Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Design and Implementation of a Network Emulator with Stochastic Network Delay Support

Lorenz Grohmann

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Christian Becker
Supervisor:	Dr. Frank Dürr
Commenced:	23. Mai 2023
Completed:	23. November 2023

Abstract

Network delays strongly impact time-sensitive distributed systems that communicate over the network. Today, many distributed systems are still using wired network connections. However, with larger installations, wired connections become less feasible, and the need for wireless communication over WIFI or 5G/6G systems becomes greater. These communication methods bring new challenges as messages experience higher network delay, jitter, and packet errors. Therefore, it is necessary to test these systems and their performance reliably. For these tests, network emulation is a promising method. With network emulation, these systems can be tested on wired hardware that behaves like a wireless one. While there are tools available that can emulate networks, there is a lack of flexibility in defining and following stochastic delay distributions, for example, to emulate the unpredictable behavior of wireless connections. This bachelor thesis presents the design and implementation of a tool that specializes in emulating network delays, which can be adapted flexibly and dynamically. This tool is implemented as a Linux Queueing Discipline, which allows it to be used on all Linux-based systems. To achieve the mentioned flexibility, we present a design that separates the generation of new delays from the enforcement. While the Queueing Discipline runs in the restricted kernel space and delays packets, the delays are calculated outside Kernel Space in User Space and then sent as a message to the Queueing Discipline. This separation allows developers to customize how delays are generated more easily or even write their own applications that interact with the Queueing Discipline. To prove the viability of our tool and explore its limitations, we tested and evaluated it. This evaluation showed that, within the limits of software network emulation, our tool can accurately and reliably delay packages both statically and following stochastic distributions.

Kurzfassung

Netzverzögerungen wirken sich stark auf zeitabhängige verteilte Systeme aus, die über das Netz kommunizieren. Heute verwenden viele verteilte Systeme noch kabelgebundene Netzwerkverbindungen. Bei größeren Installationen werden kabelgebundene Verbindungen jedoch weniger praktikabel, und der Bedarf an drahtloser Kommunikation über WIFI- oder 5G/6G-Systeme wird größer. Diese Kommunikationsmethoden bringen neue Herausforderungen mit sich, da Nachrichten eine höhere Netzwerkverzögerung, Jitter und Paketfehler aufweisen können. Daher ist es notwendig, diese Systeme und ihre Leistung zuverlässig zu testen. Für diese Tests ist die Netzwerkemulation eine vielversprechende Methode. Mit Netzwerkemulation können diese Systeme auf kabelgebundener Verbindung getestet werden, die sich wie eine drahtlose Verbindung verhält. Es gibt zwar Werkzeugs, die Netzwerkeigenschaften emulieren können, aber es mangelt an Flexibilität, wenn es darum geht, beliebige Verzögerungsverteilungen zu definieren und zu verfolgen, um zum Beispiel das unvorhersehbare Verhalten von drahtlosen Verbindungen zu emulieren. Diese Bachelorarbeit stellt den Entwurf und die Implementierung eines Werkzeugs vor, das auf die Emulation von Netzwerkverzögerungen spezialisiert ist, die flexibel und dynamisch angepasst werden können. Dieses Werkzeug ist als Linux Queueing Discipline implementiert und kann somit auf allen Linux-basierten Systemen eingesetzt werden. Um die erwähnte Flexibilität zu erreichen, präsentieren wir ein Design, das die Erzeugung neuer Verzögerungen von der Durchsetzung trennt. Während die Queueing Discipline im beschränkten Kernel Space läuft und Pakete verzögert, werden die Verzögerungen außerhalb des Kernel Space im User Space berechnet und dann an die Queueing Discipline gesendet. Diese Trennung ermöglicht es Entwicklern, die Art und Weise, wie Verzögerungen erzeugt werden, einfacher anzupassen oder sogar eigene Anwendungen zu schreiben, die mit der Queueing Discipline interagieren. Um die Tauglichkeit unseres Werkzeugs zu zeigen und seine Grenzen zu ermitteln, haben wir es getestet und evaluiert. Diese Evaluierung hat gezeigt, dass unser Werkzeug, innerhalb der Grenzen der Software-Netzwerkemulation, in der Lage ist, Pakete sowohl statisch als auch nach stochastischen Verteilungen genau und zuverlässig zu verzögern.

Contents

List of Figures

List of Tables

Listings

1 Introduction

Today, the automation of human processes is an essential topic in almost all industries. While many industries already employ automated workers to increase their production and efficiency, there are many application fields where it is currently impossible to automate large parts of their operations. One of these fields is where automated workers operate in large areas, possibly with or alongside humans. Equipment safety and, more importantly, the safety of human workers are essential in such operations. In order to ensure this, autonomous workers must always be aware of their surroundings and strictly controlled by a central intelligence that observes every part of the operation. This requirement requires a reliable communication that can especially ensure that time-sensitive messages arrive on time. An unexpected delay during communication could have severe consequences if the system does not account for the possibility.

One example of current research projects in this field is using occupational exoskeletons as an automation-assisted tool for workers in large warehouses. Another one is large-scale automated farming equipment that operates alongside human workers in agricultural applications [6g23]. In both cases, the communication must be highly reliable and able to deliver time-sensitive information on time. Without this, the whole system's performance could be corrupted, putting the equipment or, more importantly, human workers in danger.

However, in mobile installations like the two examples above, communication is a big challenge because it is impossible to communicate via wired connections in large and moving deployments. Wired connections behave reliably and predictably. That enables us to use efficient Real-Time algorithms that guarantee that information is sent and processed reliably and on time to ensure the safety of all operation members.

This is not the case with wireless connections. Wireless communication is prone to high network delays that can vary greatly, with more frequent packet loss, interference, and other issues. That is made worse because the environment heavily influences these drawbacks. In a setup with moving workers, the connection's performance can drastically change simply because a worker enters an area with different properties. That makes maintaining stable and reliable communication extremely difficult.

In order to evaluate the impact of network delay on applications, we need tools that are able to emulate the properties of wireless networks on wired connections. While there are already tools available that can emulate properties of wireless networks, they do lack in how flexible they can delay packages. We fill this gap by developing a tool that is able to emulate a network delay that can be flexibly defined to adapt to as many different use cases as possible. Our tools consist of a Linux Queueing Discipline running in Kernel Space that delays network packages and a User Space application that precalculates the delay distribution. This separation enables us to achieve the aspired flexibility.

To determine the capabilities and performance of our tool, we tested and evaluated it. This evaluation has shown that our tool is capable of delaying packages accurately and reliably.

The remaining work is structured as follows:

Chapter 2: Gives an overview of all used components to ensure a basic understanding. This overview includes the Linux Kernel architecture, Kernel modules, the Linux Network Stack, and Qdiscs.

Chapter 3: Describes the system model in which our tool will be used and defines the problems we want to solve and the goal of this work.

Chapter 4: Describes the design of our tool. First, we show possible approaches and evaluate them. After that, we detail our tool's components and explain how they affect its operation.

Chapter 5: Shows the implementation of our tool. We explain the functionality of significant components and how our tool can be used in real-world deployments.

Chapter 6: Shows the evaluation of our tool. We test the different aspects of our tool to determine if we have met our requirements and see where its limits are.

Chapter 7: Summarizes the content of this work and discuss possible future works

2 Background and Related Work

2.1 Background

This section contains a basic explanation of technologies and concepts that have been used or dealt with in this work.
That includes a basic overview of the Linux Kernel architecture and Kernel Modules, an overview of the Linux Network Stack, and a more in-depth explanation of Queueing Discipline.

2.1.1 Kernel Architecture

By design, a Linux System is separated into two parts. The User Space and the Kernel Space.

The Kernel Space is where the Linux Kernel operates. The Kernel is the core of the Linux operating system. It runs with the highest permissions and manages everything on the machine. The main task of the Kernel is the management of shared hardware resources. That mainly includes the CPU and memory but also all other hardware of the system. This management includes the distribution of these resources to processes through scheduling. The Unix Kernel architecture is called a Monolithic Architecture because every essential component is part of the Kernel as one big entity.

The User Space is where everything else runs. That includes User applications but also system services. No matter the importance or the privilege a process has, everything in User Space has significantly lower permissions than the Kernel. Every User Space process can only directly access its memory and has access to a large but limited amount of CPU instructions to perform computations. To interact with other Processes, do I/O, change system behavior, etc., they have to interact with the Kernel and request it to perform these actions on their behalf.

Processes achieve this by using System Calls (syscall) [Ker23]. System calls provide a method for applications to interact with the Kernel. They offer specific interfaces for every action that requires Kernel Space privileges and are provided directly by

the Kernel. While it is possible to invoke syscalls directly, they are usually used via wrapped functions that are part of programming languages and their system libraries.

When a process invokes a syscall, its execution halts, and the context switches to the Kernel. The Kernel then processes the data provided via the syscall and performs the desired action or raises an exception. Afterward, the process is marked as ready to continue and, once the schedular assigns it, can resume its execution.

This separation of User Space and Kernel Space and the restricted communication between them adds an essential layer of security to the system as it ensures that no process can directly modify hardware resources or other processes.

2.1.2 Linux Kernel Modules

Despite its monolithic architecture, the functionality of the Kernel can be dynamically extended and modified at runtime. That is possible using Kernel Modules. Kernel Modules are parts of the Kernel that can be loaded or removed dynamically via interfaces that the Kernel exposes to User Space. System administrators can access these interfaces via the "insmod" and "rmmod" commands as part of all Linux distributions.

Kernel Modules are wildly used to ship additional Kernel features that are not required on every system but can be loaded in on demand. They can provide drivers for system hardware, support for different filesystems, change networking behavior, and much more.

Because Kernel Modules operate as part of the Kernel in Kernel Space, they face special restrictions and risks. It is important to carefully develop Kernel Modules to not introduce bugs into the Kernel, as they will affect system stability. The Kernel runs with the assumption that it is error-free. The Kernel does and can not guarantee recovery from errors that happen during its execution. That includes errors inside Kernel Modules. A bug will likely cause a Kernel Panic and crash the entire system. Development is also challenging due to the restrictions of Kernel Space. Kernel Modules can only use functions they provide or are already part of the Kernel.

In order to add functionality to the system, Kernel Modules must interact and be interacted with from User Space. That is possible using already provided features of the Kernel. One way of doing this is through Character Devices [com]. Character Devices are a way to access physical device data. Following Unix's "everything is a file." philosophy, they are exposed as a file in the "/dev" directory on Linux systems. When a User Space application interacts with these files (for example, by reading

or writing data to them), the information is passed to the corresponding driver in Kernel Space. Using this mechanic, Kernel Modules can exchange information with User Space applications by creating a Character Device with themselves as the corresponding driver, even if they do not manage physical hardware. To do this, a Kernel Module requests creating a new character device with a unique identifier indicating the module as the driver. If the identifier is available, the Kernel will register the new character device, and the corresponding file is created in the virtual filesystem, accessible from User Space.

2.1.3 Linux Network Stack and Queuing Disciplines

The Network Stack of an Operation System allows Applications to send and receive Network Packages. It is part of the Linux Kernel. To handle Network Packages, the Kernel uses Socket Buffers (SKB) [kerb]. Every package gets represented by an SKB that contains a reference to the actual data of the packet, support structures for internal handling, and additional metadata.

When an application wants to send data over the network, it must do this through the Kernel. The application requests a socket to which it will write all data it wants to send. Once the application writes data to this socket, the Kernel creates a new SKB and copies the data into it (into Kernel Space). The Kernel can split the data into multiple SKBs, depending on the protocol the application is using. The Network Stack can then add or modify the metadata of the SKB depending on routing and similar operations. Finally, the SKB will arrive at one of the computer's physical (or virtual) Network Devices (NIC).

If packets are received, it goes the opposite way. The data received by the NIC is converted into an SKB and its metadata. This SKB will then travel upwards in the Network Stack, where its metadata is interpreted. Finally, the data is routed to the corresponding application via a socket from which the application can read to copy the data into User Space.

Network Devices have their respective Driver as part of the Kernel, where packages get submitted as part of the Network Stack. Once a Package gets submitted to the Driver, the NIC will send it. To increase the throughput of Packages, the Driver uses a first-in-first-out (FIFO) queue to store packages. Packages are added by the Driver and removed by the NIC when sending. This approach allows the whole Network Stack to operate without waiting for the NIC to be ready for the next package.

One Problem with this approach is that it introduces latency, as packages must traverse the driver queue before being sent. That is less of a problem if all Packages

are equally important, but this is not always true in real-world applications. For Example, the latency of Voice-over-IP Packages is more critical than that of TCP data stream because the quality of Voice Calls could degrade noticeably if the user has an active download in the background.

To tackle this issue, the Kernel introduces Queuing Disciplines. A Queueing Discipline (QDISC) is a module the Kernel can insert as part of the Network Stack. They are a part of the Kernel and operate in Kernel Space. QDISCs are assigned to a specific Network Interface and only affect packages that are routed to or from this interface. They act as a additional queue packages that have to traverse but with the ability to decide when and if packages can leave the queue. That allows QDISCs to alter the performance and behavior of the Network Stack significantly. For Example, they can reorder packages based on priority, limit the bandwidth or, in our use case, introduce an additional delay in the communication. In addition, multiple QDISCs can be queued together so that a package has to traverse them all. This way QDISCs can focus on their specific features without losing the functionality of other alternatives.

