

Performing Time-Sensitive Network Experiments^{*}

Neda Beheshti
Stanford University
nbehesht@stanford.edu

Yashar Ganjali
University of Toronto
yganjali@cs.toronto.edu

Monia Ghobadi
University of Toronto
monia@cs.toronto.edu

Nick McKeown
Stanford University
nickm@stanford.edu

Jad Naous
Stanford University
jnaous@stanford.edu

Geoff Salmon
University of Toronto
geoff@cs.toronto.edu

ABSTRACT

Time-sensitive network experiments are difficult. There are major challenges involved in generating high volumes of sufficiently realistic traffic. Additionally, accurately measuring system metrics is not trivial when highly precise timings are required. The majority of these problems arise because generic network software and hardware components do not provide high-precision timing guarantees.

In this paper, we study the challenges associated with performing time-sensitive network experiments in a testbed, including generating realistic network traffic, emulating delay, approximating large topologies, and collecting high-resolution packet-level measurements. We provide guidelines for setting up testbeds, paying particular attention to factors that may affect the accuracy of experimental results, and we describe obstacles encountered during our own experiments. Many seemingly minor details can have significant influence on an experiment's results and, therefore, require careful attention. We illustrate how some of these issues can be addressed by software tuning. For others, hardware support is necessary, and we demonstrate how NetFPGA, a programmable and configurable network component, can provide the required precision.

1. INTRODUCTION

It is commonly believed that the current Internet has significant deficiencies that need to be fixed. However, making changes to the current Internet infrastructure is not easy, if possible at all. Any new protocol or design implemented on a global scale requires extensive experimental testing in sufficiently realistic settings. While simulation tools provide a means for analyzing aspects of computer networks, the models they use are abstract and restricted. There are many issues in a practical setup that simulations do not consider or over simplify. For example, a real router with many stages of buffering and a unique architecture is usually modelled as a single stage output queue by most network simulation tools.

^{*}This article supersedes University of Toronto SNL Technical Report TR08-SN-UT-04-08-00

On the other hand, performing network experiments is intrinsically difficult for several reasons: *i*) Creating a network with multiple routers and a topology that is representative of a real backbone network requires significant resources, *ii*) Network components (like routers) have proprietary architectures, which makes it almost impossible to figure out all of their internal details, *iii*) Making changes to network components is not always possible, *iv*) We cannot always use real network traces (for example, when there is a feedback loop in the system) and generating high volumes of artificial traffic which closely resemble operational traffic is not trivial, and *v*) We need a measurement infrastructure which collects traces and measures various metrics throughout the network.

These problems become even more pronounced in the context of time-sensitive network experiments. These are experiments that need very high-precision timings for packet injections into the network, or require packet-level traffic measurements with accurate timing. Experimenting with new congestion control algorithms, buffer sizing in Internet routers, and denial of service attacks which use low-rate packet injections [8] are all examples of time-sensitive experiments, where a subtle variation in packet injection times can change the results significantly.

In this work we study the challenges of conducting time-sensitive experiments in a testbed. We provide guidelines for setting up such a testbed, paying particular attention to factors that may affect the accuracy of the experimental results. Although there are many examples of time-sensitive experiments in the literature, to the best of our knowledge, no previous work has studied the challenges in a systematic way.

Our experiments lead us to develop the following guidelines which are intended to aid researchers building testbeds for future time-sensitive experiments:

1. **Identify important traffic characteristics:** The network traffic in the testbed should be representative of the traffic that exists in the actual system being tested. The results of time-sensitive ex-

periments are especially sensitive to subtle traffic properties; thus, these properties must be identified and replicated in the testbed.

2. **Be wary of software configuration details and limitations:** General purpose operation systems offer many options to tweak and tune their network stacks for a wide range of scenarios. Their defaults are not appropriate for every time-sensitive experiment in every testbed environment. Additionally, operating systems often cannot provide the timings guarantees necessary in time-sensitive experiments, particularly for packet-level measurements.
3. **Be wary of hardware configuration details and limitations:** As with software solutions, hardware components may have parameters and limitations which have a negligible effect on normal operation but have a large impact on the results of a time-sensitive experiment. In certain experiments, the requirements for precise timings and configuration may actually dictate the choice of hardware.
4. **Account for the effects of scaling:** Experiments that attempt to emulate a large network in a smaller testbed face many timing issues. Hosts and components that should be logically separate must be conflated together in the testbed. It is necessary to recognize and account for these approximations when scaling down large networks.

It is not possible to create a step-by-step guide for constructing valid time-sensitive experiments that will cover every eventuality; instead, we provide the above general guidelines for approaching such experiments along with specific problems and solutions from our own experience. Combined with the description of experiments published by other researchers, this information is a useful resource for assembling testbeds and performing time-sensitive network experiments in the future.

Throughout this paper, the motivating example is a series of router buffer-sizing experiments we performed. The goal of these experiments was to determine how TCP traffic would behave if Internet-core routers have extremely small packet buffers. Although we continually refer to these experiments, this focus does not limit the paper's contribution because the same inaccuracies that we are addressing are also significant for other time-sensitive network experiments.

As Guideline 2 describes, some of the problems associated with time-sensitive experiments cannot be addressed by configuring software components. Higher levels of timing guarantees require hardware support, and the type of hardware support necessary depends on the experiment. In the context of buffer-sizing experiments, we show how we have been able to address

many of the timing issues using NetFPGA boards [3]. NetFPGA is a PCI-based programmable board containing a Field Programmable Gate Array (FPGA), four Gigabit Ethernet ports, and memory. The board can be programmed to act as an Internet router, and we have modified the current router implementation to provide some of the time-sensitive functionality required for our experiments. The changes support high-precision queue-occupancy measurements which can be used for analyzing traffic burstiness and network performance. This feature is not readily available in current commercial components and cannot be provided by software-based tools alone.

