UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Engenharia Elétrica e de Computação

Anderson dos Santos Paschoalon

Generation of Synthetic and Realistic Network Workload for Benchmarking and Tests of Emerging Techinologies

Geração de Trafego de Rede Sintético e Realistico para Benchmarking e testes de Tecnologias Emergentes

CAMPINAS
2017

Anderson dos Santos Paschoalon

Generation of Synthetic and Realistic Network Workload for Benchmarking and Tests of Emerging Techinologies

Geração de Trafego de Rede Sintético e Realistico para Benchmarking e testes de Tecnologias Emergentes

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:
"Generation of Synthetic and Realistic Network Workload for Benchmarking and Tests of Emerging Techinologies"
"Geração de Trafego de Rede Sintético e Realistico para Benchmarking e testes de Tecnologias Emergentes"

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.

Mauris malesuada egestas enim, ac interdum justo feugiat quis. Etiam ullamcorper volutpat ex, at laoreet mauris vehicula eu. Integer dignissim egestas purus, sit amet feugiat orci fringilla vel. Curabitur tincidunt ac massa non molestie. Suspendisse potenti. In molestie a tortor ac pellentesque. Cras a risus et tellus hendrerit pretium ut quis arcu.

Aliquam vehicula erat eu odio mollis, ut vehicula mauris laoreet. Cras volutpat ante sed orci bibendum maximus. Pellentesque felis elit, laoreet vitae bibendum in, malesuada ut nulla. Vestibulum pharetra faucibus blandit. Morbi tristique at leo ut ultrices. Etiam sollicitudin justo lacus, non rutrum mi imperdiet non. Cras nec mauris odio. Quisque eleifend purus interdum nulla suscipit lobortis.

Proin vel nulla scelerisque nunc vehicula tempor. Curabitur non arcu ante. Etiam eget mattis tellus. Aenean molestie cursus ex, in pharetra eros hendrerit nec. In cursus egestas interdum. Ut scelerisque fringilla maximus. In facilisis vel dui vitae hendrerit. Integer a orci vel lectus lobortis fringilla. Phasellus eleifend commodo justo, cursus vehicula lorem. Donec a libero ultrices risus tempus condimentum imperdiet eget justo.

Acknowledgements

Mauris malesuada egestas enim, ac interdum justo feugiat quis. Etiam ullamcorper volutpat ex, at laoreet mauris vehicula eu. Integer dignissim egestas purus, sit amet feugiat orci fringilla vel. Curabitur tincidunt ac massa non molestie. Suspendisse potenti. In molestie a tortor ac pellentesque. Cras a risus et tellus hendrerit pretium ut quis arcu.

Aliquam vehicula erat eu odio mollis, ut vehicula mauris laoreet. Cras volutpat ante sed orci bibendum maximus. Pellentesque felis elit, laoreet vitae bibendum in, malesuada ut nulla. Vestibulum pharetra faucibus blandit. Morbi tristique at leo ut ultrices. Etiam sollicitudin justo lacus, non rutrum mi imperdiet non. Cras nec mauris odio. Quisque eleifend purus interdum nulla suscipit lobortis.

Proin vel nulla scelerisque nunc vehicula tempor. Curabitur non arcu ante. Etiam eget mattis tellus. Aenean molestie cursus ex, in pharetra eros hendrerit nec. In cursus egestas interdum. Ut scelerisque fringilla maximus. In facilisis vel dui vitae hendrerit. Integer a orci vel lectus lobortis fringilla. Phasellus eleifend commodo justo, cursus vehicula lorem. Donec a libero ultrices risus tempus condimentum imperdiet eget justo.

“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

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec dignissim nulla vel interdum maximus. Curabitur diam turpis, tincidunt sit amet dolor ac, egestas gravida lorem. Maecenas ac varius lorem. Vivamus porta neque eros, a molestie augue fermentum sit amet. Duis ut pellentesque metus. In sollicitudin placerat risus. Phasellus auctor ac arcu ac sagittis. Pellentesque nec iaculis eros. Vivamus quis convallis felis, nec facilisis risus. Suspendisse potenti. Etiam luctus elit eu est fermentum auctor. Curabitur mattis, nulla quis posuere iaculis, dolor lectus euismod eros, eu commodo felis nunc sed tortor. Quisque aliquam leo diam, nec placerat sapien placerat et. Integer vel accumsan mauris.

Nam sodales purus ac nibh maximus, non tempor ante sagittis. Donec vulputate vehicula dapibus. Duis mollis nisl vel fermentum eleifend. Nam sollicitudin metus vitae justo bibendum ullamcorper. Mauris sollicitudin, tellus nec placerat rutrum, est lectus commodo enim, in imperdiet est elit sit amet justo. Integer scelerisque gravida facilisis. Aenean rhoncus eget metus id rutrum. Curabitur vel volutpat quam, id lobortis lectus. Aenean semper semper massa, at finibus eros varius eget. Etiam volutpat tellus est, at rhoncus ligula scelerisque sed. Sed nisi sapien, semper in pharetra eget, pellentesque ac dolor.

In at rhoncus mauris, non semper turpis. Praesent iaculis lorem faucibus interdum vehicula. Nullam nec massa feugiat, pharetra turpis vitae, dignissim eros. Aliquam vel sapien et risus vulputate luctus non at risus. Quisque ac urna massa. Donec faucibus diam eu ex egestas viverra. In feugiat nunc at erat convallis aliquam. Suspendisse consequat urna quam, non dignissim massa pretium vel. Donec tincidunt pellentesque vestibulum. Sed aliquet imperdiet libero a fringilla. Maecenas tempor augue vel accumsan imperdiet. Pellentesque accumsan quis risus non malesuada. Maecenas sodales suscipit libero, fringilla imperdiet augue elementum non. Integer dignissim vehicula libero vel volutpat. Cras vel molestie tellus, ac varius nunc.

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

Resumo

Aliquam velit metus, elementum a metus a, pharetra molestie enim. Nam et dolor blandit, egestas libero ut, porttitor sem. Proin felis lorem, ultrices vel lacus eget, dapibus ornare urna. Donec et nibh ut orci condimentum cursus feugiat aliquet enim. Donec vel purus semper, iaculis dolor vel, placerat elit. Pellentesque nec magna pretium, eleifend risus eget, dapibus elit. Nam finibus sem at mauris auctor, a sodales erat bibendum. Phasellus mattis purus vel magna dignissim auctor. Sed at arcu a arcu hendrerit viverra ut a ligula. Sed et libero nibh. Sed sit amet nunc urna. Fusce eu arcu sed urna congue dapibus. Sed accumsan eget sem vitae euismod. Nulla lorem magna, viverra sit amet blandit vitae, accumsan et ante. Nunc ultrices eleifend pulvinar.

Vivamus nisi turpis, venenatis ut massa at, porttitor ornare tortor. Quisque tristique quam non nibh posuere pellentesque. Integer egestas felis ligula, ac rutrum turpis lobortis a. Donec scelerisque non sem porta scelerisque. Etiam molestie, arcu sagittis placerat fringilla, orci lacus lobortis odio, porttitor vehicula ex risus et velit. Sed tempor interdum volutpat. Sed ac dui eget justo convallis auctor. Duis lobortis mauris lacus, eu gravida ex hendrerit non. Nullam rutrum pretium nibh, id mollis est condimentum nec. Maecenas tincidunt erat sit amet mauris tristique, quis sagittis justo lobortis. Duis augue quam, semper eget ligula id, sagittis condimentum augue. Curabitur tincidunt scelerisque sollicitudin.

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). **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 it has a much more predictable behavior. Once it is expected virtualization to impact negatively, these VNFs have to keep performance degradation and as small as possible, along with a high stability. Thus, 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 a new technology. There are many reasons for this behavior, which includes burstiness and packet sizes.

A burstier traffic can cause packet drops buffer overflows on network [Cai *et al.* 2009] [Field *et al.* 2004] [Kushida e Shibata 2002], what degrades the network performance. So, characteristics on packet-trains and inter-packet times, that cause traffic burstiness. Also, a burstier and realistic traffic will not just impact on performance, but on measurement as well. Realistic and burstiness traffic impacts on bandwidth 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 traffic will have not a single packet-size as used by constant rate traffic generator tools, but a distribution [Castro *et al.* 2010].

Also, we have that realistic workload generators are essential security research [Botta *et al.* 2012]. Generation of realistic workloads is important on evaluation 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. That's because we will have an extra overhead introduced by the virtualization layer. Also, on SDN networks, each new flow arriving o a switch introduces an extra communication load between it and the controller, which can generate a performance degradation. We also have a flow-oriented operation on SDN switches. Since its operation relies on queries on flow tables, a stress load must have the same flow properties of an actual ISP scenario.

Therefore, there is a demand for the study of the impact of a realistic traffic on this new sort of environment. How VNFs and virtualized middle-boxes and SDN testbeds will behave if stressed with a realistic traffic load in comparison to a constant rate traffic is a relevant subject. Also, how its underlying hardware will behave with this type of a load is an important case of study [Veitch *et al.* 2015].

1.2 Open-source Solutions

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.

Some traffic generator tools offer support emulation of single application workloads. But this does not correspond to real complex scenarios, with a large number of distributed hosts in an Internet Service Provider (ISP) or even a Local Area Network (LAN).

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. They can give a good control of the traffic and high rates. The most relevant examples of features are:

- Throughput, the number of bytes, number of packets.
- Protocol: most of the tools give support to network and transport protocols. Many also offer support to Link and Application protocols;
- Header and payload configuration: support to header customization includes source and destination port/addresses, QoS parameters, flags, etc. Some traffic generators also allow customizing payload bytes.
- Inter-packet time: custom inter-packet times. Some tools offer a set of stochastic distributions to control inter-packet times. The user can use these stochastic functions to emulate realistic inter-arrival times.

- On/off periods: support of packet trains periods. Many offer stochastic models to control these bursts.
- Sending time and start-time: the user can use these features to control flows timings.
- Packet-size: support for different packet sizes. Some tools offer constant values or stochastic distributions to control packet sizes.
- Many configurable flows.
- Emulation of specific applications: common examples are Web server/client communication, VoIP, HTTP, FTP, p2p applications, and so on.

But, in this case, selecting a good configuration is by itself a research project. How to use each parameter to simulate a specific scenario is a hard question [Bartlett e Mirkovic 2015] [Leland *et al.* 1994]. It is a manual process and demands implementation of scripts or programs leveraging human (and scarce) expertise on network traffic patterns and experimental evaluation.

Some tools like Swing and Harpoon, try to use the best of both worlds. Both use capture traces to set intern parameters, enabling an easier configuration. Also, Swing uses complex multi-levels 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 on setup and configuration. Swing [Vishwanath e Vahdat 2009] aims to generate realistic background traffic, but do not reach high throughputs [Vishwanath e Vahdat 2009] [Bartlett e Mirkovic 2015]. As is possible to see, this a result of the fact that its traffic generation engine is coupled to its modeling framework. You can't opt to use a newer/faster packet generator. The only way of replacing the traffic engine is changing and recompiling the original code. And this is a hard task.

Since synthetic traffic traces generation is mature in academia, the creation of custom network workloads through the configuration of high-throughput open-source is an affordable task. But it is not often available in an automatic way, and most of the times is a challenging question [Bartlett e Mirkovic 2015], and still, requires expert knowledge. It requires time, study, and is vulnerable to human mistakes and lack of validation. Such work may require weeks to complete a realistic reproduction of a single scenario, so most of the time it is just not done. We argue that widely (i.e. affordable) existing approaches can be regarded as simplistic often point solutions to more general cases.

Also, by itself, choosing which workload generator tool may fit better for the user needs is not a simple question. Tools like D-ITG[1] provide support to many different stochastic

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

functions, Ostinato[2] provide a larger support for protocols, and a higher throughput for each thread [Srivastava *et al.* 2014], and other like Seagull[3] are responsive.

1.3 *SIMITAR*: SnIffing, ModellIng and TrAffic geneRation

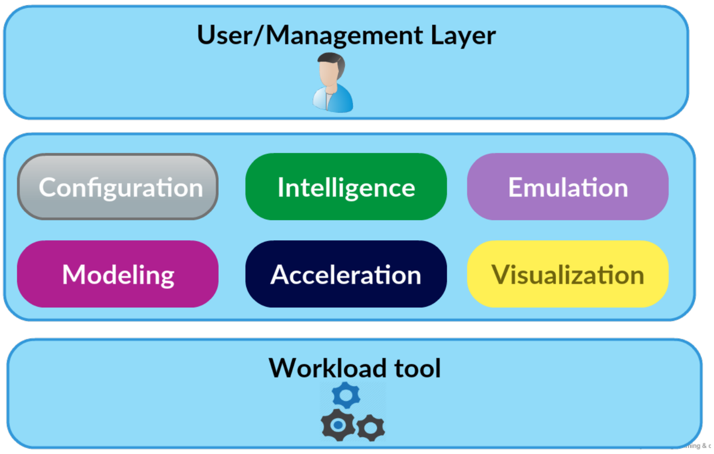


Figure 1 – Conceptual idea: a <configuration><modelling><intelligence><emulation><acceleration><visu
source>

Now, we are going to formalize these exposed concepts in a proposal of research and development. Or main targets here are:

- Survey the available open-source Ethernet workload tools available, addressing different features available;
- Define what is a realistic Ethernet traffic, and the man approaches found in literature;
- Create a general method for modeling and parameterization of Ethernet traffic, aiming to mimic any traffic provided as input;
- Create a software *tool* able of *learning* patterns and characteristics of real network traffic traces, and based on the acquired knowledge, allowing to reproduce network traffic with similar (but not equal) characteristics, *avoiding* the storage of *pcap* files. It must have a flow-oriented operation, and be able to choose the best features to represent each flow traffic. It will generate realistic Ethernet traffic based on these features learned, using available open source tools and APIs;

We call our tool *SIMITAR*, an acronym for *SnIffing, ModellIng and TrAffic geneRation*. These targets summarize a target of research, based on what we have exposed until now. We also will define some extra features, we want to have on our tool:

[2] http://ostinato.org/

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

- Traffic Flow Programmability: the tool records this parameterization on a light-weight and human-readable version of a *pcap* file. So the user can create his own models of traffic based on flows, in a platform-independent manner. He will not have to learn how to configure and script each tool.
- Easy expansion: The sniffing, modeling, and traffic generation processes must work separately, so updates will be easy to manage. Also, the traffic generation is flow-oriented (it generates each flow individually). Thus the scheduling of each flow is platform-independent. In this way, is much easier to expand the support for new traffic generation engines.

Our main goal is to offer an easier configuration, realism than the available platforms today, at a reasonable speed. We are introducing programmability and abstraction to the traffic generation, since the user may create a custom 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 file huge *pcap* with a size of many *gigas*, we create a light-weight 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 a 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.

Now, we are going to formalize these exposed concepts in a proposal of research and development. Or main targets here are:

- Survey the available open-source Ethernet workload tools available, addressing different features available;
- Define what is a realistic Ethernet traffic, and the man approaches found in literature;
- Create a general method for modeling and parameterization of Ethernet traffic, aiming to mimic any traffic provided as input;
- Create a software *tool* able of *learning* patterns and characteristics of real network traffic traces, and auto-configure the network traffic workloads, allowing to reproduce network traffic with similar (but not equal) characteristics, avoiding storage of *pcap* files.