The Queuing Disciplines of a system can be managed via the "iproute2" [Kuz] library by using the Traffic Control "tc" command [Hub01]. With this command, it is possible to assign QDISCs to Network Interfaces, remove them, and change their configuration to alter their behavior. This command is exclusively available to system administrators.

Common Queueing Disciplines are distributed as part of the Kernel, but it is possible to add new ones using Kernel Modules. To add a new QDISC, a Kernel Module has to define a set of functions that extend those of the default QDISC operations. The Module can then register a new QDISC with this set of functions, and the Kernel will make it available. Once the QDISC is registered, it can be managed via the "tc" command [Hub01].

In more detail, QDISCs provide interfaces for the Kernel through which it engages with the QDISC. The most important are the following:

- Enqueue:
  The kernel provides the QDISC with an SKB that it should enqueue. If the QDISC has enough space and meets all possible specific requirements, it will insert the package in its internal queue. If not, it will drop the package and notify the kernel.
- Dequeue:
  The Kernel asks if the QDISC has a package that it wants to send. The QDISC will return the SKB of the package that is the next inside its queue, or if its queue is empty or if it is not jet willing to send a packet, it will return NULL

- Init:
  The Kernel has started the QDISC on a new NIC. It allows the QDISC to allocate all necessary private data for its operation
- Destroy:
  The Kernel has removed the QDISC from a NIC. THE QDISC must clean up all resources used.

In addition to its basic functionality, the QDISC reports metrics concerning its performance back to the Kernel, which system Users can access from User Space. That includes statistics over dropped packages, requeued packages, and the current queue length.

2.2 Related Work

This chapter presents related research work and tools. It explains their concepts and functionality and further explains how our work differs from them.

Two tools for emulating network delays on real hardware are `dummynet` [Riz97] and `netem` [Hem+05]. Dummy Net is a tool for FreeBSD that was designed for testing network protocols. It is able to delay packages, limit bandwidth, and enforce packet loss and mode. Dummy Net is also able to enforce these restrictions based on a filtering system that can differentiate between properties like protocol or destination. Dummy Net is used and configured via the `ipfw` tool on FreeBSD-based systems [IPF22].

While Dummy Net is a powerful tool, it does not fit our requirements as it cannot delay packets based on a stochastic distribution.

NetEm offers the same functionality as Dummy Net and also extends it. NetEm is a tool for Linux-based Systems that can delay packages, restrict bandwidth, corrupt packages, duplicate packages, and more. NetEm is also able to delay packets, following simple stochastical distributions, including native support for normal and paretro-normal distributions. NetEm consists of two parts. The first part is a QDISC that is running in the Network Stack and is responsible for manipulating the network behavior, and the second part is the `tc` command which is used to configure the QDISC. The `tc` command can be called from User Space and handles communication with the QDISC in Kernel Space. In NetEms implementation `tc` calculates the delay distributions in User Space and converts them into a table of bins. This table is then sent to the QDISC, which will apply the delay based on the bins. The fact that NetEm

uses the QDISC system makes it very useful as it can easily be combined with other QDISC to extend its functionality.

NetEm is included in the source of the Linux Kernel, which means that it comes shipped with the most modern Linux Distribution.

NetEm comes close to fulfilling our requirements and offers many more additional features. But for our use case, NetEm lacks flexibility when defining and dynamically changing the delay distributions. In contrast to the NetEms approach of pre-calculating the distributions once, we will present a design that allows for much more freedom in designing custom distributions and changing the behavior on the fly without reconfiguring the QDISC. Our tool will likewise be realized as a Linux QDISC so that it can be combined with NetEm. That allows us to focus only on delay generation and still be useful in more demanding and complex scenarios by being used in combination with NetEm.

3 System Model and Problem Statement

3.1 System Model

We aim to emulate network delay inside a connection between two or multiple machines. Our system consists of three components:

1. A Sender. It sends packages through the connection.
2. A Receiver. It receives the packages sent by the Sender.
3. A Transmitter. The Transmitter sits between the Sender and the Receiver. It applies a delay to all packages it receives and then forwards them to the Receiver.

The Sender and the Receiver can be represented as one machine or a whole network of clients. The Transmitter Stores all packages it receives from the Sender. For each package P, the Transmitter will calculate a delay d_P following a Delay Distribution D. After the delay has been achieved, the Transmitter sends the package to the Receiver.

We assume that the native transmission delays d_S, between the Sender and Transmitter, and d_R, between the Transmitter and Receiver, are constant. That ensures that only our manufactured delay d_P influences the total transmission delay of packages sent from the Sender to the Receiver.

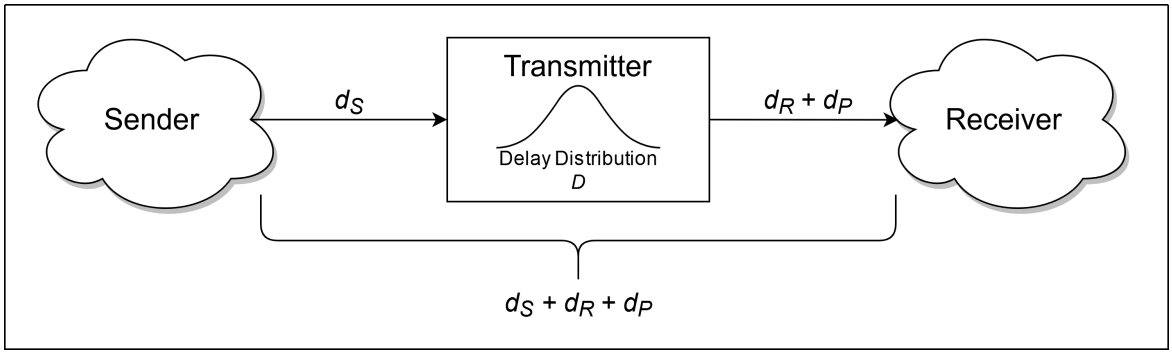


Figure 3.1: System Model

3.2 Problem Statement

This section describes the problem we aim to solve and elaborates the requirements a solution has to fulfill.

With the growing need for communication over delay-prone connections like WIFI or 5G/6G systems, there is a need for testing methods that can be used to evaluate new systems or algorithms that require these forms of communication.

While there are tools available that can emulate these networks, there is a lack of flexibility in defining network delays, for example, to emulate the unpredictable behavior of wireless connections. This work aims to design a tool that specializes in emulating network delays, which can be adapted flexibly and dynamically.

Such a tool must fulfill the following criteria:

1. It must be able to delay network packets accurately.
2. It must does not introduce additional unwanted latency into the connection.
3. It must be able to follow stochastic delay distributions.
4. It can be fine-tuned to specific use cases.
5. It has enough performance to handle modern network speeds of at least $100\,\mathrm{Mbit/s}$.
6. It must be able to work with existing tools to emulate additional behavior outside of its area of responsibility.
7. It can be deployed between any two or more members of an already existing network.

In addition to these requirements, we aim to include these optional features to make it more flexible and easier to use.

1. The tool can be deployed on any modern Linux System.
2. Multiple instances can run in the same machine on different Network Interfaces. That enables multiple connections with possible different delay distributions through one machine.
3. The tool can be configured using already present commands, like the `tc` command.

4 Approach

In this chapter we presents the design for a tool that can delay network packets following arbitrary distributions. At first, we show different approaches and assess them. Next, we present the design for a Linux QDISC that meets our requirements. We start with designing a basic QDISC that can store packets. This design is then extended step by step. In each step, we add new functionality to the design and explain why we need it and how it works.

4.1 Overview of Approaches

Our goal is to store packages of a NIC and hold them back until a defined delay has been reached. To achieve this, we need a way to efficiently intercept packets assigned to the NIC on which packets are received.

There are two types of doing this, each using a different approach.

The first type is managing and delaying the packages in User Space. One would use Kernel-by-passing techniques such as the Data Plane Development Kit (DPDK) for Linux[Sch14] in this approach. Such a platform allows for a direct connection between User Space applications and the Network Adapter. This connection allows User Space applications to act as the Network Stack and thus handle all packet manipulation. Using this, we could develop an application that collects all packages of a NIC and sends them delayed via a second NIC. An example of such a design using DPDK can be seen in Figure 4.1.

The second type is directly delaying packets in Kernel Space as part of the Network Stack. Here, this is achieved by utilizing the Linux QDISC system. As QDISCs are directly inserted into a NIC Network Stack and all packages have to traverse them, they are suited ideally for a case like this. A QDISC, in this scenario, would store all packages that are inserted into it and hold them back until the delay has been achieved. After that, it will release the packet so that it can further go down

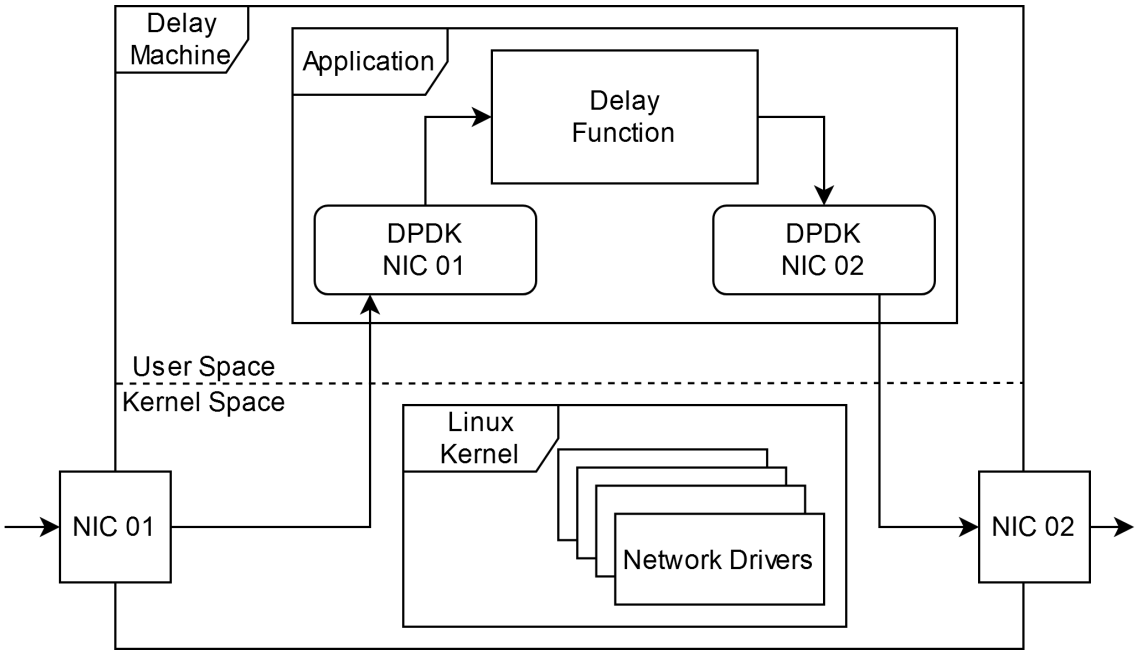


Figure 4.1: Overview of an approach using DPDK

the Network Stack. Figure 4.2 shows a basic example of using a QDISC to delay packages.

Of these two approaches, we chose to go with delaying the packets in Kernel Space. With the QDISC system, we can achieve greater flexibility as QDISCs can be chained together to add multiple behaviors. In addition because the QDISC system is a part of the Linux Kernel that is always enabled, we can expect our tool to work on any modern Linux machine out of the box.

The next choice we had was whether we should develop a new QDISC that focuses solely on delaying packages or extend or change existing QDISCs like NetEm. There are many QDISCs available that offer different services, including NetEm, which is already able to delay packets following limited stochastic distributions. While adding our functionality to tools like NetEm would be a good way of achieving our goal, we opted against it for two reasons. Firstly, by creating our own tool that only focuses on delaying, we can avoid the additional overhead that comes with other features implemented and accounted for, even if not actively used. And secondly, we follow the modular philosophy of Unix [Ray04]. We focus on doing one thing and doing it well, and if users want additional functionality, they can combine it with other QDISCs.

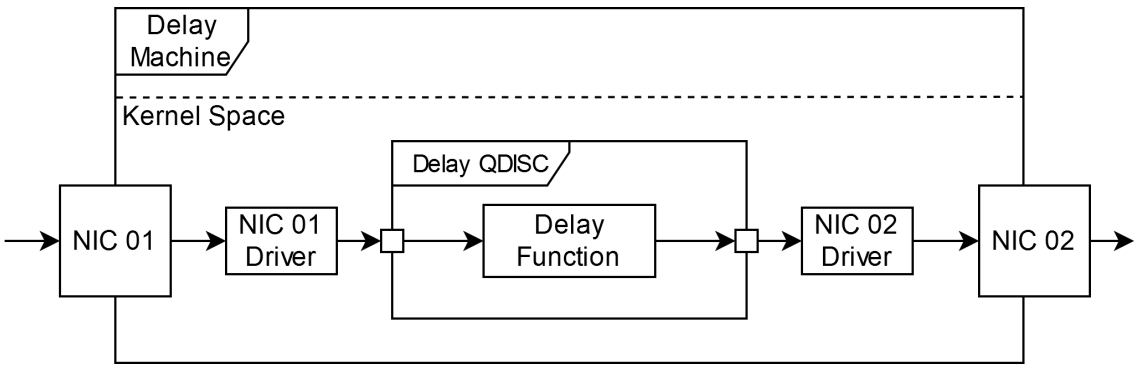


Figure 4.2: Overview of an approach using a QDISC

4.2 QDISC Design

In this section, we present the design of a QDISC that can delay packages following arbitrary stochastic distributions. We start with a design that can only store packages and extend it until it meets all our requirements, as stated in Section 3.2.

