Applying queueing theory to the Linux networking stack

Toke Høiland-Jørgensen
toke@toke.dk

Master's thesis
Computer Science and Mathematics
Roskilde University

29th August 2013

Abstract

Queueing delays are a potentially significant source of latency in modern computer networks, especially when links come under load. This phenomenon has been coined *bufferbloat*; and controlling queueing in intermediate gateway devices plays an important role in mitigating it. This thesis seeks to gain further insight into the behaviour of queueing in networks, using the Linux kernel as a realistic example of a real-world system and modelling it using mathematical queueing theory.

Three queueing theory models have been formulated to model the queueing mechanisms of a Linux-based gateway, and the numerical model predictions have been compared to the observed values. The goal of the experiments has been to ascertain how well the models predict the observed queue length distributions, how well they predict the queue waiting times, and to what extent the fairness queueing implementation in the Linux kernel can be considered to have completely independent queues.

For the M/M/1/K and M/D/1/K models, traffic was generated specifically to match the model assumptions (with exponentially distributed arrival intervals and exponential or deterministic service times). The results of this show some correlation between model predictions and observations for these models, even though it does not hold up to a rigorous statistical test. For TCP traffic (being modelled using the MMPP/D/1/K queue, based on the Markov-modulated Poisson process) this correlation is almost completely absent; although this is most likely due to the fact that the numerical calculations for the MMPP model produce dubious results that are almost entirely independent of the choice of parameterisation: remedying this has not been possible.

The observations of the system indicate that *for traffic that matches the assumptions of the models*, the queueing theory model predictions can be used as a reasonable qualitative indicator of the system behaviour. Furthermore, the experiments show the efficiency of packet transmission in the Linux kernel, which causes the queues to be more likely to be empty than predicted. Also apparent in the results data is the difference in behaviour between traffic with exponential and deterministic service times and between different traffic rates, which is to some extent matched by the model predictions and which validates the measurement and traffic generation methodology. Lastly, the interaction of the TCP control loop with the queue is apparent in the observations, corresponding well to what is expected from the theory of TCP operations.

Finally, the results show that fairness queueing can to a large extent be thought of as a set of completely separate queues. However, in the Linux kernel implementation, the queueing space is shared between individual queues, making the queue lengths be distributed differently from what a separate queue interpretation would suggest.

Resume

Forsinkelser pga kødannelse er en potentielt væsentlig kilde til høj ventetid i moderne computernetværk, især når netværksforbindelserne bliver belastet. Dette fænomen har fået navnet *bufferbloat*, og kontrol af kødannelsen i mellemliggende gateway-enheder spiller en vigtig rolle i at afbøde det. Dette speciale søger at opnå yderligere indsigt i kø-adfærd i netværk, ved hjælp af Linux-kernen som en realistisk eksempel på et virkeligt system, som modelleres med matematisk kø-teori.

Tre køteoretiske modeller er blevet formuleret til at modellere kø-mekanismerne i en Linux-baseret gateway, og de numeriske modelberegninger er blevet sammenlignet med de observerede værdier. Målet med forsøgene har været at fastslå, hvor godt modellerne forudsiger distributionen af de observerede kølængder, hvor godt de forudsiger kø-ventetiderne, og i hvilket omfang *fairness queueing*-implementeringen i Linux-kernen kan anses som bestående af helt uafhængige køer.

For M/M/1/K og M/D/1/K–modellerne blev trafik genereret specielt til at matche modelforudsætningerne (med eksponentielt fordelte ankomstintervaller, og eksponentialfordelte eller deterministiske behandlingstider). Resultaterne af disse forsøg viser en hvis korrelation mellem modelberegninger og observationer for disse modeller, selvom korrelationen ikke klarer en stringent statistisk test. For TCP-trafik (modelleret ved hjælp af en MMPP/D/1/K–kø baseret på den Markov–modulerede Poisson–proces) er denne sammenhæng næsten helt fraværende, om end dette højst sandsynligt skyldes, at de numeriske beregninger for MMPP–modellen giver tvivlsomme resultater, der er næsten helt uafhængige af valget af parameterisering. Det har ikke været muligt at afhjælpe dette.

Observationerne af systemet viser, at *for trafik der matcher antagelserne i modellerne*, kan køteoretiske modelberegninger bruges som en rimelig kvalitativ indikator for systemets opførsel. Endvidere viser eksperimenterne hvor effektivt pakketransmission fungerer i Linux-kernen, hvilket gør at køerne er mere tilbøjelige til at være tomme end forudset af modellerne. En anden ting der fremgår tydeligt tydeligt af resultaterne, er forskellen i adfærd mellem trafik med eksponentialfordelte og deterministiske behandlingstider og mellem forskellige trafikrater, hvilket til en vis grad modsvares af modellens forudsigelser, og hvilket validerer målings– og trafikgenereringsmetoden. Endelig er interaktionen mellem TCPs *control loop* og køen tydeligt i observationerne, hvilket svarer godt til hvad der forventes fra teorien om TCPs opførsel.

Afslutningsvis viser resultaterne, at *fairness queueing* i vid udstrækning kan opfattes som bestående af helt adskilte køer. Dog er køpladsen delt mellem de enkelte køer i implementeringen i Linux-kernen, hvilket gør at kølængderne fordeles anderledes end en stringent tolkning som separate køer ville indikere.

Contents

1 Introduction

Queueing delays have turned out to be a potentially significant source of latency in modern computer networks, especially when links come under load. This phenomenon has been coined *bufferbloat* [Sta12; GN11] and is an ongoing research topic in the network research community [ISOC]. The research has established that bufferbloat is a pressing issue in many scenarios, but also that mitigation measures show promising results in solving the problem, at least for wired network interfaces [Get13]. Prior work by this author includes an extensive comparison of the state of the art in Linux mitigation measures for Ethernet networks [Høi12]. More recently, the mitigation effort has gained a working group in the IETF that focuses on queue management and scheduling [Sti13].

The premier research methodology when researching new mitigation measures for bufferbloat, and in much network systems research in general, consists of doing simulations, or experiments on real-world systems. Purely theoretical models are less common, and often the validation of the theoretical models against real-world systems is skipped, which means that differences between the models and the real systems can go overlooked, for example leading to inadvertently applying models to situations where their assumptions about the network traffic characteristics do not hold. This is unfortunate, because modelling has the potential to provide predictions of system behaviour at a lower operational and computational cost than simulations and experiments. Hence, a better understanding of the relationship between models and real systems can potentially foster a wider adoption of theoretical modelling as a methodology in networking research. To gain insight into this relationship between theoretical models and real-world systems, the goal of this thesis is to model a specific real-world environment (a Linux-based gateway), and compare the model predictions to measurements of queue behaviour in the gateway.

The modelling tool employed is the mathematical topic of queueing theory, based on continuous-time Markov chains. Queueing theory is a well-established field in mathematics, and an extensive body of work on how to solve complex systems of interrelated queues exists based on it [Bol+06]. However, central to the Markov chain approach is the assumption of memoryless queues and the associated exponential distribution of traffic inter-arrival times (usually referred to as *Poisson queues* or *Poisson traffic*, due to the fact that the *number* of arrivals become Poisson-distributed when the exponential distribution is employed for inter-arrival times). While the assumption of Poisson queues might hold true for individual data streams, empirical analysis of internet traffic has indicated that the aggregate traffic is *not* exponentially distributed [Lel+94]. And indeed, it seems counter-intuitive that a complex feedback system such as TCP [RFC793] would result in independent packet arrivals at the gateway.

While real-world internet traffic may not be Poisson-distributed, it is certainly possible to generate traffic that is. Hence, a two-pronged approach is taken in this thesis: Test the simplest (and hence analytically most tractable) models against artificially generated traffic, and also employ a more sophisticated model to (attempt to) capture more realistic traffic scenarios. The latter model is based on the Markov-modulated Poisson processes (MMPP) [FM93], which is a composite model that does not assume Poisson arrivals, and which has been successfully employed to describe real-world networks before [HL86; SVA91].

The bufferbloat issue has been sought mitigated by mechanisms that do not involve changing queueing behaviour. Such approaches usually center around making transport protocols ”play nice” with other traffic, by attempting to detect induced latency on the path and back off appropriately. Such efforts include delay-based congestion control mechanisms for TCP [RFC6817], and even using machine learning to generate congestion control algorithms that perform well under certain constraints [WB13]. Further development of mitigation measures is on the agenda at a workshop at the end of September, hosted by The Internet Society [ISOC][1].

Either way, queues in intermediate gateways remain an important piece of the bufferbloat puzzle; and hence, gaining further insight into the behaviour of queues is valuable. One of the bufferbloat mitigation measures that have shown a great effect is fairness queueing [Get13; Høi12], where individual data streams are divided

[1] Incidentally, this author is one of the attendees of this workshop.

up into separate queues to minimise their influence on each other. For this reason, investigating the modelling of fairness queueing is worthwhile. While single queues are the main focus of this thesis, a digression is made into fairness queueing. The goal of this is to investigate to what extent the fairness queueing mechanism in the Linux kernel can be thought of as completely separate queues from a queueing theory point of view; the results of this investigation can then serve as a starting point for further analysis of fairness queueing mechanisms.

Summing up, the goal of this thesis is to explore how well mathematical queueing theory models describe the behaviour of a real-world Linux gateway device, by constructing queueing theory models to describe the system and comparing their predictions with measurements of the real system. Three different models of different complexity will be constructed and compared to three different types of traffic meant to correspond to each of the models' assumptions. The questions sought answered are the following:

1. How well do the models predict the steady-state queue length distribution of the queue in the Linux gateway?
2. Does the mean queue waiting time predictions of the models match the measured waiting time of the packets traversing the Linux gateway?
3. To what extent can the individual sub-queues of the fairness queueing implementation in the Linux kernel be considered to be completely independent queues?

