RENETO, a Realistic Network Traffic Generator for OMNeT++/INET

Fabien Geyer[1 2] Stefan Schneele[1] Georg Carle[2]

[1]EADS Innovation Works
Dept. IW-SI-CO
D-81663 München, Germany
{fabien.geyer,stefan.schneele}@eads.net

[2]Technische Universität München
Institut für Informatik, I-8
D-85748 Garching b. München, Germany
carle@net.in.tum.de

ABSTRACT

We present in this paper RENETO, a packet-level traffic generator for OMNeT++/INET. In order to achieve realistic traffic behavior, a first tool computes a model by doing an automatic analysis of a real traffic capture. This analysis extracts statistical distributions of different parameters of the model (e.g., packet size, inter-arrival time). Based on this first step, we then generate traffic in the OMNeT++ simulator corresponding to the observed behavior. Related traffic analysis and generator often model each studied parameter as one statistical distribution, thus treating them as statistically independent. With this work, we use the concept of linking some parameters and making them correlated, in order to mimic more accurately traffic patterns seen in reality. We apply our method to both UDP and TCP based traffic.

Categories and Subject Descriptors

I.6 [**Computing Methodologies**]: Simulation and Modeling; C.2 [**Computer Systems Organization**]: Computer-Communication Networks

General Terms

Theory, Measurement, Performance

Keywords

Network traffic generation, Network traffic modeling, Pseudo-random number generation

1. INTRODUCTION

The goal of this work is to develop a general traffic generator able to produce realistic traffic flows, based on the analysis of existing real traffic capture. The tool should be able to adapt to a wide range of protocols, and reproduces accurately traffic patterns seen in a real network at different ranges of time scales, in order to reproduce both a similar average bandwidth and burstiness.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Simutools 2013, March 05-07, Cannes, France
Copyright © 2013 ICST 978-1-936968-76-3
DOI 10.4108/icst.simutools.2013.251697

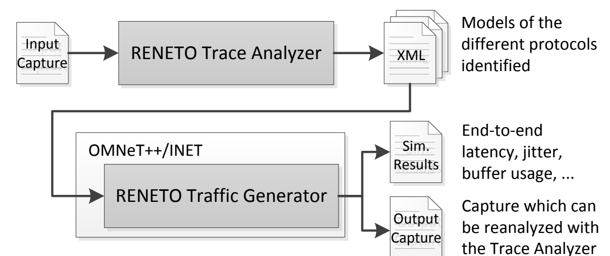

Figure 1: Complete toolchain

Behind this general challenge lie three fundamental problems. First, we have to define a model able to represent the characteristics of different protocols in a straightforward way. We base this model on the following hierarchy: user, session, connection, and packet level. Second, we need a way to extract values for those different layers from a real traffic capture. We designed a tool for doing this analysis in an automatic way, which records the statistical distribution of the various parameters of the model. Lastly, we have to reproduce those different layers in a network simulator, and generate flows which follow the statistical distributions recorded from the original capture.

In this paper, we present the design, implementation and evaluation of RENETO, a **Re**alistic **Ne**twork **T**raffic generator for **O**MNeT++ [2], which statistically reproduce network flows. While related work often represents and generates the different parameters of the model as statistically independent variables, the contribution of this paper is to propose a model which adds correlation between some parameters of the model. Such technique is useful to capture patterns which can be seen in different protocols with bursty behavior. RENETO was designed to work with any protocol on top of UDP or TCP and may be extended to other protocols in the future. The complete toolchain of our traffic generator is represented in Figure 1 with the two tools developed for this generator.

This traffic generator was developed in order to simulate realistic traffic loads to serve as input for studying appropriate scheduling policies in Ethernet switches. Although it was not designed for applications such as anomaly detection, protocol detection and categorization, the underlying model can also be used for a wide range of higher level studies, and

is not limited to the chosen network simulator.

This work is structured as follows. In Section 2, we present similar research studies. Section 3 details the model, its parameters, and how to extract values from a network capture. With Section 4, we present the design of the traffic generator developed for OMNeT++. In Section 5, we made an evaluation of the different parts of the toolchain, and compare measures on original captures with measures on synthetic traffic. Finally, Section 6 summarizes and concludes our work, and gives an overview of future improvements.

2. RELATED WORK

While generating traffic flows in a simulator is generally not difficult in itself, making it realistic is challenging. The problem of characterizing, modeling and generating network load in simulations is a core challenge when doing performance evaluations in simulators.

Work has already been performed on this problem on different simulators, but with focus on a certain type of traffic, like the generation of web traffic (HTTP) in SURGE [5] or HttpTools [10] for OMNeT++, or also the generation of synthetic multimedia flows in OMNeT++ for VoIP [6].

While such tools are useful for evaluating specific use cases, research was done on more generic tools, working with wider range of applications. For instance tools such as Swing [16] or Tmix [8] are able to analyze network capture with TCP-based protocols and generate according traffic.

The idea of correlating parameters of the model was already used in the extended version of LiTGen [15], which focused on TCP traffic. The results of their study was that correlating the packet size and the inter-arrival time of packets was beneficial to synthesize realistic traffic. We use this concept for our own work and extend it to UDP traffic.

3. TRAFFIC MODEL

We present in this section the structural model which was developed for RENETO, as well as the tool *RENETO Trace Analyzer*, which extract such model based on a network capture.