4.2.1 Core QDISC Functionality

As Section 2.1.3 explains, a QDISC is part of the Network Stack. During its operation, the Kernel interacts with the QDISC using predefined functions. The two most important are enqueue and dequeue. When packages traverse the Network Stack, they have to traverse the QDISC. The Kernel enforces this. The Kernel inserts a packet into the QDICS via the `enqueue` function. Now, the QDISC maintains and stores the packet. Later on, the Kernel will ask the QDISC if it wants to release a packet via the `dequeue` endpoint. At this point, the QDISC will release the packet by returning its SKB.

Figure 4.3 shows a simple QDISC that stores received packets in an internal queue and then releases them.

Note that the QDISC does not handle the packet data. It only interacts with the SKB, which contains references to the data and important metadata of a packet.

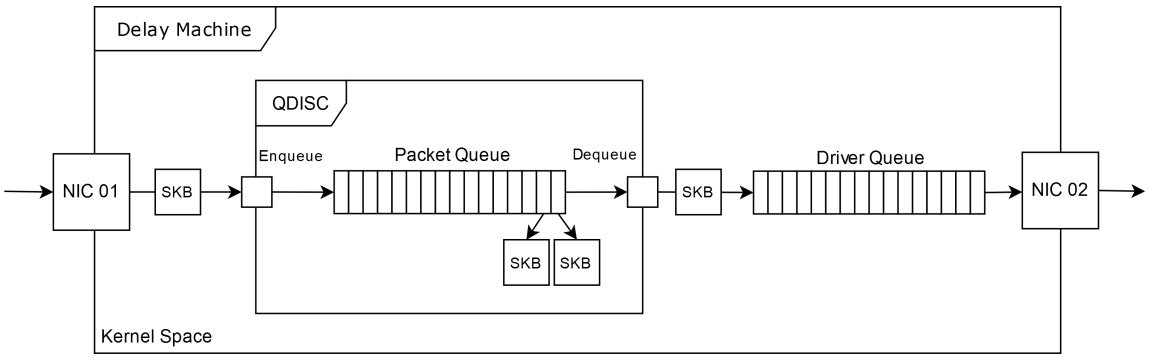


Figure 4.3: Basic design without delay

4.2.2 Delaying of packages

To create a QDISC that can hold packets back for a required time, we extend upon the core design from above. We need a way of knowing when a packet can be released.

To achieve this, we mark each packet with a timestamp that marks the earliest time the packet can be sent. This timestamp gets calculated when a packet is enqueued. The QDISC will request the current time from the Kernel and add the desired delay. The timestamp is then stored alongside ore inside the SKB. When the Kernel now asks if a packet should be sent, the QDISC checks if the current system time is greater than the earliest-send timestamp of the next packet in the queue. If this is the case, the QDISC will release the packet so that it can further traverse the Network Stack. However, if this is not the case, the QDISC will return the value NULL, which indicates to the Kernel that no packet is ready to be sent.

Figure 4.4 shows the addition of the timestamp in the setup to achieve a static delay.

This method is simple but has the problem that it is not very accurate. This problem arises because the QDISC (especially the dequeue operation) is only called from the Kernel on its behalf, typically shortly after a packet has been inserted. That means the QDISC cannot control the next time the Kernel asks if a packet should be sent, making it impossible to achieve an accurate delay. One way of solving this is to keep control until a packet is ready to be sent by busy waiting. However, this will slow down the whole system and is therefore not efficient. Instead, we overcome this issue by utilizing a watchdog timer. QDISCs can initialize a special watchdog that will signal the Kernel to call the dequeue function again. With this new addition, the QDISC will initialize this watchdog when it starts for the first time. Then, when a packet is not ready to be sent, it will register the watchdog to the exact time it is ready and can be sent. At this time, the watchdog will tell the Kernel to recheck if a packet is ready to be sent, and the QDISC will release the packet.

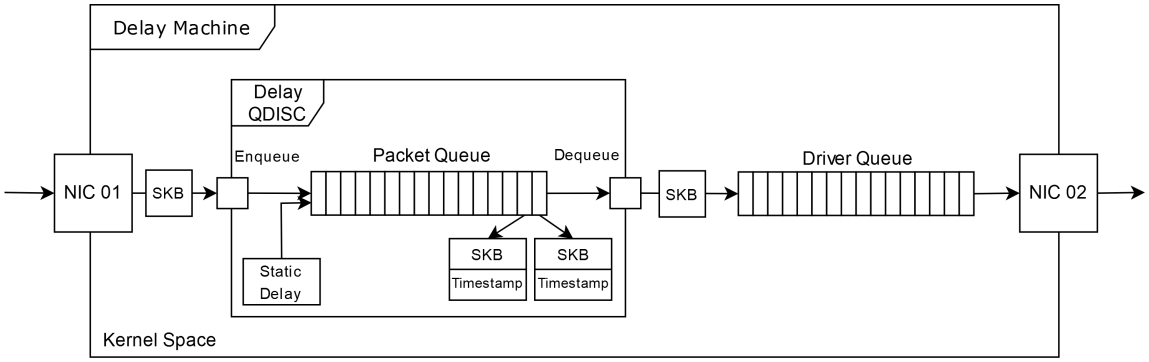


Figure 4.4: QDISC with static delay

4.2.3 Delay Generation

We need our tool to follow a defined delay distribution and delay packets accurately so that it is possible to recognize the distribution when the packets are observed from outside of our system. To accomplish that, we need to calculate a delay that follows these predefined distributions. There are two approaches for this generation: Generating the delay inside the QDISC or outsourcing it into User Space.

Delay generation inside the QDISC

This approach, which is visualized in Figure 4.5, calculates the delay inside the QDISC using a specific function that is hard coded into the Kernel Module. This function is called when a new packet is enqueued with the SKB as an argument and will calculate a corresponding delay D_p for the packet. Developers can change the function's specific distribution and additional behavior to alter the delay distribution to their needs.

This Design has the following advantages and Disadvantages:

Advantages:

- The delay-generation function has access to the SKB and all its Metadata. This is useful if we want to change the delaying distribution depending on properties like packet protocol or transmission rate.
- Generation happens inside the Kernel without switching the Context between Kernel Space and User Space, therefore causing less overhead.
- Everything is in one place, thus making the setup simpler

Disadvantages:

- Delay generation happens at the time a packet is enqueued, which can cause additional delay.
- Changing the Delay Function requires recompilation of the Kernel Module.
- Altering the Delay Function requires knowledge of Kernel Module Development and comes with the challenges associated with Kernel Space development.
- Restrictions of Kernel Space make achieving functionality more difficult to implement.

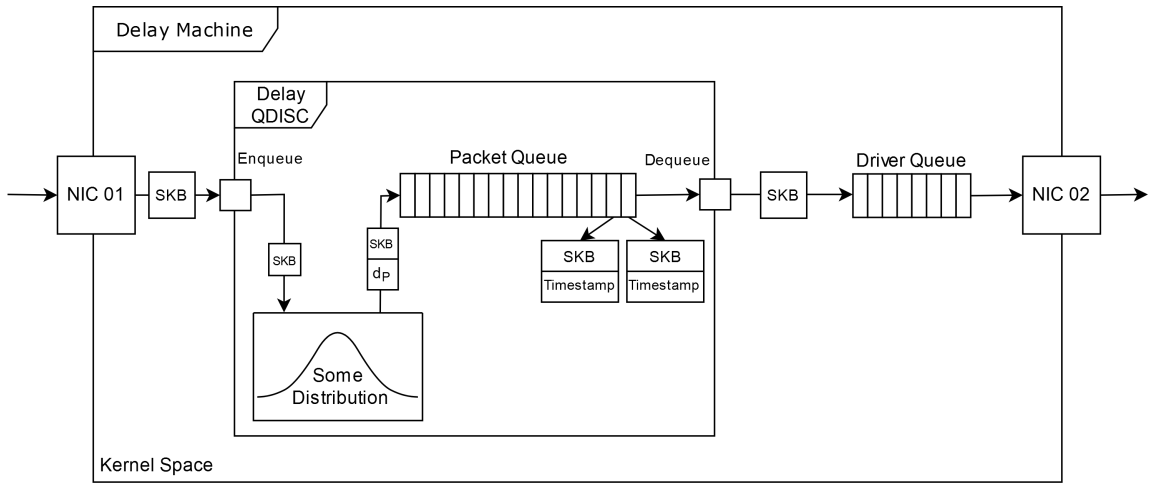


Figure 4.5: QDISC with delay calculation inside the QDISC

Delay generation inside User Space

In this approach, we outsource the generation of delays to a second application that resides in User Space. This is visible in Figure 4.6. This Application will independently calculate delays in advance based on a specific distribution and then transmit these delays to the QDISC via a specified interface. The QDISC will then add these delays to the packages on enqueuing as before. Space.

This design again has advantages and disadvantages:

Advantages:

- Delay Generation can be much more flexible, as it can use any resource it wants instead of only the Kernel-provided ones.
- Distribution calculation effort does not generate additional overhead.
- Coding does not require as much prior knowledge of deep-level Linux development.
- Developers can exchange the User Space application with their own.

Disadvantages:

- Delays can not be generated based on properties like packet protocols or arrival time, as they are not known at the generation time.
- Frequent switching between Kernel Space and User Space introduces additional overhead.

- More components make the entire system more complicated and create additional effort for deployment.

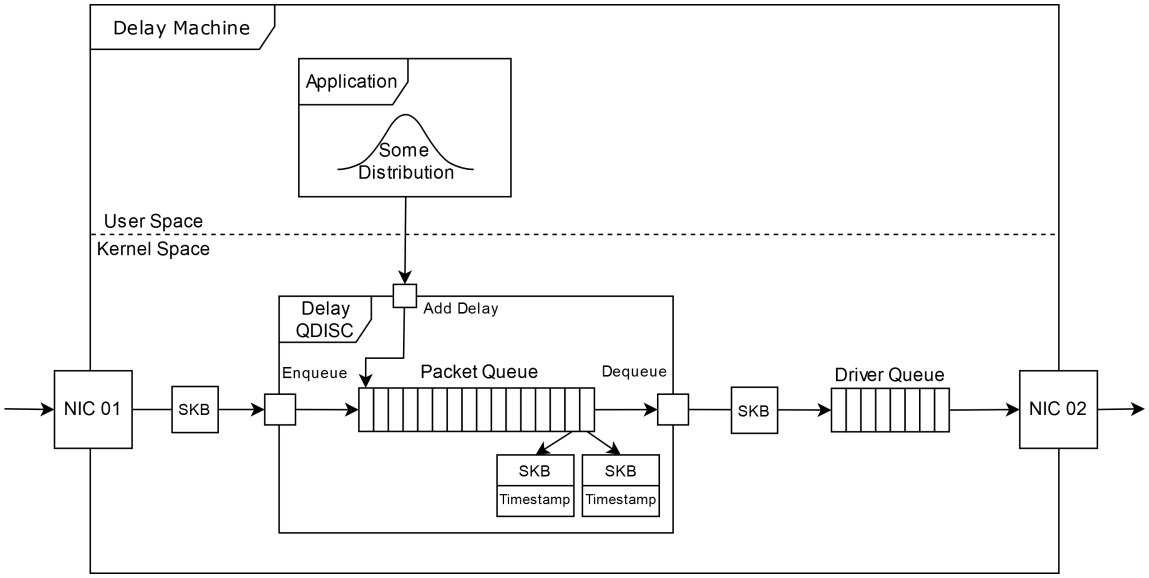


Figure 4.6: QDISC with delay calculation in User Space

Since our goal is to create a tool with support for arbitrary delay distributions, we choose the approach for User Space delay generation. This approach allows us to be much more flexible when constructing distributions and makes the whole system more user-friendly.

4.2.4 Optimeted delay generation in Userspace

We altered the design from above to create a more optimized setup to overcome the possible performance issue with frequent context switching. This optimized design can be seen in Figure 4.7. We introduce a second queue inside the QDISC that will store a large amount of pre-calculated delays for future use. Then, we change the procedure so that the User Space application can not only transmit the subsequent delay but also send an extensive list of delays that will be added to the list in one go.

With this design, we have now completely decoupled the delay generation from its enforcement. That has the benefit that the QDISC only has to retrieve the first element of the list on every enqueue, which only causes a small and consistent overhead that is not affected by the complexity of the delay calculation. One drawback of this design is that the in-advance delay generation has no insight into the QDISCs

properties or packages. Consequently, it is impossible to create delay distributions that change depending on these facts. An example of this would be a distribution that delays packets longer, the larger their size is. This limitation will be further discussed in Section 4.3.

Now, when a packet is enqueued, the QDISC will fetch the first element from the delay queue and calculate the earliest-send timestamp based on its value and the system time.

We also ensure that we only have one Application writing to the queue at any time. This guarantee allows us to use the queue without synchronization or locking, thus avoiding possible overhead.

Finally, in order to allow applications to insert new delays efficiently, we add a second endpoint that reports back the current size of the delay queue. This endpoint allows applications to wait until enough space is free before sending new delays. This procedure massively reduces the frequency of context switches and decouples the generation of delays from the enqueuing of new packages. That ensures that even very complex delay distributions do not introduce additional delay due to calculation overhead on every enqueue.