- It must have a flow-oriented operation, and be able to choose the best features to represent each flow traffic. It will generate realistic Ethernet traffic based on these features learned, using available open source tools and APIs;

We call our tool *SIMITAR*, an acronym for *SnIffing, ModellIng and TrAffic geneRation*. These targets summarize a target of research, based on what we have exposed. Now, we will define some extra features, we want our tool to have:

- Traffic Flow Programmability: the tool records this parameterization on a light-weight and human-readable version of a *pcap* file. So the user can create his own models of traffic flow based traffic, in a platform-independent manner. He will not have to learn how to configure and script each tool.
- Easy expansion: The sniffing, modeling, and traffic generation processes must work separately, so updates will be easy to manage. Also, the traffic generation is flow-oriented (it generates each flow individually). Thus the scheduling of each flow is platform-independent. In this way, is much easier to expand the support for new traffic generation engines.

Our main goal is to offer an easier configuration, realism than the available platforms today, at a reasonable speed. We are introducing programmability and abstraction to the traffic generation, since the user may create a custom 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 file huge *pcap* with a size of many *gigas*, we create a light-weight 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 a 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.

1.4 Document Overview

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

In the chapter section 2, we go deeper 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 present a brief survey of important topics on realistic traffic generation. There, we are defining some important 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 will define our tests using this set. Finally, we will discuss some main topics on emerging network scenarios: SDN and NFV.

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

The following chapter 4 is a direct sequence of the last. We present how the modeling process actually works, using a defined data set (which we are going to use in the rest of the work). We also present some evaluation methods to check the modeling quality. We also describe our used and developed algorithms. Finally, we give some implementation.

In the chapter Proof of concepts and Validation **??**, 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 highlights future works 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. First of all, we will define what is a traffic generator tool, and introduce a taxonomy that distinguishes them. After this, we will present a survey of many available open-source and free traffic generators. We introduce each tool and 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 for this topic.

We also 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 what features a realistic traffic should have and highlight important features needed to control to achieve it. We can count self-similarity and burstiness, header fields, packet size and flow features.

In the following section, we will make a brief 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 i the open-source community[1]. In fact, as exposed in the chapter 1 it is a hard task to find what is the best tool for your needs. 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, not an equipment. Another possibility a huge variety of protocols available, or even support of new protocols.

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

Alongside with the traffic generators, there are many APIs available. There are low-level APIs which enables direct packet injection and capture, 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 found is according to its abstraction level [Botta *et al.* 2010]. They are: Application-level/Special-scenarios, flow-level, packet-level and Closed-loop 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/Special-scenarios traffic generators: they try to emulate the behavior of network applications, simulating real workloads stochastically, responsively [3] or both. 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 precisely features at the flow level [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 them into the network based on Inter departure times

[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

(IDT) and packet size(PS). [Molnár *et al.* 2013] classify them as replay engines, maximum throughput generators, and 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 [Welcome to BRUTE homepage! 2003], KUTE [Welcome to BRUTE homepage! 2003], and pktgen [pktgen 2009].

Closed-loop and 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.g1

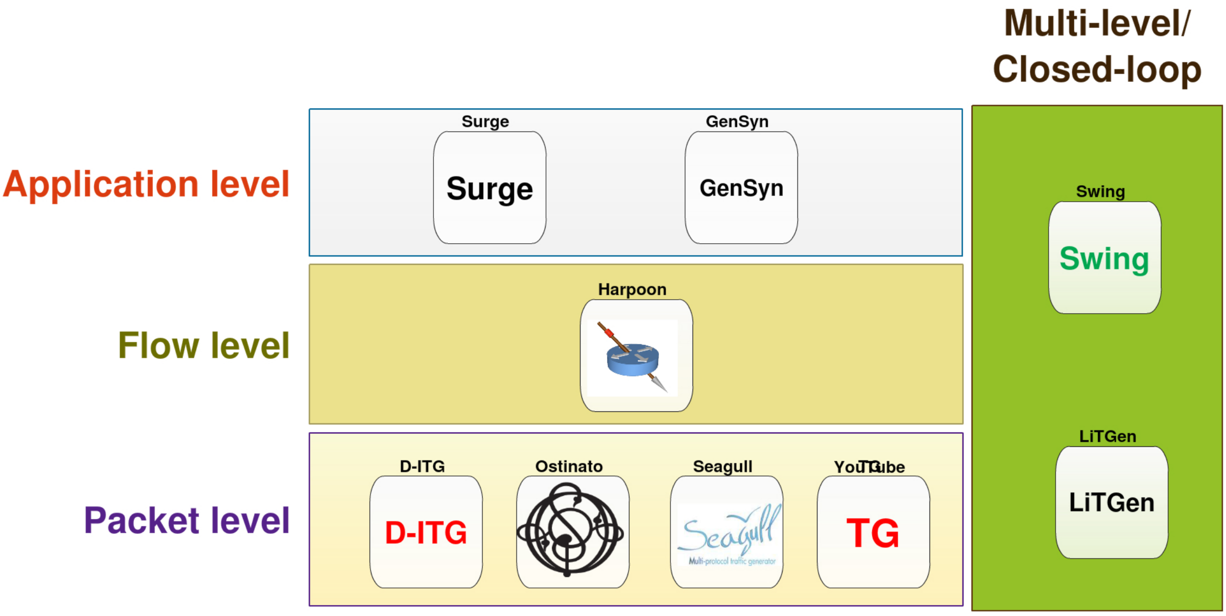


Figure 2 – Diagram representing different traffic generators, according to its abstraction layer.

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.

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].

2.2 Open-source and free traffic generator tools

In this section we will presented a short review of many open-source tools available for synthetic traffic generation and benchmark. The goal in this section is to present both the most mentioned tools in the literature, and the most recent and advanced ones. On table 1 is presented a short review of some of the main features of such tools, such as support for Operational systems, protocols, stochastic functions available for traffic generation, available interfaces, and traffic generator class. Some free, but not open-source traffic generators are listed as well.

Before present our survey, we will refer to some tools mentioned in literature, but we couldn't found source code and manual. **BRUNO** [Antichi *et al.* 2008] is traffic generator implemented aiming performance and accuracy on timings. It has many configurable parameters that allow emulation of many web server scenarios. **Divide and conquer** [Molnár *et al.* 2013]: Divide and conquer is replay engine that works in a distributed manner. It is able to split traces among multiple commodity PCs, and reply packets, to produce realistic traffic. Some others mentioned tools [D-ITG, Distributed Internet Traffic Generator 2015] we weren't able to find any reference of available features are: **UDPgen**, **Network Traffic Generator**, **Packet Shell**, **Real-Time Voice Traffic Generator**, **PIM-SM Protocol Independent Multicast Packet Generator**, **TTCP**, **SPAK, Packet Generator**, **TfGen**, **TrafGen** and **Mtools**.

2.2.1 Packet-level traffic generators

D-ITG [Botta *et al.* 2012] [D-ITG, Distributed Internet Traffic Generator 2015]: D-ITG(Distributed Internet Traffic Generator) is a platform capable to produce IPv4 and IPv6 traffic defined by IDT and PS probabilistic distributions such as constant, uniform, Pareto, Cauch, Normal, Poisson, Gamma, Weibull, and On/Off; both configurable and pre-defined for many applications, from Telnet, through online games. It provide many flow-level options of customization, like duration, start delay and number of packets, support to many link-layer and transport-layer protocols, options, sources and destinations addresses/ports. If has support for NAT traversal, so it is possible to make experiments between two different networks separated by the cloud. D-ITG can be also used to measure packet loss, jitter, and throughput. D-ITG may be used through a CLI, scripts, or a C API, that can be use to create applications and remotely control another hosts through a daemon. The code may be downloaded on its official website.

Ostinato [Ostinato Network Traffic Generator and Analyzer 2016]: Ostinato is a packet crafter, network traffic generator and analyzer with a friendly GUI("wireshark in reverse" as the documentation says) and a Python API. This tool permits craft and send packets of different protocols at different rates. Support Server/Client communication, and a huge variety of protocols, from the link layer(such as 802.3 and VLAN) to the application layer (such HTTP and SIP). It is also possible to add any unimplemented protocols, through scripts defined by the user. The code may be downloaded on its official website (http://ostinato.org/downloads).

Seagull [Seagull – Open Source tool for IMS testing 2006] [Seagull: an Open Source Multi-protocol traffic generator 2009]: Seagull is a traffic generator and test open-source tool, released by HP. It has support of many protocols, from link layer to application layer, and it support is easily extended, via XML dictionaries. As the documentation argues, the protocol extension flexibility is one of the main features. It support high speeds, and is reliable, being tested through hundreds of hours. It can also generate traffic using three statistical models: uniform(constant), best-effort and Poisson.The code may be downloaded on its official website.

BRUTE [Welcome to BRUTE homepage! 2003]: Is a traffic generator that operates on the top of Linux 2.4-6 and 2.6.x, not currently being supported on newer versions. It also support some stochastic functions(constant, poisson, trimodal) for departure time brust, and is able to simulate VoIP traffic. The source code(old release) may me fond on official website and on google code repository (https://code.google.com/archive/p/brute/).

PackETH [PACKETH 2015]: PackETH is GUI and CLI stateless packet generator tool for ethernet.It support many adjustments of parameters, and many protocols as well, ans can set MAC addresses. The download link may be found on it official website.

Iperf [iPerf - The network bandwidth measurement tool]: Ipef is network traffic generator tools, designed for the measure of the maximum achievable bandwidth on IP networks, for both TCP and UDP traffic, bu can evaluate delay, windows size, and packet loss. Although it is, originally, a command line tool, some developers have created a GUI interface, called Jperf [Iperf 2016]. There is also a Java API, for automating tests [jperf 2015]. Support IPv4 and IPv6. The source code can be found on their offical website.

NetPerf [Welcome to the Netperf Homepage]: Netperf is a benchmark tools that can be used to measure performance of many types of networks, providing tests for both unidirectional throughput, and end-to-end latency. It uses a command line interface, and has support to TCP, UDP and SCTP, both over IPv4 and IPv6. The source code can be downloaded at website homepage.

sourcesOnOff [Varet 2014] [sourcesonoff]: sourcesOnOff is a recent traffic generator released on 2014, that aims to generate realistic synthetic traffic using probabilistic models to controll on and off time of traffic flows. As shown on the paper, it is able to guarantee self-similatiry, an has support to many probabilistic distributions for the on/off times: Weibull, Pareto, Exponential and Gaussian. Supports TCP and UDP over IPv4. The source code can be found on the author personal page.

TG [Traffic Generator 2011]: TG is a traffic generator that is able to generate and receive one-way packet streams transmitted from the UNIX user level process between source and traffic sink nodes. It is controlled by a simple specification language, that enables the craft of different lengths and interarrival times distributions, such as Constant, uniform, exponential and on/off(markov2).

[4]**MGEN** [Multi-Generator (MGEN)]: MGEN (Multi-Generator) is a traffic generator developed by the Naval Research Laboratory (NRL) PROTocol Engineering Advanced Networking (PROTEAN) Research Group. It uses as input can be used to emulate the traffic patterns of unicast and/or multicast UDP and TCP IP applications. It support many different types of stochastic functions, nominated periodic, Poisson, burst jitter and clone which are able

[4] not open-source

to control inter departure times and packet size.

KUTE [KUTE – Kernel-based Traffic Engine 2007]: KUTE is a kernel level packet generator, designed to have a maximum performance traffic generator, and receiver mainly for use with Gigabit Ethernet. It works in the kernel level, sending packets as fast as possible, direct to the hardware driver, bypassing the stack. But KUTE works only on Linux 2.6, and has only be tested on Ethernet Hardware. Also, it only supports constant UDP traffic. The source code of the tools is available on its homepage.

RUDE & CRUDE [RUDE & CRUDE 2002]: RUDE(Real-time UDP Data Emitter) and CRUDE(Collector for RUDE), are small and flexible programs which runs on user-level. It has a GUI called GRUDE. It works in a similar way MGEN does, but has a higher time resolution. But it works only with UDP. The source code can be found on its homepage.

[5]**NetSpec** [NetSpec – A Tool for Network Experimentation and Measurement]: NetSpec is tool designed to do network tests, as opposed to doing point to point testing. NetSpec provides a framework that allows a user to control multiple processes across multiple hosts from a central point of control, using daemons that implement traffic sources and sinks, along with measurement tools. Also, it is able to model many different traffic patterns and applications, such as host maximum rate; Constant Bit Rate (CBR) at user level; WWW, World Wide Web; FTP, File Transfer Protocol; telnet; MPEG video; voice and video teleconference. This source and application wasn't available anymore on its webpage.

Nping []: active hosts, as a traffic generator for network stack stress testing, ARP poisoning, Denial of Service attacks, route tracing, etc. Nping CLI permits the users control over protocols headers. To source code can be downloaded on its website.

TCPreplay [Tcpreplay home]: TCPreplay is a user-level replay engine, that can use pcap files as input, and then forward, packets in a network interface. It can modify some header parameters as well. The source code of TCPreplay can be found on github https://github.com/synfinatic/tcpre

TCPivo [Feng *et al.* 2003] [TCPivo: A High Performance Packet Replay Engine]: TCPivo is a high-speed traffic replay engine that is able to read traffic traces, and replay packets in a network interface, working at kernel level. Until the present data, it is not currently supported kernel versions greater than 2.6. The source code can be fond on its official website.

NetFPGA PacketGenerator [NetFPGA]: NetFPGA PacketGenerator is a hardware-based traffic generator and capture tool, build over the NetFPGA 1G, and open FPGA platform with 4 ethernet interfaces of 1 Gigabit of bandwidth each. It is a replay engine tool which uses as input *pcap* files. It is able to accurately control the delay between the frames, with the default delay being the same in the pcap file. It is also able to capture packets, and report statistics of the traffic. The code can be downloaded on github.

NetFPGA Caliper [PreciseTrafGen]: is a hardware-based traffic generator, build on

[5] not open-source

NetFPGA 1G, built over NetThreads platform, a FPGA microprocessor which support threads programing. Different form NetFPGA PacketGenerator, Caliper is able to produce live packets. It is written in C. The code can be downloaded on github.

NetFPGA OSNT [OSNT Traffic Generator]: OSNT(Open Source Network Tester) is hardware based network traffic generator built over the NetFPGA 10G. As NetFPGA 1G, NetFPGA 10G is a FPGA platform with 4 ethernet interfaces, but with 10 Gigabits of bandwidth. OSNT is a replay engine, and is loaded with pcap traces. OSNT Source Network Test code can be found on github.

Dpdk Pktgen [Getting Started with Pktgen 2015]: Pktgen is a traffic generator measurer built over DPDK. DPDK is a development kit, a set of libraries and drivers for fast packet processing. DPDK was designed to run on any processor, but has some limitation on terms of supported NICs, that can be found on its website.