3.1 General description

A structural model was developed for RENETO taking into account the different layers of the protocol stack. The structure presented here is similar to the work done in [14, 15, 16], where individual packets are grouped into sessions and flows. Our model is made on the analysis of the layer 4 payload of the packets. This means that for TCP based protocols, we will focus on TCP messages, and not the underlying mechanisms of TCP flow control algorithms. The task of modeling TCP is delegated to the TCP models provided by INET [1].

For each studied protocol, the four following layers are used in RENETO:

Packet layer: We distinguish here two cases, depending on the layer 4 protocol of the studied application. In case of UDP based flows, we characterize the layer 4 payload size distribution of each frame in each way for a bidirectional communication, with the variable $reqSize$ for requests, and $respSize$ for responses. We characterize the distribution of inter-arrival time between packets with the variables I_{req} for requests, and I_{resp} for responses in case the connection uses more than one request/response exchange.

In case of TCP based flows, the notion of individual packet size (Ethernet frame size) is not present in our model, as it highly depends on the flavor of TCP used, as well as the condition of the network on which the trace was made. Regarding packet size, we do not look at individual packets like for UDP, but we group TCP segments together to form a layer 7 message or object. The length of this layer 7 message will define the $reqSize$ and $respSize$ variables. We also characterize the think time after reception of a message, with the variable I_{req} for the client side, and I_{resp} for the server side.

Flow layer: Because our approach is focused on TCP and UDP traffic, we define a traffic flow as a sequence of packets having the following unique parameters for a certain time: source and destination IP address, layer 4 protocol (TCP or UDP), source and destination layer 4 ports. By permuting source and destination, we capture both directions of a flow (when applicable). This definition fits both cases of streaming, where a server sends packets without expecting any reply or acknowledgments, and requests-responses between a client and server.

We characterize the distribution of number of request-responses pairs occurring in a flow (for bi-directional flows), with the variable N_{pairs}. We record also in this layer, the transport layer (TCP or UDP) which was used.

Session layer: We define a session as a group of flows occurring around the same active period initiated by the same source. Between those sessions are inactive periods. This is similar to the notion of RRE (Request/Response Exchanges) used in Swing [16] and also described as an important part of the model used in [14]. We characterize the distribution of the number of flows per session with the variable N_{flow}, as well as the distribution of time between the start of two flows with the variable I_{flow}.

User layer: Finally we define a user as the agent starting the different sessions. Similarly to the session layer, we characterize the distribution of the number of sessions initiated by a user with the variable $N_{session}$ and the distribution of time between the start of two sessions with the variable $I_{session}$.

We identify single users by their IP address. While this definition can be accepted for private networks, it does not fit completely with the Internet of today, where multiple users can be behind a single public IP address. When making a network capture and analyzing it, such point has to be kept in mind.

A summary of the model is given in Table 1. The parameters I_{req} and $reqSize$ are also recorded using a bivariate distribution, as well as the parameters I_{resp} and $respSize$. We did not use the correlation of other parameters at this step of our work.

In order to have a quick overview of how the structural model works, we take here the example of a user browsing a website (HTTP protocol). Events in the user layer of our model will typically correspond to user actions such as clicks on links. We record it in the user layer: number of events (such as a click) and time between events (such as time to read a page). When the browser loads web pages, different requests are made around the same time to first fetch the HTML page, and then other resources present on the page, such as images or scripts. The session layer records those requests: number and time between requests. As each request is an individual flow, we record the number of ex-

Layer	Variable	Description
UDP Packet	$reqSize$	Packet size of requests
	$respSize$	Packet size of responses
	I_{req}	Inter-arrival time of requests
	I_{resp}	Inter-arrival time of responses
TCP Packet	$reqSize$	Message size of requests
	$respSize$	Message size of responses
	I_{req}	Think time before request
	I_{resp}	Think time before response
Flow	N_{pairs}	Number of request-responses pairs
Session	N_{flow}	Number of flows
	I_{flow}	Time between start of flows
User	$N_{session}$	Number of sessions
	$I_{session}$	Time between start of sessions

Table 1: Summary of the model

changes, which in case of HTTP is 1 or more, as well as the time between those exchanges (processing time of the response). Finally, we look at the message size (as HTTP is based on TCP): less than 10 kb for an HTTP GET request, and around 500 kb in case the server answers with a small image (which would fit into multiple TCP segments).

3.2 Trace analysis and parameters

In order to produce realistic traffic, we use publicly available captures or network captures made on testbeds. Our approach focuses on the analysis of full packet captures for a given link, typically using the PCAP file format, giving us the ability to investigate the application layer, which provides us with more information on a flow than other capture methods such as NetFlow.

The first step for treating the capture is to assign each packet to a traffic flow. For UDP and TCP traffic, a first base is to use IP addresses, port numbers of the transport layer, as well as the timestamp of the packets. In case of TCP, the use of the TCP flags and sequence number is also taken into account, and the end of a flow can be clearly defined. This step is delegated to the library libflowmanager [17], which also handles reordering of out-of-order TCP packets. One difficulty of bi-directional flows characterization is to determine the server and the client side in case the capture does not contain all the packets of the flows. This was solved by some heuristic function based on TCP flags and use of ephemeral port numbers which works in most cases.