The testbed we assembled for our experiments consists of a mixture of commodity and customized hardware. Servers running Linux generate TCP traffic which passes through NetFPGA routers, commercial switches, and other servers providing network emulation and measurement facilities. Sections 2 and 4 contain more detailed information about our testbed's components and topology.

The rest of the paper is organized as follows. Section 2 motivates the guidelines by showing the effect of following them on the results of a particular time-sensitive experiment. In Section 3 we identify the main sources of time inaccuracy that may exist in an experiment testbed. Furthermore, in Section 4 we describe specific issues that time-sensitive experiments are susceptible to. In the same section, along with each issue, we suggest an approach for alleviating the potential inaccuracy. Finally, we conclude the paper in Section 5.

2. MOTIVATION

An important question to investigate is to what extent the previously mentioned guidelines can affect the results of an experiment. In this section, we motivate our guidelines by applying them to a particular time-sensitive experiment and produce demonstrably different results. Because the guidelines focus on addressing inaccuracies significant to time-sensitive experiments in a testbed, the results they produce should be more representative of the real network being tested. The specific areas that cause inaccuracy in a testbed are discussed in Section 3.

In this example, we build a testbed for performing tiny buffers experiments. The topology is shown in Figure 1. It is a dumbbell topology with a single point of congestion where packets from multiple TCP flows go through the bottleneck router and share a single link. The tiny buffer rules suggests that if the TCP sources are not overly bursty, then buffers that can store on the order of twenty to fifty packets are sufficient for Internet-core routers [5]. This is a typical time-sensitive experiment where the results are highly dependent on the timing of the packet arrivals at the

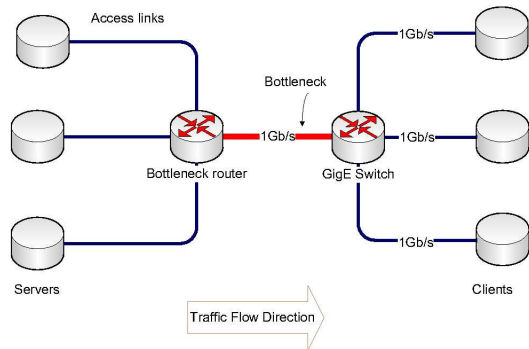


Figure 1: Topology of the buffer-sizing experiment

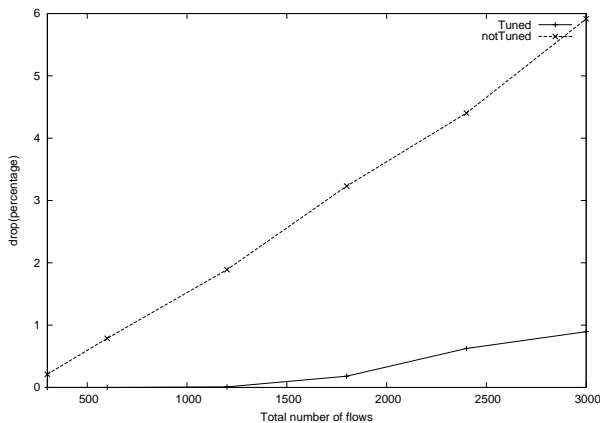


Figure 2: The effect of tuning on a sample time-sensitive experiment

router. In particular, Guideline 1 is a necessary consideration in this example because the tiny buffers results depend upon the traffic arriving to the bottleneck router being smooth rather than bursty. To obtain meaningful results, we must approximate the relevant characteristics of Internet-core traffic in a small testbed.

Here we study how our guidelines can affect the loss rate at the bottleneck link. In this example, the size of the output buffer in the bottleneck router is restricted to only twenty packets (Section 4.5 discusses the challenges of accurately setting a router’s buffer size). We tune some hardware and software components of the testbed to make the generated traffic smooth and spaced out. Each 1Gbps access link emulates 16 slower 50Mbps access links (Section 4.3 discusses how this packet pacing is accomplished). Figure 2 compares the drop percentages of the case where no tuning has been done on the testbed against the case where our guidelines have been applied.

We can observe that the drop rate is considerably less

when the testbed is appropriately tuned versus when no hardware/software tuning is performed. This should not be a surprise, because without any tuning, the traffic appears more bursty, which conflicts with the assumptions of the tiny buffers theory. However, the important point is that the experiment is more likely to produce meaningful results using a tuned testbed environment that is similar to the Internet’s core. This example shows the importance of carefully setting up a time-sensitive network experiment.

3. SOURCES OF TIME INACCURACY

The guidelines listed in Section 1 are intended to address the sources of inaccuracy in testbeds used for time-sensitive experiments, and they were developed in the course of performing our own experiments. To apply the guidelines successfully, it is useful to recognize the three main places where time inaccuracy can appear in a testbed, which are described below.

1. Traffic Generators: Generating realistic traffic is one of the key challenges in modeling a network. Experiments often use a collection of Linux boxes as traffic generators. However, creating a large number of connections, in order to model traffic in networks closer to the Internet core where thousands of flows share each link, is not a trivial task. Also, the core traffic is a mix of packets coming from a broad range of access links: some packets are generated with modems capable of sending only a few kbps, while others are generated at Gbps speeds. The complexities of traffic generation increase when trying to capture this heterogeneity of link capacities using only a limited number of physical machines.