MoonGen [Emmerich *et al.* 2015] [MoonGen] : MoonGen is a scriptable high-speed packet generator built over DPDK and LuaJIT. It is able to send packets at 10 Gbit/s, even with 64 bytes packets on a single CPU core. MoonGen can achieve this rate even if each packet is modified by a Lua script. Also, it provides accurate timestamping and rate control. It is able to generate traffic using several protocols (IPv4, IPv6, IPsec, ARP, ICMP, UDP, and TCP), and can generate different inter-departure times, like a Poisson process and burst traffic. This project can be found on github.

gen_send/gen_recv [gen_send, gen_recv: A Simple UDP Traffic Generater Application]: gen_send and gen_recv are simple UDP traffic generator applications. It uses UDP sockets, and gen_send is able to control features like desired data rate, packet size and inter packet time. You can find their source at http://www.citi.umich.edu/projects/qbone/generator.html.

mxtraf [mxtraf]: mxtraf enables that a small number of hosts to saturate a network, with a tunable mixture of TCP and UDP traffic. You can found the source code at http://mxtraf.sourceforge.net/.

Jigs Traffic Generator (JTG) [Short User's guide for Jugi's Traffic Generator (JTG)]: is a simple, accurate traffic generator. JTG process only sends one stream of traffic, and steam characteristics are defined only by command line arguments. It also supports IPv6. You can find the code at http://www.netlab.tkk.fi/ jmanner/jtg/.

***Poisson Traffic Generator**:

2.2.2 Application-level/Special-scenarios traffic generators

[8]**ParaSynTG** [Khayari *et al.* 2008]: application-level traffic generator configurable by input parameters, which considers most of the observes www traffic workload properties.

[8] not open-source

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*)

[9]**EAR** [Molnár *et al.* 2013]: traffic generator that uses a technique called "Event

[9] not open-source

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	[6]	[7]	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

Reproduction Ratio” to mimic wireless IEEE 802.11 protocol behavior.

[10]**GenSyn** [GenSyn - generator of synthetic Internet traffic]: network traffic generator implemented in Java, that mimic TCP and UDP connections, based on user behavior.

[10] not open-source

Surge [Barford e Crovella 1998]: Surge is an application level workload generator which emulates a set of real users accessing a web server. It matches many empirical measurements of real traffic, like server file distribution, request size distribution, relative file popularity, idle periods of users and other characteristics. Its source code can be found at http://cs-www.bu.edu/faculty/crovella/surge_1.00a.tar.gz.

Httperf [httperf(1) - Linux man page]: Is an application lever traffic generator to measure web server performance. It uses the protocol HTTP (HTTP/1.0 and HTTP/1.1), and offer many types of workloads while keeping track of statistics related to the generated traffic. Its most basic operation is to generate a set of HTTP GET requests and measure the number of replies and response rate. You can found its source code on GitHub https://github.com/httperf/httperf.

VoIP Traffic Generator: it is a traffic generator written in Perl, and creates multiple streams of traffic, aiming to simulate VoIP traffic. You can found its code at https://sourceforge.net/projects/v

2.2.3 Flow-level and Closed-loop and multi-level traffic generators

Harpoon [Sommers *et al.* 2004]: Harpoon is a flow based traffic generator, that is able to automatically extract form Netflow traces parameters, in order to generate flows that exhibit the same statistical characteristics measured before, including temporal and spatial characteristics. The source code of Harpoon can be found on github (https://github.com/jsommers/harpoon).

Swing [Vishwanath e Vahdat 2009] [The Swing Traffic Generator 2016]: Swing is a closed-loop multi-layer, and network responsive generator. It is able to read capture traces, and captures the packet interactions of many applications, being able to models distributions for user, application, and network behavior, stochastic and responsively. As mentioned in the previous section, Swing is able to model user behavior, REEs, connection, packets, and network. The code may be downloaded on its official website.

[11]**LiTGen**(Lightweight Traffic Generator) [Rolland *et al.* 2007] is a open-loop, multilevel traffic generator. It is able to model wireless network traffic in a peer user and application basis. This tool models the traffic in three different levels: packet level, object level (smaller parts of an application session), and session level.

2.2.4 Others traffic generation tools

NFPA [Csikor *et al.* 2015]: NFPA is a benchmark tools based on DPDK Pkgen, specialized on executing and automatize performance measurements over network functions. It works being directly connected to an specific device under tests. It uses built-in and user defined traffic traces, and Lua scripts control and collect information of DPDK Pktgen. It has an command line and Web interface, and automatically plot the results.

[11] not open-source

LegoTG [Bartlett e Mirkovic 2015]: LegoTG is a modular framework for composing custom traffic generation. It aims to simplify the combination on the use of different traffic generators and modulators on different testbeds, automatizing the process of installation, execution, resource allocation and synchronization via a centralized orchestrator, which uses a software repository. It already has support to many tools, and to add support to new tools is necessary to add and edit two files, called TGblock, and ExFile.

2.3 Overview of Network 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.

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

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 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

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)

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 application. 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].

2.4 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 Szabó [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].

2.4.1 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].

2.4.2 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 Szabó [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.

2.4.3 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 nonstationary 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.

2.4.4 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 Szabó [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].

2.4.5 Study cases

2.4.5.1 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.

2.4.5.2 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.

2.4.5.3 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 [12]. 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.

2.4.5.4 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[13], which aim to evaluate the correlation between inter-trains duration of real and

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

[13] 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.

generated traffic;

- Measurement of Autocorrelation[14] of the measured throughput of the real and synthetic traffic;
- Hurst exponent computation of the real and the synthetic trace;

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%.

2.4.5.5 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.

2.4.5.6 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

[14] 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.

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 developed tool, called SIMITAR: SnIffing, ModellIng and TrAffic geneRation, and how it works. First, we will discuss its architecture and components. Then we will present a brief tutorial of how to expand its support for a traffic generator engine.

We abstract its whole operation cycle in the figure 3. Our tool, from live captures or *pcap* file collects raw data from Ethernet traffic. It then breaks these data into different flows. It uses this data to generate a set of parameters for our traffic models, using a set of algorithms. Finally, it provides these parameters to a traffic generator engine and controls its packets injection.

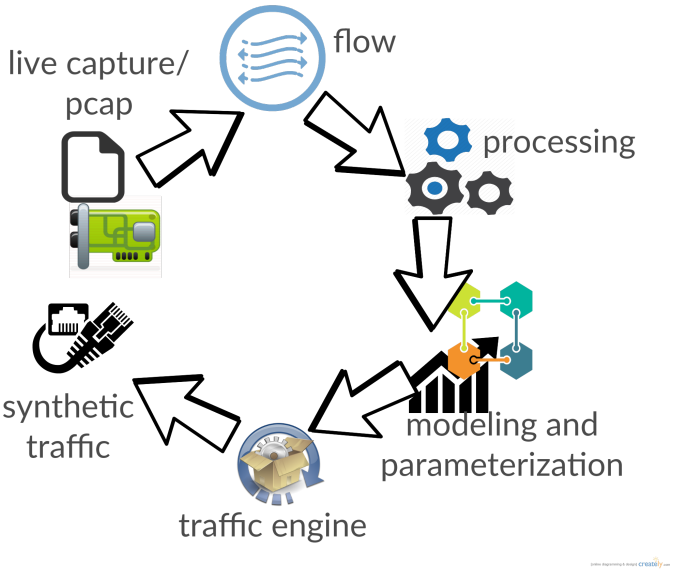


Figure 3 – This figure represents an operation cycle of SIMITAR, emphasizing each main step: sniffing, flow classification, data storing, data processing and fitting, model parameterization, and synthetic traffic generation.

3.1 SIMITAR Architecture

To meet the requirements presented in the section 1, we will present a solution, and define how each part works, as in a low-level requirement list [Sommerville 2007].

SIMITAR architecture is presented in the figure 4, and it is composed of five components: a *Sniffer*, a *SQLite database*, a *TraceAnalyzer*, a *FlowGenerator* and the *Network Traffic Generator*, as subsystem. We describe each part below.

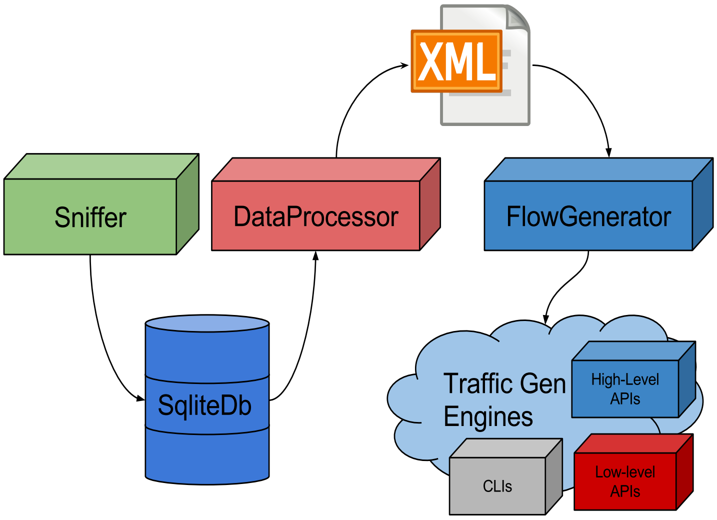


Figure 4 – Architecture of SIMITAR

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 (Link Protocol, Network Protocol, Network Source Address, Network Destination Address, Transport Protocol, Transport Source Port, Transport Destination Port). It can work over a *pcap* file or over an Ethernet interface.

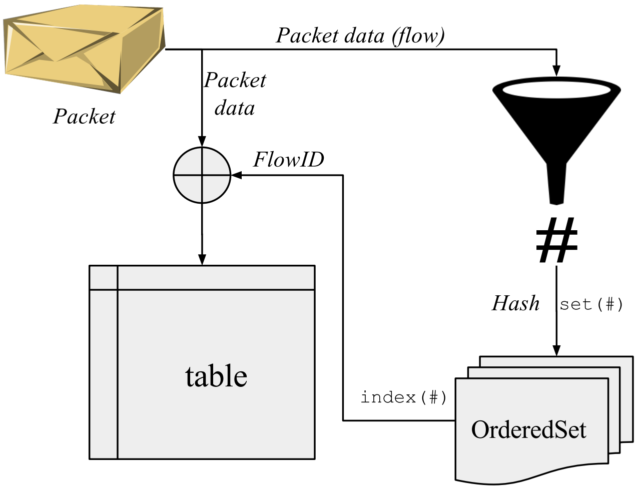


Figure 5 – SIMITAR's sniffer hash-based flow classification

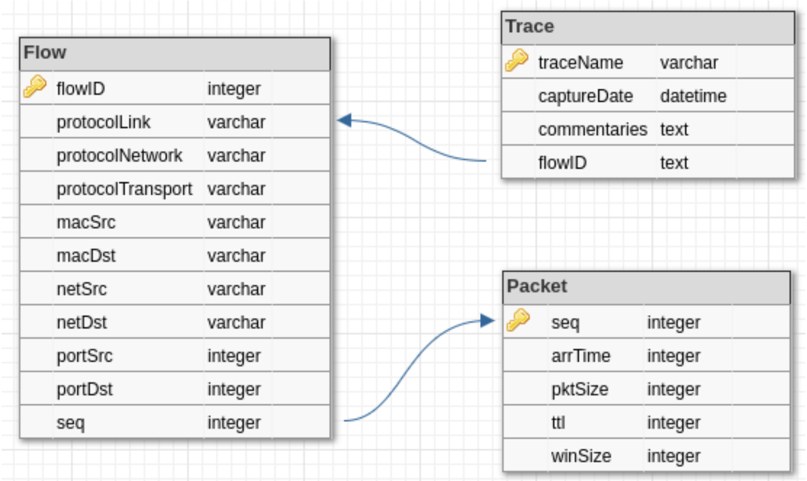
The first version of this component was implemented in Shell Script (Bash). Tsahrk[1] was used to extract header fields, and Awk to match the flows, and Sed/Awk to create the Sqlite queries. This version was too slow to operate in real time on Ethernet interfaces. On the other hand, was fast to implement, and enable the implementation of the other components, before this final version was released.

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

The second version was implemented using Python. This version used Pyshark[2] as sniffer library. The flow classification is made by a class called FlowId. It contains class we developed called OrderedSet. A set is a list of elements with no repetition, but do not keep track of the insertion order. But our OrderedSet does. Also, it make use of a 32 bits hash function of the family FNV. The listed header fields are inputs for a hash function, its value is set on the ordered set, and its order is returned. This value is set as a packet flowID.

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

3.1.2 SQLite database



Flow	
flowID	integer
protocolLink	varchar
protocolNetwork	varchar
protocolTransport	varchar
macSrc	varchar
macDst	varchar
netSrc	varchar
netDst	varchar
portSrc	integer
portDst	integer
seq	integer

Trace	
traceName	varchar
captureDate	datetime
commentaries	text
flowID	text

Packet	
seq	integer
arrTime	integer
pktSize	integer
ttl	integer
winSize	integer

Figure 6 – SIMITAR's SQLite database relational model

The database will store the collected raw data for further analysis. It is responsible for storing data from different traces in different tables. The *Sniffer* and the *TraceAnalyzer* can access the database.

We choose an SQLite database, because according to the SQLite specifications[3], it is the best option for our database. It is simple and well-suitable for an amount of data smaller than terabytes.

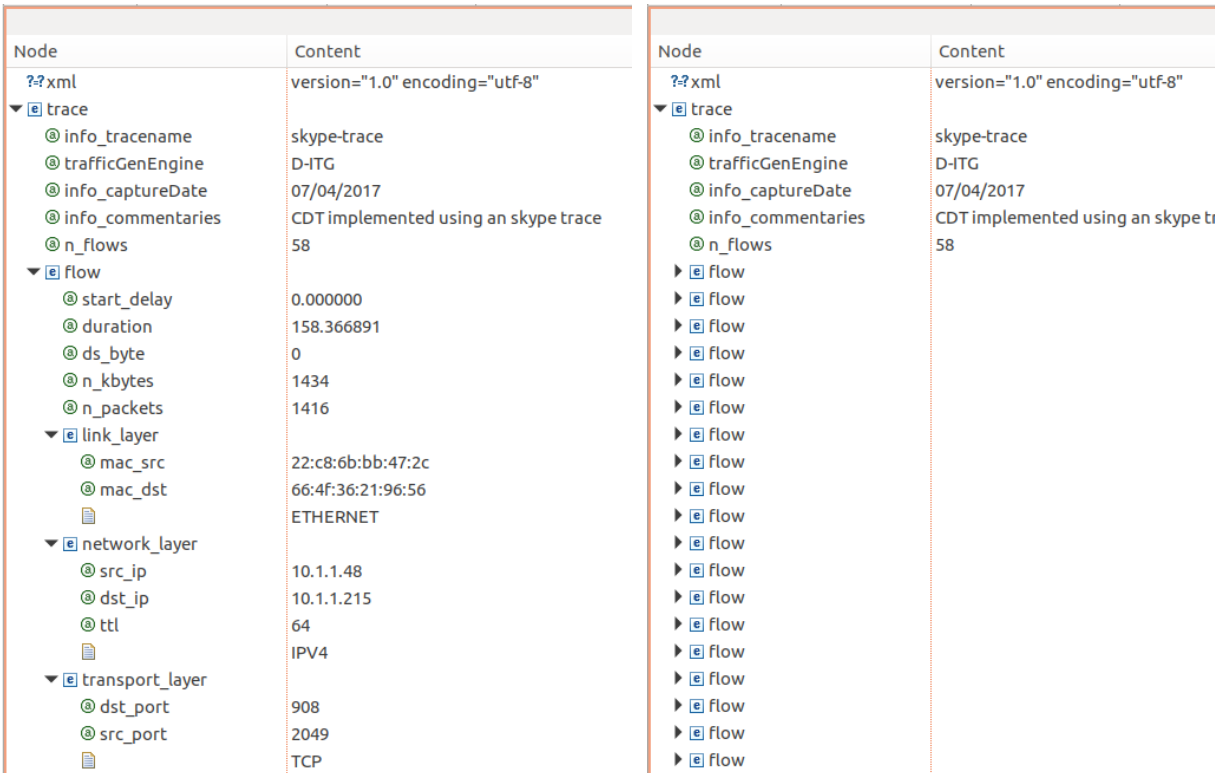
In the figure 6 we present the relational model of our database, which stores a set of features extracted from the packets, along with the FlowID calculated by the sniffer component.

