MEASURING NETWORK SERVICE LATENCIES WITH A PROGRAMMABLE DATA PLANE

Master Thesis
École Polytechnique Fédérale de Lausanne

by

Martin Takeya Weber

supervised by:

Prof Edouard Bugnion
Marios Kogias

Lausanne, EPFL, 2019

Acknowledgements

First and foremost, my sincerest thanks go to Marios Kogias, who has guided and supported me in many ways through the completion of this project, both on a conceptual and technical level, and shown a noteworthy amount of patience. Next, I would like to thank Prof. Edouard Bugnion for allowing me to do this project in the Data Center Systems laboratory, for his feedbacks on my work during the lab meetings, and the helpful ideas that allowed us to set up an evaluation test case. I would furthermore like to thank Mia Primorac for her help regarding MoonGen, Margaret Escandari for her patience with me concerning the organisational aspects, and generally the people of the DCSL and VLSC groups for having me with them during the weekly meetings. And last, but not least, Jennifer Schneider, for everything.

Lausanne, July 27, 2019 Martin Weber

Abstract

We present a tool for measuring network service latency times under high network load.
This tool is a proof-of-concept application designed to show the viability of using programmable network data planes as a basis for both generating a high rate of network traffic and precisely measuring observed latencies. Our application uses *Tofino*, a network switch that is optimised for high-speed packet processing and provides a data plane architecture that has been designed to be programmed with the P4 programming language, allowing data plane designs to be mapped onto the hardware very efficiently.
Our application can generate traffic that saturates Gigabit Ethernet links, and can measure observed latencies at a nanosecond-scale. Experiments we have conducted show that we can generate traffic rate patterns that closely approximate the Poisson distribution, and a practical experiment involved measuring the latency behaviour of the BIND9 nameserver under various (both supported and unsupported) loads.

Contents

1 Introduction

Testing a network service on its capability of enduring a high amount of incoming network traffic requires tools that can generate said traffic and quantify the service's behaviour. One way of doing so is *latency testing*: by measuring either the round-trip time of a request-response interaction or the total time required to forward a network packet to its destination, network service administrators can get concrete numeric values that help them better understand the limits of their infrastructure.

Today, there exist various tools that perform network latency testing. They are typically implemented either in hardware (and thus not very flexible) or in software (and thus hitting performance limitations caused by having to cross the hardware–software barrier).

In this work, we present an alternative approach to building latency test tools: rather than sticking to a fixed, inflexible hardware design, we take advantage of *programmable data planes*, i.e. hardware components specialised in processing network data that also expose a high level of configurability, to the point where one can program them and implement custom network processing applications essentially *in hardware* to perform all the desired operations.

To show the viability of this approach, we implement a proof-of-work application that can accurately perform latency tests by generating large amounts (Gigabits) of network traffic and collect nanosecond-scale measurement data.

This paper is structured as follows: we will first give the necessary background information required to understand and motivate our work (Chapter 2); next, we describe our goals for this project, and the approaches taken to building a latency testing tool using a programmable data plane (Chapter 3), followed by a more detailed explanation of the steps we have taken to implement the tool and the challenges we have faced (Chapter 4); next, we evaluate our tool by running a selection of tests to determine if it achieves the goals we have set (Chapter 5); and finally, we list some potential ways how this work could be continued and built upon.

As we make heavy use of a proprietary API whose details we may not disclose, we cannot open the source to the public. If you are interested in studying the code or learn-

ing any other details about our work, please write a mail to martin.weber@epfl.ch and marios.kogias@epfl.ch.

2 Background

In this chapter, we first describe the basic idea behind network latency testing and the role of a packet generator therein; then we give a brief overview of the currently existing solutions and their respective limitations to motivate our work; and finally we will explain the concept of a network data plane, as our solution is heavily based on designing one.

2.1 Network latency testing

Network services have a limit to how much traffic they can endure and how many requests they can serve within a given timeframe without failing. To ensure that services do not collapse under the load of incoming traffic, a service administrator may want to regulate the amount of traffic to which the service is exposed. To do so, however, the administrator needs to know the limits of said service. Determining the limits involves stress testing, i.e. exposing the service to critical types of interactions, like high-frequency traffic, large or complex workloads, invalid or damaged input, and others.

In this work, we will focus on the aspect of *latency testing*, i.e. observing the time required by the service to process requests under a given traffic load. In a test environment, the traffic must be generated artificially; test equipment must therefore be able to both generate high amounts of traffic and accurately measure the observed latency.

One easy and convenient way of implementing a latency tester is to write a regular userspace application that sends and receives network packets by using the underlying operating system's network capabilities, through the use of system calls. However, this rather naive approach causes several issues: first, passing content from userspace through kernelspace to the network interface card (NIC) likely involves several copy operations in addition to other processing work overhead, adding a noticeable delay to the measured times; second, the nature of preemptive scheduling found in most modern general-purpose operating systems adds an additional delay and also unpredictable variation.

There exist techniques to both avoid the scheduling issue (by assigning the process to a dedi-

cated CPU core, e.g. through Linux' `cpuset`) and to mitigate the effects of the OS's abstraction layers (by bypassing its network stack entirely and communicating directly with the NIC, e.g. with DPDK [2] or XDP [15]). Nevertheless, these approaches are limited by the capabilities of the underlying NIC; if a desired feature relevant for performing latency tests—like packet timestamping, measurement data collection, or simply packet generation—are not supported by the NIC, it needs to be handled in software, which leads to some data having to regularly pass the software–hardware barrier.

To overcome the delays and other imprecisions added by having a software component involved in packet generation and measurement operations, *application-specific integrated circuits* (ASICs) can be used, where all (or most) of the application logic is handled directly in hardware. However, this causes another type of issue: with many of the test case parameters and assumptions hardwired in the device, ASICs are typically not very adaptable to different test cases; any change to the experiment setup may require building or buying new equipment.

A more flexible variant of an ASIC is a *field-programmable gateway array* (FPGA), a large set of pre-existing circuit elements that can be programmatically combined together to build a hardware design. The FPGA can be flashed with a new design at any later point in time, allowing the hardware circuits to be adapted to changing test case requirements without having to buy new equipment, while still providing ASIC-level performance and precision. Ultimately, however, programming an FPGA is still largely an act of designing hardware, so very fundamental things like the packet processing pipeline and the communication with the NICs must be programmed first, before being able to implement the application-specific parts (which again should be conveniently adaptable to the various test cases without having to redesign the hardware elements, to provide *some* usability advantage over hardwired ASICs).

A convenient solution for a service administrator would thus be to have an FPGA-like device where the application-agnostic network processing components are already available, and the only parts that need to be programmed are the application-specific components. Ideally, the means for defining that part (e.g. an API or a domain-specific programming language) would also be more specific and less low-level than a regular FPGA utility.

This brings us to the concept of *programmable data planes*.

2.2 Data plane programming

A network device can be thought of as several entities working together; RFC 3746 [19] defines the following two:

- The *data plane* (also *forwarding plane*) is the entity whose task it is to interact with NICs and process packets. This consists of making packet routing decisions based on predefined rules that are either hardcoded or defined in lookup tables.
- The *control plane* is the entity that is more concerned about maintaining a high-level

view on the network topology: it reacts to incoming routing update packets (which it receives from the data plane), and maintains the lookup tables used by the data plane. In some sense, the control plane can be seen as the "brain" that drives the data plane.

It would seem natural that in many network devices, the data plane—which performs relatively simple operations at a high rate—is often implemented in hardware, whereas the control plane—performing less frequent, but more complex operations—is typically implemented in software. However, network devices need not necessarily adhere to this structure: the only defining feature of a data plane is that it processes packets according to given rules. A data plane may thus be all or partly hardwired (optionally with some registers for parametrisation), FPGA-based, or even mostly written in software; a data plane may also be a combination of multiple forms. The fact that not all components of the data plane are hardwired opens the possibility of modifying some aspects of it.

Based on this idea of a flexible, *programmable* data plane, various utilities have emerged to facilitate data plane programming. Two of them are *OpenDataPlane* [4], a project that provides APIs for existing programming languages to interact with and configure a data plane; and *P4* [9], a programming language for describing an entire data plane design. Both projects strive to present a uniform interface to the programmer that abstracts away the underlying architecture.

In the next two sections, we will briefly explain the P4 programming language in a bit more detail, and then describe a

2.2.1 P4

P4 [9] is a domain-specific programming language that was created in a collaboration between Barefoot Networks, Google, Intel, Microsoft Research, Princeton University and Stanford University. There are currently two versions of the language: the initial one as described in the paper ($P4_{14}$), and a second version created two years later ($P4_{16}$). In this section, we describe the concepts of the P4 language based on the $P4_{14}$ specification, but most of the concepts should be applicable to both versions.