The accuracy of traffic generated by a generic Linux-based machine is bound by the system timer’s resolution. A Linux kernel is not typically capable of providing resolutions higher than 1ms. With links running at 1Gbps, even a large packet of 1500 bytes has a transmission time of less than 12 μ s. Alternatively, commercial traffic generators can easily create a large number of connections, but they can also suffer from inaccurate protocol implementations. As an example, [11] describes differences observed between timings in a TCP Reno packet sequence generated by Spirent Communication’s Avalanche traffic generator [1] and the expected behavior of the standard TCP Reno protocol.

2. Switches and Routers: The architecture of commercial routers is not transparent. For instance, their layers of packet buffering and the existence of undocumented buffers make accurate control of the router’s buffer size impossible. Consequently, the maximum queuing delay that packets can experience cannot be reliably bounded. An example has been reported in [6] where reducing the buffer size of a Juniper T640 router to values below 1ms, resulted in increasing the packets’ de-

lays. Presumably, this is due to the activation of some *hidden* buffers when the size of the main buffer goes below a threshold.

3. Traffic Monitoring Systems: The accuracy of an experiment’s traffic and network components is irrelevant if measurements of interesting properties cannot be gathered with sufficient precision. Packet-level timings are necessary in many experiments, but measuring them with commodity computers and network cards can introduce unwanted variability. Other metrics are more difficult to collect. For example, packet arrivals, departures and drops at router buffers and exact queue sizes in routers are crucial information in a wide range of experiments, but routers are not typically capable of providing this data. Using today’s commercial routers, the only way to record such information is to collect packet traces of all packets entering and leaving the router and infer the timings and queue sizes. This indirect approach hides the exact order of events inside the router and may not provide sufficiently precise measurements.

4. OVERCOMING THE CHALLENGES

This section describes a number of challenges that can arise when running a time-sensitive network experiment in a testbed environment. We also suggest practical approaches and solutions that address these problems, based on our own experiments with tiny packet buffers in routers.

In our testbed, the hosts are Dell Power Edge 2950 servers running Debian GNU/Linux 4.0r3 (codename Etch) with kernel 2.6.18, unless otherwise noted. The servers all have on-board dual-port Broadcom NetXtreme II BCM5708 Gigabit Ethernet cards. To experiment with a variety of network cards we also installed Broadcom NetXtreme BCM5715 Gigabit Ethernet and Intel 82571EB Gigabit Ethernet cards in the hosts. The drivers and version numbers for the network cards are bnx2 1.7.1c, tg3 3.65, and e1000 7.6.15.5, respectively. For the experiments described here, we predominantly used the Intel cards. However, we also performed experiments comparing the three types of network cards. See Section 4.4 for an example.

4.1 Traffic Generation

Having traffic that is representative of the real network being studied is crucial for any time-sensitive network experiment. However, the nature of the traffic depends on the experiment. In this work, we focused on TCP traffic because TCP is the dominant transmission protocol in today’s Internet. Moreover, in the context of our buffer-sizing experiments, TCP sources reduce their transmission rate in response to packet drops in the network and, hence, are more radically affected by tiny buffers than UDP sources.

Although there are many choices for generating traffic, some considerations are independent of a particular generator. The choice of closed-loop versus open-loop workloads can have significant effects on the outcome and analysis of an experiment [15]. In the closed-loop model, clients send requests, wait for them to complete and wait an additional think time before sending a new request. The completion of old requests affects when later requests will be sent. This is not true in the open-loop model where requests are sent regardless of the state of previous requests. It has been shown in [13] that most of the Internet traffic (60-80%) conforms to a closed-loop flow arrival model.

In our testbed, we used the open-source traffic generator Harpoon [17] to create multiple TCP flows. We used a closed-loop version of Harpoon, which was modified from the open-loop original by researchers at the Georgia Institute of Technology [14]. Harpoon allows the size of each TCP transfer and the think time between transfers to be drawn from specified random distributions. In our experiments we use a Pareto distribution with mean 80KB and shape parameter 1.5 for the transfer sizes. These values are realistic, based on comparisons with actual packet traces [14]. The think times follow an exponential distribution with mean duration of one second.

Harpoon sends its traffic through the normal Linux network stack, which provides many options for tuning the behaviour of TCP. The behaviour of most of these options is well documented in many places, for example [16]. Here we briefly describe a few of the options that are of particular interest in a time-sensitive network testbed.

By default, the Linux kernel remembers various connection metrics, so that future connections can use them to set their initial conditions. This can cause undesired results in time-sensitive experiments, as a single period of congestion can affect many subsequent connections. Also, this feature should be disabled¹ if the network’s topology or parameters are changed between experiments; otherwise, the results may not be repeatable, because they will depend on previous runs.

To obtain high aggregate throughput in experiments using a limited number of machines it may be necessary to devote more memory to the socket send and receive buffers. These have a direct effect on the maximum size of the congestion window and the receive window used by a TCP flow. The TCP implementation in Linux kernel has three variables that bound this: `sysctl_tcp_mem` bounds the total amount of memory (number of pages) used by TCP for the entire host, `sysctl_tcp_rmem` sets the amount (in bytes) for the receive buffer of a single socket, and `sysctl_tcp_wmem`

¹Using the command
`sysctl -w net.ipv4.tcp.no_metrics_save=1`

sets the amount (in bytes) for a socket's send buffer. Each of these is an array of three elements, giving the minimum, memory pressure point, and maximum values for the size of these buffers. In [16], the authors provide detailed information on the default value and setup options for these TCP variables. The appropriate value for these parameters depends on the experiment, but the maximum should be large enough to hold all of the buffers for all of the concurrent TCP connections.