3.1.3 Trace Analyzer

This module is the core of our project. It creates a compact trace descriptor via the analysis of the collected data. We define here a Compact Trace Descriptor (CTD) as a human

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

[3] 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) first

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	
flow	

(b) second

Figure 7 – 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.

and machine readable file, which describes an original traffic trace trough a set of flows, each of them described by a set of parameters, like header information and analytical models.

Therefore, the Trace Analyze learns the features from raw data of traces (stored in the SQLite database) and write them in an XML file. In the figure 7 we show a directory diagram of a CDT file. It has many of many flow fields, and each one contains each parameter estimated.

3.1.3.1 Flow features

Some features are unique per flow and can be directly measured from the data. They are:

- Flow-level properties like duration of flow, start delay, number of packets per flow, number of KBytes per flow;
- Packet-level QoS metrics like bit rate and packet rate;
- Header fields, like protocols (IPv4, IPv6, TCP, UDP, ICMP, DCCP, SCTP, etc), 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

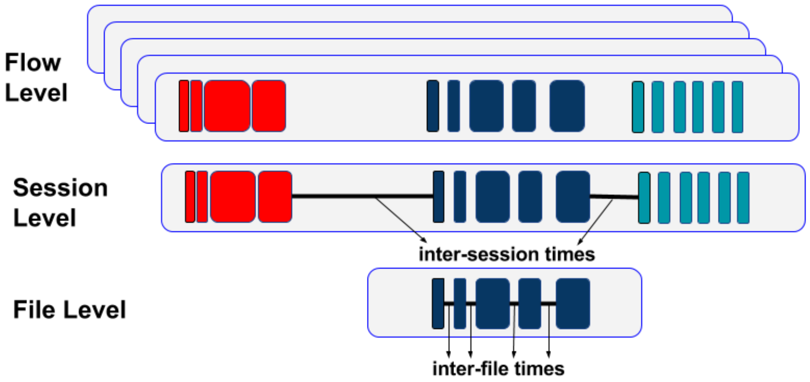


Figure 8 – The schema of the modified version of the Harpoon algorithm we adopt on SCIMITAR.

To represent IPT, we adopt a simplified version of the Harpoon 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 8.

Harpoon uses a more conventional definition of each level, based on the measurement of SYN and ACK TCP flags. Harpoons use TCP flags (SYN) to classify packets in different levels (named there: file, session and user level). Since we are just modeling inter-packet times, we choose to estimate these values, based on inter-packet times. Since we want to generalize this approach for any protocol, the distinction will be made based on the time delay between packets.

In our version of the algorithm, we define (as Harpoon does) three different layers of data transference to model and control: file, session, and flow.

For SIMITAR, a file is just a sequence of consecutive packets transmitted continuously, without large interruption of traffic. It can be, for example, packets transmitted downloading a file, of UDP packets of a connection or a single ICMP echo packet. The session-layer refers to a sequence of multiple files transmitted between a source and a destination, belonging to the same flow. And the flow level refers to the conjunct of flow classified by the Sniffer.

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

Session-layer: At the session layer, we use a deterministic approach for evaluating times between transferred files (OFF times) and the actual file transference (ON times). It is basically a deterministic ON/OFF model. 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. We did this, first because ON times can be arbitrary smalls, and they could be incompatible with acceptable ON periods for traffic generators. Second, because in the case of just one packet, the ON time would be zero. OFF times, on the other hand, are defined by the constant `m_session_cut_time`. If the time between packets of the same flow is larger than this, we consider it belonging to a different file, so it is a Session OFF time. In this case, we use the same value of the constant *Request and Response timeout* of Swing [Vishwanath e Vahdat 2009] (30 seconds) ««pode mudar, calibrar»». The control of ON/OFF periods is made by the `NetworkFlow` class.

File-layer: Here we model the inter-packet times at the file level. To estimate inter-packet times within files, we select all inter-packet times smaller than `m_session_cut_time`. All files within the same flow are considered to follow the same model. We delegate the control of the inter-packet times to the underlying workload engine tool. We ordered them, from the best to the worst. Currently, we are using eight different stochastic functions parameterizations. They are Weibull(linear regression), Normal(mean/standard deviation calculation), Exponential(mean and linear regression estimation), Pareto(linear regression and maximum likelihood), Cauchy(linear regression) and Constant(mean calculation). From those, Weibull, Pareto, and Cauchy are heavy-tailed functions, and therefore self-similar processes. ««pode mudar, calibrar»» 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) [Varet 2014] [Yang 2005]. We will enter in deeper details on this methodology on the chapter 4. The methodology of selection is presented in the figure 9

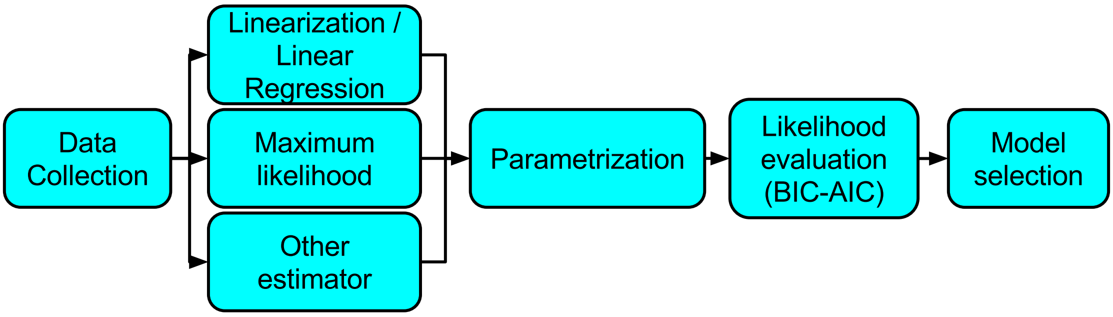


Figure 9 – Diagram of parameterization and model selection for inter-packet times and inter-file times.

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 first 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 is not a requirement, we prefer to use just simple models. But, when there is no packet we set a flag NO_MODEL, and when there is just a single packet we just use the constant model. We then calculate the BIC and AIC for each of them and sort according to the AIC value by default, but we decide to set the constant model as the first.

As is possible to see on many studies [Castro *et al.* 2010] [Ostrowsky *et al.* 2007], since the standard deviation of each mode is small, constant fittings use to give good approximations. Also, it is computationally cheaper than the others 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 indicator for the constant model.

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.

```
<flow start_delay="31.621941" duration="91.584654" ds_byte="0" n_kbytes="1"
 ↪  n_packets="13">
    <link_layer mac_src="no-mac-src" mac_dst="no-mac-dst">ETHERNET</link_layer>
    <network_layer src_ip="10.1.1.48" dst_ip="216.58.199.35"
     ↪  ttl="64">IPV4</network_layer>
    <transport_layer dst_port="34683" src_port="443">TCP</transport_layer>
    <application_layer>HTTPS</application_layer>
    <inter_packet_times>
        <stochastic_model name="pareto-ml" aic="-24.738412" bic="-23.768599"
         ↪  param1="0.152500524612205" param2="0.000024050000003"/>
        <stochastic_model name="weibull" aic="-16.155977" bic="-15.186164"
         ↪  param1="0.130469107098031" param2="0.063433369086366"/>
        <stochastic_model name="pareto-lr" aic="49.322192" bic="50.292005"
         ↪  param1="0.892551148108768" param2="0.000024050000003"/>
        <stochastic_model name="exponential-me" aic="76.776570" bic="77.746383"
         ↪  param1="0.131026317153354" param2="0.000000000000000"/>
        <stochastic_model name="normal" aic="105.882623" bic="106.852436"
         ↪  param1="7.632054549999999" param2="17.598918053235813"/>
        <stochastic_model name="exponential-lr" aic="116.913056" bic="117.882870"
         ↪  param1="0.009752000000000" param2="0.000000000000000"/>
        <stochastic_model name="cauchy" aic="128.406532" bic="129.376345"
         ↪  param1="1.935832000000000" param2="-7.245692000000000"/>
        <stochastic_model name="constant" aic="inf" bic="inf" param1="7.632054549999999"
         ↪  param2="0.000000000000000"/>
    </inter_packet_times>
    <session_times on_times="0.96856400,0.10000000,0.10000000"
     ↪  off_times="45.30807900,45.30801100" n_packets="11,1,1" n_bytes="1650,66,66"/>
    <packet_sizes n_packets="13" n_kbytes="1">
        <ps_mode1 n_packets="12" n_kbytes="1">
```

```
            <stochastic_model name="constant" aic="inf" bic="inf" param1="99.916667"
             ↪   param2="0.000000"/>
            <stochastic_model name="normal" aic="137.020397" bic="137.990210"
             ↪   param1="99.916667" param2="64.408439"/>
            <stochastic_model name="exponential-me" aic="138.504076" bic="139.473889"
             ↪   param1="0.010008" param2="0.000000"/>
        </ps_mode1>
        <ps_mode2 n_packets="1" n_kbytes="0">
        <stochastic_model name="constant" aic="inf" bic="inf" param1="583.000000"
         ↪   param2="0.000000"/>
        </ps_mode2>
    </packet_sizes>
</flow>
```

The current version of our tool, do not intrinsically supports responsiveness. The underlying traffic generator API can support it (such as Seagull [Seagull: an Open Source Multi-protocol traffic generator 2009]), but we do not generate any mathematical model for this purpose, and this may also stand as a future work we discuss on chapter **??**.

3.1.4 Flow Generator

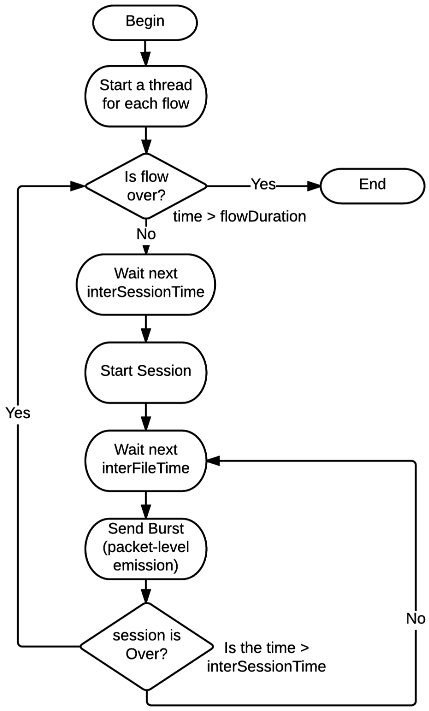

Begin
Start a thread for each flow
Is flow over?
Yes
End
time > flowDuration
No
Wait next interSessionTime
Start Session
Yes
Wait next interFileTime
Send Burst (packet-level emission)
session is Over?
Is the time > interSessionTime
No

Figure 10 – Simplified-harpoon emission algorithm

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

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

This component works in two different layers according to the traffic generators classification introduced in the chapter 2: at the Flow level, and packet level, as presented in the figure 2.

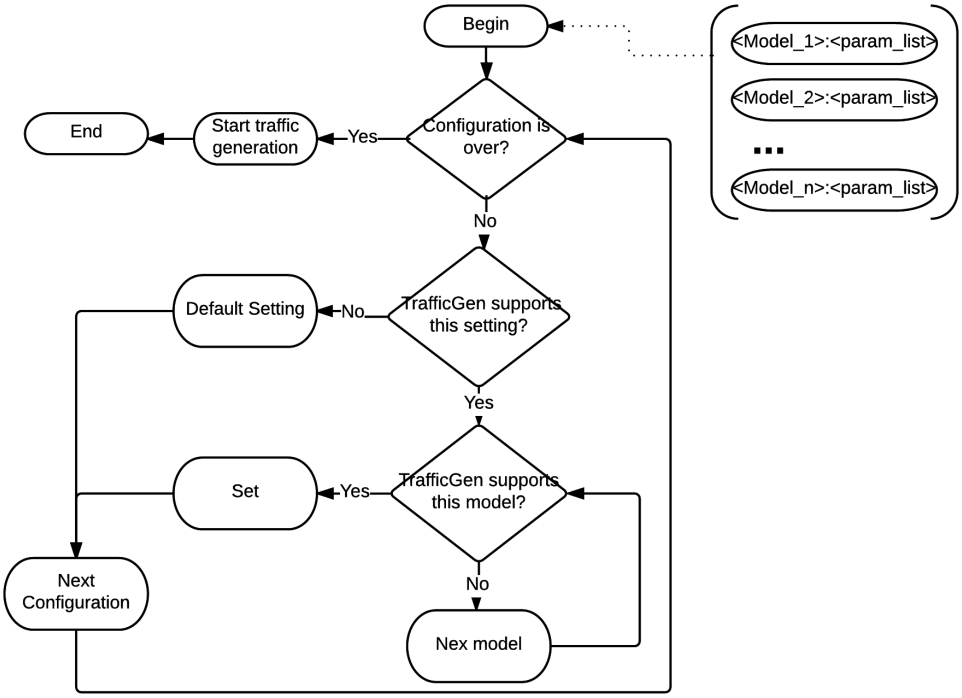


Figure 11 – Traffic engine configuration model

3.1.4.1 Flow-level Traffic Generation Operation

At the flow level, it controls each flow using the algorithm at figure **??**. This algorithm basically handles the our model defined at the figure 8.

