The Journal of Supercomputing, 35, 5–26, 2006

High Performance Internet Traffic Generators

STEFANO AVALLONE stavallo@unina.it
DONATO EMMA doemma@unina.it
ANTONIO PESCAPÈ pescape@unina.it
GIORGIO VENTRE giorgio@unina.it
University of Napoli "Federico II", Italy

Abstract. In the networking field, traffic generator platforms are of a paramount importance. This paper deals with the description of a distributed software platform for synthetic traffic generation over IPv4/v6 networks, called D-ITG (*Distributed Internet Traffic Generator*). We point our attention on the original architectural choices and evaluate the performance achieved by the platform. D-ITG supports several protocols and many traffic patterns. We tested our generation platform over different scenarios and compared it to many of the currently available, and most widely adopted, traffic generators. We found that D-ITG offers enhanced functionalities and improved performance.

Keywords: high performance internet traffic generation, distributed and parallel architectures, performance evaluation

1. Introduction

Over the last twenty years, considerable effort has been made to understand and characterize the behavior of the Internet. The extreme complexity of large topologies and their traffic characteristics make the development of analytical models difficult. Under such conditions, simulation is the most promising technique for understanding network behavior. Simulation modeling of computer networks is an effective technique for evaluating the performance of networks as well as transport and application-level protocols. Traffic generation is one of the key challenges in modeling and simulating the Internet. For a small simulation with a single congested link, simulations are often run with a small number of competing traffic sources. However, for a larger simulation with a more realistic traffic mix, a basic problem is how to introduce different traffic sources into the simulation. For this reason, most of the international researchers move toward simulation environments like *ns* [3] or others. At the same time, simulating how wide area networks behave is complicated by the heterogeneity of these networks and their fast evolution. The interaction between the traffic from the diverse suite of protocols that operate over the Internet and the hierarchical nature of the topologies are a few of the factors contributing to the complexity of such large networks [14]. We refer to [5] for a detailed description of the many difficulties involved in simulating the Internet in a realistic manner.

Taking into account this kind of analysis, this work presents a contribution to one of the most critical aspects of network architecture analysis: *synthetic generation of realistic high traffic loads over real networks*. The motivations at the base of our choice are the

following. Significant progress has been made in the last few years in tools for realistic traffic generation, for both simulations and analysis. An approach of "*real simulation*" permits to overcome some typical constraints: (i) generally we simulate protocols but we ignore how they are actually implemented in terminals and nodes; (ii) we need to consider computational aspects for applications, nodes, and systems; (iii) macro-scale evaluations highly depend on phenomena dynamics; (iv) in a simulation environment like ns there is a synchronous coordination among the simulated events.

This paper is organized in six sections. After this introduction, Section 2 summarizes the main characteristics of D-ITG. Details on the model, the architectural choices and the functional modalities of D-ITG are presented in Section 3. Section 4 shows a complete D-ITG performance analysis and a comparison with several traffic generators over a multi platform scenario. A scalability analysis of the generation platform is depicted in Section 5. Section 6 ends the paper with some conclusion remarks.

2. Distributed Internet Traffic Generator (D-ITG)

A generator of controllable, scalable, synthetic but realistic IP traffic is needed in several circumstances. For instance, we cite here the analysis of new applications and network mechanisms over the Internet and the testing of Quality of Service (QoS) architectures. D-ITG [6] has been developed for this purpose. From our point of view, we define "realistic" the traffic that is "statistically similar" to the traffic generated over a real network from real protocols/applications.

Traffic generators usually produce low traffic loads in a controlled test-bed environment where a few sources are present. D-ITG aims to simulate complex networks by generating traffic flows on a packet-by-packet basis. Basically, a traffic flow is specified through two random processes: packet Inter Departure Time (IDT)—the time between the transmission of two consecutive packets—and Packet Size (PS)—the amount of data being transferred by the packets (Figure 1). Both processes are modeled as i.i.d. series of random variables (constant, uniform, exponential, pareto, normal, cauchy, etc). By using specific combinations of IDT and PS, per protocol traffic models can be created.

It is also possible to specify a seed to initialize both IDT and PS random variables. Such a feature allows for reproducing experiments, as exactly the same traffic pattern can be repeated by specifying the same seed value.

Besides to let the user choose the probability distribution and the related parameters, D-ITG can also set them automatically. Indeed, D-ITG incorporates some of the models proposed in the literature for various application protocols. This means that the user can simulate the generation of a supported protocol traffic without having to know the required

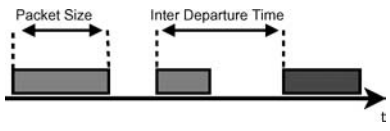

Figure 1. Inter departure time and packet size.

distributions and parameters. For the Telnet protocol, for instance, we refer to the studies of Paxson and Floyd [12, 13], which are based on the analysis of the Internet Traffic Archive (ITA) tracks. To reproduce VoIP traffic, instead, we make reference to an analysis carried out by Cisco Systems [4], which determines the required bandwidth depending on the encoding algorithm (G.711, G.729, G.723.1,. . .), the number of samples per packet, the usage of VAD (*Voice Activity Detection*) and RTP (*Real Time Protocol*) Header Compression.

D-ITG enables to evaluate a set of QoS performance metrics such as throughput, packet loss, delay (One Way Delay and Round Trip Time) and jitter. In our opinion, D-ITG is a key component for the experiments in a testing or planning phase for IP based networks. More precisely, in the context of QoS IP networks it is useful to have software architectures able to evaluate the performance of IP traffic control mechanisms supporting QoS. As for this last point D-ITG provides the setting of the TOS (Type of Service) and the TTL (Time to live) fields. Statistics related to the generated traffic flow can be collected by analyzing the information stored by both the sender and the receiver. An appropriate utility enables to determine the average values of throughput, delay, jitter and packet loss not only on the whole duration of the experiment, but also on windows of the desired duration.

D-ITG supports the generation of both IPv4 and IPv6 traffic. Also, all the communication channels established among the D-ITG components can be based on IPv4 as well as IPv6. IPv6 support has been added so as to preserve the correct operation on machines without IPv6 capabilities and to avoid increasing the complexity of the command line syntax (e.g. by adding new options).

To demonstrate the applicability, the performance and the usefulness of D-ITG, this paper details its behavior in an experimental test-bed. Also, we present a comparative analysis with the following traffic generators: Mtools [2], Rude/Crude [7], Mgen [8], Iperf [9] and UDP generator [10]. The comparative studies in this paper are conducted with CBR (Constant Bit Rate) UDP traffic, even though D-ITG is able to generate stochastic traffic patterns. In this paper we focus on the innovative solutions we introduced in the field of traffic generators and we analyze the related achieved performance. The motivations at the base of our work are presented in [16] where a complete description and analysis of related work is presented. In [15] considerations and details on different distributed D-ITG implementations (based on MPI) are presented and finally, the use of D-ITG for a comprehensive performance analysis of heterogeneous wireless networks is described in [17] and [18].