The second step is to assign each flow to a certain application class. This means determining which application layer protocol is used for each flow. Because automatic classification and deep packet inspection isn't the main focus of this work, we based our tool on the existing packet inspection library libprotoident [18]. This library bases its approach on the analysis of the first four bytes of the packet payload (application layer) observed in each direction, as well as the packet size. The library is advertised to support more than 200 application protocols based on TCP or UDP. In case a protocol isn't identified, we use the destination port number and the transport layer name to classify applications as a fallback scheme.

Once flows are classified, the last step is to build and populate for each application the RENETO structural model previously presented. For this first version of our tool, we decided to use the Empirical Cumulative Distribution Function (Empirical CDF or ECDF) for representing each parameter previously characterized. The empirical cumulative distribution function $F^n_X(t)$ of the $(x_1, \ldots, x_n)$ variates of X is defined as:

$$F^n_X(t) = \frac{1}{n}\sum_{i=1}^{n} 1\{x_i \le t\} \quad (1)$$

with $1\{x\}$ the indicator function, which returns 1 if x is true and 0 otherwise.

When the analysis is finished, each parameter of the model is then saved in an XML file. We approximate the inverse of the ECDF by evaluating it at equi-distributed points according to Algorithm 1. We define $expValues$ as an array storing the experimental values, and N_S the number of intervals we wish to have. N_S will determine the precision of the generated number compared to the original distribution. The default value of N_S in the tool is 100. We study in Section 5 the impact of N_S on the accuracy of the generated numbers.

We choose to record the inverse of the ECDF because of the method we used for the random-number generation described in Section 4.1, which involves the inverse transform sampling method.

Algorithm 1 Inverse empirical cumulative distribution

```
 1: function buildAndSampleIECDF(expValues, N_S)
    ▷ Applies only if N_S ≤ length(expValues)
 2:     iecdf ← newArray()
 3:     sort(expValues)
 4:     n ← length(expValues)
 5:     iecdf.insert(expValues[0])
 6:     count ← 1
 7:     i ← 1
 8:     while i ≤ n do
 9:         a ← floor((i · N_S)/n)
10:         if a = count then
11:             iecdf.insert(expValues[i])
12:             count ← count + 1
13:         end if
14:         i ← i + 1
15:     end while
16:     iecdf.insert(expValues[n − 1])
17:     return iecdf
18: end function
```

We described before that our model includes two correlated variables (packet size and inter-arrival time), which means that we need a bivariate CDF. We reduce this bivariate problem to two univariate problems by using the conditional distribution method. This is summarized in the following equation, where X_1 and X_2 are the two variables to record:

$$\begin{aligned} F^n_{X_1,X_2}(t) &= F^n_{X_1}(t) \cdot F^n_{X_2|X_1}(t) \\ &= F^n_{X_2}(t) \cdot F^n_{X_1|X_2}(t) \end{aligned} \quad (2)$$

By using such decomposition, we are able to record the first parameter (X_1) using the function described in Algorithm 1, and then we can record the second parameter using multiple ECDF indexed by the values of X_1. We also perform the same decomposition by using X_2 as the first parameter.

While this method for storing and generating correlated random number proved to be working well in our tests with

limited values of X_1, it is a point of improvement for future versions of RENETO.

At this point of our research we consider the parameters of our model to be stationary. In other words, this means the method explained here isn't well suited for live capture where the parameters may vary between the different times of the day.

3.3 RENETO XML-based file format

Because our approach is based on two distinct tools, we need an exchange file format between the two tools, which was based on Extensible Markup Language (XML).

The XML description of a RENETO traffic model always starts with the tag `<reneto_model>` and is ended by `</reneto_model>`. For each parameter presented in Table 1, we define a tag and save the inverse of the empirical distribution function of the parameter.

We present here the format used:

```
<?xml version="1.0"?>
<reneto_model ...>
    <numSession>...</numSession>
    <interSession>...</interSession>
    <numFlow>...</numFlow>
    <interFlow>...</interFlow>
    <numPairs>...</numPairs>
    <reqSize>...</reqSize>
    <respSize>...</respSize>
    <interReq>...</interReq>
    <interResp>...</interResp>
    <reqSize_interReq>...</reqSize_interReq>
    <respSize_interResp>...</respSize_interResp>
</reneto_model>
```

4. TRAFFIC GENERATOR

We describe in this section the modules which were developed for generating traffic in OMNeT++ as well as some underlying points for generating traffic.

4.1 Random number generation

In Monte Carlo simulations, random number generation is a core principle. Our goal here is to generate pseudo-random numbers following the same statistical distribution than the recorded parameters by RENETO Trace Analyzer. Some key requirements for this pseudo-random number generator are: produce values which pass tests for randomness, be based on a seed in order to be able to reproduce the experiments, and have a low complexity as it will be used often in the context of simulation.

As stated before, each parameter is recorded using the inverse of the empirical cumulative distribution (IECDF). One advantage of representing the distributions as cumulative distribution function is the possibility to use the inverse transform sampling method [7, Section 2.2] for generating random numbers following the same initial distribution. Figure 2 shows the basic principle of the inverse transform sampling method: we generate a random number u from a standard uniform distribution in the interval $[0, 1]$, and we find the value x_{gen} such that $ecdf(x_{gen}) = u$. The inverse transform sampling method transforms a uniform random number generator into a non-uniform random number generator.