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

3.1.4.2 Packet-level Traffic Generation Operation

At the packet level, it must use the expanded class of the `DummyFlow`. In this level, the job of our tool is to configure the underlying engine, to send a file. Therefore it uses the *getters* available, and the passed parameters, defined by the operation described in the previous session. The implementation of this level should follow the flowchart presented in the figure 11. Down below we present a simplistic code of how the D-ITG API can be used to generate packet-level traffic using our framework. A more complex configuration is possible, but it serves to illustrate the procedure. Its API documentation is available at on D-ITG web-site [4].

```cpp
void flowGenerate(const counter& flowId, const time_sec& onTime, const uint& npackets, const uint& nbytes,  const string& netInterface)
{
    char localhost[CHAR_BUFFER];
    string strCommand;
    char command[CHAR_BUFFER];
    uint i = 0;
    int rc = 0;
    getLocalIp("eth0", localhost);
    strCommand += " -t " + std::to_string(onTime);
    strCommand += " -k " + std::to_string(nbytes / 1024);
    strCommand += " -a " + getHostIP();
    if (this->getTransportProtocol() == PROTOCOL__TCP)
        strCommand += " -T TCP -D ";
    else if (this->getTransportProtocol() == PROTOCOL__UDP)
        strCommand += " -T UDP ";
    else if (this->getTransportProtocol() == PROTOCOL__ICMP)
        strCommand += " -T ICMP ";
    for(;;i++)
    {
        StochasticModelFit 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;
        }
    }
    strcpy(command, strCommand.c_str());
    rc = DITGsend(localhost, command); // it is not blocking
    usleep(onTime*10e6);
    if (rc != 0)
    {
        printf("\nDITGsend() return value was %d\n", rc);
        exit(EXIT_FAILURE);
```

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

```
    }
}
```

3.1.5 Network Traffic Generator

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

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

As we claimed before, this tools is simple of being expanded for almost any traffic generation API, since its modeling framework is not coupled to the traffic generation. The figure 12 illustrates it well. It presents three possibilities of different traffic generation engines, all being configured and scheduled in a platform-agnostic way.

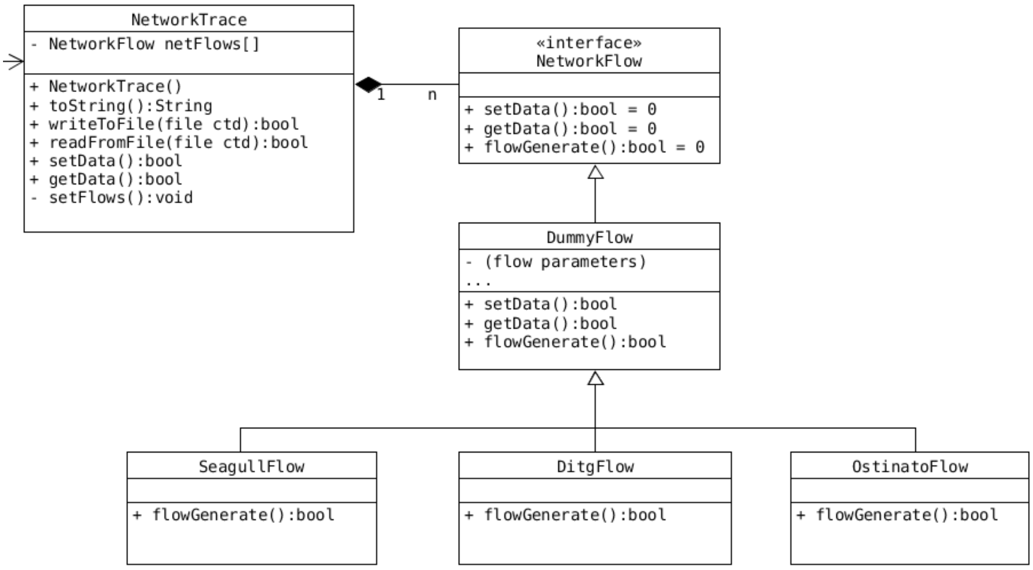


Figure 12 – Class hierarchy of NetworkTrace and NetworkFlow, which enables the abstraction of the traffic generation model of the traffic generation engine.

3.2 Usage and Use Cases

SIMITAR works in four different states: as a sniffer, packet injector, client or server. When it working as a sniffer, is produces as output a Compact Trace Descriptor file. It may operate over a *pcap* file or an Ethernet interface.

When it operates as a client or as packet injector, it is operating as a traffic generator. The difference is that as a client, the used traffic generator is connection-oriented. It needs to establish a connection with the destination host (the server), before start sending packets. It takes as an input a single destination in the command line, or list of many, in a text file passes as a parameter. Also, it uses the machine IP address as packet's source IP.

Operating as a client, it must take as an input a list of IP or MAC destination addresses, since they will not match with the ones in the original trace. As a packet injector can generate packets on the output interface, without establishing a connection. It uses the original IP and MAC destination and source addresses. As a client, the reproduction of the trace is expected to be poor in the flow level metrics exposed in the chapter 2, since it will use new addresses values.

The operation of packet injector is still under implementation, and we are using Libtins library. Currently, SIMITAR supports client mode with D-ITG, implemented using its C++ API. We intend to provide support for many others traffic generators, such Iperf and Ostinato.

When operating as a server, it just start-up the traffic generator server, and waits for server connections. Down bellow, we present simple examples of commands used to activate each operation mode.

```
# Operation as a sniffer over a Ethernet interface named <ether-interface>
./simitar --sniffer -if <ether-interface> --out/-o <cdt-name>
# Operation as a client using the traffic generator D-ITG, using a list of destination hosts of a text file
./simitar --client ditg --in <cdt-xml-name> --dst-list-ip <ip-list-file>
# Operation as a packet injector with uses the Libtins library to craft packets
./simitar --pkt-injection libtins --in <cdt-xml-name>
# Operation as a server. just runs D-ITG in the server mode.
./simitar --server/-sv ditg
```

As use cases, we can cite:

- Generate Background traffic with D-ITG: SIMITAR can work to generate background traffic over a network when it receives many addresses as input. If each destination is reachable, D-ITG will establish a connection with each destination, and recreate flows as described in the CDT file.

- Stress a Device Under Test (DUT): It can be used to stress a single device with a more self-similar realistic traffic. The CDT file can be manually modified to change some aspects of the traffic, such as the number os packets or throughput applied by the traffic generator of each flow. Some undesirable features such as Stochastic models or lesser flow may be taken out, SIMITAR does not have constraints on a number of flows and Stochastic models adopted (but must have at least one).

- Generate synthetic *pcap* files: Using virtual interfaces, we can reproduce a traffic, and capture it with a sniffer tool, such as Tcpdump, Wireshark or Tshark. We plan to automatize this process as a future work, with an operation mode of *pcap* generator. To do this we are going to use Mininet to create the virtual interfaces and Libtins to inject packets.

4 Modeling and Algorithms

4.1 Introduction

There are many works devoted to studying the nature of the Ethernet traffic [Leland *et al.* 1994]. Classic Ethernet models used Poisson related processes to express generation of traffic. Initially, it makes sense since a Poisson process represents the probability of events occur with a known average rate, and independently of the last occurence [Leland *et al.* 1994] [Haight 1967]. But studies made by Leland et al. [Leland *et al.* 1994] showed that the Ethernet traffic has a self-similar and fractal nature. Even if they are able to represent the randomness of an Ethernet traffic, simple Poisson processes can't express traffic "burstiness" in a long-term time scale, such as traffic "spikes" on long-range "ripples". This is an indicative of the fractal and self-similar nature of the traffic, that usually we express by distributions with infinite variance, called heavy-tailed. Heavy-tail means that a stochastic distribution is not exponentially bounded [Varet 2014]. Examples of heavy-tailed functions are Weibull, Pareto, Cauchy. But heavy-tailed function may guarantee self-similarity, but not necessarily they will ensure other features like good correlation and same mean packet rate.

There many consolidate works that investigate the nature of the internet traffic [Leland *et al.* 1994] [Ju *et al.* 2009] [Rongcai e Shuo 2010] [Willinger *et al.* 1997] [Cevizci *et al.* 2006], and many others on the modelling of stochastic functions for specific scenarios [Markovitch e Krieger 2000] [Field *et al.* 2004] [Kushida e Shibata 2002] [Fiorini 1999] [Kronewitter 2006] [Field *et al.* 2004]. But not as many on model choice automatization [Varet 2014].

In this chapter, we propose a method that may be implemented by software. We estimate many stochastic functions through many methodologies and select the best model through the AIC computation [Yang 2005]. Since generating random can be cost and have a bias, depending on the seed generator, we cut these problems since our method is analytical.

We made a brief bibliographic review of some works related works on this topic. Then, we will present the traffic traces we are going to use in this chapter, and in the rest of our work. After we present or selected method for parameterization and fitting choice. To test our criteria quality, we define a validation method, based on cross-validations made by simulations.

4.2 Related Works

There are plenty of works on the literature which proposes new processes and methodologies for modeling times between packets and packet trains.

Fiorini [Fiorini 1999] presents a heavy-tailed ON/OFF model, which tries to represent a traffic generated by many sources. The model emulates a multiple source power-tail Markov-Modulated (PT-MMPP) ON/OFF process, where the ON times are power-tail distributed. They achieve analytical performance measurements using Linear Algebra Queueing Theory.

Kreban and Clearwater [Kleban e Clearwater 2003] presents a model for times between job submissions of multiple users over a super computer. They show that the Weibull probability functions are able to express well small and high values of inter-job submission times. They also tested exponential, lognormal and Pareto distributions. Exponential distribution couldn't represent long-range values because it fell off too fast and Pareto was too slow. Lognormal fit well small values, but was poor on larger ones.

Kronewitter [Kronewitter 2006] presents a model of scheduling traffic of many heavy-tail sources. On his work, he uses many Pareto sources to represent the traffic. To estimate the shape parameter α they use linear regression.

4.3 Methodology

We will start defining the datasets we are going to use in the rest of this work. Then, we define our criteria of parameterization and model choice and a methodology for measure the quality of method.

4.3.1 Datasets

We start defining also define the *pcaps* datasets we are going to use in the rest of this text. We will use for datasets, and for reproduction purposes, three are public available. The first is a lightweight Skype capture, found in Wireshark wiki[1], and can be found at https://wiki.wireshark.org/SampleCaptures. The file name is `SkypeIRC.cap`, and we call it *skype-pcap*.

The second is a CAIDA[2]http://www.caida.org/home/ capture, and can be found at https://data.caida.org/datasets/passive-2016/equinix-chicago/20160121-130000.UTC. Access to this file need login, so you will have to create an account and wait for approval first. The pcap's file name is `equinix-chicago.dirB.20160121-135641.UTC.anon.pcap.gz`. We call it *wan-pcap*.

The third we capture in our laboratory LAN, through a period of 24 hours. It was captured in the firewall gateway between our local and external network. Along with other tests,

[1] https://wiki.wireshark.org/

[2] http://www.caida.org/home/

We intend to verify diurnal behavior on it. That means a high demand of packets during the day and a small in the night. We call it *lan-diurnal-pcap*.

The fourth is a capture of a busy private network access point to the Internet, available on-line on TCPreplay [Tcpreplay home] website [3], and is called `bigFlows.pcap`. We will refer to it *bigflows-pcap*.

4.3.2 Modelling Inter-Packet times and Packet Sizes

We summarize our process of modeling inter-packet times at the figure 9. We collect a set of inter-packet times from an actual traffic capture. Then, we estimate a set of parameters for stochastic functions, using different methodologies. Then, from these parametrized models, we estimate which best represent our data set, using the measure of quality AIC (Akaike information criterion). We also calculate another measure of quality called BIC(Bayesian information criterion), for comparison of results. In this chapter, we present our results obtained on our prototype implemented in Octave.

Currently, we are modeling:

- Weibull distribution, using linear regression, through the Gradient descendant algorithm;
- Normal distribution, using direct estimation the mean and the standard deviation of the dataset;
- Exponential distribution, using linear regression, through the Gradient descendant algorithm. We refer to this distribution as Exponential(LR);
- Exponential distribution, using a direct estimation of the dataset mean. We refer to this distribution as Exponential(Me);
- Pareto distribution, using linear regression, through the Gradient descendant algorithm. We refer to this distribution as Pareto(LR);
- Pareto distribution, using the maximum likelihood method. We refer to this distribution as Pareto(MLH);
- Cauchy distribution, using linear regression, through the Gradient descendant algorithm;

Now we will give a brief explanation our three procedures: Linear Regression (with the Gradient descendant algorithm), direct estimation, and maximum likelihood. We go deep into details explaining the methods on the appendix A, and present the mathematical demonstrations on appendix **??**.

[3] http://tcpreplay.appneta.com/wiki/captures.html

Some observations must be made. Since the time samples resolution used were of 10^{-6}s, all values equal to zero were set to $5 \cdot 10^{-8}$s, to avoid division by zero. To avoid divergence on tangent operation on the linearization the Cauchy function, the inter-packets CDF function values were floor-limited and upper-limited to 10^{-6} and 0.999999 respectively.

We implemented this prototype using Octave. We upload the code on GitHub, for reproduction purposes[4]

4.3.2.1 Linear regression (Gradient descendant)

Linear regression is a method for estimating the best linear curve in the format:

$$y = ax + b \tag{4.1}$$

to fit a given data set. We can use linear regression to estimate parameters of a non-linear curve expressing it on a linear format. For example, the Weibull CDF for $t > 0$ is:

$$F(t|\alpha,\beta) = F(t) = 1 - e^{-(t/\beta)^{\alpha}} \tag{4.2}$$

Manipuling the equation:

$$\alpha \ln(t) - \alpha \ln(\beta) = \ln - \ln 1 - F(t) \tag{4.3}$$

If we call $x = \ln(t)$ and $y = \ln(-\ln(1 - F(t)))$, we found a linear equation, where $a = \alpha$ and $b = -\alpha \ln(\beta)$. Having in hands a estimation of the empirical CDF of our data samples, we apply the x and y definitions to linearize the data. Using the gradient descendant, we are able to find a estimation of the linear coefficients $\hat{a}$ and $\hat{b}$. Using the inverse function of the definition of linear coefficients, we are able to find the estimated parameters $\hat{\alpha}$ and $\hat{\beta}$.

$$\alpha = a \tag{4.4}$$

$$\beta = e^{-(b/a)} \tag{4.5}$$

We present examples of linearized data, and the linear approximation achieved, as well the cost convergence of the gradient descendant algorithm in the appendix A.

Applying the inverse equations of the linear coefficients ($\hat{\alpha} = \hat{a}$ and $\hat{\beta} = e^{-(\hat{b}/\hat{a})}$) (the symbol stands for the estimated parameters). So we are able to estimate the Weibull distribution parameters.

We can summarize this procedure, in these steps:

[4] github-link-aqui

1. Linearize the stochastic CDF function F(t).
2. Apply the linearized $y = y(F(t))$ and $x = x(t)$ on the empirical CDF and times datasets, respectively.
3. Use Gradient Descendant algorithm to find linear coefficients a and b.
4. Apply the inverse equation of the linear coefficients, to determine the stochastic function parameters.

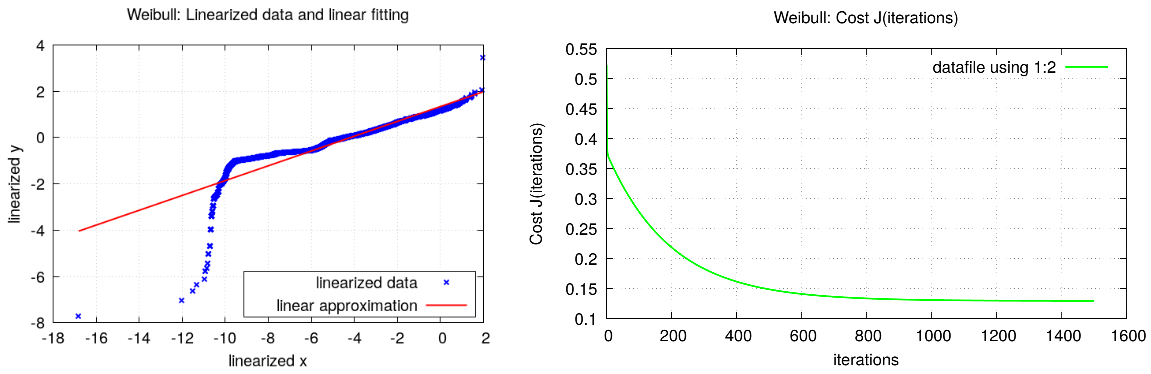
In the parameters estimation, there is an exception, since the Pareto scale (t_m) is defined by the minimum time.

