BRUTE: A High Performance and Extensible Traffic Generator

N. Bonelli, S. Giordano, G. Procissi and R. Secchi

Department of Information Engineering
University of Pisa
Via Caruso, I-56122 Pisa, Italy

{s.giordano,g.procissi,f.russo,raffaello.secchi}@iet.unipi.it

Abstract- Evaluating the performance of high speed networks is a critical task due to the lack of reliable tools to generate traffic workloads at high rates and to the inaccessibility of network equipments. To address the problem of reliable device testing, we developed an extensible traffic generator, called BRUTE (Brawny and RobUst Traffic Engine), running on top of Linux operating system. BRUTE takes advantage of the Linux kernel potential in order to accurately generate traffic flows up to very high bitrates. This software is an extensible framework that provides a number of optimized functions that easily allow the implementation of ad hoc modules for customized traffic patterns generation. A number of library modules implementing common traffic profiles (like CBR, Poisson process and Poissonian Arrival of Burst process) are already implemented and integrated into the package. Performance of BRUTE has been compared to the one of several widespread traffic generators over Fast-Ethernet and Gigabit-Ethernet networks. The results of our tests show the superior performance of BRUTE in terms of both the achieved throughput and the level of bitrate precision, under all possible values of the frame length.

Keywords: High Capacity Traffic Generation, Extensible Framework, Performance Tool.

1. INTRODUCTION

As computer networking evolves towards ubiquitous high speed environments, the need to evaluate the performance of high speed network architectures is becoming a central problem in network research. Though the analysis of high speed network devices could be faced with the aid of simulative instruments, such as the largely diffused ns2, the complexity and heterogeneity of such systems makes difficult to reproduce accurately their behavior [1][2]. In this sense, the paradigm of emulation by means of synthetic traffic traces over real networks testbeds could be the only viable direction to overcome the unavoidable simplifications arisen by simulations. As a support to this methodology, we propose the Brawny and RobUst Traffic Engine framework (BRUTE) [3], which allows the generation of flows with high level bitrate precision. As an user space application running on top of Linux operating system, BRUTE demonstrates the capabilities of the kernel 2.4/2.6 in high speed packet generation and gives the opportunity to employ high-end personal computers to perform tasks today only prerogative of specialized hardware.

In the implementation of a traffic generator, we must overcome a set of problems common to many real-time systems. In first place, the impact of latency induced by operative system calls on application layer is generally an unknown parameter during design phase. Since the dynamics of data delivery are hidden beyond the socket interface, the same application may have different responses over the different network architectures. Furthermore, the real-time requirements of network applications cannot be guaranteed in operating systems without real-time extensions (RTOS) support. As a consequence, the implementing choices for a network application represent the most critical issue to construct a high performance tool and may determine the range of applicability of program outcome. BRUTE accomplishes the high performance requirements by using POSIX.1b FIFO processes, natively implemented in Linux kernel scheduler and using sockets that, working directly on data-link layer, are able to reduce the latency experienced by outgoing frames. Moreover, BRUTE exploits the timestamp counter (TSC) register built inside the most of modern CPUs to achieve best temporal accuracy [9].

The problem of reliable real-time traffic generation is not new in networking related research and several software solutions already exist (see for instance [4][5] for a comprehensive overview on traffic generators). The novelty of our approach is to select the set of Linux features which mostly influence the behavior of the networking subsystem at high rates and exploit such characteristics to create a basement for the traffic generator architecture. More important, the design of BRUTE architecture is oriented to extensibility. That is, differently from the other traffic generators, BRUTE allows the development of modules containing the definition of new traffic models in C language with the support of an ad-hoc application program interface (API). When generating traffic BRUTE aims at
being RFC 2544 [1] compliant. The payload of UDP packets is filled as described in the appendix C of the memo (udp-echo request). Also, it is given the opportunity to use random destination addresses with an uniform distribution over 256 hosts, or the less significant 64 bits of address when IPv6 is in use. A distribution over the entire MAC address space is provided to evaluate the performance of network layer-II devices, such as bridges and switches.