3. D-ITG software architecture

D-ITG platform defines a distributed multi-component architecture for high performance Internet traffic generation in heterogeneous environments. The main components of D-ITG are: (i) Internet Traffic Generator Sender (ITGSend), (ii) Internet Traffic Generator Receiver (ITGRecv), (iii) Internet Traffic Generator Log Server (ITGLog), (iv) Internet Traffic Generator API (ITGApi); (v) Internet Traffic Generator Decoder (ITGDec). Every component, in particular ITGSend and ITGRecv, presents an internal distributed implementation. Several kinds of distributed architectures have been implemented (i.e. an MPI version is available [15]). Figure 2 shows a graphical overview on the relationship among the five main bricks of D-ITG platform.

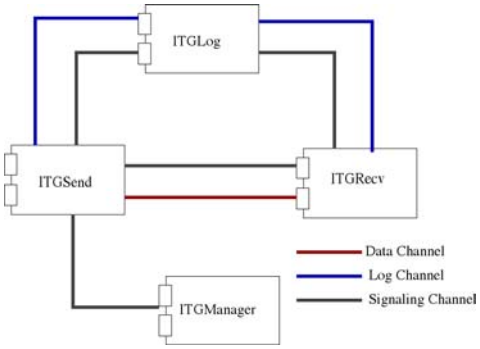

Figure 2. D-ITG software architecture.

The normal operation of our platform requires the exchange of messages among its constituent components. Figure 2 illustrates the different communication channels established for this purpose. As will be clarified later on, different signaling channels and a log channel may be needed, besides the data channel. The following subsections detail the components of the D-ITG platform and their interrelation.

3.1. Traffic Specification Protocol (TSP)

In order to set up an efficient architecture for traffic generation we defined a protocol for the configuration of experiments called Traffic Specification Protocol (TSP). TSP rules the exchange of messages between ITGSend and ITGRecv that are needed to control traffic generation. More precisely, TSP is a protocol we introduced in order to: (i) create a connection between a sender and a receiver; (ii) authenticate a receiver; (iii) exchange information on a generation process; (iv) close a sender-receiver connection; (v) detect generation events.

Figure 3 illustrates the TSP state diagram of both the receiver and the sender representing the possible transitions. More specifically, Figure 4 shows how ITGSend and ITGRecv

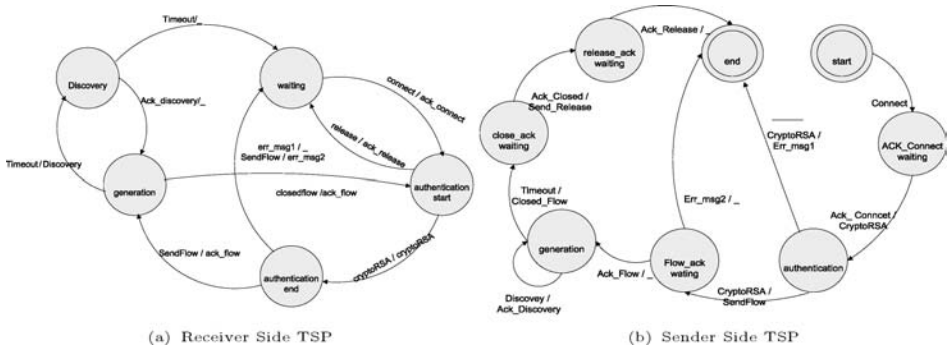

Figure 3. State diagrams of TSP at (a) Receiver Side and (b) Sender Side.

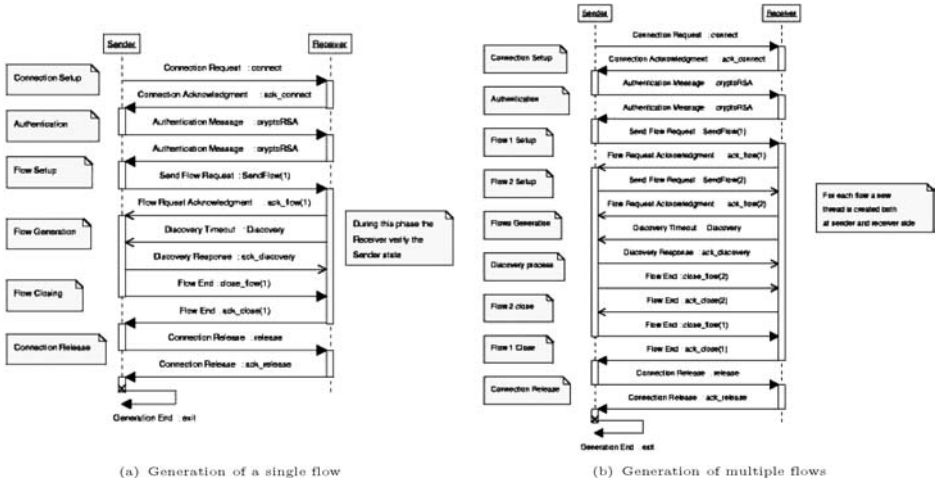

Figure 4. TSP implementation.

interact according to TSP protocol in case of the successful generation of a single flow (Figure 4(a)) and of multiple flows (Figure 4(b)). In the *single flow* case, the sender must first establish a TCP connection with the receiver. We denote such connection by "TSP (signaling) channel". Then, the receiver must be authenticated by means of a challenge-response protocol [1]. The use of this authentication method allows the sender to make sure that the receiving host really wants to receive traffic, avoiding that the generator might be used to launch attacks such as Denial of Service (Figure 5). In case of successful authentication, sender and receiver can exchange information on the generation experiment. Then, packets are sent on a different communication channel. During the generation process, the receiver

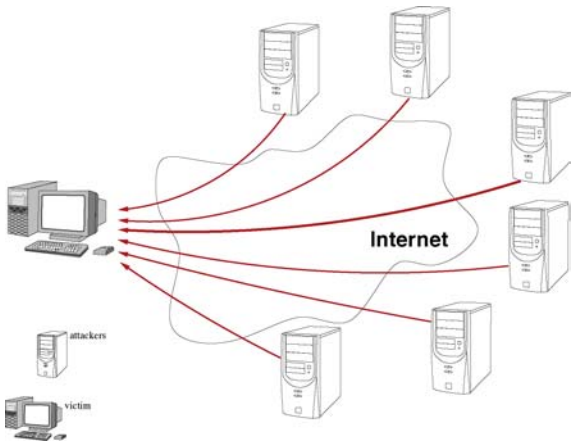

Figure 5. A wrong use: Denial of service attack by using D-ITG.

Table 1. TSP messages description

Type	Name	Description
1	Connection Request	The sender requests a connection
2	Connection acknowledgment	The receiver accepts the connection request
3	Flow generation request	The sender requests the permission to generate a flow
4	Flow ending	The sender informs the receiver about the end of a flow generation
5	Flow generation acknowledgment	The receiver grants the permission to generate a flow
6	Flow ending acknowledgment	The receiver acknowledges the end of a flow generation
7	Connection close acknowledgment	The receiver acknowledges the closing of a connection
8	Discovery request	The sender tests the receiver activity
9	Discorvery reply	The receiver replies to a sender discovery request
10	Crypto	An encrypted information is sent for authentication purposes
11	Connection close request	The sender requests the closing of a connection
12	Log configuration	The sender sends to the receiver infromation on the log server configuration
13	Log configuration acknowledgment	The receiver acknowledges the log configuration message
14	Error 1	Unable to accept the flow generation request because the specified port is unavailable
15	Error 2	Receiver authentication failed

probes the status of the sender every 60 seconds. TSP channel is also used by the sender to inform the receiver about the end of a flow generation. When the receiver acknowledges the flow closing, the sender releases the TSP connection and exits. In case of the generation of multiple flows the sequence of steps performed by the sender and the receiver is very similar to that of the *single flow* case. The main difference is that the *Flow Setup* and *Flow Closing* phases are repeated for every flow to be generated.