The rest of the thesis is laid out as follows: Section 2 provides background information on the TCP transport protocol, fairness queueing and the way queueing works in the Linux kernel. Section 3 introduces the queueing theory concepts used, and describes the three constructed models. Section 4 describes the experimental setup, and section 5 presents the results. Finally, section 6 sums up and provides some ideas for further work.

2 Background

This section provides background information on various aspects of the networking concepts and technology employed in this thesis. This includes an overview of how the Transmission Control Protocol (TCP), used in the experiments as an example of real-world internet traffic, controls packet transmission; an introduction to the concept of fairness queueing and its role in mitigating bufferbloat; and finally and overview of how queueing works in the Linux kernel.

2.1 TCP packet transmission control

A large fraction of the traffic on the public internet is TCP traffic; as such this protocol is a good example of real-world traffic to be employed in the experiments here. In the following, the focus is on how the packet transmission is controlled by the default Linux TCP implementation, the CUBIC congestion control algorithm [HRX08]. For an exposition of the detailed workings of other aspects of the protocol, the reader is referred to the prior work of this author [Høi12].

TCP uses an active acknowledgement scheme where the receiver sends acknowledgement information either tacked on to packets transmitted as part of the connection, or, if the connection mainly transfers data in one direction, as separate acknowledgement packets [RFC793]. Once a TCP connection is established between two endpoints, the sender keeps track of a *congestion window*, which governs how many packets are transmitted by the sender before pausing to await acknowledgement from the receiver. Historically, the window has been initialised to between two and four packets [RFC3390], but in current Linux kernels, this value has been increased to 10 packets following a proposal from Google [RFC6928].

Throughout the connection's lifetime, the congestion window is adjusted in response to either packet transmissions or *congestion events* that indicate the transmission path is experiencing congestion somewhere along the way. These events usually come in the form of packet drops, but can also be in the form of an explicit congestion notification mechanism [RFC3168]. The exact adjustments of the congestion window is governed by the active congestion control algorithm, of which many implementations exist [RFC2001; RFC2581; RFC5681; RFC6582]. However, the general principle is that while packets are being transmitted and no congestion events occur, the sender assumes that more bandwidth is available for the transfer, and the congestion window is increased to utilise it, probing for the level where congestion occurs. When a congestion event does occur, the window is decreased substantially, and after a while the probing for extra bandwidth is resumed.

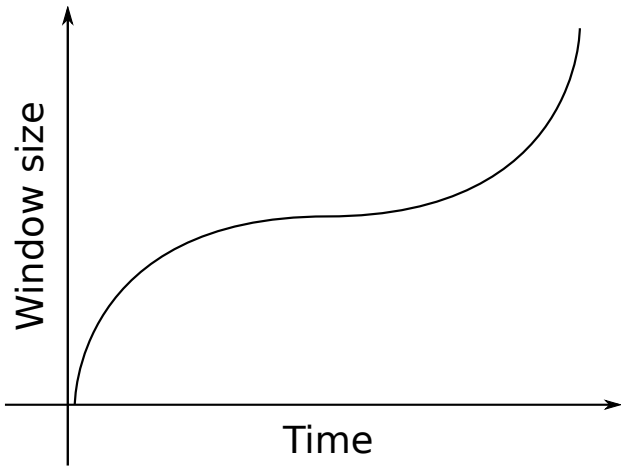


Figure 2.1: Illustration of the CUBIC growth function. The first concave part ensures rapid growth after a congestion event, the middle part ensures stability around the previous window size, and the last convex part ensures effective probing for more bandwidth. After [HRX08], reproduced from [Høi12].

The Linux kernel offers a pluggable extension mechanism to replace the TCP congestion control algorithm employed. The default is the CUBIC algorithm, which employs a cubic function of elapsed time to adjust the window (illustrated in figure 2.1). This function is scaled in such a fashion that upon a congestion event, the plateau of the curve is at the level the window was at when the last congestion event occurred. This ensures a rapid growth after the initial back-off at a congestion event, while gradually levelling off as the previous level is approached. It also ensures that in the event of more bandwidth becoming available (and

so the previous level no longer being accurate), a rapid increase in transmission speed is achieved (the last convex part of the curve), ensuring efficient utilisation of available bandwidth.

The CUBIC algorithm provides a rather aggressive increase of the congestion window, ensuring efficient utilisation of the available bandwidth [HRX08]. In the context of the Linux kernel, the implementation of the algorithm is coupled with a Hybrid Start mechanism [HR11], employed at the beginning of a connection to control the expansion of the congestion window until the first congestion event, in order to ensure a fairly rapid increase in transmission speed to utilise available bandwidth, but without overshooting the actual optimal transmission speed more than strictly necessary.

In the context of the experiments performed in this thesis, the interaction between queue length and the TCP congestion control mechanism is of interest: when the queue fills up, excess packets will simply be dropped; this is detected as a sign of congestion by the TCP control loop, causing transmission to slow down, freeing up some queue space. This in turn makes room for TCP to re-expand its window (since more packets will be absorbed by the queue). This interaction should be observable on the queue length measurements during TCP traffic flows.

2.2 Fairness queueing

Fairness queueing is a mechanism to split up packets going through a queue, separating them into individual classes and giving each class a fair share of the available bandwidth (hence the *fairness* in the name). What constitutes a class can vary, but by default in the Linux kernel, a class corresponds to a traffic flow, identified by a unique combination of source and destination network addresses, network protocol and source and destination port numbers. This type of fairness queueing is often also referred to as *flow queueing*. Since true fairness queueing (which keeps track of all traffic classes seen in the system lifetime) can be expensive to implement, a hashing-based variant is employed in the Linux kernel [McK90]. This implementation hashes the five flow identifier values into a configurable number of bins, and treats each bin as having its own queue, serving the active queues in a round-robin fashion. A variant of fairness queueing is known as deficit round-robin, where the round-robin mechanism takes into account the number of bytes dequeued by each dequeue operation and tries to ensure that each queue gets the same fair share of *bytes*, rather than just of dequeue opportunities [SV96].

Fairness queueing plays an important role in combating bufferbloat, because the isolation of flows tremendously improves latency for interactive applications in the face of congestion, especially when coupled with an active queue management algorithm of some sort [Get13; Høi12]. Such a combination is in the Linux kernel under the name of `fq_codel`, which implements a deficit round-robin scheme combined with the CoDel active queue management algorithm [NJ12].

Because of the importance of fairness queueing in bufferbloat mitigation, taking a closer look at the queueing mechanics of its implementation is warranted, so as to hopefully gain a better understanding of the mechanism. For this reason, fairness queueing is included as part of the experiments in this thesis, with the focus being on examining to what extent the individual queues behave as though they were completely isolated.

2.3 Queueing in the Linux kernel

For model predictions to be compared to a real system, it is necessary to inspect the behaviour of this system and extract the same metrics as those predicted by the model used to describe the queueing system. This means that the system that is examined needs to provide the right metrics, or be modified to do so. This makes the Linux kernel ideal, because it is both open source (and thus possible to inspect and modify), and because it already has fairly extensive statistics collection facilities. Furthermore, the Linux kernel is very widely used on today's internet, both as a server, client and intermediate gateway, and it is the system used

for prototyping and development in the bufferbloat research community. Thus, the Linux kernel represents a good example of a real-world system, and is employed as such in this thesis.

This section gives an overview of the Linux network stack, focusing on the queueing mechanisms. For a more detailed description of other parts of the network stack, see [Høi12] and [Ben06].

The Linux network stack is summarised in figure 2.2. The area of interest in this case is the Traffic Control subsystem and its interface with the device drivers, depicted on the lower-right corner of the figure. This subsystem provides a pluggable mechanism to implement various queueing disciplines in the kernel. A queueing discipline can implement a simple queue management algorithm, or a whole different queueing semantic (such as fairness queueing), and is referred to in kernel nomenclature as a *qdisc*. Each network interface has at least one qdisc attached; multiple qdiscs can be arranged in a tree structure, dequeuing up the tree, with various filtering mechanisms to enqueue traffic into different parts of the tree [Bro06; Linux].

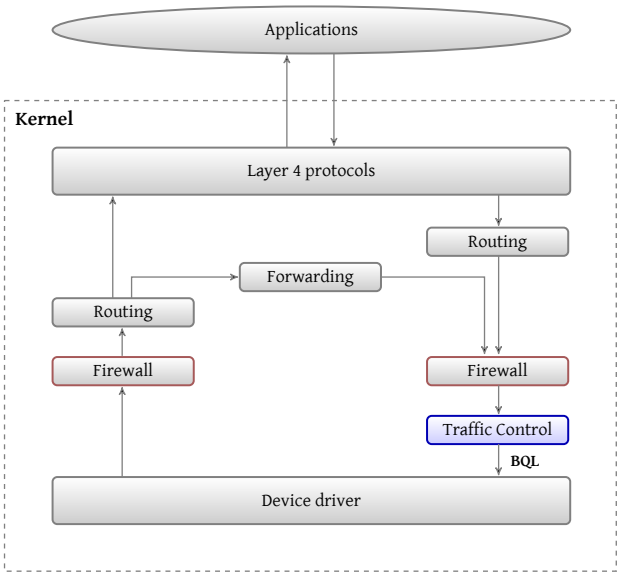


Figure 2.2: Overview of the Linux IP network stack. After [Ben06, p. 410 and p. 816], reproduced from [Høi12].

When a network packet is transmitted, it is first enqueued to the qdisc attached to the network interface the packet is going out on; unless, that is, the qdisc is empty and marked as *work conserving*, in which case an optimisation simply bypasses the qdisc and transmits the packet straight away. This enqueuing and dequeuing is controlled by the core qdisc mechanism (which also controls the qdisc tree structure if one exists), with each implemented qdisc exporting an enqueue and a dequeue function. The interface is structured so that what the qdisc does with packets internally is completely opaque to the rest of the kernel, which simply sees a queue that allows enqueuing and dequeuing of packets.

