UNICAMP

UNIVERSIDADE ESTADUAL DE CAMPINAS

Faculdade de Engenharia Elétrica e de Computação

Anderson dos Santos Paschoalon

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

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

CAMPINAS

2017

Anderson dos Santos Paschoalon

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

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

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

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

Supervisor: Prof. Dr. Christian Rodolfo Esteve Rothenberg

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

CAMPINAS

2017

COMISSÃO JULGADORA - DISSERTAÇÃO DE MESTRADO

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

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

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

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

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

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

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

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

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

Acknowledgements

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

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

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

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

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

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

Japanese proverb

Abstract

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

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

Resumo

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

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

List of Figures

List of Tables

Contents

1 Traffic Modeling and Algorithms

In this chapter we are going to go in deep details on some implementations and modelling approaches mentioned in the last chapter.

In the frist section, we are going to discuss our methodology for modelling inter pacekt times, and validade it as a good choise. The problem we want to adress are these one:

- We have a set of empirical inter-packet times, and some aproximate stochastic models that describe it. We want to know what is the best, in a analyctical way.
- We whant to know whatever if our modelling aproaches are good ot not to describe inter pacekt times.

Many works have been made in second point on the literature, but not so many on the second. So we are proposing a model, and validating it.

In the second section, we will describe breifely two others we used to calc flow's session ON and OFF times, and

1.1 Inter Packet Times Modelling

~~1.1.1 About Inter packet time and packet trains modelling~~

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 automation [Varet 2014].

In this chapter, we propose a method automatizable 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 avoid these problems since our method is analytical.

We made a brief bibliographic review of some 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.

~~1.1.2 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.

1.2

1.1.~~3~~ Methodology

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

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

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 irewall gateway between our local and external network. Along with other tests, 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 website [3], and is called `bigFlows.pcap`. We will refer to it *bigflows-pcap*.

We summarize our process of modeling inter-packet times at the figure **??**. 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[4].

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);

[2] http://www.caida.org/home/
[3] http://tcpreplay.appneta.com/wiki/captures.html
[4] https://www.gnu.org/software/octave/

- 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. 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[5]

1.1.3.1 Linear regression (Gradient descendant)

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

$$y = ax + b \tag{1.1}$$

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:

$$F(t|\alpha,\beta) = F(t) = 1 - e^{-(t/\beta)^{\alpha}} \tag{1.2}$$

Manipuling the equation:

$$\alpha \ln(t) - \alpha \ln(\beta) = \ln(-\ln(1 - F(t))) \tag{1.3}$$

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 find a estimation of the linear coefficients $\hat{a}$ and $\hat{b}$. Using the inverse function of linear coefficients, we find the weibull estimated parameters $\hat{\alpha}$ and $\hat{\beta}$.

$$\alpha = a \tag{1.4}$$

$$\beta = e^{-(b/a)} \tag{1.5}$$

[5] https://github.com/AndersonPaschoalon/ProjetoMestrado/tree/master/Tests/PrototypeDataProcessor

The gradient descendent consists in minimizing a cost function $J(\theta)$. We explain this procedure in the appendix **??**. In the figure 1a we present as examples, the linearized data for the inter arrivals from the *skype-pcap*, and in the figure **??** the cost convergence. In the appendix **??**, a complete set of these figures is presented.

Applying the inverse equations of the linear coefficients ($\hat{\alpha} = \hat{a}$ and $\hat{\beta} = e^{-(\hat{b}/\hat{a})}$) [6], we are able to estimate the Weibull distribution parameters. We can summarize this procedure, in these steps:

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 (step 4), there is an exception, since the Pareto scale (t_m) is defined by the minimum time. In the table 1 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($\hat{}$) indicates an estimated value for a parameter.

Table 1 – 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	$x_i = \ln(t_i)$	$y_i = \ln(1 - F(t_i))$	$\hat{\alpha} = -\hat{a}$	$\hat{x_m} = \min_{i=0,\ldots,m}\{x_i\}$
Weibull	$x = \ln(t)$	$y = \ln(-\ln(1 - F(t)))$	$\hat{\alpha} = \hat{a}$	$\hat{\beta} = e^{-(\hat{b}/\hat{a})}$

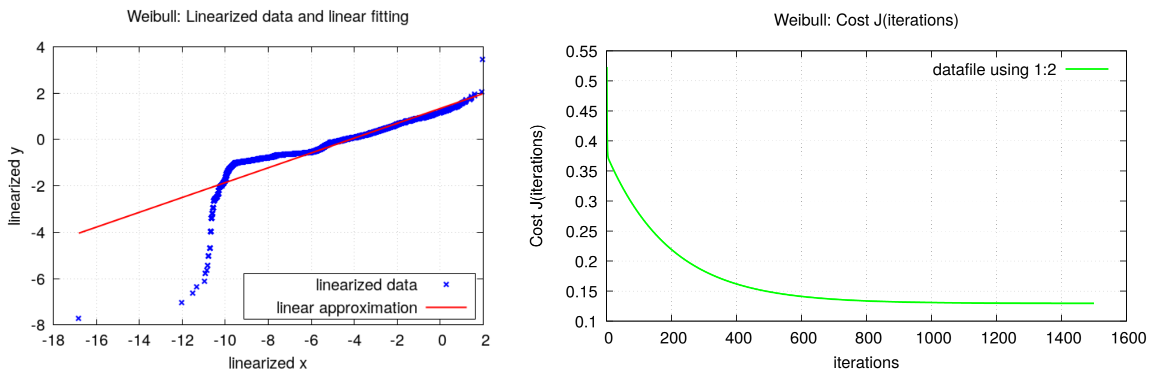
1.1.3.2 Direct Estimation

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 **??**, we have for the normal distribution:

$$(\hat{\mu}, \hat{\sigma} = (\bar{\mu}, \bar{\sigma}) \tag{1.6}$$

[6] The hat symbol ($\hat{}$) for the estimated parameters



(a) Linearized interarrival data (b) Cost function of the linear regression

Figure 1 – Linearized data and cost function of weibull linear regression

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

$$\hat{\lambda} = \frac{1}{\hat{\mu}} \tag{1.7}$$

1.1.3.3 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:

$$\hat{x_m} = \min_{i=0,\ldots,m} \{x_i\} \tag{1.8}$$

$$\hat{\alpha} = \frac{n}{\sum_{i=0}^{m}(\ln(x_i) - \ln(\hat{x_m}))} \tag{1.9}$$

where m is the sample size.

1.1.3.4 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 (Pearson's product-moment coefficient);

- 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 **??**, 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:

$$J(M) = \phi(Cr,M) + \phi(Me,M) + \phi(Hr,M) \tag{1.10}$$

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

1.1.4 Results

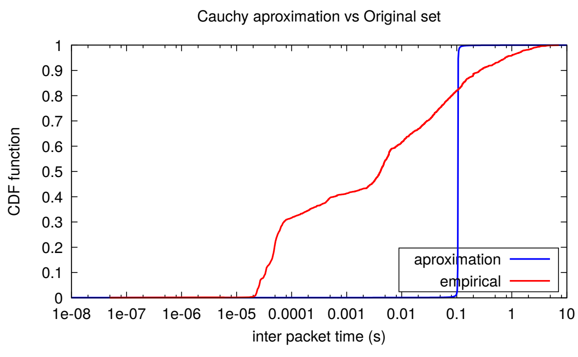
Table 2 – Results of the octave prototype, include BIC and AIC values, para estimated parameters for two of our pcap traces: *skype-pcap* and *bigflows-pcap*. *bigflows-pcap* is much larger, and has a much much smaller mean inter-packet time

	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	$-1.10e7$	$-1.10e7$	$\alpha : 2.81e-1$	$\beta : 5.54e-4$	$-5.46e7$	$-5.46e7$	$\alpha : 7.64e-2$	$\beta : 1e-6$

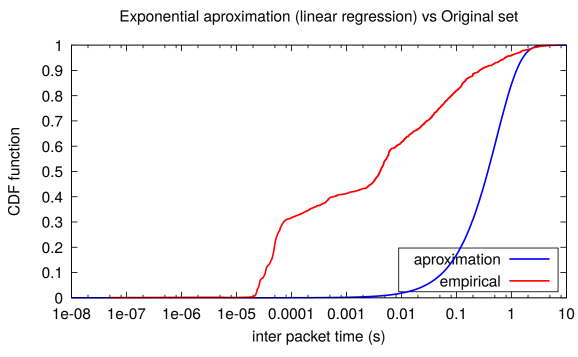
Here in this chapter we only discuss the plots achived obtained by the pcap *skype-pcap* for simplicity. The other plots are provided and commented on the appendix **??**. In the figure **??** we present all estimated CDF functions along with the empirical CDF, for the trace *skype-pcap*. They are on logscale, wich provide a better visualization for small time scales. 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 and types of fittings on each plot. Visually, the best fit seems to be the Weibull trough linear regression.

