Precision and Accuracy of Network Traffic Generators for Packet-by-Packet Traffic Analysis

Marcos Paredes-Farrera, Martin Fleury and Mohammed Ghanbari
Department of Electronic
Systems Engineering
University of Essex
United Kingdom
Email: {mpared,fleum,ghan}@essex.ac.uk

***Abstract*— Network traffic generators allow video and audio streams to be modelled. 'Bursty' traffic patterns imply packet-by-packet rather than time-averaged analysis to determine the impact on network routers. Due to computer operating system time event scheduling dependencies, the ability to generate accurate packet delivery is compromised. A real-time operating system on an x86 commodity machine is shown to result in greater accuracy and range, but with some loss in precision outside its operating range, compared to a general-purpose Linux kernel.**

I. Introduction

The use of test traffic patterns is widespread in network performance testing and network simulation. Traffic patterns are employed to examine networks and applications in controlled environments. For example, video VBR (Variable Bit Rate) traffic and aggregate CBR (Constant Bit Rate) traffic were used to test an adaptive feedback rate controller in one node of a simulated ATM network [1]; in a different scenario, traffic patterns from backbone trace files were used to test Integrated Services (IntServ) in a simulated TCP/IP network [2]. This paper is concerned with understanding how well a software network traffic generator (NTG) is able to characterize traffic patterns, and determines the NTG's precision and accuracy when working on a network textbed using commodity machines, Linux-enabled PCs. It is possible by polling to get high accuracy in the unlikely situation that no other process were active. Otherwise, only with a real-time (RT) operating system (OS) is scheduling granularity sufficient to allow the NTG to deliver packets with sufficient accuracy for Packet-by-Packet (PbP) analysis.

Network performance tools such as *netperf* [3] contain *tcp_stream_script* and *upd_stream_script* scripts as NTG. Network simulators such as *ns-2* [4] contain modules like *PackMime* and *NSWEB*, which behave as an NTG, and *NCTUns* [5] has the *stgr* NTG. An NTG is a software tool used to input specific traffic patterns into live, simulated, or testbed networks. The NTG plays a critical part by recreating real (through a trace) and theoretical (through a probability density function)traffic patterns. Many NTGs create traffic patterns by means of time-averaged metrics, that is, by controlling bandwidth and packet rate during a fixed period of time. Because of time averaging, they cannot replicate traffic events, such as those associated with 'bursty' traffic behavior that may take place at a millisecond or microsecond scale. However, software traffic generators that can reproduce trace files on commodity systems are a comparative rarity, *TCPivo* [6] and *tcpreplay* [7] being exceptions.

The events of relevance to this paper are those arising from video and audio communication, including voice over IP (VoIP). For these applications, generating traffic on a PbP basis reveals how packet loss, delay, and jitter affect quality of service at the receiver. For example, in [8] the author considers that per-packet bandwidth reporting is the most appropriate for adaptive streaming applications, because of its responsiveness to changes in available bandwidth. Video delivery should avoid regimes that result in significant packet losses at router queues. However, it is the case that streaming applications often do send a bursty stream, either for reasons of coding efficiency or when a non-real-time operating system falls behind its schedule and releases a packet burst. Therefore, one might want to identify the particular burst characteristics that certainly lead to packet loss. The same if not more stringent requirements [9] may apply to multi-player on-line games.

To create a PbP traffic on a network testbed requires a high degree of precision and accuracy, during the delivery of packets to the network at high speed. As a consequence, the NTG becomes a real-time (RT) application, which is dependent on the operating system's (OS) time event scheduling accuracy. This is because the NTG is usually descheduled between packet deliveries. Descheduling takes place because the NTG makes an OS system call to some form of 'sleep'. The alternative would be to enter a clock polling loop but, as others have observed [6], such a loop can take up as much as 80% of CPU time.

Re-scheduling accuracy is dependent on at least three factors:

1) the priority of the NTG process in the scheduling queue(s).
2) the latency of the OS kernel.
3) the ability to schedule a process to a given granularity.