Thus far we have only addressed generating traffic and tuning the TCP stack to influence some simple traffic characteristics. Time-sensitive experiments require more control over the traffic. In particular, the above steps have not accounted for all of the effects of scaling that Guideline 4 refers to. In the following sections we examine some of these inaccuracies and suggest solutions for them.

4.2 Packet Delay

Closed-loop protocols, such as TCP, are sensitive to the round trip time (RTT) between endpoints in the network. TCP's congestion control mechanism relies on positive acknowledgments from the receiver, and controls the transmission rate based on the estimated RTT. However, round trip delays that occur in a testbed do not reflect those in a real wide area network. In a testbed that physically resides in several rooms with short wires connecting computers, the observed RTT can be less than 200 μ s. Whereas RTTs on the Internet vary widely and can be as large as several hundred milliseconds. This is an application of Guideline 4 where adding delay to the packets in a testbed is necessary to emulate the long paths of a real network.

We tried two different approaches for emulating network delay: NIST Net [4] and Netem [7]. Both are software tools for emulating different network conditions within Linux. They can be used to add delay to any packet that matches filters based on IP addresses and port numbers. NIST Net delays packets as they are arriving at the system, while Netem usually delays packets as they are departing but can be configured to delay arriving packets as well.

NIST Net is an older solution and will run on older Linux kernels, while Netem is included in Linux kernels and requires no installation. But Netem only began using high resolution timers as of kernel 2.6.22 (released July 8, 2007)[9]. Prior to that kernel, the minimum granularity of Netem's delay was only 1ms which could significantly affect the interarrival times of delayed packets on a 1Gbps link. For this reason, when testing with Netem we used kernel 2.6.24. Even in older kernels, NIST Net uses a high resolution timer which gives it a delay granularity of 122 μ s [4].

In our experiments, we chose to use NIST Net to add delay because we observed some machines lock up and

become unresponsive when delaying with Netem under high network loads. For future experiments, we suggest evaluating the stability and performance of both Netem and NIST Net before choosing one.

4.3 Packet Pacing

As Guideline 1 observes, the testbed's traffic should approximate the traffic of the actual system being tested. The experiment's context may influence the interarrival times of packets belonging to each flow. Consider the packets of a single flow as they traverse the sender's access link toward the Internet core and finally arrive at the receiver's access link. This difference between link bandwidths changes the packets pacing as the packets go from one link to another.

Given that backbone networks typically run at speeds of 1 Gbps to 10 Gbps, and will run even faster in the future, almost every access link runs much slower than the backbone network. For most flows (e.g. where the user is connected to the network via modem, DSL, cable or from a 10Mbps or 100Mbps Ethernet), packets will be naturally spaced out by the network. As packets of a flow cross the boundaries from slower to faster links, the spacing between them should grow.

For experiments where the result is sensitive to packet interarrival times, such as buffer-sizing experiments, it is necessary to consider the different link bandwidths of the emulated network and pace the packets appropriately. We evaluated a number of different approaches to emulate pacing, as described below.

Initially, we tried to create pacing by using slower access links. The simplest method appeared to be to limit the speed that our network cards negotiated. With a 1Gbps link as our core link, we forced our endpoints' network cards to perform at either 100Mbps or 10Mbps. The ratio of 1Gbps core to 100Mbps access links is only a factor of 10 which can not reflect larger core to access bandwidth ratios. Endpoints with 10Mbps access links increase the ratio, but the testbed would now require 100 machines to saturate the 1Gbps link. In most testbeds this is impractical.

Our next attempt used facilities provided by Linux to shape the traffic leaving from a network interface. The queueing discipline used by the transmit queue leading to a network device can be modified, and it is possible to add a token bucket filter to shape outgoing traffic. In the simple case this shaping will affect all traffic leaving the network device and should simulate flows from a single endpoint with a slow access link. To simulate many hosts we require different pacing for different TCP flows, which can be achieved with a Hierarchical Token Bucket (HTB) [2] with multiple classes and filters matching against masked portions of the TCP port

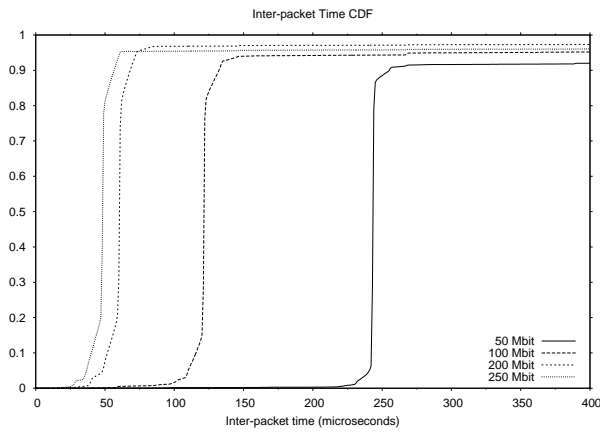


Figure 3: Packet Pacing With TSO disabled.