The core part of the qdisc mechanism also provides a mechanism to extract statistics from the qdiscs. It provides communication with userspace through the Netlink interface[2], for configuring the qdiscs and extracting statistics from them. Certain statistics are gathered for all qdiscs (such as total queue length measured in both bytes and packets), and furthermore a hook is available for each qdisc to export its own implementation–specific statistics data. All the statistics need to be understood by the userspace tool communicating with the kernel; the standard tool for configuring and extracting information is the `tc` utility distributed as part of the `iproute2` software package.

When packets are dequeued from the qdisc, they are sent down the stack to the network device driver. The driver can also queue up packets both in the driver itself, and in the device transmit ring. During the initial phases of the bufferbloat research effort, it became clear that this was happening to such great an extent that it made it completely infeasible for the queueing disciplines in the traffic control layer to control the queue length. To mitigate this, a feature dubbed Byte Queue Limits (BQL) was introduced into the kernel [Her11; Cor11]. This mechanism allows the upper layer to limit the number of *bytes* queued in the network driver, which makes it possible to predict and configure bounds on the transmit time of packets in the buffer. One of the reasons it was impossible before was that, especially in the presence of hardware offloads, a packet can vary orders of magnitude in size (up to 64KB with offloads turned on), making the discrepancy between the queue length seen by the qdisc and the actual queue length in the machine grow unmanageable. BQL, in combination with various other tuning parameters [TG12], makes it possible to bring control of the queue

[2]The Netlink interface is an IPC interface employed by the kernel that uses network packets as the communication medium, and provides both unicast and multicast semantics [RFC3549].

back in the traffic control layer where it belongs. This is an advantage for bringing down latency, and from the point of view of these experiments, it makes it possible to actually measure queue statistics with a reasonable accuracy.

2.4 Summary

This section has given an overview of some important aspects of the concepts and technology employed in this thesis. The TCP control loop mechanism is an integral part in understanding common network traffic behaviour; fairness queueing plays an important part in packet management at gateways and hence bufferbloat mitigation, and is also an object of this thesis' investigation; and finally, the Linux kernel queueing behaviour is an integral foundation for the experiments since Linux is the real-world system being investigated. Together, this background information forms the backdrop for the formulation of the queue models and the experiments, which are presented in the following sections.

3 Queue models

The modelling approach taken in this thesis is based on the mathematical field of queueing theory, using steady-state analysis of continuous-time Markov chains (CTMCs). Steady-state analysis is selected because it is analytically simpler while still providing measures that make it possible to ascertain the validity of the models. This section gives an overview of the general modelling approach, with the subsections giving the details of the specific models employed, and their predictions of queue behaviour. The exposition in this section follows that of [Bol+06].

A CTMC is a stochastic process that is indexed by continuous time parameters $t_i \in \mathbb{R}_0^+$ and has discrete state space S (usually $S \subseteq \mathbb{N}$). In the context of queues, the state of the Markov chain typically represents the number of packets (or, more generally, customers) in the queue; also referred to as the *queue length*. Furthermore, to constitute a CTMC, the stochastic process must possess the *Markov property* [Bol+06, eq. (2.30)]:

$$P(X_{t_{n+1}} = s_{n+1} | X_{t_n} = s_n, X_{t_{n-1}} = s_{n-1}, \ldots, X_{t_0} = s_0) = P(X_{t_{n+1}} = s_{n+1} | X_{t_n} = s_n), \quad \forall n \tag{3.1}$$

This property is also known as the *memoryless property* and simply states that the transition probabilities of the CTMC depends only on the current state, and not on previous states. It follows that the intervals between transitions must be exponentially distributed, since no other continuous-time distribution has the memoryless property. From the above definition, it is possible to express the state *transition probabilities* $p_{ij}(u, v)$ of the CTMC transitioning from state i to state j in the time interval $[u; v)$ as:

$$p_{ij}(u, v) = P(X_v = j | X_u = i) \tag{3.2}$$

or, in the case where the transition probability depends only on the length of the time interval $t = v - u$ and not the specific values of u and v, the CTMC is said to be *time-homogeneous*, and the transition probabilities $p_{ij}(t)$ are given by:

$$p_{ij}(t) = p_{ij}(0, t) = P(X_t = j | X_0 = i) \tag{3.3}$$

From the transition probabilities $p_{ij}(u, v)$, expressed in matrix form as $\mathbf{P}(u, v) = [p_{ij}(u, v)]$, and the state probability (row) vector $\boldsymbol{\pi}(u) = [\pi_i(u)]$ at time u, the unconditional state probabilities at time v can be expressed as $\boldsymbol{\pi}(v) = \boldsymbol{\pi}(u)\mathbf{P}(u, v), \forall u, v \in T, u \leq v$, which in the time-homogeneous case reduces to $\boldsymbol{\pi}(t) = \boldsymbol{\pi}(0)\mathbf{P}(t)$. In other words, $\boldsymbol{\pi}(t)$ is the vector of state probabilities as a function of the time parameter t.

The primary measure of interest in steady-state analysis of CTMCs is the *steady-state probabilities* $\boldsymbol{\pi}$ of the CTMC. They are independent of the initial probabilities and of the time, and are given as time limits $\pi_i = \lim_{t\to\infty} \pi_i(t)$. Not all CTMCs have a unique steady-state probability vector associated; those that do are called *ergodic* CTMCs. For a queueing system, ergodicity of the underlying CTMC implies stability of the system.

In order to derive the steady-state probabilities, the *instantaneous transition rates* $q_{ij}(t)$ are defined as:

$$\begin{aligned} q_{ij}(t) &= \lim_{\Delta t \to 0} \frac{p_{ij}(t, t + \Delta t)}{\Delta t}, i \neq j, \\ q_{ii}(t) &= \lim_{\Delta t \to 0} \frac{p_{ii}(t, t + \Delta t) - 1}{\Delta t}. \end{aligned} \tag{3.4}$$

In the time-homogeneous case, the transition rates q_{ij} are independent of time, and the *infinitesimal generator matrix* $\mathbf{Q} = [q_{ij}]$ can be defined. Then, the steady-state probabilities can be derived by means of a set of

equations known as the Kolmogorov forward differential equations for time-homogeneous CTMCs, which are given by:

$$\dot{\boldsymbol{\pi}}(t) = \frac{d\boldsymbol{\pi}(t)}{dt} = \boldsymbol{\pi}(t)\mathbf{Q} \tag{3.5}$$

Since the steady-state probabilities are, by definition, independent of time, i.e. $\lim_{t\to\infty} \dot{\boldsymbol{\pi}}(t) = 0$, the probabilities can be found by solving the system of linear equations resulting from the equation $\mathbf{0} = \boldsymbol{\pi}\mathbf{Q}$, imposing the normalisation condition that $\boldsymbol{\pi}\mathbf{1} = 1$.

Apart from the queue length probabilities, the performance measure of interest is usually the *waiting time* $\overline{W}$, defined as the mean time an arriving customer has to wait in queue before entering service. A related measure, the *sojourn time* $\overline{T}$, is the mean time a customer spends in the queueing system, i.e. both waiting and in service. An important theorem in queueing theory, called *Little's theorem* [Bol+06, eq. (6.9)], states that the mean number of customers in the system $\overline{K}$ is related to the sojourn time as $\overline{K} = \lambda\overline{T}$ where λ is the mean arrival rate. Similarly, the mean number of customers in the queue, $\overline{Q}$ is related to the waiting time as $\overline{W} = \lambda\overline{Q}$. Finally, the waiting time and sojourn time are related as $\overline{T} = \overline{W} + \frac{1}{\mu}$ where μ is the mean service rate.

In the following, the three models employed to describe the queueing behaviour in the Linux kernel are described. The queues are specified using Kendall's notation, which specifies a queue with customers (or, in this case, packets) arriving at times distributed by the distribution A, service times distributed by the distribution B and m service stations as an $A/B/m$ queue [Bol+06]. This notation can be further extended as $A/B/m/K$ for a finite queue with a maximum of K customers in the system (i.e. max queue length of $K-1$ plus one customer in service). Finite queues will be employed exclusively here, since the Linux kernel has finite queueing space. Many different distributions can be specified for A and B; M (for Markovian) indicates the exponential distribution, D is deterministic, G is general (any) distribution, and *MMPP* is the Markov-modulated Poisson process (introduced below).

The three different models examined are (in order of increasing analytic complexity): M/M/1/K with Poisson arrival and service times, M/D/1/K with Poisson arrivals and deterministic service rates, and MMPP/D/1/K with arrivals governed by the Markov-modulated Poisson process, which makes it possible to represent several alternating arrival rates in one model, while still being analytically tractable. Although Linux has many different queueing disciplines available, to keep the model analysis tractable, the simple FIFO queueing discipline is assumed in all models.

3.1 The M/M/1/K queue

This is the simplest queue model, and assumes that both arrivals and departures happen with exponentially distributed intervals. The queue is modelled as a *birth-death* process, meaning that the underlying CTMC only transitions to neighbouring states. This model is parameterised by the arrival rate, λ, and departure rate, μ, and the maximum queue length, K. Since the limit on the maximum queue length makes the underlying Markov chain finite, the birth-death process is ergodic without further assumptions on the relation between the arrival and departure rates.