The rest of the paper is organized as follows. In section 2, we discuss the implementation choices that allow satisfying real-time constraints at high rates. In section 3, we describe the most important components of BRUTE and their respective implementation. In section 4, we report an extensive comparative performance analysis with other tools and in section 5 we report final remarks.

2. DESIGN & IMPLEMENTATION

In this section, we discuss the rationale of BRUTE design. We focus on the implementing choices that are relevant to the tradeoff between performance and accuracy of traffic tool. Moreover, we show that BRUTE extensible architecture is not in contrast with its design built for performance.

Basically, a networking application can be developed as an user process, referred to as user-space application, or as a part of the Linux kernel, called kernel-space module. While the user-space applications are chosen for their portability and ease of debugging, the kernel-space modules should be used to satisfy real-time requirements. Indeed, the kernel-space processes are insensitive to the latency induced by the system-call mechanism and are not involved in the process of normal task scheduling. As we will see in the following discussion the performance achievable from kernel-space modules are in practice the same obtained by user-space applications. Thus, we developed BRUTE as an user-space application, which has the high flexibility of user-space design by means of the explicit support of extensible interfaces.

To optimize the performance of BRUTE, we have to consider three issues: the accuracy of timers, the network socket family and the policy of process scheduling.

Timer Accuracy

When developing a traffic generator, the method used for the measurement of time intervals is critical. To make robust the implementation of BRUTE, we use a busy-wait polling procedure. This procedure consists in continuously probing the timing resource till the break out of the event. Also, we need a reliable reference clock to measure time with sufficient accuracy. The POSIX standard C-library, available in the most common platforms, provides the system-call gettimeofday to access the timing resource. However, we avoid the use of such routine for various reasons. First, it provides a resolution of about one microsecond, which is not adequate to measure time intervals in high speed environments. Second, this function is characterized by a large overhead, which delays the starting time of the next statement. Finally, the execution of the gettimeofday system-call may trigger a switching of context, which introduces an extra delay to the latency of the gettimeofday itself. This occurrence is particularly heavy when used in a busy-wait polling loop.

The most of modern processing units (CPUs) are provided with a register devoted to count the number of CPU clocks. In the Intel Pentium family, this register is called time stamp counter (TSC) register and can be accessed via the instruction read time stamp counter (rdtsc), which is a non-serializing instruction. This means that the execution of rdtsc may not follow the sequential order and, hence, the getting of the current time may be delayed [7]. As a consequence, the use of rdtsc leads to a timer resolution proportional to the inverse of CPU clock rate, but in a precision of few tens of CPU clocks. Fig. 1 shows a comparison of the mean time required by a user-space application to access the timing resource using the gettimeofday system-call and the one required by accessing directly to the TSC register.

We note that the number of CPU clock cycles needed by rdtsc is more than one order of magnitude less than those needed by calling the gettimeofday function. This makes the use of rdtsc suitable to be implemented in a busy-wait loop.

Socket family

A second important problem to face is the choice of the socket family. Typically, for applications which have not stringent timing requirements, this choice is based on functional criteria. However, in the case of high speed traffic generation, a wrong choice of socket family may drastically reduce the throughput achievable from user-space. For our purpose, we analyzed two family of sockets available in Linux kernel 2.4: the INET and the PACKET socket families.

The structures of a socket are instantiated through the system call mechanism when an user-space application invokes the sendto function. Though the first part of control path is common to all the protocol families, the function that handles the software interrupt (sock_sendmsg) redirects the call towards a different control path on the basis of socket protocol family.
Since the PACKET sockets expect that the message from user-space is a complete Ethernet frame, the control path consists only in a short function (packet_sendmsg), which is in charge of setting up and serialize the frame to the outgoing queue. On the other hand, the INET socket family performs the set of operations required to building the UDP/IP and MAC headers in that the message retrieved from user-space contains only the UDP payload. In particular, INET socket is in charge of checking if socket is connected, query the routing table for the associated destination, allocating the kernel structure encapsulating the frame (sk_buff structure) and building the frame with the information passed by the calling function. Moreover, INET socket encompasses the Netfilter framework, which performs packet filtering, mangling and other operations related to address/port translation (NAT/NPT).