The TSP protocol requires the sender and the receiver to exchange different messages. Table 1 reports the purpose of each such messages. TSP messages are encoded in packets sent over the TSP channel. Figure 6 shows the format of a generic TSP packet. The type of message is specified by the *type* field. TSP packets contain the mandatory *type* field and certain other fields. Table 2 reports the meaning of some fields depending on the type of packet they are part of.

The TSP channel enables ITGRecv to be automatically informed by ITGSend (not by the user) about the transport protocol and the destination port of the flow to be generated (message #3). Hence, there is no need for ITGRecv to have an a priori knowledge of such information. In practice, this means that ITGRecv can operate as a daemon, which is always listening for new connections. Also, this architectural choice aims to improve the platform usability.

3.2. The sender

ITGSend is the sender of the D-ITG traffic generation platform. ITGSend can operate in three different modes:

Table 2. Description of TSP packet fields

	Type	3	12
Data port	Description	Port where the receiver will listen for traffic	Port where the log server listens for log information
	Type	3	12
Protocol	Description	Protocol type (UDP, TCP or ICMP)	Transport Protocol used to communicate with the log server
	Type	3, 4, 5, 6	
Flow id	Description	Identifier of the generated flow	
	Type	3	12
Dest IP	Description	Receiver IP address	Log server IP address
	Type	3	
Application Layer Protocol	Description	Simulated application layer protocol	
	Type	10	
Crypto	Description	Encrypted information needed for authentication	
	Type	10	
File name	Description	Log file name	
	Type	3	
UP Protocol	Description	Application level protocol to be simulated	

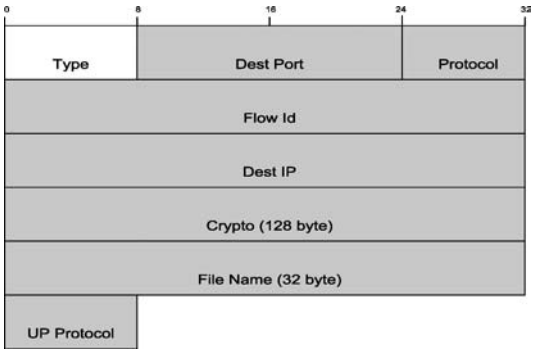

Figure 6. TSP packet format.

- *single flow mode*: In case a single flow must be generated, ITGSend itself manages the configuration of the experiment through the TSP protocol and transmits the packets of that flow (Figure 7(a));
- *multiple flows mode*: In case multiple simultaneous flows must be generated, ITGSend operates as a multi-threaded application. Figure 7(b) shows the case where all the flows share the same destination host;
- *daemon mode*: ITGSend can be launched and stay idle waiting for instructions. Thus, the generation process can be remotely controlled (Section 3.5). In this mode, each flow is generated by a separate thread as in the multiple flows mode.

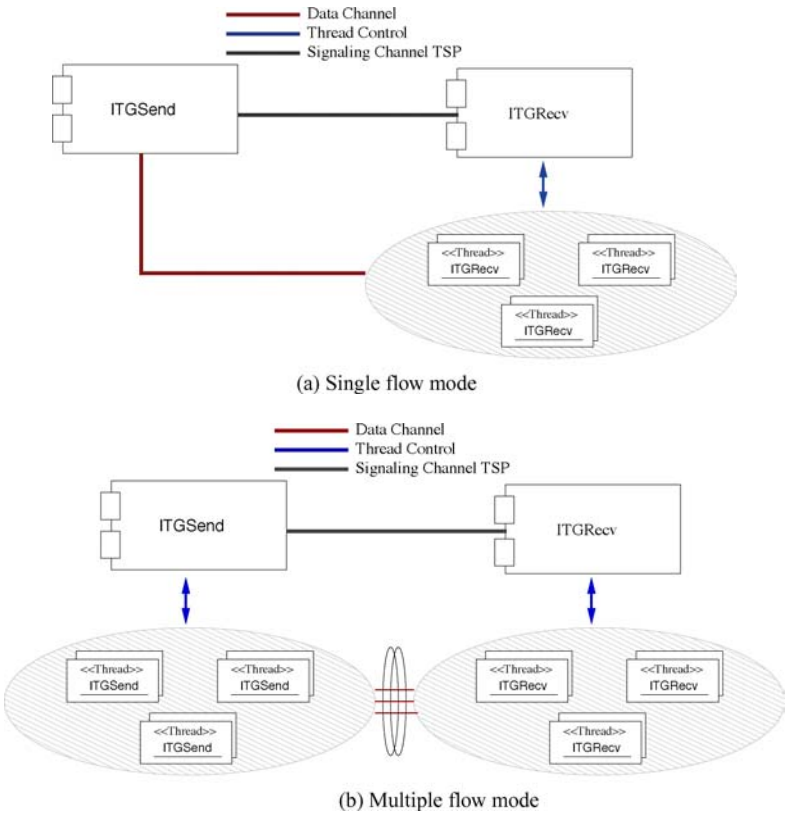

Figure 7. Traffic generation in (a) single flow mode and (b) multiple flow mode.

In single flow mode, the parameters of the flow (IDT and PS processes, destination IP address and port, duration, etc.) may be specified as command line arguments. In multiple flows mode, instead, a script file is required (with each line specifying a flow). In daemon mode, ITGSend listens for instructions on a fixed TCP port. We provide a C++ API (Application Programming Interface) to exchange messages with ITGSend launched in daemon mode.

Figure 8 shows how ITGSend operates when used in *multiple flow mode*. In this case ITGSend acts as a multi-threaded application. For each flow to generate, the main ITGSend process generates a new *flowParser* thread. Such thread first parses the input line that describes the flow to be generated. Next, it checks if there exists a *signalManager* thread connected to the destination via a TSP channel. If such a thread does not exist a new *signalManager* thread is created. This thread is responsible for the TSP protocol implementation and for the creation of the *flowSender* thread, which handles the traffic flow generation. There is only one *signalManager* for each distinct traffic destination. In this way ITGSend enables the sharing of a single TSP channel in case of multiple flows generation towards a single destination, so as to reduce the overhead due to the experiment control and setup activities. The synchronization among all the threads that cooperate in the generation of