Because of the finite queueing space, the arrival rates λ_k at queue length k are defined as:

$$\lambda_k = \begin{cases} \lambda & 0 \leq k < K \\ 0 & k \geq K \end{cases} \tag{3.6}$$

The birth-death process with birth rate λ and death rate μ implies that the infinitesimal generator matrix is of order $K+1$ and has the form:

$$\mathbf{Q} = \begin{bmatrix} -\lambda & \lambda & 0 & 0 & \cdots & 0 & 0 \\ \mu & -(\lambda+\mu) & \lambda & 0 & \cdots & 0 & 0 \\ 0 & \mu & -(\lambda+\mu) & \lambda & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \mu & -\mu \end{bmatrix} \tag{3.7}$$

The structure of $\mathbf{Q}$ makes solving the system for the queue length probabilities straight-forward. Defining $\rho = \lambda/\mu$ the queue length probabilities are given by [Bol+06, section 6.2.4]:

$$\pi_k = \begin{cases} \frac{(1-\rho)\rho^k}{1-\rho^{K+1}} & 0 \leq k \leq K, \rho \neq 1 \\ \frac{1}{K+1} & 0 \leq k \leq K, \rho = 1 \\ 0 & k > K \end{cases} \tag{3.8}$$

For the experiments, $\mu = 2500$, which is the number of 500-byte packets transferred each second at 10Mbps. The predicted values for the experiments are given in figure 3.1.

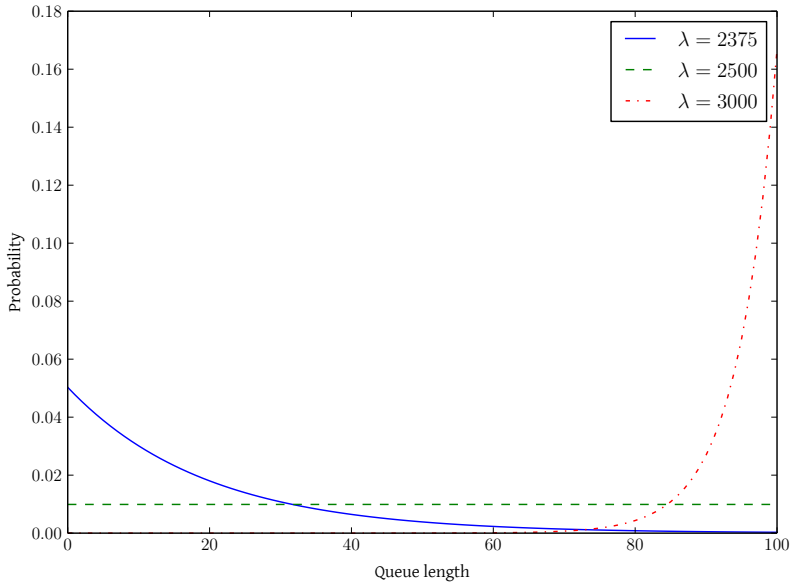


Figure 3.1: Queue length distribution predictions for the M/M/1/K model with $K = 100$ and $\mu = 2500$ for different values of λ.

The expression for the mean number of packets in the system is given by:

$$\overline{K} = \begin{cases} \frac{\rho}{1-\rho} - \frac{K+1}{1-\rho^{K+1}} \cdot \rho^{K+1} & \rho \neq 1 \\ \frac{K}{2} & \rho = 1 \end{cases} \tag{3.9}$$

From Little's theorem and equation (3.9), the sojourn time of the M/M/1/K queueing system is given by:

$$\overline{T} = \begin{cases} \frac{1}{\mu(1-\rho)} - \frac{K+1}{1-\rho^{K+1}} \cdot \frac{\rho^{K+1}}{\lambda} & \rho \neq 1 \\ \frac{K}{2\lambda} & \rho = 1 \end{cases} \tag{3.10}$$

Finally, the waiting time, $\overline{W}$ that a packet has to wait in the queue before being transmitted is given by:

$$\overline{W} = \begin{cases} \frac{1}{\mu(1-\rho)} - \frac{K+1}{1-\rho^{K+1}} \cdot \frac{\rho^{K+1}}{\lambda} - \frac{1}{\mu} & \rho \neq 1 \\ \frac{K-2}{2\lambda} & \rho = 1 \end{cases} \tag{3.11}$$

3.2 The M/D/1/K queue

This model substitutes a deterministic service time for the exponentially distributed of the previous model, and is a special case of the $M/G/1/K$ queue with general service time. A thorough analysis of this queue is given in [BG00], and is presented here adjusted to the notation used in the previous sections. As in the previous model, λ is the mean arrival rate, and μ is the service rate; except here, μ is not the mean value of a stochastic variable, but simply the actual number of packets processed per second. Then, given $\rho = \lambda/\mu$, the probability distribution of the number of customers in the system is given by [BG00, theorem 1]:

$$\begin{aligned} \pi_0 &= \frac{1}{1+\rho b_{K-1}}, \\ \pi_K &= 1 - \frac{b_{K-1}}{1+\rho b_{K-1}}, \\ \pi_k &= \frac{b_k - b_{k-1}}{1+\rho b_{K-1}}, \qquad k = 1, \ldots, K-1 \end{aligned} \tag{3.12}$$

where the coefficients b_n are given by $b_0 = 1$ and

$$b_n = \sum_{k=0}^{n} \frac{(-1)^k}{k!}(n-k)^k e^{(n-k)\rho}\rho^k. \tag{3.13}$$

The mean waiting time of a packet entering the queue is given by [BG00, theorem 3]; however, in the derivation in the article, the application of Little's theorem includes $1 - \pi_K$ as an additional term. For consistency with the other models, which instead define $\lambda_k = 0$ for $k \geq K$, this is omitted here. Instead, a similar calculation with the regular application of Little's law gives the waiting time as:

$$\overline{W} = \frac{K}{\lambda} - \frac{1}{\mu} - \frac{\sum_{k=0}^{K-1} b_k}{\lambda(1+\rho b_{K-1})} \tag{3.14}$$

3.2.1 Computing the predicted probabilities

While the expression given in equation (3.13) makes it possible to compute the queue length probabilities, a more efficient algorithm for the computation of the probabilities is outlined in section 4 of [BG00].

The algorithm computes the coefficients b_n from another set of coefficients a_n, derived as part of the derivation of the probability distribution. These coefficients are in turn derived from the probabilities, α_k of k packets arriving during the time it takes to service one packet. These probabilities are given by

$$\alpha_k = \frac{\rho^k}{k!}e^{-\rho} \tag{3.15}$$

and the algorithm computes them iteratively, as $\alpha_k = \frac{\rho}{k}\alpha_{k-1}, k \geq 1$.

Given the probabilities α_k, the coefficients a_n can be straightforwardly computed as

$$a_n = e^{\rho}\left(a_{n-1} - \sum_{i=1}^{n-1} \alpha_i a_{n-i} - \alpha_{n-1} a_0\right) \tag{3.16}$$

and finally, the coefficients b_n are given as the cumulative sum of a_n, i.e. $b_0 = a_0$ and for $k \geq 1, b_k = \sum_{i=0}^{k} a_i$ [BG00, section 4].

A Python implementation of this algorithm is given in the appendix; see section 8.5. Running it produces the predicted probability distributions shown in figure 3.2.

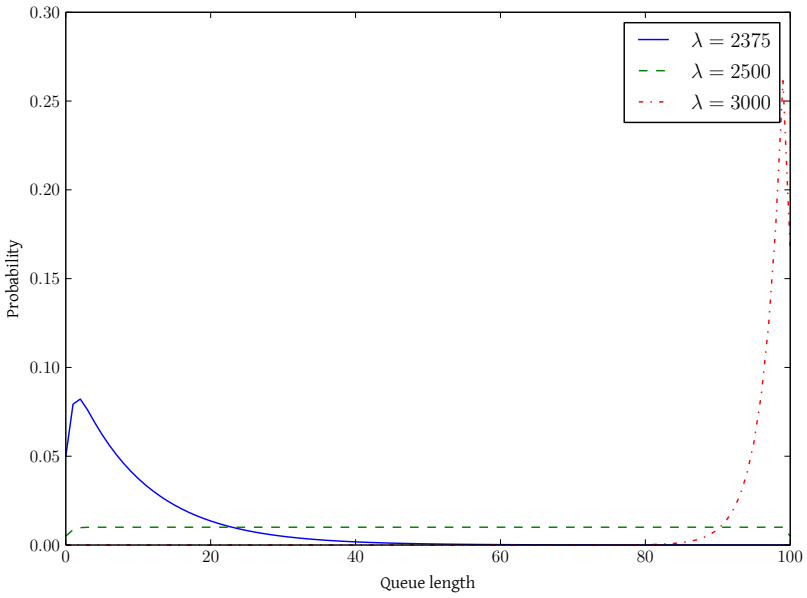


Figure 3.2: Queue length distribution predictions for the M/D/1/K model with $K = 100$ and $\mu = 2500$ for different values of λ.

3.3 The MMPP/D/1/K queue

The Markov-modulated Poisson process (MMPP) is an arrival process that has more than one arrival rate; the switch between the different arrival rates is governed by an additional underlying CTMC [FM93]. This makes it possible to model, for example, a bursty network stream, or other more complicated traffic phenomena, while still keeping the model analytically tractable. The inter-arrival times of the composite arrival process of the MMPP queue are not exponentially distributed, and the model has been successfully used to model various network arrival processes [HL86; SVA91]. This makes the MMPP model a good choice for modelling more complicated real-world networking phenomena, which is the reason it is employed here. The MMPP process is a special case of the *N*-process, which is treated extensively as the N/G/1/K queue in [Blo89], with the MMPP as a special case.

The MMPP queue works by having several arrival rates $(\lambda_1, \ldots, \lambda_m)$. A second embedded CTMC with m states governs the active arrival rate, in that when the CTMC is in state i, the system is said to be in *phase* i, and packets arrive with rate λ_i. In the following, a two-state MMPP (i.e. $m = 2$) with deterministic service time and finite queueing space is used. Similar two-state MMPP models are employed in [HL86] and [SVA91].