To evaluate the performance achieved by the two sockets, we have performed both internal measurements, storing packet timestamps into a large vector built inside the kernel, and external measurements, employing a high performance traffic analyzer (Spirent AX4000). Since recording a timestamp for each packet into the kernel vector introduces a negligible overhead, this method allows performing the least invasive measurement and precisely evaluating the time required to compete the UDP/IP control path.

The right fig. 1 shows the average time in CPU cycles units required to allocate and setup a packet (sk_buff object). The computation of the two sockets latency does not take into account the device latency, since it depends on the network driver employed. Also, we disabled any filtering operation performed by Netfilter, which has a negative influence on the performance of INET sockets only. As we could expect, the PACKET socket requires a lower time to route the packet to network driver.

However, the PACKET sockets should be selected for the development of a traffic generator also because they allow customizing the MAC and IP headers from the application layer. This feature makes possible, for instance, to generate flows with random MAC or IP destination addresses and characterize the behavior of network devices, such as routers and switches, according to the guidelines set in [1].

**Process Scheduling Algorithm**

As remembered previously, a traffic generator is required to satisfy hard real-time requirements. Although the Linux operating system does not come with true real-time extensions, the schedulers of the Linux kernel 2.4/2.6 allow to assign a different scheduling policy to each process in order to handle soft real-time applications. The scheduling policy applied by default to a normal process is time-sharing. This policy is used when the process does not have any real-time requirements. Instead, the first-in first-out (FIFO) and the round-robin (RR) scheduling policy are recommended for those applications, such as BRUTE, which need a strict control on the order of processes execution. With these scheduling policies, when a real-time process becomes executable, it is allowed to preempt any normal process under execution. However, the FIFO scheduler does not adopt time slicing for processes with the same priority. This means that, after the FIFO process is scheduled, it can be only blocked by either a I/O request or by the preemption of a higher priority real-time process. Since in our system BRUTE is the only real-time process, the use of FIFO or RR scheduling policy is equivalent in terms of performance.

3. **EXTENSIBILITY & MODULES**

As outlined in the introduction, the architecture of BRUTE is designed to be extensible. With the support of a programmable interface BRUTE eases the creation of traffic models. By using the application program interface (API) of BRUTE, the user writes the definition of a traffic model into a C-language module (T-module) without taking care of most of tasks required to develop a networking application (handling of packet headers and sockets, memory management, script parsing, etc.). This framework is not limiting even for the expert programmer, who has the possibility to fully customize the modules.

Fig. 2 shows the block diagram of BRUTE architecture. We have distinguished four components: the parser, the traffic engine, the programming support (API and macros) and the set of traffic modules (T-modules). The parser deals with the grammar and the lexical analysis of the script language. It consists of two parts: the core parser, which handles the grammar and part of the lexical analysis, and a set of user-

---

**Fig. 2 - The BRUTE architecture**

---

**SCRIPT**

**FILE**

**lexical**

**grammar**

**and part of the lexical analysis, and a set of user-**
defined micro-parsers which handle the lexical elements of a specific traffic model. Actually, the task of parsing could be accomplished by integrating the source code created by automatic parser generators, for instance Flex and Bison, into the BRUTE framework. However, the choice to place the generated code into the core parser implies to rebuild the parser and recompile the package at each attempt of extending BRUTE, which is in contrast with extensibility criteria.

On the other hand, the choice to place the generated code into a new traffic module requires the user to be in charge of writing the description of a parser in a separate module and to deal with a non-trivial language of parsing description. Therefore, the provided mechanism alleviates the user from the effort of encoding a parser.

As illustrated by figure, the parser collects data from the script file and stores the information into an internal database (structure mod_line). Then, the traffic engine examines the mod_line entries and, according to the kind the data read, instantiates the proper traffic handler (micro-engines functions) defined into the T-modules. All the user defined micro-engines are executed sequentially to generate the requested traffic.