P4 models the packet processing device as a pipeline (see Figure 2.1): an incoming packet first goes through a *parser*, where the packet header data is extracted for further use; the packet is then processed in the *match-action unit* (MAU), where various operations may be performed on the packet data itself, associated metadata, and the local state; finally, if the packet is destined for another node, it is reassembled in the *deparser* and pushed onto the link. Each of those stages may (and in some cases must) be defined by the programmer.

In the following sections, we will describe the individual components of that pipeline, and how we use them to implement our latency test application.

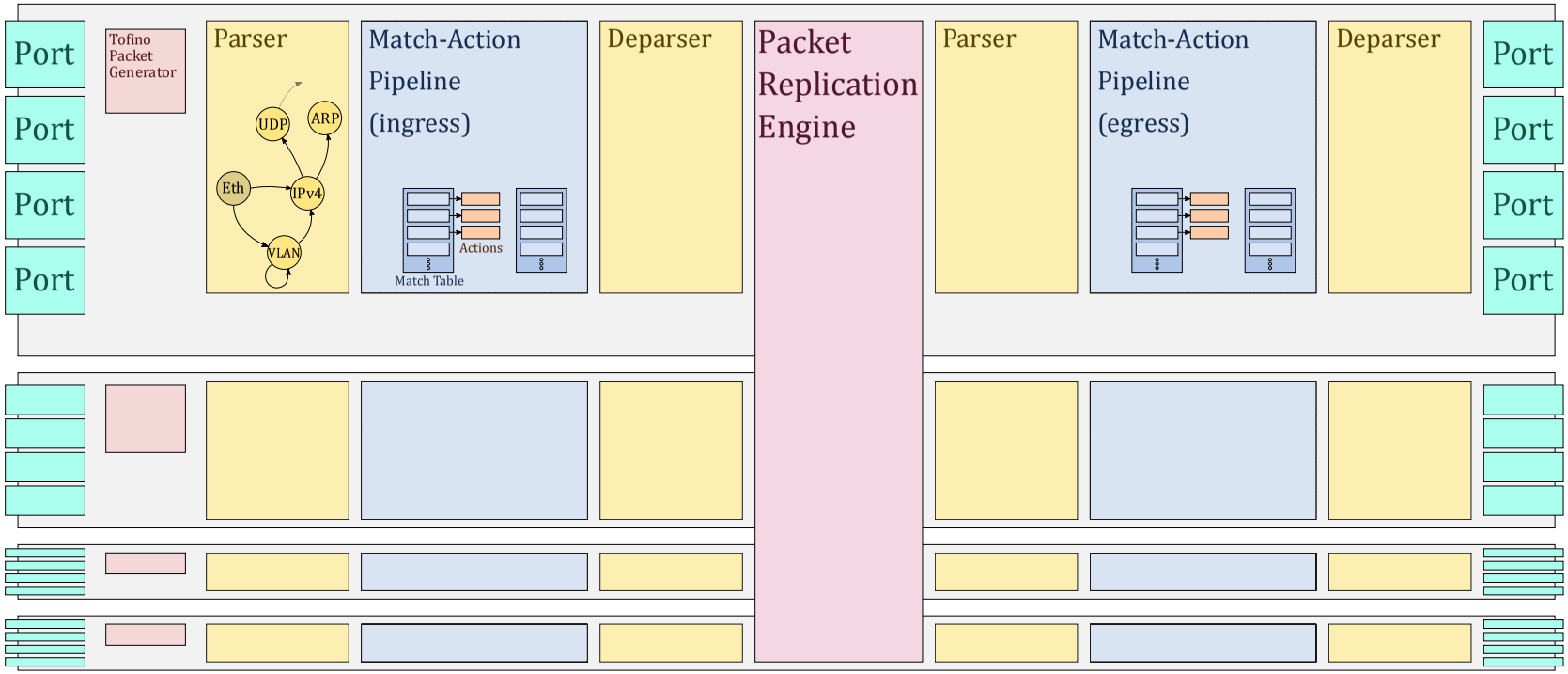


Figure 2.1: Programmable Switch Architecture (PSA) as seen by P4. A switch may consist of multiple pipelines (here we depict a switch with four pipelines), each with its own set of associated network ports and pipeline resources. Packets entering a pipeline pass through ingress first, and are then forwarded to the appropriate egress pipeline by the Packet Replication Engine (PRE).
The architecture implemented by Barefoot Networks' *Tofino* switch (described in Section 2.2.2) additionally provides a packet generator sitting between the network port and the ingress parser.

Parser

The parser is implemented as a state machine: each header parsing step is represented as one or more states, and depending on predefined rules and header data, it transitions to the next header-specific parser state. It must be noted that P4 has been designed with *protocol independence* in mind and therefore does not predefine or assume any specific network protocols like Ethernet, IP, TCP/UDP, or others; both the header definitions and the parsing rules for these (and any custom protocols) must be defined by the programmer.

Besides extracting header data from the packet itself, the parser also initialises additional *metadata* with information like the packet arrival timestamp, the packet length, the incoming network port, and other data that may be useful for processing the packet in the following stages. We are primarily interested in the timestamp.

Match-Action Unit

The MAU consists of multiple entities:

First, the data plane programmer may define *actions* that describe a set of operations to be performed on a packet. Operations may include modifying, appending or removing header data; truncating the packet; updating stateful objects like registers and counters; and modifying metadata fields. Besides using predefined metadata as described above, the programmer may also define custom metadata types for storing additional information; metadata is also used for controlling some aspects of later stages, such as which output port a packet shall be routed to, or whether a packet shall be dropped.

Next, the programmer may define *tables* that take a given input parameter (a *key*); a key may be any packet header or metadata field, like e.g. the source IP address, destination UDP port, packet size, some internal state information, or a combination of multiple such fields. Each line in the table associates a key with a given, existing action, thus creating a definition of which actions shall be performed on what kinds of packets. Table entries are filled in by the control plane.

Finally, a set of *control statements* define which tables to consult, in which order and under which conditions. Conditions may be statements like “this is a valid UDP packet” or “metadata field *xyz* for this packet is equal to zero”.

A network packet goes through two control phases: *ingress*, where an incoming packet is processed and routing decisions are made; and *egress*, where additional operations may be performed before a packet leaves the device again. Packets are forwarded from ingress to egress by the *packet replication engine*.

Packet Replication Engine

The PRE is a set of various components that handle the replication of packets from one place in the pipeline to another. This includes redirecting packets from ingress or egress back to ingress again for a second processing pass (*resubmitting* or *recirculating*), creating copies of processed packets (*cloning*), or—as mentioned above—forwarding packets from ingress to egress.

Most forwarding paths also include queues to avoid congestion caused by packets arriving in the source faster than they can be processed in the destination. This may happen if the data plane has multiple pipelines, and packets get forwarded from a source on multiple pipelines to a destination on a single pipeline; or if pushing packets from egress to the link takes longer than processing them in ingress.

* * *

For high-performance tasks like latency testing, the underlying architecture is at least as relevant as a convenient and uniformised programming interface, and picking the right device architecture is thus fundamental for building a performant latency testing data plane. In the next section, we will describe a device that has been designed with data plane programmability in mind.

2.2.2 Tofino

The *Tofino* [7] is a network switch device produced by Barefoot Networks. It provides a programmable data plane that can be programmed using the P4 programming language. As of this writing, there are 4 different models, each equipped with a different set of high-speed Ethernet ports (ranging from 10 Gb/s to 100 Gb/s). For this project, we are working with a model that exposes 32 10 Gb/s Ethernet ports.

The Tofino data plane is an ASIC that is optimised for being used with the P4 programming language. Basic hardware primitives are (ternary) content-addressable memory (TCAM) registers for mapping match tables, and SRAM registers for storing protocol header and metadata fields.

Keeping the hardware architecture and the programming language this close is relatively efficient because it reduces the amount of abstractions and translations required to map the data plane design onto the actual hardware. This is also emphasised by the fact that programming a Tofino data plane requires a modified version of P4 that reflects the fact that read-write registers are composed of special-purpose arithmetic logic units (ALUs) (rather than simply abstracting that fact away through the compiler).

Next to what is defined by the P4 model specification, the Tofino also contains a *packet generator engine*, which can be configured to generate packets either on specific types of

events, as a one-shot action, or periodically; it can send packets individually or in batches, with parameters such as the packet and batch count and the interpacket gap (IPG). Generated packets are injected into the packet processing pipeline between the NIC and the ingress parser, allowing them to be processed and forwarded like regular incoming packets.

Alongside the Tofino, Barefoot Networks also provides an extensive software development environment (SDE) that includes compilers, debug utilities, a behavioural model (i.e. emulated data plane) for testing purposes, and other tools for analysing and interacting with the data plane, including a Python API for implementing control plane applications.