In the table 7 we present a summary of the used equations in the procedure. In this notation, the subscript i means that it must be applied on every empirical value measured. The hat($\widehat{\ }$) indicates an estimated value for a parameter.

In the figure 13a we presente the linearized

Table 7 – Linearized functions, and parameters estimators, used by the linear regression

Function	Linearized x	Linearized y	Parameters Estimator	
Cauchy	$x_i = t_i$	$y_i = \tan(\pi(F(t_i) - 1/2))$	$\hat{\gamma} = \frac{1}{\hat{a}}$	$\hat{t_0} = -\frac{\hat{b}}{\hat{a}}$
Exponential	$x_i = t_i$	$y_i = \ln(1 - F(t_i)))$	$\hat{\lambda} = -\hat{a}$	
Pareto	$x_i = \ln(t_i)$	$y_i = \ln(1 = F(t_i))$	$\hat{\alpha} = -\hat{a}$	$\hat{x_m} = \min_{i=0,\dots,m}\{x_i\}$
Weibull	$x = \ln(t)$	$y = \ln(-\ln(1 - F(t)))$	$\hat{\alpha} = \hat{a}$	$\hat{\beta} = e^{-(\hat{b}/\hat{a})}$



(a) c

(b) c

4.3.2.2 Direct Estimation

The expected value $E(X)$ and variance $Var(X)$ of a random variable X of some distributions are close related with its parameters. Since the mean $\bar{\mu}$ and its standard deviation $\bar{\sigma}$ are in general good approximations for the expected value and variance, we use them to estimate parameters.

Following the notation presented at table 4, we have for the normal distribution:

$$(\hat{\mu}, \hat{\sigma} = (\bar{\mu}, \bar{\sigma}) \tag{4.6}$$

For the exponential distribution $E(X) = \frac{1}{\lambda}$, therefore:

$$\hat{\lambda} = \frac{1}{\hat{\mu}} \tag{4.7}$$

4.3.2.3 Maximum Likelihood

The maximum likelihood estimation, is a method for estimation of parameters, winch maximizes the likelihood function. We explain in details this subject in the Appendix A. Using this method for the Pareto distribution, it is possible to derive the following equations the its parameters:

$$\hat{x_m} = \min_{i=0,...,m} \{x_i\} \tag{4.8}$$

$$\hat{\alpha} = \frac{n}{\sum_{i=0}^{m}(\ln(x_i) - \ln(\hat{x_m}))} \tag{4.9}$$

where m is the sample size.

4.3.2.4 AIC and BIC

Suppose that we have an statistical model M of soma dataset $x = (x_1, ..., x_n)$, with n independent and identically distributed observations of a random variable X. This model can be expressed by a PDF $f(x, \theta)$, where θ a vector of parameter of the PDF, $\theta \in \mathbb{R}^k$ (k is the number of parameters). The likelihood function of this model M is given by:

$$L(\theta|x) = f(x_1|\theta)...f(x_n|\theta) = \prod_{i=1}^{n} f(x_i|\theta) \tag{4.10}$$

Now, suppose we are trying to estimate the best statistical model, from a set $M_1, ..., M_n$, each one whit an estimated vector of parameters $\hat{\theta_1}, ..., \hat{\theta_n}$. AIC and BIC are defined by:

$$AIC = 2k - \ln(L(\hat{\theta}|x)) \tag{4.11}$$

$$BIC = k\ln(n) - \ln(L(\hat{\theta}|x)) \tag{4.12}$$

In both cases, the preferred model M_i, is the one with the smaller value of AIC_i or BIC_i.

4.3.2.5 Validation

To see if our criterion of parameter selection is actually is able to find which is the best model, we define a validation methodology. We generate a vector with the same size form the original, with random generated data through our model estimation. Then we compare it with the original sample, trough four different metrics, all with a confidence interval of 95%:

- Correlation between the sample data and the estimated model;
- Hurst exponent;
- Mean inter packet time;
- Standard deviation of inter packet times.

The Pearson's product-moment coefficient, or simply correlation coefficient, is an expression of the dependence or association between two datasets. Its value goes from -1 to +1. +1 means a perfect direct linear correlation. -1 indicates perfect inverse linear correlation. 0 means no linear correlation. So, as close the result reach 1, more similar are the inter-packet times to the original values. To estimate it, we use the Octave's function `corr()`.

As explained before in the chapter 2, the Hurst exponent is meter self-similarity and indicates the fractal level of the inter-packet times. As close the result is from the original, more similar is the fractal level of the estimated samples from the original.To estimate this value we use the function `hurst()` from Octave, which uses rescaled range method. Finally we measure the mean and the standard deviation (as a measure of the dispersion), using the Octave's functions `mean()` and `std()`. We also present some *QQplots*, to visually compare the random-generated data and the original dataset.

As close the correlation, Hurst exponent, mean and the standard deviation is from the original dataset, the better is model fitting. Also, analyzing the mean, we can see if a certain modeling procedure tends to be more penalized for the values close, or far from zero. This means that if the mean inter-packet time tends to be smaller or higher compared to the original.

With these results in hands, we can see if AIC and BIC are good criteria for model selection for inter-packet times. To quantitatively check it, we define a cost function based on the correlation, Hurst exponent and mean. We exclude the standard deviation, because the Hurst exponent being a meter of the fractal level, also capture information about the desired dispersion of the data. So, for comparing all these results, we defined a cost function J, based on the randomly generated data values.

Being Cr the vector of correlations ordered from the greater to the smaller. Let Me and Hr defined by the absolute difference between mean and hurt exponent of the estimated values and the original dataset. Both are ordered from the smaller to the greatest values. Letting

$\phi(V,M)$ be an operator wich gives the position of a model M in a vector V, we define the cost function J as:

$$J(M) = \phi(Cr,M) + \phi(Me,M) + \phi(Hr,M) \tag{4.13}$$

The smaller is the cost J, the best is the model. Then we compare the results achieved by AIC and BIC, and J.

4.4 Results

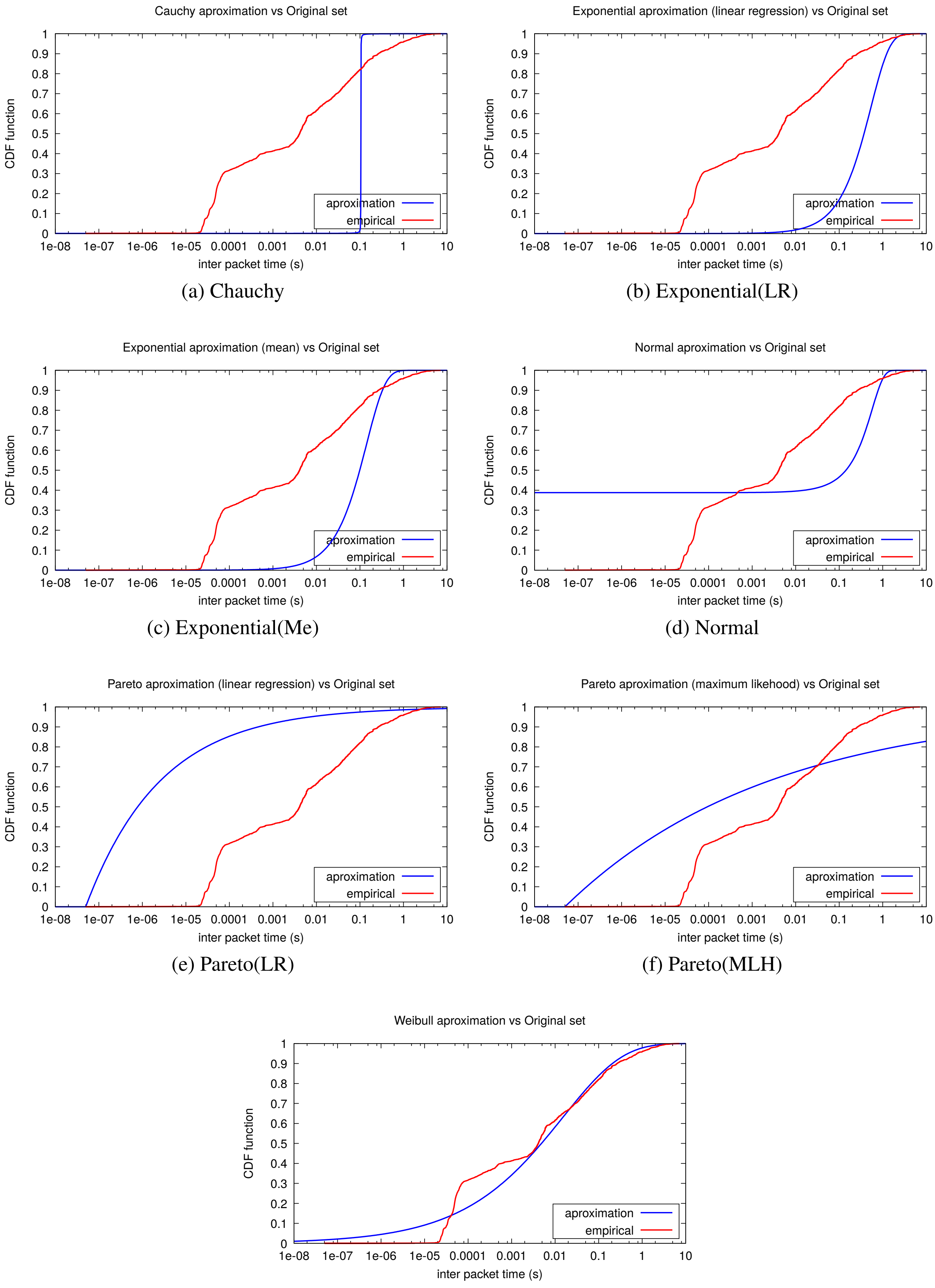
Table 8 – Results of the octave prototype, include BIC and AIC values, para estimated parameters for two of our pcap traces: *skype-pcap* and *bigflows-pcap*. *bigflows-pcap* is much larger, and has a much much smaller mean inter-packet time

	Trace							
Function	AIC	BIC	Parameters		AIC	BIC	Parameters	
	skype-pcap				lan-diurnal-pcap			
Cauchy	$2.18e4$	$2.19e4$	$\gamma: 2.59e-4$	$x_0: 1.05e-1$	$-2.85e7$	$-2.85e7$	$\gamma: 9.63e-3$	$x_0: -3.61e-3$
Exponential(LR)	$-1.61e3$	$-1.60e3$	$\lambda: 1.86$		$1.79e6$	$1.79e6$	$\lambda: 8.51e-1$	
Exponential(Me)	$-4.29e3$	$-4.28e3$	$\lambda: 7.01$		$-3.12e7$	$-3.12e7$	$\lambda: 58.78$	
Normal	$3.29e3$	$3.31e3$	$\mu: 1.43e-1$	$\sigma: 5.01e-1$	Inf	Inf	$\mu: 1.70e-2$	$\sigma: 8.56e-2$
Pareto(LR)	$-8.28e3$	$-8.27e3$	$\alpha: 2.52e-1$	$x_m: 5e-8$	$-4.60e7$	$-4.60e7$	$\alpha: 2.55e-1$	$x_m: 5e-8$
Pareto(MLH)	$-1.16e4$	$-1.16e4$	$\alpha: 9.21e-2$	$x_m: 5e-8$	$-5.03e7$	$-5.03e7$	$\alpha: 1.15e-1$	$x_m: 5e-8$
Weibull	$-1.46e4$	$-1.46e4$	$\alpha: 3.20e-1$	$\beta: 1.52e-2$	$-5.60e7$	$-5.60e7$	$\alpha: 3.34e-1$	$\beta: 1.83e-3$
	bigFlows-pcap				wan-pcap			
Cauchy	$7.14e6$	$7.14e6$	$\gamma: 1.94e0$	$x_0: -7.25$	$5.99e7$	$5.99e7$	$\gamma: 8.28e2$	$x_0: -4.52e3$
Exponential(LR)	$7.33e6$	$7.33e6$	$\lambda: 1.489e-1$		$5.68e7$	$5.68e7$	$\lambda: 2.2e-5$	
Exponential(Me)	$-1.09e7$	$-1.09e7$	$\lambda: 2.64e3$		$-6.58e7$	$-6.58e7$	$\lambda: 6.58e5$	
Normal	$-9.35e6$	$-9.35e6$	$\mu: 3.79e-4$	$\sigma: 6.60e-4$	$-6.39e7$	$-6.39e7$	$\mu: 2e-6$	$\sigma: 1e-6$
Pareto(LR)	$-1.02e7$	$-1.02e7$	$\alpha: 1.489e-1$	$x_m: 5e-8$	$-5.31e7$	$-5.31e7$	$\alpha: NaN$	$x_m: 5e-8$
Pareto(MLH)	$-1.03e7$	$-1.03e7$	$\alpha: 1.362e-1$	$x_m: 5e-8$	$-6.25e7$	$-6.25e7$	$\alpha: 3.39e-1$	$x_m: 5e-8$
Weibull	$-1.10e7$	$-1.10e7$	$\alpha: 2.81e-1$	$\beta: 5.54e-4$	$-5.46e7$	$-5.46e7$	$\alpha: 7.64e-2$	$\beta: 1e-6$

The results for the parameters estimation and AIC and BIC for *skype-pcap* and *bigflows-pcap* are at the table 8. The results for *wan-pcap* and *lan-diurnal-pcap* are on the table **??**. The difference between the values of BIC and AIC in all simulations were very small. Much smaller then the difference between the values found for each distribution. This is an indication that for our type of dataset, using AIC or BIC should will not influence significantly the results.

On our previsions, Weibull followed Pareto are the functions more indicated as good options. This was expected, since both are heavy-tailed functions. But, Cauchy, on most of the tests, seems to no present a good fitting. This is effect of the fast divergence of tangent function, when we linearize our data. Not letting the CDF values be so close to zero or avoided divergence, but still through linear regression the penalty was heavy.