Traffic Modules

Fig. 3 represents the prototype of a T-module, which contains the user-defined elements of BRUTE. Together with the set of token representing the input parameters, each T-module implements a command available in a BRUTE script. To create a new command, the programmer is just required to define the structure mod_line, which contains the list of tokens for a specific traffic model, and the micro-parser handler, which provides the portion of code devoted to the lexical parsing.

The micro-parser comes in the form of a switch statement, which is called every time a token is met during the parsing of a legal line of script. The handler gets as arguments the pointer to the input argument (atom), an integer that identifies the left-value type of the argument (token) and the instance of the mod_line structure. The programmer is just required to complete the switch statement with the proper list of cases and implement the proper parsing method according to the expected right-value types.

Finally, a T-module includes the portion of code that generate the traffic into the micro-engine handler. The programmer is required to add the body of the handler with the support of APIs and macros. This way, the user has not to worry about packets management, timers and memory allocation, while concentrating on the development of a new traffic model.

Typically, the micro-engine handler consists of two parts. The first part is devoted to the frame building, while the second is required to implement the dispatch-loop, which actually forwards the frames. In the dispatch-loop for steps are carried through:

I. brute_sendto statement, which forwards the packets to device driver
II. the portion of code updating the frame in order to obtain the subsequent (incrementing IP identifier field, computing the checksum, etc.)
III. the algorithm in charge of evaluating the departure time of the next packet
IV. the busy-wait routine which stop performing operations until the departure of subsequent packet

At the time of writing some of the T-module are already implemented

- **Constant Bit Rate.** The constant bit rate (CBR) module aims at generating frames with constant inter-departure times. Along with the common parameters used to build IPv4/v6 and UDP headers, the parameter rate is defined to specify the number of packet sent per second.

- **Poisson.** The Poisson module is used to generate frames with exponential distribution of inter-departure times. The input parameter lambda is given to define the mean number of packets to generate per second.

- **Poisson Arrival of Burst** The Poisson Arrival of Burst (PAB) is the process given by the superposition of CBR bursts scheduled according to a poissonian arrival process. It is implemented in the PAB module. The number of active bursts at a given time is a random variable having Poisson distribution independently on the distribution of the burst length. However, if the duration of each burst has a Pareto distribution. The PAB process exhibits long range dependence characteristics, which is often found in multimedia traffic traces. For this reason, the PAB process is suitable to emulate multimedia traffic. Along with the token rate defining the instantaneous burst rate, the command takes alpha and theta as parameters of Pareto distribution and lambda as mean number of burst arrivals per second.
Programming Script Language

To define the pattern of traffic generation, BRUTE is supported by a flexible script language. This language is organized in a list of statements, each occupying a single line. Each statement is made up of an optional label, a command identifier and a sequence of semicolon-terminated atoms, which represent the parameters of traffic class and can be listed in whatever order within a statement. The atoms, in turn, are structures composed by three elements: the token identifier, the symbol of assignment and the assigned value \((r\text{-value})\). An example of legal statement is given by the following line

\[
\text{lab: cbr msec=1000; rate=1000; daddr=10.0.1.10; len=512;}
\]

This command instructs the traffic engine to generate a 1Kbps CBR traffic flow with 512B long frames with the IP destination and source addresses substituted by the specified values.

When a statement is executed, the parser creates a new instance of command, which inherits from the last instance of the same command (parent) the traffic parameters that have not been explicitly assigned in the current statement. For instance, after that the CBR statement is executed, each subsequent CBR statement, without explicit definition of packet size, assume frames 512B long. Though there are no constraints in defining the list of legal tokens, in order to keep homogeneous the language, the user should adopt the semantic of preexisting modules.