2.3 Mathematical background

2.3.1 Poisson distribution

The *Poisson distribution* is a discrete probability distribution that is commonly used to model the rate at which events occur within fixed intervals of time or space [11]. An example is the number of mails arriving in a person's mailbox in a single day: seen over multiple days, there may be an average number of 4 mails arriving per day, but the number will vary from day to day; the probability of some specific number of mails arriving in a given day can be approximated by the Poisson distribution.

In a Poisson distribution, the probability of an event occurring at a rate of k times per time interval (with an average rate of λ) is

$$P(k) = e^{-\lambda} \frac{\lambda^k}{k!} \tag{2.1}$$

A graphical representation of some Poisson distributions is shown in Figure 2.2.

The time intervals between events in a Poisson distributions follow an *exponential distribution* [23]; the probability density function (PDF) of a exponential distribution with a mean of $1/\lambda$ is described by

$$E(x, \lambda) = \lambda e^{-\lambda x} \tag{2.2}$$

A graphical representation of some exponential distributions is shown in Figure 2.3.

The exponential distribution is relevant for packet generation, as it can be used to choose *interpacket gaps* to be inserted between two subsequently transmitted packets in a way that the traffic rate follows a Poisson distribution.

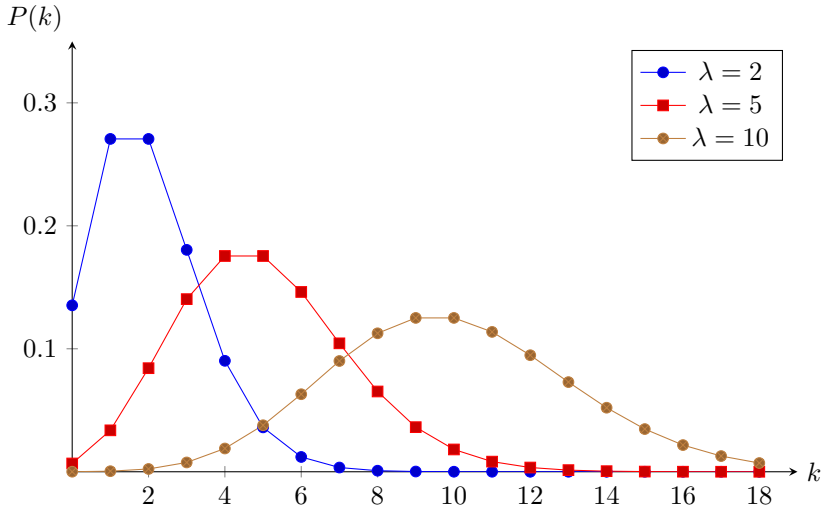


Figure 2.2: Poisson distribution for different values of λ (graph taken and adapted from [22]): k denotes the event rate for a given interval, $P(k)$ denotes the probability of that event rate being observed. Note that the Poisson distribution is only defined for integer values; the lines are displayed for visual guiding.

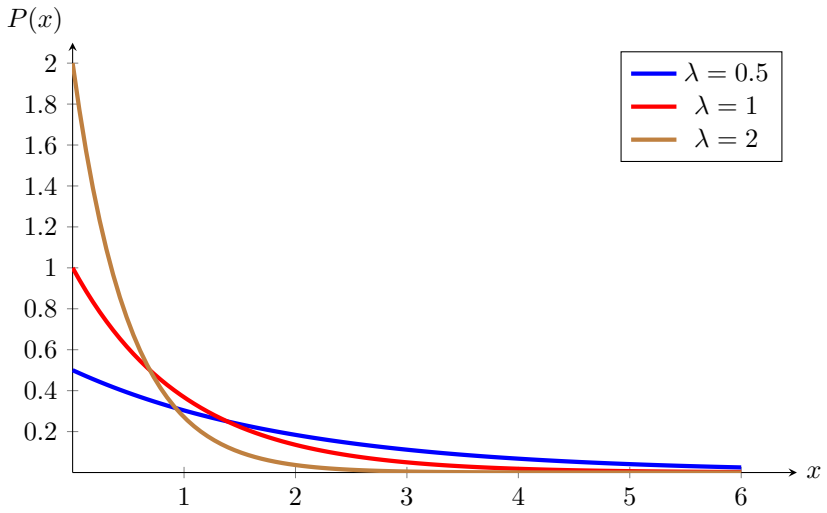
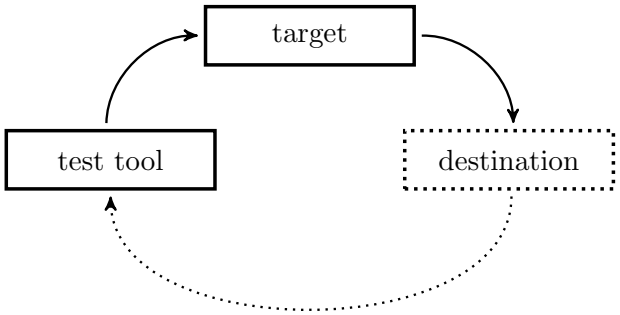


Figure 2.3: PDF of the exponential distribution for different values of λ.

3 Design

3.1 Goals

We want to devise a tool that can generate traffic to a given selection of target devices, measure the time required to process the traffic, and report the measurements back to the tester. To simplify the measurement, the expected test setup with our tool is that it is both the start and the end point of the generated traffic, i.e. all generated traffic should eventually "return" to our test device (ideally directly):



The features we seek to provide are in particular the following:

Multiple traffic types

We want to test both RPC-style services (services that accept requests, and reply back to the sender), and forwarding services (e.g. layer-2 switches and network firewalls). Ideally, our tool will be generic enough to support both cases as if they were the same.

Concretely, we intend to implement the case of testing a simple UDP echo server[1], for two reasons: first, it is the simplest test case, and it also allows us to easily compare our tool with other existing tools; second, it allows us to test forwarding devices, since a test setup where a

[1] An echo server returns a received network packet payload to its origin unmodified.

forwarding device "forwards" all incoming traffic back to our device can be considered the same as a regular UDP echo server.

Furthermore, we intend to implement the case of testing a DNS server, to demonstrate the capabilities of our tool.

Statelessness

To measure the time taken for a network packet to be processed and returned by the target device, we subtract its outgoing timestamp on the local device from its incoming timestamp. For this, we need to keep the first timestamp around until the packet returns.

A trivial way of doing so is to store it locally. However, this would only allow us to measure the latency of a single packet at once; if we send multiple packets out before the first one returns, we would need to keep multiple timestamps locally, which causes two issues: first, we cannot anticipate the required space for storing all the timestamps (and the available space is finite); and second, we need a way to match each returning packet to its corresponding timestamp (as UDP does not guarantee order of delivery).

The second issue can solved by using unique identifiers stored both with the local timestamp and in the sent packet, and using that to do the matching. But this can be improved further: instead of the timestamp identifier we can simply store the timestamp *itself* in the packet. This would also relieve us from having to store the timestamp locally (solving the space issue as well).

Our approach is thus to avoid local state entirely by storing the first timestamp in the packet; upon retrieval, we can obtain the timestamp from the packet itself, and don't need to perform any cleanup. This keeps the complexity relatively low.

Usability

Besides providing a tool for running tests, we want to provide the user a convenient way of describing a test setup. Hence, instead of passing the test parameters manually to the tool at each invocation, we have the tool read that information from a configuration file. This also makes it easier to exchange test setups between researchers without resorting to shell scripts or other wrapper tools: all that is needed to recreate a test setup is to use the same configuration file.

3.2 Design Overview

As our device acts both as sender and (final) receiver for the transmitted packets, packets will pass through the packet processing pipeline *twice*. The way they are processed the first time greatly differs from the second time.

A packet entering the pipeline from the packet generator will be routed to the target device:

1. In ingress, the target destination's MAC address, IP address and UDP port are set in the packet's Ethernet, IP and UDP headers, respectively. The output port is also defined here, such that the packet will be forwarded to the correct egress pipeline.

2. In egress, the *egress timestamp* is stored in the packet, and the packet is pushed onto the line (the timestamp is taken when the packet is dequeued and enters egress; while this does not account for the time the packet takes to pass through egress, we consider this acceptable, as there is no more queueing/buffering involved until the packet is pushed onto the line).

A packet entering the pipeline from another host will be analysed for updating the statistics:

1. In ingress, the previously set egress timestamp is read from the packet, and the *ingress timestamp* is read from the packet's associated metadata (the timestamp is taken as the packet passes through the parsing stage). Subtracting the egress timestamp from the ingress timestamp gives us the time the packet took to be returned to our switch, i.e. the latency, which is then used to update various statistics.

2. The packet is not forwarded anywhere, and thus dropped.

The following sections describe some design decisions in more detail.