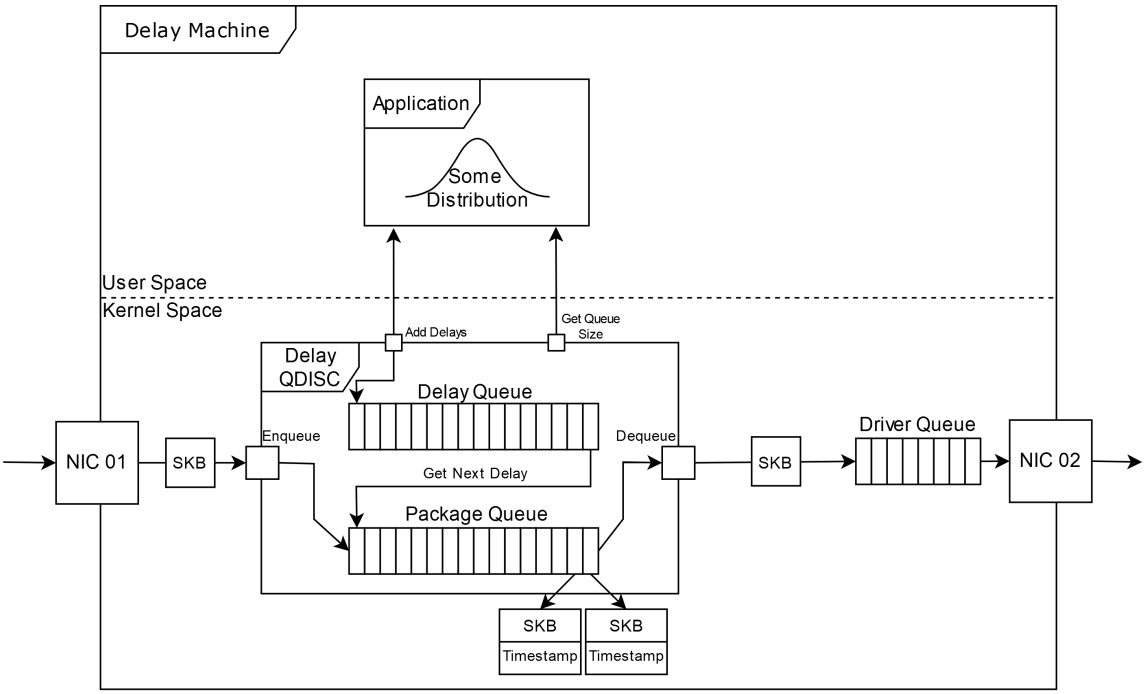


Figure 4.7: QDISC with optimized delay generation in User Space

4.2.5 Communication with User Space applications

Due to the separation between User Space and Kernel Space, we cannot simply exchange information between the delay generating application and the QDISC. That is because the two spaces are strongly separated, and communication can only happen through specific channels. To overcome this restriction, we make use of Character Devices.

When we start the QDISC on a new NIC, it creates a new Character Device corresponding to this QDISC instance. This Character Device is accessible from User Space as a file. When a process interacts with this file (e.g., by reading or writing data into it), our QDISC operates on the other side and can receive or send data through it.

Figure 4.8 shows the final QDISC design with the Character Device added.

We use this communication channel to feed new delays into the QDISC. When an application writes data into the Character Device, the QDISC receives it, converts it into new delays, and stores them for future use. Furthermore, when an application reads from the file, the QDISC reports the number of unused elements in its delay queue.

This procedure of actively checking for enough free space is not the most efficient, as it requires frequent checks and waiting (busy waiting). An alternative to this would be to use a method like `poll` that blocks the application until the QDISC notifies it when enough space is free. We choose against this method to allow applications to specify the threshold when they want to add new delays. However, while we decided against that, it remains a promising option that could be explored in future research.

The QDISC is also responsible for controlling who is allowed to interact with the file. We only want one application to interact with the Character Device at any given time. To enforce this, the QDISC keeps track of applications using the file. If there is one, it will reject all other requests to open the file. That ensures that there can only ever be one application that is feeding new delays to the QDISC.

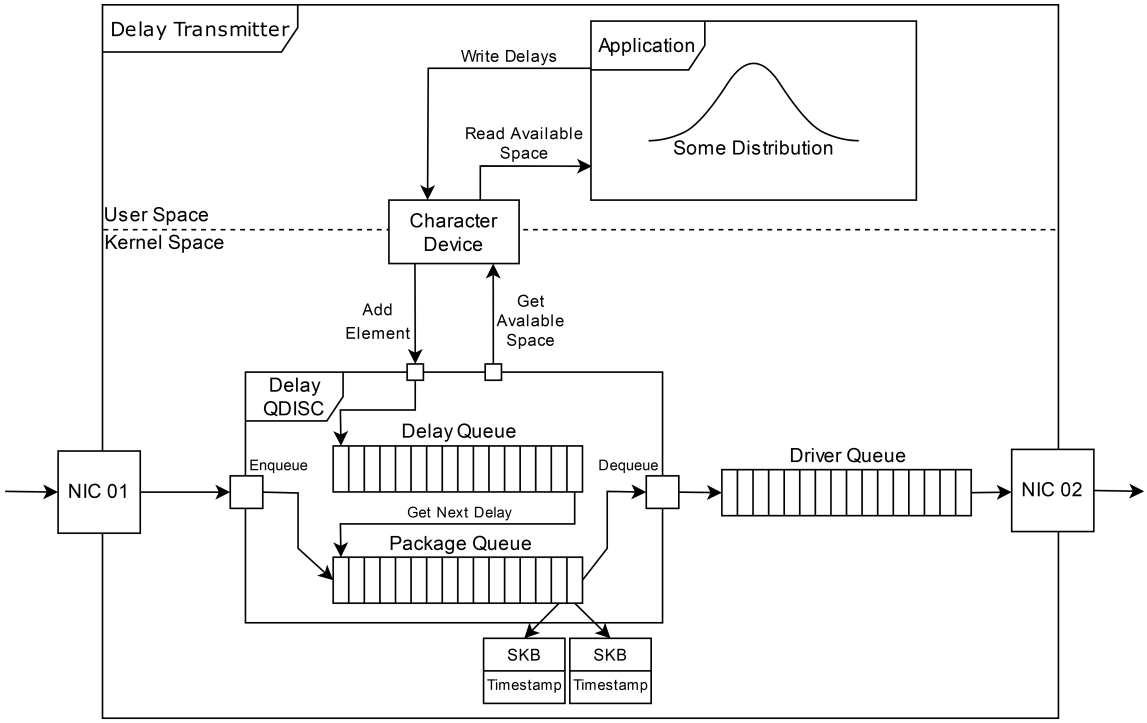
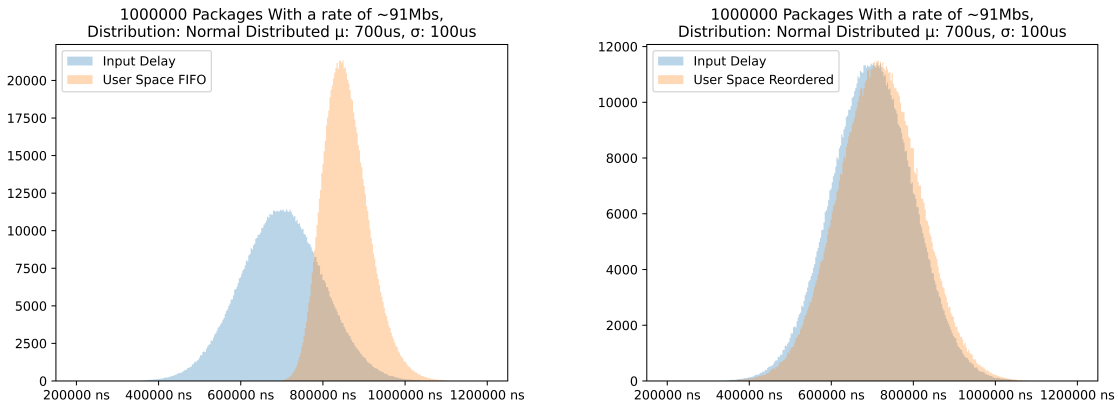


Figure 4.8: Design using a Character Device for Communication



(a) Normal Delay vs. FIFO Queue

(b) Normal Delay vs. RB-Tree

Figure 4.9: Comparison between ordered and reordered packet transmission.

4.2.6 Packet reordering

While testing the designs mentioned earlier, we noticed a significant difference between our provided delay distribution and the distribution observed on the actual packages. Figure 4.9a shows the difference between the supplied distribution and our observation. The observed distribution is offset significantly to the right and is much more pronounced in the center.

This difference occurs because we use a First-In-First-Out (FIFO) queue to store the network packages. When using a FIFO queue, in addition to their designated delay, all packages are delayed until all packages in front of them have been dequeued. That means that packets with a longer delay than the packages behind it will hold these packets with a smaller delay up even if they should have already been dequeued. This holding up of packages is the cause of the change of the distribution.

In order to achieve an accurate distribution, we have to change how we store packages in our QDISC. We need to change the FIFO packet queue to a new data structure that allows us to get the packet with the smallest earliest send timestamp efficiently. We chose a Red-Black-Tree (RB-Tree) to archive this, as the Kernel already provides an implementation of RB-Trees, with a designated pointer to the smallest entry. In addition, these RB-Trees are also optimized for system cache efficiency [Lan07]. Now packets will be inserted into the RB-Tree when enqueuing into the QDISC. On dequeue, the QDISC will retrieve the packet with the smallest earliest-send timestamp from the RB-Tree and, if ready, release it.

Using this technique to reorder the packages, we can now record the original distribution when we observe the packets sent by our machine, as shown in Figure 6.6b. The two distributions are now almost identical; there is only a tiny offset to the right that can be explained by accounting for system latency.

Because we think that the FIFO property might be desirable in select use cases, we decided not to exchange FIFO with the RB-Tree and instead offer both methods with the ability to configure which one is used.

4.3 Design Drawbacks and Limitations

While our design offers excellent flexibility and fulfills all our requirements, it has some drawbacks and limitations that we want to be transparent about and highlight.

In contrast to tools like NetEm, our QDISC is not able to delay packages on its own. It does need an Application inside User Space that is constantly creating new delays and transmits them to the QDISC. That is a drawback as it makes deployments more complicated.

It is also more complicated to configure our tool. While our tool is more flexible compared to other tools like NetEm or Dummy Net, it is also more complicated to configure the distribution. With these tools, users can simply configure them with one command, while our tool requires an Application that has to be developed to fit the user's needs. This problem is mitigated by supplying reference applications for commonly used distributions, but it still stands if users want to use a less common distribution that is not provided.

Finally, as we already mentioned in Section 4.2.4, due to the separation between delay generation and enforcement, we are not able to access packet information at the time of delay generation. That limits us in the types of delay distributions we can create. It is not possible to create distributions that delay based on packet properties. This includes important information such as packet size, arrival time, and protocol. However, while our design is not able to create such distributions on its own, it could still be used in a setup with multiple QDISCs that achieves this functionality. For example, in a setup with multiple QDISCs, our QDISC could be stationed at the beginning, delaying all packages following a general distribution, and later on, there could be another QDISC that classifies packages based on their properties and adds additional delay.

5 Implementation

In this chapter, we will present an implementation of the design we have shown in Chapter 4.

First, we will show the implementation of the QDISC´s functionality to store and delay packets.

The second section will examine the connection between the QDISC and a User Space application. We will show the implementation of a Linux Character Device that we will use as an interface to communicate with the QDISC. In addition, we will demonstrate a sample application that generates new delays following a stochastic distribution and adds them to the QDISC.

After that, we will show two optimization techniques we use to increase the QDISCs performance.

Finally, we will explain the implementation of an addition to the `tc` tool to recognize our QDISC. With this addition, users can use the tc tool to change the behavior of the QDISC.

5.1 QDISC Implementation

To demonstrate the core functionality of the QDISC, we will show how packets get delayed. That includes how delays get calculated when a packet is enqueued, how they are stored, and finally, how they are applied when dequeuing a packet. Additionally, we will explain the enqueue and dequeue interface of the QDISC for sending packets in the order they were received and for sending the packet with the shortest delay next.

5.1.1 Delaying of packages

The delaying of packages happens in three steps. First, there is the calculation of the earliest-send timestamp. Then, there is the bundling of the packet together with the timestamp. And finally, there is the holding back of packets until the delay has been achieved.

We calculate the earliest-send timestamp while a packet is enqueuing into the QDISC. The `enqueue` function can be viewed in Listing 5.1. When the QDISC receives a new SKB via the `enqueue` function, it fetches the next delay using the `get_next_delay` function. This function will retrieve the next delay from the QDISCs internal delay queue or return 0 if the queue is empty. The delay is represented by the time the packet should be delayed in nanoseconds. To then create the earliest-send timestamp, we retrieve the current system time in nanoseconds using the `ktime_get_ns()` function and add them together. We store this timestamp inside the SKB's internal data that we can access through the `delay_skb_cb` function.

```
1  static inline int delay_enqueue(struct sk_buff *skb, struct Qdisc *sch, struct sk_buff
       **to_free) {
2      struct delay_skb_cb *cb;
3      signed long long delay = get_next_delay(&qdisc_data->delay_queue);
4      // Internal Structure inside of the SKB
5      cb = delay_skb_cb(skb);
6      cb->earliest_send = ktime_get_ns() + delay;
7      ...
8      < Enquque Logic >
9      ...
10 }
```

Listing 5.1: Delay assignement on enqueue.