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. 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 higher. On the other hand, a self-similar process like Weibull and Pareto are better representing inter packet times with higher dispersion. Pareto(MLH) has a slow convergence, wich means this distribution may genarate values of inter-packet times too large.

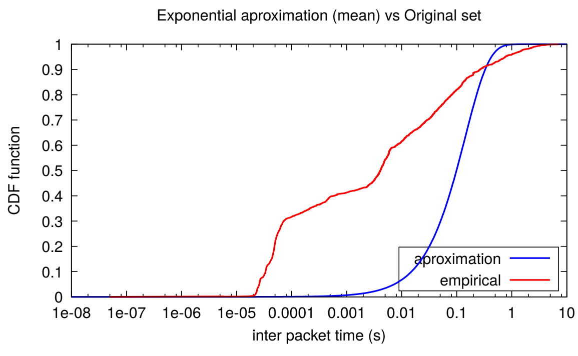
Analyzing the QQplots, we may observe that in most of the distribution, the samples (original data) has a much havier-tail effect than in the estimated data. This is verified by the almost blue horizontal lines formed by the "exes". But, the Weibull distribution follow much closer the original data. Also is possible to see that the sample has a right skew compared to the estimation on Exponential(LR), Exponential(Me), Normal, and Pareto(MLH). This means that this estimation would not represent so well small values of time in this case. On the other hand, Pareto(LR) and Weibull would not suffer from this problem.



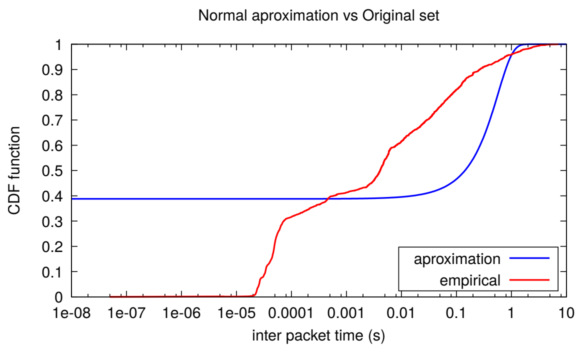
(a) Chauchy



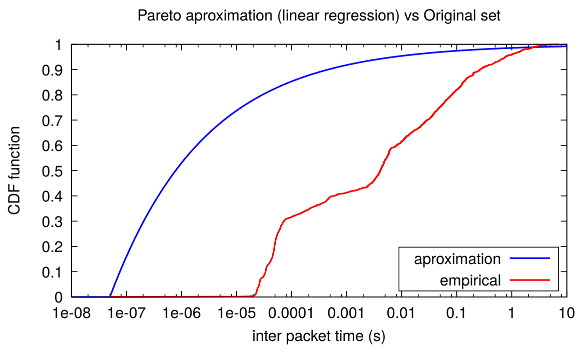
(b) Exponential(LR)



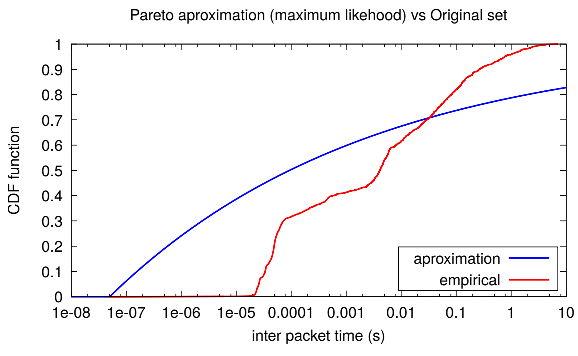
(c) Exponential(Me)



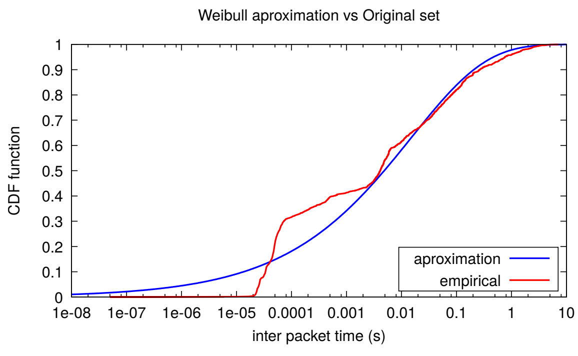
(d) Normal



(e) Pareto(LR)

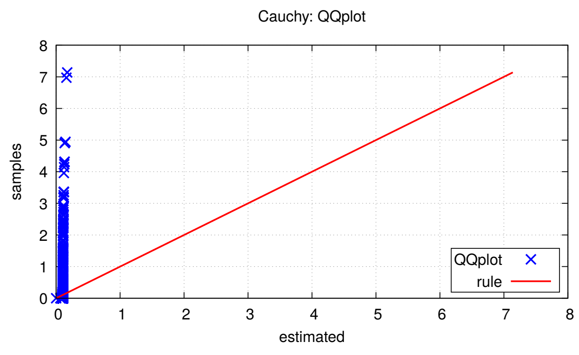


(f) Pareto(MLH)

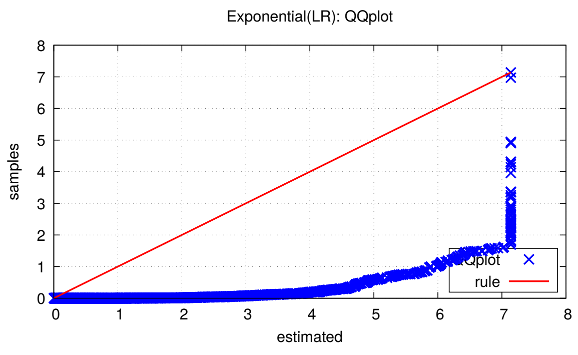


(g) Weibull

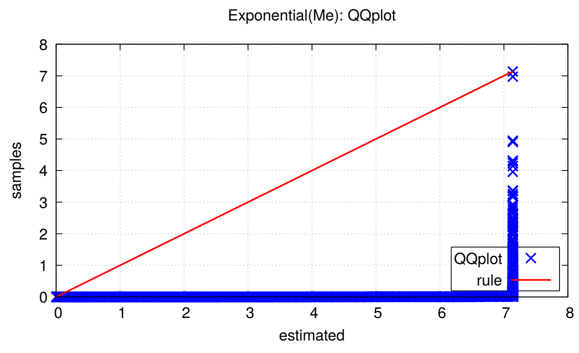
Figure 2 – CDF functions for the approximations of *skype-pcap* inter packet times, of many stochastic functions.



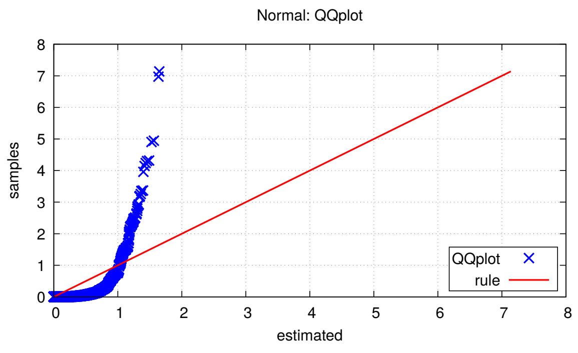
(a) Chauchy



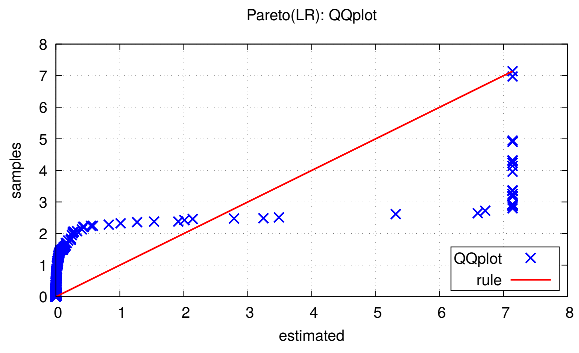
(b) Exponential(LR)



