to_string(onTime);
    strCommand += " -k " + std::to_string(nbytes / 1024);
    strCommand += " -a " + getNetworkDstAddr();

    // configure protocol
    if (this->getTransportProtocol() == PROTOCOL__TCP)
        strCommand += " -T TCP -D ";
    else if (this->getTransportProtocol() == PROTOCOL__UDP)
```

[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 11). 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 9, 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.

[10] http://www.grid.unina.it/software/ITG/manual/index.html#SECTION00047000000000000000

```
        strCommand += " -T UDP ";
    else if (this->getTransportProtocol() == PROTOCOL__ICMP)
        strCommand += " -T ICMP ";

    //configure inter-packet time model, just Weibull or Constant
    StochasticModelFit idtModel;
    for(uint i = 0;;i++)
    {
        idtModel = this->getInterDepertureTimeModel(i);
        if(idtModel.modelName() == WEIBULL)
        {
            strCommand += " -W " + std::to_string(idtModel.param1()) + " " +
             ↪   std::to_string(idtModel.param2());
            break;
        }
        else if ( idtModel.modelName() == CONSTANT)
        {
            strCommand += " -C " + std::to_string(nbytes/(1024*onTime));
            break;
        }
    }

    // it uses C strings as arguments
    // it is not blocking, so it must block until finishes
    int rc = DITGsend(localhost.c_str(), command.c_str()); // D-ITG API
    usleep(onTime*10e6); // D-ITG uses miliseconds as time unity
    if (rc != 0)
    {
        PLOG_ERROR << "DITGsend() return value was" << rc ; // our log macro for erros
        exit(EXIT_FAILURE);
    }
}
```

3.1.5 Network Traffic Generator

A network traffic generator software that should provide its API or script interface for the *FlowGenerator* component. With this tool, the user must be eble to send packets/traffic and controll atributes, such as sending time, bandwidht, number of packets, protocols, and so on. That means, any available parameter form the compact trace descriptor. If it does not have an API but an CLI, the user may have to use functions such as `fork()` or `popen()`.

3.2 Usage and Use Cases

Each SIMITAR component works as a different application:

- `sniffer-cli.py`;
- `trace.db` (sqlite3 database);

- `trace-analyzer`.
- `simitar-gen` (traffic generator).

When it working as a sniffer, it stores data from the captured traffic in the sqlite3 database. It may work over a *pcap* file or an Ethernet interface. Working as a trace a trace-analyzer, it receives the name of the trace as stored in the database, and creates an XML compact trace descriptor. The constants used in the modeling may be changed and provided as command line argument. If not, the default values will be used. As a traffic generator, it may work as a client or a server. Working as a server is necessary for closed-loop engines; tools that require establishing a connection before generating the traffic, such as Iperf and D-ITG. It will just work passively. Working as a client is acting as a traffic emitter. Open loop packet-crafter tools such as *libtins* does not require server operation the send the traffic. In the case of closed-loop tools, the destination IP addresses must be explicitly given in the command line by the options `-dst-list-ip` or `-dst-ip`. Down below, we present simple examples of commands used to activate each operation mode.

```
# @ SIMITAR/, load enviroment variables
source data/config/simitar-workspace-config.sh

# @ SIMITAR/sniffer/, execute to sniffer the eth0 interface, and create a trace entry
 ↪  called "intrig" in the database
./sniffer-cli.py new intrig live eth0

# @ SIMITAR/sniffer/, execute this command to list all traces recorded in the database
./sniffer-cli.py list

# @ SIMITAR/trace-analyzer/, execute this command to create two Compact Trace
 ↪  Descriptors, called intrig.ms.xml and intrig.sec.xml. The first is parameterized
 ↪  using milisecons, and de second, uses seconds as time unity.
./trace-analyzer --trace intrig