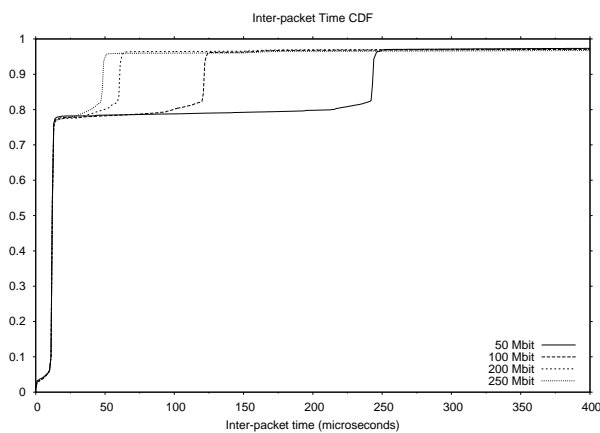


Figure 4: Packet Pacing With TSO enabled.

number. Unfortunately, in our experience, using a hierarchical token bucket caused stability problems. With 16 token bucket classes, under high network load, running the 2.6.24 Linux kernel, a sending machine locked up and required rebooting. We have not determined an exact cause for this yet.

While investigating Linux’s support for token buckets we also experimented with PSPacer [18], which paces packets by injecting gap packets between the real packets. By knowing the speed of the link and controlling the number and size of the gap packets, PSPacer can precisely control the timing of packets without using timers. The trade-off is packets are being sent at the line rate even when the data rate has been limited by pacing, but this is not a problem for fast machines directly connected by full duplex links. Also it is important to note that the gap packets are actually Ethernet PAUSE frames, so Ethernet flow control must be turned off. Again, this is only a small caveat, because Ethernet flow control is unnecessary when the test machines can

easily handle the required traffic. Ethernet flow control is discussed more in Section 4.4.

Figures 3 and 4 show the packet timings that result from pacing packets. The MTU of the network is 1500 bytes which can be transmitted in roughly $12\mu s$ at 1Gbps speeds. In this example, the machine sending the packets uses PSPacer to simulate four different access links with the speeds 50Mb, 100Mb, 200Mb and 250Mb, which can transmit 1500 bytes in roughly 240, 120, 60 and $48\mu s$, respectively. Each of the 100 TCP flows involved is assigned to a speed based on the last 2 bits of its destination port, which distributes them approximately equally between each of the simulated speeds. The figure shows the interarrival times of the four sets of packets received at another computer.

Without TSO, described in 4.4, the majority of the packets have the expected interarrival time for their speed (see Figure 3), implying pacing is working correctly. However, with TSO enabled, in Figure 4 most of the interarrival times are $12\mu s$, which is the minimum transmission time for MTU-sized packets. Thus, disabling TSO is required when employing packet pacing inside Linux. If enabled then the packets that are paced can actually contain multiple packets. Even though their arrival at the network card is paced, individual packets can then appear back to back on the wire after being segmented in hardware. As the figure shows, TSO and this form of pacing are not compatible.

4.4 Network Card Options

In addition to adjusting software parameters (Guideline 2), there are some important hardware related factors that we must pay attention to and tune if needed (Guideline 3), which are described below.

The context of an experiment affects the expected ordering of packets from different flows. Several hardware options alter the intermixing of packets. As in Section 4.3, consider packets belonging to a flow as they traverse the sender’s access link, the Internet core and the receiver’s access link. Near the edges of the Internet, close to the sender or receiver, the packets of the flow may be back to back with other packets of the same flow. However in the Internet core, where there are many flows, packets of one flow will be intermixed with packets of other flows. Therefore, if the testbed traffic is to emulate the Internet core traffic, it should consist of multiplexed flows with intermixed packets. Because there are a limited number of physical links, each carrying a large number of flows in the testbed, this mixing will not happen naturally at the switches and routers. The traffic generating hosts must be configured to inject packets belonging to different emulated users in a mixed way to the output physical links. This is an aspect of Guideline 4 as limited resources in the testbed are emulating a large scale system.

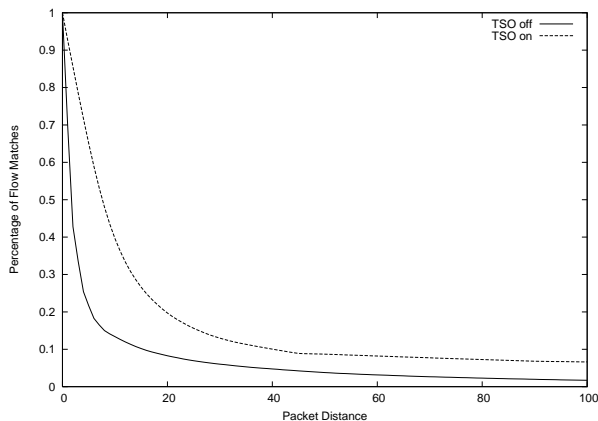


Figure 5: Examining the effect of TSO on the mixing of flows: with TSO enabled, packets of a particular flow are more likely to be followed closely by packets of the same flow.

In buffer-sizing experiments the ordering of packets can greatly affect the throughput and the fairness. If packets of an individual flow arrive back to back in a burst, then a buffer overflow may result in a large number of packet drops for a single flow, while other flows might not experience any drops. A TCP sender that notices frequent drops will reduce its sending speed.

TCP Segmentation Offload (TSO): With TCP Segmentation Offload, or TCP Large Send, TCP can pass a buffer to be transmitted that is much larger than the maximum transmission unit (MTU) supported by the medium. The work of dividing the larger packets into smaller packets is thus offloaded to the NIC. This option frees the CPU from the segmentation overhead but can adversely cause bursts of back to back packets on the wire and hence impact the accuracy of a time-sensitive experiment. This is an application of Guideline 3 where a hardware configuration affects the accuracy of experiments.