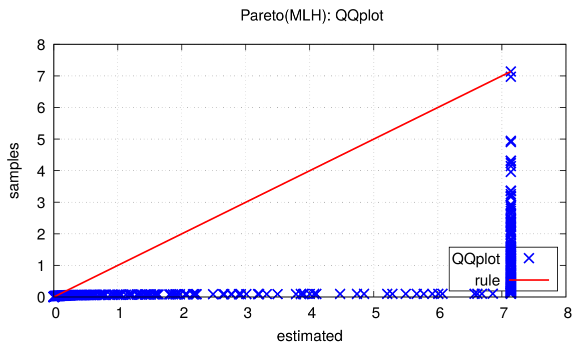
(c) Exponential(Me)



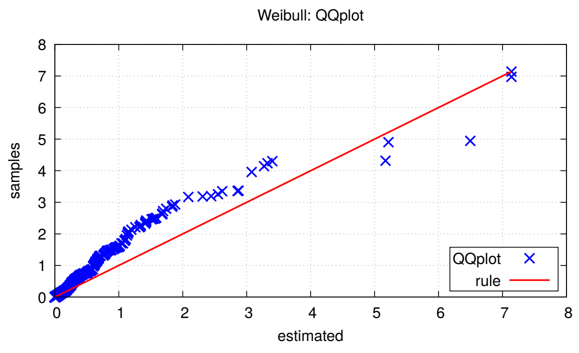
(d) Normal



(e) Pareto(LR)



(f) Pareto(MLH)



(g) Weibull

Figure 3 – CDF functions for the approximations of *skype-pcap* inter packet times, of many stochastic functions.

The results for the AIC, BIC and parameters of all four traces are at the table 2. The results for *wan-pcap* and *lan-diurnal-pcap* are on the table 2. The difference between BIC and AIC values in all simulations are very small. Much smaller then the difference of these values between the distributions. This is an indication that for inter packet times, using AIC or BIC should will not influence significantly the results.

On our previsions, Weibull and Pareto(MLH and LR) are the best options. This was expected, since both are heavy-tailed functions. But Cauchy on most of the tests, even being a heavy-tailed distribution, seems to no present a good fitting. This is effect of the fast divergence of tangent function, when we linearize our data.

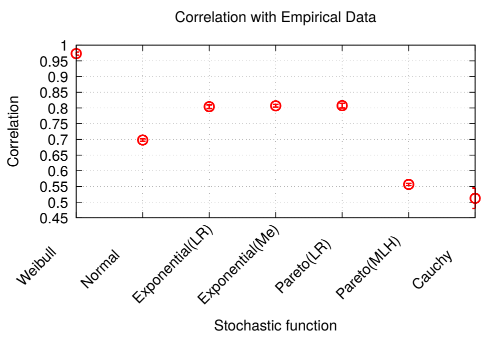
Analyzing the quality of AIC and BIC as criterion of choose on *skype-pcap*,based on the results form figure 4 we see that in therms of Correlation and Self-similarity it picket the right model: Weibull. Also in therms of mean packet rate and dispersion it is still one of the best choises (along with Exponential(Me), Pareto(LR) and Cauchy). 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.

All these results are abstracted by the cost function J. As we can see, on all pcaps, the best function selected by BIC and AIC 2 also had the small cost 5.

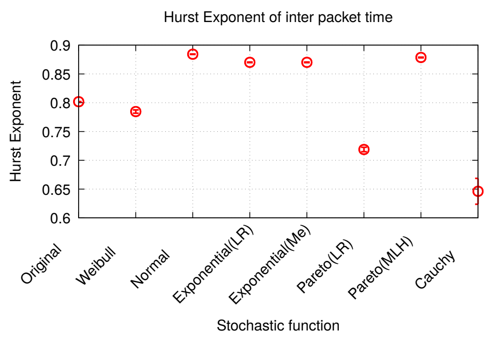
Another important observation is the fact that exponential function was able to provide the best fitting for the *wan-pcap*. The reasons for this behavior are both result of a much intense traffic with no long-range gaps, and the precision of the measurement.

Algorithm 1 stochasticModelFitting

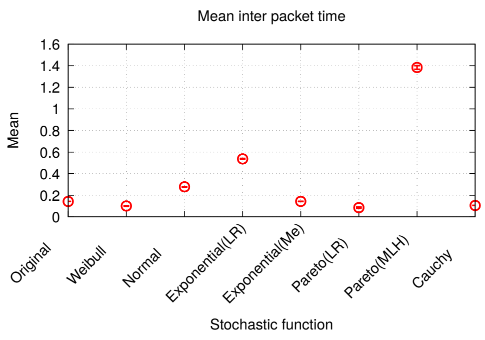
```
 1: function STOCHASTICMODELFITTING($interArrivalData, criterion$)
 2:     $m = interArrivalData.size$
 3:     $interArrivalData = interArrivalData + MIN\_TIME$
 4:     if $m < MINIMUM\_AMOUNT\_OF\_PACKETS$ then
 5:         $model\_list = \{constant\}$
 6:     else
 7:         $model\_list = \{weibull, pareto\_lr, pareto\_mlh, exponential\_me, exponential\_lr, normal,$
 8:         $cauchy, constant\}$
 9:     end if
10:     for $model$ in $model\_list$ do
11:         $model.fitting\_model(interArrivalData)$
12:     end for
13:     $model\_list.sort(criterion)$
14:     return $model\_list$
15: end function
```



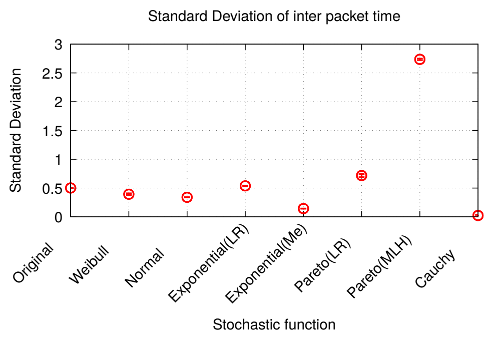
(a) Correlation



(b) Hust Exponent

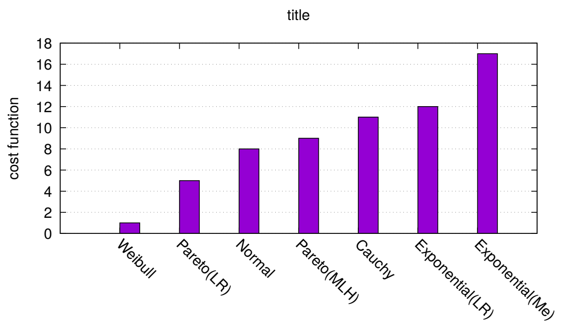


(c) Mean

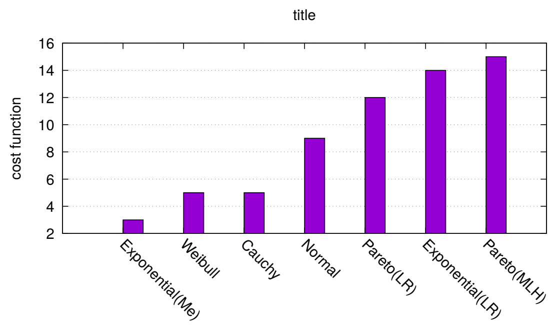


(d) Standard Deviation

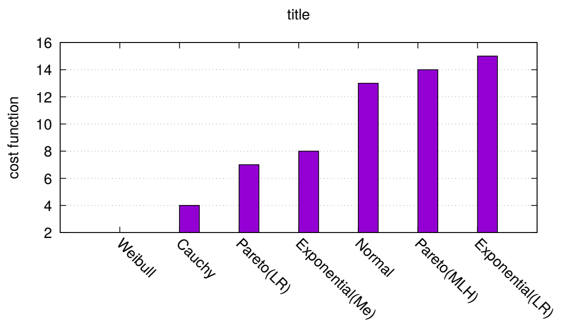
Figure 4 – Statistical parameters of *skype-pcap* and its approximations



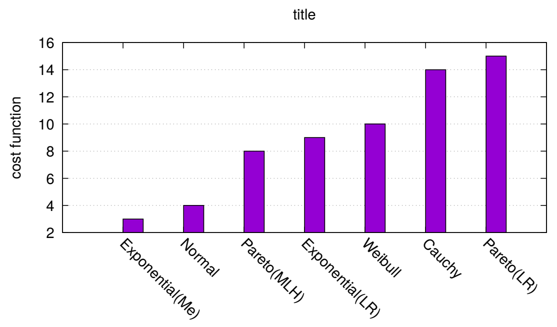
(a) *skype-pcap*



(b) *bigFlows-pcap*



(c) *lan-diurnal-pcap*



(d) *wan-pcap*

Figure 5 – Cost function for each one of the datasets used in this validation process

1.1.5 Conclusion

Also the order of the stochastic function tend to not differ much. We are able to conclude that using AIC or BIC as criteria for choosing models for inter-packet times is a good choise.