The infinitesimal generator matrix of the CTMC that governs the arrival rate is called $\mathbf{Q}^*$. The model is parameterised by the two arrival rates λ_1, λ_2 with $\mathbf{\Lambda} = \text{diag}(\lambda_1, \lambda_2)$, as well as the mean transition rates, r_1, r_2,

of the CTMC generated by $\mathbf{Q}^*$. In this nomenclature, the infinitesimal generator matrix is given by [HL86][3]:

$$\mathbf{Q}^* = \begin{bmatrix} -r_1 & r_1 \\ r_2 & -r_2 \end{bmatrix} \tag{3.17}$$

In [Blo89], the matrix is given by $\mathbf{Q}^* = \mathbf{T} + \mathbf{T}^\circ\mathbf{A}^\circ$ all of which are $m \times m$ matrices. The columns of $\mathbf{T}^\circ$ are given by the vector $\overline{\mathbf{T}}^\circ$ and $\mathbf{A}^\circ = \text{diag}(\alpha_1, \ldots, \alpha_m)$. This type of process is called a *phase-type renewal process* or *N*-process, and is a more general form of the MMPP. This alternative parameterisation is mentioned here because some of the calculations given by [Blo89] uses this parameterisation; the *N*-process will not be elaborated further on in this section. In terms of the parameterisation by the sojourn times r_1, r_2, the corresponding values are [Blo89, pgs 281–282]:

$$\mathbf{T} = \begin{bmatrix} -r_1 & r_1 \\ 0 & -r_2 \end{bmatrix}, \overline{\mathbf{T}}^\circ = \begin{bmatrix} 0 \\ r_2 \end{bmatrix}, \alpha_1 = 1, \alpha_2 = 0. \tag{3.18}$$

The composite two-state MMPP queueing system with K places, when viewed at departure epochs $\tau_0, \tau_1, \ldots$, forms a discrete semi-Markov chain with finite state space $\{0, 1, \ldots, K\} \times \{1, 2\}$. The transition matrix of this queueing system[4] is given by [Blo89, p. 283]:

$$\tilde{\mathbf{Q}} = \begin{bmatrix} \mathbf{B}_0 & \mathbf{B}_1 & \mathbf{B}_2 & \cdots & \mathbf{B}_{K-2} & \sum_{k=K-1}^{\infty} \mathbf{B}_k \\ \mathbf{A}_0 & \mathbf{A}_1 & \mathbf{A}_2 & \cdots & \mathbf{A}_{K-2} & \sum_{k=K-1}^{\infty} \mathbf{A}_k \\ \mathbf{0} & \mathbf{A}_0 & \mathbf{A}_1 & \cdots & \mathbf{A}_{K-3} & \sum_{k=K-2}^{\infty} \mathbf{A}_k \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{A}_0 & \sum_{k=1}^{\infty} \mathbf{A}_k \end{bmatrix} \tag{3.19}$$

where each $\mathbf{A}_n$ is a 2×2 matrix, with the $(i,j)^{th}$ element given as the conditional probability that at the end of a service period, there have been n arrivals, and the phase of the MMPP is j, given that at the beginning of the period it had phase i. Furthermore, $\mathbf{B}_n = \mathbf{U}\mathbf{A}_n$, where $\mathbf{U} = (\mathbf{\Lambda} - \mathbf{Q}^*)^{-1}\mathbf{\Lambda}$ [HL86][5]. The role of $\mathbf{U}$ is to keep track of the phase of the MMPP while the queue is empty.

The probabilities $\mathbf{A}_n$ are not easily discernible, but the appendix of [Blo89] gives an algorithm to compute them iteratively from the model parametrisation. From this computation, the eigenvector $\boldsymbol{\pi}$ (with associated eigenvalue 1) of the matrix $\tilde{\mathbf{Q}}$ can be computed. This eigenvector, under the normalisation condition $\boldsymbol{\pi}\mathbf{1} = 1$, consists of the stationary queue length probabilities $\pi_0, \ldots, \pi_K$, as observed at the departure epochs. Each element π_i is a two-element row vector containing the stationary probabilities of the system being in each of the two system phases while having a queue length of i. This means that the total probability for queue length i is given by $\pi_i\mathbf{1}$, i.e. the sum of the two elements; in the sequel, π_i is used to refer to this sum. While a lengthy expression for queue length probabilities at arbitrary times is given in [Blo89], for the purposes of comparing the model predictions to the experimental data, the values of $\boldsymbol{\pi}$ are deemed to be sufficiently close to the queue length probabilities for arbitrary times to be used without further calculations.

Given $\boldsymbol{\pi}$, and defining $T = 1/\mu$, the waiting time can be calculated by [SVA91, eq. (49)]:

$$\overline{W} = \sum_{k=1}^{K-1} \left((k-1)T - \frac{T}{2} \right) \cdot \pi_k \tag{3.20}$$

[3] The matrix $\mathbf{Q}^*$ is called R in [HL86], but the notation from [Blo89] is preferred here.

[4] The matrix $\tilde{\mathbf{Q}}$ is of order K rather than $K+1$, because it only considers the queue length probabilities, not the probabilities for the total number of packets in the system. This is different from the other models; the difference is taken into account by the numerical calculations, by modifying the value of K appropriately.

[5] The value for $\mathbf{U}$ is given as $\mathbf{U} = \mathbf{\Lambda}(\mathbf{\Lambda} - \mathbf{Q}^*)^{-1}$ in [Blo89], citing [HL86]. However, [HL86] gives $\mathbf{U}$ as $\mathbf{U} = (\mathbf{\Lambda} - \mathbf{Q}^*)^{-1}\mathbf{\Lambda}$. The latter value is what is used here, since for that value, the matrix $\tilde{\mathbf{Q}}$ is stochastic in the numeric calculations, whereas for the former it is not.

3.3.1 Calculating the queue length probabilities

A Python implementation of the algorithm given in [Blo89] to calculate the probabilities given above is included in the appendix (see section 8.6). The queue length predictions resulting from the calculations are shown in figure 3.3.

The iterative computation of some of the intermediate values used in the computations converges fairly quickly to values too small to be represented by the floating-point datatypes used. This is especially the case if large numbers are involved; for this reason, a millisecond time scale is used rather than the whole-second time scale used for the other calculations. The eigenvector of the $\tilde{\mathbf{Q}}$ matrix is found using the eigenvector calculation routine of the *Numpy* library, which is in turn based on the LAPACK library [Uni+13]. It computes several eigenvectors, of which the vector corresponding to the eigenvalue closest to 1 is used (since the relations above imply an eigenvalue of 1 as one of the constraints). This eigenvector is further normalised to sum to 1, thereby satisfying the other constraint given above. Because the LAPACK routine computes the right eigenvector, the transpose of $\tilde{\mathbf{Q}}$ is passed to it to obtain the left eigenvector.

The parameters for the calculations are chosen after the following rationale: The model is meant to model TCP traffic, which is characterised by increasing its transmission speed until a packet drop is detected, at which point the transmission speed is lowered, and then slowly increased once again. This transmission speed adjustment is used analogously to the MMPP parameters, meaning that the two values of λ are set to the expected high and low points of this adjustment mechanism, and the values of r should be set to the approximate reaction time of the adjustment. The rate of change between the two states is governed by the TCP control mechanisms reaction time, which is on the order of the round-trip time. With this in mind, the values of r are chosen to be $(8.0, 10.0)$, corresponding (since the r values are rates) to a round-trip time of 100 ms (for packet loss), and a slightly higher reaction time to speed back up. The values of λ are chosen to be slightly lower and slightly higher than the service time respectively, with several choices depicted in figure 3.3. Finally, the value of μ is fixed to the line speed value of $\mu = 2.5$.

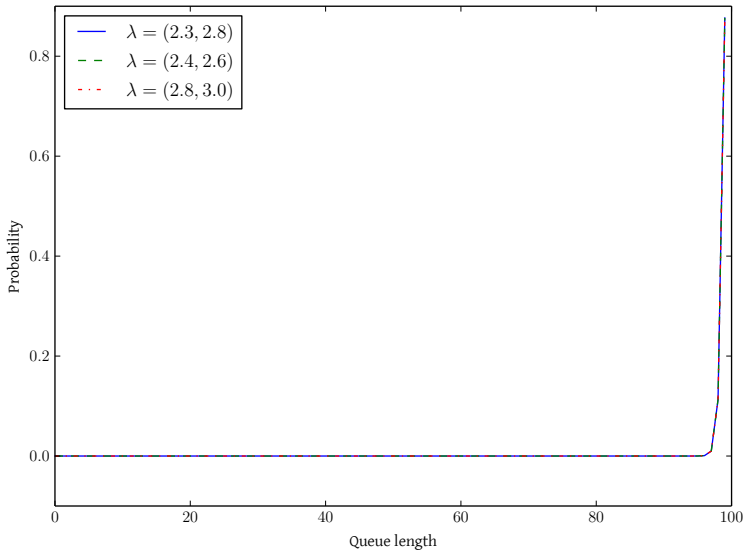


Figure 3.3: Queue length distribution predictions for the MMPP/D/1/K model with $K = 100$, $\mu = 2.5$ and $r = (8.0, 10.0)$ for different values of λ . Millisecond time scales are used to avoid calculated values diverging below the floating-point precision. The computed values are almost entirely independent of the chosen values of λ.

The computed queue length probabilities are quite insensitive to the selected parameters, as can be seen in figure 3.3. This is most likely due to an error in the calculation routines. However, every attempt has been made to have the calculations correspond to those of the cited articles, but since no explicit information on

the numerical calculations are provided by either of the articles, verifying the correctness of the implementation has not been possible, even though results of the cited articles indicate that better prediction values should be attainable.

Whatever the cause, because of insensitivity to the selected parameters, it is expected that the model predictions will match the observations poorly. However, for completeness they are still included in the results analysis in section 5.

3.4 Summary

This section has introduced the queueing theory used to construct the models, as well as the three models and their predictions of queue behaviour. These models are the M/M/1/K model which is the simplest model with the strongest assumptions on the traffic, namely exponentially distributed arrival and service times. The M/D/1/K model relaxes this assumption to deterministic service times at the cost of slightly more analytical complexity. Finally, the MMPP/D/1/K model is the most complex of the models, designed to represent composite traffic behaviour that does not assume that the traffic be exponentially distributed. Unfortunately, the numerical calculations of the predictions of the latter model have turned out to be dubious, making it unlikely that they will match the experimental observations; however, they are still included in the experiments for completeness.