The major drawback of this method for generating non-uniform random variates is that we need to know the inverse of the CDF. In our case, we sampled the inverse of

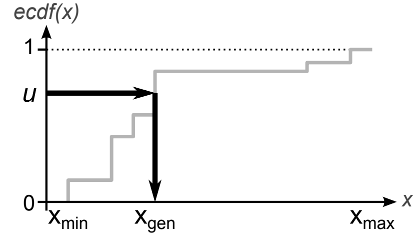

Figure 2: Inverse transform sampling method applied to a discrete distribution

the ECDF at several equidistant points as explained in Algorithm 1. We then use one of the algorithms described in Algorithm 2 to generate pseudo-random numbers. In effect, the algorithm only accesses the IECDF table and returns the retrieved value, with piecewise constant interpolation by default, or linear interpolation if needed, making it an efficient algorithm.

Algorithm 2 Inverse transform sampling algorithm

1: **function** GETVARIATE(*iecdf*)
▷ *Version with piecewise constant interpolation (Default)*
2: $u \leftarrow$ randomUniformNumber(0, 1)
3: $i \leftarrow floor(u \cdot N_S)$
4: **return** $iecdf[i]$
5: **end function**

6: **function** GETVARIATE(*iecdf*)
▷ *Version with a linear interpolation*
7: $u \leftarrow$ randomUniformNumber(0, 1)
8: $i \leftarrow floor(u \cdot N_S)$
9: **if** $i = N_S$ **then**
10: **return** $iecdf[i]$
11: **end if**
12: **return** $iecdf[i] + (iecdf[i+1] - iecdf[i]) \cdot u \cdot (N_S - i)$
13: **end function**

We will detail in Section 5.2 if other interpolation method of the inverse of the ECDF would be more beneficial to our generator than the method we used here.

We used the pseudo-random generator provided by OMNeT++ as our source for uniformly distributed random numbers, which by default is the commonly used MT19937 [12] Uniform Pseudo-Random Number Generator. This pseudo-random number generator has a period of $2^{199317} - 1$ and is as fast or faster than the `rand` function from ANSI C, which makes it suitable for Monte Carlo simulations. We used the default seeds provided by OMNeT++ for this generator.

4.2 Packet generation process

In order to generate packets, we follow the same process defined for the trace analysis in Section 3.2, but in the reverse order. More precisely for the client side, we start by simulating the user layer, which instantiates sessions, and the session layer then instantiates the flows. This process is presented in Algorithm 3. The function GETVARIATE(model.X) corresponds here to the generation of a sample of the parameter X of the model, using the method described in Section 4.1.

The last step is to generate the packets. This task is handled by the flow and packet layers, which send the packets to the UDP layer, respectively TCP layer, as presented in Algorithm 4, respectively Algorithm 5.

Algorithm 3 Simulation of user and session layers

User layer
1: **function** SIMULATEUSER(*model*)
2: $numSession \leftarrow$ GETVARIATE($model.N_{session}$)
3: $t \leftarrow$ getCurrentSimulationTime()
4: **while** $numSession > 0$ **do**
5: At time t call SIMULATESESSION(*model*)
6: $t \leftarrow t +$ GETVARIATE($model.I_{session}$)
7: $numSession \leftarrow numSession - 1$
8: **end while**
9: **end function**

Session layer
10: **function** SIMULATESESSION(*model*)
11: numFlow $\leftarrow$ GETVARIATE($model.N_{flow}$)
12: $t \leftarrow$ getCurrentSimulationTime()
13: **while** numFlow > 0 **do**
14: At time t call SIMULATEFLOW(*model*)
15: $t \leftarrow t +$ GETVARIATE($model.I_{flow}$)
16: $numFlow \leftarrow numFlow - 1$
17: **end while**
18: **end function**

Algorithm 4 Simulation of flow and packet layers : UDP

1: **function** SIMULATEFLOW(*model*)
2: $numPairs \leftarrow$ GETVARIATE($model.N_{pairs}$)
3: $t \leftarrow$ getCurrentSimulationTime()
4: **while** $numPairs > 0$ **do**
5: $packetSize \leftarrow$ GETVARIATE($model.reqSize$)
6: At time t emit packet with size *packetSize*
7: $t \leftarrow t +$ GETVARIATE($model.I_{req}|packetSize$)
8: $numPairs \leftarrow numPairs - 1$
9: **end while**
10: **end function**

Algorithm 5 Simulation of flow and packet layers : TCP

1: **function** SIMULATEFLOW(*model*)
2: $numPairs \leftarrow$ GETVARIATE($model.N_{pairs}$)
3: **while** $numPairs > 0$ **do**
4: $packetSize \leftarrow$ GETVARIATE($model.reqSize$)
5: Emit packet with size *packetSize*
6: Wait for reply from the server
7: $numPairs \leftarrow numPairs - 1$
8: **if** $numPairs > 0$ **then**
9: Wait for GETVARIATE($model.I_{req}|packetSize$)
10: **end if**
11: **end while**
12: **end function**

We presented in Algorithm 4, respectively Algorithm 5, the version of our algorithm with the correlation between packet size and inter-arrival time as shown on line 7, respectively line 9.

In our model definition, the server side corresponds to the entity replying to requests from clients. In other words, the server waits for request messages and responds with reply messages. This means that only the packet and flow layer is simulated on the server side, with algorithms similar to the ones presented for the client side.

4.3 OMNeT++ modules

The target simulator of RENETO is OMNeT++ [2], and its framework INET [1] which contains models for several protocols commonly found in a network. For our target application we focus on the following models provided by INET: Ethernet, IP, ARP, UDP and TCP.