Also, except by the Pareto function modeled using the Maximum likelihood method, have comparative results between the simulations and the AIC/BIC goes. The best fittings according to the AIC/BIC usually returned very accurate models, with a small error between the mean and fractal level, and correlation close to one. The models selected as the worsts usually returned poor results.

Also, it is important to notice that in some cases, some stochastic functions may perform poorly, and in other provide an accurate fitting, what justify the application of the criteria in all new experiments. For example, Weibull usually performs pretty well, but in some cases, perform poorly, and in others may even diverge.

We do not rely on only one type of parameterization. This turns our methodology more robust, since a linear regression may diverge. But always there will be a peaceable model, since our estimations for Normal, Exponential (Me) and Pareto (MLH) cannot. Models which the linear regression diverges, will have very high and positive BIC/AIC estimations, and will not stand as primary options.

Table 3 – Application matach 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

1.2 Others Algorithms and Methods

In this section we breef present another methods used by the *TraceAnalyzer* component. The first to calculate the session ON and OFF times. The second classify applications based on its tranport protocol and ports. In the table **??** we summarize the stack of models used to create the Compact Trace Descriptor (CTD file).

Algorithm 2 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 $deltaTime[i] > cutTime$ then
            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 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
    $pktCounter.push(pktCounterSum)$
    $fileSize.push(fileSizeSum)$
    return $onTimes, offTimes, pktCounter, fileSize$
end function
```

1.2.1 On/Off Times estimation

We developed an algorithms to calculate the times between the ON and OFF times of the *files*, and the number of packets and bytes as well. It uses a list of packet arrival times(relative to the first), time between packets and packet sizes. It defines a minimum time of ON acceptable. This times is used for two reasons:

- In case of a *file* of just one packet, it will not produce a On time of zero
- It avoid a *file* On time too small, wich may be less than acceptable for packet generator tools.

As explained in the previous chapter, a *file* is defined by a large OFF time, we call $cutTime$. If a inter packet $deltaTime$ time is larger than the $cutTime$ it defines an OFF time, and the ON time is calculated base on the last OFF time recorded. If the first $deltaTime$ defines an OFF time, or if it is the first ON time, its calculation is threated separetely. In the last ON time, it is not defined by an OFF time, so we deal with it after the loop. It keep counters to calc the number of bytes and packets within every defined ON time. It returns four vectors: $onTimes$, $offTimes$, $pktCounter$ and $fileSize$. Letting n be the size of $onTimes$, $offTimes$ has a size $n-1$. The algorithm is presented in the Alforithm 2.

1.2.2 Application classification

We also developed an simple test to guess the application protocol, based on the port numbers and the tranport protocol used by each flow. We currently are classifing the application protocols presented in the table 3. If a flow matches the port number, and the transport protocol, it is classified as belongig to an application protocol.

2 Proof of Concepts and Validation

2.1 Validation Tests

As proof of concepts for our tool, we are going to use Mininet's emulated testbeds. We automate all tests using scripts, so our experiments are are fully reproducible. They are responsible for running all the proposed tests in this chapter and perform the calculations. It includes:

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

Each test is organized as a Python packege, responsible for tigger all the applications and procedures. The parameters for each simulation can be configured by a *config.py* file. The files *README* on each packege, provide a tutorial to run correctely each test. With all tools instaled, less then fifteen minutes is enought for run each test again. A complete specification of our scenario we show at the table 4, including the hardware specificatins and the software versions.

For each test, we generate a set of plots to compare the original and synthetic trace. Two of them we use for mere visual comparison: flows per second and bandwidth. To compare the realism quality of the generated traffic, we plot the flows cumulative distribution function (CDF) [Sommers *et al.* 2004], and the Wavelet multiresolution analysis. On every case, the more closer the plots are, the more similar the traffics are according to each perspective. Also, we wrote in the table 5 a complation of each traffic statistics.

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

Table 4 – Experiments specification table

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

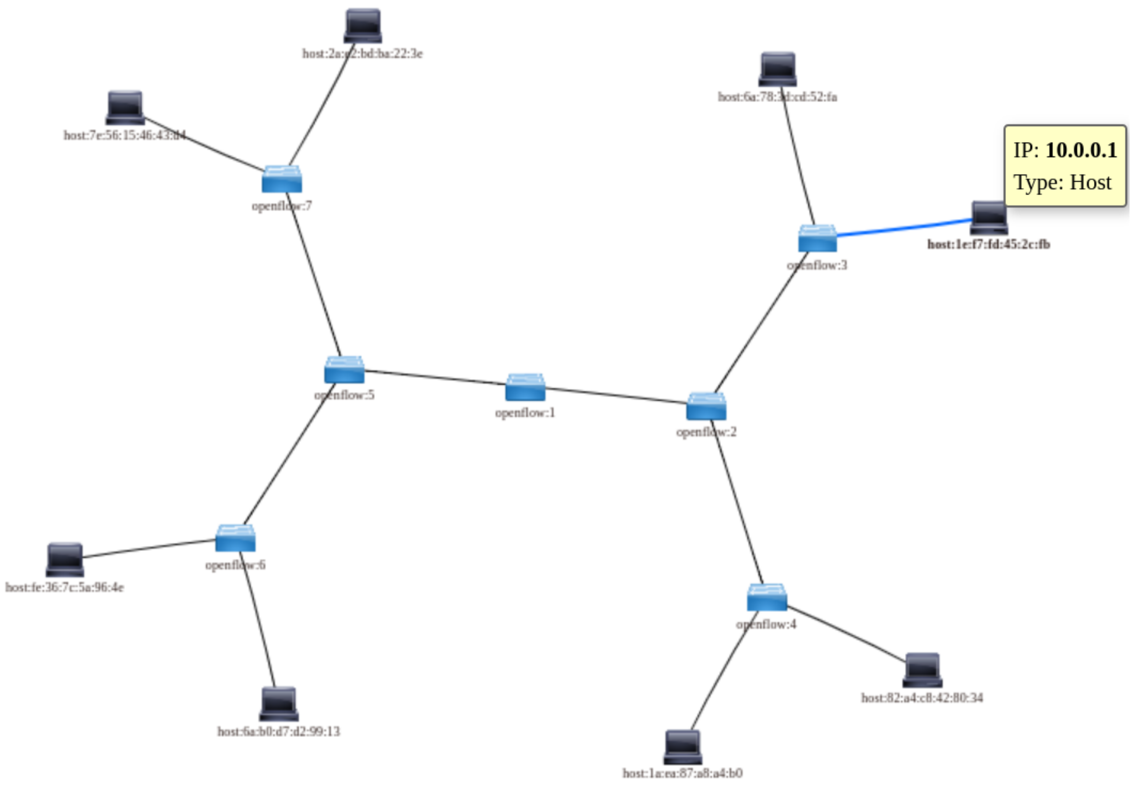


Figure 6 – Tree SDN topology emulated by mininet, and controlled by OpenDayLight Beryllium

We use as testbeds: a tree topology (figure 6), similar to tests performed by Swing [Vishwanath e Vahdat 2009] [Vishwanath e Vahdat 2008] [Bartlett e Mirkovic 2015], and a one-hop connection of two hosts. Both topologies are SDN networks and have OpenDayLight Beryllium as the controller.

For generating the traffic on the host *h1* with IPv4 address 10.0.0.1. The traffic

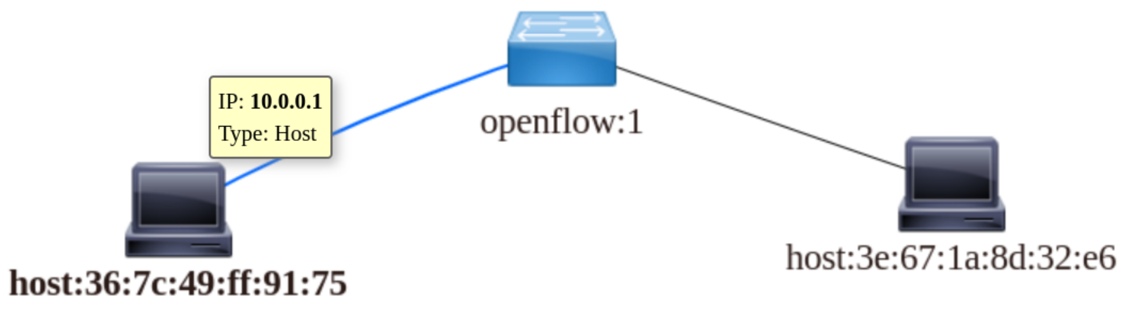


Figure 7 – Single hop SDN topology emulated by mininet, and controlled by OpenDayLight Beryllium

is captured from the host interface with TCPdump in a *pcap* format. We organized the project directory tree as follows[1]: the software is at *SIMITAR*. All validation tests and project prototypes we aved at *Tests* directory. The software documentation is at *Docs* directory.