The enforcing of delays happens when a packet is dequeued. When the QDISCs `dequeue` function is called, we compare the next SKBs earliest-send timestamp with the current system time. If the system time exceeds the timestamp, we dequeue the packet. However, if this is not the case, we return `NULL`, which signals to the Kernel that no packet will be dequeued, and wait for the next time the function is called.

This procedure ensures packets are sent after the earliest-send timestamp but is wildly inaccurate. That is because the Kernel does not know when to ask if a packet is ready for dequeuing next time. This problem results in significantly longer delays than requested.

To ensure that packets are dequeued as soon as they are ready, we employ a QDISC watchdog, as introduced in Section 4.2.2. This watchdog is an entity

outside our QDISC that will notify the Kernel once a packet is ready for dequeuing. We create a new watchdog when a new instance of the QDISC is started using the `qdisc_watchdog_init` function. To then use the watchdog, we utilize the `qdisc_watchdog_schedule_ns` function. With this function, we supply the watchdog with a timestamp in nanoseconds. This timestamp is then internally passed to a Kernel high-resolution timer [Tho]. This timer is configured to have QDISCs `dequeue` function as its callback function. When the system time reaches the supplied timestam the timer triggers and `dequeue` is called.

```
1  static inline struct sk_buff *delay_dequeue(struct Qdisc *sch) {
2      struct delay_qdisc_data *qdisc_data = qdisc_priv(sch);
3      struct sk_buff *skb; struct delay_skb_cb *cb; u64 now;
4
5      skb = < get next SKB >
6      cb = delay_skb_cb(skb);
7      now = ktime_get_ns();
8
9      if( cb->earliest_send < now ) {
10         < send packet >
11     }
12     qdisc_watchdog_schedule_ns(&qdisc_data->watchdog, cb->earliest_send);
13     return NULL;
14 }
```

Listing 5.2: Delay enforcement on dequeue.

5.1.2 Enqueue and Dequeue with Reordering

We want to send the packages in an orderly sequence sorted by the smallest earliest-send timestamp. We achieve this using a Red-Black-Tree instead of a FIFO queue to store our packages. The Kernel already offers an efficient implementation of RB-Trees [Lan07]. This implementation includes functions that handle the core functionality of RB-Trees, including the logic for rebalancing the tree. However, developers have to implement the logic for inserting new nodes themselves. We make use of the fact that the SKB data structure already includes the `rb_node` struct that is used to order packages inside RB-Trees.

Enqueue

The first thing we do when `enqueue` is called is to check whether we can accept a new packet. We do this by checking if the number of packages inside the QDISC is

smaller than the configured limit. If this is not the case, we drop the packet using the provided `qdisc_drop` function and return the value `NET_XMIT_DROP` to signal the Kernel that a packet was dropped. However, if we have space available, we will continue the enqueueing process.

We must insert the new SKB into the RB-Tree during the enqueueing process. Listing 5.3 shows the enqueue procedure with an RB-Tree. To insert the next SKB, we must figure out at what position it should be in the tree or, more precisely, which node its parent is. To determine that, we traverse the tree and compare the earliest-send timestamps until we arrive at the correct position. Once we have found the correct position, we link the SKB to its parent, using the provided `rb_link_node` function, and finally insert it into the tree using the `rb_insert_color` function. During the insertion, the `rb_insert_color` function does balance the tree, so we do not have to be concerned with that.

At the end of the enqueueing process, we return the value `NET_XMIT_SUCCESS`, which signals to the Kernal that the packet was successfully enqueued.

```
1  static inline int delay_enqueue_reorder(struct sk_buff *skb, struct Qdisc *sch, struct
       sk_buff **to_free) {
2      if(likely(sch->q.qlen < sch->limit)) {
3          struct rb_node **node; struct rb_node *parent;
4              < ... >
5          node = &qdisc_data->package_queue_root.rb_node;
6          parent = NULL;
7          while (*node) {
8              struct sk_buff *skb2;
9              parent = *node;
10             skb2 = rb_to_skb(parent);
11             if(delay>=delay_skb_cb(skb2)->earliest_send) {
12                 node = &parent->rb_right;
13             } else {
14                 node = &parent->rb_left;
15             }
16         }
17         rb_link_node(&skb->rbnode, parent, node);
18         rb_insert_color(&skb->rbnode, &qdisc_data->package_queue_root);
19         < ... >
20         return NET_XMIT_SUCCESS;
21     } else {
22         printk(KERN_INFO "Package Queue ist Full! Dropping Package\n");
23         qdisc_drop(skb, sch, to_free);
24         return NET_XMIT_DROP;
25     }
26 }
27 }
```

Listing 5.3: QDISC enqueue with RB-Tree.

Dequeue

Listing 5.4 shows the `dequeue` function that uses an RB-Tree. When `dequeue` is called, we need to fetch the packet with the smallest earliest-send timestamp from the RB-Tree. Inside the RB-Tree, this packet is represented by the leaf to the far left. It is a common goal to access the smallest entry of an RB tree, so the `skb_rb_first` function is provided to access it directly. After we receive the packet and if it is ready to be dequeued, we erase it from the RB-Tree using the provided `rb_erase` function. We also have to restore the SKBs dev property that gets overwritten when an SKB gets inserted into the RB-Tree. This property specifies the network interface to which the package belongs and is essential for the operation of the Network Stack. To restore it, we retrieve the network interface on which the QDISC operates, using the provided `qdisc_dev` function, and set the information inside the SKB.

Finally, we release the packet by returning the SKB.

```
1  static inline struct sk_buff *delay_dequeue_reorder(struct Qdisc *sch) {
2      < ... >
3      struct sk_buff *skb;
4      skb = skb_rb_first(&qdisc_data->package_queue_root);
5      // Check if RB-Tree was empty
6      if(unlikely(skb != NULL)){
7          struct delay_skb_cb *cb;
8          u64 now;
9          cb = delay_skb_cb(skb);
10         now = ktime_get_ns();
11         if( cb->earliest_send < now) {
12             sch->q.qlen--;
13             rb_erase(&skb->rbnode, &qdisc_data->package_queue_root);
14             skb->dev = qdisc_dev(sch);
15             skb->next = NULL;
16             skb->prev = NULL;
17             return skb;
18         }
19     }
20     < ... >
21     return NULL;
22 }
```

Listing 5.4: QDISC dequeue with RB-Tree.

5.1.3 Enqueue and Dequeue in FIFO order

As stated in Section 4.2.6, we still want to offer the option to turn off reordering and instead have packets traverse the QDISC in the order they were enqueued.

To manage packets in FIFO order, we can rely on the Kernel's already-provided implementations. The Kernel provides a reference implementation of the QDISC functions that uses an internal FIFO Queue. This espacially includes the `qdisc_enqueue_tail` and `qdisc_dequeue_head` functions.

Enqueue

Likewise to Section 5.1.2, when we enqueue a new packet into the QDISC in FIFO configuration, we first have to check if we can accept a new packet. After this step, we add the packet into the FIFO queue. We do this by using the mentioned `qdisc_enqueue_tail` function. This function takes the SKB and adds it to the end of a linked list. This linked list belongs to this instance of the QDISC and is stored as a pointer inside its metadata. When the packet is added successfully, the `qdisc_enqueue_tail` function returns `NET_XMIT_SUCCESS`, which we return ourselves to indicate the successful enqueue to the Kernel.

```
1  static inline int delay_enqueue_fifo(struct sk_buff *skb, struct Qdisc *sch, struct
       sk_buff **to_free) {
2      struct delay_qdisc_data *qdisc_data = qdisc_priv(sch);
3      if(likely(sch->q.qlen < sch->limit)) {
4          < ... >
5          return qdisc_enqueue_tail(skb, sch);
6      } else {
7          printk(KERN_INFO "Package Queue ist Full! Dropping Package\n");
8          qdisc_drop(skb, sch, to_free);
9          return NET_XMIT_DROP;
10     }
11 }
```

Listing 5.5: QDISC enqueue with FIFO order.

Dequeue

When dequeue is called, we have to check if the next packet is ready to be dequeued. For this, we have to access the packet without removing it from the queue in case it is not yet ready. We do this by using the provided `qdisc_peek_head` function that returns the first element of the linked list without removing it. Then, if the packet is ready,

we need to remove it from the queue, so we call the `qdisc_enqueue_tail` function that again returns the first SKB and also removes it from the queue.

Finally, we again release the packet by returning the SKB.

```
1  static inline struct sk_buff *delay_dequeue_fifo(struct Qdisc *sch) {
2      struct delay_qdisc_data *qdisc_data = qdisc_priv(sch);
3      if(likely(!sch->q.qlen == 0)) {
4          struct sk_buff *skb;
5          skb = qdisc_peek_head(sch);
6          < ... >
7          if( <delay has been achieved> ) {
8              return qdisc_dequeue_head(sch);
9          }
10     }
11   return NULL;
12 }
```

Listing 5.6: QDISC dequeue with FIFO order.

5.2 Character Device Implementation

We use a Caracter Device to communicate with the User Space and add new Delays to our QDISC. We will show how the Caracter Device is created, how applications can interface with it, and offer an example of how a Python script can interact with it.

5.2.1 Character Device creation

Each Character Device is represented by a major and minor number that identifies the device. To create a new Character device, we first have to request a new set of major and minor numbers under which we can register the device. We do this using the `alloc_chrdev_region` function that assigns us a new set of major and minor numbers.

We now have access to a new Character Device, but this device is not yet exposed to User Space. To make the device accessible, we use the `device_create` function to create a file through which users can interact. These files are typically placed inside the `/dev` directory on Linux systems. In our case, we create the device file at `/dev/sch_delay/<interface_name>` where the interface name is the name of the interface on which the QDISC is deployed.

Finally, we assign the Character Device a set of functions that get called when interacting with the device. This happens through the `cdev_init` function.

5.2.2 Character Device Interfaces

Character devices can offer numerous interfaces, but we only require the interfaces `open`, `read`, `write` and `release`. This section will explain these interfaces and show how we use them.

Open

The function assigned to open is called whenever a new process tries to open the Character Device. In this function, viewable in Listing 5.7, we perform three essential actions.

Firstly, we control the access to the file. We only allow one process to interact with the file at any moment. To do this, we track if the file was opened but not yet closed. If this is the case, we know that another process is interacting with it. To then deny access to the file, we return the value `-EBUSY`, signaling that the file is currently in use.

Secondly, we access the internal data of the QDISC and store it inside the private data of the file object. This is necessary so that the `read` and `write` functions know to which instance of the QDISC they belong.

Finally, we acquired a lock on our Kernel Module using the `try_module_get` function. This lock indicates that a process is interacting with the module. That is essential information for the Kernel because it has to ensure that a Kernel Module is not removed while a process is interacting with it.

```
1  int sch_delay_device_open(struct inode *inode, struct file *file) {
2    struct delay_qdisc_data *qdisc_data = container_of(inode->i_cdev,
3              struct delay_qdisc_data, c_dev);
4      file->private_data = qdisc_data;
5      if (qdisc_data->chr_dev_open_count) {
6          return -EBUSY;
7      }
8      qdisc_data->chr_dev_open_count++;
9      try_module_get(THIS_MODULE);
10     return 0;
11 }
```

Listing 5.7: Character Device open operation

Release

The release function is when a process closes the Character Devices file. In this function, viewable in Listing 5.8, we clean up allocated resources and prepare for the next time the file is opened.

We set the private data of the file to `NULL` to avoid possible confusion in the future and mark the file as closed so that the next process can open it. Finally, we release the lock on the Kernel Module so that the Kernel knows that nothing is interacting with it and can safely be removed if so desired.

```
1 int sch_delay_device_release(struct inode *inode, struct file *file) {
2   struct delay_qdisc_data *qdisc_data = container_of(inode->i_cdev,
3             struct delay_qdisc_data, c_dev);
4     file->private_data = NULL;
5     qdisc_data->chr_dev_open_count--;
6     module_put(THIS_MODULE);
7     return 0;
8 }
```

Listing 5.8: Character Device release operation

Write

The task of our Character Device is to allow processes to add delays to the QDISC. We achieve this using the `write` interface. When a process writes data to the Character Device, we can access this data inside the Character Device in Kernel Space. To use this to add new delays, we interpret all data we receive as a list of long longs, which we then store in an internal FIFO queue. These unsigned longs represent the time in nanoseconds for how long a packet should be delayed. These delays are later used to create the earliest-send timestamps.

Because the buffer in which our data is stored is located in User Space, we can not access it directly. Instead, we use the `kfifo_from_user` function, which the Kernel provides. This function allows us to read a section of the buffer and then store it in a FIFO queue that we can access.

```
1 ssize_t sch_delay_device_write(struct file *file, const char *buffer, size_t len,
      loff_t *offset) {
2     struct delay_qdisc_data *qdisc_data = file->private_data;
3     int i;
4     int ret;
5   unsigned int copied;
6
```

```
7     for (i = 0; i < len; i=i+8) {
8         ret = kfifo_from_user(&qdisc_data->delay_queue, buffer+i, 8, &copied);
9     }
10    return len;
11 }
```

Listing 5.9: Character Device write operation

Read

As mentioned in chapter 4.2.4, we only want to add new delays when enough space is available to avoid unnecessary overhead. We would rather do fewer transfers with many delays instead of many transfers with few delays. To allow for this, we report the number of unused elements inside the delay queue when a process reads from the Character Device. Inside the `read` function, we retrieve the number of free elements via the `kfifo_avail` method. We then manually write this number encoded as an unsigned long into the buffer from which the process reads. Similar to the `write` operation, we must use a special function to interact with the buffer because it lies in User Space. We use the `put_user` function that writes data to a specific position in the buffer.

```
1 ssize_t sch_delay_device_read(struct file *file, char *buffer, size_t len, loff_t *
      offset) {
2     struct delay_qdisc_data *qdisc_data = file->private_data;
3     unsigned long data = kfifo_avail(&qdisc_data->delay_queue)/8;
4
5     put_user((data >> (8*0)) & 0xff, buffer++);
6     put_user((data >> (8*1)) & 0xff, buffer++);
7     put_user((data >> (8*2)) & 0xff, buffer++);
8     put_user((data >> (8*3)) & 0xff, buffer++);
9     *offset += 4;
10    return 4;
11 }
```

Listing 5.10: Character Device read operation

5.2.3 Example Application

In the following, we will show a minimal Python application that generates new delays following a normal distribution. It will communicate with the QDISC through the Character Device and insert new delays periodically.