4 Experimental setup

This section describes the experiments being run and their purpose, as well as the testing tools used to generate the traffic and collect the measurements. The experimental setup used for testing the models is shown in figure 4.1. The queue is measured at the gateway machine, with packets being transmitted from the test client towards the test server. The latency inducer device (a Linux-based router using the `netem` qdisc) adds latency between the gateway and test server (50ms in each direction), in order to make the reaction time of the TCP flows better match that seen in a typical internet context. The links going to the latency inducer are capped at a speed of 10 Mbps, from which the service rate of 2500 packets per second (of each 500 bytes) is derived. The gateway runs a patched Linux kernel (see section 4.2.1 below), while the client and server run standard Linux kernels [Linux].

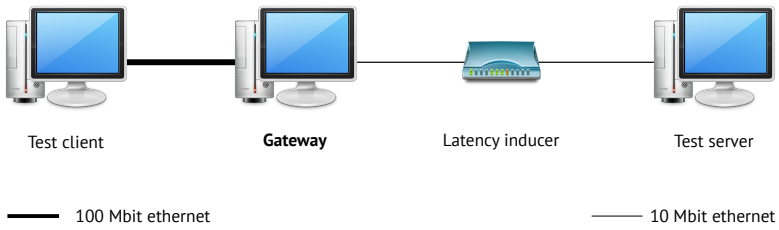


Figure 4.1: Test setup. Packets are sent from the set client towards the test server, and the queue is measured on the gateway. The latency inducer (a Linux-based router using the `netem` qdisc) adds 50ms latency in each direction, to make the behaviour of the TCP flows correspond better to real-world conditions.

4.1 Experiments conducted

The following three experiments are run to test the different types of model predictions: a steady-state queue length measurement, a waiting time measurement, and a fairness queueing measurement. Each is described in detail below.

Common for all tests is that the gateway is configured according to the bufferbloat community best practices [TG12]. The `codel` and `fq_codel` qdiscs are configured on the network interface where traffic is measured, providing single queue and fairness queueing semantics respectively. These qdiscs both implement the CoDel AQM algorithm on their queues; however, because the aim of the tests is to have the actual setup be as close to the model assumptions as possible, the AQM mechanism is disabled. The CoDel qdisc family is preferred over the default FIFO queue, because they timestamp packets on arrival, making it possible to directly measure the waiting time; and disabling the actual dropping of packets makes it possible to use this functionality while still functioning as a FIFO queue as assumed in the models.

4.1.1 Steady-state queue length measurements

The purpose of the steady-state queue length tests is to compare the model predictions of queue length probabilities with the actual observed queue lengths. Each test is run for a total of 300 seconds, with the statistics readout interval set to 50 ms (i.e. 20 readings per second). The first and last 100 measurements are discarded, to give the queue time to reach its steady state. A (normalised probability) histogram of the observed queue length measurements is compared to the predicted probabilities. Also, a χ^2 test between expected and observed values is performed; meaning that the queue length observations are viewed as a series of multinomial trials, with the queue length measurements being the observed values.

This experiment is repeated for the following types of traffic, corresponding to the different models:

1. Traffic generated to have Poisson arrival and service times (by the testing tool described below in section 4.2.3) with three different arrival rates, respectively slightly below, equal to, and slightly above the departure rate (which is given by the outgoing link speed). This is intended to correspond to the M/M/1/K model predictions.
2. As above, but with deterministic service times (i.e. fixed-size packets), intended to correspond to the M/D/1/K model predictions.
3. TCP traffic generated by the `netperf` network benchmarking tool [Jon13]. Three separate tests are run, with one, five and 30 concurrent flows respectively. This is meant to approximate real-world traffic, and correspond to the MMPP/D/1/K model predictions.

It is expected that the queue length measurements of each type of traffic will, to a certain extent, correspond to the value predicted by their respective models, although how exact the correspondence turns out is hard to predict. The exception is the MMPP model which, as noted previously, shows predicted values that do not seem to be impacted by the choice of parameterisation; for this reason, it is expected that the model predictions will match the observations poorly, although other interesting features may emerge from the experiments.

4.1.2 Waiting time measurements

The waiting time experiment aims to compare the observed waiting time with that of the model predictions. This is done by running a series of shorter tests, calculating the mean waiting time observed during this test. The standard deviation of the measured values is computed (under the assumption that the measured averages are normal distributed), and compared to the model predictions.

The traffic is generated as both Poisson and deterministic traffic by the traffic generation tool, set to a traffic rate of 3000 packets per second; this rate is selected to measure traffic that builds up a significant queue. The measurements run at queue lengths of 20, 50 and 100 packets. Each measurement is run for 6 seconds, and 50 measurements are done at each of the queue lengths.

It is expected that the observed waiting times will correspond to the model predictions about as well or better than the queue length measurements (since these are related by Little's law). In particular, the linear relation between queue length and waiting time is expected to hold for both predictions and observations, although the exact relations may vary somewhat.

4.1.3 Fairness queueing measurements

The purpose of this measurement is to investigate to what extent fairness queueing can be thought of as being comprised of identical independent queues. As the main focus of this thesis is on the single queues, only two measurements are done for fairness queueing. These are both done using Poisson traffic, with transmission rates equal to and slightly higher than the service rate. Additionally, the traffic generation tool is configured to distribute the packets evenly between 10 flows, which is achieved by randomly varying the destination port for each packet transmitted (the port number is used as part of the value that is hashed into flow buckets in the `fq_codel` implementation), between ten different values. Each active flow queue is measured over the course of the test, and the result it compared to the results for a single queue, and for model predictions with the service rate set to 1/10th of those of the single-queue models.

It is expected that the fairness queueing implementation will correspond quite well to actually having separate queues for each traffic flow. However, one feature is a possible source of discrepancy: in the `fq_codel` qdisc, the queue space is shared between the active flows, which means that the individual queues do not have a hard limit on their length. This means that the observed queue lengths are likely to be distributed

around the maximum, rather than strictly cut off as the model might predict, making exact correspondence between model and observation less likely.

4.2 Testing tools

As part of carrying out the experiments and gathering the required data, several tools have been developed. These include a patch that modifies the Linux kernel statistics gathering mechanism, a tool to gather the statistics emitted by the patched kernel, and a traffic generation tool to generate traffic with the right properties. These tools are presented in turn in the following subsections.

4.2.1 Linux kernel modifications

As outlined in section 2.3, the Linux kernel is already equipped with an extensive mechanism for gathering traffic statistics, which could conceivably be utilised without further modification for most of the required tests. However, for the experiments performed, a patch to the kernel has been introduced which modifies the statistics gathering behaviour somewhat. In the following, the motivation for developing this patch is presented, followed by a more detailed description of how the patch accomplishes the stated goals.

One main motivation behind the patch is to augment the statistics measurement mechanism to incorporate the values needed for the experiments performed. To this end, the patch augments the statistics gathered and exported by the `fq_codel` qdisc, to also include queue length and wait time statistics for each sub-queue of the fairness queueing mechanism, thus making it possible to compare each queue to the model predictions. Furthermore, the patch adds a configuration parameter to the sysfs configuration mechanism that makes it possible to disable the qdisc bypass capabilities. This bypass mechanism is an optimisation that skips enqueueing to the qdisc completely when it is empty and a packet is ready to be transmitted. While this is more efficient, it skews the queue length measurements, making it less likely that they conform to the model predictions. Thus, turning it off makes it possible to verify the model predictions under circumstances that conforms better to the model assumptions.

Apart from these concrete modifications to the kernel behaviour, the main feature of the patch is to change the statistics information to be broadcast at configurable intervals from the kernel (using the netlink IPC multicast semantics). In mainline Linux, statistics information is emitted when requested from a userspace application. In order to respond to such a request for statistics data, the kernel has to lock the data structure containing the data; this can be avoided when using a broadcast scheme, because the data can be broadcast while the lock is already held at packet enqueue or dequeue.

When frequent polling is required, as is the case for the performed experiments, avoiding extra locking can be advantageous. However, this advantage might be overshadowed by the fact that additional checks have to be added to every packet enqueue and dequeue, which in most cases are significantly more frequent than the statistics observation interval. The reason the broadcast approach has been pursued regardless of the questionable efficiency gains is to validate the idea of statistics broadcasting as a more general mechanism that might be useful in a wider context within the kernel. A possible use for this could be to enable applications to subscribe to network statistics either to present to the user, or to perform online configuration and tuning of parameters in reaction to events. Such events might include, apart from the statistics broadcast in the current incarnation of the patch, such things as packets drops or notifications of congestion events.

The patch has been posted to the *bloat-devel* mailing list for feedback from the bufferbloat community [Høi13], and revisions have been incorporated based on this feedback. Every attempt has been made to make the implementation in the patch as efficient as possible, by rate limiting the statistics broadcast and only sending out data if a userspace application is listening. Furthermore, the intrusiveness of the patch has been limited by making the whole mechanism optional on kernel compilation. Even so, private feedback garnered from the posting to the bufferbloat community has indicated that having the patch accepted into the mainline Linux

kernel is unrealistic due to the operation mode of allocating a new packet buffer for netlink communication in the packet fast path, and due to the specific diagnostic focus the patch currently has [Hem13]. However, an interest in the general approach has also been indicated [Tah13], but fleshing out a version that might be more widely applicable remains to be done.

The rest of this section describes the working of the patch in more detail. The source code is included in the appendix (see section 8.1).