Table 1 summarizes the list of tokens common to all the modules. The tokens belong to the user-defined list of identifier declared in a T-module. Atoms are parsed from the script file by means of the core parser and, according to the affiliate command, are passed to the related micro-parser in order to be lexically evaluated. The right values can be either static or dynamic, depending on the method used by the programmer. While static atoms admit only constant elements, dynamic atoms allow assignments in which also variables and functions can be used. A variable implements a dynamic object. Since the T-modules do not share objects, the variables are the only way to allow the inter-communication among modules. Within a token assignment it is also allowed the call of a user defined function.

As concerns the dynamic properties of the BRUTE script language, dynamic atoms admit also additive and subtractive assignments, useful in iterative procedures. For instance, the following portion of script describes an UDP traffic flow starting at the rate of 2Kpps and subsequently increasing the rate by 1Kpps at every second

\[
\text{cbr msec=1000; rate=1000; len=udp_data(18); lab: cbr msec=1000; rate+=1000; off msec=100; loop counter=10; label=lab;}
\]

where the \textit{off} command inserts a silent period of 100 microseconds between two consecutive CBR bursts. In this example, the statement \textit{loop} allows to jump iteratively to the specified label for a number of times defined by the token \textit{counter}.

Finally, BRUTE scripts allow the use of functions for several purposes. For instance, the \textit{udp_data} function in previous example allow to calculate, from UDP payload, the total frame length, including UDP/IP header, MAC header and the Ethernet CRC.

### 4. PERFORMANCE EVALUATION

In this section, we present the results of tests carried out in a test-bed running Linux kernel 2.4. With these tests we intend to determine the maximal performance of BRUTE and evaluate the stability and the reliability of the packet flows generated. Also, we present a comparison with others traffic generation tools widely used by the research community. Among kernel-space tools, we considered UDPGEN [11], an UDP traffic generator designed to achieve high performance over Gigabit-Ethernet. UDPGEN consists in a Linux kernel module able to operate directly on the network device driver bypassing the Linux kernel networking subsystem. Since UDPGEN architecture is strictly related to that of kernel, it lacks of extensibility and cannot take advantage of the support of kernel-space extensible interface. Moreover, we considered two user-space traffic generators, namely MGEN [10] and RUDE [8]. RUDE is a tool configurable with different patterns of traffic. It is provided with a non-extensible script language, which is able to instantiate more simultaneous patterns of traffic. However, the software architecture of both tools does not provide any explicit support for extensible interfaces and, as we will see later, these tools are not suitable to work at high rates, especially with small frames. We performed several measurements in two scenarios with different hardware. Our first scenario consisted in a single-processor PC equipped with a Fast-Ethernet network adapter, while the second in a dual-processor PC equipped with a Gigabit-Ethernet card. Table 2 summarizes the details of the employed hardware. The results of tests show that only BRUTE and UDPGEN are able to generate packets at the line-speed over Fast-Ethernet with the smallest frame-size.

**Performance over Fast-Ethernet**

In the first test, we determined the maximal throughput achievable with CBR traffic a Fast-Ethernet scenario. In the comparative tests, we increased the rate generated by each traffic tool by small steps in order to reach the upper performance limit. The traces were captured on wire by means of Spirent AX4000 traffic analyzer.

<table>
<thead>
<tr>
<th>token</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>msec</td>
<td>duration of statement (µsec)</td>
</tr>
<tr>
<td>saddr/sport</td>
<td>source address and port</td>
</tr>
<tr>
<td>daddr/dport</td>
<td>destination address and port</td>
</tr>
<tr>
<td>tos</td>
<td>IP type of service</td>
</tr>
<tr>
<td>ttl</td>
<td>IP time to live</td>
</tr>
</tbody>
</table>

Table 1 - List of common tokens
Fig. 5.a shows the throughput (in frame per time unit) of all considered tools with respect to the frame length. BRUTE, similarly to the kernel-space tool UDPGEN, is able to saturate the link capacity at each frame length. We recall that the maximum theoretical frame rate (1) is obtained by dividing the link capacity by the frame length increased by the inter-packet-gap (IPG) term (equivalent to 96 bit in Ethernet standard) and the 64 bits of synchronization preamble.