Disabling TSO² is crucial if any type of sender-side packet pacing is used, as explained in Section 4.3. We performed a number of experiments to assess TSO's effect. We were mainly interested in the resulting flow mixture and burstiness, which led us to choose the following distance-probability metric: for every packet in the aggregate traffic, determine its TCP flow and look at the subsequent packets to find those belonging to the same flow. Calculate the probability of this match occurring for each offset ahead in the packet sequence. In practice we looked 100 packets ahead. Although it is difficult to interpret the resulting data from a single trace, this metric is useful for comparing different traces.

²Using the command `ethtool -K ethX tso off` where `ethX` is the interface name.

Figure 5 illustrates the effect of TSO on traffic burstiness. The distance-probability metric described above is compared for two experiments: *TSO on* and *TSO off*. Both experiments involve 400 long-lived flows on a single network path with RTT of 100ms. The only difference between the two experiments is whether TSO is enabled on the network cards. The figure shows that enabling TSO increases the likelihood that a subsequent packet will be from the same flow. In other words, packets of a particular flow are more likely to be followed closely by packets of the same flow. In a core network, we expect packets of different flows are well mixed, especially if switches and routers multiplex them in a round-robin order. Therefore, disabling TSO better approximates core traffic by increasing the mixing of the flows.

Interrupt Coalescing (IC): High-bandwidth network interfaces usually use Interrupt Coalescing. Rather than raising interrupts for every single arrival and departure, packets are collected and one single interrupt is generated for multiple packets. IC decreases the per-packet interrupt processing overhead. However, IC also introduces queuing delays and alters the pacing of packets. It has been shown in [12] that enabling IC can make several packets appear to form a burst, even though that may not be the case, and can affect how packet inter-arrivals are correlated over short time periods. IC can also cause bursty delivery of TCP ACKs to the sender which leads to bursty transmission of data segments.

Network card drivers usually provide a way to control the behavior of IC. On Linux, the conventional way to configure IC is through `ethtool` command. However, the details for configuring interrupt coalescing behavior varies depending on the type and driver of the NIC in use. As mention in section 4.1, we have had specific experience with Linux using an Intel NIC with `e1000` driver, a Broadcom NIC with `bnx2` driver and a different Broadcom NIC with `tg3` driver and noticed that not only these device drivers require different parameters for adjusting the IC receive (Rx) and transmit (Tx) operations, but also the outcome is different.

As an example, Figure 6 shows the difference between the CDF of packet interarrival times when IC is turned off for the above mentioned cards. All three experiments involve sending 50 TCP flows from a server machine to a client machine using the Harpoon traffic generator explained in section 4.1. The emulated RTT is 100ms using NIST Net delay emulator. It can be inferred from this figure that both Broadcom NIC cards have more variations in the interarrival times than the `e1000` card. The `e1000` has more than 80% of its interarrival times at roughly 12 s which is the expected time for a 1Gbps line and packets of size 1500 bytes. To turn off the IC in `e1000` and `tg3` drivers, we used the command

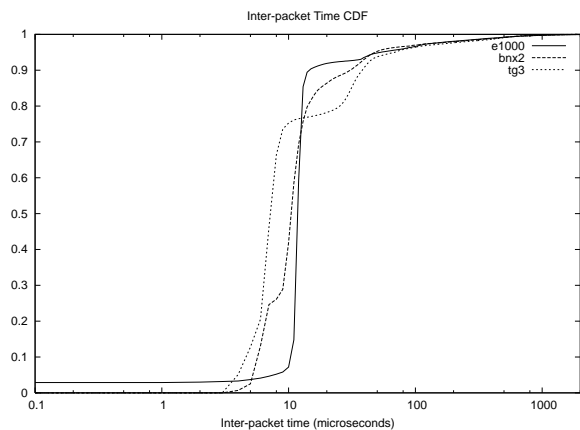


Figure 6: CDF of packet interarrival times for different NIC cards

```
ethtool -C ethX rx-usecs 0 rx-frames 0
          rx-usecs-irq 0 rx-frames-irq 0 tx-usecs 0
          tx-frames 0 tx-usecs-irq 0 tx-frames-irq 0
```

where `ethX` is the interface name, however setting the `rx-usecs` parameter to 0 with the `bnx2` driver is problematic. With version 1.4.44 of `bnx2` it appeared to block packet transmission entirely, and with versions 1.6.9 and 1.7.1c it caused delays up to one second on a simple ping. The minimum usable value for this parameter seems to be 1. Thus, we concluded that tuning IC is dependent on the hardware type and also driver implementation and extra care should be taken when performing time sensitive experiments. As with the TSO, this is an application of Guideline 3 where it is important to be wary of the type of hardware and its configuration in a testbed.

Ethernet Flow Control: Another difference between different NIC cards that we observed was the setup of IEEE 802.3x Ethernet flow control standard. In our testbed we noticed that, by default, Intel cards have flow control enabled whereas Broadcom cards disable it. Therefore, when performing time-sensitive network experiments with a mixture of hardware, researchers should explicitly enable or disable flow control³, even if the desired setting is the default. For experiments on traffic burstiness and flow mixture, we observed unnecessary queuing in the NetFPGA boards and thus we decided to turn off the Ethernet flow control. Additionally, Section 4.3 describes a situation where flow control must be turned off.

4.5 Research-Friendly Router

Commercial routers are usually presented as black

³Using the command `ethtool -A ethX rx off/on tx off/on` for turning off/on flow control where `ethX` is the interface name.