The broadcast of qdisc statistics is done each time a packet is enqueued or dequeued. A new routine is added (named simply `qdisc_broadcast_stats()`; see lines 187–258 in the code listing in section 8.1), which is called each time a packet has been either enqueued or dequeued. Each of the calling functions already locks the qdisc to do their work, so no further locking is needed in `qdisc_broadcast_stats()`.

The patch specifies a new netlink multicast group, `RTNLGRP_TC_STATS` for statistics broadcast; the broadcast mechanism first checks if any userspace applications are subscribed to this group before sending out the statistics information, to avoid needlessly allocating the netlink message. Next, it checks whether enough time has passed since the last statistics transmission. This interval is configurable from userspace via the sysfs configuration file system, and defaults to 200 milliseconds. This rate limits the statistics broadcast, and effectively makes it possible to set a sampling rate for the statistics data.

Following these checks, a netlink packet buffer is allocated, and filled with the statistics information, before being broadcast to the netlink multicast group. The statistics information is filled in using the existing mechanism in the kernel, to avoid duplicating code. A slight modification to the existing mechanism is needed for it to work when called with the qdisc lock already held; this is done in a manner that keeps compatibility with existing code.

4.2.2 Queue statistics reader

To parse the statistics broadcast from the patched kernel, a userspace tool has been developed that listens for the netlink packets broadcast from the kernel and parses the queue statistics contained in them. Each reading is timestamped on reception, before being written to a data file for post-processing. The source code of the tool is included in the appendix, see section 8.2.

The developed tool is a small program written in C that uses the `libnl` library to communicate with the kernel via the netlink protocol. The program is structured with a main part that reads in the supplied command line options and sets up the netlink listening socket to receive the packets that are broadcast by the kernel, containing the statistics information. The core part of the client application parses the netlink message attributes containing those statistics that are common for all the kernel qdiscs. Since each qdisc in the kernel may supply additional measures, the client program is modularised with separate handlers for the different qdiscs, which parse the structures emitted by their respective qdiscs. Handlers exist for the `codel` and `fq_codel` qdiscs.

The output format is similarly modularised, with the output format being selectable on the command line when running the application. There's a simple text output format for human consumption as well as output suitable for machine reading and post-processing of the data, in the form of the common structured data formats CSV and JSON.

4.2.3 Traffic generator

To generate the Poisson-distributed traffic used for testing the models, a traffic generation tool has been developed. The tool generates traffic (in the form of UDP packets) at a specified rate and with a specified destination. The traffic rate is specified in either packets per second or bit rate, and the interval between subsequent packets is calculated from the user input. The application busy waits (or calls into the system

`usleep()` if the interval is above a threshold of 10 milliseconds) between each subsequent packet transmission.

To simulate Poisson-distributed traffic, the interval can optionally be varied to conform to the exponential distribution. In this case, each time a packet is sent out, the wait time to the next packet is calculated as $-log(rand())/pps$ where pps is the mean sending rate measured in packets per second, and $rand()$ generates a uniformly random number r such that $0 < r \leq 1$.

Poisson service time can also be simulated, by varying the packet size by the same exponential distribution function. This is limited by the bounds allowed by the hardware, the minimum allowed packet size being 62 bytes (Ethernet + IPv6 header + UDP header overhead) and maximum size 1514 bytes (max Ethernet frame size). For all test runs the (mean) packet size is set to 500 bytes. The source code of the traffic generation tool is included in the appendix, see section 8.3.

4.3 Summary

Three different experiments are conducted to compare the model predictions with the observed behaviour: a steady-state queue length measurement with three different types of traffic (Poisson, deterministic and TCP), corresponding to the three queue models described in section 3. Furthermore a waiting time measurement and a fairness queueing measurement are performed. To conduct the experiments, a kernel patch is introduced that changes the statistics measurement to broadcast statistics at a configurable interval, adds additional statistics for fairness queueing and disables qdisc bypass. Along with the patch is an accompanying userspace tool used to gather the broadcast statistics, and a traffic generation tool used to generate the required types of traffic at a configurable rate. Together, these tools form the basis for the experiments, the results of which are presented in the next section.

5 Results

This section presents the measurement results for the conducted experiments (described in section 4.1). The experiments are treated in each of the following subsections, before being summed up at the very end. The data analysis tools used to produce the result graphs and statistical tests are included in the appendix (see section 8.7).

5.1 Steady-state queue length measurements

The queue length measurements for the traffic generated by the traffic generator are shown in figure 5.1. For each measurement, a (normalised) histogram of the observed queue lengths is plotted along with the predicted values (and their 95th percentile interval) as computed for the M/M/1/K and M/D/1/K models. For each measurement, a χ^2 test is done that compares the observed values with the predicted ones. As is apparent from figure 5.1, all of these tests show a test probability of zero, indicating that if the experiments are viewed as a set of multinomial trials against the predicted values, the models are not a statistically significant predictor of real-world performance. However, such a χ^2 test is a rather strict condition to impose on the measurement results; and a qualitative analysis of the results provides interesting insights, as outlined in the following.

The first interesting feature of the results is apparent on figures 5.1a to 5.1c: The spike near the left-hand side of the graph shows that the queue is significantly more likely to be empty than the model results predict[6]. One likely reason for this is fact that the Linux kernel is designed to transmit packets with as little overhead as possible, optimising for the case when the queue is empty. One optimisation (the qdisc bypass mechanism) is disabled by the kernel patch applied for the measurements; however, other interactions between network stack layers can result in a similar effect. One such interaction is buffering in lower layers of the stack (in the network device driver or hardware); while the BQL mechanism limits this, it does not eliminate it completely. Such buffering will conflate measurements of several queue lengths into the empty queue measurement. If these measurements were instead spread out over the lower several queue lengths, the observed results would follow the predicted values closer than is the case on the graphs.

Another interesting feature of the results as seen in figure 5.1, is that the queue length measurements for the deterministic service time traffic show that the queue is more likely to be full than the model predicts, for the two lowest arrival rates. The effect is most marked in figure 5.1d, when comparing to both the model predictions and the corresponding results for the traffic with exponential service time in figure 5.1c. This suggests that either the traffic generation tool generates traffic at a slightly higher rate than configured, or the actual transmission rate is slightly lower than the nominal 10Mbps the network interface is configured to transmit at. The effect is less marked in figure 5.1f, which might be explained by the fact that in this case the queue is mostly full anyway, and so an ”extra” arrival will just get dropped, rather than accumulate as extra queue length.

Another interesting feature of the generated traffic measurements is the fact that the graphs for the high arrival rates (figures 5.1e and 5.1f) correspond closer to their respective model predictions than to that of the opposite model. This difference is not discernible at the other rates, but is an encouraging sign that the difference in the models actually does mirror a difference in real-world behaviour.

The final thing to note about the results in figure 5.1 is that the differences between each of the measured traffic rates correspond fairly well to what might be expected. That is, figures 5.1e and 5.1f show longer queue lengths when compared to the other graphs, which is to be expected when the traffic rate is higher. This is

[6] The reason why the measurement spike is at a queue length of one rather than zero is the following: The measurement mechanism is called after each packet enqueue and dequeue, with a timer setting a lower bound on the interval between measurements. This timer is more likely to expire between two consecutive enqueues than between an enqueue and the dequeue following immediately after, leading to a measurement spike at a queue length of 1 rather than 0 (when the queue is mostly empty).

a validation of both the traffic generation tool and the measurement methodology in the kernel: it seems that the traffic generated is a good match for the configured rate, and the measured queue lengths appear to actually be an accurate representation of the kernel queue at the traffic control level. However, that the values differ somewhat from the model predictions, emphasises the fact that the packets queued in lower layers of the network stack (as well as potentially other sources of overhead) are a source of error between the queues that can be measured, and the actual number of packets queued in the system.

The measurements for the TCP traffic are shown in figure 5.2. Inspecting the graphs reveals little correlation between the histogram shapes and the graph of the predicted values. This is to be expected, given the model's prediction values independence of choice of parameters. The correspondence with the model predictions is also non-existent from a statistical significance standpoint. However, a couple of interesting properties of the TCP data streams are visible on the graphs.

In figure 5.2a, the back-off and subsequent speed-up of the transmission rate is apparent. This is seen by the wide range of measured queue lengths, where a full queue causes packet drops, which results in a significant lowering of the transmission rate, causing the queue length to decrease. The spike between 60 and 70 packets in the queue is the level at which the TCP transmission rate converges after several cycles of transmission rate adjustments. When several concurrent flows compete for the bandwidth, each of them will alternate between backing off and increasing the transmission rate. As can be seen in figures 5.2b and 5.2c, the aggregate behaviour of this is that the queue is being kept closer to full the more concurrent flows are in effect.

Overall, the queue length results show several interesting features of the observed traffic, even though it does not pass the stringent statistical correlation test; and the method for gathering the statistical data is validated by the observations of the different traffic rates and their correlation with queue lengths. For TCP traffic, the model predictions are way off (as expected), but the TCP control loop interaction with the queue is of some interest.

5.2 Waiting time measurements

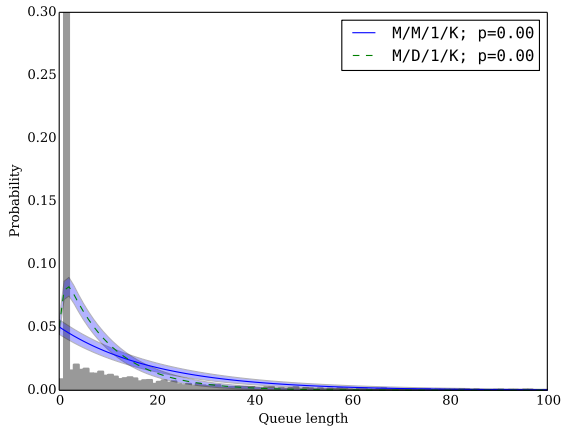
The results of the waiting time measurements are shown in figure 5.3. Waiting times are measured using the mechanism of the `codel` qdisc which timestamps packets on enqueue and measures the waiting time on dequeue. Measurements are done of traffic generated by the traffic generator with both exponential and deterministic service times. Comparing these measurements to the predicted values of the models, the observed values are somewhat higher than the predictions.