One client module and one server module is available

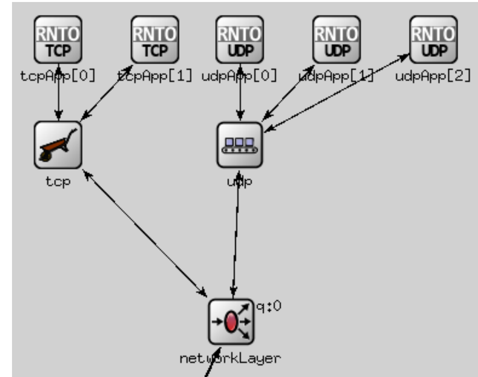

Figure 3: Screenshot of the OMNeT++ `StandardHost` model with 2 TCP and 3 UDP RENETO applications

for TCP, respectively UDP. They implement the *ITCPApp* module interface, respectively *IUDPApp* module interface, defined in INET, making them compatible with the TCP, respectively UDP, stack of INET.

An example of such model is presented on Figure 3, where we used the `StandardHost` model and attached TCP and UDP applications.

5. EVALUATION AND VALIDATION

We evaluated our tool on two common public traces for reproducibility and comparison purpose. We also used one of our own traces to demonstrate the capabilities of our tool with UDP traffic, which correspond to an industrial use case.

We used the LBNL-FTP-PKT trace [13] as a first benchmark for our tool. This trace is composed of more than 3 million anonymized packets of FTP control request and responses (no data transfer). All packet information is available (also data) such that the packet inspection and protocol classification library can be used.

The second trace which was used is the 3-day ACM SIGCOMM'01 conference capture [4] composed of more than 16 million packets. In this capture, packets were truncated to record only up to the transport layer and exclude all application layer for anonymization purpose. For this case, packet classification is based on port numbers.

Finally, the last trace used for this evaluation is a trace of SNMP traffic made on our own testbed. It is composed of 1426 packets.

5.1 RENETO Trace Analyzer

Packet classification.

As we know exactly what is in the LBNL-FTP-PKT trace (FTP control packets), we are able to evaluate if the protocol identification library chosen for this tool is efficient and correct. Table 2 gives the result of the tool for the LBNL-FTP-PKT trace. As we can see less than 1% of the packets were misclassified. Some packets with a server port of 21 couldn't be identified and others were misclassified as SMTP, because FTP and SMTP share some commands and reply codes. A misclassified flow can be linked to a flow which has been cut due to the start or the end of the capture in the middle of the flow.

Protocol	Number of packets	Expected
FTP Control	3231941 (99.0%)	100%
SMTP	4549 (0.14%)	0%
TCP Port 21	27462 (0.84%)	0%

Table 2: Packet classification result of LBNL-FTP-PKT trace

Trace	Number of packets	Execution
SNMP	1.426	0.05 s
LBNL-FTP-PKT	3.264.050	34.5 s
SIGCOMM'01	16.329.537	195.3 s

Table 3: CPU execution time of RENETO Trace Analyzer

For the second trace, as packets were truncated to remove all layers above layer 4, no protocol identification could be used. Instead, as a first idea for packet classification, we used the server port number. 2227 different server ports were reported by our tool.

For our last trace, SNMP was correctly identified for all the packets by the protocol identification library.

Execution time.

The CPU execution time was measured on a computer equipped with an Intel Core2Duo E8400 at 3 GHz. It is summarized in Table 3. As we can see, the execution time is almost linear in the number of packets.

5.2 Pseudo random number generation

As the pseudo-random number generator is a core element for the generation of network traffic, we evaluate it here and also look at the number of intervals N_S needed to achieve a good precision.

Continuous distributions.

We use the u-error as recommended in [9] for evaluating our pseudo random number generator, which is defined as:

$$\epsilon_u(u) = |u - F(G^{-1}(u))| \quad (3)$$

for $u \in [0,1]$, with F the cumulative distribution function, and G^{-1} the approximation of the inverse cumulative distribution function. We can see that if $G^{-1} = F^{-1}$, we will get an error $\epsilon_u = 0 \ \forall u \in [0,1]$.

We used the math library of boost[1] for performing our tests, which provides a direct access to the CDF and the inverse CDF of a wide range of well-known distributions. We first evaluated the precision of the CDF and ICDF functions by using the u-error previously described in Equation 3, with $G^{-1} = F^{-1}$. Results showed an u-error below 10^{-15} for the tested distributions.

We then evaluated our method for generating pseudo-random numbers based on a simple linear interpolation. We directly filled the internal IECDF table of our generator with evaluated values of the exact ICDF values provided by the library. We can then evaluate how many samples are necessary in order to reach a certain maximal u-error. We evaluated the u-error in 1000 different points chosen randomly according to a uniform distribution and stored the maximal value of u-error. Results are presented in Figure 4 in case of a normal distribution. Results for other distributions are similar.