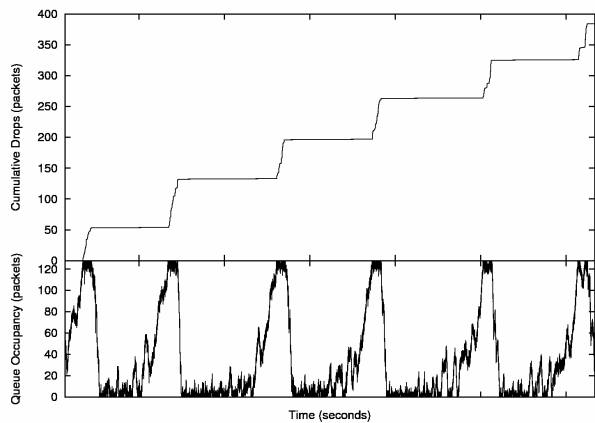


Figure 7: Precise queue occupancy and packet drops monitored with the NetFPGA

boxes by their manufacturers. The details of their architectures are closely guarded secrets, and, although they do allow some configuration and monitoring, they are of limited use in time-sensitive experiments which require precise control and measurement at a per-packet level. For example, Section 3 describes a case where a commercial router appeared to silently employ extra buffering when instructed to only use a small buffer.

Fortunately, NetFPGA boards [3] provide a useful platform to build a research-friendly router. The boards each contain a Field Programmable Gate Array (FPGA), four Gigabit Ethernet ports, and memory. The existing router design allows the size of packet buffers to be set precisely, and researchers can verify the behavior of the router by examining any part of its design.

In addition to requiring precise configuration, the analysis of a time-sensitive network experiment may depend on timings of the internal state of a router. In particular, understanding the behavior of small queue buffers requires monitoring the exact queue occupancy, a feature that commercial routers do not provide. Figure 7 shows a sample of the queue occupancy and packet drop information that the NetFPGA router can provide. In this example, the queue size has been limited to 128 packets, and drops are evident whenever the queue occupancy reaches that value.

To obtain this detailed information, we modified the NetFPGA router design to support instrumenting the router’s output queues. When a packet arrives, departs, or drops at an output queue, the size of the queue and the current clock of the NetFPGA, which has an 8ns granularity, are recorded. Multiple events are then collected in a single event packet which is periodically sent out a specified router port. The event packets contain enough information to reconstruct the queue occupancy’s evolution over time. In our experiments, the

event packets are sent to the host CPU over the PCI bus (rather than being sent over the physical ports of NetFPGA). This ensures that the event packets do not share any output queues or ports with the data packets and have no effect on the timing of data flows. Further details about both the NetFPGA router and our modifications are explained in the Appendix.

In the current implementation, per-flow metrics are very difficult to obtain. The events recorded when packets interact with the router output queues do not contain enough information to identify which TCP flow the packets belong to. Future work will explore classifying events by the flow they belong to and building flow-specific event packets to study how flows interact in a router and how they affect queue occupancies.

4.6 Precise Measurement

Obtaining meaningful experimental results requires accurate measurements. In experimental testbeds there are usually three different domains with properties that need to be measured: inside end-hosts, inside network devices like routers, and on the wire itself. Each presents different difficulties, and measuring some performance metrics requires more precise timing information than others.

The easiest properties to measure are those within the end-hosts. Properties such as the number and size of flows and size of congestion windows are relatively easy to obtain. Operating systems, like Linux, provide statistics about their network stacks. Note that these properties do not depend on precise timings.

Measuring the internal state of a router is more difficult. As described in Section 4.5, commercial routers expose few internal details and can hinder time-sensitive experiments. The same section shows how to obtain the necessary transparency and flexibility using routers based on NetFPGA.

The final accurate measurements we need are from packets travelling on the wire. Although operating systems running on commodity hardware can provide information about each packet sent and received on a network link, the timing accuracy is often not sufficient. There are too many other considerations in the designs of such systems to provide accurate timings. For example, as described in Section 4.4, computers can save CPU cycles spent handling packets by having network cards coalesce multiple packet arrivals together and signal a single interrupt to the system. From the software’s perspective, the packets will have all arrived nearly simultaneously and the true timing information is lost.

Hardware support is necessary to obtain accurate timings of packet transmission times on the wire. The precise reporting capabilities of NetFPGA are useful here as well. Unlike in a general-purpose computer, in a NetFPGA-based router the processing delays experi-

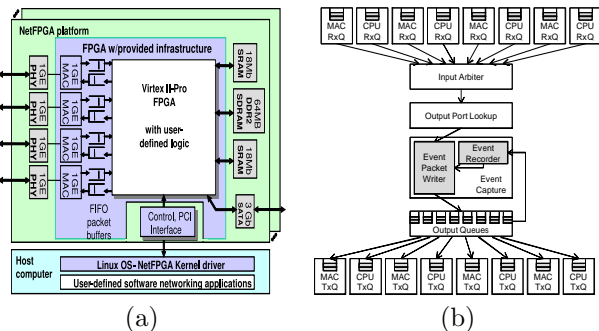


Figure 8: (a) NetFPGA 2.1 block diagram, (b) Router with event capture subsystem.

enced by each arriving and departing packet are small and bounded. Thus, the precise timings that NetFPGA provides for arrivals to and departures from its output queues are also accurate arrival and departure times for packets transmitted on the adjacent links.