3.2.1 Traffic rate control

To regulate the rate at which traffic hits the target device, after each transmitted packet, our tool needs to wait for a certain amount of time before transmitting the next one. This idle time between packets is called the *interpacket gap* (IPG): time during which no data is transmitted onto the link. The larger the IPG, the lower the rate (and vice-versa).

The packet generator engine in the Tofino can be configured to generate packets with a given IPG. It can also apply some jitter to the given IPG; however, it appears that this cannot be controlled very tightly. But as described in Section 2.3.1, we intend to hit the target device with very specific patterns of varying traffic rates, for which the packet generator's rate control mechanism appears insufficient.

We therefore devise a mechanism to control the traffic rate in the *data plane* instead: rather than applying any IPGs at the packet generator, we let it generate packets as fast as possible; in our packet processing pipeline, we then transmit only a subset of the generated packets while dropping the rest. The IPG is thus defined by the number of packets that are dropped between two subsequently transmitted packets (see Figure 3.1).

```
.---------.         .         .         .         .---------.
|    P1   | (drop)    (drop)    (drop)    (drop)  |    P2   |
.---------.         .         .         .         .---------.
          <------- "interpacket gap" ------->
```

Figure 3.1: Dropping packets to artificially create an inter-packet gap.

3.2.2 Statistics

For storing the collected data, we can again not store each packet's measured latency individually (or we would be severely limited by the available memory space for the amount of packets we can send for a single experiment).

Our approach is therefore to use a *latency histogram* instead: for each range of predefined latencies, we keep a counter that indicates how often a latency in that given range was observed. At the end, the collected data gives us a list of latency ranges and how often they were hit. To allow the user to get a usable breakdown of observed latencies, the latency ranges are configurable.

Besides maintaining a histogram, we also store the maximum and minimum latencies observed, and the total number of packets sent and received (allowing us to determine the overall packet drop rate).

4 Implementation

4.1 Packet format

4.1.1 Timestamp

As our primary goal is to measure the latency—and as we cannot realistically store timestamp information for all "in-flight" packets locally—, being able to store the timestamps in the packets themselves is most important. Our method described above only requires us to store the *egress* timestamp in a packet, i.e. the timestamp when a packet has left our switch.

Embedding a timestamp in a raw UDP packets intended for testing L2-switches, network firewalls and UDP echo servers is trivial, as the content is irrelevant for the communication to work correctly, and space is also not an issue, as Ethernet frames must be at least 64 bytes[1] long, which translates to a minimum size of 18 bytes for a UDP packet payload (which is sufficient for storing a 48-bit timestamp).

Embedding a timestamp in a DNS request, on the other hand, is less trivial: while a DNS response packet mostly contains the same content as its corresponding request, the content *does* matter, and we cannot embed arbitrary data without interfering with regular DNS functionality; it is also important that the information is retained in the reply.

We have found one field in the DNS message format that can hold any custom value: the *transaction ID*. It is set by the DNS client in the request and is returned by the server in the response message unmodified, allowing a client to match a response to its corresponding request. As the field is only 16 bits wide, we embed additional data in the UDP header, exploiting the fact that we are in full control over all incoming network packets at our switch: we can embed 16 bits of the timestamp in the UDP source port of our outgoing DNS request message, which will then find themselves as the UDP *destination* port in the DNS response message (which has no influence on our packet processing otherwise), resulting in a total of

[1] Strictly speaking, 64 *octets*. In this chapter, we assume bytes to be octets (8-bit bytes).

32 bits available for storing a timestamp in a DNS message[2].

A limitation is that with 32 bits we can only represent timestamps up to $2^{32} - 1$ nanoseconds (roughly 4 seconds), after which they overflow and wrap around. While an overflowing timestamp is not immediately an issue (the timestamp subtraction for calculating the latency will still yield the correct result), a timestamp that overflows *more than once*—or even just a latency of 2^{32} ns or more—will cause the latency to wrap as well, leading to wrong results. However, we consider 4 seconds to be an unrealistic latency in a datacentre context, so this limitation should not be too severe.

4.1.2 Host ID

To support sending packets to multiple target hosts, we also need to be able to match any returning packet to the right host. An early version of our data plane design simply maintained a single statistic for all hosts, but we found that data not to be very informative (especially when testing multiple very different hosts).

To avoid maintaining any local state for sent packets, our approach is thus now to embed the host ID in the sent packet as well. This is again trivial for raw UDP packets (it adds another 16 bits, which still fits into the 18-byte size limit of a minimum-sized UDP packet); however, it turns out to be impossible for DNS requests (at least without severely reducing the amount of space available for storing the timestamp). As a consequence, we have decided not to support the testing of multiple DNS hosts simultaneously.

4.2 Packet generation

The Tofino packet generator engine consists of eight *packet generator applications* that can each be configured individually. Each application reads the packet data from a shared *packet buffer* (so an application must be told the offset and length of the packet data in the buffer). As the Python API provided by the Tofino SDE also covers the packet generator engine, populating the packet buffer can be conveniently done using libraries like Scapy [5].

As already mentioned, we cannot rely on the Tofino's packet generator engine to do traffic rate control, we instead apply the method of dropping packets later in the packet processing pipeline as described in the Design chapter. We choose to drop packets in egress—after it has passed through the queue—to simplify calculations for producing the desired transmission rate, as the processing time for each packet is roughly the same, regardless of whether it is transmitted or dropped.

[2] We could also just store the 48 bits in the source MAC address with the same reasoning (we do not care about our MAC address for incoming packets), but we concluded that this may have unintended side-effects, as we would be thrashing any involved node's MAC address tables by essentially "announcing" a different MAC address with every new packet.

In this section, we describe the steps involved in producing the right traffic rate.

4.2.1 Determining the interpacket gap

We control the packet rate by setting the number of packets to be dropped for producing a given IPG. However, we expect the user input (from the configuration file) to be given in "packet transmissions per second"; we therefore need to perform a conversion.

The first step is to determine the *interarrival time*, i.e. the time difference between two subsequently transmitted packets:

$$\text{interarrival time [ns]} = \frac{10^9}{\text{transmission rate [pkts/s]}} \tag{4.1}$$

Next, we determine how many packets are generated during that interarrival time; for this, we need the *intergeneration time*, i.e. the time difference between two subsequently *generated* packets:

$$\text{packets per IA time} = \frac{\text{interarrival time}}{\text{intergeneration time}} \tag{4.2}$$

As this also includes the packet to be transmitted, we need to remove that one to get the number of packets in an interpacket gap:

$$\text{packets per IPG} = \text{packets per IA time} - 1 \tag{4.3}$$

—or to summarise:

$$\text{packets per IPG} = \frac{10^9}{\text{intergeneration time} \times \text{transmission rate}} - 1 \tag{4.4}$$

The transmission rate is an input parameter, so it is known. The intergeneration time can be determined experimentally (this is more precisely described in Section 5.1).

4.2.2 Varying the interpacket gap

Poisson point process

To produce packet rates that follow the Poisson distribution, the interarrival times need to follow an exponential distribution (Section 2.3.1). Generating a random value T from a exponential distribution with mean λ can be done using a random value U from a uniform distribution in the interval $(0,1)$ as follows [23]:

$$T = \frac{-\ln(U)}{\lambda} \tag{4.5}$$

While this looks fairly trivial, we must keep in mind that the data plane has rather limited arithmetic capabilities, and operations like dividing values or calculating a natural logarithm may not be performed efficiently enough.

To work around this limitation, we apply a method called the *Poisson point process*: instead of generating random IPG values in the data plane at runtime, we pre-generate a sample set of values in the control plane during the setup, when speed is not very important. At runtime, the data plane then only needs to pick one of the pre-generated values in a uniformly random way.

The pre-generated values are passed to the data plane through a table; picking a value from that table at random thus consists of generating a uniformly random index. However, while P4 *does* provide the directive `modify_field_rng_uniform()`, we have found—at least on the provided switch emulator—that directive not to be "fast" enough, i.e. multiple subsequent calls to `modify_field_rng_uniform()` yielded the same value, and the generated values only changed once every few seconds.

We thus had to resort to an alternative approach: the Tofino switch implements a hash algorithm called `random`. While this hash algorithm yields the same value for the same input value, we may now pass a highly frequently changing value as input (such as the current timestamp), and we get a sufficiently frequently changing random variable as output.

Packet rate control

Dropping the packet in egress is achieved by using a register that stores the currently active IPG. For each generated packet, we then decide whether to forward or drop it as shown in Algorithm 1.

Algorithm 1 Dropping packets in egress

```
if register = 0 then
    forward packet
    register ← random IPG from table
else
    drop packet
    register ← register − 1
end if
```

Packet count control

To control the total number of packets to be sent to the target device, we initially configured the packet generator application such that it used the "one-shot" packet generation trigger type, and simply configured it to generate a specific number of packets before stopping.