[1] http://www.boost.org/libs/math/

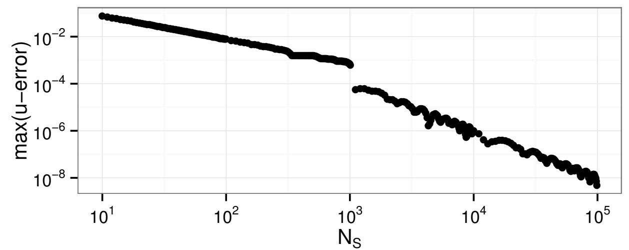

Figure 4: Number of samples N_S needed for achieving a maximal u-error using an exact ICDF

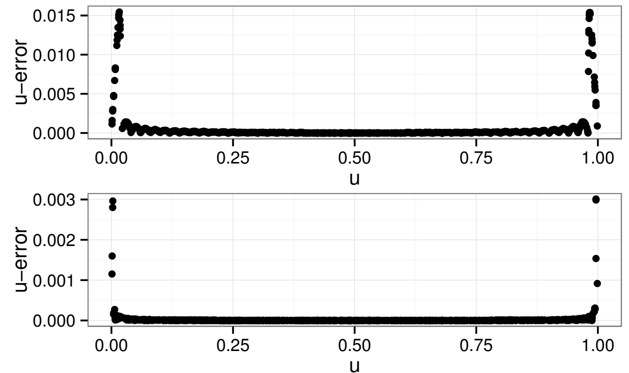

Figure 5: u-error for $N_S = 50$ (top) and $N_S = 250$ (bottom)

We then evaluated the complete toolchain of our pseudo-random number generator, namely generate observations from a known distribution, store them in our generator, and generating pseudo-random numbers based on those observations. The observations are generated using the inverse transform method with the exact ICDF provided by the library boost.

We evaluate the u-error in 1000 uniformly distributed points in the u-scale, as presented in Figure 5, with a number of sample N_S of 50 and 250. A normal distribution was used for generating those plots. As we can see on the figure, with a small number of sample N_S the tails of the distribution, where u is near 0 and 1, have a larger error than the middle, due to the linear interpolation which does not fit the tails of a normal distribution. Such pitfall is to be expected because we sampled the IECDF curve with a fixed interval size.

Finally, we evaluated the number of sample N_S needed for reaching a maximal u-error as presented in Figure 6. We can see that we reach a similar accuracy than for Figure 4 for $N_S < 500$ and we reach a minimal achievable error of 10^{-3} for the $N_S \geq 500$.

Discrete distributions.

Because of the discontinuities in the CDF of discrete distributions, the u-error previously used cannot be reused as it is for measuring the error in discrete distributions. We modify the u-error and introduce here the ud-error ϵ_{ud} which

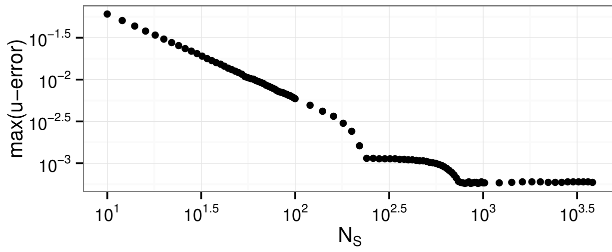

Figure 6: Number of samples N_S needed for achieving a maximal u-error using observations of the distribution

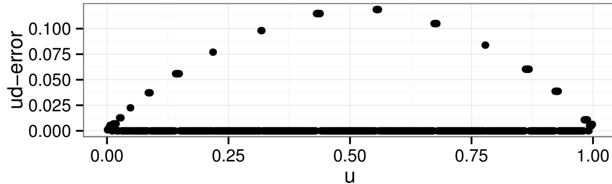

Figure 7: ud-error for $N_S = 100$

applies to discrete distribution with the following equation:

$$\epsilon_{ud} = |F(F^{-1}(u)) - F(G^{-1}(u))| \quad (4)$$

We evaluated the ud-error with a binomial distribution. Results are presented on Figure 7, where we can see that errors happen around the discontinuity points, while for the rest the error is exactly 0. This result can be explained by the fact that we used here regular intervals to sample the curve, and not variable intervals which could match the discontinuity points.

Finally, we made a comparison between different interpolation methods to see if the method we chose was relevant enough, using confidence intervals as presented in [11, Chap. 10]. This method is similar to the evaluation of the absolute x-error ϵ_x presented by the authors of [9] and defined as:

$$\epsilon_x = |F^{-1}(u) - G^{-1}(u)| \quad (5)$$

We used here the distribution of request message size from the LNBL-FTP-PKT trace, which can be seen in Figure 9, and the interpolation methods provided by GNU-R. Like for the u-error, we evaluated the generated number from the inverse transform sampling method in 1000 uniformly distributed points in the u-scale, using 3 different size of samples N_S. Results are presented on 8.

The use of a piecewise constant interpolation brings better results than other interpolation method as the difference is around 0 for the evaluated N_S values. This result confirms our choice for a piecewise constant interpolation for discrete distributions.

5.3 RENETO Traffic Generator

For this evaluation, we generate traffic between a certain number of clients, and one server. Each client is directly connected to the server with an 100 Mbits/s Ethernet connection.

Generation of FTP Control traffic.