We use SIMITAR v0.4.2 (Eulemur rubriventer)[2], as tagged at the GitHub repository. SIMITAR already have two functional traffic generator engines: Iperf and libtins C++ API.

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

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

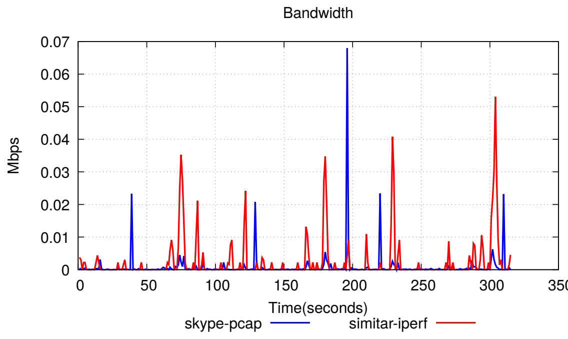
2.2 Results

We display our results in the figures 9 to 11, and in the table 5, where the original and the synthetic traffics are compared. As we can see in the figure 8, the generated traffics are

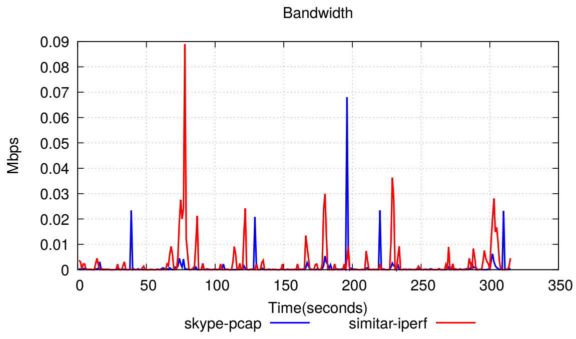
[1] https://github.com/AndersonPaschoalon/ProjetoMestrado
[2] We label the tags of SIMITAR control version on GitHub as lemurs species names (https://en.wikipedia.org/wiki/List_of_lemur_species)

Table 5 – Sumary of results comparing the original traces (italic) and te traffic generated by SIMITAR, with the description of the scenario.

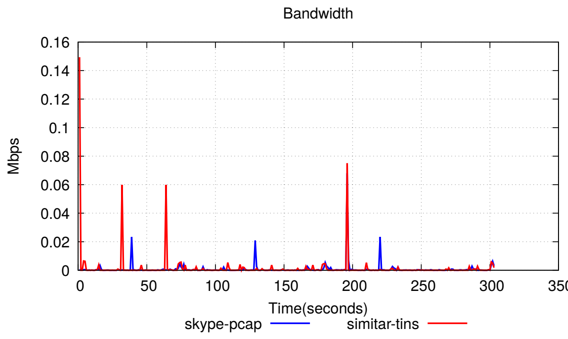
	skype-pcap	skype, one-hop, iperf	skype, tree, iperf	skype, one-hop, libtins	*lgw10s-pcap*	lgw10s, one-hop, libtins
Hurst Exponent	0.601	0.618	0.598	0.691	0.723	0.738
Data bit rate (kbps)	7	19	19	12	7252	6790
Average packet rate (packets/s)	3	4	5	6	2483	2440
Average packet size (bytes)	260,89	549,05	481,14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



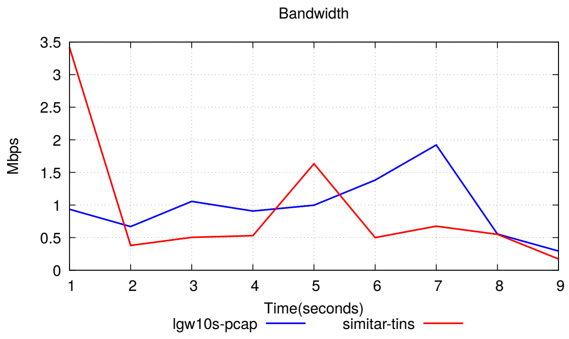
(a) *Iperf, single-hop, skype-pcap*



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



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

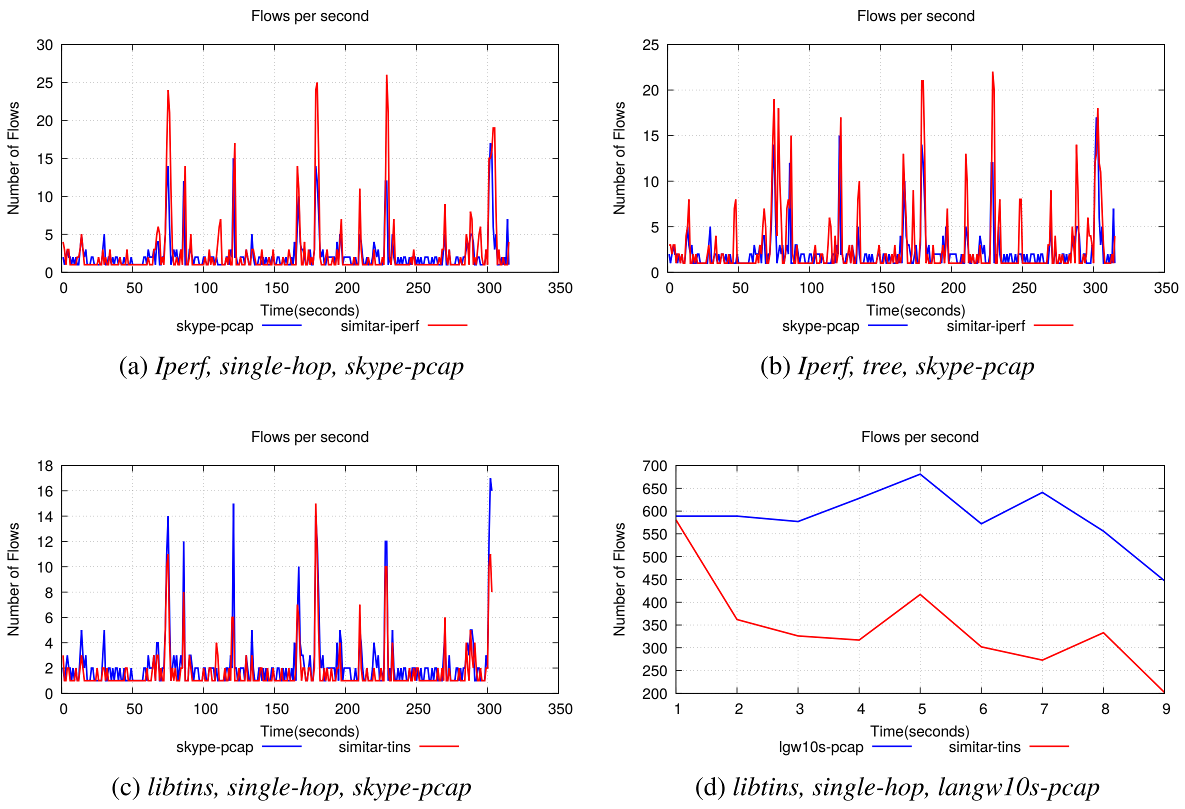


(d) *libtins, single-hop, langw10s-pcap*

Figure 8 – Traces bandwidth.

not identical regarding bandwidth, however both presents fractal-like shape. The Hurst exponent of inter-packet times in every case has an error smaller than 10% compared to the original in every case. This result indicates that in fact, the fractal-level of each synthetic traffic is indeed similar to the original.

The plot of flows per second seems much more accurate visually since most of the peaks match. Indeed, no visual lag between the plots. We can analyze it precisely observing the cumulative flow distribution10, where the results were almost identical on every plot. However, when SIMITAR is replicating the traffic of *lgw10s-pcap* the number of flows per second decreases. It happens because of the methodology of traffic generation of the class *TinsFlow* since it sends the packets of each stream as fast as possible. So each flow occurrence is restricted to



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

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

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

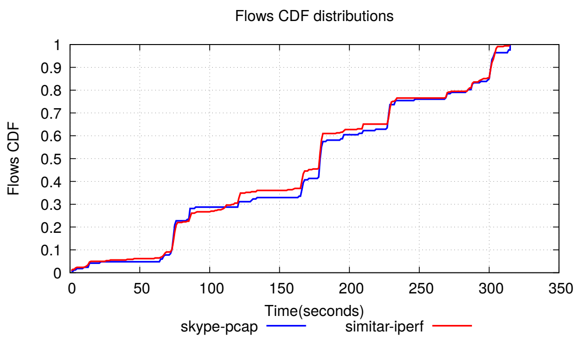
(d) *libtins, single-hop, langw10s-pcap*

Figure 9 – Flow per seconds

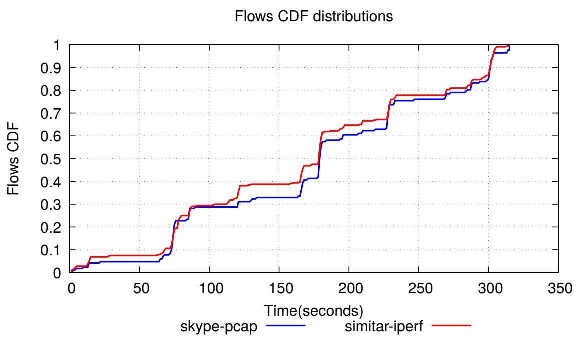
smaller intervals. This behavior can be observed as well in the bandwidth plot. It is much larger in the first seconds and small at the end.

The best results achieved by SIMITAR were on the flow distribution characterization. On every experiment made, the cumulative distribution of flows was almost identical. The small imprecisions on the plots are expected and should result from threads and process concurrence of resources and imprecision on the sleep/wake signaling on the traffic generation side. Imprecisions on packet capture buffer may count as well since the operating system did the packet timing. This was our most significant achievement in our research. This result means that our method of flow scheduling and independent traffic generation were effective and efficient on replicating the original traffic at the flow-level. The actual number of flows was much more significant when SIMITAR used Iperf and about the same amount but little small when used libtins. This discrepancy happens with Iperf because it establishes additional connections to control the signaling and traffic statistics. So, this more substantial number of flows comes from accounting, not just the traffic connections, but also with the signaling as well. With libtins, the number of flows is small, because, if it fails to create a new traffic flow, this flows generation execution is aborted.

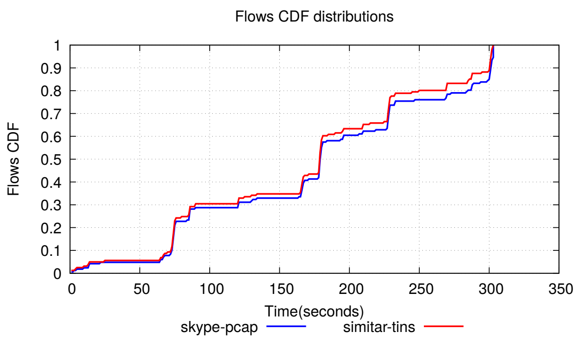
On Wavelet multiresolution analysis of inter-packet times, the results have changed more in each case. The time resolution chosen was ten milliseconds, and it is represented in



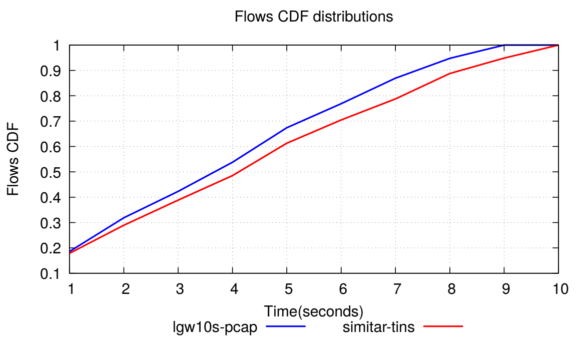
(a) *Iperf, single-hop, skype-pcap*



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



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



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

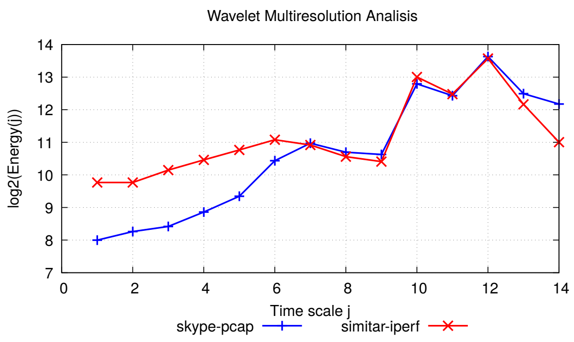
Figure 10 – Flows cumulative distributions.

$\log_2$ scale. The actual time pf each time-scale j value is given by the equation:

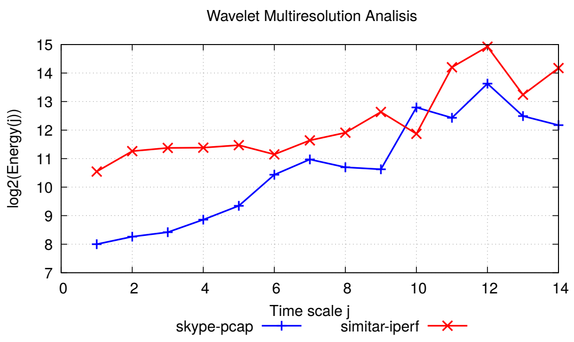
$$t = \frac{2^j}{100}[s] \tag{2.1}$$

In the first case (figure 11a), SIMITAR using Iperf in a single-hop scenario, on small time scales the energy level of the synthetic traffic remained almost constant, which indicates white-noise characteristics. The original skype traffic increased linearly, an indication of fractal shape. The synthetic trace started to increase at the time scale 5-6 (300-600 milliseconds). After this scale, the error between the curves become very small. One possible reason for this behavior is the fact that the constant which regulate the minimum burst time (`DataProcessor::m_min_on_time`) is set to 100 milliseconds. For small time scales the traffic become similar to white noise since Iperf just emits traffic with constant bandwidth. For larger scales, it captures the same fractals patterns of the original trace. It also captures a periodicity pattern at the time-scale of 9 seconds. The authors of [Vishwanath e Vahdat 2009] measured the same periodicity pattern. But on this trace, we observe some periodicity at 11 and 13 time-scales (20 and 80 seconds).

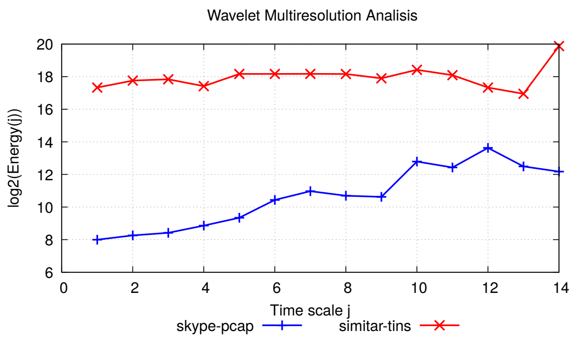
In the second case, on a tree topology on small time scales, the same white-noise behavior is identified. We identify a similar behavior on the energy behavior on larger time-scales, but with a larger gap between the curves. Packet collision on switches requiring retransmission