However, with our method of packet rate control, it is now impossible to anticipate the number of packets to be generated, as that number now also contains all the packets that will be dropped in egress for the rate control, and we can only make a rough estimate based on the average number of packets that will be dropped (as the IPGs are not known ahead of time).

To work around this, we had to apply three changes:

- In the packet generator engine, instead of a one-shot generation of a known number of packets, we keep generating packets infinitely.
- In the packet processing pipeline, we drop all generated packets in egress as soon as the number of transmitted packets (tracked by the TX counter) reaches the desired number of packets.
- In the control plane, we periodically check the TX counter, and as soon as we detect that the desired number of packets have been sent, we stop the packet generator.

The first change guarantees that we transmit a sufficient number of packets, the second change guarantees that we do not transmit more packets than necessary, and the third change guarantees that our application eventually stops.

4.3 Routing

Packets are injected into the pipeline through a dedicated port (in our case port 68), which allows us to distinguish them from packets entering our data plane from the network.

We allow splitting the traffic up to multiple targets: for each packet, the ingress controller picks the next host in a lookup table, essentially cycling through the target hosts in a round-robin fashion. It then populates the following fields in the packet's header: MAC address, IPv4 address and UDP port number; the switch port number is also set in the corresponding metadata, to allow the packet replication engine to push the packet into the right egress pipeline.

A target is thus defined by the following 4-tuple: switch port, MAC address, IP address, UDP port. It is possible to specify the same target more than once, e.g. to route twice the amount of traffic to a certain host in comparison with others.

4.4 Statistics

As described in the Design section, we aggregate measured latency values into a histogram. In P4, a histogram can be implemented by using lookup tables with ranged keys (where the key is the measured latency), and by defining counters that are associated to each table entry. Each table lookup that results in a match then increments the associated counter.

While implementing this histogram, we hit a technical limitation: on the Tofino, ranged table keys are limited to 20 bits, whereas timestamps (and thus calculated latencies) have 48 bits[3]. We have decided to truncate the latency values to their lowest 20 bits and apply the table matching on those; it is thus possible to define only ranges from 0 to $2^{20} - 1$ ns (roughly 1.049 ms), which we consider sufficient (any latency that cannot be matched is associated a default entry `more` in the table in our histogram).

4.5 Control plane

The Tofino SDE exposes a C API that can be used to interact with the control plane, and a wrapper API for Python 2, allowing us to write our control plane application in Python.

When run, our control plane performs the following operations:

1. Read all the experiment parameters from a configuration file, `config.yml` (Section 4.5.1);
2. Initialise the tables (target host information, histogram ranges, basic switch information like the IPv4 and MAC address), and registers and counters (TX/RX counters, IPG counters);
3. Configure and start the packet generator application; then poll the TX register(s) until it hits the desired number of transmitted packets, and stop the packet generator;
4. Synchronise the registers and counters, and display the histogram for the measured latencies, TX/RX counts, and maximum/minimum measured latency; save the histogram to `histogram.csv`.

4.5.1 User configuration

There are numerous parameters that configure the behaviour of our latency testing tool. We therefore provide a convenient way for the user to describe a test setup, by writing a configuration file. The configuration file is then read by the control plane application to set up and run the packet generator and the data plane.

The format used is YAML [8]; the configuration consists mainly of key-value options and lists of such options. The list of available options is shown on Page 21.

[3] Except for DNS packets, which can only hold 32 bits of timestamp data.

```
pipe: {numeric}                    # On models with multiple pipelines, this
                                   # defines into which pipe the generated
                                   # packets are injected.

model: [behavioural, edgecore]     # Whether we run on the behavioural model or
                                   # the physical Tofino switch.

local:                             # Network parameters for the switch itself
  - macDstAddr: {MAC address string}
    ipDstAddr: {IPv4 address string}
    udpPort: {numeric}

hosts:                             # Network parameters for target devices
  - switchPort: {numeric}          # Output port on the switch for this target
    macDstAddr: {MAC address string}
    ipDstAddr: {IPv4 address string}
    udpPort: {numeric}
    dummy: {boolean}               # Not a real host (packets dropped in egress)
  - switchPort: {numeric}
    ...

pkttype: [raw, dns]

pktlen: {numeric}                  # Ethernet frame length
                                   # Ignored for `pkttype` == 'dns'

pktcount: {numeric}                # Number of packets to be transmitted to each
                                   # target

pktrate: {numeric}                 # Total packet transmission rate (split in
                                   # case of multiple targets) in packets/second

ipg: {numeric}                     # Number of packets to be dropped between two
                                   # subsequently transmitted packets.
                                   # Conflicts with `pktrate`.

pktrate: {numeric}                 # Number of packets to be sent per second.
                                   # Conflicts with `ipg`.

ia_distribution: [fixed, poisson]  # Interarrival distribution

histogram_ranges:                  # List of ranges in the histogram
  - [{numeric}, {numeric}]         # - Defines start and end (both inclusive) of
  - [{numeric}, {numeric}]         #   a range.
    ...
```

5 Evaluation

This chapter is split into four sections: first, we describe the experiment that determines the packet *intergeneration time* (Section 4.2.1); next, we run an experiment to show the limits of our Poisson distribution generator; then, we run a latency measurement and compare our results with MoonGen; and finally, we show a practical application of our tool, by testing a DNS nameserver.

The data plane was developed on an Ubuntu 16.04 machine, using the Tofino SDE version 8.4.0 (and the provided behavioural model). For the experiments, it was compiled and applied on the physical Tofino with the SDE version 8.3.0, along with Numpy 1.8.2 for generating the exponential distribution variates. For the individual test setups, we have used further tools that will be described in their respective sections.

5.1 Packet rate conversion

To determine the intergeneration time, we generate a large number of packets, and only transmit the first and the last one to a target host, while dropping all the packets in between. On the target host, we can extract the embedded timestamps of both received packets, and we know how long it takes to generate and drop the given number of packets.

As the packet size may have an influence on the durations, we run the experiment for various different packet sizes; for each packet size, we drop 50,000 packets, and obtain transmission interarrival times as shown in Table 5.1.

Plotting the values gives us a graph as seen in Figure 5.1, which leads us to the assume the following properties:

1. For all packet lengths above a certain threshold t, the packet intergeneration time is proportional to the packet length, plus a constant duration, i.e. it can be approximated by a function $f(x) = ax + b\,;\ \forall x \geq t$.

Packet length [B]	Interarrival time [ns]
64	521,306
128	521,307
192	668,805
256	826,131
384	1,140,946
512	1,455,600
768	2,084,909
1024	2,714,548
1518	3,933,989

Table 5.1: Interarrival times (in nanoseconds) for various Ethernet frame lengths (in bytes) when dropping 50,000 packets.

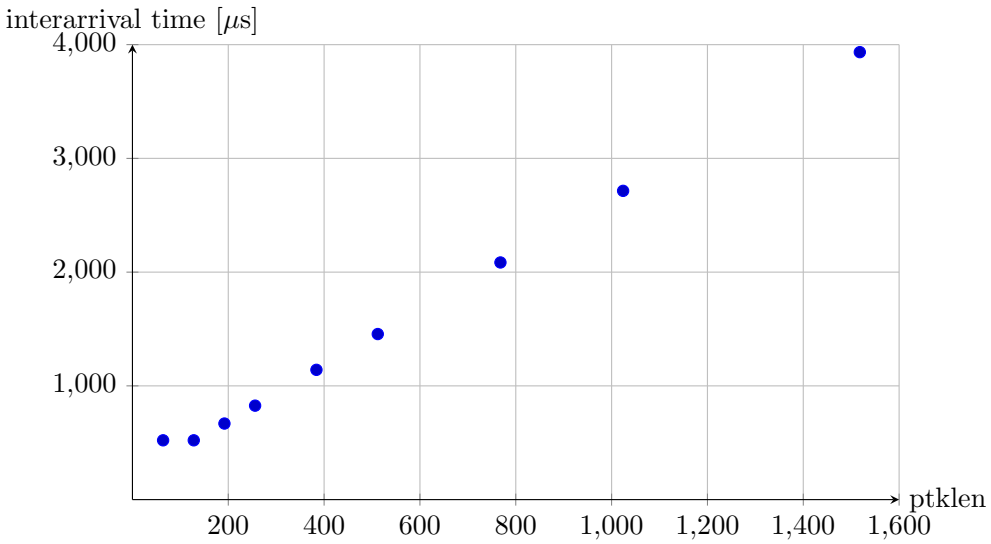


Figure 5.1: Interarrival times from Table 5.1 visualised.

2. For all packet lengths below a certain threshold t, there is a certain "minimum" intergeneration time, regardless of the packet length, i.e. it can be approximated by a function $g(x) = c\,;\ \forall x \leq t$.