For this part, we generated traffic according to the FTP Control model extracted from the LNBL-FTP-PKT trace

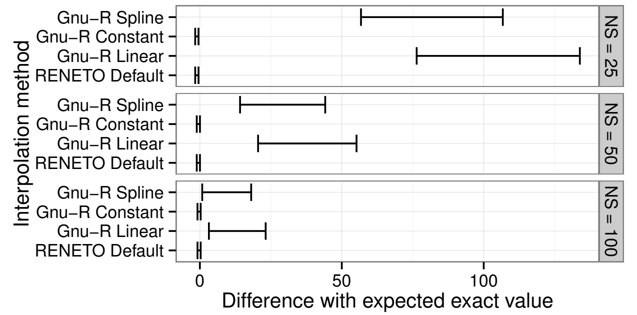

Figure 8: Comparison of interpolation function for a discrete distribution

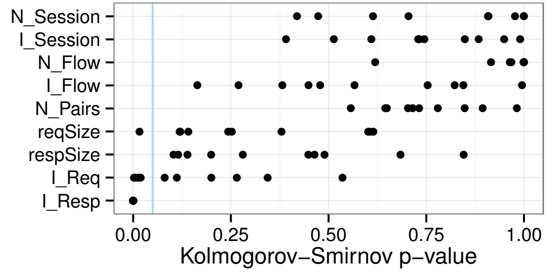

Figure 10: Result of a two sample Kolmogorov-Smirnov test between original trace and simulation with 10 simulation runs

and 50 clients were simulated.

Figure 9 presents the different parameters of the model using the Empirical Cumulative Distribution Function as a representation. On the same plots, we have the model of the original capture (plain line), as well as the model of the simulation (dashed line).

With a first visual verification, we can see that the simulation reproduces accurately the parameters recorded by the analysis tool. We performed a two sample Kolmogorov-Smirnov test for each parameter, which is a two-sided test for the null hypothesis that the two cumulative distribution functions are drawn from the same continuous distribution. Results are presented in Figure 10 with different simulation runs, meaning that the pseudo-random number generator has a different seed at each run. We can see from the p value and with a significance level of $\alpha = 0.05$, that the difference between the model and the simulation is not significant enough to say that they have a different distribution, except for the I_{resp} and I_{req} parameters which are highly dependent on where the trace is made on the network as our model does not account the topology of the original trace.

Generation of SNMP traffic.

For this part, we generated SNMP traffic according to the SNMP model extracted from our own trace. This trace has the advantage of containing some patterns regarding packet size and inter-arrival time. We show on Figure 11 this correlation between packet size and inter-arrival time, where we see the benefit of this introduced correlation for the simulation. Generating packets in the lower-right part of the plot (large packets with a small inter-arrival time) would generate more bandwidth utilization than in the original capture.

Finally we also investigated the ability of our model to

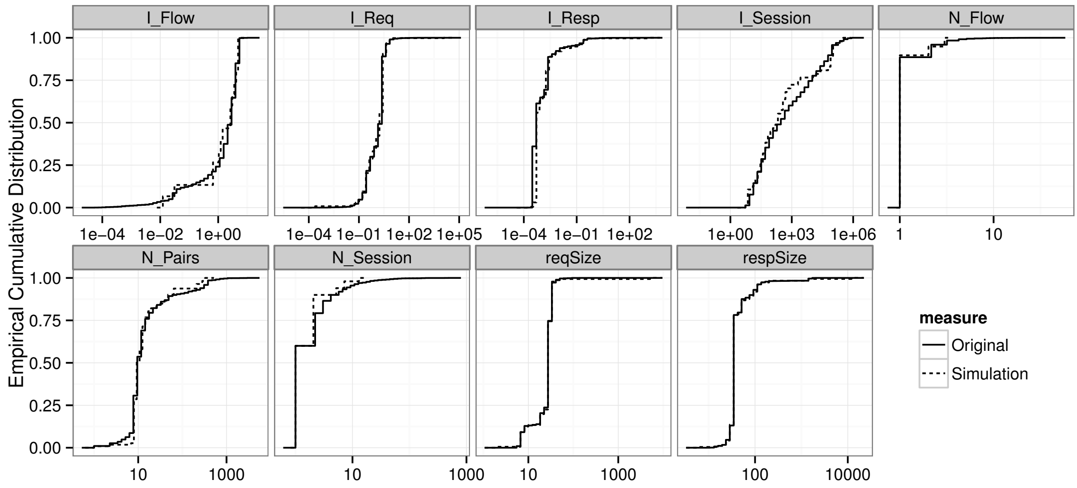

Figure 9: Empirical Cumulative Distributions of the different parameters of the model after analysis of the LNBL-FTP-PKT trace, and simulation based on the model. Time are given in seconds, and packet size in Bytes

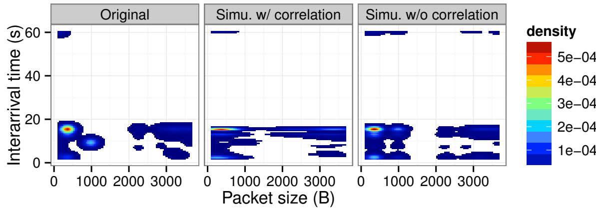

Figure 11: Correlation between inter-arrival time and packet size for the SNMP trace

reproduce Long-Range Dependence or Self-Similarity found in the initial trace. We used the Logscale Diagram Estimate or LDE, as presented in [3], based on discrete wavelet transform. This tool was used to match the packet arrival time series. We use it as a comparison tool between the original capture, and the simulated traffic.

By performing a linear regression on the curve, we compute the scaling exponent α. The analysis of the original traffic resulted of an $\alpha = 2.25$, which suggests self-similar traffic as α is greater than 1.

The LDE of the capture and the 10 runs of the simulation is presented on Figure 12. Each simulation run corresponds to a different seed for the pseudo-random number generator. We can see from the plot that the seed has a limited influence on the variance.

As exposed in the Figure 11, we separated here the plots to see if the use of the correlation between parameters improves the quality of the simulated traffic regarding self-similarity. We can see that the use of the correlation brings more precision to the simulation than without.