The application consists of two components.

Firstly, we have an endless loop, viewable is Listing 5.11, that will request the number of free elements inside the QDISCs delay queue. This loop will check periodically if the free space inside the QDISCs delay queue exceeds a defined threshold. Once this threshold is reached, the application generates new delays and writes them to the Character Device.

```
1  dev = os.open("/dev/sch_delay/<interface_name>", os.O_RDWR)
2  while True:
3      free = int.from_bytes(os.read(dev,8), "little")
4      if free <= MIN_DATA_SIZE:
5          print("Queue is to full, sleeping...")
6          time.sleep(SLEEP_INTERVAL)
7          continue
8      else:
9          print("Writing %s Entries" % free)
10         os.write(dev, generate_data(free))
11         time.sleep(SLEEP_INTERVAL)
```

Listing 5.11: Example Application Main Loop

The second component is a function that generates new delays, viewable in Listing 5.12. We generate a list of delays following a distribution created by the `numpy` library. In this example, we use a normal distribution with a mean value of $700\,\mu s$ and a standard deviation of $100\,\mu s$. Each element of this list is then encoded as an 8-byte number in little-endian representation so that the QDISC can read it as a `long long`. Finally, we concatenate all these numbers into one large byte array that is then written to the Character Device.

```
1  def generate_data(count):
2      delays = numpy.random.normal(100_000, 700_000, count).tolist()
3      data = bytearray()
4      for x in delays:
5          data.extend(abs(int(x)).to_bytes(8,"little"))
6      return data
```

Listing 5.12: Example Application Delay Generation

5.3 Performance Optimizations

To increase the performance of the QDISC and reduce possible overheads, we make use of two optimization techniques. These techniques are hints for the CPU branch prediction and inline function definition.

When developing for the Kernel, developers can include hints for the compiler that indicate which outcome of a branch is the most likely. The compiler then uses this hint to arrange the resulting machine code so that the likely branch will be the one that the CPU branch prediction follows. Developers can indicate this to the compiler by encapsulating the branch condition inside a `likely()` or `unlikely()` statement [Ker17]. While this technique is useful, it must be used carefully as a wrong hint will introduce more overhead than no hint.

```
1 __always_inline signed long long get_next_delay(struct kfifo *delay_queue) {
2     if(likely(!kfifo_is_empty(delay_queue))){
3         unsigned char data_buffer[8];
4         signed long long delay;
5         kfifo_out(delay_queue, data_buffer, 8);
6         delay = *(signed long long *) data_buffer;
7         return delay;
8     }
9     return 0;
10 }
```

Listing 5.13: Next Delay Function with Overhead Optimization.

Using this technique, we can avoid the overhead of necessary checks for edge cases that must be accounted for but should not occur during regular operation. One example of this can be seen in Listing 5.13. Here, each time we want to fetch the subsequent delay from the internal delay queue, we have to check whether the queue has any elements left. On its own, this would introduce additional overhead on each packet enqueue, but as we always expect the queue to be filled during the QDISCs operation, we mark this branch as the likely one.

The use of inline functions has a similar effect. When a function is annotated as `inline`, it tells the compiler that, in instances where the function is called, it should not make a function call and instead paste the content of the function at this point [kera]. This allows developers to outsource specific functionality into their own functions, thus making the codebase more manageable while not impacting performance through function call overhead. Because the inline annotation only creates a hint for the compiler, it is not guaranteed that there is a function call anyway. If we want to strictly enforce the inline functionality, we have to instead use `__always_inline` annotation. In the same Figure 5.13, you can see that we declared this function to always be inline so that we can use it when a packet is enqueued without additional overhead. While this method is very convenient, it should not be used too widely as it can lead to negative implications, as discussed by Linux Torvalds here [Tor03].

5.4 Configuration Through tc

To apply the QDISC to a NIC and to change its configuration, we use the `tc` command. This command is part of the `iproute2` package on Linux Distributions and offers an interface to manipulate the machine's network traffic behavior [Kuz]. We can already use this tool to add our QDISC to a NIC's Network Stack using a command similar to 5.14.

```
1 $ tc qdisc add dev <interface_name> root delay
```

Listing 5.14: Starting a new QDISC Instance with tc

In this example, the QDISC `delay` is added to the root of a Network Devices outgoing Network Stack so that a packet goes through the QDISC before the NIC sends it.

However, if we want to change the QDISC's configuration, we first need to extend the source code of `tc` so that it can identify our QDISC and knows which arguments are allowed and how to parse them. To do that, we append the source code of tc with a module corresponding to our QDISC. Listing 5.15 shows the added function that is called when our QDISC is addressed. The `tc` command recognizes the `delay` keyword in its arguments and will hand all following arguments to our function. In this function, we then parse the arguments for our QDISC and construct a `tc_delay_qopt` structure that contains the new configuration. After all arguments are successfully parsed, the function passes the options to the `addattr_l` function, which sends them to the QDISC. This communication occurs through the Netlink socket interface [KKKS03].

```
1  static int delay_parse_opt(struct qdisc_util *qu, int argc, char **argv,
2          struct nlmsghdr *n, const char *dev)
3  {
4    int ok = 0;
5    struct tc_delay_qopt opt = {};
6    while (argc > 0) {
7      if (strcmp(*argv, "limit") == 0) {
8        NEXT_ARG();
9        if (get_size(&opt.limit, *argv)) {
10         fprintf(stderr, "%s: Illegal value for \"limit\": \"%s\"\n", qu->id, *argv);
11         return -1;
12       }
13       ok++;
14     }
15     else if (strcmp(*argv, "reorder") == 0) {
16       ...
17       < additional arguments >
18       ...
```

```
19      }
20      argc--; argv++;
21    }
22    if (ok) {
23      addattr_l(n, 1024, TCA_OPTIONS, &opt, sizeof(opt));
24    }
25    return 0;
26  }
```

Listing 5.15: TC Parse Function

The Kernel then receives the message containing the new configuration and checks to which QDISC instance it belongs. The Kernel then calls the `change` function of the corresponding QDISC with the new configuration as an argument. Inside this method, the QDISC can evaluate the new configuration and then, if necessary, change its settings and behavior.

```
1 $ tc qdisc change dev <interface_name> root delay limit 1500 reorder true
```

Listing 5.16: Configuring a Running QDISC with tc

Listing 5.16 shows an example where an already running instance of our QDISC is configured to allow a maximum of 1500 packets in its internal queue and enable package reordering.

6 Evaluation

In this chapter, we will test and evaluate our tool. First, we will show a testing setup that enables us to precisely measure the delay our tool introduces into a connection between systems. Using this setup, we will test our tool and evaluate its performance in handling packet throughput, accuracy in packet delay, and consistency. Finally, we will summarize our findings.

6.1 Testing Setup and Procedure

To evaluate the performance and accuracy of our tool, we created a testing setup that enables us to monitor and capture the delay distribution injected into a network connection. Our testing setup consists of 3 Servers. Two of these servers communicate using a wired connection over fiber, and the third server monitors the connection and observes the delay.

The first server, which we will call the Sender or Receiver, has two NICs that are directly connected to the second server, called the Delayer. The Sender sends messages to the Receiver. These packets are then routed through the Delayer, who introduces the delay. We execute the Sender and Receiver on the same physical machine, separating them through Network Namespaces [Bie13]. We need this separation to ensure the messages leave the system and traverse the Delayer.

The Delayer also has two NICs configured so that all packets received from the Sender on one NIC will be routed back to the Receiver through the second NIC. Our QDISC operates on the second NIC's egress port, so all packets leaving are delayed.

The final server, which we will call the Observer, taps both fiber connections between the Sender and Delayer and the Delayer and Receiver. It does capture all transmitted packages. It is equipped with a `NT40A01_4X1` network capture card from Napatech. This capture card can record and timestamp up to 4 data streams with nanosecond precision [20].

With the capture data, we can calculate the delay of packets by comparing their timestamp and calculating the time it took for them to be received by the Delayer and then forwarded to the Receiver.

While testing, we send UDP packages from the Sender to the Receiver, as already explained. Each packet is equipped with a unique ID so that we can identify corresponding packets when comparing the recorded data. We also include a second consistent ID that enables us to differentiate our packages from others that were sent during our tests. With all these additions taken into account, we achieve a packet rate of around $130,000$ Packages per $100\,\mathrm{MB/s}$ of bandwidth.

Figure 6.1 shows this described setup.

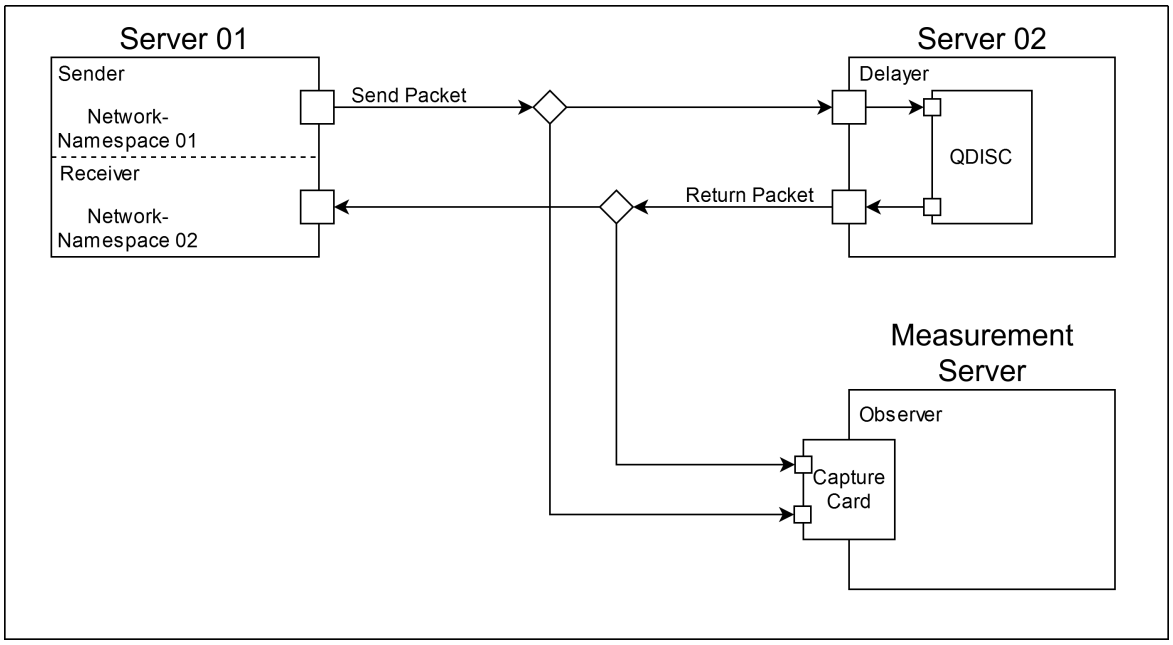


Figure 6.1: Testing / Evaluation Setup

In our testing environment, the Sender/Receiver and Delayer are realized by two identical Servers, each equipped with an Intel Xeon E5-1650 CPU. The Observer runs on a more powerful Server equipped with an Intel Xeon E5-2687W CPU. This additional computational power is not required for recording the packages, but it makes the post-processing of the measured data faster.

6.2 Evaluation of Overhead

We Stated in our requirements in Section 3.2 that our tool should introduce the least possible amount of overhead, i.e., introducing additional delay while processing packets.

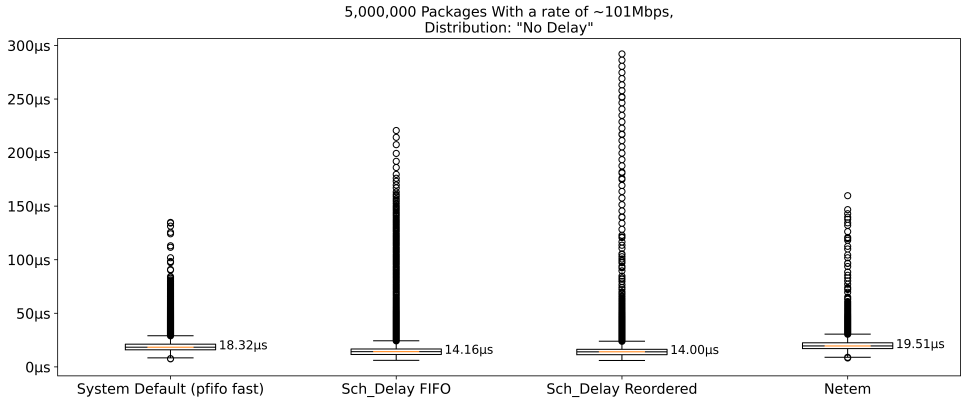


Figure 6.2: Processing Overhead across different QDISCs

To evaluate the overhead that is caused by our QDISC processing packets, we will compare the performance of the connection when it uses the default network setup versus when it uses our QDISC without applying a delay. We include our QDISC in both configurations, once with reordering enabled and once with FIFO order. That compares the compute overhead between the built-in QDISC FIFO queue and the RB-Tree used for reordering. For the system default, we use the `pfifo_fast` QDISC that is defined as the standard QDISC in the Linux Kernel source code and should therefore be the default on most Linux Distributions. Finally, we also included the NetEm QDISC in our comparison as it is the current go-to standard for emulating packet delay on Linux Systems. This test was performed at around a bandwidth of $100\,\mathrm{MB/s}$, corresponding to around $130,000$ Packages per second. Figure 6.2 shows the measurements and their comparison.