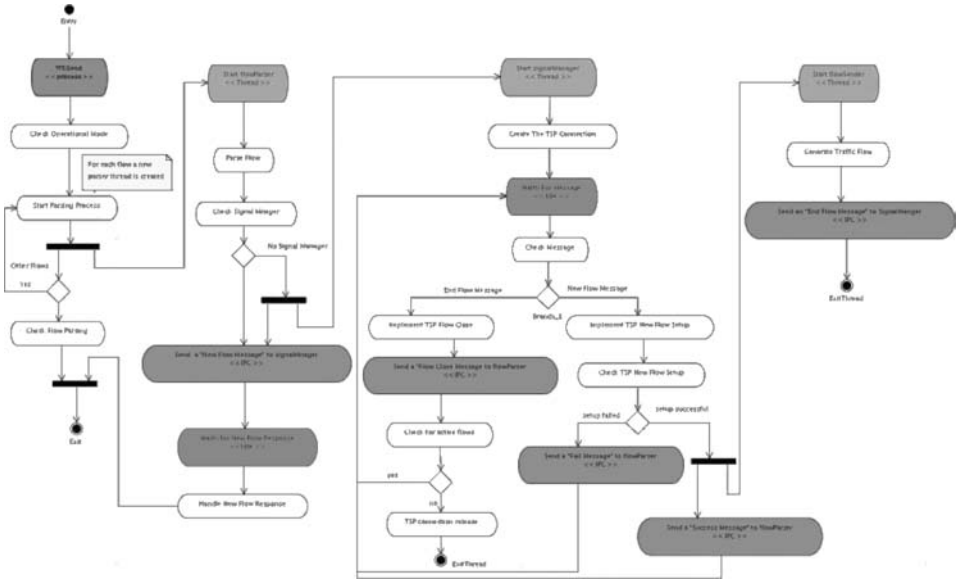

Figure 8. ITGSend activity diagram for a multi flows generation experiment.

each flow is made using the *inter process communication* (IPC) over Unix-like platforms, or the *event communication* over Windows platforms. For example, as Figure 8 shows, the creation of a new *flowSender* thread starts when *signalManager* receives a *NEW FLOW START* IPC message or an equivalent windows events. In the same way *signalManager* uses an IPC message, or an equivalent windows event, in order to communicate the result of the *flowSender* creation and closing.

As far as IPv6 generation, ITGSend implicitly infers the IP version from the address of the destination host. If such address is an IPv6 address, then IPv6 sockets are established for signaling and traffic generation. Of course, if the kernel does not support IPv6, the generation experiment will fail.

To collect statistics about the generation experiment, it is necessary for ITGSend to store some information in the payload of each packet it sends. Such information includes the identifier of the flow the packet belongs to, a sequence number and the time the packet was sent. Also, ITGSend may create a log file (either locally or on a remote log server), which can be processed at a later stage by ITGDec to provide information about the generated traffic.

The real traffic generation is heavily influenced by the CPU scheduling. Indeed, several processes (both user and kernel level) can be running on the same PC and this has a bad impact on the quality of the generated flow. Since the real-time support of the operating systems where ITGSend can be used is not very efficient (due to their scheduling mechanism and the inevitable timer granularity), it was necessary to use a strategy. A variable records the time elapsed since the last packet was sent; when the inter-departure time must be awaited, this variable is updated. If its value is less than inter-departure time the remaining time is awaited, otherwise the inter-departure time is subtracted from the value of this variable

and no time is awaited. This strategy guarantees the required bit rate, even in presence of a non real-time operating system. Also the choice of a multi-threaded implementation of ITGSend is tied to the need of limiting the interference among the generations of different simultaneous flows.

3.3. The receiver

ITGRecv is the receiver component of the D-ITG traffic generation platform. It always operates as a concurrent daemon (Figure 10), listening for new TSP connections on a specified TCP port (9000).

Figure 9 shows how ITGRecv operates to receive a new traffic flow. When a TSP connection request arrives, ITGRecv generates a new *signalManager* thread that is responsible for the TSP protocol implementation. Each single flow is received by a separate thread: for each TSP New Flow message it receives, signalManager creates a new *flowReceiver* thread. Such thread: (i) creates the receiving socket; (ii) receives the packets sent by ITGSend; (iii) generates a log file that describes the received flow at packet level. Like ITGSend, the coordination among all the threads involved in the receiving process is made using the *inter process communication* (IPC) over Unix-like platforms, or the *event communication* over Windows platforms. In Figure 9 the main IPC messages (windows events) used in the new flow setup process are shown.

As far as IPv6 generation, we have to distinguish between Linux and Windows OSs. Under Linux Operating System, ITGRecv first tries to establish an IPv6 socket for signaling. If the kernel is IPv6-enabled, such operation is successful and ITGRecv will be able to reply to both IPv4 and IPv6 connection requests. If not, an IPv4 socket is established and ITGRecv will only reply to IPv4 connection requests. Thus, there is no need to specify the IP version, since it will be auto-detected. Also, a single ITGRecv instance is able to receive both IPv4 and IPv6 traffic. The type of socket established to actually receive the traffic is determined by the type of destination address sent by the sender via the TSP protocol. Under Windows

Figure 9. ITGRecv activity diagram.

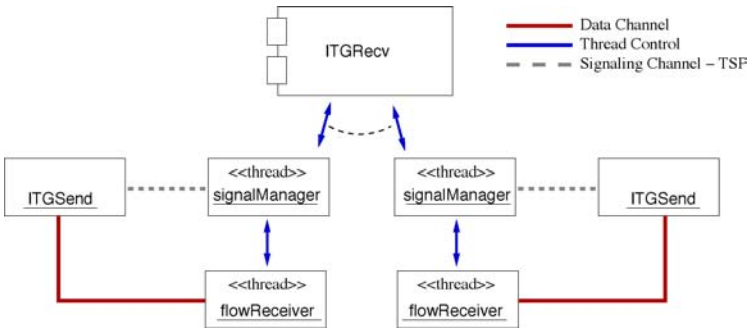

Figure 10. ITGRecv daemon model.

Operating System, instead, the same procedure cannot be used. Indeed, if an IPv6 socket is established, it will not reply to IPv4 connection requests. Therefore, it is necessary to specify whether ITGRecv has to receive IPv6 traffic. In case ITGRecv has to receive both IPv4 and IPv6 traffic, it is necessary to run two different instances of ITGRecv.

The log file generated by ITGRecv can be stored either locally or on a remote log server. Then, it can be processed at a later stage by ITGDec to provide information about the received traffic.

3.4. The log server

ITGSend and ITGRecv can either log information about the generated flows in a local log file or send such information to a remote log server. ITGLog is the log server of the D-ITG platform, which receives and stores the information received from multiple senders and receivers. A sender (receiver) intending to delegate the logging activity to ITGLog has first to setup a signaling channel toward the log server. The communication is ruled by a signaling protocol that allows each sender (receiver) to register on, and to leave, the log server. The information to be logged is then sent using a log channel, which may be either a reliable channel (TCP) or an unreliable channel (UDP).

ITGLog proves to be useful in different scenarios. We propose a couple of examples:

- Real-time controller (Figure 11(a)): a log server may be useful in those cases where the sender is requested to adapt its transmission rate to the capacity of the channel toward the receiver. For instance, we refer to the delivery of real-time multimedia contents. The receiver may send information to a log server. Such information are not stored, but passed to a real-time controller which analyzes it “on-the-fly” to adjust the sender’s rate.
- Devices with limited storage resource (Figure 11(b)): if a device with limited storage resource, such as a PDA (Personal Digital Assistant), is used to send or receive a traffic flow, a log server can be used to collect the log information that can not be stored on such a device.