# @ SIMITAR/simitar-gen/, execute these commands to generate traffic using the
 ↪  intrig.sec.xml compact trace descriptor. It is stored at the directory
 ↪  "../data/xml/".
# Libtins
./simitar-gen --tool tins --mode client --ether eth0 --xml ../data/xml/intrig.sec.xml
# Iperf
./simitar-gen --tool iperf --mode client --ether eth0 --xml ../data/xml/intrig.sec.xml
 ↪  --dst-ip 10.0.0.2
./simitar-gen --tool iperf --mode server --ether eth0 --xml ../data/xml/intrig.sec.xml
```

Bibliography

ABRY, P.; VEITCH, D. Wavelet analysis of long-range-dependent traffic. *IEEE Transactions on Information Theory*, v. 44, n. 1, p. 2–15, Jan 1998. ISSN 0018-9448.

BARFORD, P.; CROVELLA, M. Generating representative web workloads for network and server performance evaluation. *SIGMETRICS Perform. Eval. Rev.*, ACM, New York, NY, USA, v. 26, n. 1, p. 151–160, jun. 1998. ISSN 0163-5999. Disponível em: <http://doi.acm.org/10.1145/277858.277897>.

BARTLETT, G.; MIRKOVIC, J. Expressing different traffic models using the legotg framework. In: *2015 IEEE 35th International Conference on Distributed Computing Systems Workshops*. [S.l.: s.n.], 2015. p. 56–63. ISSN 1545-0678.

BOTTA, A.; DAINOTTI, A.; PESCAPE, A. Do you trust your software-based traffic generator? *IEEE Communications Magazine*, v. 48, n. 9, p. 158–165, Sept 2010. ISSN 0163-6804.

BOTTA, A.; DAINOTTI, A.; PESCAPé, A. A tool for the generation of realistic network workload for emerging networking scenarios. *Computer Networks*, v. 56, n. 15, p. 3531 – 3547, 2012. ISSN 1389-1286. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1389128612000928>.

BOX, G. E. P.; JENKINS, G. M.; REINSEL, G. C. *Time Series Analysis: Forecasting and Control*. 3. ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

CAI, Y.; LIU, Y.; GONG, W.; WOLF, T. 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*. [S.l.: s.n.], 2009. p. 7068–7073. ISSN 0191-2216.

CAIDA Center for Applied Internet Data Analysis. http://www.caida.org/home/. [Online; accessed January 11th, 2017].

CASTRO, E.; KUMAR, A.; ALENCAR, M. S.; E.FONSECA, I. A packet distribution traffic model for computer networks. In: *Proceedings of the International Telecommunications Symposium – ITS2010*. [S.l.: s.n.], 2010.

CEVIZCI, I.; EROL, M.; OKTUG, S. F. Analysis of multi-player online game traffic based on self-similarity. In: *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games*. New York, NY, USA: ACM, 2006. (NetGames '06). ISBN 1-59593-589-4. Disponível em: <http://doi.acm.org/10.1145/1230040.1230093>.

CSIKOR, L.; SZALAY, M.; SONKOLY, B.; TOKA, L. Nfpa: Network function performance analyzer. In: *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*. [S.l.: s.n.], 2015. p. 15–17.

DPDK – Data Plane Development Kit. 2016. http://dpdk.org/. [Online; accessed May 14th, 2016].

EMMERICH, P.; GALLENMüLLER, S.; RAUMER, D.; WOHLFART, F.; CARLE, G. Moongen: A scriptable high-speed packet generator. In: *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*. New York, NY, USA: ACM, 2015. (IMC '15), p. 275–287. ISBN 978-1-4503-3848-6. Disponível em: <http://doi.acm.org/10.1145/2815675.2815692>.

FIELD, A. J.; HARDER, U.; HARRISON, P. G. Measurement and modelling of self-similar traffic in computer networks. *IEE Proceedings - Communications*, v. 151, n. 4, p. 355–363, Aug 2004. ISSN 1350-2425.

GRIGORIU, M. Dynamic systems with poisson white noise. *Nonlinear Dynamics*, v. 36, n. 2, p. 255–266, 2004. ISSN 1573-269X. Disponível em: <http://dx.doi.org/10.1023/B:NODY.0000045518.13177.3c>.

HAN, B.; GOPALAKRISHNAN, V.; JI, L.; LEE, S. Network function virtualization: Challenges and opportunities for innovations. *Communications Magazine, IEEE*, v. 53, n. 2, p. 90–97, Feb 2015. ISSN 0163-6804.

HUANG, P.; FELDMANN, A.; WILLINGER, W.; ARCHIVES, T. P. S. U. C. A non-intrusive, wavelet-based approach to detecting network performance problems. unknown, 2001. Disponível em: <http://citeseer.ist.psu.edu/453711.html>.

KOLAHI, S. S.; NARAYAN, S.; NGUYEN, D. D. T.; SUNARTO, Y. Performance monitoring of various network traffic generators. In: *Computer Modelling and Simulation (UKSim), 2011 UkSim 13th International Conference on*. [S.l.: s.n.], 2011. p. 501–506.

KREUTZ, D.; RAMOS, F.; VERISSIMO, P. E.; ROTHENBERG, C. E.; AZODOLMOLKY, S.; UHLIG, S. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, v. 103, n. 1, p. 14–76, Jan 2015. ISSN 0018-9219.