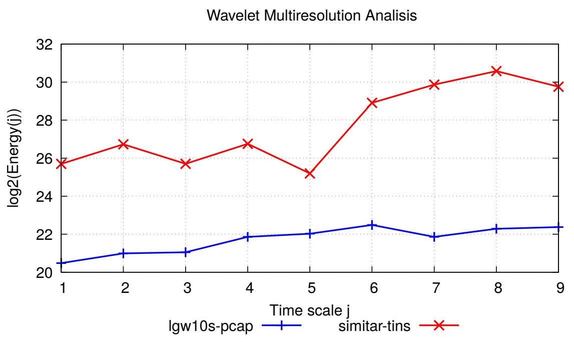
(a) *Iperf, single-hop, skype-pcap*



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



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



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

Figure 11 – Wavelet multiresolution energy analysis.

packets explains this behavior. In fact, as we can see in the table 5, two hundred more packets are leaving the client on the tree topology compared to the one-hop.

On the last two plots, where we use libtins as packet crafter, the energy level is much higher, and the curves are much less correlated. SIMITAR are not modeling inter-packet with libtins, and are sending packets as fast as possible. We may observe on the figures 8c and 8d that than have much higher peaks compared to the original *skype-pcap*. As we see in the figure 8c, it just captured some periodicity characteristics at the time larger than 11-12 (20-40 seconds). SIMITAR determines larger periods between packets with session OFF times, and the session cut time (`DataProcessor::m_session_cut_time`) is 30 seconds, that explain this behavior.

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

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

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

2.3 Conclusions

SIMITAR was designed to work at flow-level and packet level. At the flow-level, our methodology can achieve great results. In fact, The cumulative distribution of flows is almost identical on each case. From the perspective of benchmark of a middle-box such as an SND switch, this is a great result, since its performance depends on the number of registered flows. However, because of packets exchanged by signaling connection, the traffic generated by Iperf, even following the same cumulative flow distribution, had created more streams then expected.

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

Once the current implementation of the Flow Generator still does not uses the whole set of parameters from the Compact Trace Descriptor, and many optimizations yet to be made, this is a satisfactory result that validates and proves the potential of our proposed methodology. Even we designing SIMITAR to prioritizing modularity over finner control, when we compare these current results with the more consolidated realistic traffic generator available, they are closer. At the flow-lever our results are at least as good as the achieved by Harpoon [Sommers *et al.* 2004] and Swing [Vishwanath e Vahdat 2009]. At the scaling characteristics, on lightweight traces, they are already comparable in quality.

3 Conclusion and Future Work

3.1 Future Work

With the work done so far and the results achieved, we may identify the current limitation of our project, and plan improvements to solve these issues. We can list as main weak points right now:

- Poor performance of the Trace Analyzer and Sniffer components;
- Weak scaling results of the traffic generator using *libtins*.

And as further implementations, but with low priority, we may have:

- Support for new traffic generator tools;
- APIs for the Flow Generator on Python and Lua, wich have cosolidate traffic generator APIs;
- A *pcap-crafter* component. Instead of creating traffic, creates a *pcap* file.

Identified these issues and possible future works, we conceived approaches and methods to achieve these results. They include improvements on the implementation, calibration of constants[1], a finner control of packet injection, use of new technologies and newer implementations. As we always did in the project, we create a task list of activities and organize them by priority and goal. They are listed in the table 6.

(1) The major issue of SIMITAR now is optimizing data processing for creating the compact trace descriptor. The performance becomes an issue when processing large *pcap* files with more dozens of thousands of flows. The time expended for processing traces, in this case, is exceeding tens of hours. In the current implementation, the linear regression execution is mono-thread, and the stop criterion is just the number of iterations. Parallel processing, and creating stop criteria based on convergence in addition to the number of iterations will improve the performance, along with some code optimizations.

(2) Crating an option for merging flows is a possibility to improve the performance of traffic with several thousands of flows and Gigabits of bandwidth, such as from WAN captures. A merge criterion, for example, is to consider just network headers on flow's classification.

[1] Calibration here means to test many values, and see which one can improve our outcomes for a large variety of traces of traces.

Table 6 – Future Work: task list

	Future Work		Priority
1	Optimizing linear regression	Performance	***high***
2	Flow-merging option		*medium*
3	Smart flow-scheduler and thread/process management		***high***
4	`DataProcessor::minimumAmountOfPackets`		low
5	DPDK KNI interfaces		*medium*
6	Multi-thread C++ sniffer		***high***
7	Cluster operation (high speed traffic-gen)		***high***
8	Model inter-packet times on `TinsFlow`	Tool support	***high***
9	D-ITG flow generator (`DitgFlow`)		low
10	DPDK flow generator (`DpdkFlow`)		*medium*
11	Ostinato flow generator (`OstinatoFlow`)		low
12	`DataProcessor::min_time`	Calibration	low
13	`DataProcessor::m_min_on_time`		low
14	`DataProcessor::m_session_cut_time`		low
15	Minimum number of packets per flow option		low
16	Python API/Lua for traffic generation	New features	low
17	Crafter of pcap files		low

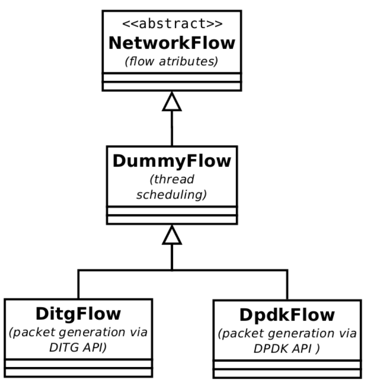
(3) Currently, all the flow threads are instantiated once the traffic generations start. A smarter traffic generation where SIMITAR instantiates each thread when the traffic, and join when it is inactive should reduce the overhead for traces with a large number of flows. For traffics with a massive amount of streams, a methodology for merging them should reduce computational costs and enable replication of traffic with a much more significant bandwidth. Another improvement in traffic generations is reduced memory consumption per thread and process, and optimize the code.