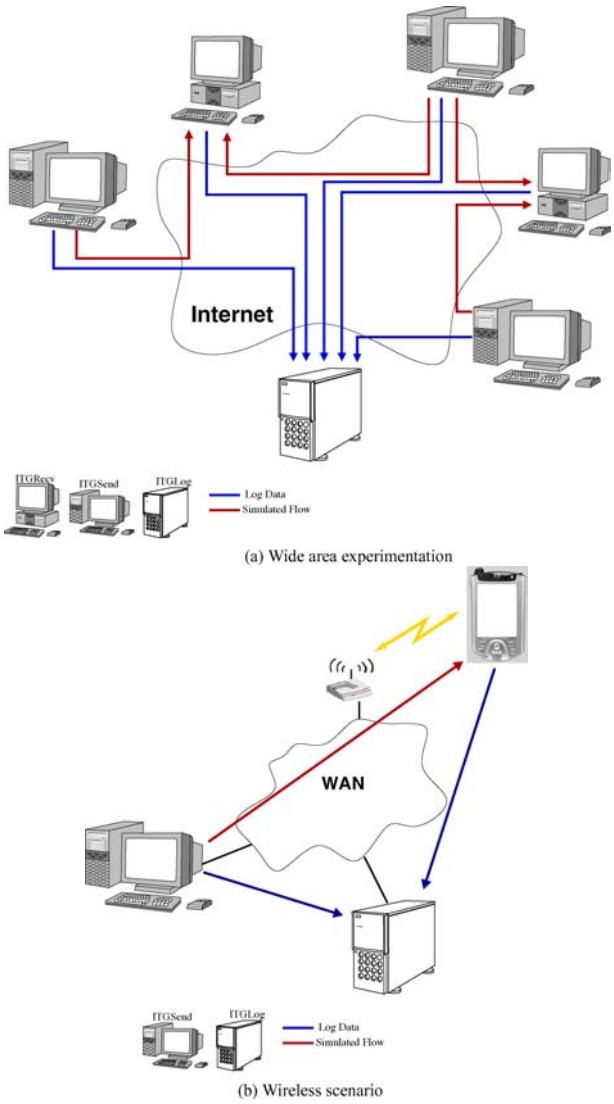

Figure 11. Using ITGLog in (a) wide area experimentations and in (b) wireless scenario.

3.5. The remote controlling

As described before, ITGSend can be launched in daemon mode and stay idle waiting for instructions. We provide ITGApi, a C++ API, to remotely control ITGSend. Such API includes a function that enables to send a message to ITGSend. Through this message, it is possible to issue the generation of a traffic flow. The syntax of this message is the same as that used to request a flow generation from the command line. ITGApi also provides a

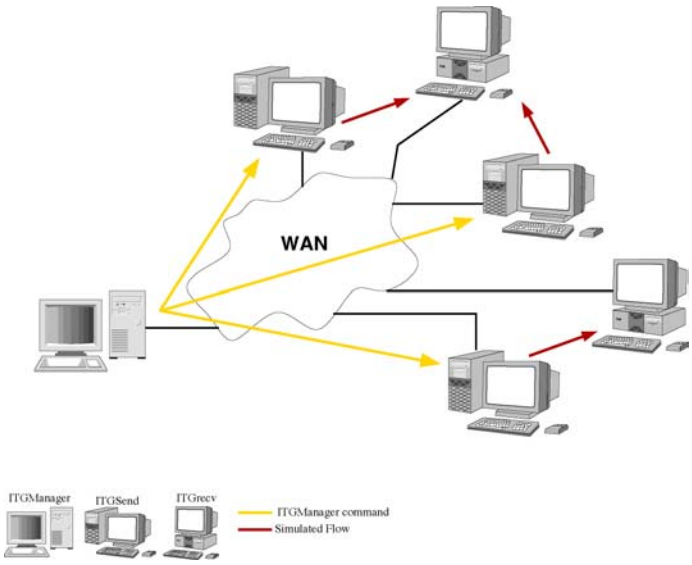

Figure 12. Controlling the whole traffic over the network.

non-blocking function that catches messages sent by ITGSend. Indeed, ITGSend replies with a message to indicate both the start and the end of the flow generation. It is possible to remotely control more than one sender, as depicted in Figure 12. In this way, an ad-hoc component, called ITGManager in the figure, can control the whole traffic crossing the network using ITGApi.

This feature can be used, for instance, to test centralized routing algorithms in a real environment. Indeed, we can assume the presence of a "network controller" which receives flow requests and determines the path that the corresponding flow must follow in order to satisfy flow requirements and optimize network resource usage. We are assuming that the network architecture allows to explicitly route flows (e.g. MPLS). After the path has been established, it is possible to issue the generation of the traffic flow. By collecting statistics about average delay, jitter and packet loss related to several flows, it is possible to compare the performance of different traffic engineering algorithms.

3.6. The decoder

Statistics related to the generated traffic flows can be collected by analyzing the information that both ITGSend and ITGRecv store in their own log file. For each packet, the following information are stored: (1) flow identifier; (2) sequence number; (3) source address and port; (4) destination address and port; (5) transmission time; (6) receiving time; (7) packet size. It is worth mentioning that log files are binary files, in order to reduce their size and speed up I/O operations with respect to text files.

ITGDec is the utility of the D-ITG platform that processes log files and displays the average values of delay, jitter, throughput and packet loss for each flow. ITGDec can also

determine the average values of such measures over time intervals of the desired duration. The measured delay value needs further clarification. We distinguish between the one-way-delay mode and the round-trip-time mode. In the former case, ITGSend logs when the packets are sent, and thus can not know the receiving time of packets. So it fills the corresponding field inside the log file with the transmission time. Hence, the resulting delay (and, consequently, jitter) is null for each packet. The delay value resulting from the processing of the log file of ITGRecv instead represents the one-way-delay. In the latter case, the analysis of the log file stored by ITGSend returns the round-trip-time, while the analysis of the log file stored by ITGRecv returns the one-way-delay.

4. Performance evaluation and comparative analysis

To study the maximum data rate D-ITG can achieve, we performed an extensive experimental analysis. In order to trace a first reference study and—at the same time—a precise data rate evaluation we need simple and controllable test-beds. Hence, we used two back-to-back connected PCs, whose characteristics are reported in Table 3.

We have also studied, using the same workstations, the D-ITG performance over a local machine in order to (i) study the interference between sender and receiver processes (in a real environment multiple senders and receivers can be running over the same machine) and (ii) isolate the dependencies from network dynamics. We found that D-ITG is able to reach high data rates on both transmission and reception sides. In the next subsections we first detail the experimental results obtained in a scenario composed by two workstations with Linux OS (the scenario in which D-ITG reaches the best performance). Next, we summarize the results obtained in:

- distributed scenarios
  - two workstations with Windows OS
  - two workstations, one with Linux OS and the other with Windows OS

Table 3. Experimental parameters

Hardware details	Intel Pentium 4 2.6 Ghz—CPU cahce 512 RAM: 1024 MB Hard Disk: Maxtor 6Y080L0 (Fast ATA/Enhanced IDE Compatible Ultra ATA/133 Data Transfer Speed 2MB Cache Buffer Quiet Drive Technology 100% FDB (fluid dynamic bearing) motors)
Network Details	2 PCs with Gigabit Ethernet back-toback connection Ethernet Controller: 3Com Gigabit LOM (3c940) 3Com Corporation 3c905C-TX/TX-M
Software Details	**Linux**: Linux Mandrake 9.1 with kernel 2.4.21-013mdk and Linux RedHat 9 with kernel 2.4.22 **Windows**: Windows XP Professional 2002, Service Pack 1
Experiment Duration	$T = 60$ s
Traffic Details	CBR, Constant Bit Rate Protocol: UDP C = packet per second (pps or pkt/s) c = packet size (byte)

- local scenarios
  - one workstation with Linux OS
  - one workstation with Windows OS

Finally, we summarize an analysis which compares D-ITG to other widely used traffic generators. We refer for more detailed results to [20].