From this test, we can make three observations.

The first observation is that we introduce less latency and, therefore, less processing overhead into the connection than the other QDISCs. That is because we only focus on delaying packets and nothing else. Both `pfifo_fast` and `netem` offer additional functionality that introduces additional overhead even if not applied.

The second observation is that both configurations of our tool produce more outliers than `pfifo_fast` and `netem`. This behavior can be traced back to the performance

optimizations we described in Section 5.3. There, we noted the danger of hinting at the wrong branch that will proceed. That is exactly what happens here. Inside the `get_next_delay` function, we check if there are elements inside the delay queue. As we expect this queue to always be filled during the QDISCs operation, we predict this statement to be true. However, during this test, the delay queue is always empty. That causes us to always go down the not-predicted path, thus creating additional overhead.

Finally, we can observe that the overhead of all tools varies by about $5\,\mu\mathrm{s}$ in both directions from the median. This observation indicates that this variance is caused by general Network Stack processing overhead and is outside our control.

By examining these results and considering their causes, we can conclude that our QDISC does not contain significant processing overhead and does not introduce significant unwanted latency into the connection.

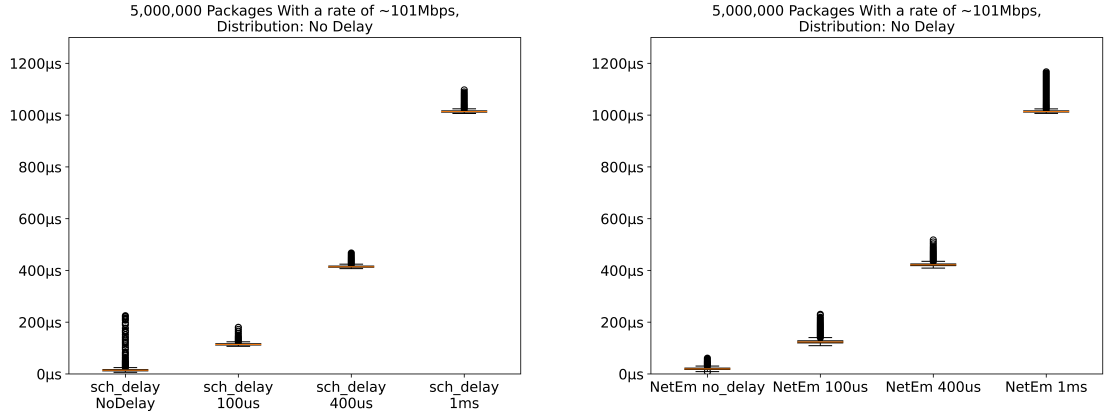
6.3 Evaluation of Delay Accuracy

Next, we want to test how accurate our tool can delay packets. We split this test into two parts. First, we check how accurately our tool can delay packets with a static delay. This test will tell how precise our tool can enforce delays and will also observe changes in overhead when we delay packets. After that, we will test how accurately our tool can follow stochastic delay distributions. This test will observer how accurately the enforced distribution resembles the configured one.

6.3.1 Delay Accuracy with Static Delay

To test how accurately we can enforce static delays, we configured both tools to delay packets by a static amount. We then measured how long it took for these packets to traverse the system and compared the result to the configured delay. We repeated this test across four configurations, once with no delay as a baseline, then $100\,\mu\mathrm{s}$, $400\,\mu\mathrm{s}$ and $1\,\mathrm{ms}$. This test was performed at around a bandwidth of $100\,\mathrm{MB/s}$, corresponding to around $130,000$ Packages per second. Since we concluded in Section 6.2 that there is no significant difference in overhead between forbidden reordering (enforced FIFO) and allowed reordering, we chose to include only our tool in enforced FIFO configuration, as there is no need for reordering with static delays.

Figure 6.3 shows the measurement for multiple static delay configurations for both our QDISC and NetEm. The measured values are listed in Table 6.1.



(a) Delay Accuracy Sch Delay

(b) Delay Accuracy NetEm

Figure 6.3: Comparison of Static Delay Accuracy between Sch Delay and NetEm

	Median		**Mean**		**Lower Quartile**		**Upper Quartile**	
	`sch_delay`	`netem`	`sch_delay`	`netem`	`sch_delay`	`netem`	`sch_delay`	`netem`
No Delay	14.15	19.46	14.60	19.77	11.68	16.99	16.80	22.39
100 μs	114.22	123.48	114.77	124.28	111.94	119.59	117.06	127.88
400 μs	414.80	421.44	416.39	421.94	412.34	418.31	418.15	425.03
1 ms	1014.37	1013.55	1017.09	1014.02	1011.75	1011.37	1017.94	1016.41

*All values are given in microseconds (μs).

Table 6.1: Static Delay Value Comparison between Sch Delay and NetEm

By looking at the measured values, we can see that our tool is able to introduce the wanted delay with a high accuracy. When we look at the median of the measured delay, we can see that it is at almost the exact value of the configured delay, only offset by the overhead of around $14\,\mu$s we measured in Section 6.2. The same applies to the lower and upper quartiles. Only in the mean can we observe a slight increase in delay, which can be explained by the fact that larger delays mean more packets are stored inside the QDSIC. That can cause a small amount of additional overhead when a packet gets inserted or removed from the queue. This overhead was not visible in Section 6.2 because packets did leave the QDISC imminently after enqueue, as they weren't delayed. Nevertheless, these results show that our QDISC is able to delay packages with an accuracy of about $2\,\mu$s. In addition to that, we can observe that the delayed measurements maintain the same consistency that we measured in Section 6.2 of around $5\,\mu$s around the median.

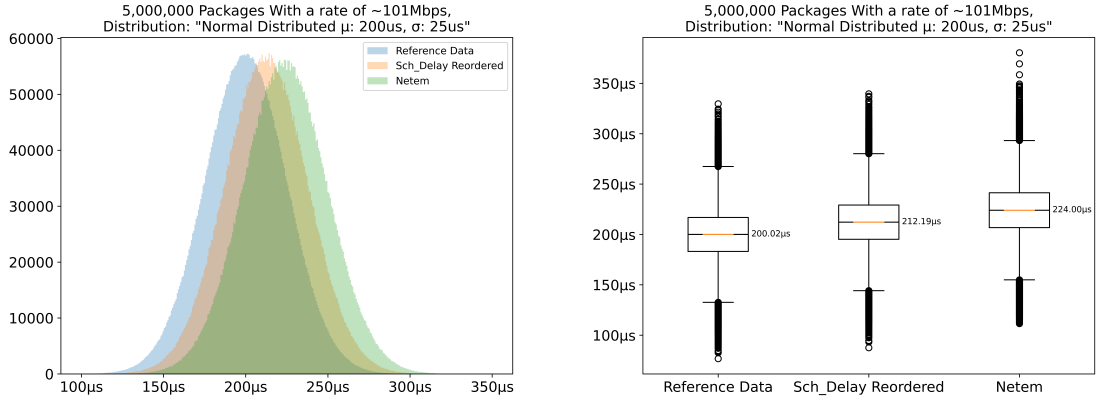


Figure 6.4: Distribution Accuracy between Sch Delay and NetEm

We can also observe that our QDISC can add delays across all measurements more accurately than NetEm. While we achieve the above mentioned accuracy of around $2\,\mu s$, NetEm is further away with one of about $5\,\mu s$. Note that this accuracy is the offset from the baseline measurement with no applied delay. That means that NetEm introduces additional overhead on top of the already measured one in Section 6.2, due to the extra processing necessary for introducing the delay.

Finally, we can observe that the outliers we observed when our tool was not applying a delay are not present when it is delaying packages. This observation reinforces our statement from Section 6.2 that these overheads are caused by incorrect branch predictions that backfire when no delays are handed to the QDISC.

6.4 Delay Accuracy with Distributed Delay

Now that we know that our QDISC can delay packets with an accuracy of $2\,\mu s$, we want to test if and how accurately it is able to delay packets following a stochastic delay distribution. In the testing setup, we now use our tool and NetEm, both set to delay packets following a normal distribution [Sin98] with a mean of $200\,\mu s$ and a standard deviation of $25\,\mu s$. Our tool is configured to reorder packets by their earliest-send timestamp to prevent the inevitable inaccuracies with FIFO shown in Section 4.2.6. For this test, we again observed and captured the transmission of 5,000,000 packets at around $100\,\mathrm{Mbit/s}$, corresponding to around $130,000$ Packages per second. To evaluate the accuracy, we compare the measured values from both setups to the original distribution they are configured to emulate.

	Median	Mean	Lower Quartile	Upper Quartile
Reference Data	200.00	199.98	183.12	216.84
Sch Delay Reordered	212.19	212.18	195.18	229.19
NetEm	224.06	224.16	206.84	241.42

Table 6.2: Distribution Accuracy Values

Figure 6.4 visualizes these measurements both as a histogram and as a boxplot.

We can see that both tools produce delays that follow normal distributions that resemble the configured one and are offset slightly to the right. This fact shows that both tools are able to compute and apply the stochastic distribution.

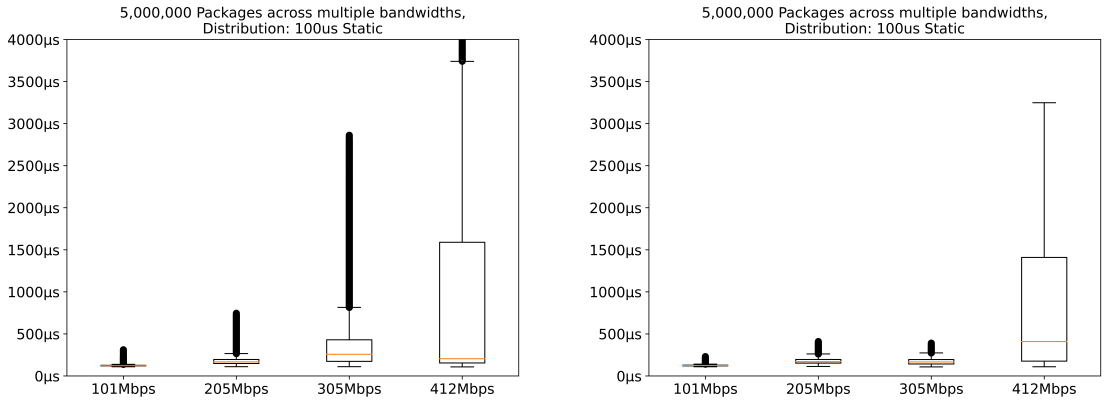
Next, we can observe that the measured distributions are not offset by the same amount. NetEm creates a distribution that is offset by $24\,\mu\mathrm{s}$ while our tool creates one that is only offset by $12\,\mu\mathrm{s}$. This observation becomes more meaningful after we adjust the offsets for the overhead we measured in Section 6.2. After adjusting, our tools distribution is offset by around $2\,\mu\mathrm{s}$ to the left while NetEms is still offset by around $5\,\mu\mathrm{s}$ to the right. This difference shows the impact of the difference in delay generation. NetEm introduces additional overhead now that it has to perform a more complex calculation for each packet inserted because it needs to calculate the delay from the distribution. Our tool, instead, does not have to perform different calculations regardless of different distributions, as it does not calculate the delays but only applies them.

By taking all these observations into account, we can conclude that our tool is able to delay packages following stochastic distributions with the same accuracy of $2\,\mu\mathrm{s}$.

6.5 Evaluation of Throughput

Our requirements in Section 3.2 stated that a suitable tool must be able to delay packages accurately while under a load of at least $100\,\mathrm{Mbit/s}$. As all previous evaluations in this chapter have been conducted at a rate of $100\,\mathrm{Mbit/s}$, we can already consider this requirement fulfilled. Regardless of that, we now want to test how our tool performs at rates that exceed this requirement and observe how this impacts performance in terms of overhead and accuracy.