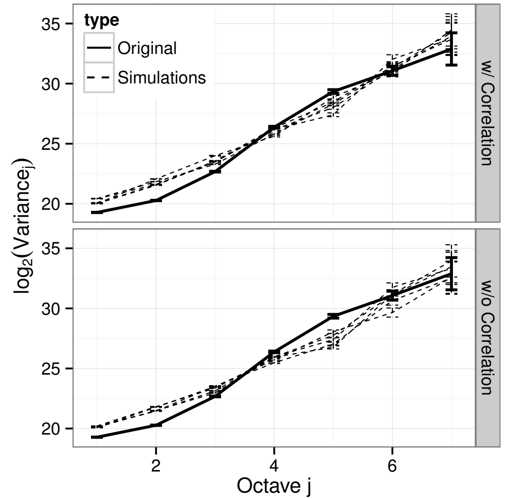

Figure 12: Logscale Diagram Estimate of the original traffic and the generated traffic in case of SNMP

6. CONCLUSION AND FUTURE WORK

We presented in this paper RENETO, a Realistic Network Traffic generator for OMNeT++/INET, along with the two tools which form RENETO.

We developed a structural traffic model which is based on four layers: packets, flows, sessions and users. Our method is based on the analysis of a traffic capture, where the parameters of our model are recorded using their empirical cumulative distribution. Based on this analysis our OMNeT++ modules are able to reproduce traffic which will follow the distributions previously recorded.

Our tools were evaluated in this paper against three captures and we created two types of synthetic traffic: FTP Control and SNMP. We showed that the generated traffic was comparable to the traffic of the original capture. We also evaluated the internal pseudo-random number generator in order to see if it was suitable for our use-case.

We showed the benefit of the correlation between packet size and inter-arrival time on UDP traffic. Models developed with this tool will serve as a base for future studies on scheduling and buffer utilization in switches, which will require realistic traffic models.

7. ACKNOWLEDGMENTS

We would like to thank Alexander Klein for his comments and suggestions in the early days of the work, and Emanuel Heidinger for his constructive remarks.

8. REFERENCES

[1] INET Framework for OMNeT++/OMNEST. http://inet.omnetpp.org/, Accessed 01.03.2012.

[2] OMNeT++ 4.2.2. http://www.omnetpp.org, Accessed 01.09.2012.

[3] P. Abry, P. Flandrin, M. Taqqu, and D. Veitch. Wavelets for the analysis, estimation and synthesis of scaling data. *Self-Similar Network Traffic and Performance Evaluation*, pages 39–88, 2000.

[4] A. Balachandran, G. Voelker, P. Bahl, and P. Rangan. Characterizing user behavior and network performance in a public wireless LAN. In *ACM SIGMETRICS Performance Evaluation Review*, volume 30, pages 195–205. ACM, 2002.

[5] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '98/PERFORMANCE '98, pages 151–160, New York, NY, USA, 1998. ACM.

[6] M. Bohge and M. Renwanz. A realistic VoIP traffic generation and evaluation tool for OMNeT++. In *OMNeT++ 2008: Proceedings of the 1st International Workshop on OMNeT++ (hosted by SIMUTools 2008)*, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[7] L. Devroye. *Non-uniform random variate generation*, volume 4. Springer-Verlag New York, 1986.

[8] F. Hernandez-Campos, K. Jeffay, and F. D. Smith. Modeling and generating TCP application workloads. In *BROADNETS 2007 - Fourth International Conference on Broadband Communications, Networks and Systems*, pages 280–289, September 2007.

[9] W. Hörmann and J. Leydold. Continuous Random Variate Generation by Fast Numerical Inversion. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 13(4):347–362, 2003.

[10] K. V. Jónsson. HttpTools: a toolkit for simulation of web hosts in OMNeT++. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, SIMUTools '09, pages 70:1–70:8. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009.

[11] A. M. Law and D. W. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill Higher Education, 3rd edition, 2000.

[12] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[13] R. Pang and V. Paxson. A High-level Programming Environment for Packet Trace Anonymization and Transformation. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 339–351. ACM, 2003.

[14] F. Ricciato, A. Coluccia, A. D'Alconzo, D. Veitch, P. Borgnat, and P. Abry. On the Role of Flows and Sessions in Internet Traffic Modeling: An Explorative Toy-Model. In *Global Telecommunications Conference, 2009. GLOBECOM 2009. IEEE*, pages 1–8, December 2009.

[15] C. Rolland, J. Ridoux, and B. Baynat. LiTGen, a Lightweight Traffic Generator: Application to P2P and Mail Wireless Traffic. In S. Uhlig, K. Papagiannaki, and O. Bonaventure, editors, *Passive and Active Network Measurement*, volume 4427 of *Lecture Notes in Computer Science*, pages 52–62. Springer Berlin Heidelberg, 2007.

[16] K. V. Vishwanath and A. Vahdat. Realistic and Responsive Network Traffic Generation. In *Proceedings of ACM SIGCOMM 2006*, SIGCOMM '06, pages 111–122, New York, NY, USA, 2006. ACM.

[17] WAND Network Research Group. libflowmanager. http://research.wand.net.nz/software/libflowmanager.php, Accessed 01.09.2012.

[18] WAND Network Research Group. libprotoident. http://research.wand.net.nz/software/libprotoident.php, Accessed 01.09.2012.