KUSHIDA, T.; SHIBATA, Y. Empirical study of inter-arrival packet times and packet losses. In: *Proceedings 22nd International Conference on Distributed Computing Systems Workshops*. [S.l.: s.n.], 2002. p. 233–238.

LELAND, W. E.; TAQQU, M. S.; WILLINGER, W.; WILSON, D. V. On the self-similar nature of ethernet traffic (extended version). *IEEE/ACM Transactions on Networking*, v. 2, n. 1, p. 1–15, Feb 1994. ISSN 1063-6692.

LIBTINS: packet crafting and sniffing library. http://libtins.github.io/. [Online; accessed May 30th, 2017].

MAWI Working Group Traffic Archive. http://mawi.wide.ad.jp/mawi/. [Online; accessed January 11th, 2017].

MOLNáR, S.; MEGYESI, P.; SZABó, G. How to validate traffic generators? In: *2013 IEEE International Conference on Communications Workshops (ICC)*. [S.l.: s.n.], 2013. p. 1340–1344. ISSN 2164-7038.

NETFPGA. https://github.com/NetFPGA/netfpga/wiki/PacketGenerator. [Online; accessed December 12th, 2016].

OSNT Traffic Generator. https://github.com/NetFPGA/OSNT-Public/wiki/OSNT-Traffic-Generator. [Online; accessed December 12th, 2016].

OSTINATO Network Traffic Generator and Analyzer. 2016. http://ostinato.org/. [Online; accessed May 14th, 2016].

OSTROWSKY, L. O.; FONSECA, N. L. S. da; MELO, C. A. V. A traffic model for udp flows. In: *2007 IEEE International Conference on Communications*. [S.l.: s.n.], 2007. p. 217–222. ISSN 1550-3607.

PACKETH. 2015. http://packeth.sourceforge.net/packeth/Home.html. [Online; accessed May 14th, 2016].

PRECISETRAFGEN. https://github.com/NetFPGA/netfpga/wiki/PreciseTrafGen. [Online; accessed December 12th, 2016].

RONGCAI, Z.; SHUO, Z. Network traffic generation: A combination of stochastic and self-similar. In: *Advanced Computer Control (ICACC), 2010 2nd International Conference on*. [S.l.: s.n.], 2010. v. 2, p. 171–175.

SOCKETS. https://www.gnu.org/software/libc/manual/html_node/Sockets.html. [Online; accessed May 30th, 2017].

SOMMERS, J.; BARFORD, P. Self-configuring network traffic generation. In: *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*. New York, NY, USA: ACM, 2004. (IMC '04), p. 68–81. ISBN 1-58113-821-0. Disponível em: <http://doi.acm.org/10.1145/1028788.1028798>.

SOMMERS, J.; KIM, H.; BARFORD, P. Harpoon: A flow-level traffic generator for router and network tests. *SIGMETRICS Perform. Eval. Rev.*, ACM, New York, NY, USA, v. 32, n. 1, p. 392–392, jun. 2004. ISSN 0163-5999. Disponível em: <http://doi.acm.org/10.1145/1012888.1005733>.

SRIVASTAVA, S.; ANMULWAR, S.; SAPKAL, A. M.; BATRA, T.; GUPTA, A. K.; KUMAR, V. Comparative study of various traffic generator tools. In: *Engineering and Computational Sciences (RAECS), 2014 Recent Advances in*. [S.l.: s.n.], 2014. p. 1–6.

TCPDUMP & Libpcap. http://www.tcpdump.org/. [Online; accessed May 14th, 2016].

TCPDUMP/LIBPCAP public repository. http://www.tcpdump.org/. [Online; accessed May 30th, 2017].

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

THE OpenDayLight Platform. 2017. https://www.opendaylight.org/. [Online; accessed May 29th, 2017].

TRAFFIC Generator. 2011. http://www.postel.org/tg/. [Online; accessed May 14th, 2016].

VARET, N. L. A. Realistic network traffic profile generation: Theory and practice. *Computer and Information Science*, v. 7, n. 2, 2014. ISSN 1913-8989.

VISHWANATH, K. V.; VAHDAT, A. Evaluating distributed systems: Does background traffic matter? In: *USENIX 2008 Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2008. (ATC'08), p. 227–240. Disponível em: <http://dl.acm.org.ez88.periodicos.capes.gov.br/citation.cfm?id=1404014.1404031>.

VISHWANATH, K. V.; VAHDAT, A. Swing: Realistic and responsive network traffic generation. *IEEE/ACM Transactions on Networking*, v. 17, n. 3, p. 712–725, June 2009. ISSN 1063-6692.

WILLINGER, W.; TAQQU, M. S.; SHERMAN, R.; WILSON, D. V. Self-similarity through high-variability: statistical analysis of ethernet lan traffic at the source level. *IEEE/ACM Transactions on Networking*, v. 5, n. 1, p. 71–86, Feb 1997. ISSN 1063-6692.

WU, W.; HUANG, N.; ZHANG, Y. A new hybrid traffic generation model for tactical internet reliability test. In: *Reliability and Maintainability Symposium (RAMS), 2015 Annual*. [S.l.: s.n.], 2015. p. 1–6.

YANG, Y. Can the strengths of aic and bic be shared? a conflict between model indentification and regression estimation. *Biometrika*, v. 92, n. 4, p. 937, 2005. Disponível em: <+http://dx.doi.org/10.1093/biomet/92.4.937>.

Appendix