For all x (both above and below the threshold t), the interarrival time function p is then simply

$$p(x) = \max(f(x), g(x)) = \max(ax + b, c) \tag{5.1}$$

We will now experimentally determine a, b and c.

5.1.1 $f(x) = ax + b\,;\ \forall x \geq t$

To verify the first property of our assumption, we try to see if we can approximate a linear function of the form $f(x) = ax + b$.

In a first step, we determine a. We do so by calculating the slope of the function in each section lying between two adjacent data points. The results are shown in Table 5.2 (note that we assume packet sizes 64 and 128 to be affected by the second property, so we ignore them for this step). As we can see, the slope is almost the same in most sections, with the exception of the rightmost part (1,024 to 1,518), with a slightly (< 1%) higher slope. We consider this sufficient to assume a value for $a = 2460.369160$ ns/B (for dropping 50,000 packets).

Section	ΔLength [B]	ΔDuration [ns]	Slope [ns/B]
192 .. 256	64	157,326	2458.218750
256 .. 384	128	314,815	2459.492180
384 .. 512	128	314,654	2458.234370
512 .. 768	256	629,309	2458.238280
768 .. 1024	256	629,639	2459.527340
1024 .. 1518	494	1,219,441	2468.504040
Average			2460.369160

Table 5.2: Slope of the approximated function in the sections between two adjacent data points (again for dropping 50,000 packets).

In a second step, we determine b. We do so by first applying a second, hypothetical function $f'(x) = ax$ (based on a found above), on each packet length x. Intuitively speaking, this gives us the intergeneration time for each length under the assumption that there is no added constant overhead time (i.e. $b = 0$). We may then calculate $b_x = f(x) - f'(x)$ (i.e. how much the measured intergeneration time differs from the "hypothetical", no-overhead duration for each packet length x), and find the values as shown in Table 5.3.

Packet length [B]	$f'(x) = ax$ [ns]	$f(x) = ax + b$ [ns]	b_x [ns]
192	472,390.878720	668,805	196,414.121280
256	629,854.504960	826,131	196,276.495040
384	944,781.757440	1,140,946	196,164.242560
512	1,259,709.009920	1,455,600	195,890.990080
768	1,889,563.514880	2,084,909	195,345.485120
1024	2,519,418.019840	2,714,548	195,129.980160
1518	3,734,840.384880	3,933,989	199,148.615120
Average			196,338.561330

Table 5.3: Differences between expected function values based on previously calculated a (with the approximation function going through $(0,0)$) and the measured values, giving us an average constant per-packet "overhead" duration b. The digits after the decimal point are greyed out for readability.

5.1.2 $g(x) = c\,;\ \forall x \leq t$

Based on the data points for the packet lengths 64 and 128 (which differ by only one nanosecond), we can assume a constant lower bound c of 521,306 ns (again for dropping 50,000 packets).

5.1.3 Results

The values for a, b and c that we have found are all for 50,000 packets, so the final step is to transform them into more useful values (note that we divide them by 50,00**1**, as an interpacket gap equivalent to dropping X packets means an interarrival time equivalent to the dropping $X+1$ packets):

- the duration for a single *byte*: $a/50{,}001$ = **0.049206 ns/B**;
- the constant overhead duration for a single packet: $b/50{,}001$ = **3.926693 ns**;
- the minimum duration for a single packet: $c/50{,}001$ = **10.425911 ns**.

We may now use these values to complete the conversion from the user-given packet rate (in packets per second) to an interpacket gap (in packets) as described in Section 4.2.1 of the Implementation chapter, and conduct the remaining experiments.

5.2 Limitations of our approach to Poisson

Given our approach to rate control, we can only generate packets with interarrival times that are a multiple of a packet generation time. This may become problematic when we

generate traffic at higher rates and/or with larger packet sizes, as the interarrival times become smaller, and the approximation with packets becomes less precise, as small variations (caused intentionally by Poisson) have a more noticeable effect compared to larger interarrival times.

In this experiment, we measure the timestamp of each received packet at the target host, allowing us to compare the observed interarrival time distribution with a theoretical, "ideal" exponential distribution.

5.2.1 Setup

We establish a 10 Gbit Ethernet link between the Tofino and a server in the EPFL IC cluster. On the Tofino, we generate UDP packets of a given size, at a given Poisson rate; on the target server, we run a DPDK-based UDP echo server that logs the (local) timestamp of each received packet.

As the 10 Gbit Ethernet link is rather limiting and does not allow us to generate larger packets at a higher rate without experiencing packet drops, we split the traffic up to *eight* target hosts (where seven of them are simply dummy hosts[1]). This results in only an eighth of the traffic effectively passing through the link, allowing us to simulate a higher transmission rate with larger packet sizes. An unfortunate side effect of this is that a setup on the Tofino for generating packets at a rate of e.g. 200 Kp/s *really* only results in an observed traffic rate of 25 Kp/s on the target host, but with an interarrival time distribution granularity of 200 Kp/s nevertheless; the resulting distribution is thus only $1/8^{\text{th}}$ as granular as it could be for 25 Kp/s. In the context of this section, however, this "worsening" is helpful to better illustrate the effects of increasing the packet size and/or rate.

5.2.2 Results

After the experiment, we use the collected timestamps to calculate the cumulative distribution function (CDF) over the measured interarrival times to visualise the effects of increasing the packet rate and/or the packet size; the resulting CDFs are shown in Figures 5.2 and 5.3.

Figure 5.2 shows the CDF for runs at an average packet rate of 200 Kp/s with four different packet sizes. We observe that the packet size does indeed have an impact on the quality of the interarrival time distribution: for smaller packet sizes (64 and 256 bytes), we observe a distribution that is very close to ideal, whereas for larger packets (1024 and 1500), we see a visible deviation from the ideal.

Figure 5.3 shows the CDF for runs at an average packet rate of 2 Mp/s with the same packet sizes as before (200 Kp/s, Figure 5.2). We observe that the packet *rate* also has an impact on the quality of the interarrival time distribution: 64-byte packets at 2 Mp/s result in a distribution that is only marginally more accurate than the distribution for 1500-byte packets in the

[1] Packets routed to a dummy host in the Tofino pipeline are dropped at the end of egress; see Section 4.5.1.

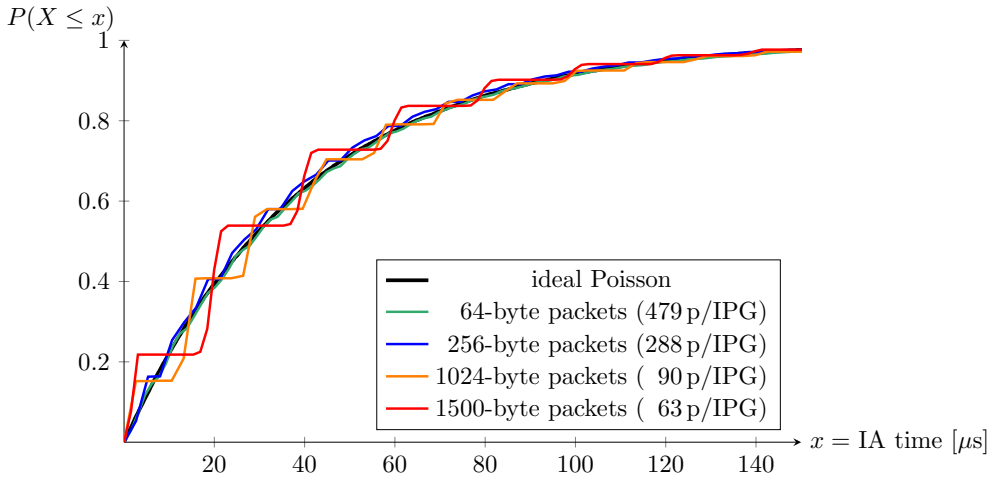


Figure 5.2: Cumulative distribution functions (CDFs) for measured interarrival times at 200 Kp/s. The black line represents the CDF of an "ideal" exponential distribution.