Factor one is ameliorated by ensuring that either there are no or few other competing processes or that the NTG runs at high priority or as a privileged RT process. In this study, the NTG is run as a normal application with no special priority, enabling

direct comparison between OSs but not gaining from factor two. A Linux kernel was patched with the Kansas University RT (KURT) kernel [10]. In KURT, only RT processes or tasks benefit from a specialized kernel patch and, hence, it is surmised that the principle advantage gained by the NTG from the RT OS is factor three, scheduling granularity. Incidentally, this contradicts a finding [11] that, under KURT-like OS, "[normal] Linux processes ... cannot support time-sensitive applications".

Scheduling granularity is determined by the resolution of the system clock. A limited clock resolution can restrict the validity of network measurement results, as was observed by Paxson [12] for the packet-pair technique, which was limited by the ten millisecond clock granularity in some types of Unix and Windows-like OS. To evaluate the performance of an NTG for PbP analysis, one ideally needs to know the traffic pattern's precision and accuracy in the microsecond range.

Precision and accuracy have been widely treated as synonyms. However, they have different meanings. Precision is related to the quality and stability of a system that makes it possible to obtain the same or very similar values or measurements. On the other hand, accuracy is how close the created value is to the true value. A system may be precise but not accurate; or accurate but not precise.

This paper addresses the methodology for assessing the precision and accuracy of an NTG and uses NCTUns's *stgr* [5] as an example. Exceptionally, *stgr* does allow PbP analysis. Because of the way that the NCTUns simulator is formulated (it employs protocol stack re-entry) it was possible to extract the simulator so that it could be used in a real network setting. The examination contemplates the amount of error that may be introduced into a traffic pattern when using a general-purpose or a real-time (RT) Linux OS. In particular, an acceptable range in time and packet length for traffic generation is obtained, based on the type of OS and network setting and how an RT kernel can improve the accuracy and stability of generated CBR traffic.

II. Methodology

In this paper's methodology, video and audio traffic patterns are classified into two types, CBR and VBR. CBR traffic is defined as "*a traffic pattern with a steady bit rate during a time interval*"; and VBR as "*a traffic pattern with a changing bit rate during a time interval*". Variable traffic behavior can be reproduced from recorded network trace-files [2][13] and well-known mathematical probability density distributions (pdfs) such as normal, exponential and Pareto [14][15]. However, stochastic generation of traffic does not reveal per-application traffic characteristics.

PbP traffic analysis is better suited to the analysis of changes in traffic patterns, giving greater detail in comparison with an analysis based on average metrics. For example, if the NTG is required to generate a CBR traffic pattern during a time interval, then bandwidth, a time average metric, is normally used. The CBR pattern is obtained by fixing the bandwidth level and varying the number of packets in a given time period, which is defined by the metric *bandwidth*. This approach may not be accurate enough to generate precise and repeatable traffic patterns, because the NTG relies on the "time period" to calculate the bandwidth level. The NTG may change the number of packets without regard to a constant separation and number in order to achieve the bandwidth level desired. A CBR traffic pattern based on the *bandwidth* metric probably will appear as a VBR traffic pattern from a PbP perspective.

The uncertainty increases if one requires VBR with the metric *bandwidth*. The traffic pattern may be generated by changing the bandwidth levels in every time period according to some random distribution. Although a random pattern based on bandwidth level changes is obtained, it is difficult to obtain the same traffic pattern even if one uses the same seed, because there are many ways to obtain the same bandwidth level by varying the packet length (PL), packet inter-arrival time (PIAT), and packet rate. Consequently, the traffic pattern cannot be termed 'accurate', because the method to generate the bandwidth depends on the algorithm used by the NTG. If a different NTG is applied, it possibly will use a different algorithm to change the bandwidth levels.

For the reasons outlined, it is necessary to redefine some of the concepts and models used to characterize traffic patterns. A traffic pattern can be modelled by using the PL, PIAT and packet throughput (PT) metrics, with the PT metric being dependent on the PIAT and PL variables. PT is given by (1).

$$PT_n = \frac{PL_n}{t_{n+1} - t_n} = \frac{PL_n}{PIAT_n}, \tag{1}$$

in which the $\{t_i\}, i = 1, 2, 3, \ldots, n$ are arrival times at the receiver.[1] Hence, a NTG can produce a specific traffic pattern based on the PT metric by shaping the behavior of the PIAT and PL metrics.