(4) Constant `DataProcessor::minimumAmountOfPackets`: SIMITAR only estimates stochastic models for inter-packet times if the number of packets for this flow is larger than this amount. If it is smaller, SIMITAR just uses the constant model. We decide to do it for two reasons. First, because with a small sample, the modeling accuracy is poor. Second, we avoid the cost of the parameterization process. Now, its value is set to Its value today is set to 30. Increasing this amount, we may achieve a more reasonable performance for individual flows, and reduce the processing time.

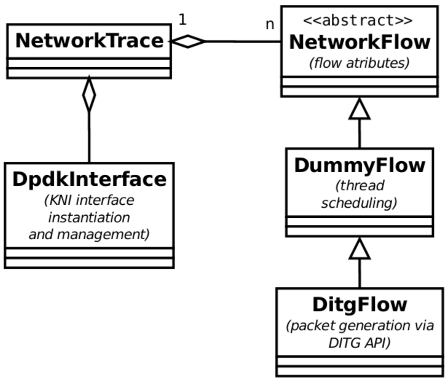
(5) One possibility to improve the traffic generation performance issue DPDK Kernel NIC Interface (KNI interfaces) [2]. DPDK KNI interfaces allow applications from the user's space interact with DPDK ports. In this way, we may achieve a faster packet processing. This can be integrated as an insatiable feature of SIMITAR (figure 12b).

(6) Another issue is the sniffer component is taking to much time to extract data from large *pcap* files. Implementing a C/C++ multi-thread sniffer, including the packet process-

[2] http://dpdk.org/doc/guides/prog_guide/kernel_nic_interface.html



(a) DpdkFlow



(b) DpdkInterface

Figure 12 – Class diagram for DPDK support expansions. On (a), we have a implementation of traffic generation based on DPDK. On (b) we are using DPDK KNI interfaces.

ing and the SQL queries, will improve its performance.

(7) Since the *compact trace descriptor* describes n different flows, it can be splitted on m different files, each describing n/m flows, and still describe the same traffic. With m synchronized machines woking in parallel, each using one of these m parts of the original *compact trace descriptor*, with use of link-agregattion techinique[3] we can achieve high speeds on bare-metal.

(8) SIMITAR's current implementation using *libtins* to generate the packets does not model inter-packet times. To avoid overheads on random numbers calculation, these values must be calculated before the actual traffic generation. Modelling inter-packet times the scaling characteristics of libtins traffic should improve.

(9-11) Expand SIMITAR to other traffic generator tools, as we present in the figure 12. D-ITG offers many stochastic functions for customization of inter-packet times, Ostinato provides a rich set of headers and protocols, and DPDK a high performance on packet generation. Each tool can offer a different result on traffic generation, each with benefits and drawbacks, such as computational cost, and development complexity. Also, DPDK provides the possibility to bypass the Linux network stack, called packet acceleration. So, this implementation can enable for SIMITAR high-bandwidth along with traffic realism.

(12) Calibrate the constant `DataProcessor::min_time`: smallest time considered for inter-packet times. We use this value to avoid inter-packet times equals to zero due to the sniffer resolution. A zero-inter-packet time would make the procedure fail. This value can change the fitting accuracy. Today, this value is $5e-8$.

(13) Calibrate the constant `DataProcessor::m_min_on_time`: this value controls the small ON time that a *file* can have. It can change the generated traffic precision. Currently this value is 0.1s.

[3] http://www.comutadores.com.br/configurando-link-aggregationetherchannel-entre-cisco-e-hpn/

(14) Calibrate the constant `DataProcessor::m_session_cut_time`: this value defines whatever a file transference still active or has ended. It defines the smallest OFF time acceptable. This value will change how many files SIMITAR will transfer per flow, and how many times it will call the underlying traffic generator. This constant affects the computational performance and traffic realism.

(15) Constant *Control a minimum number of packets required for SIMITAR create a new flow*: this would reduce the number of flows created, improving its computational performance. Since SIMITAR manages each flow in a different thread, it will create fewer threads simultaneously, reducing overheads. This action should not impact on throughput and scaling characteristics because few packets would be ignored. But it can increase the precision of the most significant flows since all the threads are competing for each other on the Operational System. But, the number of flows created would be smaller.

(16) Currently, SIMITAR only enables the programming of flow traffic generation in C++. Adding Python and Lua support for the Flow Generator component, we can enable expansion for Python/Lua traffic generation APIs (such as Ostinato and MoonGen APIs), without creating C++ wrappers.

(17) **PcapGen: a compact *pcap* lybrary**: Create a component capable of generate synthetic *pcap* files, using Compact Trace Descriptors(CTDs) files. We can implement this using SIMITAR as a packet injector, but in an emulated host interface, for example, using Mininet. Then, the traffic can be collected, using a tool such as TCPDump or Tshark. We present a diagram of this idea in the figure 13. This expansion would enable SIMITAR to work as trace library for pcap-based benchmark tools.

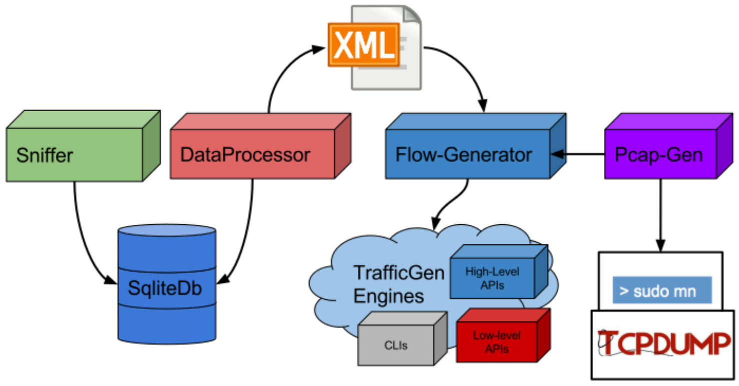


Figure 13 – Using SIMITAR for generation synthetic *pcap* files, CTD files: a component schema

3.2 Final Conclusions

In this dissertation we discuss our process of conceiving, researching, definition and development and validation of an realistic traffic generator able to at the same time fulfill gaps others proposals haven't, and achieve results comparable to the available proposals. This was not a linear procedure. Was indeed an espiral procedure, as proposed by Sommervile on Software Engineering [Sommerville 2007]. Clealy its not identical, because we adapt it for research purposes. At the figure 14 we show a diagram of who it was done.