5. CONCLUSION

Network experiments are an inseparable part of design and deployment of new techniques and protocols in any network, especially the Internet. In this paper, we study some of the challenges associated with time-sensitive network experiments. We provide guidelines for setting up testbeds, paying particular attention to factors that may affect the accuracy of the experimental results. We further bring attention to the minor software/hardware details that need to be considered in order to get realistic results in a time-sensitive network experiment. We also show how some of these timing issues can be alleviated by tuning software components, while others require accuracy that can only be provided by hardware components of the system. This shows the importance of having configurable/programmable network hardware that can be modified quickly and tuned for specific experiments. We employ one such component (NetFPGA) throughout the paper and show how we have used it to address time-sensitivity issues that we were not able to solve by modifying software parts of the system.

APPENDIX

NetFPGA Router

The NetFPGA is a platform that was designed for network hardware teaching and research [10]. It consists mainly of a PCI form-factor board that has an FPGA, four 1-Gigabit Ethernet ports, and memory in SRAM and DRAM. Figure 8(a) shows the components in more details. Several reference designs have been implemented on NetFPGA: an IPv4 router, a learning Ethernet switch, and a NIC.

All the designs run at line rate and follow a simple five-stage pipeline. The first stage, the *Rx Queues*, receive packets from the Ethernet and from the host CPU via DMA and handles any clock domain crossings. The second stage, the *Input Arbiter*, selects which input queue in the first stage to read a packet from. This arbiter is currently implemented as a packetized round-robin arbiter.

The third stage, the *Output Port Lookup*, implements the design specific functionality and selects the output destination. In the case of the IPv4 router, the third stage will check the IP checksum, decrement the TTL, perform the longest prefix match on the IP destination address in the forwarding table to find the next hop, and consult the hardware ARP cache to find the next hop's MAC address. It will then perform the necessary packet header modifications and send the packet to the fourth stage.

The fourth stage is the *Output Queues* stage. Packets entering this stage are stored in separate SRAM output queues until the output port is free. At that time, a packet is pulled from the SRAM and sent out either to the Ethernet or to the host CPU via DMA. The fifth stage, the *Tx Queues*, is the inverse of the first stage and handles transferring packets from the FPGA fabric to the I/O ports.

The Buffer Monitoring design augments the IPv4 router by inserting an *Event Capture* stage between the Output Port Lookup and the Output Queues. It allows monitoring the output queue evolution in real-time with single cycle accuracy. This stage consists of two main components: an *Event Recorder* module and a *Packet Writer* module.

The Event Recorder captures the time when signals are asserted and serializes the events to be sent to the Packet Writer, which aggregates the events into a packet by placing them in a buffer. When an event packet is ready to be sent out, the Packet Writer adds a header to the packet and injects it into the Output Queues. From there the event packet is handled just like any other packet. Figure 8(b) shows the router's five stage pipeline augmented with the Event Capture stage to monitor the output queues.

A. REFERENCES

- [1] Avalanche traffic generator. <http://www.spirentcom.com>.
- [2] Hierarchical token bucket. <http://luxik.cdi.cz/~devik/qos/htb/>.
- [3] The netfpga project. <http://www.netfpga.org/>.
- [4] M. Carson and D. Santay. Nist net: a linux-based network emulation tool. *SIGCOMM Comput. Commun. Rev.*, 33(3):111–126, July 2003.
- [5] M. Enachescu, Y. Ganjali, A. Goel, N. McKeown, and T. Roughgarden. Routers with very small buffers. In *Proceedings of the IEEE Infocom*, Barcelona, Spain, April 2006.
- [6] Y. Ganjali. *Buffer Sizing in Internet Routers*. PhD thesis, Stanford University, Department of Electrical Engineering, March 2007.
- [7] S. Hemminger. Network emulation with netem. In *Linux Conf Au*, April 2005.
- [8] A. Kuzmanovic and E. Knightly. Low-rate tcp-targeted denial of service attacks (the shrew vs. the mice and elephants). In *Proceedings of ACM SIGCOMM*, August 2003.
- [9] Netem: A linux network emulator. <http://www.linux-foundation.org/en/Net:Netem>, 2008.
- [10] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. NetFPGA—An Open Platform for Gigabit-Rate Network Switching and Routing. In *MSE '07: Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, pages 160–161, Washington, DC, USA, 2007. IEEE Computer Society.
- [11] R. Prasad, C. Dovrolis, and M. Thottan. Evaluation of avalanche traffic generator, 2007.
- [12] R. Prasad, M. Jain, and C. Dovrolis. Effects of interrupt coalescence on network measurements. *Passive and Active Measurements (PAM) conference*, April 2004.
- [13] R. S. Prasad and C. Dovrolis. Measuring the congestion responsiveness of internet traffic. *PAM*, 2007.
- [14] R. S. Prasad, C. Dovrolis, and M. Thottan. Router buffer sizing revisited: the role of the output/input capacity ratio. In *CoNEXT '07: Proceedings of the 2007 ACM CoNEXT conference*, pages 1–12, New York, NY, USA, 2007. ACM.
- [15] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: a cautionary tale. In *NSDI'06*, pages 239–252, Berkeley, CA, USA, 2006. USENIX Association.
- [16] M. Smith and S. Bishop. Flow control in the linux network stack. Technical report, Computer Laboratory, University of Cambridge, England, February 2005.
- [17] J. Sommers and P. Barford. Self-configuring network traffic generation. *ACM/USENIX IMC*, 2004.
- [18] R. Takano, T. Kudoh, Y. Kodama, M. Matsuda, H. Tezuka, and Y. Ishikawa. Design and evaluation of precise software pacing mechanisms for fast long-distance networks. *3rd Intl. Workshop on Protocols for Fast Long-Distance Networks (PFLDnet)*, 2005.