The traffic patterns generated by the *stgr* NTG use the suite of protocols TCP/IP (TCP, UDP, ICMP, *etc.*), and one can select the type of protocol and number of sessions. The traffic is created in different forms by controlling the packet rate and bandwidth, and also by flooding a network in (*greedy mode*)[2]. One can create PbP traffic patterns under UDP[3] by establishing the behavior of the PL and PIAT in every packet through an input trace file. For example, to create a CBR traffic of 10 Mbps, use a PL = 1000 bytes = 8000 bit and a PIAT = 0.0008 s. Thus, one can obtain a 10 Mbps level both in the PbP metric PT and the time average metric *bandwidth*. Because the traffic is generated at the packet level, the traffic pattern characteristics are known with high certainty.

UDP was used in the traffic patterns rather than TCP to avoid TCP's congestion and flow control mechanisms. For example, as is well-known, Nagle's algorithm [17] accumulates sequences of small messages into larger TCP packets. In general, for PbP analysis, fragmentation is avoided by

[1] Though used in Paxson's well-known *tcptrace* tool [16], this metric is otherwise not common in traffic analysis.

[2] In *g*reedy mode,the sender sends as many packets as possible, with a fixed packet length of 1000 bytes, the only option available with TCP

[3] Other options are available such as constant, uniform and exponential.

checking for the maximum transport unit, as are packets below the minimum length (60 bytes on Ethernet). PL includes the link-layer and IP headers, 42 bytes on the test network.

On Pentium processors, the time-stamp register (TSR) is updated on each clock cycle. If using the TSR for providing aperiodic scheduling interrupts, it may still be necessary to adjust for the inaccuracies of programmable interrupt timer (PIC) chip, the MC146818, timings. This is because the PIC provides a measure of the clock speed of the processor, but due to quantization of timings, it introduces an error which can accumulate over time. The KURT system used herein (refer to Section III-A) corrects for this by means of Network Time Protocol (NTP) updates [18].

The TSR is read by the `gettimeofday` system call for timing purposes (not scheduling) whether in normal Linux or in KURT. On a 1.8 GHz Pentium Xeon with Linux kernel 2.4 [6] the `gettimeofday` call was found to take 1.16 μs and, hence, overhead from this source is not expected. Similarly, the TSR is read by the `usleep` call whether in normal Linux or in KURT. However, unlike `gettimeofday`, as already mentioned in Section I, `usleep` causes a descheduling of the calling process and a rescheduling on expiration of the sleep period. It is the ability to reschedule a process to the granularity of `usleep` that is critical. `usleep` takes a single argument – the number of microseconds and is unaffected by most Unix signals except `SIGALARM`.

The local Advanced PIC (APIC) from Pentium P6 systems[4] is also now accessible by software with resolution of a few microseconds [19] and there is also the Advanced Configuration and Power Interface (ACPI) chip with similar resolution.

III. Results

The NCTUns *stg* NTG (available in source code form) was ported[5] to work on a Linux system (as NCTUns originally ran on the OpenBSD OS). The port consisted of changing environment names and other minor incompatibilities between the two Unix variants.

The ported *stg* uses the `sleep()` and `usleep()` commands in trace file mode to control packet delivery. These commands are normally used to control events by delaying by a specific amount of time before executing the subsequent command. In these tests, `sleep` was not used as its granularity is too coarse.

Two variants of the Linux OS were tested, a normal, general-purpose Linux kernel, Linux kernel v. 2.4.9, and the same OS[6] patched with the Kansas University RT (KURT) kernel [10]. The KURT kernel modification allows event scheduling with a resolution of tens of microseconds. However, notice that the NTG runs as a normal Linux application and does not use any special operation from the KURT kernel. It simply delivers packets with the fine granularity that *usleep* provides.

KURT decreases kernel latency by running Linux itself as a background process controlled by a small RT executive (RTE). Scheduling is timed by using the TSR (Section **??**) to act as an aperiodic timer, whilst synthetically providing the standard 10 ms clock for those kernel modules expecting this clock granularity. For other RT Linuces, some of which do not use an RTE for scheduling, refer to [11]. *Hourglass* [20] is a tool to understand the scheduling behavior of the different Linux patches and kernel variants, as the behavior is not transparent. Real-time tasks under Kurt run not as Linux tasks but as tasks controlled directly by the RTE, and must submit an explicit scheduling file to the RTE.