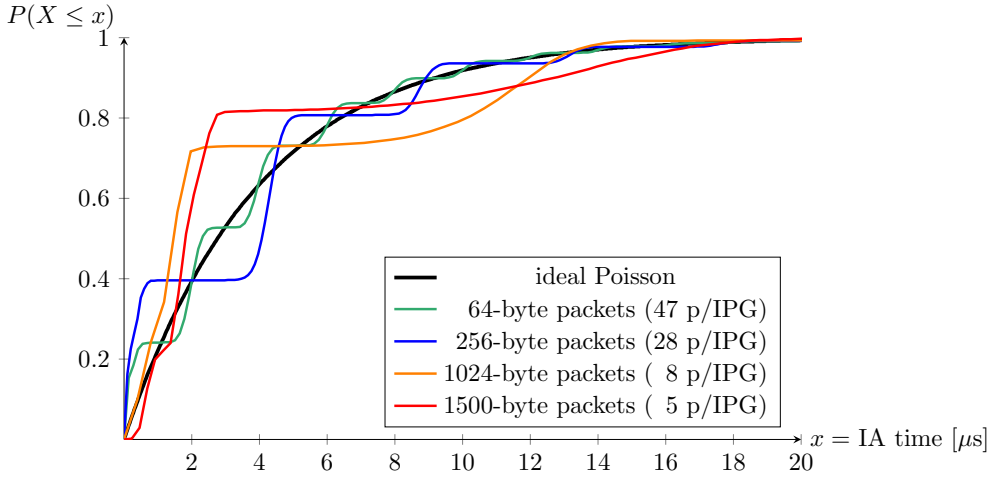


Figure 5.3: CDFs for measured interarrival times at 2 Mp/s.

200 Kp/s case; the 1024-byte and 1500-byte packets at 2 Mp/s barely resemble an exponential distribution at all.

5.3 Comparison with MoonGen

To put the capabilities of our packet generator into context, we compare it with at least one other existing tool for packet generation and latency measurements: MoonGen [13]. In particular, we intend to run a latency measurement experiment on the same application with both our packet generator and MoonGen, and see if we get similar results, or whether (and how) our tool differs from MoonGen.

In this experiment, we set up an echo server that simulates a network service with a given service time, and subsequently run both our packet generator and MoonGen and have them both collect latency histograms.

5.3.1 Setup

On a server in the EPFL IC cluster, we run a DPDK-based UDP echo server that is configured to simulate a service time of 10 microseconds by delaying the response by 10 µs for each packet it receives.

We then configure both our packet generator and MoonGen to generate 256-byte packets (Ethernet frames) at average rates of 10, 50 and 80 Kp/s, using Poisson (this corresponds to 10, 50 and 80 percent of the theoretical maximum rate that can be handled on the echo server; given the varying rates through Poisson, we expect some packet bursts—and thus queueing on the server side—to occur, resulting in above-minimum latencies).

On the Tofino, we transmit 1 million packets. On MoonGen, we let it run for about 2 minutes, so that we transmit at least 100,000 timestamped packets.

As the histogram generated by our tool cannot represent values above 1 ms in any meaningful way, we consider all packets with latencies above 1 ms to be lost.

5.3.2 Results

Figure 5.4 shows the complementary cumulative distribution function (CCDF) for the observed rates (curves for MoonGen are dashed).

For lower rates (10, 25 and 50 Kp/s), our tool observes the same latencies as MoonGen, at least for 99.99 % of data points. For 80 Kp/s, however, we observe an inexplicable deviation from the values measured by MoonGen.

This is due to a (yet unresolved) issue with the histogram table matching in the Tofino: for

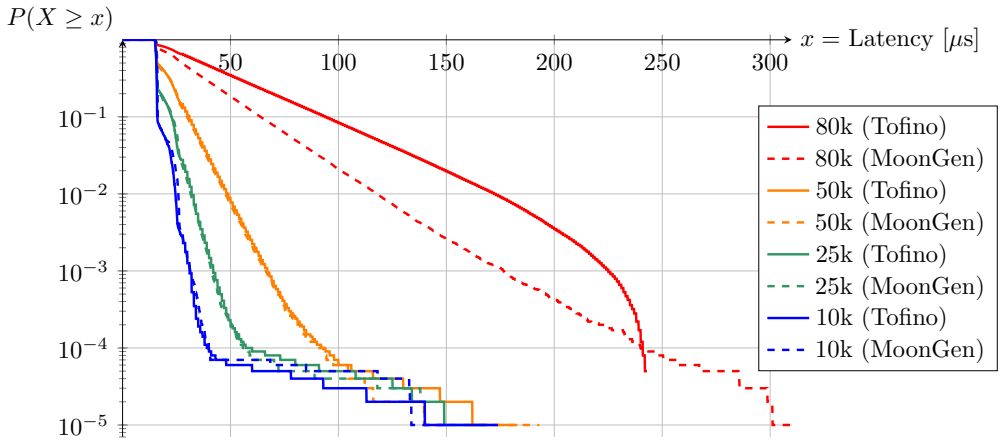


Figure 5.4: CCDFs for latencies of a server configured with a service time of 10 µs, as measured by both the Tofino and by MoonGen, at 10 %, 25 %, 50 % and 80 % of the theoretically maximum sustainable load (100kp/s).

some selections of the of the latency ranges, we also obtain wildly differing results for the lower packet rates (e.g. latencies above 26.3 µs would not be matched and instead be assigned to the bucket for latencies above 1 ms). We therefore assume that the deviation we observe here may also be related this inexplicable table matching behaviour.

In the absence of that misbehaviour, however, we see that our measurements are accurate enough to yield results that are almost identical to MoonGen.

5.4 Practical Application: BIND9

In this section, we use our tool to perform a latency test on an existing application, the Internet Systems Consortium's reference implementation of a domain name server, *BIND9* [14].

5.4.1 Setup

We establish a 10 Gbit Ethernet link between a server in the EPFL IC cluster and the Tofino; the server runs Ubuntu 16.04, and a packaged version of BIND9 (`1:9.10.3.dfsg.P4-8ubuntu1.14`), configured to allow recursion. On the Tofino, we generate 100,000 DNS request packets and collect the obtained latency histogram data; we do so for various different rates. The looked up domain name is predefined and does not change throughout the experiment. We run a few queries before collecting data, to allow BIND9 to cache the record locally (and thus not skewing the results through additional delay introduced by recursive lookups).

The goal is to determine what traffic rates the DNS server can handle without dropping any significant number of requests, and how the latency behaves for higher rates. We expect the overall latency to be lower for low packet rates (as the server does not experience any major

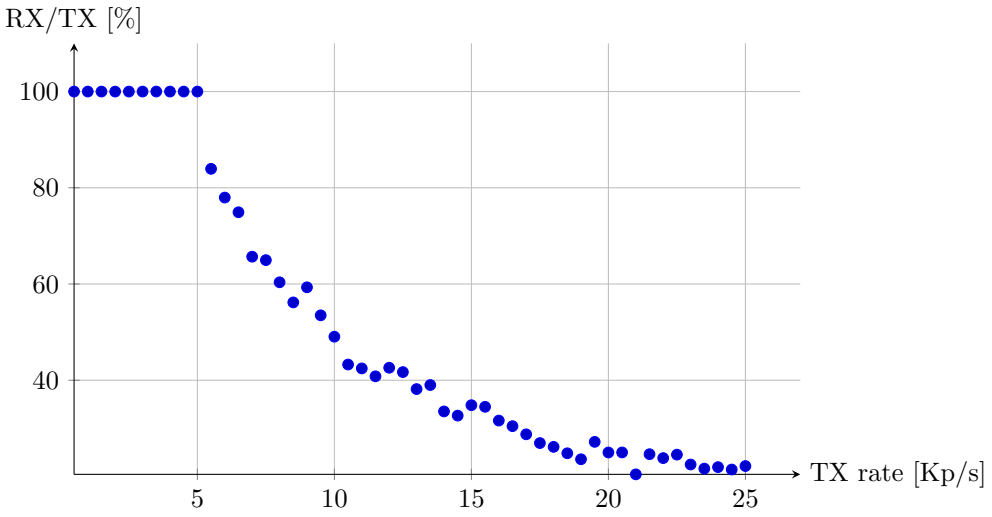


Figure 5.5: Percentage of traffic returning to the switch, for various packet rates.

queueing for incoming packets); we also expect latencies to be come significantly worse once we overstep a certain packet rate threshold that the server can handle, combined with packet loss.

5.4.2 Results

The packet drop rate is reflected in Figure 5.5, which shows the percentage of returned packets for various packet rates; the CDFs for the latency histograms are shown in Figure 5.6.

For packet rates from 0.5 through 5 Kp/s, we observe the opposite of what we expected: the higher the rate, the *lower* the overall measured latency.

We do not know why we observe this distribution. One possible explanation is that higher packet rates may lead the server OS to prioritise the BIND9 process, as it experiences a relatively high load of traffic (and thus work), but this would need to be verified on the server itself, e.g. by running BIND9 on a dedicated CPU core through Linux' `cpuset`. We also don't know whether and how the Top-of-the-Rack (ToR) switch sitting between the Tofino and the target server influences the latency for higher-rate traffic (e.g. whether it prioritises some higher-rate data flows).

Above a certain traffic rate (5.5 Kp/s and higher), we then observe the expected: if we surpass the capacity of the server, packets exhibit significantly higher latencies (this also correlates with the number of packets being dropped).