4.1. Distributed experimentation over linux platform

We studied the performance of D-ITG at both sender and receiver side. This analysis requires the traffic generator to store information at both sides. The logging process at sender side may influences the maximum achieved transmission data rate. Clearly, the generated data rate influences the data rate measured at receiver side. Hence, we present both the results in the case of storing at sender and receiver side and the results in the case of storing only at receiver side. We denote by C the packet rate (packet/s), c the packet size (byte) and t the flow duration (s).

4.1.1. Log information stored at both sender and receiver side. Figure 13 depicts the performance achieved by D-ITG in terms of generated and received data rate. In case of $C = 75000$ pkt/s, $c = 1024$ byte, $t = 60$ s D-ITG reaches a generated data rate equal to 612 Mbit/s resulting in a loss rate equal to 0.5% (the loss rate being defined as: $(1 - \frac{MeasuredDataRate}{ExpectedDataRate})$). All the generated packets have been received by ITGRecv. Increasing the requested data rate results in a greater error. Thus, we have found an upper bound to the maximum achievable sending data rate that is less than the theoretical one (equal to 1000 Mbit/s with the used network interfaces). This scenario allows us to identify the maximum sending rate supported by ITGSend in case it logs. Moreover, it is possible to assert that ITGRecv is capable to receive all the packets that ITGSend generates, and then it is not possible to identify the maximum receiving data rate that characterizes ITGRecv. Finally, it is possible to presume that ITGSend can reach an higher data rate when the logging process does not interfere with the generation process.

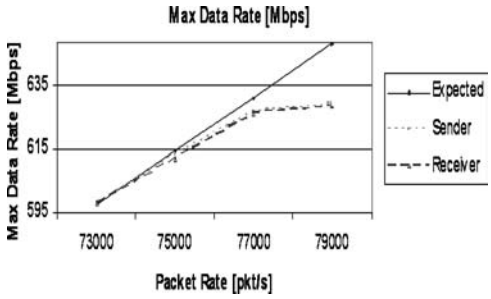

Figure 13. Generated and received data rate over Linux devices (distributed and log at both sender and receiver side).

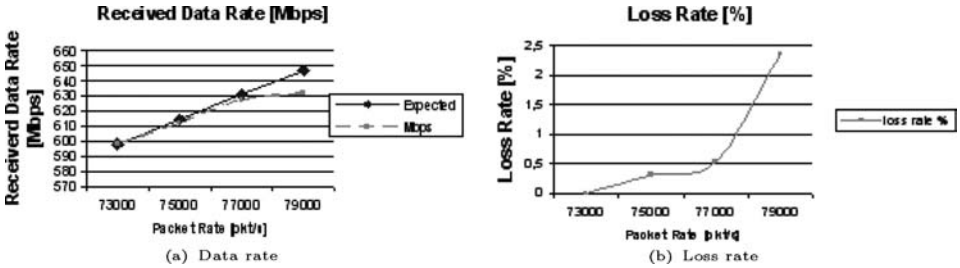

Figure 14. Distributed analysis at receiver side over Linux platform: (a) data rate (b) loss rate.

4.1.2. Log information stored at receiver side. Figure 14(a) shows the performance in terms of the received data rate achieved by D-ITG when the receiver only logs. In Figure 14(b) the loss rate as a function of packet rate is reported. We have found an upper bound to the maximum achievable received data rate that, also in this case, is less than the theoretical one. In case of $C = 77000$ pkt/s, $c = 1024$ byte, $t = 60$ s D-ITG reaches a received data rate equal to 627 M bit/s with a loss rate equal to 0.5%. This upper bound may be either the maximum achievable data rate at receiver side or the maximum sustainable data rate at sender side. As shown in the previous subsection, logging both at sender and receivers sides, the maximum achievable data rate at sender side is equal to 612 M bit/s that is less than the 627 M bit/s measured in this scenarios. So we can say that, using the workstations described in Table 3 with Linux OS, ITGSend can generate at least 627 M bit/s. As far as the maximum achievable received data rate considering this scenario we can only say that it is greater than or equal to 627 Mbit/s. Considering two senders that send packets to a single receiver as a scenario that may be used to evaluate the maximum achievable data rate at receiver side is a pitfall: in this case ITGRecv generates two threads that must share the available resources. As we will show in Section 5.2 this interference is very little but in any case reduces the overall performance at receiver side.

4.2. Summary of other scenarios

Table 4 shows the result of the analysis in all the considered distributed scenarios. For each row the first column describes the tested scenario. The second column contains the data rate measured at sender side when ITGSend logs or the data rate measured at receiver side when ITGSend does not log. The third column contains the measured data rate at receiver side. The symbol $\geq$ indicates that the measured data rate can not be assumed as an exact limit of the data rate supported by D-ITG but only as a lower bound on it.

As mentioned above, we have observed the best performance using Linux on both sender and receiver sides. Under Windows OS D-ITG behave worse than under Linux OS. As the last two rows of Table 4 show, using Linux as one end point of the generation process and Windows as the other end point, we have been able to determine the maximum data rate achieved by D-ITG under Windows OS. In particular, ITGSend reaches a maximum sending data rate equal to 242 Mbit/s, when ITGRecv reaches a maximum receiving data rate equal to 483 Mbit/s. This result shows that the bottleneck in the generation process

Table 4. D-ITG performance in distributed scenarios

Scenarios	Sender side data rate	Receiver side data rate
Two workstations with Linux OS—Both sender and receiver log	612 Mbit/s	≥612 Mbit/s
Two workstations with Linux OS—The receiver only logs	≥627 Mbit/s	≥627 Mbit/s
Two workstations with Windows OS—Both sender and receiver log	161 Mbit/s	≥161 Mbit/s
Two workstations with Windows OS—The receiver only logs	≥242 Mbit/s	≥242 Mbit/s
One workstation with Linux OS and the other with Windows OS—The receiver only logs	≥627 Mbit/s (Linux)	483 Mbit/s (Windows)
One workstation with Linux OS and the other with Windows OS—The receiver only logs	242 Mbit/s (Windows)	≥627 Mbit/s (Linux)

Table 5. D-ITG performance in local scenarios

Scenarios	Sender side data rate	Receiver side data rate
Linux OS—Both sender and receiver log	511 Mbit/s	≥511 Mbit/s
Linux OS—The receiver only logs	≥611 Mbit/s	≥611 Mbit/s
Window OS—Both sender and receiver log	102 Mbit/s	≥102 Mbit/s
Windows OS—The receiver only logs	≥241 Mbit/s	≥241 Mbit/s

under Windows OS is ITGSend. Since this is an expected result, we can reasonably assume that such a result may be also valid under Linux OS. Therefore, we can argue that 627 Mbit/s is the value of the maximum data rate that ITGSend can achieve under Linux OS.

Table 5 shows the result of the analysis in all the considered local scenarios (both sender and receiver on the same machine). Also in this case we have found that D-ITG reaches better performance under Linux OS. In the case the receiver only logs, the measured data rates are not far from those achieved in the distributed scenarios: a rate decrease of 26 Mbit/s for Linux and 1 Mbit for Windows. In case both sender and receiver log the rate decrease is greater: 101 Mbit/s for Linux and 59 Mbit/s for Linux. Such results confirm that sender and receiver running on the same machine interfere with each other.

4.3. Comparative analysis

In the previous two subsections we have presented the performance achieved by D-ITG in different and heterogeneous (in terms of OSs) scenarios. In all the working conditions D-ITG has reached a high data rate on both transmission and reception sides. However, these data rates are always inferior to the theoretical maximum. In order to assess the performance achieved by D-ITG we have developed a comparative analysis among D-ITG and other widely used network traffic generators (see [16] for an exhaustive description of the available Internet network traffic generators). The scenarios we have considered are the same used for the performance analysis of D-ITG. In all the tested conditions we have

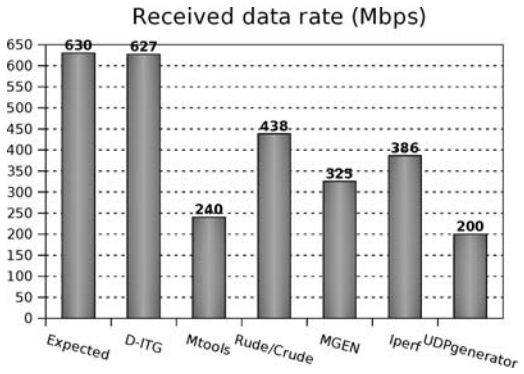

Figure 15. Distributed received data rate over Linux platform (log at receiver side)).

found that D-ITG reaches the maximum data rate on both sides of the generation process. Figure 15 shows the measured data rate at receiver side in case of two Linux workstations when ITGRecv only logs. In all the remaining scenarios we have found similar results.

5. Scalability analysis

We think that the built-in "scalability propriety" is of paramount importance in a networking tool. Thus, we present a D-ITG scalability analysis at both sender and receiver side. More precisely, we measured the sender and receiver performance while the number of the threads/flows was varying. We did not take into account ITGLog scalability analysis because the Log server undergoes a very low workload (with respect to ITGSend and ITGRecv components). The scalability analysis has been carried out on both Windows and Linux platforms. In order to study the performance of the sender and the receiver we stored information at both sender and receiver side.

5.1. Scalability analysis at sender side

Linux platform. In order to work with the maximum data rate we used the parameters that permit to reach the D-ITG best performance ($C = 75000$ pkt/s, $c = 1024$ byte, and therefore Maximum Data Rate equal to 612 Mbit/s with an error rate equal to 0.5%). In the case of n flows/threads, in order to maintain the same maximum value of data rate we configured each flow with a packet rate equal to $\frac{75000}{n}$ pkt/s. Figure 16(a) shows that the data rate is almost constant and equal to the maximum data rate when the number of flows is varying. We can conclude that the D-ITG Linux sender presents an optimal behavior in terms of scalability.

Windows platform. In this case we set up our experimentation in the same way of the Linux platform. In order to work with the maximum data rate we used the parameters that permit to reach the D-ITG best performance over Windows platform ($C = 20000$ pkt/s, $c = 1024$ byte, and therefore Maximum Data Rate equal to 163 Mbit/s with an error rate

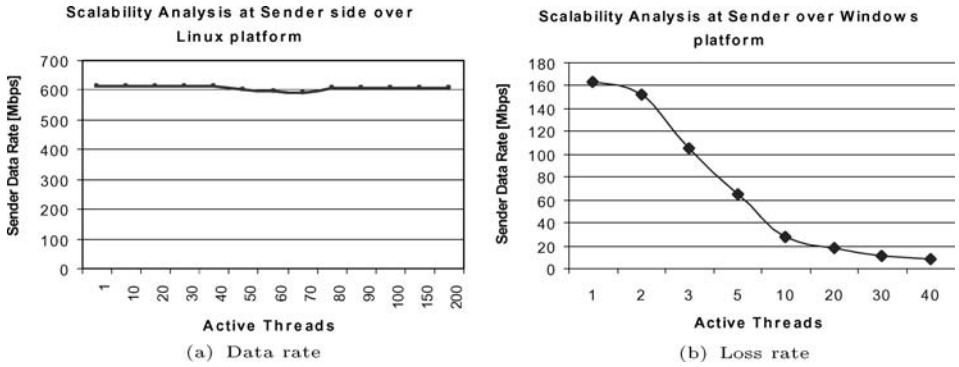

Figure 16. Scalability analysis at sender side: (a) Linux platform and (b) Windows platform.

equal to 1.5%). In the case of n flows/threads in order to maintain the same maximum value of data rate we configured each flow with a packet rate equal to $\frac{20000}{n}$ pkt/s. Figure 16(b) shows that over Windows platform there is a considerable degradation of the performance with respect to D-ITG implementation over Linux platform. Comparing Figure 16(a) to Figure 16(b) it is simple to understand that the D-ITG Windows Sender is less scalable than D-ITG Linux Sender.

5.2. Scalability analysis at receiver side

Linux platform. In this case we used the same number of threads used in the D-ITG Linux Sender analysis. Figure 17(a) shows that up to five flows the receiver capabilities are the same as that achieved with one flow. In the worst case, when the number of flows (n) is equal to 200 we measured a data rate equal to 278 Mbit/s.

Windows platform. In this case, in order to analyze the scalability of the receiver we used the Linux ITGSend. This choice was motivated by the following reasons: (i) D-ITG Linux Sender presents higher performance than D-ITG Windows Sender; (ii) the D-ITG Windows

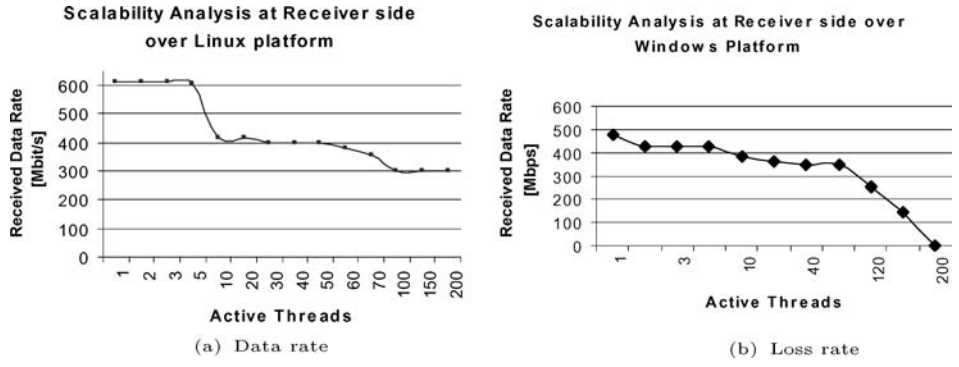

Figure 17. Scalability analysis at receiver side: (a) Linux platform and (b) Windows platform.

Table 6. Execution time of code test

Component	Time
ITGSend (Linux)	$SW\ Ex = 0.13$ s
ITGRecv (Linux)	$SL\ Ex = 0.13$ s
ITGSend (Windows)	$RW\ Ex = 0.41$ s
ITGRecv (Windows)	$RL\ Ex = 0.23$ s

Sender is less scalable than D-ITG Linux Sender. In the case of a single flow/thread we measured a received data rate equal to 480 Mbit/s. In the case of two flows/threads we observed a reduction equal to 10% (Figure 17(b)). Observing Figure 17(a) and (b) we can conclude that the D-ITG Windows Receiver is less scalable than D-ITG Linux Receiver, but—in the case of Windows platform—the receiver shows more acceptable scalability proprieties with respect to the sender.

5.3. Discussion on achieved performance over different platforms

In order to determine the time when the packet is sent or received (to be included in the log file) and the time interval elapsed since the transmission of the last packet sent, the D-ITG Linux implementation uses the *gettimeofday()* function, whereas the Windows implementation uses *getSystemTime()* and *QueryPerformanceCounter()* functions. In this subsection, we want to investigate the efficiency of these different functions when they are used in the D-ITG platform. Indeed, the different behavior (and the consequent achieved performance) of these crucial functions may be one of the factors causing the different performance between the Linux and the Windows implementation. To study the influence of these functions, we carried out the following test. We ran the pieces of code invoking the above mentioned functions 100.000 times and obtained the results shown in Table 6. As we can see, the execution time under Windows is longer than that under Linux. If we try to link these results with the performance achieved under Windows and Linux OSs (see Table 4) we find that: (i) at sender side we have $\frac{SWEx}{SLEx} \approx 3.2$ and $\frac{MaximumSustainableRateLinux}{MaximumSustainableRateWindows} \approx 3.8$; (ii) at receiver side we have $\frac{RWEx}{RLEx} \approx 1.7$ and $\frac{MaximumSustainableRateLinux}{MaximumSustainableRateWindows} \approx 1.3$. These ratios are similar but not equal. So, we are far from stating that the different performance achieved under the two platforms is strictly linked to the different execution time of the functions used to retrieve the current time. Other causes must be considered such for example the system overhead on performing threading, or the different message buffering for communication. However, by taking into account that the time functions are called for each sent/received packet (e.g. 77000 times per second in the example shown) the overhead of these functions is one of the factors that influence the achieved performance. Further deep investigation is needed to full understand the different achieved performance.

6. Conclusions

In this paper we presented a traffic generation platform that we called D-ITG. In order to present the high performance of D-ITG, an experimental analysis has been conducted over

Linux and Windows platform. D-ITG showed the best performance over both platforms and it was able to generate at high transfer rate with high values of packet size. More precisely, we conducted several experiments to compare the data rate achieved by D-ITG to that achieved by other traffic generators in different scenario. The performance obtained in the case of storing at receiver side only and storing at both sender and receiver side has been evaluated both when sender and receiver are on the same machine and when they are on different machines. The results obtained shows that D-ITG achieves the data rate value closest to that expected, both under Windows and Linux. Also, we presented a scalability analysis (in terms of achieved throughput and number of concurrent flows) of our generation platform, which showed good results, especially for the Linux implementation of the sender. D-ITG presents a (internal and external) distributed architecture and thanks to this architecture it is able to reach high performance. Here we use the term "performance" in a wide sense, to refer to a collection of indicators: (i) generated bit rate; (ii) received bit rate; (iii) scalability; (iv) usability; (v) supported stochastic traffic patterns; (vi) supported protocols; (vii) novel entities (Log Server and Manager); (viii) supported platforms (a multi platform tool running on Linux, Windows, and Linux Familiar). D-ITG is currently downloadable and freely available at `www.grid.unina.it/software/ITG` and, to the best of our knowledge, in terms of software architecture, operative mode and achieved results, no other similar platforms are available.

Acknowledgments

This work has been carried out partially under the financial support of the "Ministero dell'Istruzione, dell'Universitá e della Ricerca (MIUR)" in the framework of the FIRB Project "Middleware for advanced services over large-scale, wired-wireless distributed systems (WEB-MINDS)", and of the European Union under the E-Next Project FP6-506869. We would like to thank Alessio Botta and Salvatore Guadagno for their valuable support and anonymous reviewers for their suggestions.

References

1. C. Adams. The Simple Public-Key GSS-API Mechanism (SPKM), RFC 2025, October 1996.
2. S. Avallone, M. D'Arienzo, M. Esposito, A. Pescapé, S. P. Romano, and G. Ventre. Mtools. *IEEE Network, Software Tools for Networking 2002*, 16(5):3. ISSN 089080445.
3. L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu. Advances in network simulation. *IEEE Computer*, 33(5):59–67, 2000.
4. Cisco Systems. "Traffic Analysis for Voice over IP," white paper, http://www.cisco.com/en/US/tech/tk652/tk701/technologies_white_paper09186a00800d6b74.shtml.
5. S. Floyd and V. Paxson, "Difficulties in simulating the Internet." *IEEE/ACM Trans. on Networking*, 9(4):392–403, February 2001.
6. http://www.grid.unina.it/software/ITG
7. http://www.atm.tut.fi/rude
8. http://mgen.pf.itd.nrl.navy.mil
9. http://dast.nlanr.net/Projects/Iperf/
10. http://www.citi.umich.edu/projects/qbone/generator.html

11. S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.
12. V. Paxson. Empirically-derived analytic models of wide-area TCP connections. *IEEE/ACM Trans. on Networking*, 2(4):316–336, 1994.
13. V. Paxson and S. Floyd. Wide-area traffic: The failure of poisson modeling. *IEEE/ACM Trans. on Networking*, 3(3):226–244, 1995.
14. V. Paxson and S. Floyd. Why we don't know how to simulate the Internet, In *Proceedings of the 1997 Winter Simulation Conference*, SCS, December 1997.
15. A. Pescapé, S. Avallone, D. Emma, and G. Ventre. Performance evaluation of an open distributed platform for realistic traffic generation, *Performance Evaluation: An International Journal* (Elsevier Journal), 60:359–392, 2005.
16. S. Avallone, A. Pescapé, and G. Ventre. Analysis and experimentation of Internet Traffic Generator, In *Proceedings of New2an'04, Next Generation Teletraffic and Wired/Wireless Advanced Networking*, pp. 70–75—ISBN 952-15-1132-X.
17. S. Avallone, A. Pescapé, and G. Ventre. Distributed Internet traffic generator (D-ITG): Analysis and experimentation over heterogeneous networks. Poster at ICNP 2003.
18. G. Iannello, A. Pescapé, G. Ventre, and L. Vollero. Experimental analysis of heterogeneous wireless networks. In *Proceedings of WWIC 2004, Wired/Wireless Internet Communications 2004*. LNCS Vol. 2957, 2004, pp. 153–164, ISBN: 3-540-20954-9.
19. C. Petzold. *Programming Windows*. 5th edition, Microsoft Press, published 11/11/1998, ISBN 1-57231-995-X.
20. S. Avallone, D. Emma, A. Pescapé and G. Ventre. "A Distributed Multiplatform Architecture for Traffic Generation." In *Proceedings of International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, San Jose, California (USA), July 2004.