\[
\text{maxrate} = \frac{\text{link\_capacity}}{\text{frame\_length} + \text{preamble} + \text{IPG}}
\]  

(1)

We can notice that BRUTE outperforms MGEN and RUDE when generating traffic with small frame size. A similar behavior is also observed in Fig. 5.b, which depicts the percentage of average error rate as a function of frame length. While the throughput of BRUTE is unbiased at each rate, the other traffic generators are subjected to polarization errors even at low rates. Since BRUTE is a user-space application, these performance improvements must be attributed to the implementing choices described in previous section. Further, we observed an increased stability in rate generation. The graph of Fig. 4 shows that the standard deviation error versus the requested frame-rate is smaller in the case of BRUTE. This fact suggests that the performance of BRUTE are not obtained at expenses of rate generation accuracy.

**Performance over Gigabit-Ethernet**

The tool shows better performance in the Gigabit Ethernet scenario with respect to other user-space tools, both in terms of polarization error and achievable rate. Fig. 7.a shows the maximum throughput of all the considered tools as a function the frame length. In the worst-case frame size, BRUTE reaches about 650Kfps, while UDPGEN reaches 740Kfps. The latter, being a kernel-space application well justifies the difference in term of max. rate (which are only 13% higher than BRUTE). On the other hand, thanks to the implementation choices, BRUTE achieves about two times the throughput of others user-space traffic tools.

It is noteworthy that, while BRUTE and UDPGEN are able to saturate the whole link capacity with frames of 256 bytes, the other user-space tools require at least frames of 512B. The packet rate generated by BRUTE results unbiased in the range up to the maximal rate, with a bias error percentage less than 0.1% (Fig. 7.b).

Fig. 6 represents the standard deviation of the instant rate calculated using an adaptive running window whose size varies, according to the nominal rate, in order to include a

<table>
<thead>
<tr>
<th>Hardware employed in measurements</th>
<th>100Mbs</th>
<th>1Gbps</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Pentium IV 2.4GHz</td>
<td>Xeon 2.66GHz hiperth. disabled</td>
</tr>
<tr>
<td>Motherboard</td>
<td>Asus P4PE</td>
<td>SuperMicro X5D-PE-G2</td>
</tr>
<tr>
<td>Memory</td>
<td>512MB</td>
<td>1GB</td>
</tr>
<tr>
<td>Ethernet</td>
<td>3com 3c905C TX/TX-M</td>
<td>Intel PRO1000 LX (PCI-X)</td>
</tr>
</tbody>
</table>

| Table 2 Hardware employed in measurements |

---

SPECTS '05 844 ISBN: 1-56555-300-4
constant number of frames. The standard deviation is represented in frame per second. Again, the small amount of fluctuations around the average rate with as compared to the other tools (including the kernel-space UDPGEN) indicates the high level of stability in rate generation achieved by BRUTE.

5. CONCLUSIONS

We proposed BRUTE, a high performance tool for the generation of Ethernet traffic, IPv4 and IPv6 compliant. Developing BRUTE, we can show that general purpose personal computer is able to generate reliable high speed streams over both Fast- and Gigabit-Ethernet. BRUTE is not designed to be a pure packet generator. On the opposite, its extensible design, based on user-space, is reference architecture to develop tools to test high speed network devices. The modules are written with the support of APIs and macros which hides the user most of details concerning building frames, sockets handling and timers related issues. Moreover, the traffic engine is programmable through a user-level script language, which is used to instantiate a set of micro-engines in order to generate a sequence of traffic patterns. BRUTE script language is simple and flexible, since automatically extends commands and lexical properties each time the user add a new traffic modules. The software performance has been verified in Fast- and Gigabit-Ethernet scenarios. In both cases, BRUTE successfully generates flows at a considerably higher than other widespread user-space traffic tools. The flow generated by BRUTE is stable and unbiased in all the considered range. Also, the instante rate standard deviation, compared to that achieved by kernel-space traffic tool, is small.

References