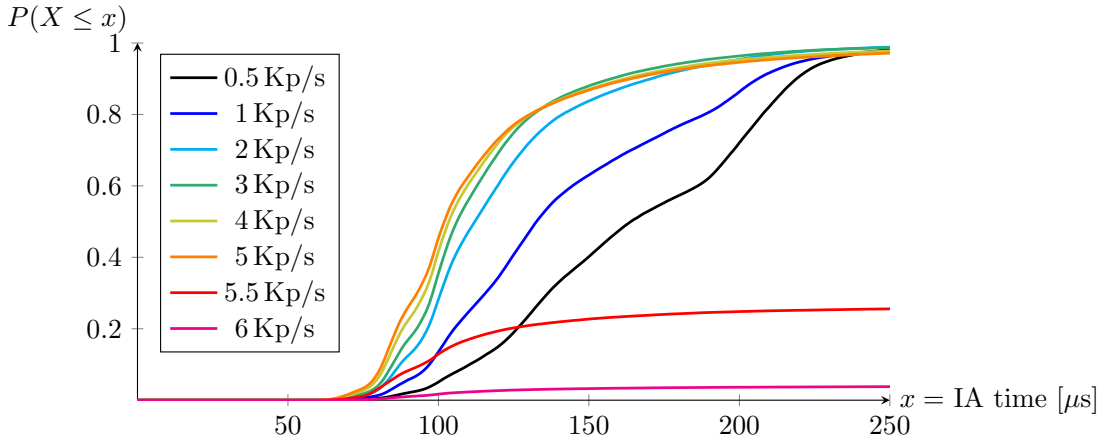


Figure 5.6: CDF for observed DNS request-response times at various rates. As our histograms group all latencies above 1,000 µs together, the CDF simply immediately completes to 1 there, so we do not show the graph all the way up to there.

6 Future Work

As mentioned before, this is currently mostly a proof of concept, with the main goal of showing the viability of using a programmable data plane for the purposes of conducting latency performance tests. Hence, there are a series of things that are left to do, either to just polish the few remaining rough edges in our code, or to implement more functionality, to support various other test cases and scenarios.

Varying packet size and content

Currently, the packet size and content is fixed during the runtime of an experiment, mainly because the Tofino's packet generator engine uses a fixed buffer offset and length to generate its packets. However, there are directives defined by P4 (`truncate`) that allow altering a packet size either according to predefined rules, or (within some margins) at random.

The main challenge here will be to adapt the mathematical formula that derives the interpacket gap from the packet size and packet rate: as one of these parameters are now constantly changing, a fixed conversion may no longer be sufficient.

Modifying a packet content can also be achieved—to some extent—in the data plane: we already do so for modifying and embedding packet header data at various protocol levels. However, these are always associated to some predefined header structure. We have yet not found any mechanism that would allow modifying arbitrary chunks of data in a packet's body.

If so, we could extend our DNS scenario (Section 5.4) to generate DNS requests for various different domains instead of a single fixed one.

Better multi-target support

The multi-target support is currently very sketchy. Mostly this is just related to the frontend (the configuration file): the packet rate is split up, but the packet count is applied to each host individually.

This could be improved by duplicating each packet from ingress to egress for each target (instead of picking a single target each time in a round-robin fashion).

As packet rate control already happens in egress anyway, this would not require much adaption there, but it would avoid the "time stretching" observed in Section 5.2.

No polling

The control plane currently polls the TX register associated to each host to determine when to stop the packet generator. This may introduce some minor delays that may in some rare cases have an influence on the test result (e.g. high-latency packets returning in the timeframe between the transactions having already stopped, but the control plane not having wrapped up the experiment yet).

A way to avoid the polling would be to instead forward a "signal" from the data plane to the control plane in form of a network packet to the CPU port; the control plane—that listens to that port—can thus be notified through more traditional means (like `epoll` or `pselect`), without performing regular polls and synchronisation actions to the data plane.

This mechanism could then also be generalised further, to establish a form of communication channel from the data plane to the control plane (and vice-versa), through the use of regular network channels.

Better selection of latency ranges in histogram

A downside of grouping ranges of measured latencies together is that we may miss some information. An example would be a latency range within which values follow a bimodal distribution: our approach to building the histogram would not show that, and relevant information may thus be missed. The user could manually adjust the latency intervals to more granularly represent the ranges with many values, but there is no guarantee that all the information will be captured.

It would be reasonable to add a mechanism to have the control plane automatically evaluate the data points from an measurement run, adjust the ranges, and run another round of measurements with the adjusted histogram (this could potentially be repeated over multiple iterations). The automated approach makes it more likely to get an optimised histogram, as there is no longer the need for the tedious and error-prone process of manually adjusting histogram ranges.

Ultimately, the user would no longer need to manually specify histogram ranges at all.

7 Related work

Generally, the existing tools can be split into two categories: *packet generators* and *RPC generators*. [16]

Packet generators include tools for measuring network nodes like switches and routers, typically in a data centre environment. Tools that fall in this category are MoonGen [13], TRex [6], Spirent [21] and IXIA [3]. s Application RPC generators include tools that measure the latency of services that reply to given requests (such as in our case a DNS nameserver). Tools that fall into this category are Mutilate [17], YCSB [10] and CloudSuite [1].

8 Conclusions

Network devices exposing a programmable data plane (like the Tofino) appear to be a viable option for high traffic generation and latency measurements; given their programmability on a low level, they are highly adaptable to specific test scenarios.

Despite some technical limitations imposed by the specific architecture we have used, and even with our rather basic proof-of-concept control plane and data plane applications, we are able to conveniently collect information about a target device's latency performance.

Bibliography

[1] CloudSuite – A Benchmark Suite for Cloud Services. http://cloudsuite.ch/. Last visited: June 21, 2019.

[2] DPDK. https://www.dpdk.org/. Last visited: June 21, 2019.

[3] Ixia Makes Networks Stronger. https://www.ixiacom.com/. Last visited: June 21, 2019.

[4] OpenDataPlane. https://opendataplane.org/. Last visited: June 21, 2019.

[5] Scapy. https://scapy.net/. Last visited: July 10, 2019.

[6] TRex. https://trex-tgn.cisco.com/. Last visited: June 21, 2019.

[7] Barefoot. Product Brief Tofino Page. https://barefootnetworks.com/products/brief-tofino/. Last visited: June 21, 2019.

[8] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. YAML. https://yaml.org/. Last visited: July 10, 2019.

[9] Pat Bossart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, 44(3):88–95, Jul 2014.

[10] Brian Cooper. Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB. Last visited: June 21, 2019.

[11] E Bruce Brooks. The Poisson Distribution. http://www.umass.edu/wsp/resources/poisson/index.html. Last visited: June 14, 2019.

[12] Paul Emmeric, Sebastian Gallenmüller, Gianni Antichi, Andrew W. Moore, and Georg Carle. Mind the Gap – A Comparison of Software Packet Generators. *ANCS '17 Proceedings of the Symposium on Architectures for Networking and Communications Systems*, pages 191–203, May 2017.

[13] Paul Emmerich, Sebastian Gallenmüller, Sebastian, Daniel Raumer, Florian Wohlfart, and Georg Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conference 2015 (IMC'15)*, Tokyo, Japan, October 2015.

[14] Internet Systems Consortium. BIND9. https://www.isc.org/bind/. Last visited: June 21, 2019.

[15] IO Visor Project. XDP – eXpress Data Path. https://www.iovisor.org/technology/xdp. Last visited: June 21, 2019.

[16] Marios Kogias, Stephen Mallon, and Edouard Bugnion. Lancet: A self-correcting latency measuring tool. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA, 2019. USENIX Association.

[17] Jacob Leverich. Mutilate: high-performance memcached load generator. https://github.com/leverich/mutilate. Last visited: June 21, 2019.

[18] Eric Martin. How does one interpret probability density greater than one? What is the physical significance of probability density? Is it just a mathematical tool? https://www.quora.com/How-does-one-interpret-probability-density-greater-than-one. Last visited: June 14, 2019.

[19] Network Working Group. RFC 3746 – Forwarding and Control Element Separation (ForCES) Framework. https://tools.ietf.org/html/rfc3746. Last visited: June 21, 2019.

[20] Mia Primorac, Edouard Bugnion, and Katerina Argyraki. How to Measure the Killer Microsecond. *ACM SIGCOMM Computer Communication Review*, 47(5), Oct 2017.

[21] Spirent. Spirent TestCenter Test Modules and Chassis. https://www.spirent.com/products/testcenter/platforms/modules. Last visited: June 21, 2019.

[22] TEX – LATEX StackExchange (user: schwabenchris). Plot the poisson function properly [duplicate]. https://tex.stackexchange.com/questions/282806. Last visited: June 14, 2019.

[23] Wikipedia, the Free Encyclopedia. Exponential Distribution. https://en.wikipedia.org/wiki/Exponential_distribution. Last visited: June 14, 2019.