In the figure **??** we present all estimated CDF functions along with the empirical CDF, for the trace *skype-pcap*. In the case of the normal function, all values smaller than zero, were set to zero on the plot. Is possible to see different accuracies on each plot, and types of fittings. Visually, the best fit seems to be the Weibull trough linear regression. In the figure **??**



(a) Chauchy

(b) Exponential(LR)

(c) Exponential(Me)

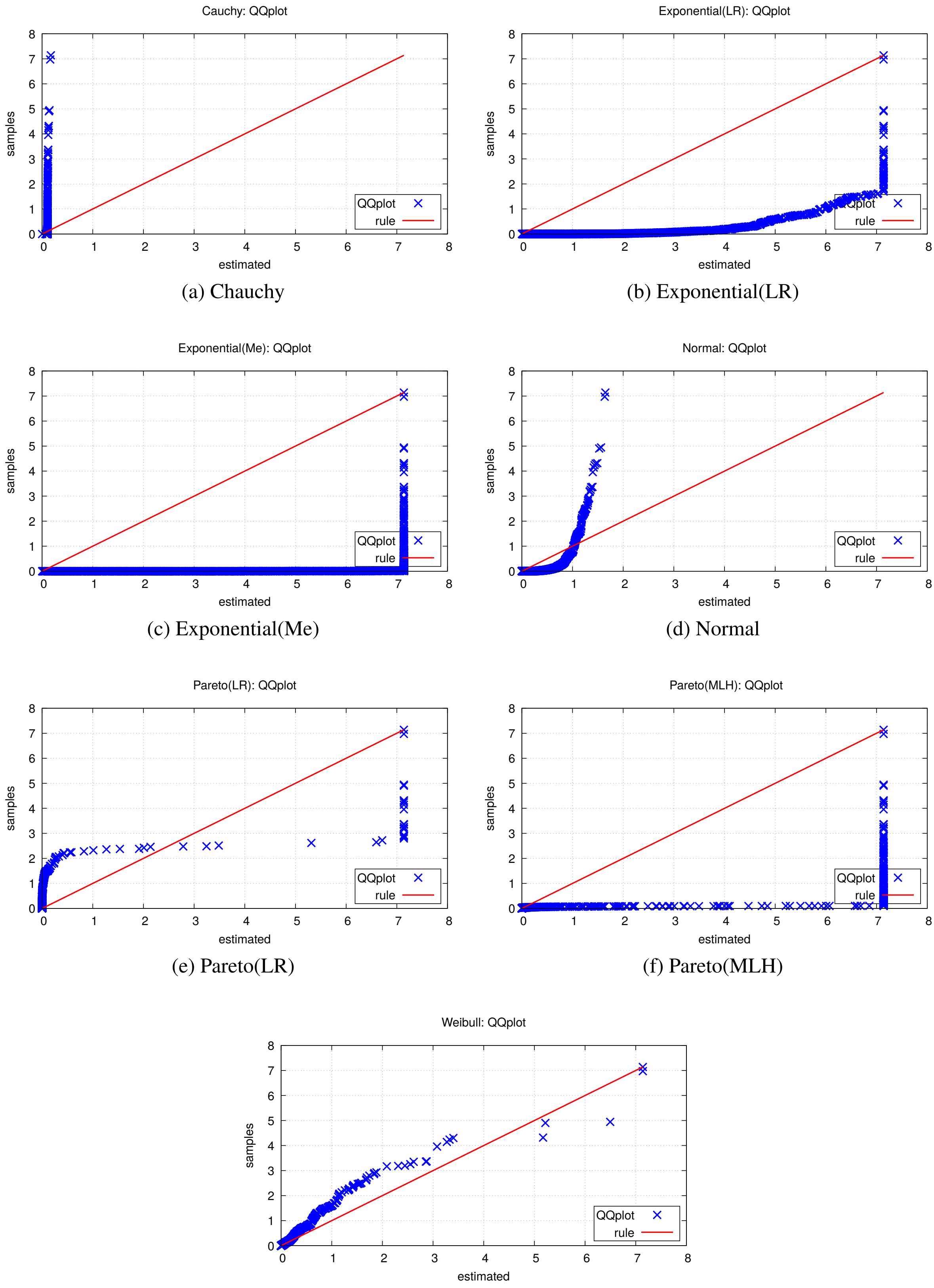
(d) Normal

(e) Pareto(LR)

(f) Pareto(MLH)

(g) Weibull

Figure 13 – CDF functions for the approximations of *skype-pcap* inter packet times, of many stochastic functions.



(a) Chauchy

(b) Exponential(LR)

(c) Exponential(Me)

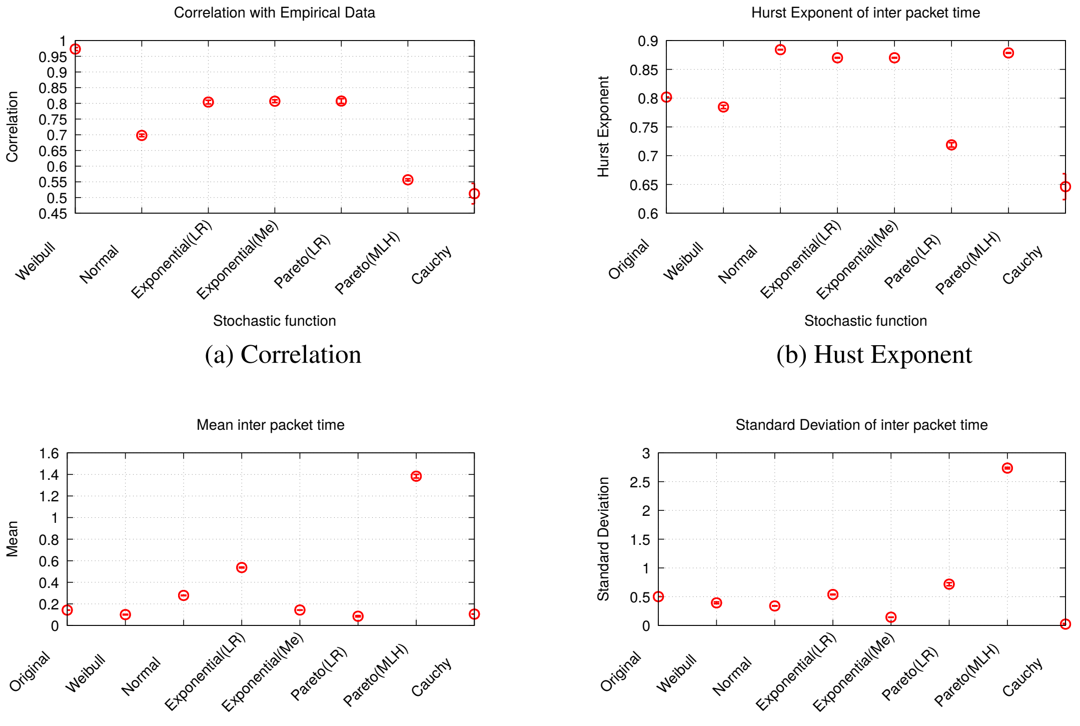
(d) Normal

(e) Pareto(LR)

(f) Pareto(MLH)

(g) Weibull

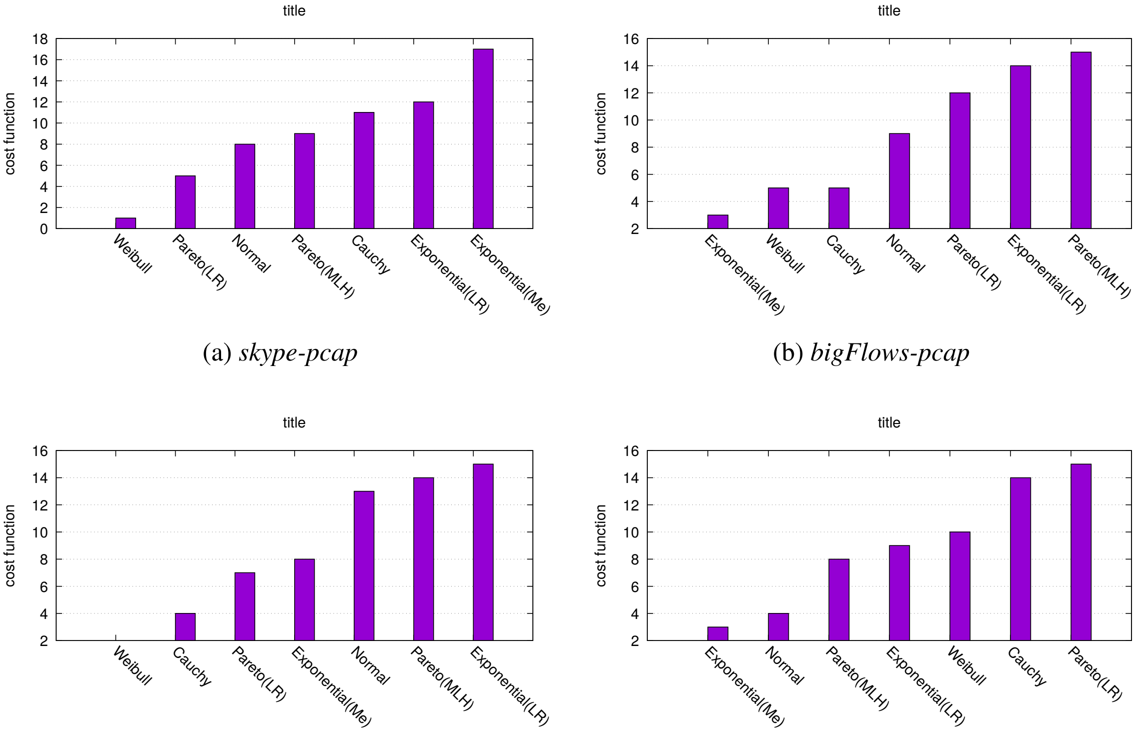
Figure 14 – CDF functions for the approximations of *skype-pcap* inter packet times, of many stochastic functions.



(a) Correlation (b) Hust Exponent

(c) Mean (d) Standard Deviation

Figure 15 – Statistical parameters of *skype-pcap* and its approximations



(a) *skype-pcap* (b) *bigFlows-pcap*

(c) *lan-diurnal-pcap* (d) *wan-pcap*

Figure 16 – Cost function for each one of the datasets used in this validation process

we present the fittings with higher BIC and AIC values for the *bigflows-pcap*. On our previsions. Analyzing the plots, and what they would mean, our Cauchy fitting would impose an almost contant traffic , with the inter packet time close to the mean. On the trace *skype-pcap* the exponential plots seems to not represent well the small values of inter packet times. On the other hand, on *bigflows-pcap* they are expressed much well. This is due fact that an exponential process is good at representing values close to the mean. But, it fails to represent values too small and too higher. The *bigflows-pcap* has a much smaller dispersion. On the other hand, a self-similar process like Weibull and Pareto are better representing inter packet times with higher dispersion.

Analyzing the quality of AIC and BIC as criterion of choose on *skype-pcap*, we see that in therms of Correlation and Self-similarity it picket the right model: Weibull. Also in therms of mean packet rate and dispersion, is still a good choice, since it is still one of the best choices. The third and the fourth choices (Pareto(LR)) and Exponential(LR) also are good options in most of these metrics. But, Pareto(MLH) is presented as the second best choice, and it had poor results in comparison to the others, especially on mean, correlation and dispersion. In this example, we illustrate it at the table **??**.

The trace *bigflows-pcap* has a much higher inter packet rate, and a smaller dispersion. Weibull is the model chosen by AIC/BIC, and it is the second best on correlation and self-similarity (Exponential(Me) and Cauchy are the best respectively). Exponential(Me) chooses as the second preferable, has the closest to the original mean and standard deviation. Comparing the plots of the CDF function, the Weibull seams to express well both small and larger values of inter packet times, while Exponential(Me) is preciser on the mean values, as we can see at figure **??**.

Algorithm 1 calcOnOff

```
function CALCONOFF($arrivalTime, deltaTime, cutTime, minOnTime$)
    $m = deltaTime.length() - 1$
    $j = 0$
    $lastOff = 0$
    $pktCounterSum = 0$
    $fileSizeSum = 0$
    for $i = 0 : m$ do
        $pktCounterSum = pktCounterSum + 1$
        $fileSizeSum = fileSizeSum + psSizeList[i, 1]$
        if $i == 1$ then                                      ▷ if the first is session-off time
            $j{+}{+}$
            $onTimes.push(minOnTime)$
            $offTimes.push(deltaTime[i])$
            $pktCounter.push(pktCounterSum)$
            $fileSize.push(fileSizeSum)$
            $pktCounterSum = 0$
            $fileSizeSum = 0$
        else                                                  ▷ base case
            $pktCounter.push(pktCounterSum)$
            $fileSize.push(fileSizeSum)$
            $pktCounterSum = 0$
            $fileSizeSum = 0$
            if $j == 0$ then
                $onTimes.push(arrivalTime[i-1])$
                $offTimes.push(deltaTime[i])$
            else                                              ▷ others on times
                $onTimes.push(max(deltaTime[i-1] - deltaTime[lastOff], minOnTime))$
                $offTimes.push(deltaTime[i])$
            end if
            $lastOff = i$
        end if
    end for
    $pktCounterSum = pktCounterSum + 1$
    $fileSizeSum = fileSizeSum + psSizeList[m]$
    if $lastOff == m - 1$ then                                ▷ if last is session-off
        $onTimes.push(minOnTime)$
    else                                                      ▷ base last case
        if $lastOff \neq 0$ then
            $onTimes.push(arrivalTime[m] - arrivalTime[lastOff])$
        else
            $onTimes.push(arrivalTime[m])$                    ▷ there was just on time
        end if
    end if
    $pktCounter.push(pktCounterSum)$
    $fileSize.push(fileSizeSum)$
end function
```

Table 9 – Application classification table

Application Protocol	Transport Protocols	Transport Ports
HTTPS	TCP	443
FTP	TCP	20, 21
HTTP	TCP	80
BGP	TCP	179
DHCP	UDP	67, 68
SNMP	UDP, TCP	161
DNS	UDP, TCP	53
SSH	UDP, TCP	22
Telnet	UDP, TCP	23
TACACS	UDP, TCP	49

4.5 Others Algorithms

4.5.1 On/Off Times estimation

4.5.2 Application classification

4.6 Implementation Details

4.6.1 Modeling Process

4.6.2 Matlab prototyping and C++ Implementation

4.6.3 Software Engineering process and Lessons Leaned

4.7 Conclusion

We can conclude by this evaluation, that our chosen methodology for selecting stochastic models for inter packet data is appropriated. The first thing to observe is that, for a high

First, we do not rely on only one type of parametrization. This turns our methodology more robust, since a linear regression may diverge. But aways there will be an packable model, since our estimations for Normal, Exponential(Me) and Pareto(MLH) cannot. Models wich the linear regression diverge, will not have god BIC/AIC estimations, and will not be picked.