We want to test how higher bandwidths affect both the accuracy with static delays and the accuracy with distributed delays. To achieve this, we repeat the tests from Section 6.3.1 and Section 6.4. This time, we will repeat the same test across four different data



(a) Sch Delay **(b)** NetEm

Figure 6.5: Delay Accuracy Across Multiple Bandwidths

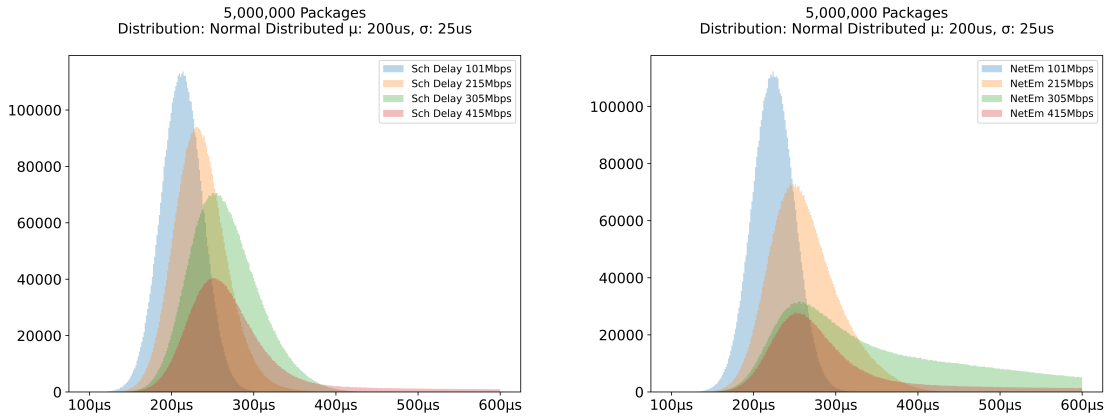
rates to observe how the results are affected. We performed the tests at $100\,\mathrm{Mbit/s}$ (around $130,000$ Packages per second), $200\,\mathrm{Mbit/s}$ (around $260,000$ Packages per second), $300\,\mathrm{Mbit/s}$ (around $390,000$ Packages per second) and $400\,\mathrm{Mbit/s}$ (around $520,000$ Packages per second).

6.5.1 Static Delay Accuracy With Increasing Bandwidth

We start with observing the accuracy of static delays across these four bandwidths. To do this, we configured both tools to delay packets by $100\,\mu\mathrm{s}$ and then measured how long the real latency was. Figure 6.5 shows the results for both our tool and NetEm.

We can clearly observe that the accuracy of our tool gets increasingly worse with increasing bandwidth. Even the jump from $100\,\mathrm{Mbit/s}$ to $200\,\mathrm{Mbit/s}$ creates an average difference of about $40\,\mu\mathrm{s}$. And beyond $200\,\mathrm{Mbit/s}$, the delays can vary so much that our tool becomes completely unable to create accurate delays.

We can also see that NetEm manages to maintain a feasible accuracy still at $300\,\mathrm{Mbit/s}$ while our tool is already wildly inaccurate from this point on. Even when both tools are vastly inaccurate at $400\,\mathrm{Mbit/s}$, NetEm manages to be more accurate by a small amount when compared to our tool. This difference shows the impact of the delay generation in User Space. At these high bandwidths, we have to transmit many more delays to the QDISC, which causes many more context switches. This high amount of context switches causes so much overhead that we are not able to accurately delay packages anymore.



(a) Sch Delay **(b)** NetEm

Figure 6.6: Distribution accuracy across multiple bandwidths

However, we can also observe similarities between both tools. Both tools have a similar shift from $100\,\mathrm{Mbit/s}$ to $200\,\mathrm{Mbit/s}$ and both tools are extremely inaccurate at $400\,\mathrm{Mbit/s}$. This observation leads us to believe that both these observations are, to a part, caused by the limitations of software delay enforcement. That is because handling such high packet rates causes overheads in the whole system, not just inside the QDISCs.

These observations lead us to the conclusion that our tool is realisticly only suited to emulate delays up to a bandwidth of about $200\,\mathrm{Mbit/s}$. Up to this speed, we still manage to create a reasonably accurate and consistent delay, but beyond $200\,\mathrm{Mbit/s}$, we cannot achieve this anymore. For use cases that require a static delay on bandwidths greater than $200\,\mathrm{Mbit/s}$, it is better suited as its design creates less overhead.

6.5.2 Distributed Delay Accuracy With Increasing Bandwidth

Lastly, we want to test how the accuracy of delay distributions is affected by increasing bandwidths. For this test, we configured both tools to delay packets following the distribution from Section 6.4, a normal distribution with a mean of $200\,\mu\mathrm{s}$ and a standard deviation of $25\,\mu\mathrm{s}$. We then tested both tools across all four bandwidths as explained in Section 6.5. Figure 6.6 shows the results for both our tool and NetEm.

Similar to Section 6.5.1, we can observe that the accuracy decreases with the increasing bandwidth. We can see that the measured distribution is shifted to the right and flattened with the increasing bandwidth. However, in contrast to our observations in Section 6.5.1 in this test, our tool remains more accurate than NetEm. As in Section

6.5.1, this difference can be traced back to the way delays are generated. But this time, it is NetEm that suffers from its design. With the more complex computation that NetEm has to perform on each enqueue, NetEm creates more overhead than the extra context switches we require. This leads to the observation we made during these tests where NetEms accuracy drops noticeably faster than one of our tools.

With all the gathered information, we conclude that our tool performs the best at bandwidths at or below $100\,\mu\mathrm{s}$. For use cases that require higher bandwidth, it depends on what functionality is required; for stochastic distributions, it is better to use our tool as it still can provide reasonable accuracy that is better than the alternative. However, for use cases that require a static delay, it is better to use tools like NetEm that create less overhead in these scenarios.

6.6 Summary

The evaluation of our tool has shown that it can accurately and consistently delays packets according to arbitrary distributions. We have observed that we can delay packets with an accuracy of $2\,\mu\mathrm{s}$, both with static delays as well as with distributed delays. Furthermore, we have seen that we can maintain this accuracy across millions of packets with a consistency of at max $5\,\mu\mathrm{s}$ deviation.

We have also shown that our tool is efficient and performant, such that it does not introduce unwanted computational overhead. The only exception is a small overhead that happens if no delay is applied, which does not concern us as this is not the use case of our tool.

We have also seen that our tool achieves a higher accuracy than the alternative NetEm. We concluded that this is because of our design of separating the delay generation from its enforcement. This gives us an advantage over NetEm, as we do not have to calculate the delays inside the QDISC, which makes us more efficient when processing packets.

However, we have also seen that these benefits only apply at data rates of $100\,\mathrm{Mbit/s}$ or less. This is both due to a limitation of our design as well as a limitation of software delay emulation. Beyond $100\,\mathrm{Mbit/s}$, the accuracy does deteriorate for both our tool and NetEm, to which we compared ourselves. Here, depending on the use case, either our tool or NetEm was better suited. NetEm managed to maintain a better accuracy in static delays at data rates above $200\,\mathrm{Mbit/s}$. Our tool managed to maintain better accuracy in distributed delays at these rates. But both tools were significantly more inaccurate than at $100\,\mathrm{Mbit/s}$.

With these observations, we concluded that delay emulation in software is only sensible in data rates at or below $100\,\mathrm{Mbit/s}$ ($130,000$ Packets per second). Within these data rates, our tool delivers excellent performance with high accuracy and consistency. With higher packet rates, our tool can still be used for creating distributed delays, but alternatives like NetEm are better suited for static delays. At data rates greater than $300\,\mathrm{Mbit/s}$, neither tool is able to delay packets accurately, and a solution outside of network emulation in software should be used.

7 Summary and Future Work

In this work, we presented the design and implementation of a tool for Linux-based systems that emulates network delays. Our goal was to develop a tool to emulate network delays following given stochastic distributions on wired connections. The main focus of this tool is the ability to flexibly define delay distributions and finetune them to specific use cases. We achieved this flexibility with a design that separates the execution of delaying from the calculation delay values.

The delay is enforced by a Linux Queueing Discipline (QDISC) running in Kernel Space. This Queueing Discipline stores packages that traverse the Network Stack and hold them back until the required delay is achieved. This QDISC can be deployed onto the Network Stack of Network Interfaces so that all packages assigned to it traverse the QDISC and get delayed.

The delay generation is outsourced into User Space. The QDISC creates a Character Device through which a User Space application can submit pre-calculated delays to the QDISC. This application can be altered or entirely replaced by another to suit the desired use cases better.

With this separation, we benefit from the performance of delaying packets in Kernel Space while using the flexibility of User Space to generate a wide variety of delay distributions. Thanks to QDISCs ability to be chained together, we can also combine our tool with outer QDISCs.

The evaluation of our tool has shown that it can delay packets and follow stochastic distributions with an accuracy of $2\,\mu\text{s}$. However, it also showed that these capabilities deteriorate with increasing bandwidth. This limitation is caused by a combination of increasing overhead due to context switches and the limitations of delaying packages in software.

As far as future research is concerned, we consider several aspects promising.

The evaluation showed that the amount of context switches becomes problematic with increasing bandwidth. Here, future research could consider refining the process of submitting delays to mitigate the increasing overhead. That could, for example, mean following our remark from Section 4.2.5 that an approach that uses `poll` to check

whether new delays should be submitted could improve the efficiency of submitting new delays to the QDISC.

The evaluation has also shown that there are limitations to delaying packages in software. Future research could explore the possibility of adapting our design to hardware solutions. For example, a design could be created where the delay is still calculated on a machine in user space, but instead of using an underlying QDISC, the delay could be directly transferred to a programable (FPGA) network card that introduces the delay.

Finally, future research could focus on the limitation of our tool, not being able to delay packages based on their properties, for example, delaying packets based on their size or arrival time. This limitation, as explained in Section 4.3, is caused by the separation of delay generation and enforcement. A possible solution for this limitation could be redesigning the QDISCs communication interface so that packet information is accessible in User Space. Another possibility would be to develop a new QDISC that specializes in these kinds of delays and could then be used together with our tool.

Bibliography

[20] *Link™ NT40A01 SmartNIC*. NT40A01-SCC-4×1. napa:tech. 2020. URL: https://marketing.napatech.com/acton/attachment/14951/f-2b3b6497-735a-43f7-8386-23c11c356e14/1/-/-/-/-/Napatech_Data_Sheet_Link_NT40A01_SmartNIC.pdf (cit. on p. 53).

[6g23] D. 6g. *Use Cases and Architecture Principles*. Tech. rep. DETERMINISTIC6G, 2023 (cit. on p. 15).

[Bie13] E. W. Biederman. *ip-netns(8) — Linux manual page*. Jan. 2013. URL: https://man7.org/linux/man-pages/man8/ip-netns.8.html (cit. on p. 53).

[com] T. kernel development community. *Character device drivers¶*. URL: https://linux-kernel-labs.github.io/refs/heads/master/labs/device_drivers.html (cit. on p. 18).

[Hem+05] S. Hemminger et al. *Network emulation with NetEm*. 2005 (cit. on p. 21).

[Hub01] B. Hubert. *tc(8) — Linux manual page*. Dec. 2001. URL: https://man7.org/linux/man-pages/man8/tc.8.html (cit. on p. 20).

[IPF22] F. 13.2. *IPFS FreeBSD Manual Pages*. June 2022. URL: https://man.freebsd.org/cgi/man.cgi?query=ipfw&manpath=FreeBSD+9-current&format=html (cit. on p. 21).

[kera] kernel.org. *Inlining in Linux — The Linux Kernel documentation*. URL: https://www.kernel.org/doc/local/inline.html (cit. on p. 50).

[kerb] kernel.org. *struct* $sk_buff^-TheLinuxKerneldocumentation$. URL: https://docs.kernel.org/networking/skbuff.html (cit. on p. 19).

[Ker17] KernelNewbies. *likely() and unlikely()*. Dec. 2017. URL: https://kernelnewbies.org/FAQ/LikelyUnlikely (cit. on p. 50).

[Ker23] M. Kerrisk. *syscalls(2) — Linux manual page*. May 2023. URL: https://man7.org/linux/man-pages/man2/syscalls.2.html (cit. on p. 17).

[KKKS03] A. Kleen, H. M. Khosravi, A. Kuznetsov, J. H. Salim. *Linux Netlink as an IP Services Protocol*. RFC 3549. July 2003. DOI: 10.17487/RFC3549. URL: https://www.rfc-editor.org/info/rfc3549 (cit. on p. 51).

[Kuz] A. Kuznetsov. *iproute2*. https://github.com/iproute2/iproute2 (cit. on pp. 20, 51).

[Lan07] R. Landley. *Red-black Trees (rbtree) in Linux*. Jan. 2007. URL: https://www.kernel.org/doc/Documentation/rbtree.txt (cit. on pp. 36, 41).

[Ray04] E. S. Raymond. *The Art of Unix Programming*. 2004. URL: http://www.faqs.org/docs/artu/ (cit. on p. 27).

[Riz97] L. Rizzo. "Dummynet: A Simple Approach to the Evaluation of Network Protocols." In: *SIGCOMM Comput. Commun. Rev.* 27.1 (Jan. 1997), pp. 31–41. ISSN: 0146-4833. DOI: 10.1145/251007.251012. URL: https://doi.org/10.1145/251007.251012 (cit. on p. 21).

[Sch14] D. Scholz. "A look at Intel's dataplane development kit." In: *Network* 115 (2014) (cit. on p. 26).

[Sin98] V. P. Singh. "Normal Distribution." In: *Entropy-Based Parameter Estimation in Hydrology*. Dordrecht: Springer Netherlands, 1998, pp. 56–67. ISBN: 978-94-017-1431-0. DOI: 10.1007/978-94-017-1431-0_5. URL: https://doi.org/10.1007/978-94-017-1431-0_5 (cit. on p. 58).

[Tho] I. M. Thomas Gleixner. *hrtimers - subsystem for high-resolution kernel timers*. URL: https://www.kernel.org/doc/Documentation/timers/hrtimers.txt (cit. on p. 41).

[Tor03] L. Torvalds. *Inlining functions (Linus Torvalds)*. Accessed on 05.10.2014. 2003. URL: https://yarchive.net/comp/linux/inline.html (cit. on p. 50).

All links were last checked on November 21, 2023

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature