UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

PH.D. THESIS

IN

INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

NETWORK TRAFFIC CONTROL DESIGN AND EVALUATION

PASQUALE IMPUTATO

TUTOR: STEFANO AVALLONE

XXXI CICLO

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE
DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

Abstract

Recently, the term bufferbloat has been coined to indicate the uncontrolled growth of the network queueing time. A number of network traffic control strategies have been proposed to control network queueing delay. Active Queue Management (AQM) algorithms such as RED, CoDel and PIE have been proposed to drop packets before the network queues become full and to notify upper layers, e.g., transport protocols, about possible congestion status. Innovative packet schedulers such as FQ-CoDel, have been introduced to prioritize flows which do not build queues. Strategies to reduce device buffering, e.g., BQL, have been proposed to increase the effectiveness of packet schedulers.

Network experimentation through simulators such as ns-3, one of the most used network simulators, allows the study of bufferbloat and to evaluate solutions in a controlled environment. In this work, we aligned the ns-3 queueing system to the Linux one, one of the most used networking stacks. We introduced in ns-3 a traffic control module modelled after the Linux one. Our design allowed the introduction in ns-3 of schedulers such as FQ-CoDel and of algorithms to dynamically size the buffers such as BQL. Also, we devised a new emulation methodology to overcome some limitations and increase the emulation fidelity. Then, by using the new emulation methodology, we validated the traffic control module with its AQM algorithms (RED, CoDel, FQ-CoDel and PIE). Our experiments prove the high fidelity of network emulation and the high accuracy of the traffic control module and AQM algorithms.

Then, we show two proposals of design and evaluation of traffic control strategies by using ns-3. Firstly, we designed and evaluated a traffic control layer for the backlog management in 3GPP stacks. The approach improves significantly the flows performance in LTE networks. Secondly, we highlighted possible design flaws in rate based AQM algorithms and proposed an alternative flow control approach. The approach allows the improvement of the effectiveness of AQM algorithms.

Our work will allow researchers to design and evaluate in a more accurate manner traffic control strategies through ns-3 based simulation and emulation and to evaluate the accuracy of other modules implemented in ns-3.

Papers published or under review

- Design and implementation of traffic-control module in ns-3, P. Imputato and S. Avallone, Workshop on ns-3 (WNS3), 2016
- Traffic differentiation and multiqueue networking in ns-3, P. Imputato and S. Avallone, Workshop on ns-3 (WNS3), 2017
- Network emulation support in ns-3 through kernel bypass techniques, P. Imputato, S. Avallone and T. Pecorella, International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS), 2017
- An analysis of the impact of network device buffers on packet schedulers through experiments and simulations, P. Imputato and S. Avallone, Simulation Modelling Practice and Theory (SIMPAT), 2018
- Smart backlog management to fight bufferbloat in 3GPP stacks, P. Imputato, N. Patriciello, S. Avallone, J. Mangues-Bafalluy, accepted for publication in Consumer Communications & Networking Conference (CCNC), 2019
- Enhancing the fidelity of network emulation through direct access to device buffers, P. Imputato, S. Avallone, under review in Journal of Networks and Computer Applications (JNCA)
- Avoiding potential design flaws in queue disciplines: the PIE case, P. Imputato, S. Avallone, M. Tahiliani and S. Patil, under review in IEEE Network Magazine (NETMAG)

Contents

List of Figures

Chapter 1

Introduction

In recent years, the network research community paid more attention to the uncontrolled growth of network queueing time. The term bufferbloat has been coined to indicate the presence of unnecessary network delay. The bufferbloat is due to higher network layers, e.g., transport layers, which try to keep high network utilization by sending as much data as possible. Network flow should be transmitted over the network with the minimum delay required at the rate of the bottleneck link over the path. However, network elements have been developed with buffers to absorb packet bursts. Such buffers are usually oversized and the availability of low-cost memory has increased their dimension further. Unfortunately, unmanaged network buffers lead to higher than necessary delays and increase the occurrence of network congestion events. Network experimentation in controlled environments with simulation, emulation or testbed is the first approach to better understanding the problem and to design solutions. With the increasing number of wireless links in todays networks, simulation is particularly useful to study the bufferbloat.

In this work, we first analyzed the networking stack of the Linux kernel, one of the most used networking stack, and experimentally characterize its traffic control infrastructure. Then, we introduced in ns-3, one of the most used network simulators, a traffic control module modeled after the Linux one. We explored possible approaches to perform traffic control module validation and to this aim, we devised an alternative technique to support network emulation in simulators. Finally, we presented two study cases in which we designed a new traffic control layer in LTE networks and we gained insights on the design and evaluation of AQM algorithms.

In the following, we provide context information about i) network traffic control to reduce the network queueing delay, ii) network experimentation approaches to design and evaluate traffic control solutions. Then we state the motivation for this work and highlight the contribution. Finally, we provide an outline of the thesis.

1.1 Context

In the following sections, we provide an overview about network traffic control and of the approaches to network experimentation to design and evaluate traffic control strategies.

1.1.1 Network traffic control

Network traffic control includes activities to increase network resources utilization and to increase traffic flows delivery efficiency. In terms of network performance parameters, traffic control activities aim to provide i) *high network throughput* and ii) *low flow delay* [1]. The condition of high network throughput allows the use of all transmission resources while the condition of low flow delay allows to achieve packets transport with the minimum requested network delay. A packet should be ideally delivered to the receiver at the rate of the bottleneck link along the network path and with packet delay composed of transmission delay plus processing delay of the intermediate elements [2].

Network transport protocols, e.g., TCP, achieve the transport level communication between end nodes and exploit all the bandwidth available. The TCP Congestion Avoidance (CA) mechanism tries to avoid network congestion while sharing the bandwidth in a fair manner among different flows. Basically, the CA mechanism tries to discover network congestion status since the network lost a packet. As a consequence of lack of notification of lost packet, the sender will reduce its sending rate to relieve the network congestion status. A congestion notification occurs since a full FIFO buffer drops a packet.

Indeed, network elements, e.g., routers or access points, usually are designed with FIFO per-interface buffers in order to absorb packets burst. The ability to absorb bursts allows to keep packets waiting for transmission avoiding network device starvation intervals, i.e., time intervals in which the device is ready to transmit other packets but there are no packets. Then, the network delay has an important contribution in the *queueing delay*, i.e., waiting time in the queues, contributes the most to the network delay. Network buffers are usually sized according to the BDP (Bandwidth Delay Product) rule. The presence of low-cost memory has in practice increased their dimension. The term *bufferbloat* has been coined to indicate the uncontrolled growth of network queueing time due, among others, to the general attitude of higher layers to send as much data as possible to exploit the network bandwidth. Unmanaged buffers exacerbate the bufferbloat problem [2]. Such a phenomenon occurs in each network segment where packet queueing occurs.

A number of strategies called Active Queue Management (AQM) algorithms, such as RED [3], CoDel [4] and PIE [5] have been designed to contrast the uncontrolled growth of the queueing time. The idea is that an AQM algorithm drops some packets to notify upper layers of possible congestion status. Algorithms such as RED uses the queue length as metric of congestion status, while algorithms such as CoDel and PIE use the estimated queueing time as metric of network congestion. These algorithms can be used in different levels of the network stack, to manage device level buffers or network level buffers.

Network elements rely on packet schedulers to regulate the flows in the outgoing path. Basically, a network scheduler is in charge of choosing the next packet to transmit to the device. Packet schedulers have been proposed to provide prioritization

to flows which do not build queues while controlling the queueing delay through AQM algorithms. The most promising ones are FQ-CoDel [6] and CAKE [7]. In FQ-CoDel, the scheduler separates the flows (by five-tuple hashing) and regulates the queueing delay of each flow by using CoDel. The idea is to separate good flows, i.e., which do not build queues, from bad flows, i.e., which build queues and prioritize good flows over bad flows. CAKE adds a traffic shaper to an idea of flow separation similar to that in FQ-CoDel. In CAKE the shaper aims to reduce the out of control buffering, i.e., device buffering.

From a practical point of view, network elements are often implemented within the Linux kernel. For instance, Android-based systems rely on Linux kernel and use its networking subsystem for network interfaces such as LTE and WiFi. The Linux kernel has the traffic control infrastructure to provide support for quality of service [1]. This layer sits between the IP layer and the device layer. A number of elements, such as packet filters and shapers have been introduced in Linux traffic control. The most important component is the *queueing discipline* (qdisc) which has the role of keeping the packets waiting for transmission to the device. In Linux, AQM algorithms and packet schedulers have been implemented as qdisc. Then, traffic control sends packets from the qdisc to the device to enqueue packets in its queue.

The network device queues, called device rings, are managed by the device driver. Each device has at least one ring for packet transmission and one for packet reception. In the incoming path, received packets are passed to the upper layer which processes them. In the outgoing path, there is a *flow control* between the device and the upper layer. Basically, the device driver stops the transmission ring when there is no more space for packets, i.e., the device queue is full. This mechanism slows down the upper layer to send more packets to the device. A qdisc can manage packets waiting for a queue restart event from the device. After the transmission of a number of packets, the device will notify the upper layer restarting the queue. Recently, an algorithm called BQL has been introduced in the Linux kernel to dynamically determine the size of the device transmission ring. The idea is to reduce the device buffer size keeping the same level of network utilization. Unfortunately, a number of devices, and in particular WiFi and LTE devices, lack proper support to a mechanism such as BQL to reduce the impact of device buffer. Also, the impact of device ring reduction on device such as WiFi and LTE is not clear. Indeed, such devices adapt the link layer bandwidth based on the queued data.

1.1.2 Network experimentation

Network research relies on different options to design and to evaluate solutions. Controlled environments through simulation, emulation and testbed are possible approaches to understand the bufferbloat, to prototype and to evaluate solutions [8]. With the increasing number of wireless links in nowadays networks, the simulation is gaining growing interest.

Indeed, simulation offers numerous benefits, including reproducibility of wireless scenarios or to recreate scenario with a large number of systems and the ability to isolate the effects of undesired factors. Another important aspect is the experimentation *flexibility*. The simulation has a low level of complexity and allows researchers to quickly prototype and experiment with innovative technologies, e.g., not available (or expensive) in real testbed [9]. However, the simulation challenge is the *credibility* [8]. A simulator should reproduce as much accurately as possible the real networking stack under evaluation in order to draw credible conclusions. Also, the simulated modules should be validated through comparison with real implementations when available. Moving from simulation towards network emulation and testbed increases experimentation credibility reducing the experimentation flexibility.

Experimentation on testbed allows the researcher to asses the performance of real protocols or algorithms in real networking scenarios. However, the experimentation flexibility is limited to the technologies available and to specific implementation limitations. For instance, in the bufferbloat context, manufacturers introduce queues in-firmware unaccessible to software implementations. In these cases, researchers have limited possibilities to isolate the firmware contribution in an evaluation of the effectiveness of bufferbloat countermeasures.

Emulation, as an intermediate strategy to network experimentation, typically includes network emulation and environment emulation. In environment emulation, the real implementations of protocols at the higher layers of the network stack are used, while the channel access function and the transmission through a channel are simulated. Various tools can be used to simulate specific features of a transmission channel, including Netem [10], Dummynet [11], network simulators supporting the injection of traffic from the real world (such as ns-3 [12] and OMNet++ [13]). In network emulation, instead, simulated components interact with real hosts through a real network. This approach exploits the ability of simulators such as ns-3 and OMNet++ to exchange packets with real network devices and schedule events in real-time. The ability of a simulator, e.g., ns-3, to support network emulation is a fundamental ability. Indeed, in an emulation scenario, simulator can be integrated to explore other interesting scenarios. Basically, the simulation allows more flexibility to study simulated networks or simulated applications.

In this work, we targeted the ns-3 network simulator as a tool to evaluate the bufferbloat countermeasures [12]. The design of ns-3 is inspired by Linux kernel networking stack which is one of the most used networking kernel. Also, ns-3 is the most used network simulator in research centers and academia to evaluate the effectiveness of proposed solutions. ns-3 allows full stack simulation and has support for network emulation. ns-3 allows the simulation of wired and wireless communication technologies, e.g., Ethernet and WiFi, and offers support for the simulation of technologies in the 5G context, e.g., LTA-A, LTE-NR, mmWawe, 802.11ax.

1.2 Motivation

Designing traffic control strategies often relies on network simulation. Then, it is important to have accurate tools to accurately evaluate packet schedulers, AQM algorithms and dynamic queue sizing algorithms.

Unfortunately, ns-3 *lacked* a Linux equivalent traffic control infrastructure. The measured parameters of network delay, throughput and packet loss were strongly affected by this aspect. Some *design limitations*, i.e., a single queue level at device level with network header encapsulated into the transport header [14], denied the introduction of modern schedulers such as FQ-CoDel, and algorithms to dynamically size buffers such as BQL.

In order to asses the effectiveness of AQM algorithms and to design new algorithms, it is important to rely on tools able to *recreate* the most realistic network stack scenarios. A number of AQM algorithms have been initially proposed with the support of network simulation in ns-2 and ns-3 [3]–[5]. However, such algorithms were evaluated at device layer, i.e., neglecting the impact of device buffer, which is not what occurs in a real system. In literature there are studies which try to understand the effectiveness of AQM algorithms in real systems using experiments on testbeds [15], [16]. However, the experimentation is limited to available technologies which support the use of optimization strategies such as BQL.

The implementations of AQM algorithms and packet schedulers available in ns-3 was *not validated* against real implementations. This was due to some *emulation support limitations* implemented in ns-3, i.e., the device queue status was not accessible to the ns-3 process. The emulation limitations did not allow to setup a proper scenario to compare in a fair manner real and simulated implementations.

1.3 Contribution

This work analyzes the network stack of the Linux kernel and experimentally characterizes the impact of the device buffer on network packet schedulers. Then, we present the design of the ns-3 traffic control module, which was modeled after the Linux one. In order to validate the introduced module, we devised a new methodology to perform network emulation and used it to validate traffic control, AQM algorithms and schedulers implemented in ns-3. Finally, we move on design and evaluation of traffic control strategies. More specifically the work contribution consists of:

- an analysis of the impact of device buffer on packet schedulers in general and on AQM algorithms in particular;
- the design and implementation of the traffic control module in ns-3 modeled after the Linux one;

- the design and implementation of a methodology to improve the emulation fidelity and to allow ns-3 modules validation;
- a software traffic control layer for backlog management in 3GPP stacks;
- insights on AQM design flaws and design of flow control aware AQM algorithms.

1.4 Thesis structure

The rest of this work is structured as follows:

- in chapter 2 we analyze the Linux networking stack with a focus on the queueing system and experimentally prove the impact of the device buffer on the effectiveness of packets schedulers in general and of AQM algorithms in particular;
- in chapter 3 we present the design and implementation of the traffic control module in the ns-3 network simulator and a preliminary performance evaluation which compares the previous and the proposed stack;
- in chapter 4 we present a new methodology to support network emulation and its implementation in ns-3. Then we use the new methodology to validate the traffic control module and AQM algorithms;
- in chapter 5 we present two proposals on design and evaluation of new network traffic control strategies;
- finally, in chapter 6 we draw the conclusions of this work.

Chapter 2

An experimental characterization of the impact of device buffer on packet schedulers

Most of the packet delay can be usually ascribed to the time spent in the many queues encountered by the packet. In this chapter, we focus on the queues employed by the traffic control infrastructure and by the network device drivers. Reducing the queuing time due to the former is the objective of a plethora of scheduling algorithms developed in the past years and referred to as AQM algorithms. Conversely, the impact of the additional queuing in the buffer of the network device driver on performance and on the effectiveness of AQM algorithms has instead received much less attention. In this chapter, we report the results of an experimental analysis we conducted to gain a better insight into the impact that network device buffers (and their size) have on performance. We also provide insights of the effectiveness of Dynamic Queue Limits (DQL), an algorithm recently introduced in the Linux kernel to dynamically adapt the size of the buffers held by network device drivers. The experiments we conducted show that DQL not only enables to reduce the queuing time in the network device buffers, which is essential to ensure the effectiveness of AQM algorithms, but also enables to keep latency stable, which is important to reduce the jitter.

2.1 Introduction

Network devices make a wide use of buffers to temporarily store packets waiting to be transmitted. The size of such buffers heavily influences network performance, primarily packet delay and loss probability. The buffer size should be large enough to accommodate packet bursts, but it should not be excessively large, in order to avoid that the latency experienced by packets grows in an uncontrolled manner, a problem which is commonly referred to as bufferbloat [2]. A common rule-of-thumb is to make the buffer size proportional to the link bandwidth (i.e., the so called Bandwidth-Delay Product). However, the use of such a rule has been questioned, e.g., in [17], where authors claim that much smaller buffers can be employed.

In practice, the availability of memory chips at low cost often leads to the use of oversized buffers.

Limiting the queuing delay resulting from the use of large buffers has been the objective of a plethora of Active Queue Management (AQM) algorithms that have been defined by the research community in the last decades. RED (Random Early Drop) [3] has been one of the first AQM algorithms to be proposed and has been followed by a number of variants, such as Adaptive RED [18], Gentle RED [19] and Nonlinear RED [20]. RED and its derivatives can be classified [21] as queue-based schemes, since they aim to maintain the queue length at a target level. Other algorithms, such as BLUE [22] and AVQ (Adaptive Virtual Queue) [23] use instead the packet arrival rate as a congestion measure and hence can be classified as rate-based schemes. A combination of the two approaches is proposed by algorithms such as REM (Random Exponential Marking) [24]. More recently, CoDel (Controlled Delay) [4] has been proposed to specifically address the bufferbloat problem. CoDel aims to keep the queuing delay below a target value and drops packets that exceed such threshold *after* they have been dequeued. Furthermore, a few AQM algorithms, such as SFB (Stochastic Fair Blue) [25], AFCD (Approximated-Fair and Controlled Delay) [26] and FQ-CoDel [6], were proposed with the aim of additionally providing fairness among different flows.

The interaction with the TCP congestion control mechanism has been taken into account in the design of some AQM algorithms. The PI (Proportional Integral) [27] controller is among the earliest of such approaches and is based on a linearized dynamic model for a TCP/AQM system. PIE (Proportional Integral controller Enhanced) [5] computes the drop probability based on both the departure rate and the queue length. The stability of PIE is demonstrated by using a TCP fluid model. PI^2 [28] extends PIE with the objective to improve the coexistence between *classic* congestion controls (TCP Reno, Cubic, etc.) and *scalable* congestion controls (Data Center TCP). In [29], authors evaluate the interaction among AQM algorithms (such as CoDel) and low-priority congestion control techniques (such as LEDBAT [30]) through both experiments and simulations. A theoretical analysis is instead presented in [31] to study the interaction between Compound TCP and REM.

AQM algorithms have been implemented in real systems such as the Linux operating system as *queuing disciplines* within the Traffic Control (TC) infrastructure [1]. As such, AQM algorithms manage packets before they are handed to the device driver. The latter employs its own buffer, usually called *transmission ring*, to avoid the starvation of the network interface. Thus, packets are enqueued again after being scheduled by the queuing discipline. Also, a flow control strategy is implemented to regulate the passing of packets from the queuing discipline to the transmission ring. While the interactions between TCP congestion control and AQM algorithms have been addressed by a number of works, the impact of such additional queuing and the related flow control remains largely unexplored. To the best of our knowledge, only [32] recognizes the presence of the transmission rings, but authors just

model them as FIFO queues to evaluate their impact on the service guarantees of fair-queuing schedulers.

The goal of this chapter is to fill this gap by evaluating the impact of transmission rings and flow control on network performance, in general, and on the effectiveness of AQM algorithms, in particular. We conducted a thorough experimental campaign which provided us with a number of insights. For instance, if the queuing discipline does not differentiate among traffic flows and its capacity is not saturated, the size of the transmission ring has no impact on network performance; when the queuing discipline assigns different priority levels to flows, the latency experienced by prioritized flows increases with the size of the transmission ring. It turns out, however, that sizing the transmission ring is not trivial: a too small ring causes a throughput loss, a too large ring causes a high packet delay. To complicate things further, for a given transmission ring size, the packet delay is affected by multiple factors, including packet size and CPU load. Also, rather unexpectedly, latency may exhibit a large variance and exceed by far the waiting time in the queuing discipline and in the transmission ring.

Another contribution of this chapter is an experimental evaluation of Dynamic Queue Limits (DQL), a mechanism that has been recently introduced in the Linux kernel with the goal of dynamically computing the number of packets to store in the transmission ring in order to prevent starvation. Our experiments showed that DQL is effective in keeping latency low when AQM algorithms or priority packet schedulers are used, while it has no impact when a queuing discipline that enqueues all the packets in a single queue is used below its capacity. Also, when DQL is used, latency turns out to be rather stable and equal to the waiting time in the queuing discipline and in the transmission ring.

The rest of this chapter is structured as follows. Section 2 provides some background information about how queuing works in Linux, while Section 3 presents an in-depth description of DQL Section 4 presents the results of our experimental campaign. Finally, Section 5 concludes the chapter.

2.2 Background

Applications write data in the buffer of their sockets to have it delivered to the destination. On Linux systems, the size of a socket buffer is kept small by a mechanism called TCP Small Queues (TSQ). Basically, an application is prevented to write data in a socket buffer if there are already (approximately) two packets not yet transferred to the network interface card, for that socket. The goal of this mechanism is to reduce round-trip times and hence mitigate the bufferbloat issue. Data is then processed by the transport layer protocol. In case of TCP, data is segmented into packets and passed to the network layer when the congestion control engine determines that new packets can be sent. A technique that is often enabled on Linux systems is TCP Segmentation Offload (TSO), which allows large data segments (even larger than the

Maximum Transmission Unit of the underlying network) to be sent to the network interface card, provided that the latter is able to perform segmentation and hardware checksumming. Using TSO relieves the CPU of the task of segmenting data and allows to deal with a smaller number of packets.

Once the network layer has determined the outgoing network interface for a packet (based on the routes available in the routing table), the packet is passed to the TC infrastructure, which hands the packet to the queuing discipline (henceforth, queue disc) installed on the outgoing device. A queue disc can enqueue the packet, mark the packet (e.g., in case it supports Explicit Congestion Notification) or drop the packet (either when the packet is enqueued or later, when the packet is dequeued). The queue disc is in charge of scheduling the enqueued packets and therefore can be employed to enforce traffic prioritization strategies. Enqueuing a packet in a queue disc triggers a call to the `__qdisc_run` function, which requests to dequeue at most a configurable (through the dev_weight parameter of the proc filesystem) number of packets from the same queue disc.

Each packet dequeued by a queue disc is handed to the network device driver through the ndo_start_xmit callback implemented by the driver. Such routine enqueues the received packet in a transmission ring and requests the DMA (Direct Memory Access) to transfer the packet to the network device. Given that the device may not be ready to receive packets and the DMA has to acquire the system bus, multiple such requests may queue up. The ndo_start_xmit callback also takes some actions to perform flow control. To this end, a `struct netdev_queue` element is associated with each transmission ring (e.g., 802.11 drivers use four rings for Quality of Service support, while Ethernet drivers use multiple rings to increase performance in case multiple processor cores are available). When the ndo_start_xmit callback enqueues a packet in one of the rings, it checks whether such ring can accommodate another packet. If not, it changes the status of the corresponding struct netdev_queue element to stopped, so that the queue disc refrains from sending further packets.

When a transfer of packets is completed, the network device raises an interrupt and the corresponding handler (provided by the device driver) is then executed. Such handler moves the packets transferred to the device to a list of completed packets (so as to make room in the transmission ring), which are actually freed when the software interrupt of type NET_TX_SOFTIRQ is executed. Freeing the memory is a time consuming operation and therefore is not performed by the interrupt handler, which should be executed in the least possible time. The interrupt handler also participates in the flow control. If there is enough room in the transmission ring, the netif_tx_wake_queue function is called, which ensures that the corresponding struct netdev_queue element is not stopped and, if it was stopped, adds the corresponding queue disc to the list of queue discs that will be served the next time the software interrupt of type NET_TX_SOFTIRQ is executed. Serving a queue disc means requesting it to dequeue packets by calling the __qdisc_run function. Such technique allows to ensure that packets are pulled from the queue disc if there is

available space in the device transmission ring.

A transmission ring is made of a configurable number of *descriptors*, each of which can hold a pointer to a memory region. A single packet may require multiple descriptors, especially if *scatter/gather I/O* is available. Scatter/gather I/O allows the system to perform DMA I/O operations with non contiguous blocks of data. Given that the various protocols of the network stack allocate their own buffer to store their data (e.g., the protocol header), exploiting scatter/gather I/O allows the kernel to avoid copying all such buffers into a single block of data. Finally, we mention that some interrupt mitigation techniques are often employed. Indeed, high speed networking can generate thousands of interrupts per seconds, which may lead the CPU to spend most of its time serving such interrupt requests. Some network interface cards implement interrupt coalescence mechanisms [33] to reduce the rate at which interrupts are generated. The Linux kernel includes NAPI (New API), an infrastructure which disables interrupts and enforces polling when the traffic load is high, while enables interrupts when the traffic load is low. Most of the newer network device drivers in the Linux kernel support NAPI.

2.3 Dynamic Queue Limits

Dynamic Queue Limits is a mechanism that has been recently introduced in the Linux kernel to dynamically determine the maximum amount of bytes (denoted as "limit") that shall be stored in the transmission ring of a device. The goal is to keep such amount around the minimum value that guarantees to avoid starvation. The computed limit is enforced by exploiting the existing flow control mechanism. Given that the limit is computed in bytes, the algorithm is also known as BQL (Byte Queue Limits). Computing the limit in bytes rather than packets enables to more precisely estimate the queuing time, given that the transmission time of a packet is not constant, but it is directly proportional to the packet size. The introduction of DQL in the Linux kernel consisted in the addition of the library implementing the algorithm and in a few changes to the network stack.

The DQL library introduces the struct dql type to store information needed by the DQL algorithm and provides five functions:

- dql_init, which initializes the information stored in a struct dql element;
- dql_reset, which resets the information stored in a struct dql element;
- dql_queued, which updates the total amount of bytes ever enqueued in the transmission ring;
- dql_avail, which returns the difference between the current limit and the amount of bytes queued in the transmission ring (can be negative);
- dql_completed, which records the size of the packets whose transmission to the device has been completed and recalculates the limit.

```
DQL_QUEUED(queuedObjects)
1  lastObjCnt ← queuedObjects
2  numQueued ← numQueued + queuedObject

DQL_AVAIL()
1  return adjLimit − numQueued

POS(a, b)
   return max((a − b), 0)

DQL_COMPLETED(completedObjects)
 1  completed ← numCompleted + completedObjects
 2  limit ← currLimit
 3  ovlimit ← POS(numQueued − numCompleted, limit)
 4  inProgress ← numQueued − completed
 5  prevInProgress ← prevNumQueued − numCompleted
 6  allPrevCompleted ← (completed − prevNumQueued ≥ 0)
 7  if ((ovlimit AND ¬inProgress) OR (prevOvlimit AND allPrevCompleted))
       then ▷ The queue is starved
 8            limit+ = POS(completed, prevNumQueued) + prevOvlimit
 9            slackStartTime ← current time
10            lowestSlack ← UINTMAX
11       else if (inProgress AND prevInProgress AND ¬allPrevCompleted)
                 then ▷ The current limit may be too high
12                    slack ← POS(limit + prevOvlimit, 2 ∗ completedObjects)
13                    if (prevOvlimit)
14                      then slackLastObjs ← POS(prevLastObjCnt, prevOvlimit)
15                      else slackLastObjs ← 0
16                    slack ← max(slack, slackLastObjs)
17                    if (slack < lowestSlack)
18                      then lowestSlack ← slack
19                    if (current time > slackStartTime + slackHoldTime)
20                      then limit ← POS(limit, lowestSlack)
21                           slackStartTime ← current time
22                           lowestSlack ← UINTMAX
23  limit ← min(max(limit, minLimit), maxLimit)
24  if (limit ≠ currLimit)
25     then currLimit ← limit
26          ovlimit ← 0
27  adjLimit ← limit + completed
28  prevOvlimit ← ovlimit
29  prevLastObjCnt ← lastObjCnt
30  numCompleted ← completed
31  prevNumQueued ← numQueued
```

FIGURE 2.1: Pseudo-code of the function of the DQL library

The dql_queued function (Fig. 2.1) is meant to be called everytime new packets are enqueued in the transmission ring to communicate the size of the packets that have been just enqueued. The dql_queued function records this value in the *lastObjCnt* variable and updates the *numQueued* variable storing the total amount of bytes ever enqueued in the transmission ring.

The dql_avail function (Fig. 2.1) returns the difference between the current limit and the amount of bytes stored in the transmission ring. Indeed, *adjLimit* is set by the dql_completed function to the sum of the limit and the total amount of bytes ever dequeued from the transmission ring. As described later, the device queue is stopped if this difference is negative, i.e., if the amount of bytes in the queue exceeds the limit. When the queue is stopped, no packets are sent from the upper layers. Hence, DQL keeps the amount of bytes stored in the transmission ring around the computed limit.

A key role is played by the dql_completed function (Fig. 2.1), which is meant to be called when the device driver is notified that the transmission of some packets to the device has been completed, to communicate the size of such packets (which are dequeued from the transmission ring). The dql_completed function recalculates the DQL limit based on the amount of bytes enqueued/completed in the last intervals. To the purpose of illustrating the operation of the dql_completed function, we denote by:

- t_i^c the time when the i-th call to dql_completed is made;
- t_i^q the time when the last call to dql_queued prior to the i-th call to dql_completed is made;
- $Q(t)$ the total amount of bytes ever enqueued in the transmission ring at time t;
- $C(t)$ the total amount of bytes ever dequeued from the transmission ring at time t;
- $\Delta(t) = Q(t) - C(t)$ the amount of *in-flight* bytes, i.e., those bytes that are in the transmission ring waiting for their transmission to the device to start or to be completed, at time t;
- $q(t_{i-1}^q, t_i^q) = Q(t_i^q) - Q(t_{i-1}^q)$ the amount of bytes enqueued in the time interval $(t_{i-1}^q, t_i^q]$. Note that there may be multiple calls to dql_queued in such time interval;
- $c(t_i^c) = C(t_i^c) - C(t_{i-1}^c)$ the amount of bytes dequeued from the transmission ring at time t_i^c;
- $limit_i$ the limit computed by the i-th call to dql_completed.

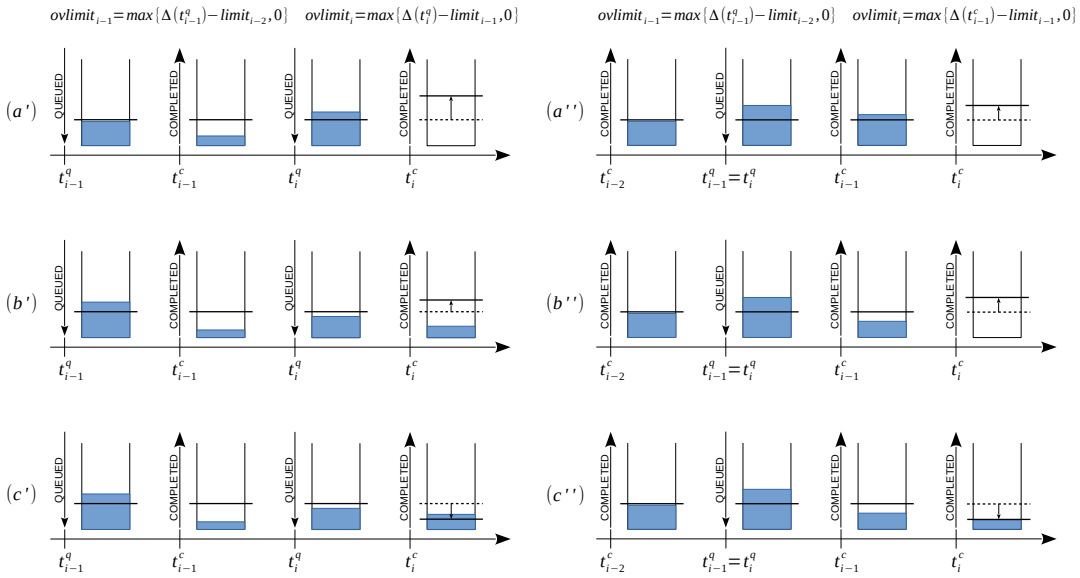


FIGURE 2.2: Limit re-calculation done by the dql_completed function. Starvation occurs in the cases of the top two rows, where the limit is consequently increased. The last row shows instead two cases where the limit is too high and is consequently decreased.

It follows that, when the i-th call to dql_completed is made, parameters are computed as follows:

$$ovlimit_i = max\left\{Q(t_i^q) - C(t_{i-1}^c) - limit_{i-1}, 0\right\} \tag{2.1}$$

$$inProgress_i = Q(t_i^q) - C(t_i^c) = \Delta(t_i^c) \tag{2.2}$$

$$prevInProgress_i = Q(t_{i-1}^q) - C(t_{i-1}^c) = \Delta(t_{i-1}^c) \tag{2.3}$$

$$allPrevCompleted_i = (C(t_i^c) - Q(t_{i-1}^q) \geqslant 0) \tag{2.4}$$

$$prevOvlimit_i = \begin{cases} 0 & \text{if the } (i-1)\text{-th call} \\ & \text{updated the limit} \\ ovlimit_{i-1} & \text{otherwise} \end{cases} \tag{2.5}$$

Equations (2.2) and (2.3) are obtained by considering that $Q(t_i^q) = Q(t_i^c)$, as, by definition, there are no other calls to dql_queued in between t_i^q and t_i^c.

Figure 2.2 illustrates the operation of dql_completed in some scenarios. We denote by COMPLETED the event of calling the dql_completed function and by QUEUED the event of calling the dql_queued function. We consider the case where a COMPLETED event is preceded by a QUEUED event and the case where it is preceded by another COMPLETED event. In particular, we consider two sequences of events: QUEUED-COMPLETED-QUEUED-COMPLETED (left column) and COMPLETED-QUEUED-COMPLETED-COMPLETED (right column). At the top of the figure is indicated how eq. (2.1) can be written for each of such cases. The goal is to show how the limit is adjusted after the last COMPLETED event depending on what happened in the previous events. The various cases analyzed are described hereinafter.

(a') When dql_completed is called at time t_i^c, *ovlimit* is computed as the difference between the amount of bytes in the queue at time t_i^q ($\Delta(t_i^q)$) and the current limit. If, as is in this case, *ovlimit* is positive and the queue is empty at time t_i^c (hence *inProgress* is null), the queue is considered starved (line 7 of the pseudo-code) and the limit is increased. The rationale is that, if after the last event (a QUEUED event, in this case) there is an amount of bytes in the queue larger than the limit and now the queue is empty, it means that the limit has to be increased (recall that DQL keeps the amount of bytes stored in the transmission ring around the computed limit). The limit is then increased (line 8 of the pseudo-code) by:

$$max\left\{C(t_i^c) - Q(t_{i-1}^q), 0\right\} + prevOvlimit_i \tag{2.6}$$

which can be written as

$$max\left\{q(t_{i-1}^q, t_i^q) - \Delta(t_i^c), 0\right\} + prevOvlimit_i \tag{2.7}$$

or

$$max\left\{c(t_i^c) - \Delta(t_{i-1}^c), 0\right\} + prevOvlimit_i \tag{2.8}$$

Given that in this case the queue is empty at time t_i^c, from eq. (2.7) it follows that the limit is increased by the amount of bytes enqueued at time t_i^q plus the difference between the amount of bytes in the queue and the limit at time t_{i-1}^q, if such a difference is positive and the $(i-1)$-th call to dql_completed did not update the limit. Finally, we note that, if *ovlimit* is null, i.e., the amount of bytes stored in the queue is less than the limit at time t_i^q, then the limit is not increased. Indeed, it might be that the upper layers are not generating heavy traffic and hence there is no point in increasing the limit.

(a'') This case differs from the previous one in that the i-th call to the dql_completed function is preceded by a COMPLETED event. Consequently, *ovlimit* is computed as the difference between the amount of bytes in the queue at time t_{i-1}^c ($\Delta(t_{i-1}^c)$) and the current limit. Thus, the rationale is the same as case (a'): if after the last event (a COMPLETED event, in this case) there is an amount of bytes in the queue larger than the limit and now the queue is empty, it means that the limit is too low. Given that $C(t_i^c) = Q(t_{i-1}^q)$ in this case, from eq. (2.6) it follows that the limit is increased by *prevOvlimit*, i.e., by the difference between the amount of bytes in the queue and the limit at time t_{i-1}^q, if such a difference is positive and the $(i-1)$-th call to dql_completed did not update the limit.

(b') This case shows the other situation where the queue is considered starved, i.e., when *prevOvlimit* is positive and *allPrevCompleted* is true. For *prevOvlimit* to be positive, the $(i-1)$-th call to dql_completed must have not updated the limit and the amount of bytes in the queue must exceed the limit at time t_{i-1}^q. Also, *allPrevCompleted* must be true, i.e., the transmission to the device of all the bytes in the queue at time t_{i-1}^q must have been completed by the time the

i-th call to dql_completed is made. Thus, the queue is considered starved if the amount of bytes in the queue exceeds the limit at time t^q_{i-1} and the amount of bytes completed since the last COMPLETED event is higher than the amount of bytes in the queue after the last COMPLETED event ($C(t^c_i) - Q(t^q_{i-1}) = c(t^c_i) - \Delta(t^c_{i-1}) \geqslant 0$), or, equivalently, the amount of bytes left in the queue is less than the amount of bytes enqueued since the last COMPLETED event ($C(t^c_i) - Q(t^q_{i-1}) = q(t^q_{i-1}, t^q_i) - \Delta(t^c_i) \geqslant 0$). In this case, the limit is increased (eq. (2.7)) by the sum of the difference between the amount of bytes in the queue and the limit at time t^q_{i-1} (which is certainly positive) and the difference between the amount of bytes enqueued since the last COMPLETED event and the amount of bytes left in the queue at time t^c_i (which is certainly non-negative).

(b") This case differs from the previous one in that the *i*-th call to the dql_completed function is preceded by a COMPLETED event. Like case (b'), in order for *prevOvlimit* to be positive, the $(i-1)$-th call to dql_completed must have not updated the limit and the amount of bytes in the queue must exceed the limit after the last QUEUED event. However, unlike case (b'), the queue must be empty at time t^c_i in order for *allPrevCompleted* to be true (in this case, $C(t^c_i) - Q(t^q_{i-1}) = C(t^c_i) - Q(t^c_i)$, which cannot be positive). In this case, from eq. (2.6) it follows that the limit is increased by *prevOvlimit*, i.e., by the difference between the amount of bytes in the queue and the limit at time t^q_{i-1} (which is certainly positive). Finally, we note that this case differs from case (a'') because the amount of bytes in the queue does not have to exceed the limit at time t^c_{i-1}. If it does not, the condition (*ovlimit* AND $\overline{inProgress}$) is false and hence it is not sufficient to detect the starvation described in this scenario.

(c') This case shows a situation where the limit is considered too high and is therefore decreased. According to line 11 of the pseudo-code, the limit is considered too high if the queue is not empty now, it was not empty after the last COMPLETED event and the transmission to the device of all the bytes that were in the queue after the last COMPLETED event has not been completed yet ($C(t^c_i) - Q(t^q_{i-1}) = c(t^c_i) - \Delta(t^c_{i-1}) < 0$). Then, the slack, i.e., the amount of excess data, is computed as the maximum between (i) the difference between the limit plus *prevOvlimit* (in the case of figure, $\Delta(t^q_{i-1})$) and two times the amount of bytes completed since the last COMPLETED event ($c(t^c_i)$) and (ii) the difference between the amount of bytes enqueued at time t^q_{i-1} and *prevOvlimit*, if *prevOvlimit* is positive and such a difference is positive, or 0, otherwise. The limit is then decreased by the lowest of the slack values computed since the last time the limit was updated, only if a minimum amount of time has elapsed since such update. Finally, we note that cases (b') and (c') only differ in the amount of bytes dequeued from the transmission ring at time t^c_i ($c(t^c_i)$). These amounts differ for few bytes. Nonetheless, this difference is

such that *allPrevCompleted* is true for case (b') and false for case (c'). As a consequence, the limit is increased for case (b') and decreased for case (c'). We believe that such an observation deserves further investigation.

(c") This case differs from the previous one in that the i-th call to the dql_completed function is preceded by a COMPLETED event. The condition tested to determine whether the limit is too high (line 11 of the pseudo-code) is now satisfied if the queue is not empty now and it was not empty after the last COMPLETED event. Indeed, in this case *allPrevCompleted* is false iff the queue is not empty at time t_i^c. The limit is then decreased according to the procedure described for the previous case.

Finally, the limit computed as described above is adjusted in order to be not less than a configurable minimum value and not greater than a configurable maximum value (line 23 of the pseudo-code).

The introduction of DQL also required a few changes to the network stack. Firstly, an element of type struct dql has been added to the struct netdev_queue element associated with the transmission ring of a device to store the variables used by DQL. Secondly, device drivers do not directly call the functions of the DQL library, but they call two newly introduced functions which also take care of starting and stopping the device queues:

- netdev_tx_sent_queue, which is intended to be called by the network device driver when a packet is received from the network stack (i.e., in the ndo_start_xmit callback). This function first calls dql_queued and then, if the value returned by dql_avail is negative, the device queue is stopped. Thus, if the amount of bytes in the queue exceeds the current limit after a call to dql_queued, then the device queue is stopped and no more packets are sent from the upper layers.

- netdev_tx_completed_queue, which is intended to be called by the network device driver when a transfer of packets to the device is completed (i.e., in the interrupt handler or the polling callback provided by the device driver). This function first calls `dql_completed` and then, if the value returned by `dql_avail` is non negative, the device queue is started. Also, if the device queue was stopped, the corresponding queue disc is added to the list of queue discs that will be served the next time the software interrupt of type NET_TX_SOFTIRQ is executed. Thus, if the device queue was stopped and now the amount of bytes in the queue is below the limit, then we start pulling packets again from the upper layers.

Finally, we note that it suffices to (properly) call the above two functions for a device driver to support DQL.

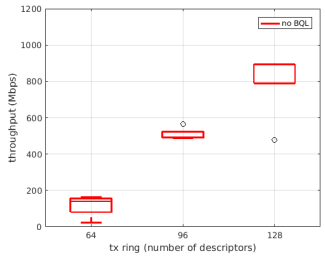
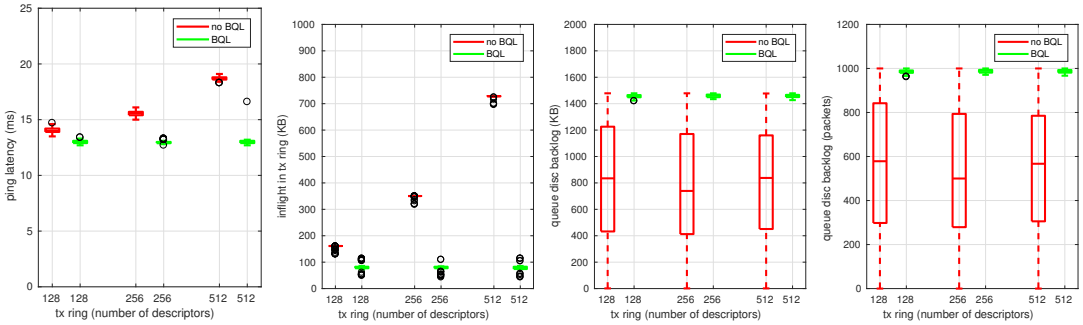
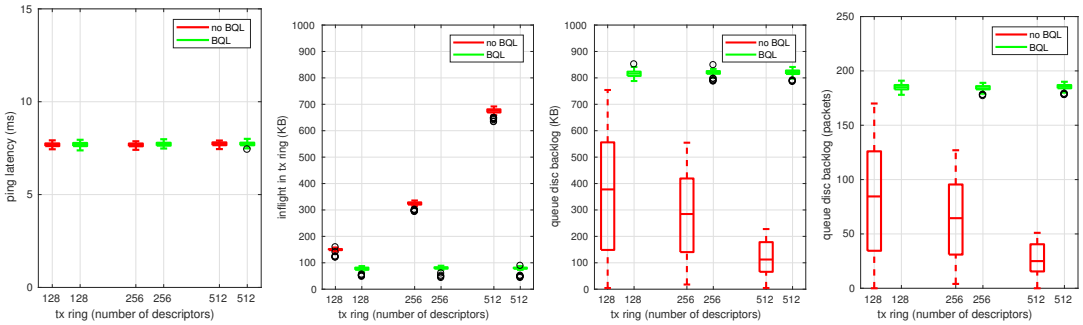


FIGURE 2.3: Throughput loss with small transmission rings

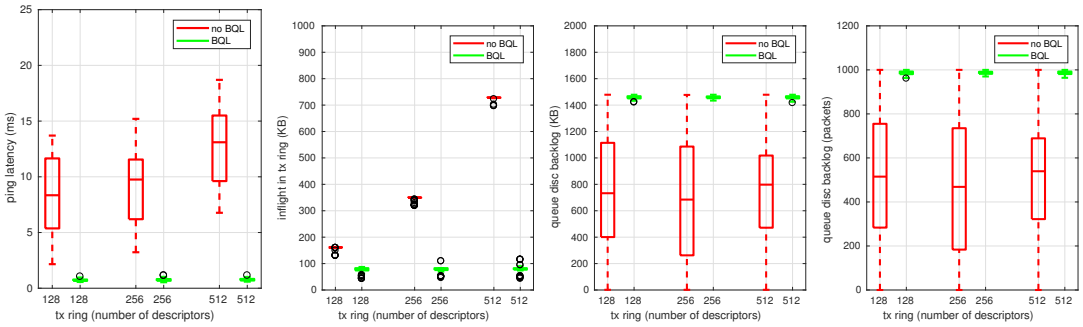


(A) Latency (B) TX ring occupancy (C) Queue disc backlog (bytes) (D) Queue disc backlog (pkts)

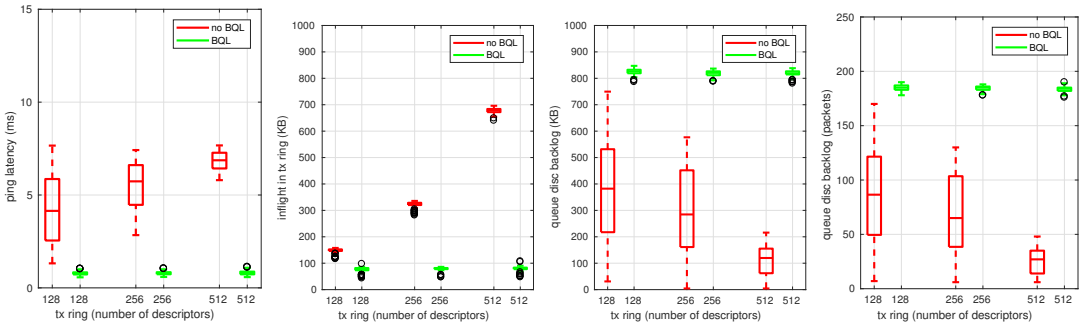


(E) Latency (F) TX ring occupancy (G) Queue disc backlog (bytes) (H) Queue disc backlog (pkts)

FIGURE 2.4: Results with a pfifo-fast queue disc. Figures on the first row (from (a) to (d)) show the results with TCP Small Queues and Segmentation Offload disabled (*router* scenario), while figures on the second row (from (e) to (h) show the results with TCP Small Queues and Segmentation Offload enabled (*host* scenario).

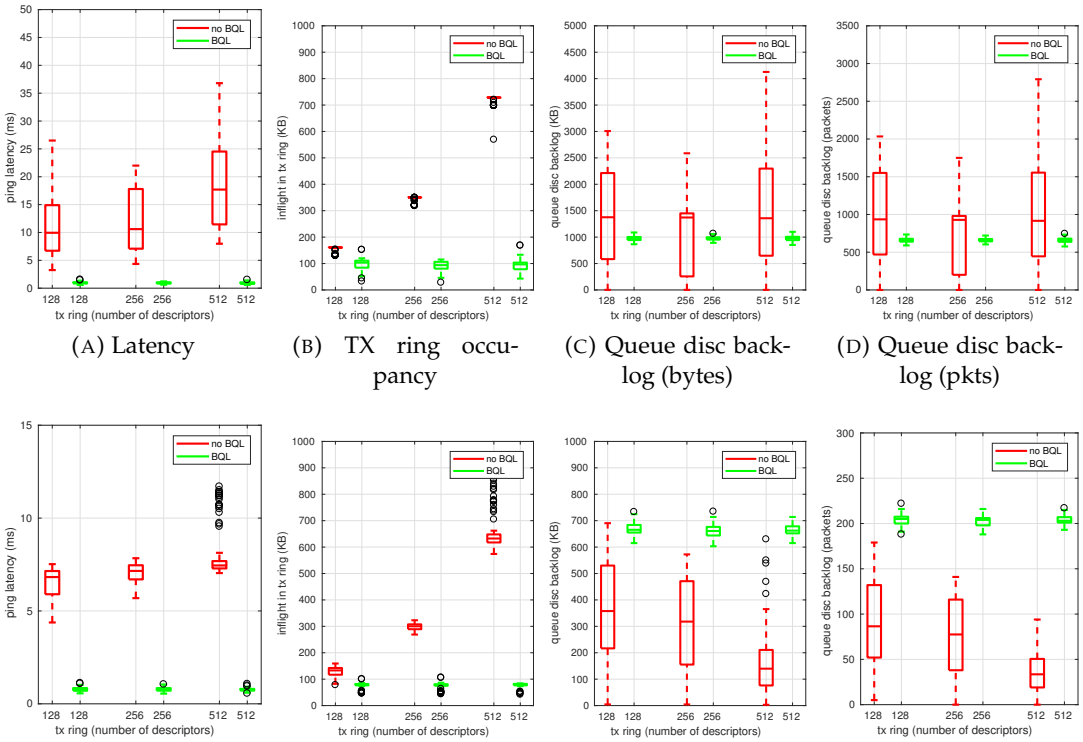


(A) Latency (B) TX ring occupancy (C) Queue disc backlog (bytes) (D) Queue disc backlog (pkts)



(E) Latency (F) TX ring occupancy (G) Queue disc backlog (bytes) (H) Queue disc backlog (pkts)

FIGURE 2.5: Results with a pfifo-fast queue disc and a prioritized flow. Figures on the first row (from (a) to (d)) show the results with TCP Small Queues and Segmentation Offload disabled (*router* scenario), while figures on the second row (from (e) to (h) show the results with TCP Small Queues and Segmentation Offload enabled (*host* scenario).



(A) Latency (B) TX ring occupancy (C) Queue disc backlog (bytes) (D) Queue disc backlog (pkts)

(E) Latency (F) TX ring occupancy (G) Queue disc backlog (bytes) (H) Queue disc backlog (pkts)

FIGURE 2.6: Results with an FQ-CoDel queue disc. Figures on the first row (from (a) to (d)) show the results with TCP Small Queues and Segmentation Offload disabled (*router* scenario), while figures on the second row (from (e) to (h) show the results with TCP Small Queues and Segmentation Offload enabled (*host* scenario).

2.4 Experimental results

We used two desktop computers running the Linux 4.11 kernel and equipped each with an Intel 1Gbps Ethernet network interface card using the Linux e1000 driver. The default size for the transmission ring, as set by the driver, is 256 descriptors. The two computers are connected back to back with an Ethernet cross cable. One of the computers acts as traffic sender, the other as traffic receiver. Netperf is used to generate network traffic. In the following, we present the insights that we gained by running a thorough experimental campaign aiming at assessing the impact that various configurations of the network stack have on network performance. In all the experiments, the configuration is only changed on the sender node, while the receiver node is left with the default configuration. We considered two scenarios, one where TCP Small Queues (TSQ) and Segmentation Offload (TSO) are both disabled and another one where TSQ and TSO are both enabled. The former aims to reproduce the case of an intermediate bottleneck router, while the latter aims to reproduce the case of a host, e.g., a server.

A small transmission ring causes a throughput loss.

We generated 100 simultaneous TCP flows by concurrently running 100 TCP_STREAM tests in order to saturate the link capacity despite the limit imposed by TCP Small Queues (both TSQ and TSO were enabled). Experiments were conducted with three distinct transmission ring sizes: 64, 96 and 128 descriptors. The default pfifo_fast queue disc was used. The duration of each experiment was set to 30 seconds and the average throughput over the whole duration of the experiment was measured. Each experiment was repeated 5 times. Results of these experiments (Fig. 2.3) clearly show a considerable throughput loss when the size of the transmission ring is too small. As stated earlier, it has to be noted that a single packet may require multiple descriptors, especially when scatter/gather I/O and TSO are enabled, thus a transmission ring with 64 descriptors may contain as few as 4 packets. The throughput loss is likely due to the starvation of the network device, which occurs when the device is ready to receive packets from the device driver but the transmission ring is entirely occupied by packets that have been transferred to the device but have not been freed yet. We recall that packets stay in the transmission ring until the CPU is able to serve the interrupt request generated by the network device (or is able to poll the device). Thus, the probability of starvation increases with smaller transmission rings and is exacerbated by the adoption of interrupt mitigation techniques. The outcome of these experiments is therefore that the transmission ring should be sufficiently large to avoid incurring throughput losses.

With the default pfifo_fast queue disc, latency is unaffected by the transmission ring size or by the adoption of BQL if the queue disc is not full.

In addition to the 100 one-way TCP flows used to saturate the link capacity, an ICMP Echo Request message is generated every 0.5 seconds by using the ping application. These messages were used as probes to measure the latency experienced by packets. We considered sufficiently large transmission rings (128, 256 and 512 descriptors) to avoid incurring throughput losses. In fact, the throughput achieved in all the experiments was around 890 Mbps. The default pfifo_fast queue disc was used. Each experiment lasted 30 seconds and was repeated 5 times. For each experiment, we collected: *i*) the round trip time of every ICMP Echo Request/Reply exchange; *ii*) the amount of bytes in-flight, i.e. queued in the transmission ring, at the sender, every 0.5 seconds; and *iii*) the backlog of the queue disc, in terms of both bytes and packets, every 0.5 seconds. Both in the router scenario (Fig. 2.4a) and host scenario (Fig. 2.4e), the latency is not affected by the transmission ring size when BQL is enabled. This result is rather expected, given that BQL autonomously limits the amount of bytes in the transmission ring and thus is unaffected by the transmission ring size (provided that the latter is sufficiently large). Two other observations can be made by looking at Fig. 2.4e (host scenario): *i*) when BQL is disabled, the latency is not affected by the transmission ring size, too; *ii*) the latency experienced with or without BQL is the same. These two results are less intuitive and have a common explanation: given that the pfifo_fast queue disc does not drop packets if there is room in its queues, the number of packets queued in the whole system (queue disc and transmission ring) is the same in all the cases. Indeed, as Fig. 2.4h shows, there are no more than 200 packets in the queue disc, which is therefore not full (by default, the capacity of pfifo_fast is 1000 packets). Such a limit can be explained by considering that packets are generated by 100 processes, each of which is allowed by TCP Small Queues to have about two packets in flight (i.e., queued in the queue disc or in the transmission ring). We can generalize this result by stating that BQL has no effect on the packet latency if the queue disc enqueues all the packets in the same queue and is not full. In this scenario, despite the latency is the same with or without BQL, there is a difference in how packets are distributed between the queue disc and the transmission ring, as shown by Figs. 2.4f and 2.4g. It can be seen that BQL keeps both the queue disc backlog and the amount of bytes in the transmission ring constant. With BQL disabled, instead, the amount of bytes in the transmission ring grows with the ring size, while the queue disc backlog decreases. Thus, BQL has no effect on the packet latency, but helps keep the amount of bytes in the transmission ring low and the queue disc backlog high, independently of the transmission ring size. We will highlight hereinafter the benefits brought by BQL thanks to such a behavior.

In the router scenario, where TSQ is disabled, the queue disc is filled up to its capacity in all the cases (Fig. 2.4d). Hence, differences in latency (Fig. 2.4a) are due to the different amount of bytes queued in the transmission ring (Fig. 2.4b). Therefore, latency grows with the transmission ring size when BQL is disabled. Instead, BQL

limits the amount of bytes queued in the transmission ring and therefore allows to reduce the latency.

Measuring the backlog in terms of both bytes and packets allows us to determine the average packet size. In the router scenario, the average packet size is 1500 bytes (equal to the Ethernet MTU), while in the host scenario it is about 4 KB (it is allowed to be greater than the MTU because TCP Segmentation Offload is enabled). Despite the different average packet size, we observe that the transmission ring occupancy is similar in the router and host scenarios (Figs. 2.4b and 2.4f). The reason is likely that the big packets queued in the home scenario take more descriptors than the regular packets in the router scenario.

BQL allows to keep the latency of prioritized flows low; without BQL, the latency is higher and affected by the transmission ring size.

The pfifo_fast queue disc offers three priority bands. We considered the same settings as in the previous experiment, except that the ICMP Echo Request messages used as probes to measure latency carry the Assured Forwarding AF42 DiffServ Code Point (DSCP), so that they are enqueued in the highest priority band (band 0) of the pfifo_fast queue disc. The 100 one-way TCP flows used to saturate the link capacity have a normal priority and are enqueued in band 1 of the queue disc. Given that the traffic load and the queue disc are the same as in the previous experiment, we get the same results in terms of throughput (not shown), queue disc backlog and transmission ring occupancy, for both the router and host scenarios. Different results in terms of latency are instead obtained, due to the fact that probe packets have a higher priority in this experiment. We can observe (Figs. 2.5a and 2.5e) that BQL allows to keep the latency of the prioritized packets very low (about 1 ms), while in the absence of BQL the latency is significantly higher (median values range from 4 to 13 ms) and is affected by the transmission ring size. This result can be explained by considering that probe packets experience no delay in the queue disc because only probe packets are enqueued in the highest priority band and their rate is rather low (2 messages per second). Hence, probe packets only accumulate delay in the transmission ring. Without BQL, the amount of bytes queued in the transmission ring increases with the transmission ring size (Figs. 2.5b and 2.5f), and so does the latency. Instead, BQL limits the amount of bytes queued in the transmission ring and therefore the latency is lower than without BQL.

BQL allows AQM algorithms to bring the expected benefit, i.e., to reduce latency; without BQL, the latency is higher and grows with the transmission ring size.

As in the previous experiments, we considered 100 one-way TCP flows and a round-trip probe flow (with the same priority as the TCP flows). However, in order to assess the impact of BQL on the effectiveness of AQM algorithms, we installed an FQ-CoDel queue disc instead of the default pfifo_fast queue disc. FQ-CoDel attempts to direct flows to distinct queues, which are served in a round robin-like fashion in order to ensure fairness among flows. Each queue is managed according to the CoDel algorithm, which aims to limit the latency experienced by packets by dropping packets whose sojourn time within the queue is above a certain threshold. Given that, by default, FQ-CoDel creates 1024 queues, it is very likely that each of the flows we generate, including the ICMP Echo Request messages, is enqueued in a distinct queue. Moreover, the data rate of probe packets (about 1.5 Kbps) is likely such that the flow of probe packets is always treated as a "new" flow by FQ-CoDel and hence gets priority over the other flows. Therefore, again, probe packets experience little to no delay in the queue disc, while they accumulate delay in the transmission ring. As shown by Figs. 2.6a and 2.6e, BQL allows to keep the latency very low (about 1 ms), while in the absence of BQL the latency is significantly higher (median values range from 7 to 17 ms) and grows with the transmission ring size. This result can be again explained by considering that BQL keeps the amount of bytes queued in the transmission ring rather low, while the amount of in-flight bytes grows with the transmission ring size when BQL is disabled (Figs. 2.6b and 2.6f). Consequently, when BQL is enabled, AQM algorithms work well because little delay is added to the time spent in the queue disc, which AQM algorithms aim to control. Without BQL, instead, packets experience an additional amount of time, which grows with the transmission ring size, after they have been dequeued by the queue disc. Such an additional delay, therefore, ends up eliminating the benefits brought by AQM algorithms. Finally, we mention that the measured throughput (not shown) is around 890 Mbps in all the experiments.

BQL allows to keep the queue disc backlog, and hence the latency, very stable, thus reducing the jitter.

By observing the figures reporting the queue disc backlog (in terms of both bytes and packets) for all of the previous experiments, it can be noted that the queue disc backlog is rather stable when BQL is enabled, while it widely oscillates (between a few kilobytes and hundreds or thousands of kilobytes) when BQL is disabled. These results, along with the observation that the transmission ring occupancy is very stable (with or without BQL), suggest that, when BQL is enabled, CPU cycles are (approximately) equally devoted to enqueuing and dequeuing packets from the queue disc over short time intervals. Instead, when BQL is disabled, CPU cycles are

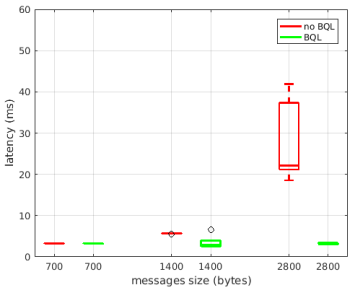


FIGURE 2.7: Latency variation with message size

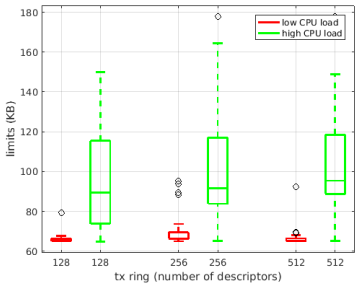


FIGURE 2.8: Impact of CPU load on the BQL limit

no longer equally shared over short time intervals (we measure the backlog every 0.5 seconds). When the CPU cycles devoted to dequeuing (enqueuing) packets prevail, the backlog decreases (increases). Such a behavior, observed when BQL is disabled, might be due to the network device driver aggressively waking up the queue disc as soon as some space is freed in the transmission ring when the transmission queue is stopped (which is often the case because the transmission ring is filled up to its capacity when BQL is disabled). As we described earlier, waking up a queue disc means that the queue disc is added to the list of queue discs that will be served the next time the software interrupt is executed. Serving a queue disc means requesting it to dequeue (at most) a predefined number of packets. Therefore, waking up the queue disc repeatedly may lead to a situation where more packets are dequeued than enqueued, thus causing the observed decrease in the queue disc backlog.

The explanation above is supported by the analysis of the measured latency. We consider, as an example, the case of pfifo_fast with high priority probe packets and transmission ring size equal to 512, in the router scenario. The amount of bytes queued in the transmission ring is rather stable (about 725 KB, Fig. 2.5b), hence a packet entering the transmission ring waits in the queue for about 725 KB / 890 Mbps = 6.5 ms. However, Fig. 2.5a shows that some probe packets experience larger delays (the 75th percentile is about 19 ms), despite probe packets have high priority and therefore do not stand in the queue disc. A possible explanation is that probe packets accumulate delay before being enqueued in the queue disc, due to the CPU being busy dequeuing packets. Thus, probe packets that manage to be quickly enqueued in the queue disc experience a latency close to the waiting time in the transmission ring (the 25th percentile of the latency is around 7ms). Other probe packets wait to be enqueued in the queue disc for a longer time (up to the time needed to dequeue all the packets). Assuming that a probe packet arrives when the queue disc backlog is 1500KB (i.e., the 75th percentile according to Fig. 2.5c) and the enqueue operation is deferred until the queue disc is empty, the probe packet would wait for 1500KB / 890 Mbps = 13 ms in order to be enqueued in the queue disc and 6.5 ms in the transmission ring, for a total of 19.5 ms (which is approximately the 75th percentile of the latency).

BQL allows to keep the latency independent of the packet size; without BQL, latency increases with the packet size.

We considered one round-trip high priority TCP flow used as a probe to measure packet latencies and 200 one-way TCP flows with variable message size (700, 1400 and 2800 bytes) to saturate the link capacity. We used the default pfifo_fast queue disc and the default size for the transmission ring (256 descriptors). Figure 2.7 shows that BQL is able to keep the latency experienced by packets constant, independently of the packet size. Without BQL, instead, latency increases with the packet size. This result can be explained by considering that the latency experienced by the packets of a prioritized flow is mainly due to the time spent in the transmission ring, as discussed earlier. Also, the transmission time for a packet is directly proportional to its size. Without BQL, every descriptor in the transmission ring can point to a (fragment of a) packet, thus the amount of bytes queued in the transmission ring (and hence the time required to transmit them) depends on the packet size (and to the adoption of techniques such as TSO and scatter/gather I/O). BQL, instead, enforces a limit on the amount of bytes queued in the transmission ring, which makes its performance independent of the size of the packets.

BQL adapts the amount of bytes stored in the transmission ring based on the current CPU load, too.

We repeated the previous experiments while keeping all the CPU cores busy (their utilization was close to 100%) with the compilation of ns-3. We measured the limit computed by BQL and compared it to that achieved during the previous experiments (when the CPU load was low). Figure 2.8 reports the results of such comparison when an FQ-CoDel queue disc is used (very similar results are obtained in the other cases). It can be observed that the limit computed by BQL is generally higher and more variable when the CPU load is high. This result is consistent with the fact that, when the CPU load is low, the software interrupt of type NET_TX_SOFTIRQ (which frees the packets transferred to the device and requests the queue disc to dequeue packets) is executed rather regularly and hence there is no need to store a large amount of bytes in the transmission ring. When the CPU load is high, instead, the software interrupt is executed less frequently, hence there is the need to store a larger amount of bytes in the transmission ring to avoid starvation.

2.5 Conclusions

In this chapter, we reported the results of a thorough experimental campaign we conducted to evaluate the performance of various schedulers, including a simple FIFO scheduler, a priority scheduler and an AQM algorithm, when DQL is enabled or disabled and with varying transmission ring size. Experimental results indicated

that DQL generally allows to reduce latency, which instead increases with the transmission ring size when DQL is disabled. Such a result is particularly important to preserve the effectiveness of AQM algorithms, which aim to control the latency experienced by packets. DQL does not enable a latency reduction when a queue disc that enqueues all the packets in a single queue is used below its capacity. Furthermore, latency is rather stable when DQL is used, while it oscillates widely when DQL is disabled.

Chapter 3

Design and implementation of the traffic control module in ns-3

The Linux networking subsystem relies on the traffic control infrastructure to process both the incoming and the outgoing packets. One of the most important components of the traffic control is the queueing discipline, whose role is to store packets waiting for transmission and select the next packet to pass to the network interface. The Linux traffic control enables to perform scheduling, shaping of the egress traffic, policing of the ingress traffic, and dropping of both ingress and egress traffic.

In this chapter, we present the design and implementation of the traffic control layer as an additional module in ns-3. This layer sits in between the netdevices and the network layer. We also present the design and implementation of the base class introduced to model a queueing discipline. Finally, we report a preliminary evaluation of our work, consisting in a number of tests that properly compare the new stack to the previous one.

3.1 Introduction

The Traffic Control infrastructure [1] of the Linux kernel enables to perform a number of actions on both outgoing packets, before they are handed to the netdevice for transmission, and incoming packets, before they are processed by the network layer protocols. In this work, we focus on the transmission path taken by packets. Once the output interface and the next hop for an outgoing packet have been selected, the packet is enqueued into a queueing discipline (queue disc), which determines how the packet will be treated. A number of queue discs have been implemented in the Linux kernel, including simple FIFO (First In First Out) schedulers, such as pfifo_fast, fair-queuing schedulers, such as Deficit Round Robin (DRR) [34], Stochastic Fairness Queuing (SFQ) [35] and Quick Fair Queuing (QFQ+) [36], and Active Queue Management (AQM) algorithms, such as Random Early Drop (RED) [3] and Controlled Delay (CoDel) [4]. A queue disc stores the packets waiting for transmission and decides which packet to pass to the network interface when it is requested to dequeue a packet.

When a queue disc is requested to dequeue a packet depends on the implemented flow control mechanisms. Basically, enqueuing a packet into a queue disc triggers a number of consecutive requests of dequeuing a packet. This process can be halted by the netdevice driver (or by the network stack itself) by putting its netdevice queue into a *stop* state. The netdevice driver usually stops its transmission queue when it is full or its occupancy is above a given threshold. When the netdevice is able to receive packets again, the driver can *start* its transmission queue. Additionally, the netdevice driver can *wake* a queue disc, i.e., request it to dequeue a packet, when its transmission queue is empty or its occupancy is below a given threshold.

Currently, ns-3 is lacking an equivalent of the Linux Traffic Control infrastructure. No flow control mechanism is implemented and packets are only stored in the netdevice transmission queues. Consequently, AQM algorithms such as RED and CoDel can only manage the packets stored in the netdevice queues, which is not what happens in Linux. This chapter presents the work done to introduce an equivalent of the Linux Traffic Control infrastructure into ns-3. We believe that our work will allow researchers to carry out more realistic simulations and to evaluate AQM algorithms more precisely.

The rest of this chapter is organised as follows. In section 2 we provide an overview of the Linux Traffic Control and of the current status of ns-3. Section 3 describes the model and design of the proposed Traffic Control module for ns-3. Section 4 describes the experiments we performed with the new architecture and the results we obtained. In section 5 we draw the conclusions.

3.2 Background

In this section, we describe the Linux Traffic Control infrastructure and the ns-3 queue system.

3.2.1 Linux traffic control

The Traffic Control is a component of the network subsystem of the Linux kernel. This component supports multiple operations needed to provide Quality of Service (QoS), including shaping and scheduling of egress traffic, policing of ingress traffic, dropping of both ingress and egress traffic.

The Traffic Control relies on three fundamental components to perform the above mentioned operations, i) queue discs, ii) classes, iii) filters. A queue disc can be added to every network interface in the Linux kernel and determines how packets outgoing from the interface are treated. A simple queue disc is fifo, which does no processing and is a pure FIFO queue with queue limit expressed in packets or bytes. It stores a packet when the network interface cannot handle it immediately. The default queue disc in Linux is pfifo_fast, which consists of a three band queue acting

as a priority queue. The priority assigned to each packet may depend on the Type of Service (ToS) value or Diffserv Codepoint (DSCP) carried by the packet. More complex queue discs are available in the Traffic Control component. A typical taxonomy divides queue discs in classful (i.e., support classes) and classless (i.e., do not support classes). A classful queue disc can contain multiple classes, each of which has a child queue disc attached. Each class can be configured with distinct parameter values, so as to reserve a distinct treatment to different traffic classes. For instance, the prio queue disc is a container for a configurable number of classes which are served in order of priority. A packet filter can be used by a classful queue disc to classify packets based on different criteria. The most advanced filter available is u32 that can use anything in the header for classification. Recently, after the appearance of multi-queue netdevices (such as Wifi), some multi-queue aware queue discs have been introduced. Multi-queue aware queue discs handle as many queues (or queue discs – without using classes) as the number of transmission queues used by the netdevice on which the queue disc is installed. An attempt is made, also, to enqueue each packet in the "same" queue both within the queue disc and within the netdevice.

3.2.2 ns-3 queue system

In this section, we analyze ns-3 for what concerns the queuing system adopted at the network and netdevice layers.

The network layer has no queuing system. In case the IPv4 stack is employed (the same holds for the IPv6 stack), packets generated from the upper layers are passed to the Ipv4L3Protocol, which determines the right Ipv4Interface for packet forwarding. Then, the Ipv4Interface passes the packet to the corresponding netdevice. Thus, it is not possible to differentiate traffic at this layer and hold packets whose transmission to the netdevice failed (and are therefore dropped). Netdevices store the packets waiting for transmission in a queue. Such a queue contains packets with the data link header already added and is modelled through the base class Queue. Derived classes are DropTailQueue, RedQueue and CoDelQueue. DropTailQueue is a classical first in first out limited queue. The two models of AQM, RedQueue and CoDelQueue, both presently derive from class Queue and can be only installed on the Csma and PointToPoint netdevices but not on Wifi and LTE netdevices. The reason is that Wifi and LTE do not use subclasses of the base class Queue to implement their netdevice queues.

The ns-3 netdevices do not support any form of flow control between the network and netdevice layers. Indeed, netdevices have no means to request the network layer to stop sending packets and all the packets passed to the netdevices when their queues are full are inevitably lost.

The base class Queue does not provide easy visibility of the IP and transport headers. The non trivial access or modification of the IP header have hindered the addition of the Explicit Congestion Notification (ECN) support in ns-3 [14]. The ns-3

ECN support should remove the L2 header (of different length for a different netdevice) then remove the L3 header and apply the ECN policy. Also, the non simple access to the transport header, e.g. TCP, has hindered the ns-3 support to the recent internet aware queue discs such as FlowQueue-CoDel (FQ-CoDel). Those queue discs need access to the 5-tuple of IP protocol, source and destination IP addresses and port numbers.

3.3 Design and implementation

In this section, we describe the traffic control module and its design.

3.3.1 Module description

In order to add support for the features described in the previous section, we decided to introduce a new layer that sits above the netdevice and below the IP forwarding layer. The main consequence is that it requires flow control between the new layer and each of the netdevice queues. For each netdevice queue, it is necessary to keep a status bit which indicates if further packets can be passed to the netdevice for the transmission. The netdevice (or the network layer) can stop the passing of further packets when a resource becomes unavailable (e.g. the netdevice queue is full) and wake up the upper layer when the resource becomes available again.

Packets received by the traffic control layer for transmission to a netdevice can be passed to a queue disc to perform scheduling and policing. A netdevice can have a single (root) queue disc installed on it. Installing a queue disc on a netdevice is not mandatory. If a netdevice does not have a queue disc installed on it, the traffic control layer sends the packets directly to the netdevice.

As in Linux, a queue disc may contain distinct elements:

- queues, which actually store the packets waiting for transmission;
- classes, which allow to reserve a different treatment to different packets;
- filters, which determine the queue or class which a packet is destined to.

Notice that a child queue disc must be attached to every class and a packet filter is only able to classify packets of a single protocol.

The traffic control layer interacts with a queue disc in a simple manner: after requesting to enqueue a packet, the traffic control layer requests the queue disc to "run", i.e., to dequeue a set of packets, until a predefined number ("quota") of packets is dequeued or the netdevice stops the queue disc. A netdevice may stop the queue disc when its transmission queue(s) is/are (almost) full. Also, a netdevice may wake the queue disc when its transmission queue(s) is/are (almost) empty. Waking a queue disc is equivalent to make it run.

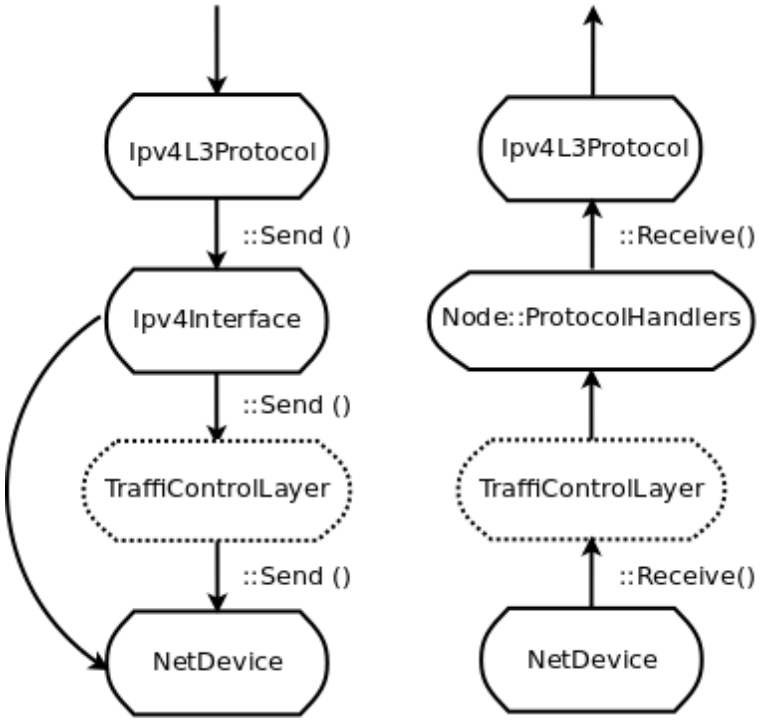


FIGURE 3.1: The send and receive path on internet enabled nodes after the introduction of the traffic control layer (IPv4 case).

3.3.2 Design

The new internet enabled node stack with Traffic Control is illustrated in Figure 3.1 (for the IPv4 case, the IPv6 case is similar). A TrafficControlLayer object is aggregated to every internet enabled node. The new layer intercepts packets that transit both in the input and output directions. Currently, scheduling of outgoing packets is supported, while policing of incoming packets is not supported (since the equivalent of the Linux ingress queue disc has not been implemented yet). The IPv{4,6} interfaces uses the aggregated TrafficControlLayer object to send packets down, instead of calling NetDevice::Send() directly. After the analysis and the process of the packet, when the flow control mechanism allows it, TrafficControlLayer will call the Send() method on the right netdevice. The IPv{4,6} interfaces call the NetDevice::Send() directly only in the case of packets destined to the loopback interface. To receive packets, instead, the callback chain, that (in the past) involved the node protocol handlers and the netdevice, is extended to involve TrafficControlLayer.

A TrafficControlLayer object holds a reference (smart pointer) to the objects representing the queue discs installed on each netdevice of the node. An abstract base class, class QueueDisc, is subclassed to implement specific queue discs. A subclass is required to implement the following methods:

- bool DoEnqueue (Ptr<QueueDiscItem> item): enqueue a packet;
- Ptr<QueueDiscItem> DoDequeue (void): dequeue a packet;
- Ptr<const QueueDiscItem> DoPeek (void) const: peek a packet;

- bool CheckConfig (void) const: check if the configuration is correct.

The base class QueueDisc implements:

- methods to add/get a single queue, class or filter and methods to get the number of installed queues, classes or filters;
- a Classify method which classifies a packet by processing the list of filters until a filter able to classify the packet is found;
- methods to extract multiple packets from the queue disc, while handling transmission (to the netdevice) failures by requeuing packets.

The base class QueueDisc holds the list of attached queues, classes and filter by means of three vectors accessible through attributes (InternalQueueList, QueueDiscClassList and PacketFilterList).

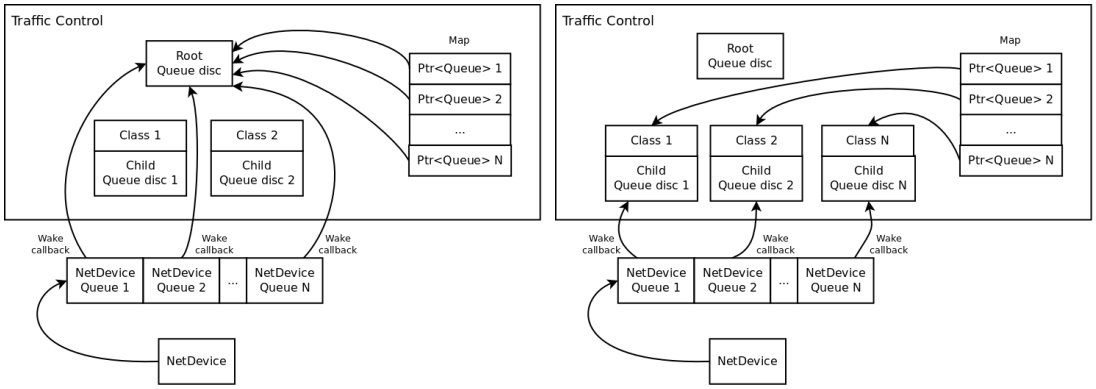
Internal queues are implemented as (subclasses of) Queue objects. A Queue stores QueueItem objects, which consist of just a Ptr<Packet>. Since a queue disc has to store at least the destination address and the protocol number for each enqueued packet, a new class, QueueDiscItem, is derived from QueueItem to store such additional information for each packet. Thus, internal queues are implemented as Queue objects storing QueueDiscItem objects. Also, there could be the need to store further information depending on the network layer protocol of the packet. For instance, for IPv4 and IPv6 packets it is needed to separately store the header and the payload, so that header fields can be manipulated, e.g., to support ECN. To this end, Ipv4QueueDiscItem and Ipv6QueueDiscItem are derived from QueueDiscItem to additionally store the packet header and provide protocol specific operations such as ECN marking.

Classes are implemented via the QueueDiscClass class,
which just consists of a pointer to the attached queue disc. Such a pointer is accessible through the queue disc attribute. Classful queue discs needing to set parameters for their classes can subclass QueueDiscClass and add the required parameters as attributes.

An abstract base class, PacketFilter, is subclassed to implement specific filters. Subclasses are required to implement two virtual private pure methods:

- bool CheckProtocol (Ptr<QueueDiscItem> item) const:
  check whether the filter is able to classify packets of the same protocol as the given packet;
- int32_t DoClassify (Ptr<QueueDiscItem> item) const:
  actually classify the packet.

PacketFilter provides a public method, Classify, which first calls CheckProtocol to check that the protocol of the packet matches the protocol of the filter and then calls DoClassify. Specific filters subclassed from PacketFilter should not be placed



(A) Classful queue disc (B) Multi-queue aware queue disc

FIGURE 3.2: Queue discs in Traffic Control.

in the Traffic Control module but in the module corresponding to the protocol of the classified packets.

In Linux, information about the status of a transmission queue of a netdevice is stored in the struct netdev_queue, which includes a qdisc field that is mainly used to solve the following problems:

- if a netdevice transmission queue is (almost) empty, identify the queue disc to wake;
- if a packet will be enqueued in a given netdevice transmission queue, identify the queue disc which the packet must be enqueued into.

The latter problem arises because Linux attempts to determine the netdevice transmission queue which a packet will be enqueued into before passing the packet to a queue disc. This is done by calling a specific function of the netdevice driver, if implemented, or by employing fallback mechanisms (such as hashing of the addresses) otherwise. The identifier of the selected netdevice transmission queue is stored in the queue_mapping field of the struct sk_buff, so that both the queue disc and the netdevice driver can get the same information. In ns-3, such identifier is stored in the m_txq member of the QueueDiscItem class.

Concerning the qdisc field of the struct netdev_queue in Linux, such a field cannot be similarly stored in a object NetDeviceQueue, because it would make the network module depend on the Traffic Control module. Instead, this information is stored in the TrafficControlLayer object aggregated to each node. In particular, a TrafficControlLayer object holds a map which stores, for each netdevice, a vector of Ptr<QueueDisc>. The size of such a vector is the number of netdevice transmission queues and each element of this vector is a pointer to the queue disc to activate when the above problems occur. The SetRootQueueDiscOnDevice method takes care of configuring such a map, based on the wake mode of the root queue disc. If the wake mode of the root queue disc is WAKE_ROOT, then all the elements of the vector are pointers to the root queue disc. If the wake mode of the root queue

disc is WAKE_CHILD, then each element of the vector is a pointer to a distinct child queue disc. This requires that the number of child queue discs matches the number of netdevice queues. It follows that the wake mode of a classless queue disc must necessarily be WAKE_ROOT. These two configurations are illustrated in Figure 3.2.

Finally, we mention that the queue disc installed on a netdevice, along with the associated packet filters, classes and internal queues, can be removed by calling the method DeleteRootQueueDiscOnDevice of the TrafficControlLayer class.

3.3.3 Implementation Issues

The Ipv{4,6} Interface add the IP header to the packet before passing the packet to the underlying layer. Receving a packet with the IP header already attached makes it inefficient for the Traffic Control layer to manipulate the header, e.g., to perform ECN markings. For this reason, we changed the behavior of the internet stack so that the IP header and the IP payload of a packet are sent separately to the Traffic Control layer. This required modifications both to IPv4 (L3 protocol, ARP cache, ARP L3 protocol) and IPv6 (L3 protocol, extensions, ICMPv6, NDisc cache). The IP header is now added to the packet after the packet is dequeued from the queue disc.

The Traffic Control module cannot depend on the internet module, in order to avoid that future, alternative to internet, L3 modules have to depend on internet (through the dependency on Traffic Control) and to avoid a circular dependency (given that internet depends on Traffic Control). As a consequence, the Traffic Control layer cannot manipulate IP headers, which is necessary, e.g., to perform ECN marking, or filter packets based on the content of the IP header. As described earlier, this problem has been solved by enqueuing packets as (pointer to) QueueDiscItem objects which are actually either Ipv4QueueDiscItem or Ipv6QueueDiscItem objects. Likewise, using an abstract PacketFilter class allowed us to define protocol specific packet filters in the respective modules instead of in the Traffic Control module.

Other minor issues needed to be addressed. For instance, incorrect packet drops may be traced because the queue discs requeues packets whose transmission to the netdevice failed. Thus, if a netdevice drops a packet because, e.g., its queue is full, such a packet is traced as lost while it is actually requeued by the queue disc and retransmitted as soon as the netdevice is ready to receive packets again. A workaround for this issue is to compute the number of packets that have been actually dropped as the difference between the number of dropped packets as reported by the netdevice drop trace and the number of requeued packets.

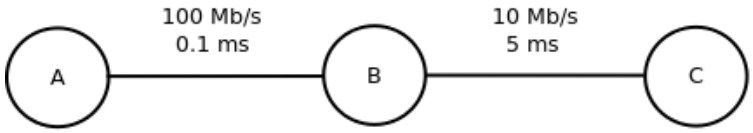
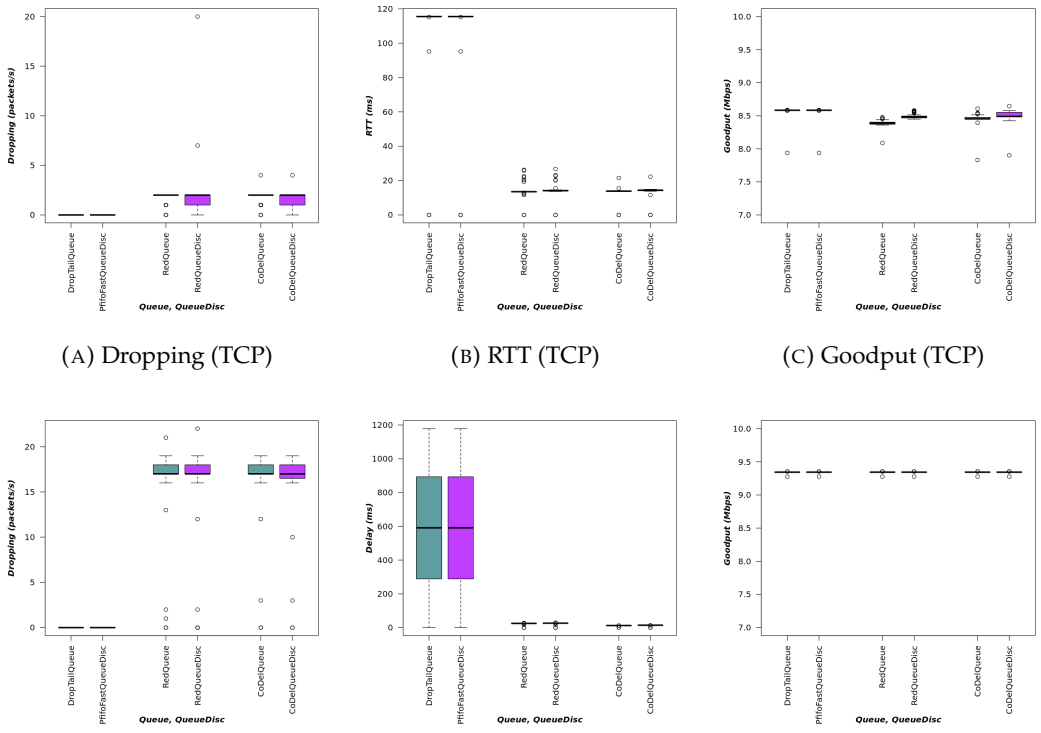


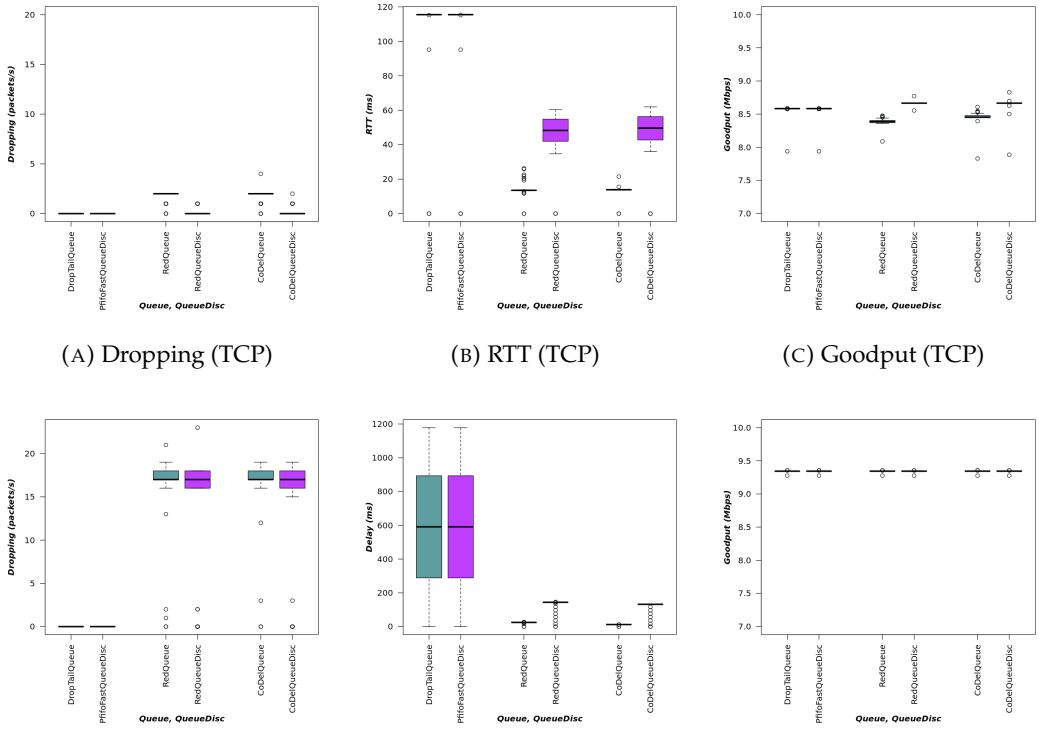
FIGURE 3.3: The network topology used for the validation tests.



(A) Dropping (TCP) (B) RTT (TCP) (C) Goodput (TCP)

(D) Dropping (UDP) (E) RTT (UDP) (F) Goodput (UDP)

FIGURE 3.4: Plots of the first scenario.



(A) Dropping (TCP) (B) RTT (TCP) (C) Goodput (TCP)

(D) Dropping (UDP) (E) RTT (UDP) (F) Goodput (UDP)

FIGURE 3.5: Plots of the second scenario.

3.4 Results

3.4.1 Simulation Settings

For all of the experiments described hereinafter, the simple three node topology reported in Figure 3.3 was used. Nodes A and B are connected by means of a point-to-point link having a data rate of 100 Mb/s and a delay of 0.1 ms. Nodes B and C are connected by means of a point-to-point bottleneck link having a data rate of 10 Mb/s and a delay of 5 ms. Two scenarios have been considered, each of which compares the current ns-3 stack with the new stack featuring the Traffic Control layer:

- the first scenario aims, to some extent, to validate the proposed Traffic Control layer by comparing the results obtained with the current and the new stacks in similar configurations. In particular, the current stack is evaluated by using netdevice queues having a size of 1000 packets, while the new stack is evaluated by using queue discs having a size of 1000 packets and netdevice queues having a size of 1 packet;

- the second scenario aims to highlight that queuing at the netdevice layer has a non negligible impact on the performance of AQM algorithms like RED and CoDel and that such an effect cannot be observed with the current ns-3 stack. The current stack is evaluated by using netdevice queues having a size of 100

packets, while the new stack is evaluated by using queue discs having a size of 1000 packets and netdevice queues having a size of 100 packets.

An OnOff traffic generator is installed on node A, while a packet sink is installed on node C. The OnOff data rate is 100 Mb/s in the TCP simulations and 10 Mb/s in the UDP simulations. The TCP version is New Reno. The packets size is 1458 bytes. The generated traffic is not marked with any QoS information. Three configurations are compared. For the current stack, we consider DropTail, RED and CoDel as the types of netdevice queues. For the new stack, we consider PfifoFast, RED and CoDel as the types of queue discs and DropTail as the type of netdevice queues. RED and CoDel are configured with the same parameter values when comparing the current and the new stack. RED is configured by setting the LinkBandwidth and LinkDelay attributes to the corresponding values of the bottleneck link, MeanPacketSize to the packet size, MinTh to 5 packets, MaxTh to 15 packets, the Gentle parameter to true. CoDel is configured by setting Interval to 100 ms and Target to 5 ms.

3.4.2 First Scenario

To evaluate the effects of the introduction of the Traffic Control module, the current stack (with a netdevice queue size of 1000 packets) is compared to the new stack (with a netdevice queue size of 1 packet and a queue disc size of 1000 packets). When evaluating the new stack, the netdevice queue size is set to 1 packet in order to reduce the netdevice queueing delay, which is outside the control of the AQM algorithm, while focusing on the effectiveness of the AQM algorithm and the interactions between the Traffic Control layer and the netdevice. Note that this is equivalent to turn off the hardware offload feature and set the kernel Byte Queue Limits (BQL) to a maximum of one packet in a real system [15]. This case, occurring in a real system, could not be currently modeled in ns-3.

The results are reported in Figure 3.4. As noted below, when evaluating the new stack, the queue discs take advantage of the opportunity to deliver two packets to the netdevices, one immediately transmitted and another queued in netdevice queue.

In the case of TCP, the presence of the netdevice queue leads to a minor dropping activity in the queue discs in the cases of RED and CoDel (Figure 3.4a). We also noticed that the time elapsed between two consecutive packet droppings is higher when using the new stack and the difference between these values for the current and the new stack grows with the progress of the simulation. The dropping appears smoother in the new stack. Also, the Round Trip Time (RTT), which takes into account the netdevice queueing delay, is slightly greater when adding the Traffic Control layer, in the cases of RED and CoDel (Figure 3.4b). The minor dropping activity with the Traffic Control layer enables to achieve higher goodput than the current stack, in the cases of RED and CoDel (Figure 3.4c). When using PfifoFast/Droptail, there is no noticeable difference between the current and the new stacks.

In the case of UDP, the presence of the netdevice queue being able to accommodate one additional packet makes no difference, because, contrarily to TCP, UDP does not adapt its sending rate based on the estimated RTT. The dropping activity remains substantially unchanged in all three cases (Figure 3.4d). We note that no dropping activity occurs due to the netdevice queue (current stack) or the queue disc (new stack) being full, because the dropping is null in the PfifoFast/DropTail case. The delay also remains substantially unchanged in all three cases (Figure 3.4e). When using RED, the delay remains unchanged and is equal to about 20 ms; when using CoDel, the delay remains unchanged, too, and is equal to about 10 ms. The goodput remains constant in all three cases (Figure 3.4f).

The obtained results show that the new stack, in this scenario, behaves very similarly to the current one.

3.4.3 Second Scenario

The second scenario aims to evaluate the effectiveness of some AQM algorithms in the common scenario in which a netdevice queue introduces a non negligible delay. The queue management algorithm is unaware of the time spent in the underlying netdevice queue, which can limit the effectiveness of the AQM algorithm. This is the case of all the netdevices for which BQL is not available and the sizing of the netdevice queue is difficult. For instance, this is the case of Wifi netdevices [15]. This case could not be evaluated in the current ns-3 stack.

The results are reported in Figure 3.5. In this case, the queue disc can send downwards 100 packets (queued in the netdevice queue) in addition to the packet being transmitted.

In the case of TCP, the netdevice queue limits the benefits of using an AQM algorithm. The dropping activity reflects the ability to deliver 100 packets to the netdevice. With the new stack, the dropping activity of the AQM algorithms (RED and CoDel) is reduced with respect to the current stack (Figure 3.5a). Consequently, the limited effectiveness of the AQM algorithms leads to a higher RTT, with respect to the current stack, in the cases of RED and CoDel (Figure 3.5b). In these cases, the RTT is about 50 ms. Given the reduced dropping compared to the current stack, the goodput improves and it is slightly greater than that achieved by PfifoFast/DropTail (Figure 3.5c). When using PfifoFast/Droptail, there is no noticeable difference between the current and the new stacks.

In the case of UDP, the dropping activity tends to be slightly less and more smooth with the new stack, in the cases of RED and CoDel, while remains unchanged in the PfifoFast/DropTail case (Figure 3.5d). We note that no dropping activity occurs due to the netdevice queue (current stack) or the queue disc (new stack) being full. The delay is affected by queueing in the netdevice (Figure 3.5e). In this scenario, the netdevice queue introduces a non negligible delay. With the new stack, the delay is about 140 ms and 130 ms when using RED and CoDel, respectively. The goodput remains substantially unchanged in all three cases (Figure 3.5f)

The results obtained, in this scenario, show a behavior which cannot be observed with the current ns-3 stack. Such behavior is encountered in real systems where the netdevice queue introduces a non negligible delay that limit the effectiveness of the AQM algorithms.

3.5 Conclusions

In this chapter, we presented the design, implementation and a preliminary evaluation of a traffic control module for ns-3. Our code has been integrated into ns-3 starting from the ns-3.25 release. We believe that our work will allow researchers to carry out more realistic simulations and to evaluate AQM schemes more precisely. One of the advantages of our traffic control infrastructure is that AQM schemes can now be tested on any netdevice, including, e.g., Wifi and LTE.

The proposed stack allows to introduce in ns-3 Internet aware schedulers such as FQ-CoDel. Indeed, a queue disc is now able to access to five tuple to separate the flows. Also, the isolation of the device queue allows to study the effect of its dimension on the effectiveness of packets schedulers and AQM algorithms. Finally, the device queue can be sized dynamically by using strategies such as BQL.

Chapter 4

Enanching the network emulation fidelity to support simulated modules validation

Researchers from academia, industry and research centers often resort to emulation to overcome the drawbacks associated with network simulation and experimental evaluation. Emulation is broadly classified in environment emulation, usually carried out by running real code in Virtual Machines (VMs) or containers, and network emulation, typically involving network simulators that exchange packets with the real world. In this chapter, we focus on network emulation, which is often exploited for rapid prototyping and testing of network protocols and algorithms. We identify the limitations of the approach currently used by various network simulators to provide network emulation and design an alternative solution based on netmap, a framework for high speed packet I/O which is available on multiple operating systems. We argue that the proposed solution to network emulation provides extremely accurate results in terms of packet latency and packet drops and prove our claim by means of an extensive experimental campaign. We also show that by building upon an accurate network emulation mechanism it is possible to validate the implementation of protocols found in network simulators against their implementation in real network stacks. We present the results of the experiments we conducted to validate the ns-3 implementation of various packet schedulers against their Linux counterpart.

4.1 Introduction

Researchers have multiple options to evaluate the performance of both newly designed and existing network protocols. Simulation is certainly a widely adopted instrument. In fact, simulation offers numerous benefits, including repeatability of tests, possibility to recreate scenarios with a large number of systems, ability to isolate the effects of undesired factors. However, simulators only provide reliable and realistic results if the network stack is accurately implemented and all the relevant

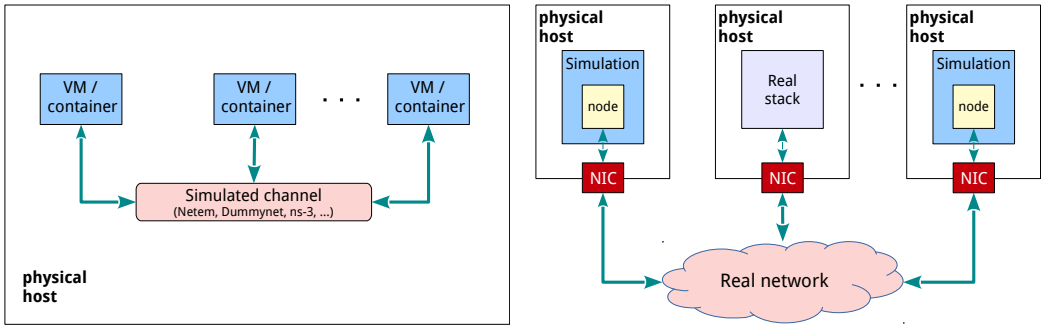


FIGURE 4.1: (a) In environment emulation, multiple Virtual Machines or containers running on a physical host communicate through a simulated channel. (b) In network emulation, nodes created within simulations can communicate, both between them and with real hosts, through a real network.

interactions among the various protocols are properly modeled. Contrarily, performing experiments in a real testbed allows to reliably test how a protocol will behave in the real world. However, setting up a testbed of meaningful size requires resources and expertise. A third option available to evaluate the performance of network protocols is emulation. Emulation can be seen as a hybrid approach, in the sense that it combines the usage of real implementations for some components with the simulation of some others. Depending on what components are simulated, emulation approaches can be broadly classified into two categories: *environment* emulation and *network* emulation [37].

In environment emulation, the real implementations of protocols at the higher layers of the network stack are used, while the channel access function and the transmission through a channel are simulated. Such an approach is usually implemented by having real code run in Virtual Machines (VMs) [38] or containers [39] and having them communicate through a simulated channel (Fig. 4.1a). Various tools can be used to simulate specific features of a transmission channel, including Netem [10], Dummynet [11], DEMU [40], network simulators supporting the injection of traffic from the real world (such as ns-3 [12] and OMNet++ [13]) and the mac80211_hwsim virtual driver, which is a Linux kernel module emulating transmissions over a Wi-Fi channel. Alternatively, some network simulators, including ns-3 (through the Direct Code Execution framework [41]) and NCTUns [42], offer the possibility of executing real code directly from within a simulation, thus eliminating the need of setting up VMs or containers. The environment emulation approach is typically used when the communication network is difficult to set up, either because the communication technology is not available [43] or because a testbed of the required size cannot be set up [44], or when it is desired to create a controllable environment eliminating the impact of external factors [45], [46].

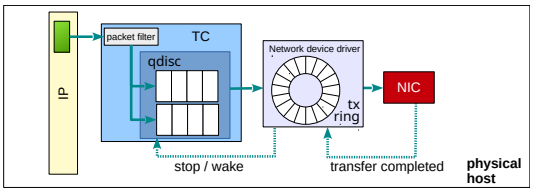
In network emulation, instead, simulated components interact with real hosts

through a real network (Fig. 4.1b). This approach exploits the ability of simulators such as ns-3 and OMNet++ to exchange packets with real network devices and schedule events in real-time. Network emulation requires that a real communication network is available. This requirement may also be met [47] by exploiting large scale testbeds such as ORBIT, Emulab and PlanetLab. When a real network infrastructure is available, network emulation may be preferred to real experiments in order to take advantage of the rapid prototyping enabled by simulators [9]. Implementing a new protocol is usually easier within a simulator rather than a real network stack, hence researchers often use network emulation to evaluate their proposals under real world conditions [48], [49]. Additionally, network emulation can be used to validate the implementation of protocols within a simulator against the implementation included in a real network stack.

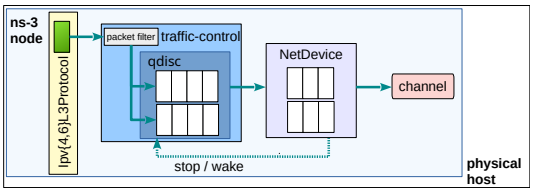
The work presented in this chapter is motivated by the limitations of the network emulation capabilities of simulators such as ns-3 and OMNeT++ in terms of accuracy of fundamental performance indicators such as packet loss and latency. Such limitations stem from the techniques used to exchange packets with real network devices. Network emulation in ns-3 is only available for Linux, because it relies on the packet socket mechanism, which is a feature specific to Linux. OMNeT++ uses instead *libpcap*, a C/C++ library for network traffic capture available for multiple operating systems. However, libpcap uses the packet socket mechanism on Linux systems, hence OMNeT++ ultimately exploits the same mechanism as ns-3 on Linux systems.

In this chapter, we focus on Linux systems and illustrate the relevant interactions between the network stack and the network device drivers in order to explain the drawbacks of using a packet socket to exchange packets with a network device. Then, we describe the design of a new ns-3 *NetDevice* aiming to overcome such drawbacks. The main reason for focusing on ns-3 is that it accurately models the interactions between the network stack and the network devices, thanks to the introduction of the *traffic-control* module that precisely reproduces the behavior of the *Traffic Control* (TC) infrastructure of the Linux kernel. The proposed ns-3 NetDeviceis based on the use of *netmap*, a framework for high speed packet I/O [50], and hence is named NetmapNetDevice. As a valuable side effect, our NetmapNetDevice enables ns-3 to gain network emulation support on all the operating systems supported by netmap, i.e., FreeBSD and Windows, in addition to Linux.

We present the results of experimental tests we conducted to demonstrate that the network emulation approach based on our NetmapNetDevice provides results in terms of throughput, latency and packet dropping rate that are extremely similar to those obtained by using the real Linux stack. Conversely, the current network emulation approach based on the packet socket mechanism fails to provide accurate results. Additionally, we show that the use of netmap allows to reduce the per-packet processing cost in terms of CPU usage, which opens up the possibility of increasing the throughput achievable in network emulation scenarios.



(A)



(B)

FIGURE 4.2: Schematic representation of the network stack of: (a) a Linux host equipped with a single network interface card; (b) an ns-3 node with one NetDevice using a simulated channel.

Another contribution in this chapter is the validation of various Active Queue Management (AQM) algorithms that are available as *queuing disciplines* both in Linux and ns-3 (RED [3], CoDel [4] and FQ-CoDel [6]), which, as shown hereinafter, is made possible by the availability of an accurate network emulation mechanism. Showing that ns-3 packet schedulers behave the same as their Linux counterpart is an important result, because it ensures researchers that the simulation results they get accurately reproduce real world results. In fact, packet schedulers heavily impact performance measures such as throughput, latency and packet loss.

In the remainder of this chapter, after providing some background information on the Linux TC infrastructure, the packet socket mechanism and netmap (Section 4.2), we present the design of the ns-3 NetDevice based on netmap (Section 4.3). The results of our experimental campaign are illustrated in Section 4.4. Finally, we conclude the chapter in Section 4.5.

4.2 Background

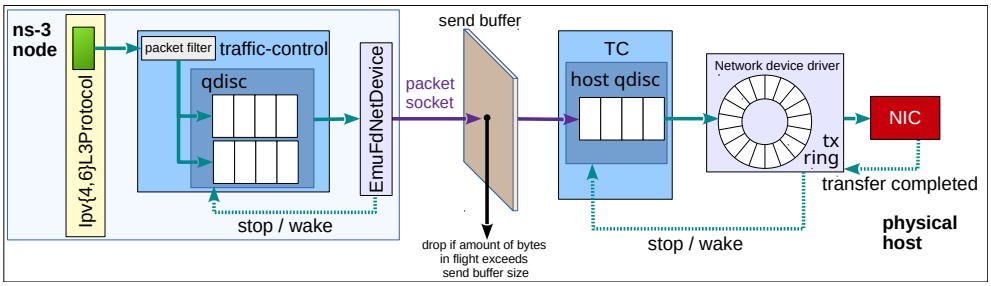
4.2.1 The Linux TC infrastructure and the ns-3 traffic-control module

We first provide some background information about the Linux TC infrastructure [1]. Based on the routes available in the routing table, the IP layer determines the outgoing network interface for each packet that needs to be sent over the network and passes it to the TC infrastructure (by calling the `dev_queue_xmit` function), where the packet is handled by the queuing discipline (henceforth, qdisc) associated with the outgoing device (Fig. 4.2a). A qdisc can enqueue the packet, mark the packet (e.g., in case it supports Explicit Congestion Notification) or drop the packet (either before the packet is enqueued or later, after the packet is dequeued). The qdisc is in charge of scheduling the queued packets and therefore can be employed to enforce traffic prioritization strategies.

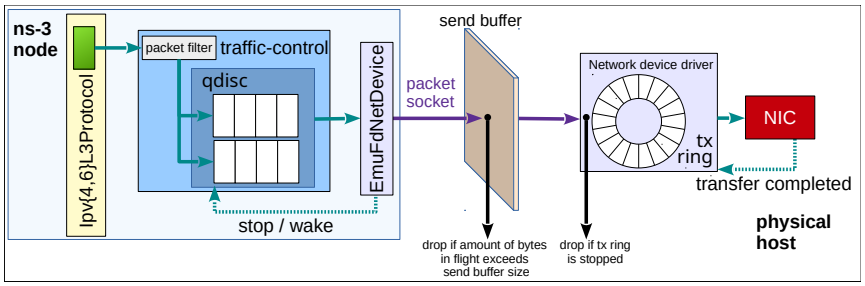
Immediately after passing a packet to a qdisc, the TC infrastructure requests the same qdisc to dequeue at most a configurable number of packets. Dequeued packets are handed to the network device driver, which stores them in a circular queue, named *transmission ring*, established in memory shared with the network device (or NIC – Network Interface Card). Each slot in the ring (*descriptor*) stores some information about a packet (or a fragment of a packet), including its length and the address of the physical memory where it is stored. Packets are usually transferred asynchronously to the network device through DMA (Direct Memory Access). When the transfer is completed, the network device updates the head of the circular queue and notifies the device driver by raising an interrupt.

If the rate at which packets are passed to the transmission ring is higher than the rate at which packets are transmitted over the network, the number of packets queued in the ring grows until the ring becomes full. In such a situation, there is no room for further packets dequeued from the qdisc, which would therefore be dropped. In order to avoid dropping packets, Linux defines the following *flow control* strategy. When the transmission ring has not enough room for another packet, the network device driver *stops* the transmission ring, so that the TC infrastructure refrains from sending further packets down to the network device driver. Clearly, the transmission ring needs to be *restarted* as soon as there is available room, so that the TC infrastructure can resume sending packets down the stack. To this end, every time the network device driver is notified that some packets have been removed from the transmission ring, it checks whether a driver-specific number of descriptors are available in the ring. If so, and the transmission ring is stopped, the network device driver *restarts* the transmission ring and *wakes* the qdisc, i.e., it requests the qdisc to perform multiple dequeue operations, until either a configurable number of packets have been dequeued or the transmission ring is stopped, whichever occurs first. This flow control strategy, therefore, ensures that no packet is passed to the network device driver when the transmission ring is full and that packets are pulled from the qdisc if there is available space in the transmission ring.

It is worth to point out that the time spent by packets waiting in the transmission ring adds to the time spent in the qdisc, which is what AQM algorithms aim to control. Therefore, in order to preserve the effectiveness of AQM algorithms, the waiting time in the transmission ring should be minimized. We note that the size of the transmission ring, in terms of number of descriptors, can be configured via software tools. The size of the transmission ring should allow to minimize the waiting time, while ensuring that starvation does not occur (i.e., the NIC is ready to transmit a packet over the network but the ring is empty). Unfortunately, the minimum amount of packets to store in the transmission ring to avoid starvation is not a fixed value, but depends on multiple factors, including packet size and CPU load [51]. The Linux kernel includes an algorithm named BQL (Byte Queue Limits), whose aim is to adaptively determine a *limit* indicating the minimum amount of bytes to store in the transmission ring to avoid starvation. Network device drivers supporting BQL



(A)



(B)

FIGURE 4.3: Schematic representation of the network stack in the emulated scenario with: (a) packet socket; (b) packet socket with the `PACKET_QDISC_BYPASS` option enabled.

notify the BQL library every time packets are enqueued or dequeued from the transmission ring. Such information allows BQL to update its limit, stop the transmission ring if the amount of queued bytes exceeds the limit and restart the transmission ring if it was stopped and the amount of queued bytes is below the limit.

In order to enhance adherence of the ns-3 network stack to a real network stack, we introduced the traffic-control module in ns-3.25 [52], [53]. The traffic-control module adds a new layer (Fig. 4.2b), sitting between IP (Integrated Protocol) and the *NetDevices* (i.e., the objects representing network devices), that reproduces the Linux TC infrastructure as accurately as possible. The traffic-control module hosts queuing disciplines and packet filters. Proper infrastructure is in place to enable NetDevices to perform flow control. Indeed, a NetDevice can stop its queues (which are the analogous of the transmission rings) when there is not enough room for another packet and restart them when a packet can be accommodated. We introduced the BQL library in ns-3 as well [51]. We added flow control and BQL support to a number of NetDevices, which start and stop their queues in much the same way Linux network device drivers do.

4.2.2 The ns-3 network emulation approach based on packet sockets and its limitations

The FdNetDevice is an ns-3 NetDevice able to read and write from a file descriptor. Network emulation capability is provided by ns-3 through the EmuFdNetDevice,

which is a specialization of the FdNetDevice that creates a packet socket to an underlying physical device and uses the returned file descriptor. In such a way, ns-3 simulations can send/receive packets to/from a real network device. Indeed, a packet socket injects packets at some point along the normal transmission path. By default, packets are enqueued in the qdisc installed on the network device bound to the packet socket (Fig. 4.3a). If the `PACKET_QDISC_BYPASS` socket option is enabled, packets are sent to the network device if its transmission ring is not stopped and dropped otherwise (Fig. 4.3b).

As described earlier, network emulation is typically used to test a prototype implementation of a higher layer (application, transport or network) protocol or algorithm in real world conditions. To actually test the proposed solution in real world conditions, however, it is necessary that in the emulated scenario (Fig. 4.3) packets experience the same delay and the same dropping probability as they would in the real scenario (Fig. 4.2a). Indeed, for instance, the behavior of transport layer protocols such as TCP is highly affected by such measures. Given that the main source of delay in the real stack is the queuing time, it is necessary that the overall amount of packets queued in the various buffers between the traffic control (included) and the network device is similar in the two scenarios. However, this is not enough to ensure an accurate network emulation. Indeed, qdiscs implementing AQM algorithms take decisions about dropping packets based on the qdisc backlog or the packet queuing time within the qdisc. Therefore, it is also required that the backlog of the ns-3 qdisc in the emulated scenario is kept similar to the backlog of the Linux qdisc in the real scenario.

Given that packets accumulate in the Linux (ns-3) qdisc when the transmission ring (EmuFdNetDevice queue[1]) is stopped, the requirement above is met if the EmuFdNetDevice queue is stopped (restarted) in the emulated scenario when the transmission ring would be stopped (restarted) in the real scenario. To this end, we distinguish between two cases:

1. when not using BQL, it is the device driver that stops the transmission ring when there are no descriptors available in the transmission ring and restarts it when some descriptors are freed. Therefore, the EmuFdNetDevice must be capable to detect that the overall amount of packets queued in the host qdisc (if present) and in the transmission ring equals the transmission ring size, so as it can properly stop and restart its queue.

2. when BQL is enabled, it is the BQL library that stops the transmission ring

[1] An EmuFdNetDevice does not really use any queue, since every received packet is immediately sent through the socket; however, it may declare that its queue is stopped for the purpose of preventing the qdisc from sending further packets

when the amount of bytes queued in the transmission ring exceeds the computed BQL limit and restarts it when the amount of bytes queued in the transmission ring goes down below the computed BQL limit. Hence, the transmission ring can be stopped even if it is not full. To compute the limit, the BQL library requires to be periodically notified of the amount of bytes enqueued and dequeued from the transmission ring. In the emulated scenario, given that the EmuFdNetDevice has no means to get the limit computed by the Linux BQL library and hence cannot stop its queue when the overall amount of bytes queued in the host qdisc (if present) and in the transmission ring exceeds such limit, it is necessary to enable the ns-3 BQL library. However, the ns-3 BQL library needs to be notified by the EmuFdNetDevice about the amount of bytes the EmuFdNetDevice sends to the host qdisc (or directly to the transmission ring) and the amount of bytes that are removed from the transmission ring because they have been transferred to the device.

Unfortunately, the EmuFdNetDevice cannot satisfy the two requirements above due to some limitations of the packet sockets. Like all the types of sockets, the packet socket uses a *send buffer* to limit the amount of bytes *in flight*. The send buffer is not really a buffer, but rather a counter that is incremented (by the packet size) when a packet is sent (to the host qdisc or directly to the transmission ring) and decremented when the *destructor* of a packet is called by the device driver (upon notification that the transfer of the packet to the device has been completed). When the socket has to send a packet and the counter exceeds the send buffer size, the packet is dropped (unless the socket is in blocking mode). It follows that:

- The send buffer size is an upper bound on the overall amount of bytes that can be queued in the host qdisc (if present) and in the transmission ring in the emulated scenario. To meet the first of the requirements above, we would need the send buffer size to be such that the number of packets in flight cannot exceed the transmission ring size. However, the fundamental issue here is that the transmission ring size is measured in terms of number of packets, while the send buffer size is measured in bytes. Given that packets have variable sizes, it is not possible to find a value for the send buffer size that ensures that the number of packets in flight does not exceed the number of packets that can be stored in the transmission ring.

- The EmuFdNetDevice sends data over the packet socket by calling the *write* function, which returns the number of bytes written. Hence, the EmuFdNetDevice is able to notify the ns-3 BQL library about the amount of bytes sent to the host qdisc (or directly to the transmission ring). However, the EmuFdNetDevice has no means to determine the amount of bytes transferred to the network device. The EmuFdNetDevice could infer such value from the counter of the packet socket that keeps track of the inflight bytes, but this value is not

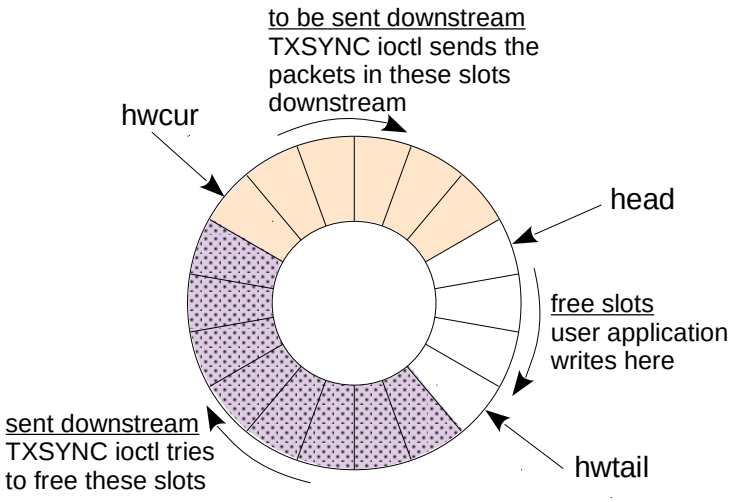


FIGURE 4.4: Representation of the three different parts in which the netmap ring can be divided.

accessible from the user space. Hence, the second of the requirements above cannot be met.

The analysis above led us to conclude that the packet socket is not suitable for the purpose of achieving an accurate network emulation mechanism. Therefore, we looked at alternative tools. The analysis above shows that a key requirement is the ability for user space applications (in our case, ns-3) to detect how many packets/bytes are in flight, i.e., queued in the various buffers of the networking stack waiting to be transferred to the network device.

4.2.3 The netmap framework for high speed packet I/O through direct NIC access

A number of software frameworks, such as netmap, DPDK [54] and PF_RING ZC [55], aim to reduce the I/O cost of processing network packets. The main applications of these frameworks fall in the high rate packet generation and packet capture. Such frameworks employ a number of common techniques (bypass of the default network stack, preallocation of packet buffers, usage of memory mapped in user space to avoid packet copy, processing batch of packets to reduce the number of syscalls) to decrease the number of CPU cycles required to transmit/receive packets [56]. However, the reason why we looked at these frameworks in the attempt to improve the accuracy of network emulation is not because of the reduced I/O cost of packet processing they ensure. Rather, we were looking for alternative techniques allowing an application to send packets to a network device, given that the packet socket, as shown in the previous section, is not suited for our purposes. Among the mentioned frameworks, we selected netmap because it is open source, is available on multiple operating systems (Linux, FreeBSD and Windows) and serves our purposes even without having to patch the network device drivers.

Two operation modes are provided by netmap: *native* and *generic*. The native mode is faster, but requires modified kernel drivers. The generic mode is considered as a fallback mode because it is slightly slower (there is an extra copy for each packet), but it does not require to modify the kernel drivers. In order to use netmap, it is necessary to load the netmap kernel module and the modified kernel driver for the network device, if the native mode is to be used. At this point, the network device can still be used by any application through the kernel network stack. When an application opens a file descriptor on */dev/netmap* and uses it to switch the device to netmap mode, that application only can use the network device (through the netmap stack). The kernel network stack will be used again when the user application closes the file descriptor.

When a network device is switched to netmap mode, a distinct netmap ring is allocated for each transmission (receiver) ring used by the device driver and mapped in the user process memory area. Each netmap ring has the same size, in terms of slots, as the corresponding transmission (receiver) ring (Fig. 4.4). The user application can use the usual primitives that operate on file descriptors to write packets to the netmap ring (e.g., `write`) and to check the availability of free slots (e.g., `select` and `poll`). Packets written to the netmap ring are not automatically sent to the downstream buffer (i.e., the qdisc or the transmission ring, as described hereinafter). It is the responsibility of the user application to periodically invoke a system call, an `ioctl` request of type TXSYNC, which sends to the downstream buffer all the packets queued in the netmap ring that have not been sent yet. In this way, a single system call is invoked to transfer a batch of packets. It is to be noted that packets sent to the downstream buffer stay in the netmap ring until the downstream buffer has "consumed" (i.e., dequeued) them. Indeed, given that the netmap ring has the same size as the downstream buffer, removing packets that have not been consumed by the downstream buffer would free some slots in the netmap ring. Such slots could be filled by new packets written by the user application, which could not be accommodated by the downstream buffer when the TXSYNC `ioctl` request is made. Freeing the netmap ring slots occupied by packets that have been consumed by the downstream buffer is the main other task performed by the TXSYNC `ioctl` request.

To allow netmap to operate in native mode, the device driver needs to be modified to provide a function serving the TXSYNC `ioctl` request. Such function inserts the packets queued in the netmap ring directly in the transmission ring of the device driver (packets are not copied, pointers only are copied). Also, it frees the slots occupied in the netmap ring by the packets that have been removed from the transmission ring (because the device has notified the driver that the transfer has been completed). Additionally, netmap overrides the interrupt service routines handling the interrupts raised by the device to signal the transmission or reception of packets by replacing them with a function that notifies such events to the user application. It is worth to note that the modified drivers lack proper calls to the BQL library, hence BQL is not available when using netmap in native mode. Currently, a number

of device drivers have been modified to support netmap, ranging from drivers for common 1Gbps adapters (e.g., the Intel e1000 driver) to drivers for specialized high performance 10Gbps adapters (e.g., the Intel ixgbe driver).

The generic mode of netmap allows it to work with unmodified kernel drivers. On Linux, packets in the netmap ring are sent to the kernel network stack by calling the `dev_queue_xmit` function, which, as described earlier, hands packets to the TC infrastructure. Two alternatives are available here. By default, a netmap-aware qdisc called *netmap_generic* is automatically installed on the device operating in netmap mode. This qdisc has the same size as the transmission ring, and hence of the netmap ring. In this case, the TXSYNC `ioctl` frees the packets in the netmap ring that have been dequeued from the netmap_generic qdisc. It turns out that the maximum amount of packets in flight is twice the size of the transmission ring: both the netmap_generic qdisc and the transmission ring can be full, and the netmap ring contains the packets queued in the netmap_generic qdisc.

The other option available in generic mode can be selected by configuring a parameter accessible via the *sysfs* virtual filesystem. In this case, the netmap_generic qdisc is not installed, hence the qdisc previously installed on the device operating in netmap mode is used. From the point of view of freeing the packets in the netmap ring, netmap behaves like when it is used on other operating systems, where packets written to the netmap ring are directly copied into slots of the transmission ring. This means that the TXSYNC `ioctl` frees the packets in the netmap ring that have been removed from the transmission ring (because the device notified the driver that it retrieved those packets). Therefore, the overall amount of packets stored in the qdisc and in the transmission ring cannot exceed the transmission ring size.

4.3 Exploiting netmap to enhance the fidelity of network emulation

We now present the design of a new ns-3 NetDevice, named NetmapNetDevice, that exploits netmap to provide accurate network emulation. Since netmap uses file descriptor based communication to interact with the real device, the straightforward approach to design the new NetDevice is to have it inherit from the existing FdNetDevice and implement a specialized version of the operations specific to netmap. The operations that require a specialized implementation are the initialization, because the NIC has to be put in netmap mode, and the read/write methods, which have to make use of the netmap API to coordinate the exchange of packets with the netmap rings.

In the *initialization* stage, the network device is switched to netmap mode, so that the user application (ns-3, in our case) is able to send/receive packets to/from the real network device by writing/reading them to/from the netmap rings. Following the design of the FdNetDevice, a separate reading thread is started during the initialization. The task of the reading thread is to wait for new incoming packets in

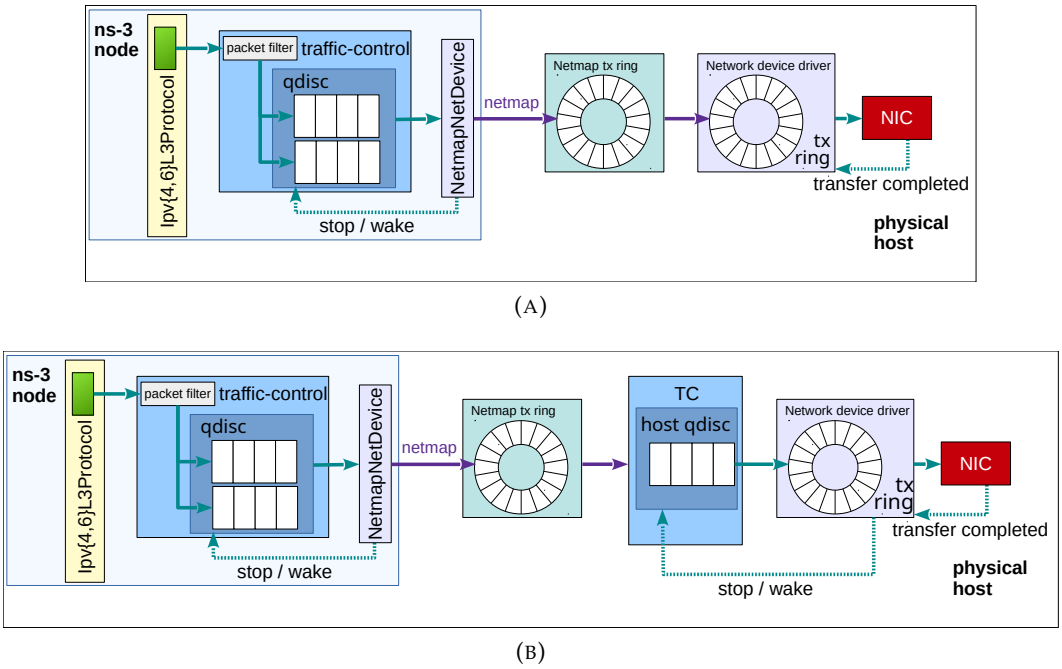


FIGURE 4.5: Schematic representation of the network stack in the emulated scenario with: (a) netmap in native mode; (b) netmap in generic mode.

the netmap receiver rings, in order to schedule the events of packet reception. In the initialization of the NetmapNetDevice, an additional thread, the *sync* thread, is started. The sync thread is required because, in order to reduce the cost of the system calls, netmap does not automatically transfer a packet written to a slot of the netmap ring to the transmission ring or to the installed qdisc. It is up to the user process to periodically request a synchronization of the netmap ring. Therefore, the purpose of the sync thread is to periodically make a TXSYNC `ioctl` request, so that pending packets in the netmap ring are transferred to the transmission ring, if in native mode, or to the installed qdisc, if in generic mode. Also, as described hereinafter, the sync thread is exploited to perform flow control and notify the BQL library about the amount of bytes that have been transferred to the network device.

The *read* method is called by the reading thread to retrieve new incoming packets stored in the netmap receiver ring and pass them to the appropriate ns-3 protocol handler for further processing within the simulator's network stack. After retrieving packets, the reading thread also synchronizes the netmap receiver ring, so that the retrieved packets can be removed from the netmap receiver ring.

The NetmapNetDevice also specializes the *write* method, i.e., the method used to transmit a packet received from the upper layer (i.e., the ns-3 traffic control layer). The write method uses the netmap API to write the packet to a free slot in the netmap transmission ring. After writing a packet, the write method checks whether there is enough room in the netmap transmission ring for another packet. If not, the

NetmapNetDevice stops its queue[2], so that the ns-3 traffic control layer does not attempt to send a packet that could not be stored in the netmap transmission ring. A stopped NetmapNetDevice queue needs to be restarted as soon as some room is made in the netmap transmission ring. The sync thread can be exploited for this purpose, given that it periodically synchronizes the netmap transmission ring. In particular, the sync thread also checks the number of free slots in the netmap transmission ring in case the NetmapNetDevice queue is stopped. If the number of free slots exceeds a configurable value, the sync thread restarts the NetmapNetDevice queue and wakes the associated ns-3 qdisc. The NetmapNetDevice also supports BQL: the write method notifies the BQL library of the amount of bytes that have been written to the netmap transmission ring, while the sync thread notifies the BQL library of the amount of bytes that have been removed from the netmap transmission ring and transferred to the NIC since the previous notification.

We now discuss how accurate the network emulation provided by the NetmapNetDevice is, both in native and generic mode, by checking whether the NetmapNetDevice queue is stopped (restarted) in the emulated scenario when the transmission ring would be stopped (restarted) in the real scenario. Again, we distinguish between two cases:

- when BQL is not enabled, the NetmapNetDevice should be able to stop its queue as soon as the transmission ring is stopped. When netmap operates in native mode (Fig. 4.5a), the netmap ring contains the same packets stored in the transmission ring (after a synchronization). Given that the NetmapNetDevice is able to query the status of the netmap ring, the NetmapNetDevice is able to stop its queue when the transmission ring is full and restart it when enough room is made. When netmap operates in generic mode (Fig. 4.5b), the default configuration including the netmap_generic qdisc is not suited, because the amount of packets in flight is twice the size of the transmission ring. Hence, the NetmapNetDevice would not stop its queue as soon as the transmission ring is full. Instead, when the netmap_generic qdisc is not used, the amount of packets in flight equals the size of the transmission ring. In this case, the NetmapNetDevice is able to properly stop and restart its queue.

- when BQL is enabled, the ns-3 BQL library needs to be notified of the amount of bytes enqueued into the transmission ring and dequeued from it. As explained earlier, such information is provided by the write method and the sync thread both in native mode and in generic mode without the netmap_generic qdisc. In generic mode with the netmap_generic qdisc, instead, the sync thread can return less accurate information, because slots in the netmap ring are freed when packets are dequeued from the qdisc instead of when packets are removed from the transmission ring.

[2]A NetmapNetDevice does not really use any queue, since every received packet is immediately written to the netmap ring; however, it may declare that its queue is stopped for the purpose of preventing the qdisc from sending further packets

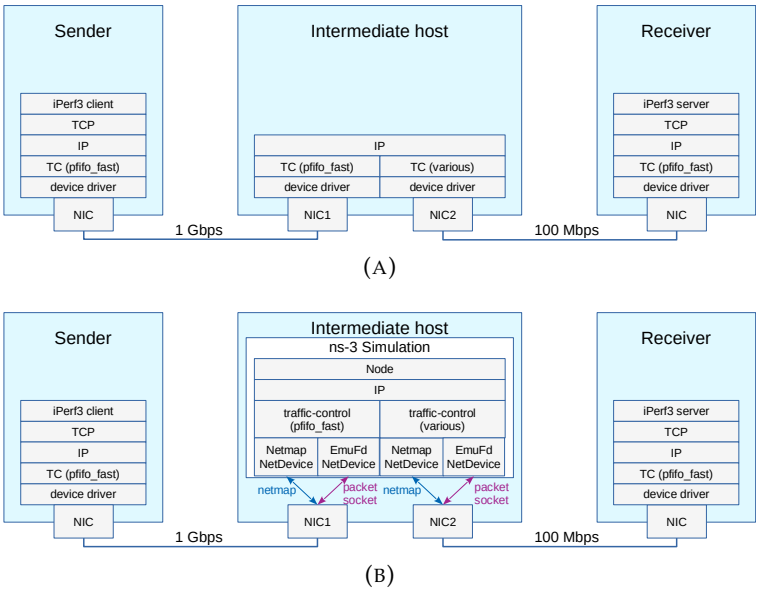


FIGURE 4.6: The testbed used for experiments is comprised of three physical hosts. (a) In the real scenario, the Linux network stack is used. (b) In the emulated scenario, an ns-3 simulation runs on the intermediate host. The simulation scenario includes a single node with two EmuFdNetDevices or two NetmapNetDevices, connected each to one of the NICs of the physical host.

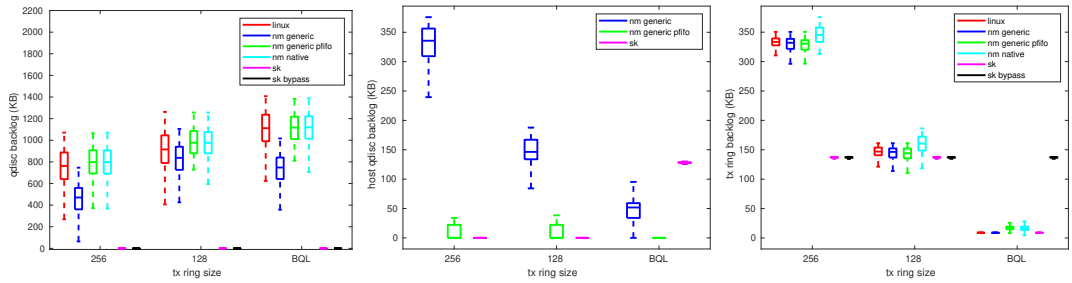
Therefore, we expect that the NetmapNetDevice ensures an accurate network emulation when netmap operates in native mode and in generic mode without the netmap_generic qdisc. Instead, accuracy may be lower when netmap operates in generic mode with the default netmap_generic qdisc.

The NetmapNetDevice can operate with netmap in both native and generic mode. In particular, if the modified kernel driver for the NIC has been loaded, the native mode is used. Otherwise, the generic mode is used. Finally, we mention that the NetmapNetDevice has three parameters:

- a boolean value indicating whether or not to install the netmap_generic qdisc when operating in generic mode. Given that the main usage of the NetmapNetDevice is to perform network emulation experiments and, as shown in this work, network emulation is more accurate when the netmap_generic qdisc is not installed, this value defaults to false.

- the sync thread sleep time. The default value is 200 us, which is the same as the interrupt coalescence timer used by device drivers like tg3 or e1000e. This value is a good compromise between the need of keeping the netmap ring in sync with the transmission ring and the need of reducing the number of system calls.

- the predefined number of slots in the netmap ring that need to be freed before restarting the NetmapNetDevice queue. This value defaults to 32, which is the same as the value used by device drivers like tg3 or e1000e.



(A) qdisc backlog (B) host qdisc backlog (C) transmission ring backlog

FIGURE 4.7: Comparison between the Linux stack and the network emulation techniques under test: in flight bytes

4.4 Experimental results

4.4.1 Assessing the accuracy of network emulation techniques

We conducted a thorough experimental campaign to compare the results achieved by the real Linux network stack (labelled as *linux* in the following figures) to different network emulation techniques: packet socket with (*sk bypass*) and without (*sk*) the `PACKET_QDISC_BYPASS` option enabled, netmap in native mode (*nm native*), netmap in generic mode with the netmap_generic qdisc (*nm generic*) and with the default pfifo_fast qdisc (*nm generic pfifo*), which is a simple First-In First-Out packet scheduler. For a fair comparison, it is necessary that the EmuFdNetDevice and the NetmapNetDevice are passed the same traffic pattern as the Linux network device driver in the real scenario. To this end, we devised a setup with three nodes connected back to back through Ethernet cross cables (Fig. 4.6). The intermediate host acts as a router between the sender and the receiver and is configured to use the Linux stack in the real scenario (Fig. 4.6a) and the ns-3 stack in the emulated scenario (Fig. 4.6b). Sender and receiver nodes use instead the Linux stack in both scenarios. In this way, the traffic is generated by the same application running on a Linux host both in the real scenario and in the emulated scenario. Also, the Linux implementation of TCP is used in both scenarios, thus avoiding that possible differences in the Linux and ns-3 implementations of TCP may affect the results of our tests.

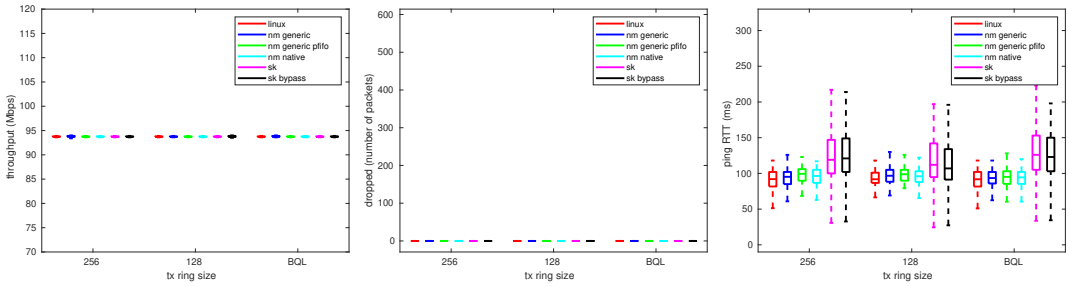
In order to compare the Linux stack and the ns-3 stack under meaningful test conditions, the rate of the link between the sender and the intermediate host is 1 Gbps, while the rate of the link between the intermediate host and the receiver is 100 Mbps. In this way, some backlog accumulates in the compared network stacks, so as to better highlight discrepancies in their behavior. The focus is thus on the network stack associated with NIC2 (an Intel 1 Gbps Ethernet adapter using the Linux e1000e driver), which is where packets accumulate and hence contributes the most to the resulting latency and packet drops (the return path is lightly loaded because only traversed by Echo Reply messages and TCP acknowledgments). We evaluated the network emulation techniques in three different test conditions:

- the size of the transmission ring of NIC2 is set to the default value for the Linux e1000e driver (256 descriptors) and BQL is disabled (both in Linux and ns-3);
- the size of the transmission ring of NIC2 is set to half the default value for the Linux e1000e driver (128 descriptors) and BQL is disabled (both in Linux and ns-3);
- the size of the transmission ring of NIC2 is set to the default value for the Linux e1000e driver (256 descriptors) and the Linux (ns-3) implementation of BQL is enabled in the real scenario (emulated scenario).

Each of the three hosts is equipped with an Intel i7-6700 CPU and a 16GB RAM, and runs the Linux 4.11 kernel. TCP traffic is generated by an iPerf3 client running on the sender node and destined to the iPerf3 server running on the receiver node. iPerf3 is a tool for active measurements of the maximum achievable bandwidth on IP networks. For the experiments in the emulated scenario (Fig. 4.6b), we use ns-3.28 compiled in *optimized* mode. We run an ns-3 simulation on the intermediate host where one ns-3 node has two NetDevices (both of type EmuFdNetDevice or NetmapNetDevice), each of which is associated with one of the real NICs. Proper routes are installed in the routing table of the ns-3 node, so that packets received by the NetDevice associated with NIC1 are forwarded to the NetDevice associated with NIC2 (and viceversa). In this set of experiments, all the qdiscs (but the host qdisc when netmap is used in generic mode with the default netmap_generic qdisc) are of type pfifo_fast. For the experiments requiring to disable BQL, we set the minimum value for the BQL limit to a value higher than the transmission ring capacity, obtained by multiplying the number of descriptors by the packet size (1500 bytes).

For each experiment, we performed 5 consecutive tests, each of which lasting 30 seconds. The transient state during which queues build up lasts for about a couple of seconds, thus we believe that 30 seconds are enough to capture the steady-state evolution of the quantities we measure. Also, results are rather similar from one test to another, so we believe that repeating an experiment 5 times is enough to minimize the impact of undesired factors. For each experiment, we take samples of a number of quantities (throughput, latency, packet drops, qdisc backlog and transmission ring backlog) to compare the Linux stack to the various network emulation techniques. The empirical distribution of the set of samples collected for each quantity in each experiment is summarized by means of box plots. On each box, the central mark indicates the median, and the bottom and top edges of the box indicate the 25th and 75th percentiles, respectively. The whiskers extend to the most extreme data points not considered outliers.

In the following, we first analyze the amount of bytes buffered in the network stack (below the IP layer) by looking at the backlog of the qdiscs and of the transmission ring on NIC2. Then, we show the results in terms of user visible metrics such as throughput, packet drops and round-trip time.



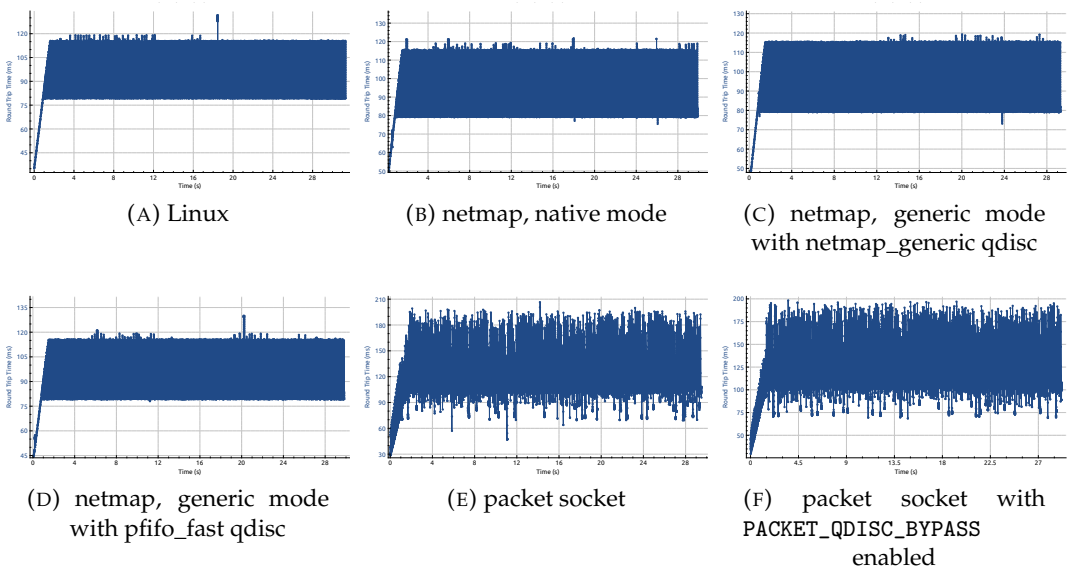
(A) throughput (B) packet dropped by the qdisc (C) ping round-trip time

FIGURE 4.8: Comparison between the Linux stack and the network emulation techniques under test: throughput, packet drops and round-trip time.

Transmission ring backlog

For each experiment, we collected the amount of bytes queued in the transmission ring of NIC2 every millisecond by reading the *inflight* variable (accessible via *sysfs*) that is updated by the BQL library. Figure 4.7c shows the distribution of the collected samples in the real scenario and in the emulated scenarios. Given that the traffic generator attempts to saturate the channel capacity, we expect that the transmission ring is full most of the time when BQL is not enabled. Figure 4.7c shows that this is actually the case for the Linux stack and for all the emulated scenarios based on netmap[3]. In the emulated scenarios based on the packet socket, the backlog of the transmission ring is constantly equal to a value which is smaller than the capacity of the transmission ring, independently of its size (256 or 128 descriptors). Such result is explained by considering that the amount of bytes that the socket send buffer size allows to be in flight is not enough to saturate the transmission ring. When BQL is enabled, the backlog in the transmission ring does not exceed the limit that is dynamically computed by the BQL library. Given that such limit is typically smaller than the socket send buffer size, the backlog measured when the packet socket is used is similar to that achieved in the real scenario. When the `PACKET_QDISC_BYPASS` option is enabled, however, the transmission ring still contains as many bytes as allowed by the socket send buffer size. The reason is that, in such a case, the packet socket routine that sends packets to the transmission ring only checks whether the transmission ring has been stopped by the device driver, thus ignoring the situations where the transmission ring has been stopped by the BQL library.

[3]The backlog of the transmission ring when using netmap in native mode is not collected by reading the *inflight* variable because the modified network device driver does not notify the BQL library of the amount of bytes enqueued and dequeued from the transmission ring. Instead, the backlog of the netmap ring is collected.



(A) Linux (B) netmap, native mode (C) netmap, generic mode with netmap_generic qdisc

(D) netmap, generic mode with pfifo_fast qdisc (E) packet socket (F) packet socket with `PACKET_QDISC_BYPASS` enabled

FIGURE 4.9: Round-trip time of every TCP segment acknowledged by the receiver in a single test run (with BQL enabled).

Qdisc backlog

For each experiment, we collected the amount of bytes stored by the various qdiscs involved every millisecond. In particular, Fig. 4.7a shows the distribution of the collected samples for the Linux qdisc installed on NIC2 in the real scenario (Fig. 4.2a) and for the ns-3 qdisc installed on the NetDevice associated with NIC2 in the emulated scenarios. Figure 4.7b, instead, shows the distribution of the collected samples for the additional host qdisc installed on NIC2 in the emulated scenarios with netmap in generic mode (Fig. 4.5b) and with the packet socket with the `PACKET_QDISC_BYPASS` option disabled (Fig. 4.3a). From these figures, it can be observed that:

- when using netmap in native mode, the backlog of the ns-3 qdisc is very similar to that of the Linux qdisc (Fig. 4.7a). This result proves that the NetmapNetDevice queue is stopped (restarted), either by the NetmapNetDevice itself or by the ns-3 BQL library, when the transmission ring would be stopped (restarted), if netmap operates in native mode;

- when using netmap in generic mode with the default netmap_generic qdisc and BQL is disabled, the backlog of the host qdisc is similar to the backlog of the transmission ring (Figs. 4.7b and 4.7c). This result is expected because, as discussed earlier, the strategy used to free the slots in the netmap ring allows twice as many packets in flight as the transmission ring size. The backlog of the ns-3 qdisc is smaller than that of the Linux qdisc. The reason is that the NetmapNetDevice queue is stopped less frequently than the transmission ring in the real scenario and therefore less packets accumulate in the qdisc. As discussed earlier, when BQL is enabled, the ns-3 BQL library receives inaccurate information about the amount of bytes transmitted

by the device (the BQL library *sees* a device that is able to transmit more bytes than it actually does, just because packets sent to the driver are removed from the netmap ring). Indeed, when BQL is enabled, the host qdisc has a non-negligible backlog and the backlog of the ns-3 qdisc is different than that of the Linux qdisc;

- when using netmap in generic mode with the pfifo_fast qdisc, the backlog of the ns-3 qdisc is very similar to that of the Linux qdisc (Fig. 4.7a). The host qdisc has a negligible backlog when BQL is disabled (likely due to the fact that, when enough room is made in the transmission ring after it has been stopped, the qdisc is awaken; however, in the meantime the qdisc is actually served, netmap may transmit packets that therefore accumulate in the host qdisc). This result proves that netmap is able to provide accurate network emulation also when operating in generic mode without the default netmap_generic qdisc.

- when using the packet socket and BQL is disabled, we already observed that the socket send buffer size prevents the transmission ring from being full. Consequently, the transmission ring is never stopped and the qdisc backlog is null. When BQL is enabled, there is some backlog in the host qdisc because the packets that are sent over the packet socket when the transmission ring is stopped by the BQL library are buffered in the host qdisc.

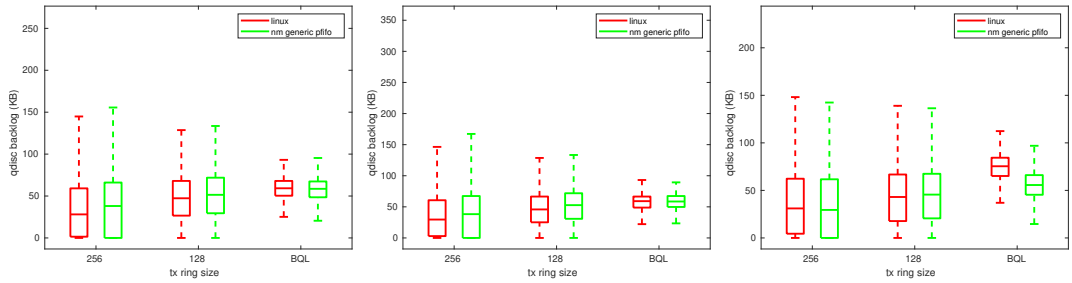
These results show that the backlog of the ns-3 qdisc is similar to that of the Linux qdisc only when using netmap in native mode or netmap in generic mode without the default netmap_generic qdisc. We recall that the qdisc backlog is an important parameter, because AQM algorithms usually decide to drop packets based on the qdisc backlog or the waiting time in the qdisc.

Throughput

For each experiment, we collected the average throughput over time intervals of 100 milliseconds, as measured by the iPerf3 server. Figure 4.8a shows that the throughput is very stable across the whole duration of every experiment, for both the real and the emulated scenarios, and the stable value is about 94 Mbps in all the experiments. This result shows that the EmuFdNetDevice and the NetmapNetDevice are able to sustain transmission rates close to 100 Mbps.

Packet drops

For each test, we measured the cumulative number of packets dropped by the Linux qdisc installed on NIC2 (in the real scenario) or by the ns-3 qdisc installed on the NetDevice associated with NIC2 (in the emulated scenarios), every millisecond. Figure 4.8b shows that in all the cases the qdiscs do not drop packets. This result is expected because we showed earlier that the qdiscs are not full (the default capacity of a pfifo_fast qdisc is 1000 packets). However, it is to be mentioned that packets may



(A) CoDel (B) FQ-CoDel (C) RED

FIGURE 4.10: Backlog of the AQM algorithms in the validation experiments.

be dropped elsewhere, e.g., when using the packet socket and the amount of bytes in flight exceeds the socket send buffer size.

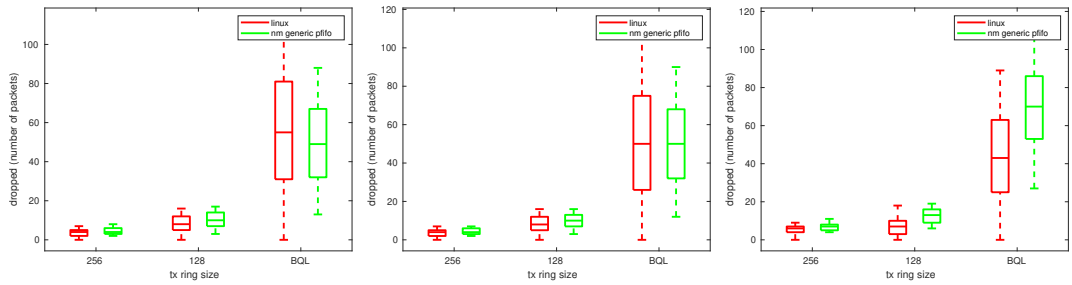
Round-trip time

For each experiment, we collected the round trip time of ICMP Echo Request/Reply messages sent by the sender host every 5 milliseconds and destined to the receiver host. Figure 4.8c shows that the results obtained by using netmap are very similar to those obtained with Linux for all the experiments. We observe that in the case of netmap operating in generic mode with the default netmap_generic qdisc, the latency is the same as in the other cases because the sum of the backlog of the ns-3 qdisc and the host qdisc roughly equals the backlog of the ns-3 qdisc in the other cases (Figs. 4.7). However, if the ns-3 qdisc implemented an AQM algorithm, the results would have been different because of the different backlog of the ns-3 qdisc. Figure 4.8c also shows that the latency experienced when the packet socket mechanism is used is rather different than that achieved with the real stack.

In order to have a more in-depth look at the results in terms of latency, we also captured, for every test, the trace of the packets sent/received by the sender host by using *wireshark*. Then, we used the same tool to plot the round-trip time of every single TCP segment acknowledged by the receiver. Figure 4.9 shows the resulting plots for a single test run with BQL enabled (all the other test runs show a very similar behavior). These plots confirm that the NetmapNetDevice is able to provide a very accurate network emulation, given that packets experience a latency that is extremely similar to that experienced with the real stack. Contrarily, it appears clear that the EmuFdNetDevice using the packet socket mechanism is not able to provide a similar level of accuracy.

4.4.2 Validation of the ns-3 implementation of AQM algorithms

The experiments presented in the previous subsection prove that the NetmapNetDevice is able to guarantee an accurate network emulation, even when netmap is used in generic mode (with the pfifo_fast qdisc). The backlog of the ns-3 qdisc is



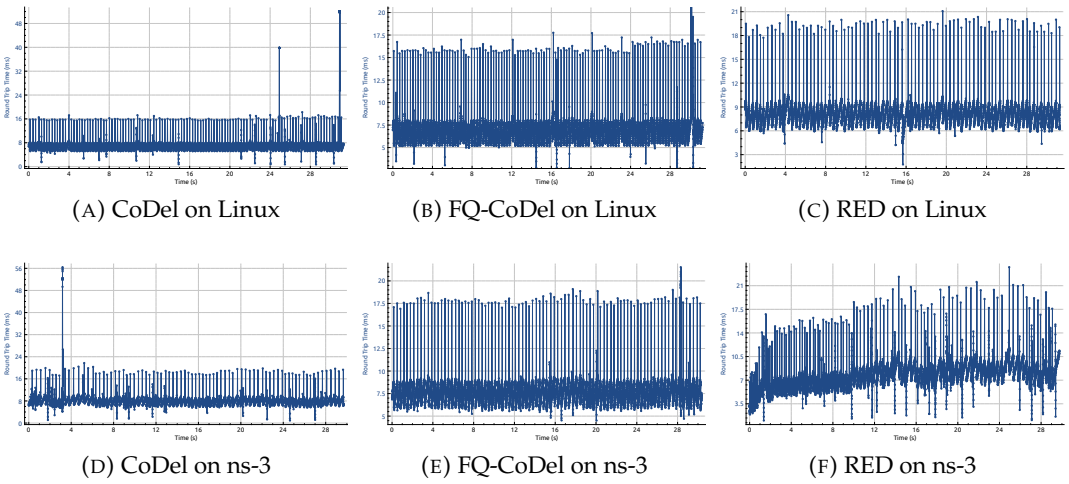
(A) CoDel (B) FQ-CoDel (C) RED

FIGURE 4.11: Cumulative number of packets dropped by the AQM algorithms in the validation experiments.

very similar to that of the Linux qdisc in the real scenario and it is just like if the ns-3 qdisc was running on a real device. We can build upon such result to validate the ns-3 implementation of some AQM algorithms (RED, CoDel and FQ-CoDel) against their Linux implementation. The same setup as in Fig. 4.6 can still be used, with the only difference that the AQM algorithm under test is installed on NIC2 (on the NetmapNetDevice associated with NIC2) in the real (emulated) scenario. For these experiments, netmap operating in generic mode with the pfifo_fast qdisc is used.

We omit the figures regarding the throughput (because it is constantly equal to about 94 Mbps in all the tests) and the transmission ring backlog (because it is influenced by the emulation mechanism rather than the qdisc) and focus on the metrics that are mainly affected by AQM algorithms: qdisc backlog, number of packets dropped by the qdisc and latency. Figure 4.10 shows that the backlog of the ns-3 qdisc is very similar to that of the Linux qdisc for all the experiments. A bit of an exception is the case of RED when BQL is enabled, because the backlog of the Linux qdisc is higher. Figure 4.11 shows that the cumulative number of packets dropped by the ns-3 qdisc is very similar to that of the Linux qdisc for all the experiments. Again, there is a small discrepancy in the case of RED when BQL is enabled, because the ns-3 qdisc drops more packets. This result is consistent with the previous one: the ns-3 implementation of RED drops more packets than the Linux one when the backlog is high and hence the resulting backlog is lower than that of the Linux qdisc.

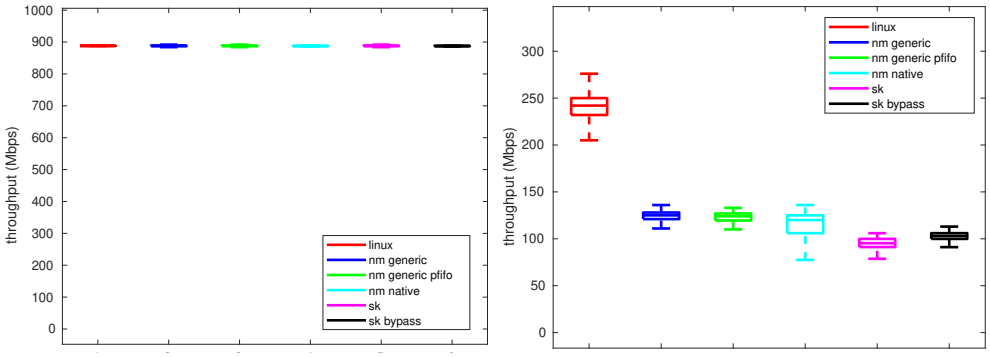
The comparison in terms of latency is again shown by taking one test run with BQL enabled and reporting the plots of the round-trip time of the acknowledged TCP segments as produced by wireshark. Figure 4.12 shows that the ns-3 version of CoDel and FQ-CoDel behave extremely similarly to their Linux counterpart. As highlighted above, some slight discrepancy can be instead observed in the behavior of RED when the qdisc backlog is high. With this caveat, we can state that the experiments we conducted prove the validity of the ns-3 implementation of CoDel, FQ-CoDel and RED.



(A) CoDel on Linux (B) FQ-CoDel on Linux (C) RED on Linux

(D) CoDel on ns-3 (E) FQ-CoDel on ns-3 (F) RED on ns-3

FIGURE 4.12: Round-trip time of every TCP segment acknowledged by the receiver in a single test run (with BQL enabled).



(A) Packet size: 1000B (B) Packet size: 100B

FIGURE 4.13: Throughput achieved with different packet sizes.

4.4.3 Analysis of the maximum achievable data rates

We conclude our experiments by analyzing the maximum data rates achievable by ns-3 when using the network emulation mechanisms studied in this work. We continue using the setup illustrated in Fig. 4.6, with the only difference that the capacity of the link between the intermediate host and the receiver is 1 Gbps. When evaluating the ability of a network stack to transmit packets, it is appropriate to measure the number of packets transmitted per second, given that there is a cost associated to the processing of every single transmitted packet. In our emulated scenarios, both ns-3 and the network emulation mechanism (i.e., the packet socket or netmap) contribute to the per packet processing cost[4]. We conducted a first test by generating packets of 1000 bytes. In this case, a 1Gbps link is saturated by transmitting 125 kpps (kilo packets per second). Figure 4.13a shows that the ns-3 stack with any of the considered network emulation mechanisms is able to achieve the same throughput as the

[4]In the considered setup, the ns-3 processing consists of routing packets from one NetDevice to another. The ns-3 processing cost may be different in other scenarios (e.g., packets generated by ns-3 that traverse the whole ns-3 stack).

Linux stack (about 900Mbps). Hence, both the EmuFdNetDevice and the NetmapNetDevice are able to route 125 kpps. Then, we conducted another test by generating packets of 100 bytes. In this case, a 1Gbps link is saturated by transmitting 1.25 Mpps. Fig. 4.13b shows that the Linux stack achieves 240 Mbps (corresponding to 300 kpps), while the ns-3 stack achieves about 120 Mbps (corresponding to 150 kpps) when using the NetmapNetDevice and about 100 Mbps (corresponding to 125 kpps) when using the EmuFdNetDevice. Thus, the optimizations implemented by netmap enable a throughput increase of about 20% with respect to the packet socket in this scenario. Therefore, the NetmapNetDevice allows to reduce the per packet processing cost with respect to the EmuFdNetDevice, which translates in a higher throughput when the maximum packet rate is bounded by the CPU usage.

4.5 Conclusions

In this chapter we presented the work carried out to enhance the network emulation capabilities of network simulators such as ns-3. We first described the limitations of the current approach based on packet sockets, which is adopted by various network simulators. Then, we introduced netmap as an alternative technique and presented the design of the new NetmapNetDevice for ns-3. We conducted a thorough experimental campaign to show the accuracy of the network emulation mechanism based on netmap. We considered two different NIC ring sizes and the usage of BQL, in order to perform the tests under different backlog conditions. Building upon the accurate network emulation provided by the NetmapNetDevice, we performed other experiments to validate the ns-3 implementation of various AQM algorithms (CoDel, FQ-CoDel, RED) against the Linux implementation. The experimental results showed the accuracy of the ns-3 implementation of such qdiscs, except for a slight discrepancy in the case of RED when the qdisc backlog is high.

Chapter 5

Proposals of design and evaluation of traffic control strategies

In this chapter, we move on to the design and evaluation of traffic control strategies in communication networks. The examples exploit simulation and, when possible, results comparison with emulation and testbed. We show two examples, the first in the context of 3GPP stacks while the second in the context of AQM algorithms. In the first case, we designed and evaluated a traffic control strategy on top of LTE stack by using simulation with ns-3. In the second case we prove design flaws in rate based AQM algorithms through comparison of emulation and testbed results, then we propose possible solutions.

5.1 Introduction

In this chapter we rely on simulation, emulation and testbed approaches to design and evaluate traffic control strategies. In these examples, we exploit the ns-3 traffic control module and its AQM algorithms implemented and validated in this work. More specifically, we exploited ns-3 based simulation to design a traffic control layer in LTE networks while we rely on comparison of ns-3 based emulation and testbed results to highlight design flaws in rate based AQM algorithms and to propose possible alternative solutions.

In the first example, we present the introduction of traffic control module on 3GPP stacks. We rely on ns-3 simulation to design a new software layer in 3GPP stack which aims to improve the traffic performance. The proposed layer aims to relieve the network operator management overhead while controlling the queueing delay of traffic flows on RLC queues of 3GPP stacks.

In the second example, we prove some design flaws in rate based AQM algorithms through comparison of ns-3 based emulation and testbed results, then we propose possible solutions. Indeed, we prove that the rates estimated from such algorithms depends on the specific flow control implemented by the device. Then, we propose a timestamp approach to improve the accuracy and an alternative flow control to consider it by design.

The rest of this chapter is structured as follows. Section 2 presents the design and evaluation of a software traffic control in 3GPP stacks. Section 3 proves design flaws in rate based AQM algorithms and propose possible different approaches. Finally, Section 4 concludes the chapter.

5.2 A software traffic control in 3GPP stack

The word bufferbloat entered the dictionary of scientists and researchers since 2011 when Jim Gettys first discovered it in residential settings. It is a term to define the existence of unreasonably large and full buffers inside any network. Over the years, technology standards have not kept up with research on this matter. Buffering policies are considered implementation-specific, and the designers' efforts went instead towards defining different cooperating layers in the 3GPP standard to provide a guaranteed Quality of Service (QoS) for paying customer. Different flow requirements are reflected in various bearer (a virtual transport pipe) properties, that so are served differently from the network.

Every User Equipment (UE) device has an "always on" default bearer activated at the time of device initialization. Other dedicated bearers, to serve flows with different priorities, are left for particular customers (such as public safety operators, or companies with significant contracts). Moreover, an Evolved Node B (eNB) encapsulates an end-to-end data bearer over a radio bearer for the over-the-air transmission. Unfortunately, opening and closing a bearer is not straightforward, and requires signaling between the network user and the network operator, increasing the network management effort. Then, the operator is prone to keep a fixed number of opened bearers to relieve the network management overhead. From our investigation in Android devices, and according to [57], the operator uses a single bearer for a regular user where all the user applications data (such as HTTPS video transfer, PUT/GET HTTPS request, a Skype call, a WhatsApp message) pass through. Therefore, in the rest of this case study, we assume this typical model in which all the user data goes into the default bearer.

In this case study, we present a novel way to solve bufferbloat problem in 3GPP mobile networks, hopefully shedding some light on this problem for the current standardization process of 5G New Radio (NR). In this proposal, we connect the Linux Traffic Control infrastructure on top of the 3GPP stack, employing a cross-layer approach to fight bufferbloat. In practice, the 3GPP and the IP level apply a mild form of *flow-control*, which will allow packets to be stored for a while inside the IP layer itself. The flow control is managed by the Byte Queue Limits (BQL) algorithm [51]. We present a preliminary evaluation of BQL in LTE as a means for keeping the size of the Radio Link Control (RLC) buffers at the minimum value that ensures that throughput is not impacted due to starvation. In this way, IP can do packet scheduling between different flows directed to the same bearer, prioritizing interactive traffic over bulk traffic by reusing existing packet schedulers and

Active Queue Management (AQM) algorithms, such as FQ-Codel [6]. The result is the possibility of performing QoS-based decisions even with a single bearer. We analyze our architectural proposal with ns-3 simulations, comparing in different scenarios the performance obtained with and without flow control, testing both BQL and DynRLC (one of the most promising proposals already present in the literature to dynamically sizing the RLC buffer size on the eNB). The ns-3 network simulator has an advanced Long Term Evolution (LTE) model (LENA) [58] and a complete representation of the traffic control layer at IP level, designed by taking inspiration from the Linux kernel [52]. In this way, we can show that by having a limited amount of buffering inside the LTE devices, the latency and the throughput performance improves dramatically, as well as providing fair scheduling between different flows. Being tied with the reality allows, in a line of principle, a device vendor to port our findings to existing Android devices with a minimum effort.

The rest is organized as follows: in Section 5.2.1 we present the related work, and why our proposal differs. In Section 5.2.2 we briefly explain part of the 3GPP stack affected by our proposal. In Section 5.2.3 we detail the core of our proposal and explain how we have implemented it in the ns-3 network simulator. In Section 5.2.4 we present the simulation results, and then in Section 5.2.5 we present our conclusions.

5.2.1 Related Work

In [57], authors characterize the traffic of the LTE networks to estimate the bufferbloat-inducted latency on the UEs. Trying to mitigate the problem, the authors introduced a flow control to limit the amount of data injected into the device firmware and propose a differentiation scheme in the Android Traffic Control (TC). The work does not consider the use of fairness scheme or the cross effect of their flow control on the AQM. Indeed, they study the poor performance of CoDel on the Android stack.

In [59], the authors propose an algorithm to dynamic sizing the RLC buffer for the eNBs, i.e., DynRLC, and introduce a per-flow fair queueing strategy at PDCP layer. A proposed flow control regulates the passing of packets between the two layers. They tune the proposed algorithm to reduce over queueing at RLC layer. The work does not consider either the use of AQM in their PDCP queueing scheme or the use of advanced fair queueing algorithms such as FQ-CoDel. Also, they do not consider and evaluate BQL.

Another attractive way to try to reduce the bufferbloat phenomena is to insert, at various levels in the stack, pure AQM algorithms to manage queues [60], [61]. However, employing an AQM (that can decide to drop packets if the algorithm has the perception that the amount of data stored in the buffer is not appropriate) can lead to issues when coupled with the MAC layer decisions. First, the AQM algorithms have to be modified to avoid the possibility of dropping a partial piece of a packet. In fact, at RLC level the packet can be segmented: what would happen with an algorithm such as CoDel, when the head is segmented and a piece transmitted, while the other portion (the new head) has to be dropped because its waiting

time is higher than the threshold? This issue is investigated in details in [62], with a modification to the CoDel algorithm proposed (and implemented) to avoid such effect. However, another question arises: in the uplink, the amount of data stored in the buffer is passed through Buffer Status Report (BSR) to the eNodeB. A similar message is exchanged between MAC and RLC (inside the eNodeB) in the downlink case. Therefore, dropping entire packets inside the radio stack is problematic for the MAC scheduler. Given the reported buffer status, some space may be reserved for a particular data radio bearer. Due to the unexpected drops, at the moment of the transmission, the bearer could have less than the previously reported data, leading to an under-usage of the resources, and therefore to degraded performance. For this reasons, we perform any necessary drop at a higher level.

In [63], the authors propose a dynamic adjustment of the TCP receiver window to reduce the excess of bytes in flight, that in turn is helping the network to buffer less overall data. The problem of this widespread idea (limiting the receiver window to restrict the sender is used successfully in many other fields) is that until all the devices are updated, network-level bufferbloat created by non-patched terminals will continue to impact flows coming from updated devices. Nevertheless, limiting the receiver window still has the potential issue to limit the overall transmission rate in case something is not such as the receiver is guessing.

Our proposal is to introduce a flow control similar to the one in [59] for both UE and eNB, on top of the LTE/NR protocol stack, that uses the BQL algorithm (already available in the Linux kernel) to dynamically size the RLC buffers. On this flow control, for both UE and eNB, we exploit a consolidated IP TC infrastructure, conversely to what in [57], [59] and [62]. Our strategy allows to avoid changes in the AQM algorithms, conversely to what in [62], since we keep a FIFO RLC buffer size regulated with BQL. Indeed, our approach avoids interference on the control plane of LTE and its mechanism of BSR. Finally, our proposal works regardless of the TCP receiver and sender windows and its congestion avoidance algorithms, conversely to what in [63].

5.2.2 Background

In the 3GPP model, the layer responsible for storing the data waiting for transmission over the air is RLC, with one buffer for each bearer. The RLC layer, to which Data Radio Bearer (DRB)s belong, provides that data fragmentation and reassembly feature. There are various modes for the RLC layer. For instance, there is the unacknowledged mode (UM), that will not perform retransmissions, and the acknowledged mode (AM), which contains retransmission buffers (per-DRB) in case the data is not ACKed by the receiver.

The algorithm that determines what data goes into the air is called Radio Data Scheduler (RDS), and is inside the eNB MAC layer. It decides how to fill in time and frequency each slot, according to some policy (e.g., round-robin or proportional fair). It uses as input the BSR messages that come from the RLC layer of each connected

UE to prepare the part of the frame that will be used by a UE to transmit uplink data. The eNodeB MAC layer uses similar messages (transmission opportunity, TxOpp) from its own RLC layer to prepare the frequency/time blocks that will be used to send downlink (from the UE perspective) data.

This model discourages the usage of AQM algorithms, as we argued in the Section 5.2.1, because dropping one or more packets at RLC layer with AQM algorithms invalidates the BSR information that the MAC uses to do the appropriate scheduling. So, in the majority of cases a FIFO queue is used: as the networking community knows since RFC 2308, uncontrolled FIFO queues lead to problems such as flow lockout and a difficult sizing process, which induce increasing latency. Recently an algorithm named DynRLC has been proposed [59] to dynamically control the RLC buffer size on the eNB. The idea is to define an RLC queueing time target, i.e., a DeLay Threshold (DLT), and sizing the RLC buffer to keep the queueing delay under DLT. DynRLC periodically estimates the average queueing time and defines the current size of the RLC buffer according to DLT.

5.2.3 Adding TC on top of the 3GPP stack

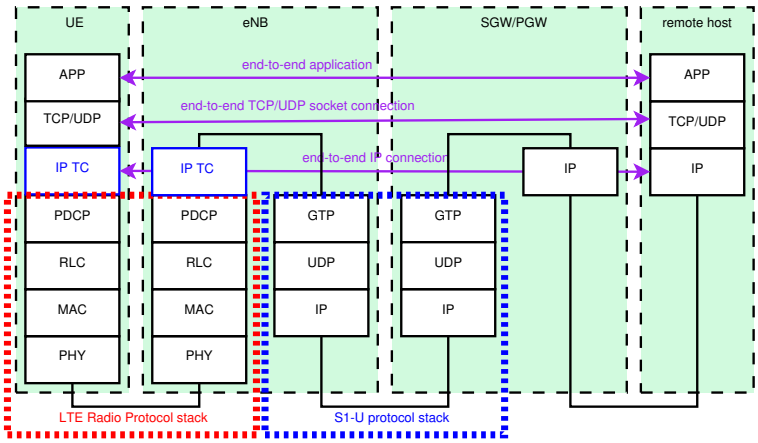


FIGURE 5.1: LTE-EPC data plane protocol stack with the introduction of TC on top of the LTE model.

In this section, we present our approach to fighting bufferbloat in LTE, which consists in placing the IP TC infrastructure on top of the LTE stack and adequately handling the flow control between the two. On most UE implementation, the 3GPP model for the Radio Access Network (RAN) stays inside the Link layer of the TCP/IP stack. Therefore, the implementation is limited to the flow control between the operating system and the LTE firmware, a practice that has already been proposed in the literature, as we reviewed in Section 5.2.1. For what regards eNB, to manage downlink flows, the infrastructure can be added as an independent application, following the recent trend on Multi-access Edge Computing [64].

As shown by Figure 5.1, the TC is introduced both on UEs and eNBs. On the UE side, IP packets generated by a local application are sent to TC and enqueued

into the (single) qdisc virtually linked to the LTE device. Android devices, having a Linux kernel, already include the traffic control infrastructure, as shown in [57].

On the eNB side, instead, the approach we use to introduce TC is rather distinctive. We think about the TC layer as a standalone application, which receives GTP decapsulated packets and then enqueues them into the qdisc corresponding to their destination UE. Over a single LTE device on the eNB, there are multiple scheduling qdiscs installed, one for each UE attached to the eNB. In the Linux kernel, this is possible by using a classifier qdisc (i.e., multi-queue aware qdisc) that can differentiate the packets based on their RTNI, enqueuing them in the appropriate child scheduling qdisc. Each child qdisc manages the packets destined to a particular transmission queue on the device, which is represented by the RLC buffer. We note that our approach exploits and extends the use of in-kernel, already existing, multi-queue aware qdiscs. The proposed method also requires the introduction of a flow control mechanism between the traffic control and the RLC buffers. In particular, we apply the flow control mechanism between each pair of RLC buffer and its connected qdisc. The flow control involves a soft-stopping mechanism (the qdisc do not move packets to the RLC queue based on the dynamic quota defined by BQL algorithm), as we explained in Section 4.2. We have also implemented the hard-stopping mechanism (the RLC buffer stops the qdisc when it can not enqueue more bytes), but in all our simulations it did not come into play. Indeed, the BQL quota was always less than the RLC buffer size.

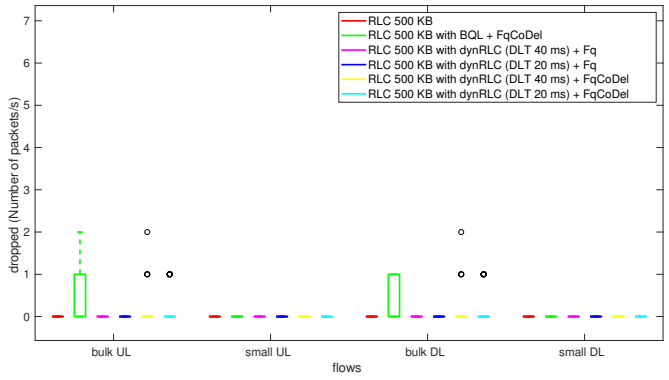
We evaluated the proposed approach by using ns-3 simulations [52], [53]. To this purpose, we implemented the necessary changes to allow RLC to notify the traffic control and the BQL library when a packet is enqueued or dequeued, to support flow control and BQL. We have implemented the *wake*, *stop*, *NotifyEnq*, *NotifyDeq* notifications. In our implementation, we assume the cooperation of the LTE device for both UE and eNB. Such an approach has been followed in [59] for the eNB.

If this cooperation is not feasible, for instance in a UE with closed firmware or other issues, implementing this flow control is still possible by exploiting information about the number of bytes enqueued and transmitted by the device firmware, such as in [57].

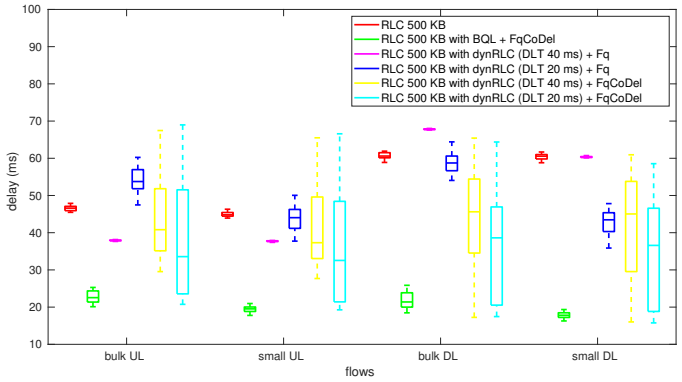
Finally, it is important to note that the support for BQL allows keeping a backlog in TC regardless of the actual physical RLC buffer size. Also, our approach allows exploring the use of different strategies tailored to the LTE/NR cases.

5.2.4 Results

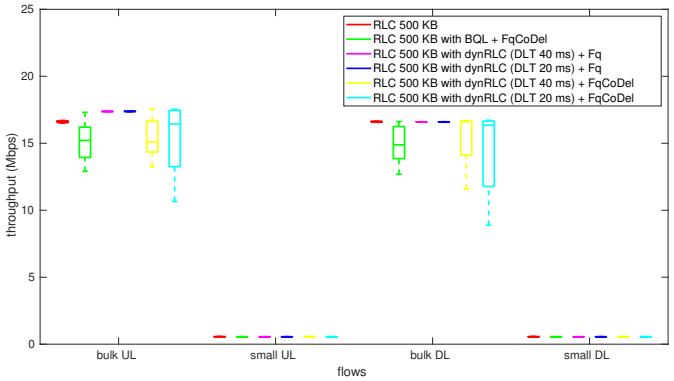
For all the experiments described from now on, the simple LTE topology reported in Figure 5.4 was used. A number of UEs are attached to the eNB which is connected to the EPC with a point-to-point link having a data rate of 10 Gbps and a delay of 10 ms. The basic idea is to evaluate the effects on the RAN performance due to reduced queueing at the RLC layer and to the introduction of flow-control and traffic-control in LTE.



(A) Dropped

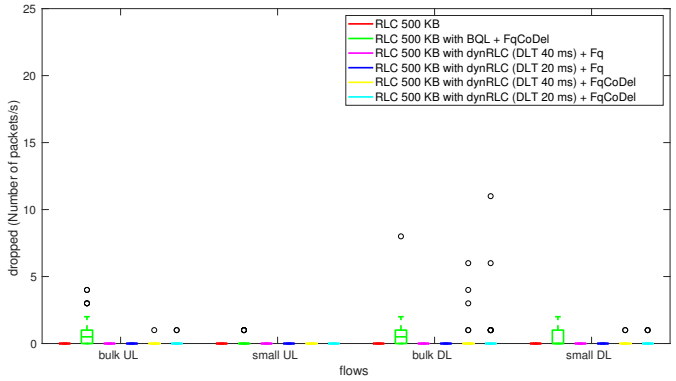


(B) Delay

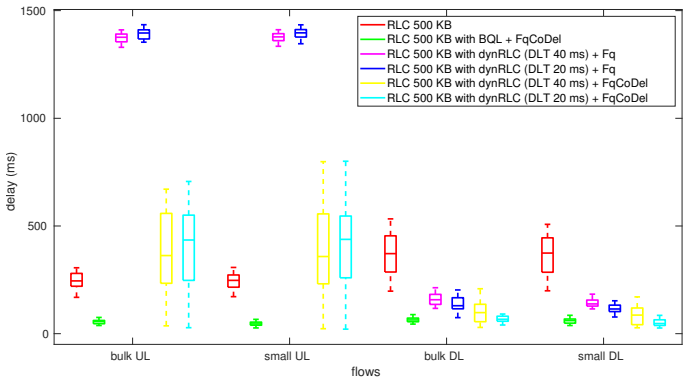


(C) Throughput

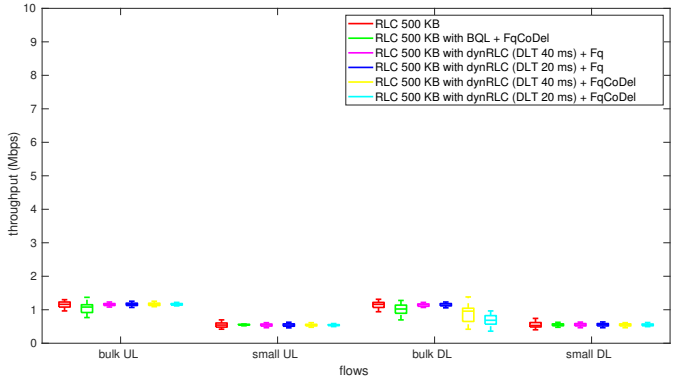
FIGURE 5.2: Evaluation of the impact of flow control and TC on LTE performance in single UE scenario.



(A) Dropped



(B) Delay



(C) Throughput

FIGURE 5.3: Evaluation of the impact of flow control and TC on LTE performance of one UE in a multiple UEs scenario. Other UEs present very similar results.

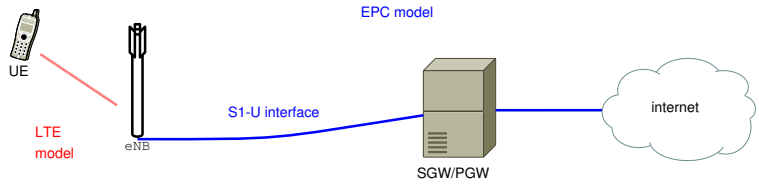


FIGURE 5.4: The network topology used for the validation tests.

We evaluate two scenarios. In the first one, a single UE is connected to the eNB; in the second one, ten UEs are connected to the eNB. UEs are in a fixed position. The RAN is configured with 25 Resource Blocks (5 MHz), and the maximum expected throughput of both UL and DL is approximately 16 Mbps in a single UE scenario. The default bearer is configured with RLC in UM mode.

In both scenarios, each UE manages four TCP flows. The basic idea is to saturate the UL and DL paths with two bulk streams (labelled as "bulk UL" and "bulk DL") while evaluating the performance of two interactive UL and DL small flows (labelled as "small UL" and "small DL"). The bulk streams generate constant traffic load able to saturate the network paths. The small flows are generated by *on-off* traffic sources that create application data at a rate of 500 Kbps in both scenarios. The traffic patterns are defined according to [57], [65], in particular for the number of concurrent TCP connections and the proportion between UL and DL traffic. The TCP version used is New Reno and the TCP segment size is 1400 bytes. The generated traffic is not marked with any IP QoS information, nor distributed through different LTE bearers.

We compare four approaches:

- In the first one, there is no flow control and all the data is queued at the RLC layer. The RLC buffer has a physical size of 500 KB according to [66] and [57].
- In the second approach, we evaluate our proposal. The flow control is regulated by BQL. The traffic control layer is configured with an FQ-CoDel qdisc on the UEs and the eNB. On the eNB, there is one instance of FQ-CoDel for each UE attached. The RLC buffer has a physical size of 500 KB.
- In the third approach, we evaluate the proposal in [59]. The flow control is regulated by DynRLC algorithm. TC is configured with an FQ qdisc on the UEs and the eNB. On the eNB, there is one instance of FQ for each UE attached. The RLC buffer has a physical size of 500 KB.
- In the fourth approach, we evaluate the case of flow control regulated by DynRLC and TC configured with FQ-CoDel. The traffic control layer is configured with an FQ-CoDel qdisc on the UEs and the eNB. On the eNB, there is one instance of FQ-CoDel for each UE attached. The RLC buffer has a physical size of 500 KB.

The parameters of BQL and FQ-CoDel are set to their default values. DynRLC is evaluated with two values of DLT defined according to [59].

Single UE scenario

In this scenario, we compare the four approaches listed above in the case only one UE is attached to the eNB. We aim to show that adding flow control allows better backlog management. Indeed, the TC packet scheduler can manage the backlog in order to reduce experienced delay as well as differentiate between small and bulk flows carried by the same bearer.

Results are reported in Figure 5.2. The presence of an AQM-based queue disc at the traffic-control level causes some packet drops in the bulk UL and bulk DL flows. The dropping is more consistent when the flow control is regulated by BQL (Figure 5.2a).

The one-way delay is reported in Figure 5.2b. In case of flow control regulated by BQL, all the flows have a reduced delay (median values reduced of 50% for the UL flows and of 60% for the DL flows) and FQ-CoDel provides a differentiation among flows with slightly reduced and constant delay for the small UL and small DL flows. DynRLC performs as expected in the DL path providing differentiation between bulk DL and small DL flows.However, in case DLT is 40 ms there is an increase in the bulk DL delay. In the UL path, there is differentiation among the flows in case DLT is 20 ms (with an increase in the bulk UL delay) while no differentiation in case DLT is 40 ms. The addition of FQ-CoDel induces further delay reduction among the flows in particular in case DLT is 20 ms. However, the experienced delays are highly variable.

The throughput (Figure 5.2c) is slightly reduced for the bulk UL and bulk DL when FQ-CoDel is used, according to the performance of AQMs with TCP flows, while the throughput is preserved for the small UL and small DL flows.

Multiple UEs scenario

This scenario aims to evaluate the effect of the introduction of TC when multiple UEs compete for the resources. Indeed, the eNB scheduler in this scenario divides the available bandwidth between ten UEs.

The results for the first UE are reported in Figure 5.3. The other UEs have no significant differences in performance. The presence of TC helps differentiate and manage the different IP flows on the UEs (UL path), and for each UE on the eNB (DL path).

The FQ-CoDel qdiscs on both UE and eNB drop some packets in particular of the bulk UL and bulk DL flows to control their sending rate (Figure 5.3a). Since the per-UE expected throughput is smaller than in the single UE scenario, the small flows can now contribute significantly to network congestion. Indeed, occasionally

dropping occurs in case of small UL and small DL flow, more consistent in case of flow control regulate by BQL, due to FQ-CoDel which try to keep reduced latency.

The one-way delay is reported in Figure 5.3b. In case of flow control regulated by BQL, all the flows have a reduced delay (median values reduced of 70%), and the qdiscs provide differentiation for the small flows in a similar way as the single UE scenario. We observe that the delay of all the flows for all the UEs is just slightly higher than the value of the single UE scenario. Conversely, without flow-control, the delay is twice higher for the UL flows and almost twice higher for the DL flows compared to those of the single UE scenario. The fact that the flows delay remains rather stable even in presence of a competing UEs is due to the FQ-CoDel scheduling algorithm which manages the queues based on the actual queueing time. DynRLC performs as expected in the DL path providing differentiation between bulk DL and small DL in all cases. The addition of FQ-CoDel induces further improvements and in case DLT 20 ms DynRLC perform as BQL. In the UL path, in this scenario, DynRLC shows worsening of the experienced delay in all cases. Indeed, DynRLC is designed for the DL path based on timeout triggered notifications of the BSR from the UE to eNB. This approach shows worsening of UL performance since the algorithm is not able to cut the UL RLC buffer size.

The throughput is reduced for the bulk UL and bulk DL flows in cases FQ-CoDel (a more significative reduction in case DL flows with DynRLC), while the throughput is preserved and more stable for the small UL and small DL flows (Figure 5.3c).

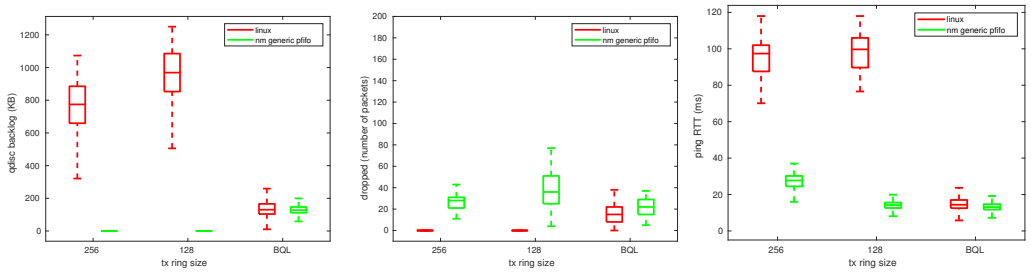
5.2.5 Conclusions

In this work, we investigate the bufferbloat problem in 3GPP protocol stacks proposing an innovative approach to keep stable and reduced network latency. The approach relieves the network operator from network management to open/close bearers. The idea is to connect the IP TC infrastructure on top of the 3GPP model and define a form of flow control between the layers. For the first time, we evaluate an algorithm, called BQL, as a means to dynamically move the flow control threshold (i.e., the amount of data that can be stored in RLC) in both UE and eNB. We compared the performance of BQL to another algorithm, called DynRLC, designed and presented in literature for dynamically sizing the RLC buffer on the eNB, as a means to reduce the RLC buffer size. We evaluated DynRLC also at the UE side, and tested it for the first time in a multiple UEs scenario.

In our evaluation, performed with the ns-3 simulator, BQL overcomes DynRLC. BQL efficacy leads to a significant performance increase of TC, which can perform packet scheduling on flows directed on the same bearer, preserving small flows over bulk streams.

Future works include the evaluation of the effectiveness of TC with BQL (or any other innovative algorithm to dynamically define a quota of suitable RLC buffer) in more dynamic scenarios, with variable channel conditions as well as handoff between base stations.

5.3 Flow control aware AQM algorithms



(A) Backlog (B) Dropped (C) RTT

FIGURE 5.5: Performance comparison of PIE in testbed and emulated scenario.

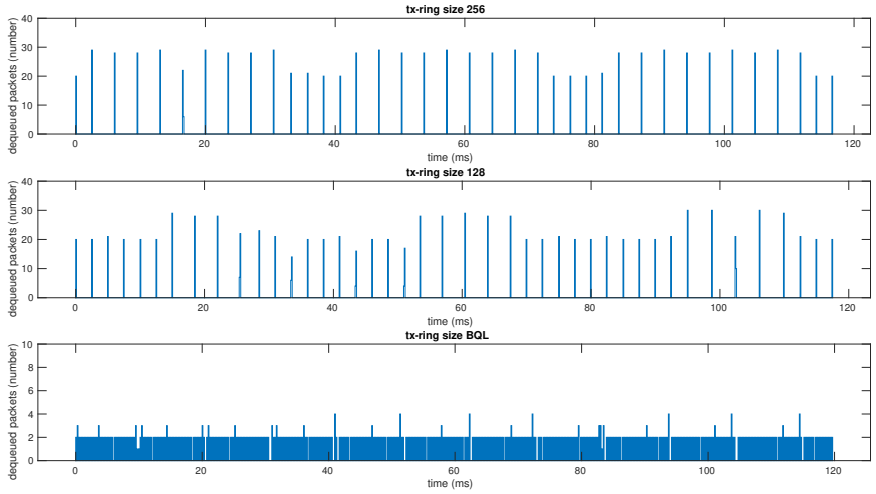
AQM algorithms have been proposed to keep the delay experienced in a queue under control. Basically, an AQM algorithm takes a decision to drop a packet when it recognizes a congestion status.

The first generation of AQM algorithms, i.e., RED [3], take the decision on dropping based on the queue *lenght*. The idea in RED is to control the queue length to keep reduced queueing delay. Then, the algorithm drops an incoming packet when the queue length is over a threshold. Unfortunately, such algorithms require tuning. Indeed, different network bandwidth requires different target queue length to keep the experienced delay under a threshold. This aspect has limited their deployments in communication networks. In recent years, AQM algorithms have been introduced which aim to overcome the limitation of the first generation. Algorithms such as CoDel and PIE have been proposed to keep the network queueing delay stable without requiring any tuning of the parameters.

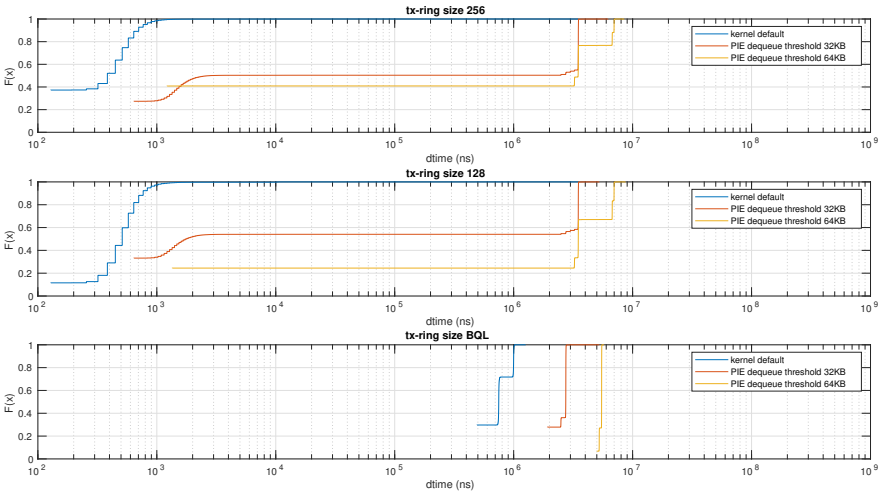
CoDel [4] introduced the idea to keep the queueing time stable by directly measuring it. Basically, CoDel attaches the *timestamp* of enqueue when a packet comes in its queue. Then, at dequeue time, CoDel evaluates the packet queueing time. If the time exceeds a given target, CoDel drops an outgoing packet to signal the congestion status to the upper layer.

PIE has been proposed to overcome the timestamp based approach [5]. It has been proposed by Cisco, standardized by IETF and used as a base for DOCSIS-PIE [67], [68]. The basic idea in PIE is to estimate the actual *departure rate*. Then, PIE estimates the queueing time based on the estimated departure rate. The algorithm decides if the next incoming packet should be dropped or not based on the queueing time and the desired target value.

The design of AQM algorithms usually entrusts on simulated environments such as ns-3. RED, CoDel and PIE have been proposed with the support of ns-2 simulations. Then, their effectiveness by design is proved in simulated environments. Unfortunately, a number of further factors in real systems, which are not considered in the design phase, can heavily restrict the effectiveness of these strategies. Some of



(A) Histograms of number of dequeued packets

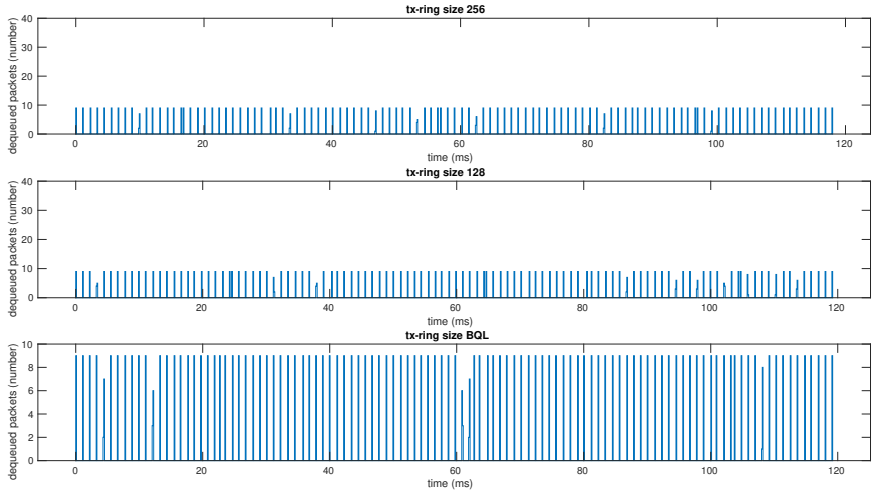


(B) ECDF of dtime

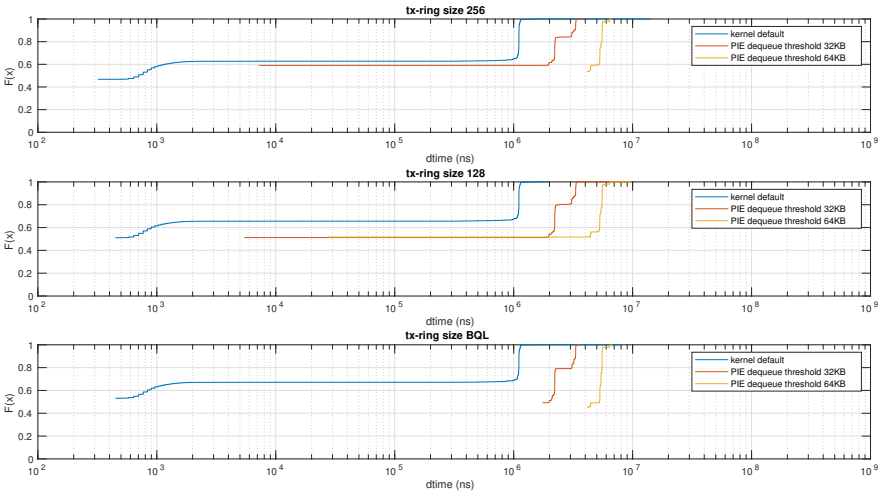
FIGURE 5.6: Hisograms of number of packets dequeued from PIE and ECDF of dtime for e1000e network adapter in testbed scenario.

them can require re-design of some parts of the algorithms. Our focus is on the impact of flow control, i.e., how the device starts/stops its queue, between the device and the AQM algorithm.

In this work we target PIE, a rate based AQM algorithm, to highlight the impact of flow control on its performance. We extensively test PIE in emulation through ns-3 [69] and testbed scenario following the RFC suggestions. We prove some PIE limitations due to design flaws. PIE has a critical parameter in its departure rate estimator and our work suggest the use of a timestamp based approach to improve its performance. Also, we propose an alternative approach to take explicitly into account the impact of flow control by design.



(A) Histograms of number of dequeued packets



(B) ECDF of dtime

FIGURE 5.7: Hisograms of number of packets dequeued from PIE and ECDF of dtime for tg3 network adapter in testbed scenario.

In the following, in Section 5.3.1 we provide details on PIE departure rate estimator and its possible effects due to different flow control implementations. Then, in Section 5.3.2 we propose a possible alternative flow control relation to reduce the impact of flow control by design. In Section 5.3.3 we prove the inaccurate evaluations of PIE and its flow control specific performance, then we outline the performance of PIE considering the flow control by design.

5.3.1 The PIE departure rate estimator

The dequeue rate estimation method employed by PIE is based on measurement cycles. After dequeuing a packet, if no measurement cycle is running and the current qdisc backlog exceeds a given *queue threshold*, a new measurement cycle is started.

The RFC propose a queue threshold from 10 KB to 64 KB. Given that a meaningful estimate of the dequeue rate can only be obtained starting from a certain number of packets dequeued, the idea is to avoid starting a new measurement cycle when the qdisc backlog is low. A counter keeps track of the amount of bytes dequeued since the beginning of the current measurement cycle. When such counter exceeds the queue threshold, the measurement cycle ends. The dequeue rate in the measurement cycle is then computed as the ratio of the amount of bytes dequeued (provided by the counter) to the duration of the cycle, i.e., *dtime*. The average dequeue rate is then derived as a weighted average of the last dequeue rate sample and the previous average dequeue rate value. The average dequeue rate is then used to determine the probability of dropping newly enqueued packets. Basically, the idea is to derive the dropping probability based on the comparison between the time it takes to dequeue the current qdisc backlog at the current estimated average dequeue rate and the target delay, which is a PIE configuration parameter.

The device buffer and flow control strategy can change the evaluated departure rate. Indeed, the PIE departure rate will evaluate the rate of writing in the device buffer when the measurements cycle starts and ends in a single burst. Indeed, the flow control mechanism usually reclaims further packets after the transmission of a number of packets, i.e., a batch of packets.

5.3.2 Considering the impact of flow control on AQM design

The traditional relation between device and AQM is inspired to pull based approach. Indeed, the device stops the upper layer when its queue is full and reclaims other packets when some space is made in its queue after the transmission over the medium. We explore the possibility to realize a push based approach in which the upper layer, i.e., the AQM, stops itself in order to avoid device overqueueing and effects due to device specific flow control implementations. In a congestion status, the idea is to estimate the actual available bandwidth at the device layer and then enforce a dequeue rate (from the upper layer) slightly higher than the available bandwidth.

In a fluid model, if we indicate with $C_{avl}(t)$ the available bandwidth at time t, with $C_{deq}(t)$ the enforced dequeue rate at time t, with $inflight(t)$ the number of bytes inflight in the device buffer and with $desiredInflight(t)$ a target of desired inflight avoiding overqueueing it is:

$$C_{deq}(t) = C_{avl}(t-\tau) + k \cdot [desiredInfligth(t) - inflight(t)] \tag{5.1}$$

The dequeue rate should be equal to the estimated available bandwidth, estimated with a delay τ, plus a contribution which follows the difference between the desired and actual inflight. The idea is to design scheduling disciplines in general and AQM algorithms in particular that are aware of device queue and then able to self-enforce the dequeue rate to avoid both overqueueing and starvation.

5.3.3 Results

In order to accurately evaluate the effectiveness of PIE and highlight possible flaws in its departure rate estimator, we compare the PIE performance in testbed scenario with the one in the emulated scenario through ns-3. For the comparison, we used the framework presented in chapter 4 based on e1000e bottleneck device. The results are reported in Figure 5.5. The expected PIE behavior is its ability to keep stable the backlog, thanks to dropping activity, and to control the experienced RTT. According to what we argue in chapter 2, the effectiveness of PIE should be more evident when the device buffer size is controlled by BQL.

The results in the emulated scenario show that PIE drops some packet (5.5b) in all the configurations aiming to control the backlog (5.5a) and keep the RTT stable (5.5c). PIE is more effective when BQL is enabled than when it is not. The estimated departure rate was about 12 MBps in 128 and 256 cases and of about 11 MBps in BQL case.

In the testbed scenario, the effectiveness of PIE is limited to BQL case. Indeed, in the other cases, PIE does not drop packets and the backlog is out of control. Hence, the experienced delay in cases 128 and 256 is higher than the one in BQL case. We note that when BQL is enable the accuracy of the PIE implemented in ns-3 is high.

In order to understand the unexpected behavior of PIE in the testbed scenario when BQL is not enabled, we evaluate how PIE works in that scenario. The results are reported in Figure 5.6. We evaluated the number of packets dequeued from PIE (5.6a) in a period of time corresponding to time requested to transmit an MTU in our testbed (120 μs for 1500 B at 100 Mbps) and the ECDF of the dtime parameter of PIE (5.6b). The results show that in cases 128 and 256 packets are dequeued in bursts of about 20 packets from PIE and enqueued to the device queue. Hence, since the default kernel queue threshold is of 10 KB, the evaluation of the departure rate starts and ends in a single burst. Since the burst requires few nanoseconds to conclude its transfer to the device (just the packet pointer is copied), the estimated data rate is very high. Conversely, in BQL cases, the number of dequeued packets is more smooth and of about 2 packets (in this cases are requested more dequeue events to trigger an evaluation). In order to estimate the departure rate, we traced the dtime parameter of PIE. The ECDF in Figure 5.6b shows that with a probability of 80% the estimated dtime is of about 600 ns in cases 128 and 256 and then the estimated departure rate is of 16 GB ps. In case of BQL, the estimated rate is of 10 MB ps. We experimented with queue threshold of 32 and 64 KB according to what defined in the RFC. The results show that the PIE estimator works in BQL cases while it has a limited effectiveness in the other cases also for threshold up to 64 KB. Indeed, just the 40% of the sampled dtime have a reasonable value (over 2 ms).

We evaluated the same metrics performance in testbed scenario on another network adapter. We target the tg3 adapter since it differs from e1000e adapter in the implemented flow control mechanism. The results are reported in Figure 5.7. The idea was to generalize our findings and highlight design guidelines. With tg3, packet

bursts of about 10 packets are dequeued in all cases. Hence the estimated departure rates are higher than expected since 70% of sampled dtime are under 1 ms for the kernel default case (queue threshold of 10 KB). In cases 128 and 256 the estimated data rate is higher than expected for 32 and 64 KB cases (50% of samples dtime under 2 ms), while is reasonable for BQL case with threshold of 32 and 64 KB.

Then, we evaluated by numeric simulation the performance of the proposed flow control strategy. We used the models reported in [5] for TCP and PIE dynamics. We added the device queue (with a physical dimension of 500 packets) and modeled two versions of flow control. The flow control push based has been modeled according to 5.1, i.e., PIE self-enforces a dequeue rate close to the available bandwidth estimated at device queue. The flow control pull based has been modeled to exploit all the device queue capacity, i.e., the device reclaims other packets when some space is available in its queue.

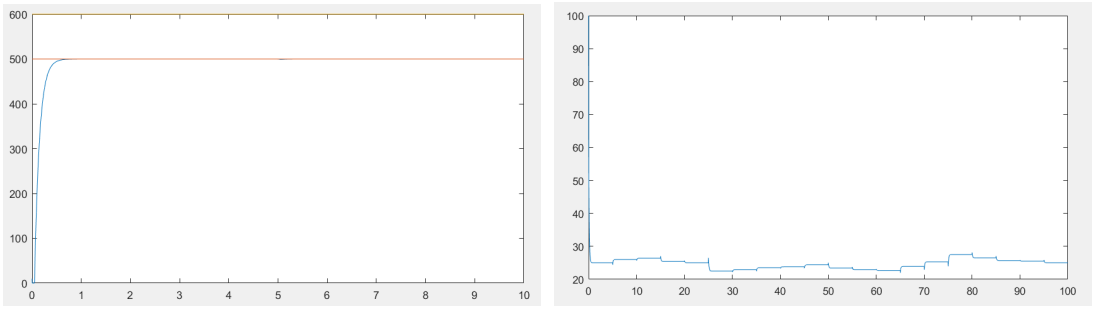
The push based flow control is able to reduce the usable device buffer avoiding overqueueing and starvation (Figure 5.8). Indeed, the device queue occupancy in the push based approach is of about 15 packets while it is always full (500 packets) in case of pull based approach. The PIE performance in term of dropping probability and queueing delay in its queue (without the additional queueing delay in the device queue) is reported in Figure 5.9. PIE is able to drop some packets during the simulation period in order to keep the delay under control. The PIE performance is the same in both flow control cases. However, the push based flow control reduces the device queueing delay reducing the overall delay.

5.3.4 Conclusions

The experiments show that the PIE departure rate estimator is a critical component. Indeed, it estimates higher than physically feasible data rate and its estimations depend from the specific implementation of flow control. Hence, we suggest the use of timestamp based approach to estimate the queueing time in PIE (applying the CoDel idea in PIE). We highlight that rate based algorithms designed mainly with simulation and emulation approaches can have flaws in their estimator. Then, we propose an alternative approach to flow control in order to take into account the impact of flow control by design. Future works include the evaluation of the proposed flow control in emulation and tesbed scenario.

5.4 Conclusions

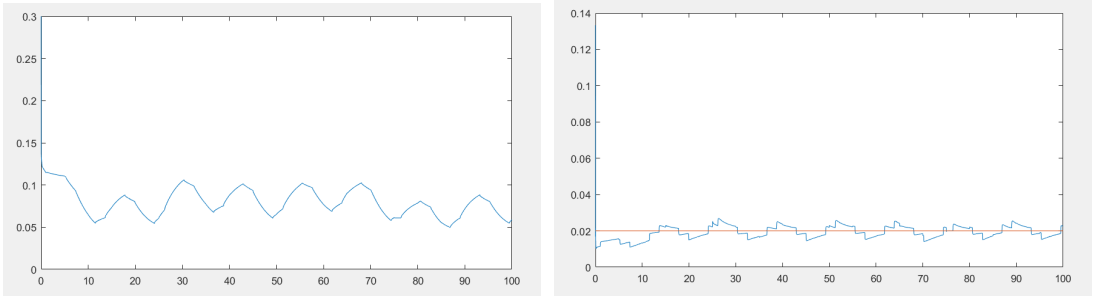
In this chapter, we presented two proposals of design and evaluation of network traffic control strategies. We presented the design of a traffic control strategy for 3GPP stacks by introducing the traffic control module on top of LTE. Then we tested the effectiveness of FQ-CoDel and BQL in LTE context. Also, we targeted rate based



(A) Occupancy of device queue in pull based approach

(B) Occupancy of device queue in push based approach

FIGURE 5.8: Effectiveness of push based approach to reduce the device queue usable buffer.



(A) Dropping probability

(B) Queueing delay

FIGURE 5.9: PIE performance.

AQM algorithms to highlight possible flaws. We show design flaws in rate estimation procedures of algorithms such as PIE and propose to use timestamp based approach.

Chapter 6

Conclusion

Network traffic control includes activities to increase the traffic flows delivery efficiency and to increase the network resources utilization. The activities include the use of network protocols and algorithms to control the performance of network traffic flows. Recently, the term bufferbloat has been coined to indicate the uncontrolled growth of the queueing time in communication networks. The bufferbloat depends, among others, on the general attitude of TCP, the most used transport protocol, to increase its sending rate until a packet is lost. Network elements usually are designed with per interface buffer in order to absorb packets burst. Such buffers are usually oversized and the presence of low-cost memory elements has increased further their dimension. In the context of traffic control, mitigation strategies has been proposed at various networking stack layers to reduce the queueing time. For instance, an algorithm called BQL has been proposed in the Linux kernel to reduce the device queueing time without impact on throughput. Linux relies on a traffic control infrastructure to perform flows regulation. The infrastructure includes packets schedulers which aim to choose the next packet to be transmitted to device. The most promising components in traffic control to fight bufferbloat are the AQM algorithms. The AQM algorithms aim to drop packets before the queue become full, i.e., when they recognize a possible congestion status, to notify the upper layer about a possible congestion status. Then, upper protocols responsive to packet dropping reduce their sending rate.

The first step to understanding the bufferbloat, to design and validate traffic control solutions, is the network experimentation in controlled environments. For instance, experimentation in testbed allows isolating external factors to focus the researchers' attention on the main causes of bufferbloat. With the increasing number of wireless networks, the simulation is gaining growing attention. Indeed, the simulation allows high flexibility in network experimentation, reproducing scenario with very large number of network elements and with early-stage communication technologies. However, the simulation challenge is the credibility. Indeed, the simulator should reproduce the real networking stack under test and the relation among the layers. One of the most used network simulators in academia and in research center is ns-3. ns-3 allows full stack and multi-protocols networks simulation. ns-3 has emulation support for the integration in testbeds. The design of the ns-3 simulator is

inspired to Linux networking stack. Unfortunately, ns-3 lacked a Linux equivalent traffic control layer. Hence, just a reduced number of traffic control strategies, e.g., RED, was available in the simulator and their use was allowed at the device layer, which is not what happens in Linux networking stack. Some design limitations did not allow the introduction of recent strategies such as FQ-CoDel. Also, in ns-3 was not possible to evaluate the impact of device buffer on AQM and packet schedulers. The available modules, e.g., RED, was not validated against real implementations. Indeed, some emulation limitations in ns-3 did not allow to realize a proper comparison scenario. Hence, bufferbloat study, design and evaluation of traffic control strategies based on simulation and emulation through ns-3 in technologies such as LTE and WiFi was inaccurate.

We analyzed the Linux networking stack with a focus on traffic control and device buffer and their relations. Then, we experimentally characterized the impact of device buffer on traffic control strategies such as packet schedulers and AQM algorithms. We designed and introduced in ns-3 a Linux equivalent traffic control module and the flow control between traffic control and the device buffer. Our design allowed the introduction of modern packets schedulers such as FQ-CoDel and the introduction of algorithms to dynamically size the device buffer such as BQL in ns-3. In order to validate the introduced modules against real implementations, we devised a new emulation methodology. We designed a new device in ns-3 which gain direct access to device buffers to increase the fidelity of emulation. We evaluated the new device and prove its emulation accuracy. Then, we validated, by using the new device, the traffic control module and RED, CoDel, PIE and FQ-CoDel. ns-3 has now the ability to accurately evaluate the effectiveness of traffic control on top of all the devices.

By using the validated modules, we designed a traffic control layer on 3GPP stacks and we highlight design flaws in rate based AQM algorithms. In the first case, we designed a software traffic control layer on top of LTE stack and evaluated its performance by using ns-3. The approach is inspired to Linux stack and we evaluated the performance of BQL and DynRLC, an algorithm proposed in LTE context to sizing the RLC buffer, to keep traffic control effective. We evaluated the performance in multiple UEs scenario and observed the invariance of the experienced delay of a single UE regardless of the number of connected UEs thanks to FQ-CoDel (which limit the queueing time based on the actual departure rate). In the second case, we prove some design flaws in rate based AQM algorithms due to the impact of flow control between the device and traffic control. Then we propose to use timestamp-based approach (to estimate the queueing time) and an alternative flow control to take into account its effects by design. The idea is to enforce a traffic control sending rate close to the estimated available bandwidth.

In this work, we improved the adherence of the ns-3 simulator stack to Linux networking stack. The stack adherence allows to reproduce behaviors occurring in real systems, e.g., allow to reproduce the impact of device buffer on AQM algorithms,

and to experiments in a more accurate manner, e.g., experiments of the effectiveness of traffic control in wireless networks. The new emulation methodology we devised in this work is quite general and improve the emulation credibility. The methodology applies to simulators and emulators which rely on standard socket mechanisms to communicate with real networks. The new device implemented in ns-3 allows to validate the simulated stack up to the traffic control module. This outcome increase the simulation credibility with ns-3 since traffic control has not a negligible impact on performance parameters usually evaluated such as packet loss, throughput, and delay. Also, the validation of AQM algorithms allows increasing their credibility in simulated and emulated scenarios with ns-3. Hence, the use of AQM algorithms, e.g., CoDel, in any queue of the stack can now be considered reliable.

The ns-3 simulator is now a reliable tool to design and evaluate traffic control strategies to guarantee high network utilization and low delay. Other AQM algorithms can now be designed and tested through ns-3. For instance, AQM algorithms based on different congestion measures can now be introduced and evaluated accurately in ns-3. Also, the interaction between AQM algorithms and MAC strategies to improve the network utilization, e.g., frame aggregation in WiFi, LTE BSR mechanism, BQL or other dynamic sizing algorithms, can be now explored and characterized. Alternative form of flow control can be proposed and evaluated through ns-3, e.g., TCP small queue alternatives, to reduce the in node buffering. We highlight that the proposed validation methodology applies to simulators which use standard socket mechanism to communicate with real networks. Finally, the device implemented in ns-3 to increase the emulation credibility can be used to validate other modules, e.g., TCP congestion mechanism such as BBR, to increase the transport layer simulation credibility, or to asses the accuracy of other ns-3 algorithms.

Bibliography

[1] W. Almesberger, J. Salim, and A. Kuznetsov, "Differentiated services on Linux", in *Proceedings of the Global Telecommunications Conference (Globecom)*, vol. 1B, IEEE, 1999, pp. 831–836. DOI: 10.1109/GLOCOM.1999.830189.

[2] J. Gettys and K. Nichols, "Bufferbloat: Dark Buffers in the Internet", *Commun. ACM*, vol. 55, no. 1, pp. 57–65, Jan. 2012, ISSN: 0001-0782. DOI: 10.1145/2063176.2063196. [Online]. Available: http://doi.acm.org/10.1145/2063176.2063196.

[3] S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance", *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, 1993, ISSN: 1063-6692. DOI: 10.1109/90.251892. [Online]. Available: http://dx.doi.org/10.1109/90.251892.

[4] K. Nichols, V. Jacobson, A. McGregor, and J. Iyengar, "Controlled Delay Active Queue Management", IETF, draft-ietf-aqm-codel-07, 2017.

[5] R. Pan, P. Natarajan, C. Piglione, M. S. Prabhu, V. Subramanian, F. Baker, and B. VerSteeg, "PIE: A lightweight control scheme to address the bufferbloat problem", in *Proceedings of HPSR)*, 2013, pp. 148–155. DOI: 10.1109/HPSR.2013.6602305.

[6] T. Hoeiland-Joergensen, P. McKenney, D. Taht, J. Gettys, and E. Dumazet, "The FlowQueue-CoDel Packet Scheduler and Active Queue Management Algorithm", IETF, draft-ietf-aqm-fq-codel-06, 2016.

[7] T. Høiland-Jørgensen, D. Täht, and J. Morton, "Piece of CAKE: A comprehensive queue management solution for home gateways", *CoRR*, vol. abs/1804.07617, 2018. arXiv: 1804.07617. [Online]. Available: http://arxiv.org/abs/1804.07617.

[8] T. Henderson. (2009). Network simulation and emulation in a testbed era, [Online]. Available: https://slideplayer.com/slide/2524318/.

[9] G. Carneiro, H. Fontes, and M. Ricardo, "Fast prototyping of network protocols through ns-3 simulation model reuse", *Simulation Modelling Practice and Theory*, vol. 19, no. 9, pp. 2063–2075, 2011, ISSN: 1569-190X. DOI: https://doi.org/10.1016/j.simpat.2011.06.002.

[10] S. Hemminger, "Network emulation with NetEm", in *Proceedings of Australia's 6th National Linux Conference (LCA)*, 2005, pp. 1–9.

[11] L. Rizzo, "Dummynet: A Simple Approach to the Evaluation of Network Protocols", *SIGCOMM Comput. Commun. Rev.*, vol. 27, no. 1, pp. 31–41, 1997, ISSN: 0146-4833. DOI: 10.1145/251007.251012. [Online]. Available: http://doi.acm.org/10.1145/251007.251012.

[12] G. F. Riley and T. R. Henderson, "The ns-3 Network Simulator", in *Modeling and Tools for Network Simulation*, K. Wehrle, M. Gunes, and J. Gross, Eds., Springer, Berlin, Heidelberg, 2010, pp. 15–34. DOI: 10.1007/978-3-642-12331-3_2.

[13] A. Varga, "OMNet++", in *Modeling and Tools for Network Simulation*, K. Wehrle, M. Gunes, and J. Gross, Eds., Springer, Berlin, Heidelberg, 2010, pp. 35–59. DOI: 10.1007/978-3-642-12331-3_3.

[14] B. P. Swenson and G. F. Riley, "Implementing explicit congestion notification in ns-3", in *Proceedings of the 2014 Workshop on ns-3*, ser. WNS3 '14, Atlanta, Georgia, USA: ACM, 2014, pp. 1–8, ISBN: 978-1-4503-3003-9. DOI: 10.1145/2630777.2630782. [Online]. Available: http://doi.acm.org/10.1145/2630777.2630782.

[15] T. Høiland-Jørgensen, P. Hurtig, and A. Brunstrom, "The good, the bad and the wifi: Modern aqms in a residential setting", *Computer Networks*, vol. 89, pp. 90–106, 2015. DOI: 10.1016/j.comnet.2015.07.014. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84940377644&partnerID=40&md5=38873e7f2234bd6dd6326c3c0a85292d.

[16] Y. Guo, F. Qian, Q. A. Chen, Z. M. Mao, and S. Sen, "Understanding on-device bufferbloat for cellular upload", in *Proceedings of the 2016 Internet Measurement Conference*, ser. IMC '16, Santa Monica, California, USA: ACM, 2016, pp. 303–317, ISBN: 978-1-4503-4526-2. DOI: 10.1145/2987443.2987490. [Online]. Available: http://doi.acm.org/10.1145/2987443.2987490.

[17] G. Appenzeller, I. Keslassy, and N. McKeown, "Sizing Router Buffers", *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 4, pp. 281–292, 2004, ISSN: 0146-4833. DOI: 10.1145/1030194.1015499.

[18] S. Floyd, R. Gummadi, and S. Shenker, "Adaptive RED: An Algorithm for Increasing the Robustness of RED Active Queue Management", Tech. Rep., 2001.

[19] V. Rosolen, O. Bonaventure, and G. Leduc, "A RED Discard Strategy for ATM Networks and Its Performance Evaluation with TCP/IP Traffic", *SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 3, pp. 23–43, 1999, ISSN: 0146-4833. DOI: 10.1145/505724.505728.

[20] K. Zhou, K. L. Yeung, and V. O. Li, "Nonlinear RED: A simple yet efficient active queue management scheme", *Computer Networks*, vol. 50, no. 18, pp. 3784–3794, 2006, ISSN: 1389-1286. DOI: http://dx.doi.org/10.1016/j.comnet.2006.04.007.

[21] G. Abbas, Z. Halim, and Z. H. Abbas, "Fairness-Driven Queue Management: A Survey and Taxonomy", *IEEE Communications Surveys Tutorials*, vol. 18, no. 1, pp. 324–367, 2016, ISSN: 1553-877X. DOI: 10.1109/COMST.2015.2463121.

[22] W. chang Feng, K. G. Shin, D. D. Kandlur, and D. Saha, "The BLUE active queue management algorithms", *IEEE/ACM Transactions on Networking*, vol. 10, no. 4, pp. 513–528, 2002, ISSN: 1063-6692. DOI: 10.1109/TNET.2002.801399.

[23] S. Kunniyur and R. Srikant, "Analysis and Design of an Adaptive Virtual Queue (AVQ) Algorithm for Active Queue Management", *SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 4, pp. 123–134, 2001, ISSN: 0146-4833. DOI: 10.1145/964723.383069.

[24] S. Athuraliya, S. H. Low, V. H. Li, and Q. Yin, "REM: active queue management", *IEEE Network*, vol. 15, no. 3, pp. 48–53, 2001, ISSN: 0890-8044. DOI: 10.1109/65.923940.

[25] W.-C. Feng, D. D. Kandlur, D. Saha, and K. G. Shin, "Stochastic fair blue: A queue management algorithm for enforcing fairness", in *Proceedings of IEEE INFOCOM*, vol. 3, 2001, pp. 1520–1529. DOI: 10.1109/INFCOM.2001.916648.

[26] L. Xue, S. Kumar, C. Cui, P. Kondikoppa, C.-H. Chiu, and S.-J. Park, "Towards fair and low latency next generation high speed networks: {afcd} queuing", *Journal of Network and Computer Applications*, vol. 70, pp. 183 –193, 2016, ISSN: 1084-8045. DOI: http://dx.doi.org/10.1016/j.jnca.2016.03.021.

[27] C. V. Hollot, V. Misra, D. Towsley, and W.-B. Gong, "On designing improved controllers for AQM routers supporting TCP flows", in *Proceedings of IEEE INFOCOM*, vol. 3, 2001, pp. 1726–1734. DOI: 10.1109/INFCOM.2001.916670.

[28] K. De Schepper, O. Bondarenko, I.-J. Tsang, and B. Briscoe, "PI2: A Linearized AQM for Both Classic and Scalable TCP", in *Proceedings of CoNEXT*, Irvine, California, USA: ACM, 2016, pp. 105–119, ISBN: 978-1-4503-4292-6. DOI: 10.1145/2999572.2999578.

[29] Y. Gong, D. Rossi, C. Testa, S. Valenti, and M. Taht, "Fighting the bufferbloat: On the coexistence of AQM and low priority congestion control", *Computer Networks*, vol. 65, pp. 255 –267, 2014, ISSN: 1389-1286. DOI: http://dx.doi.org/10.1016/j.bjp.2014.01.009.

[30] M. Kuehlewind, G. Hazel, S. Shalunov, and J. Iyengar, "Low Extra Delay Background Transport (LEDBAT)", Internet Engineering Task Force (IETF), RFC 6817, 2012.

[31] G. Raina, S. Manjunath, S. Prasad, and K. Giridhar, "Stability and Performance Analysis of Compound TCP With REM and Drop-Tail Queue Management", *IEEE/ACM Transactions on Networking*, vol. 24, no. 4, pp. 1961–1974, 2016, ISSN: 1063-6692. DOI: 10.1109/TNET.2015.2448591.

[32] L. Rizzo and P. Valente, "On service guarantees of fair-queueing schedulers in real systems", *Computer Communications*, vol. 67, pp. 34 –44, 2015, ISSN: 0140-3664. DOI: http://dx.doi.org/10.1016/j.comcom.2015.06.009.

[33] R. Prasad, M. Jain, and C. Dovrolis, "Effects of interrupt coalescence on network measurements", in *Proceedings of PAM*, Springer Berlin Heidelberg, 2004, pp. 247–256, ISBN: 978-3-540-24668-8. DOI: 10.1007/978-3-540-24668-8_25.

[34] M. Shreedhar and G. Varghese, "Efficient fair queueing using deficit round-robin", *IEEE/ACM Transactions on Networking*, vol. 4, no. 3, pp. 375–385, 1996, ISSN: 1063-6692. DOI: 10.1109/90.502236. [Online]. Available: http://dx.doi.org/10.1109/90.502236.

[35] P. McKenney, "Stochastic fairness queueing", in *Proceedings of the Ninth Annual Joint Conference of the IEEE Computer and Communication Societies (INFOCOM)*, IEEE, 1990, pp. 733–740. DOI: 10.1109/INFCOM.1990.91316.

[36] F. Checconi, L. Rizzo, and P. Valente, "Qfq: Efficient packet scheduling with tight guarantees", *IEEE/ACM Transactions on Networking*, vol. 21, no. 3, pp. 802–816, 2013, ISSN: 1063-6692. DOI: 10.1109/TNET.2012.2215881. [Online]. Available: http://dx.doi.org/10.1109/TNET.2012.2215881.

[37] K. Fall, "Network emulation in the VINT/NS simulator", in *Proceedings IEEE International Symposium on Computers and Communications*, 1999, pp. 244–250. DOI: 10.1109/ISCC.1999.780820.

[38] J. Ahrenholz, C. Danilov, T. R. Henderson, and J. H. Kim, "CORE: A real-time network emulator", in *IEEE Military Communications Conference*, 2008, pp. 1–7. DOI: 10.1109/MILCOM.2008.4753614.

[39] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible Network Experiments Using Container-based Emulation", in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '12, Nice, France: ACM, 2012, pp. 253–264, ISBN: 978-1-4503-1775-7. DOI: 10.1145/2413176.2413206. [Online]. Available: http://doi.acm.org/10.1145/2413176.2413206.

[40] S. Aketa, T. Hirofuchi, and R. Takano, "DEMU: A DPDK-based network latency emulator", in *IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, 2017, pp. 1–6. DOI: 10.1109/LANMAN.2017.7972145.

[41] D. Camara, H. Tazaki, E. Mancini, T. Turletti, W. Dabbous, and M. Lacage, "DCE: Test the real code of your protocols and applications over simulated networks", *IEEE Communications Magazine*, vol. 52, no. 3, pp. 104–110, 2014, ISSN: 0163-6804. DOI: 10.1109/MCOM.2014.6766093.

[42] S. Y. Wang and C. C. Lin, "NCTUns 6.0: A Simulator for Advanced Wireless Vehicular Network Research", in *IEEE 71st Vehicular Technology Conference*, 2010, pp. 1–2. DOI: 10.1109/VETECS.2010.5494212.

[43] M. Pieska and A. Kassler, "TCP performance over 5G mmWave links - Trade-off between capacity and latency", in *13th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, 2017, pp. 385–394. DOI: 10.1109/WiMOB.2017.8115776.

[44] A. Flammini, E. Sisinni, and F. Tramarin, "IEEE 802.11s performance assessment: From simulations to real-world experiments", in *IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*, 2017, pp. 1–6. DOI: 10.1109/I2MTC.2017.7969752.

[45] E. Weingartner, H. Vom Lehn, and K. Wehrle, "Device driver-enabled wireless network emulation", in *4th International ICST Conference on Simulation Tools and Techniques (SIMUTools)*, 2011, pp. 188–197. DOI: 10.4108/icst.simutools.2011.245584.

[46] T. Kawai, S. Kaneda, M. Takai, and H. Mineno, "A Virtual WLAN Device Model for High-Fidelity Wireless Network Emulation", *ACM Transactions on Modeling and Computer Simulation*, vol. 27, no. 3, pp. 1–24, 2017, ISSN: 1049-3301. DOI: 10.1145/3067664.

[47] S. Yadav, M. S. Gaur, and V. Laxmi, "Ns-3 emulation on ORBIT testbed", in *International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, 2013, pp. 616–619. DOI: 10.1109/ICACCI.2013.6637243.

[48] J. Núñez-Martínez, J. Baranda, and J. Mangues-Bafalluy, "Experimental evaluation of self-organized backpressure routing in a wireless mesh backhaul of small cells", *Ad Hoc Networks*, vol. 24, pp. 103 –114, 2015, ISSN: 1570-8705. DOI: https://doi.org/10.1016/j.adhoc.2014.07.021.

[49] H. Fontes, R. Campos, and M. Ricardo, "Improving ns-3 Emulation Support in Real-World Networking Scenarios", *EAI Endorsed Transactions on Industrial Networks and Intelligent Systems*, vol. 16, no. 9, Aug. 2015. DOI: 10.4108/eai.24-8-2015.2261074.

[50] L. Rizzo, "Netmap: A Novel Framework for Fast Packet I/O", in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12, Boston, MA: USENIX Association, 2012, pp. 1–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342821.2342830.

[51] P. Imputato and S. Avallone, "An analysis of the impact of network device buffers on packet schedulers through experiments and simulations", *Simulation Modelling Practice and Theory*, vol. 80, no. Supplement C, pp. 1–18, 2018, ISSN: 1569-190X. DOI: https://doi.org/10.1016/j.simpat.2017.09.008.

[52] P. Imputato and S. Avallone, "Design and implementation of the traffic control module in ns-3", in *Proceedings of the Workshop on Ns-3*, ser. WNS3 '16, Seattle, WA, USA: ACM, 2016, pp. 1–8, ISBN: 978-1-4503-4216-2. DOI: 10.1145/2915371.2915382. [Online]. Available: http://doi.acm.org/10.1145/2915371.2915382.

[53] P. Imputato and S. Avallone, "Traffic differentiation and multiqueue networking in ns-3", in *Proceedings of the Workshop on Ns-3*, ser. WNS3 '17, Porto, Portugal: ACM, 2017, pp. 79–86, ISBN: 978-1-4503-5219-2. DOI: 10.1145/3067665.3067677. [Online]. Available: http://doi.acm.org/10.1145/3067665.3067677.

[54] Intel Corporation. (2013). Impressive Packet Processing Performance Enables Greater Workload Consolidation. White paper.

[55] *PF_RING ZC*, http://www.ntop.org/products/pf_ring/pf_ring-zc-zero-copy/, Accessed: 2018-06-11.

[56] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, "Comparison of frameworks for high-performance packet IO", in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2015, pp. 29–38. DOI: 10.1109/ANCS.2015.7110118.

[57] Y. Guo, F. Qian, Q. A. Chen, Z. M. Mao, and S. Sen, "Understanding On-device Bufferbloat for Cellular Upload", in *Proceedings of the 2016 Internet Measurement Conference*, ser. IMC '16, Santa Monica, California, USA: ACM, 2016, pp. 303–317, ISBN: 978-1-4503-4526-2. DOI: 10.1145/2987443.2987490. [Online]. Available: http://doi.acm.org/10.1145/2987443.2987490.

[58] G. Piro, N. Baldo, and M. Miozzo, "An LTE module for the Ns-3 Network Simulator", in *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, ser. SIMUTools '11, Barcelona, Spain: ICST, 2011, pp. 415–422, ISBN: 978-1-936968-00-8. [Online]. Available: http://dl.acm.org/citation.cfm?id=2151054.2151129.

[59] R. Kumar, A. Francini, S. Panwar, and S. Sharma, "Dynamic control of rlc buffer size for latency minimization in mobile ran", in *2018 IEEE Wireless Communications and Networking Conference (WCNC)*, 2018, pp. 1–6. DOI: 10.1109/WCNC.2018.8377190.

[60] T. Høiland-Jørgensen, MichałKazior, D. Täht, P. Hurtig, and A. Brunstrom, "Ending the Anomaly: Achieving Low Latency and Airtime Fairness in WiFi", in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, CA: USENIX Association, 2017, pp. 139–151, ISBN: 978-1-931971-38-6. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/hoilan-jorgesen.

[61] M. Zhang, M. Mezzavilla, J. Zhu, S. Rangan, and S. S. Panwar, "The bufferbloat problem over intermittent multi-gbps mmwave links", *CoRR*, vol. abs/1611.02117, 2016. arXiv: 1611.02117. [Online]. Available: http://arxiv.org/abs/1611.02117.

[62] Y. Dai, V. Wijeratne, Y. Chen, and J. Schormans, "Channel Quality Aware Active Queue Management in Cellular Networks", in *2017 9th Computer Science and Electronic Engineering (CEEC)*, 2017, pp. 183–188. DOI: 10.1109/CEEC.2017.8101622.

[63] H. Jiang, Y. Wang, K. Lee, and I. Rhee, "Tackling Bufferbloat in 3g/4g Networks", in *Proceedings of the 2012 Internet Measurement Conference*, ser. IMC '12, Boston, Massachusetts, USA: ACM, 2012, pp. 329–342, ISBN: 978-1-4503-1705-4. DOI: 10.1145/2398776.2398810. [Online]. Available: http://doi.acm.org/10.1145/2398776.2398810.

[64] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing - A key technology towards 5G", ETSI White Paper.

[65] X. Zhang, Y. Xu, H. Hu, Y. Liu, Z. Guo, and Y. Wang, "Profiling Skype video calls: Rate control and video quality", *2012 Proceedings IEEE INFOCOM*, pp. 621–629, 2012.

[66] R. Bestak, P. Godlewski, and P. Martins, "RLC buffer occupancy when using a TCP connection over UMTS", in *The 13th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, vol. 3, 2002, 1161–1165 vol.3. DOI: 10.1109/PIMRC.2002.1045210.

[67] B. Prisco. (2015). Review: Proportional integral controller enhanced (pie) active queue management (aqm), [Online]. Available: http://www.bobbriscoe.net/projects/latency/piervw_tr.pdf.

[68] CableLabs. (2014). Active queue managements in docsis 3.x cable modems, [Online]. Available: https://www.cablelabs.com/wp-content/uploads/2014/06/DOCSIS-AQM_May2014.pdf.

[69] P. Imputato, S. Avallone, and T. Pecorella, "Network emulation support in ns-3 through kernel bypass techniques", in *Proceedings of the 11th EAI International Conference on Performance Evaluation Methodologies and Tools*, ser. VALUETOOLS 2017, Venice, Italy: ACM, 2017, pp. 259–260, ISBN: 978-1-4503-6346-4. DOI: 10.1145/3150928.3150966. [Online]. Available: http://doi.acm.org/10.1145/3150928.3150966.