It is important to note that *stg* was *n*ot run as a RT task under KURT but as a Linux task. However, as other processes were kept to a minimum (see comments on the Linux router), after application of `usleep`, *stg* was high if not at the top of the re-scheduling queue. It is the ability to re-schedule at the granularity achievable by KURT that affects the differences observed in the tests.

A. Time event scheduling accuracy

The computers for this test was a Pentium 4 with CPU clock at 1.7 GHz, 512 MB main memory, 40 GB hard disk, and a 10/100 Mbit/s IntelPro 400 network cards.

The time range was split into 54 testing points. The testing points observe the following number sequence: *1E-6, 2E-6 ... 8E-6, 9E-6, 1E-5 2E-5... 8E-5, 9E-5, 1E-4, 2E-4 ... 1*.

The `usleep()` time test consisted of a loop incorporating a `gettimeofday()` and `usleep()` functions. The test ran by calling the `gettimeofday()` function and storing the time in an array and then executing the `usleep()`function with the time delay specified by the testing point. All these instructions were in a loop that iterated 1000 times. Afterwards, `gettimeofday()` was again called, differences were taken and the mean delay value was calculated. By comparing the expected time delay and the mean delay value obtained, the time delay accuracy was obtained.

In Fig. 1, three curves are used, “Expected”, “Normal Kernel”and “RT Kernel”. The *Te-axis* represents the input value for the `usleep()` function, and the *Tm-axis* represents the delay measured with the subtractions of consecutive `gettimeofday()`. The “Expected”curve is used as a reference, depicting the ideal response for a perfect time delay; and the latter two curves represent the tests results for both the Normal and RT Kernel. In theory, if the timer in the OS and computer is highly accurate, the results curve must follow the ideal expected curve.

One can observe that neither of the two OS analyzed can provide microsecond resolution when using `usleep()`. The best performance was observed with the RT kernel, which presented a linear time delay from 0.0001 to 1 s, in contrast to the 0.02 to 1 s range with the normal kernel. Despite the use of the `usleep` command based on the TSR, the normal kernel reverts to the millisecond resolution of the task

[4]The local APIC differs from the global APIC on Intel multiprocessors. The system bus clock is used rather than the TSR clock.

[5]Downloadable from: `http://privatewww.essex.ac.uk/~mpared/perf-tools/srtg.tgz`

[6]KURT is not compatible with Linux kernel v. 2.6.

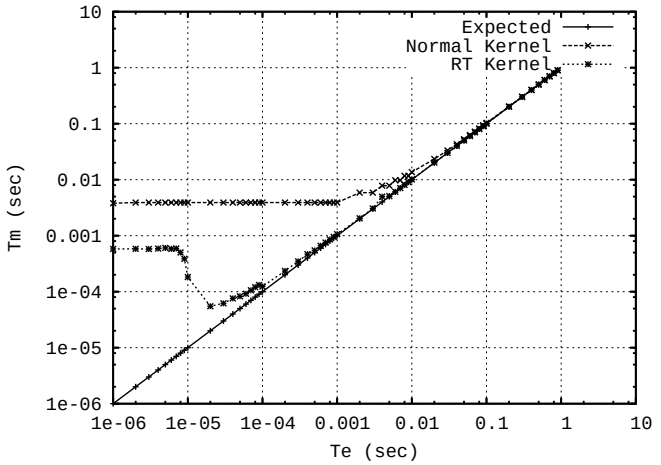

Fig. 1. Time granularity test, T_e is expected time axis, T_m is the measured time axis. Note the log-log scale.

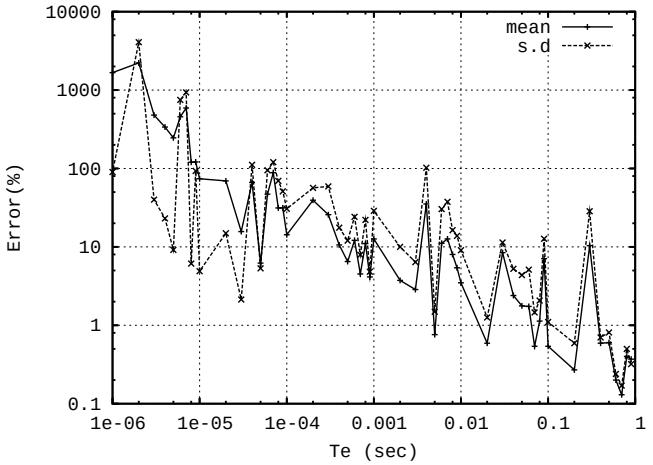

Fig. 2. Polling test with the normal Linux kernel, showing the percentage mean and s.d. of the error. T_e is expected time axis. Note the log-log scale.

scheduler based on the PIC timer. However, the normal kernel presented a stable transition from the 0.02 second linear state to the steady state of 0.004 seconds, giving the opportunity to correct this displacement. The RT kernel presented a local minimum of 0.55E-4 s in the *Tm-axis*, before returning to the 0.6E-6 s level This local minimum coincides with the observations of Srinivasan [10]. He found that the 99.5 % of kernel module events occur within 19 μs of their scheduled time when using the KURT kernel. The minimum is found in the 20 microsecond range in the *Te-axis*. This means that, after crossing the 19 microsecond limit, events are overlapped and, hence, modify the delay time.

Polling was also investigated as an alternative timing mechanism. The end-point of the target time interval is pre-calculated. The time interval is then determined by entering a very tight loop, polling the time registers with `gettimeofday()` until a comparison shows that the end-point has been reached. The results showed almost non-existent errors for the RT kernel up to one microsecond resolution and the normal kernel to two microsecond resolution. However, as soon as external processes are introduced as is necessary when sending data through the TCP/IP stack to the Ethernet interface, then the precision deteriorates. As Fig. 2 shows, there is no particular pattern to the error, which is dependent on the CPU load at any one time and the duration of the measured interval.

B. PIAT precision and accuracy test

Fig. 3 shows the topology used for the traffic generator accuracy test. The network was formed by two computers and a Linux router. Again, the computers and router were all Pentium 4s with CPU clock at 1.7 GHz, 512 MB main memory, 40 GB hard disk, and four 10/100 Mbit/s IntelPro 400 network cards. The "Host A" contains the sender *stg*, which generates UDP packets in the trace file mode; and "Host B" the receiver *rtg* program. In the "Host A" implements the two types of OS to be tested: a normal Linux Kernel, and RT Linux

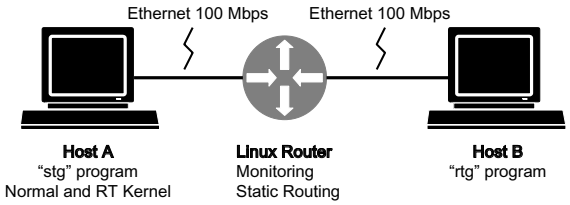

Fig. 3. Experimental setting for accuracy test

Kernel (KURT extension).

Measurement was performed with *tcpdump* on the Linux router, which was setup with static routing, in the Ethernet link connected to the receiver. Normal Linux is run as `gettimeofday` uses the TSR for *tcpdump* timestamps and the TSR is not influenced by the O.S. variant. During monitoring the number of processes running was kept to a minimum, with no graphic mode and no instantaneous screen reports. Argarwal [21] established that the hardware specification was the key restriction on traffic monitoring: the CPU, system bus and memory speeds. By using the UNIX command `top` the CPU load was found, with the Pentium 4 computers incurring a 15 to 25% CPU load when one of the interfaces routed UDP traffic in *greedy* mode and monitoring with *tcpdump* presented 0% packet drops.

ASCII text files with the UDP CBR traffic patterns were supplied to the traffic generator in trace file mode. Fixing PL at 60 bytes and a constant PIAT formed a two minute CBR traffic pattern. The test consisted of instructing the traffic sender,*stg*, to create a set of two minute CBR traffic patterns with different PIATs. Based on the results obtained in the time scheduling event, accuracy test, the the following ranges were established for the test. With the normal Linux kernel the range is from 1E-3 to 1E-1 and for the RT Kurt kernel it is from 3E-5 to

1E-1 seconds. The sequence number is similar to the one used in the last test: 1E-3, 2E-3...9E-3, 1E-2, ...etc.

The purpose of this test was to establish the PIAT ranges that the traffic generator cannot replicate and also observe which type of OS is better suited to precise and accurate delivery of packets. It is well known that different events and processes can affect how a packet is delivered. However, trying to analyze all of them individually is not possible. For that reason, the measurement was performed indirectly at the router, because it is desirable to observe the sum of the possible errors that can be found during the transmission and routing. During the test, the accuracy of the measurements was observed using relative error, by comparing the mean value with the expected value from equation (2); and the precision by comparing the s.d with the expected value, see equation (3). Percentages are employed as a succinct scale for the measurements.

$$PIAT(mean)error = \frac{|PIAT_E - PIAT(mean)|}{PIAT_E} \times 100 \quad (2)$$

$$PIAT(s.d.)error = \frac{PIAT(s.d.)}{PIAT_E} \times 100 \quad (3)$$

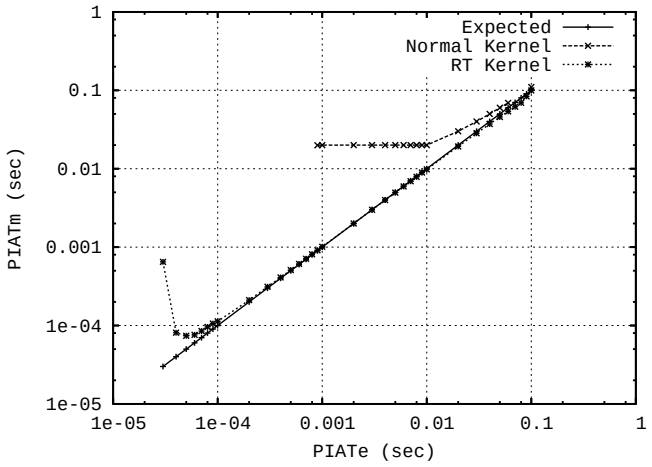

Fig. 4. PIAT delivery test, $PIAT_e$ is expected time axis, $PIAT_m$ is the measured time axis. Note the log-log scale.

Fig. 4 shows how the expected PIAT differs from the measured PIATs, whereas Figs. 5 and 6 show the resulting accuracy and precision in terms of percentage of error. In terms of accuracy, the RT *Linux* kernel presented the best performance with less than 26% of error from the expected value, in comparison with 50% with a normal kernel in the linear area. The RT kernel also presented a better performance in a wider range when delivering packets with very small PIAT, from 1E-1 to 6E-3 s.

In terms of precision, the percentage of dispersion was better in the normal kernel with less than 15% in the linear region in comparison with less of 35% with RT kernel.

By weighting the accuracy and precision, and by establishing a limit of 35% as maximum error, one can find an

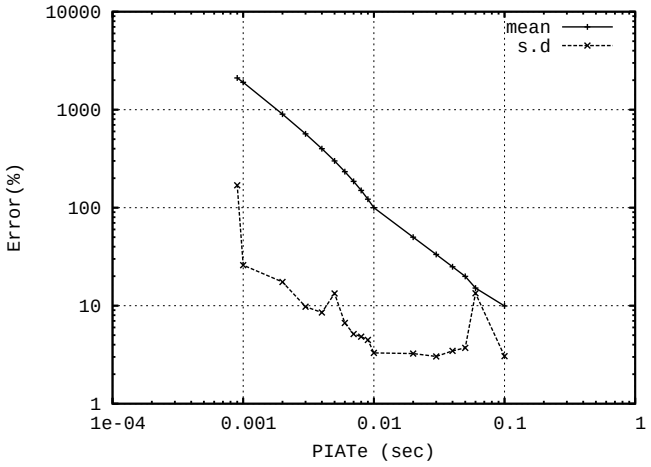

Fig. 5. PIAT test with the normal Linux kernel, showing the percentage mean and s.d. of the error. Note the log-log scale

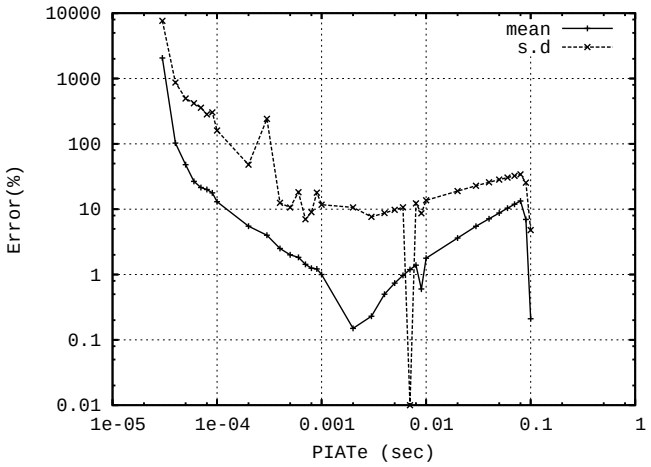

Fig. 6. PIAT test with the KURT RT Linux kernel, showing the percentage mean and s.d. of the error. Note the log-log scale

acceptable range for traffic generation based on the type of OS.

The safe range with the normal *Linux* kernel under test is from 0.03 to 0.1 seconds, and, hence, if one goes below the 0.03 second limit, the error added to the PIAT will affect the traffic pattern. It is possible to extend the range to 0.01, because the normal kernel has good precision. It is possible to consider the displacement seen in Fig. 4 in the range of 0.03 to 0.01 as a systematic error that can be corrected by subtracting the error.

The safe range when using a RT kernel is from 4E-1 to 0.1 seconds and higher time values. Due to the changes in the precision, it is not possible to correct or improve the range obtained beyond that. However, the range presented with a RT system is much wider than the one obtained with a normal kernel. A RT kernel can improve the accuracy and stability of generated UDP CBR traffic. It is possible to create packets with a resolution of 0.0001 s with a RT kernel in comparison to the 0.02-second resolution obtainable from a normal kernel.

The error fluctuates between 0.15% and 14% with a RT kernel, which is considerably better than the error found with the normal kernel from 15% to 34%.

IV. Related work

In [6], a large and impressive study was performed as a prelude to preparation of the *TCPivo* NTG. This study, as a by-product, showed that, at the time of the study, *tcpreplay* had deficiencies on Linux x86 systems, due to the timing mechanisms employed. The main motivation for *TCPivo* appears to have arisen from the need to dimension multi-player games traffic, which is typically centered upon a small number of servers. As such hardware NTGs, such as those from IXIA[7], were deemed unsuitable as they reproduced pdfs rather than traces. Because [6] used very large traces (of the order of one million packets) a problem arose with trace file management, which was largely solved by memory mapping and double-buffering. On a normal Linux system, [6] compared polling to `usleep` and found that while polling (repeatedly calling `gettimeofday` up to packet delivery time) was much more accurate than `usleep` it consumed of the order of 80% of CPU time. To arrive at a compromise between `usleep` and polling, reducing CPU usage but retaining accuracy, firm timers using the APIC timer are introduced. To address the problem of long non-preemptable paths within Linux v. 2.4, two patches [22][23] were introduced.

V. Conclusion

This paper presented a study on how an example network traffic generator, a modified version of NCTUns *stgr*, delivers packets into the network using a packet-by-packet trace file mode. Two analyzes were performed: the first to determine the internal time event scheduling clock granularity on the host on which the traffic generator was installed. Two Linux operating systems were compared: a normal, general-purpose Linux kernel and a real-time *Linux* kernel with the KURT extension. It was found that the RT kernel improves to a great degree the accuracy of the delay command `usleep()`, which is used to control the delivery of packets in the *stgr* program. Polling represents a very accurate and stable method of timing for normal or RT kernel alike. Unfortunately, it is not a stable or robust measurement method if incorporated into a general-purpose traffic generator, as accuracy is strongly dependent on CPU load. A more detailed second test was performed to obtain the precision and accuracy of the packet delivery by checking packet inter-arrival time. The methodology was based on an indirect measurement of the packets on a Linux router in a simple network testbed. This technique will show, if they exist, the systematic errors added by the host. The paper discussed the theoretical meaning of precision and accuracy, and how an understanding of these concepts can be applied to the experiments. It was found that the RT kernel gave a high accuracy for the packet delivery in comparison with the normal kernel. However, it was also observed that the normal kernel presented better precision in its critical regime than the RT kernel. Therefore, the operating regime in which the RT kernel will give precise and accurate results should be known in advance for meaningful analysis of video and audio streaming across a network or at a tight link.

[7]Refer to `http://www.ixiacom.com`

References

[1] R. Q. Hu and D. W. Petr, "A predictive self-tuning fuzzy-logic feedback rate controller," *IEEE/ACM Transactions on Networking*, vol. 8, no. 6, pp. 697–709, 2000.

[2] H. Fu and E. W. Knightly, "A simple model of real-time flow aggregation." *IEEE/ACM Transactions on Networking*, vol. 11, no. 3, pp. 422–435, 2003.

[3] S. Parker and C. Schmechel, "Some testing tools for TCP implementors," IETF, RFC 2398, August 1998.

[4] K. Breaslu, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, X. Ya, and H. Yu, "Advances in network simulation," *IEEE Computer*, vol. 33, no. 5, pp. 59–67, 2000.

[5] S. Y. Wang, C. L. Chou, C. H. Huang, C. C. Hwang, Z. M. Yang, C. C. Chiou, and C. C. Lin, "The design and implementation of the NCTUns 1.0 network simulator," *Computer Networks*, vol. 42, no. 2, pp. 175–197, 2003.

[6] W. Feng, A. Goel, A. Bezzaz, W. Feng, and J. Walpole, "TCPivo: A high-performance packet replay engine," in *ACM SIGCOMM Workshop on Models, Methods and Tools fro Reproducible Network Research*, 2002, pp. 57–64.

[7] A. Turner, *tcpreplay 3.x Manual (Beta)*, 2002, `http://tcpreplay.sourceforge.net/`.

[8] R. Rejaie, "On integration of congestion control with Internet streaming applications," in *PacketVideo workshop*, 2003.

[9] W. Feng, F. Chang, W. Feng, and J. Walpole, "Provisioning on-line games: A traffic analysis of a busy counter-strike server," in *Internet Measurement Workshop*, 2002.

[10] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus, "A firm real-time system implementation using commercial off-the-shelf hardware and free software." in *IEEE Real Time Technology and Applications Symposium*, 1998, pp. 112–120.

[11] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole, "A measurement-based analysis of the real-time performance of the Linux kernel," in *Real Time Technology and Applications Symposium (RTAS)*, 2002, pp. 133–154.

[12] V. Paxson, "End-to-end Internet packet dynamics," *IEEE/ACM Transactions on Networking*, vol. 7, no. 3, pp. 277–292, 1999.

[13] M. W. Garrett and W. Willinger, "Analysis, modeling and generation of self-similar VBR video traffic," in *Communications architectures, protocols and applications, SIGCOMM '94*, 1994, pp. 269–280.

[14] E. Fuchs and P. E. Jackson, "Estimates of distributions of random variables for certain computer communications traffic models," *Communications of the ACM*, vol. 13, no. 12, pp. 752–757, 1970.

[15] P. Danzig, S. Jamin, R. Caceres, D. Mitzel, and D. Estrin, "An empirical workload model for driving wide-area TCP/IP network simulations," *Internetworking:Research and Experience*, no. 3, pp. 1–26, 1992.

[16] V. Paxson, "Automated packet trace analysis of TCP implementations," in *ACM SIGCOMM'97*, 1997, pp. 167–179.

[17] J. Nagle, "RFC 896 —congestion control in IP/TCP internetworks," 1984.

[18] D. L. Mills, "Internet time synchronization: the Network Time Protocol," *IEEE Transactions on Communications*, vol. 39, pp. 1482–1493, 1991.

[19] U. Walter, "APIC timer module for Linux," 2004, `http://www.oberle.org/apic_timer.html`.

[20] J. Regehr, "Inferring scheduling behavior with Hourhlass," in *USENIX Annual Technical Conference*, 2002, pp. 143–156.

[21] D. Agarwal, J. M. Gonzalez, G. Jin, and B. Tierney, "An infrastructure for passive network monitoring of application data streams," in *Passive and Active Measurements (PAM) workshop*, 2003.

[22] A. Morton, "Linux scheduling latency," 2001, `http://www.zip.com.au/ãkpm/linux/schedlat.html`.

[23] R. M. Love, "The Linux kernel preemption project," 2001, `http://www.zip.com.au/ãkpm/linux/schedlat.html`.