5 Proof of Concepts and Validation

5.1 Testbeds

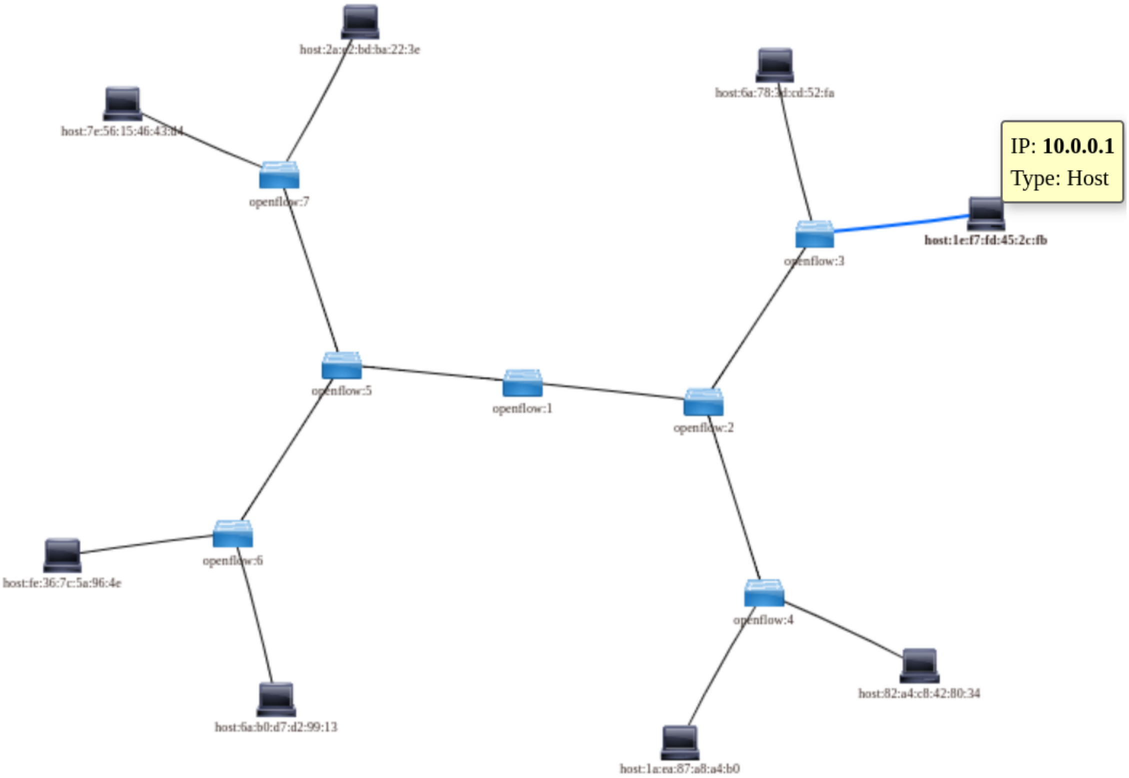


Figure 17 – xxxx

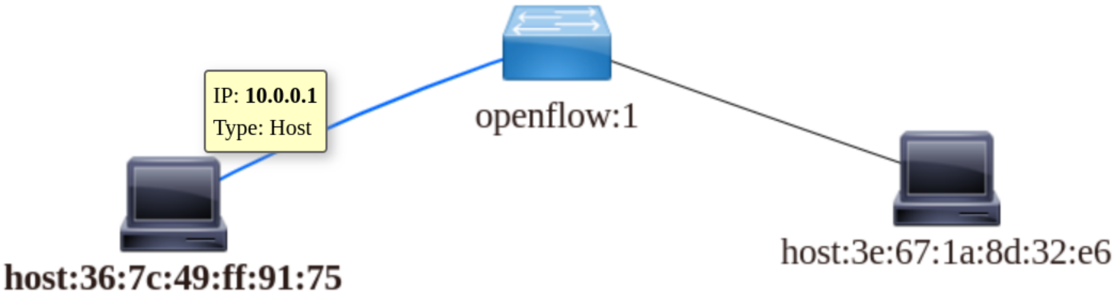


Figure 18 – xxxx

5.2 Scalling Evaluation

5.3 Packet and Flow Metrics Evaluation

6 Conclusion and Future Work

6.1 Future Work

6.1.1 Melhorando resultados

6.1.2 Calibração das ferramentas

Para extensão do trafego gerado, é necessário um estudo da melhor calibração para cada ferramente, como o D-ITG, Iperf, Libtins, o que pode melhorar os resultados

6.1.3 Improoving performance

6.1.3.1 Improving database storage

basicamente eu vou aqui propor um novo modelo de database mais eficiente, no qual menos dados serão arnmazenados

6.1.3.2 Improving DataProcessor Performance

Melhorar os algoritimos adicionando técnicas adicionais de modelagem, como condição de parada para o algoritimo gradient descendent, stochastic gradient descendent, entre outros

6.1.3.3 Sniffer baseado em SDN Switch

Isso mudaria o tipo de dado adiquirido, porém os algorítimos implementados ainda poderiam ser utilizados, já que eles trabalham com qualquer tipo de dado. Haveriam mudanças em como eles seriam tratados pelo data processor

6.1.3.4 Implementação

- Sniffer em C++ para melhorar a performance - Network Trace com uma API em Python

6.1.3.5 Kernel bypass usando o DPDK

Automatizar a criação de KKNI interfaces pelo FlowGenerator, o que possibilitaria kernel bypass na geração do trafego, melhorando a performance

6.1.4 Novos trabalhos

6.1.4.1 PcapGen

Mininet+TCPdump+LibtinsFlow=> possibilidade de captura de trafego gerado em uma interface virtual, gerando portante traces sintéticos. Isso possibilitaria a criação de uma biblioteca de traces, para aplicações de banchmark como o NFPA.

6.1.4.2 Extender para novas ferramentas e bibliotecas

Ampliar para o moongen, ostinato, Seagull, NetFPGA, etc

6.1.5 Benchmarking

Oferecer a possibilidade de automatização de medições e testes extendendo a ferramenta

Bibliography

K. Kant and V. Tewari and R. Iyer. https://nmap.org/nping/ . [Online; accessed May 14th, 2016].

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.

ANTICHI, G.; PIETRO, A. D.; FICARA, D.; GIORDANO, S.; PROCISSI, G.; VITUCCI, F. Bruno: A high performance traffic generator for network processor. In: *2008 International Symposium on Performance Evaluation of Computer and Telecommunication Systems*. [S.l.: s.n.], 2008. p. 526–533.

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.

D-ITG, Distributed Internet Traffic Generator. 2015. http://traffic.comics.unina.it/software/ITG/. [Online; accessed May 14th, 2016].

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>.

FENG, W.-c.; GOEL, A.; BEZZAZ, A.; FENG, W.-c.; WALPOLE, J. Tcpivo: A high-performance packet replay engine. In: *Proceedings of the ACM SIGCOMM Workshop on Models, Methods and Tools for Reproducible Network Research*. New York, NY, USA: ACM, 2003. (MoMeTools '03), p. 57–64. ISBN 1-58113-748-6. Disponível em: <http://doi.acm.org/10.1145/944773.944783>.

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.

FIORINI, P. M. On modeling concurrent heavy-tailed network traffic sources and its impact upon qos. In: *1999 IEEE International Conference on Communications (Cat. No. 99CH36311)*. [S.l.: s.n.], 1999. v. 2, p. 716–720 vol.2.

GEN_SEND, gen_recv: A Simple UDP Traffic Generater Application. http://www.citi.umich.edu/projects/qbone/generator.html. [Online; accessed January 11th, 2017].

GENSYN - generator of synthetic Internet traffic. http://www.item.ntnu.no/people/personalpages/fac/poulh/gensyn. [Online; accessed May 14th, 2016].

GETTING Started with Pktgen. 2015. http://pktgen.readthedocs.io/en/latest/getting_started.html. [Online; accessed May 14th, 2016].

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>.

HAIGHT, F. A. *Handbook of the Poisson Distribution*. New York: John Wiley & Son, 1967.

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.

HTTPERF(1) - Linux man page. https://linux.die.net/man/1/httperf. [Online; accessed January 11th, 2017].

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>.

IPERF. 2016. https://sourceforge.net/projects/iperf/files/. [Online; accessed May 14th, 2016].

IPERF - The network bandwidth measurement tool. https://iperf.fr/. [Online; accessed May 14th, 2016].

JPERF. 2015. https://github.com/AgilData/jperf. [Online; accessed May 14th, 2016].

JU, F.; YANG, J.; LIU, H. Analysis of self-similar traffic based on the on/off model. In: *2009 International Workshop on Chaos-Fractals Theories and Applications*. [S.l.: s.n.], 2009. p. 301–304.

KANT, K.; TEWARI, V.; IYER, R. Geist: a generator for e-commerce internet server traffic. In: *2001 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS*. [S.l.: s.n.], 2001. p. 49–56.

KHAYARI, R. E. A.; RUCKER, M.; LEHMANN, A.; MUSOVIC, A. Parasyntg: A parameterized synthetic trace generator for representation of www traffic. In: *Performance Evaluation of Computer and Telecommunication Systems, 2008. SPECTS 2008. International Symposium on*. [S.l.: s.n.], 2008. p. 317–323.

KLEBAN, S. D.; CLEARWATER, S. H. Hierarchical dynamics, interarrival times, and performance. In: *Supercomputing, 2003 ACM/IEEE Conference*. [S.l.: s.n.], 2003. p. 28–28.

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.

KRONEWITTER, F. D. Optimal scheduling of heavy tailed traffic via shape parameter estimation. In: *MILCOM 2006 - 2006 IEEE Military Communications conference*. [S.l.: s.n.], 2006. p. 1–6. ISSN 2155-7578.

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.

KUTE – Kernel-based Traffic Engine. 2007. http://caia.swin.edu.au/genius/tools/kute/. [Online; accessed May 14th, 2016].

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].

MARKOVITCH, N. M.; KRIEGER, U. R. Estimation of the renewal function by empirical data-a bayesian approach. In: *Universal Multiservice Networks, 2000. ECUMN 2000. 1st European Conference on*. [S.l.: s.n.], 2000. p. 293–300.

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.

MOONGEN. https://github.com/emmericp/MoonGen. [Online; accessed May 14th, 2016].

MULTI-GENERATOR (MGEN). http://www.nrl.navy.mil/itd/ncs/products/mgen . [Online; accessed May 14th, 2016].

MXTRAF. http://mxtraf.sourceforge.net/. [Online; accessed January 11th, 2017].

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

NETSPEC – A Tool for Network Experimentation and Measurement. http://www.ittc.ku.edu/netspec/. [Online; accessed May 14th, 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].

PKTGEN. 2009. http://www.linuxfoundation.org/collaborate/workgroups/networking/pktgen. [Online; accessed May 14th, 2016].

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

ROLLAND, C.; RIDOUX, J.; BAYNAT, B. Litgen, a lightweight traffic generator: Application to p2p and mail wireless traffic. In: *Proceedings of the 8th International Conference on Passive and Active Network Measurement*. Berlin, Heidelberg: Springer-Verlag, 2007. (PAM'07), p. 52–62. ISBN 978-3-540-71616-7. Disponível em: <http://dl.acm.org/citation.cfm?id=1762888.1762896>.

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.

RUDE & CRUDE. 2002. http://rude.sourceforge.net/. [Online; accessed December 12th, 2016].

SEAGULL – Open Source tool for IMS testing. 2006. http://gull.sourceforge.net/doc/WP_Seagull_Open_Source_tool_for_IMS_testing.pdf. [Online; accessed May 14th, 2016].

SEAGULL: an Open Source Multi-protocol traffic generator. 2009. http://gull.sourceforge.net/. [Online; accessed May 14th, 2016].

SHORT User's guide for Jugi's Traffic Generator (JTG). http://www.netlab.tkk.fi/ jmanner/jtg/Readme.txt. [Online; accessed January 11th, 2017].

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>.

SOMMERVILLE, I. *Software Engineering*. Addison-Wesley, 2007. (International computer science series). ISBN 9780321313799. Disponível em: <https://books.google.com.br/books?id=B7idKfL0H64C>.

SOURCESONOFF. http://www.recherche.enac.fr/ avaret/sourcesonoff. [Online; accessed December 19th, 2016].

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].

TCPIVO: A High Performance Packet Replay Engine. http://www.thefengs.com/wuchang/work/tcpivo/. [Online; accessed May 14th, 2016].

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

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

THE Swing Traffic Generator. 2016. http://cseweb.ucsd.edu/ kvishwanath/Swing/. [Online; accessed May 14th, 2016].

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.

VEITCH, P.; MCGRATH, M. J.; BAYON, V. An instrumentation and analytics framework for optimal and robust nfv deployment. *IEEE Communications Magazine*, v. 53, n. 2, p. 126–133, Feb 2015. ISSN 0163-6804.

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.

WELCOME to BRUTE homepage! 2003. http://wwwtlc.iet.unipi.it/software/brute/ . [Online; accessed May 14th, 2016].

WELCOME to the Netperf Homepage. http://www.netperf.org/netperf/. [Online; accessed May 14th, 2016].

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

APPENDIX A – Revision of Probability

A.1 Stochastic Process

A.2 Random variable

A.3 Probability Distribution Function (PDF)

A.4 Cumulative Distribution Function (CDF)

A.5 Expected value, Variance, Mean and Standard Deviation

A.6 Self-similarity

A.7 Heavy-tailed distributions

A.8 Hurst Exponent

A.9 Maximum likelihood function

A.10 Akaike information criterion

A.11 Bayesian information criterion

A.12 Estimation of parameters of stochastic distributions

A.12.1 Linear Regression and Gradient Descendant Algorithm

A.12.2 Maximum likelihood estimation

A.12.3 Others methods

APPENDIX B – UML Project Diagrams

B.1 Class Diagrams

B.2 Sequence Diagrams

APPENDIX C – TODO List

C.1 Chapter 1 - Introduction - Introduction

C.2 Chapter 2 - Bibliographic Revision

Liks úteis:

» http://www.backtrack-linux.org/forums/showthread.php?t=8740

» https://sudks.wordpress.com/2016/04/27/tools-traffic-networking/

» http://www.linuxlinks.com/article/20120428060904 28/BenchmarkTools.html

» http://uperf.org/

1. TODO **Geist** [Kant *et al.* 2001] is an application level traffic generator for stress web servers. http://kkant.net/geist/ http://kkant.net/geist/doc/geist_doc.pdf
2. TODO https://wiki.fd.io/view/Project_Proposals/TRex https://github.com/cisco-system-traffic-generator/trex-core
3. TODO TFGEN/MCAST: http://www.pgcgi.com/hptools/
4. TODO Poisson traffic generator http://www.spin.rice.edu/Software/poisson_gen/
5. TODO http://netsniff-ng.org/ (trafgen)
6. TODO Synthetic self-similar traffic generation http://glenkramer.com/trf_research.shtml
7. TODO https://github.com/steerapi/udpgen
8. TODO http://nutsaboutnets.com/netstress/
9. TODO https://sourceforge.net/projects/ip-packet/
10. TODO PacGen http://sourceforge.net/projects/pacgen/ Packet Forger
11. TODO NTGen http://nssl.eew.technion.ac.il/files/Projects/NTGen_v2.0/html/index.htm
12. TODO epb - Ethernet Package Bombardier http://maz-programmersdiary.blogspot.com.br/2012/05/ep ethernet-package-bombardier.html
13. (Programa não encontrado) FTP traffic generator: https://pdfs.semanticscholar.org/5756/cf54e46c38d
14. (Programa não encontrado) **NetSpec**: http://www.cs.columbia.edu/ hgs/internet/traffic-generator.html http://www.ittc.ku.edu/netspec/

C.3 Chapter 3 - Architecture

C.4 Chapter 4 - Modeling

C.5 Chapter 5 - Prof of concepts

C.6 Chapter 6 - Usage cases

C.7 Chapter 7 - Conclusion