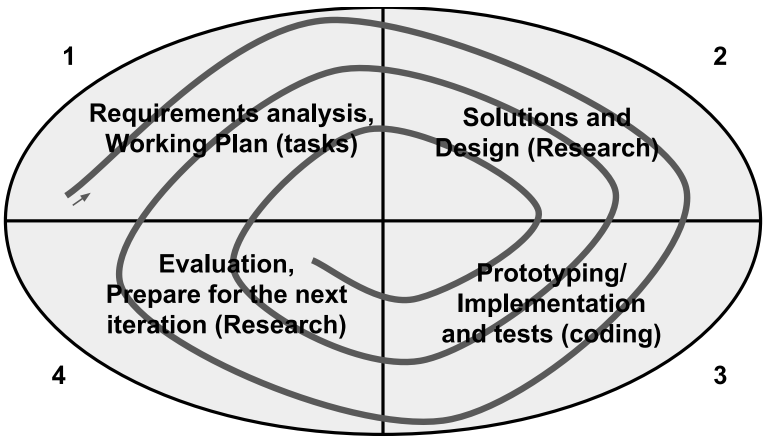

1
2
Requirements analysis, Working Plan (tasks)
Solutions and Design (Research)
Evaluation, Prepare for the next iteration (Research)
Prototyping/ Implementation and tests (coding)
4
3

Figure 14 – Spiral research and development procedure

We started defining our goals creating a task list, formalizing it on "Working Plan" documents. These tasks lists covered the whole process and included research, the study of new subjects, implementation, and testing. Next, search on the literature the state of things, related works and points for innovation. At this point, we defined the concepts of our traffic generator. We create an architecture model using UML, defining the software components, classes and sequence diagrams. This initial step was crucial since, in later stages, changes in the architecture design are much more difficult. In fact, some small changes such as directory structure and new classes are inevitable. But structural changes are impracticable. At this point we implemented a simplistic prototype, and validate it[4].

After this, we search for more solutions and methodologies that could be applied to solve our problems and improve the performance of our prototype. In this procedure, we researched many topics that at the end didn't fit our intentions, such as Machine Learning and Neural Networks. But others such as Linear regression, maximum likelihood, and information criterions (BIC/AIC) were indeed satisfactory. Again, we planned a methodology, procedure, and validate these ideas, as prototypes. Here the seed for the first part of the chapter 1 and the article "Using BIC and AIC for Ethernet traffic model selection. Is it worth?" (in the appendix **??**). Then we research for more related works to gain insights and find platforms and libraries

[4] In the appendix **??**, this prototype is validated in the article "Towards a Flexible and Extensible Framework for Realistic Traffic Generation on Emerging Networking Scenarios"

able to fulfill our requirements. In this step, we designed our algorithms presented in chapter 1, and chose the APIs, packages, and libraries, such as Armadillo [Sanderson e Curtin 2016], a C++ linear algebra library. Many ideas we used to conceive our ideas were inspired by previous works and solutions, such as D-ITG [Botta *et al.* 2012], Harpoon [Sommers e Barford 2004], Swing [Vishwanath e Vahdat 2009] sourcesOnoff [Varet 2014], and LegoTG [Bartlett e Mirkovic 2015]. This procedure continued with incremental upgrades until we arrived on the version presented on the chapter 2.

In the chapter 1 we achieve a significant contribution of our work, which shows that the information criteria AIC and BIC are efficient analytical methods for select models for inter-packet times. Both can infer a good model, even evaluated according to different types of metrics, without any simulation. Also, we show evidence that for Ethernet traffic of data, choosing AIC and BIC make almost no difference. As far as is our knowledge, this is a complete study on the use of AIC and BIC on inter-packet times modeling of Ethernet traffic.

In the end, we were able to implement a functional implementation of the solution proposed in the chapter **??**. Our main contribution here is the development of a traffic generator tool that at the same time can provide realistic synthetic traffic, and its modeling framework is extensible for most of the traffic generation libraries and tools. In fact, our current implementations use tow very distinct packet-crafters: *libtins*, a C++ library designed for the implementation of sniffers and traffic generators, and Iperf, a traffic generator commonly used to measure bandwidth.

Out tests performed on the chapter 2 intend to focus on packet-level, flow-level and scaling metrics, as we state on chapter **??**. The results were notably good at the flow-level. SIMITAR were able to replicate the flow-cumulative distribution with high accuracy, and with *libtins*, the number of flows as well. The number of flows was larger because it establishes additional connections for signaling and statistics control. On scaling and packet matrics, the results still have to be improved. Iperf as the traffic generator tool, still being limited, has proved to be efficient on replication the scaling characteristics for the Skype traffic. Since it establishes socket connections to generate the traffic, we believe that this fact makes it accurate on replication traffic from applications.

Finally, we created a list of improvements and future works, aiming the improvement of the software, inclusing, performance better processing time and packet generation, and the realismo on the traffic generated. In fact, the greater bootleneck of the project resides on processing performance. The results on realism, even with a lot of room for improovments, already have a good quality. In the is we believe to be possible overcome the current limitations. Also, on this limitation

Bibliography

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.; 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>.

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>.

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.

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

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.

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.

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.

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.

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.

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.

SANDERSON, C.; CURTIN, R. Armadillo: a template-based c++ library for linear algebra. *Journal of Open Source Software*, v. 1, 2016.

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>.

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

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

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

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

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