However, as expected, both the model predictions and the observations show a clear linear relation between the queue length and the waiting time. While the exact relation is somewhat different, this does validate one of the important queueing theory assumptions (Little's law). Furthermore, it helps underscore the importance of keeping queues in check to avoid unnecessary latency in real networks. Also as expected, the MMPP model predictions are way off compared to both the observed values and the other models' predictions.

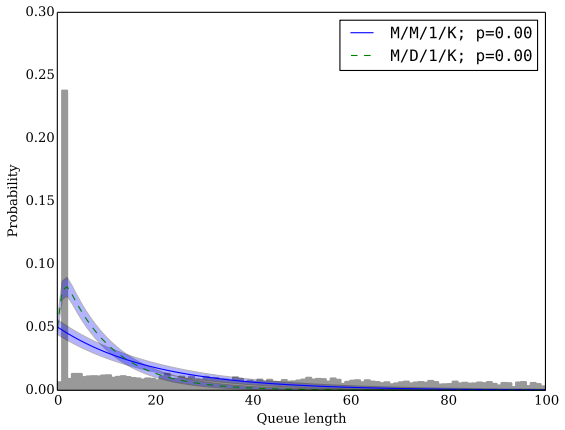
5.3 Fairness queueing measurements

The measurements of the queue lengths for fairness queueing are shown in figure 5.4. As is apparent from the figures, the queue is fairly evenly spread out between the active queues. Modelling the individual queues as independent with λ and μ values each $1/10$ of the total traffic rates (for the ten active queues), it is apparent that the model prediction performance is comparable to the single queue case.

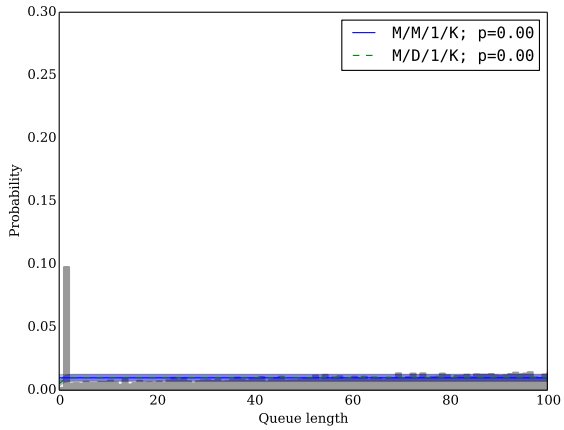
However, as expected, the shared queue means that the results do not match the single queue case completely. The global limit is set to 1000 packets, giving each queue an average limit of 100, but because of this shared nature, the maximum queue lengths are distributed around the maximum rather than up to it. Looking at the graph of the predicted values, it is conceivable that the observed values would follow the predicted values better in the presence of hard queue length limits.



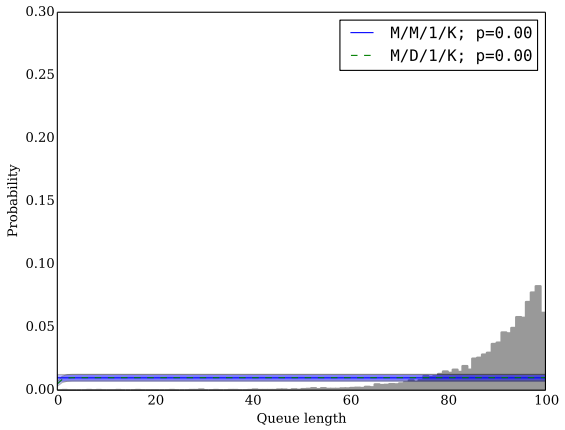
(a) Exponential service time, $\lambda = 2375$ pps (N=5559).



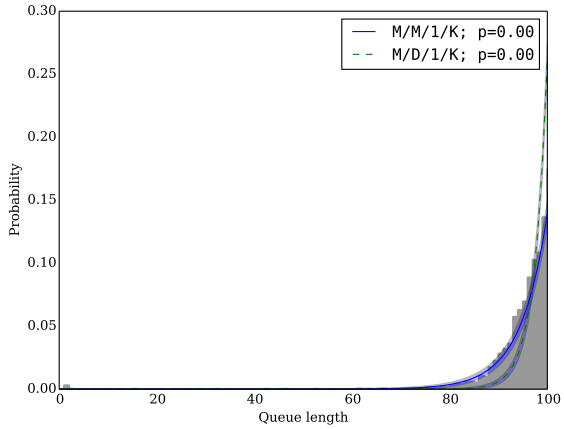
(b) Deterministic service time, $\lambda = 2375$ pps (N=5576).



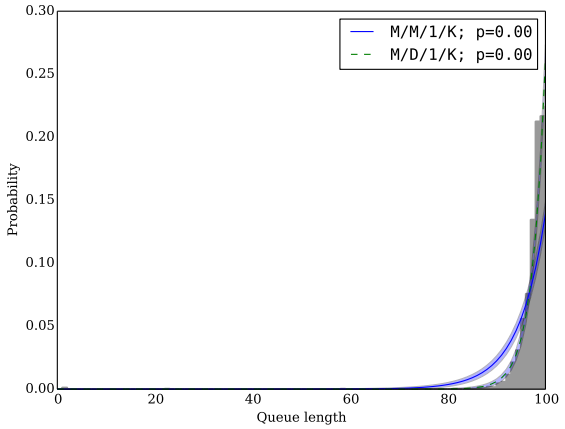
(c) Exponential service time, $\lambda = 2500$ pps (N=5569).



(d) Deterministic service time, $\lambda = 2500$ pps (N=5584).

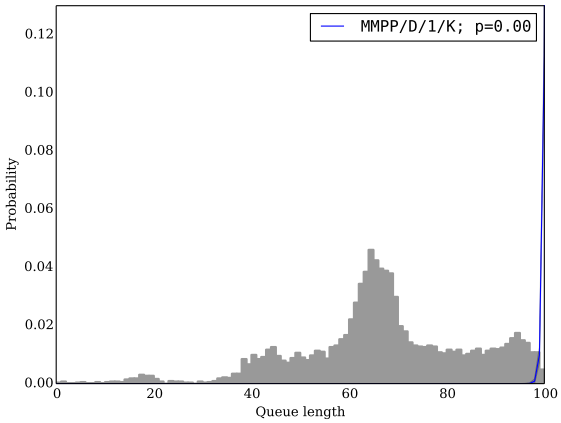


(e) Exponential service time, $\lambda = 3000$ pps (N=5147).

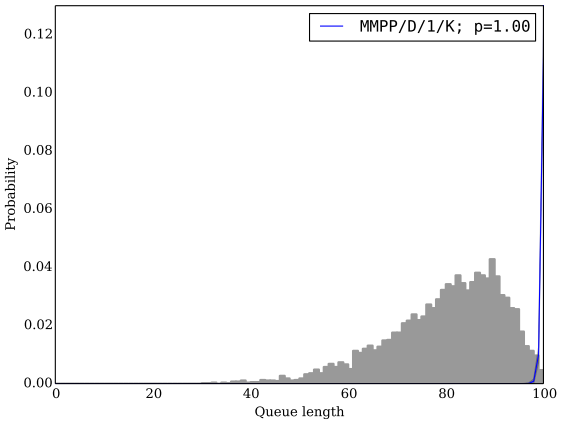


(f) Deterministic service time, $\lambda = 3000$ pps (N=5387).

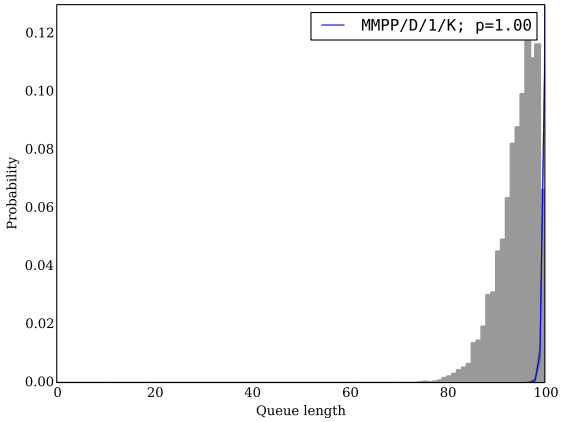
Figure 5.1: Steady-state queue length measurement results for traffic generated by the traffic generator. The graphs in the left column show traffic with exponential service time (varying packet size), while the right column shows deterministic service time (fixed packet size). Each graph shows the normalised histogram of the observed queue lengths with the predicted values for the M/M/1/K and M/D/1/K queues superimposed. The shaded areas around the lines show the 95th percentile interval (i.e. $\pm$ two standard deviations) of the predicted values. The test probability for a χ^2 test comparing the observations to the predicted values is shown in the plot legends. The values of N are the total number of observations for each measurement.



(a) 1 TCP flow (N=5563).

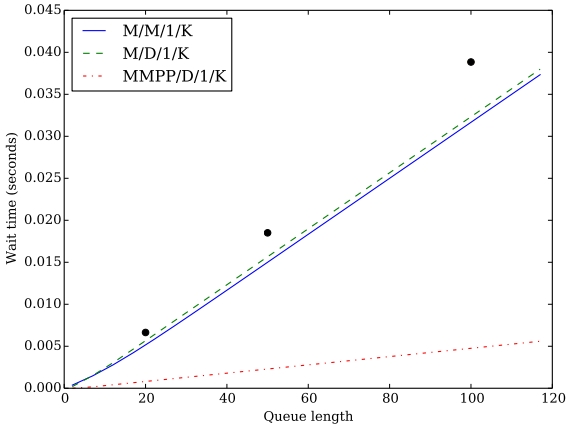


(b) 5 TCP flows (N=5583).

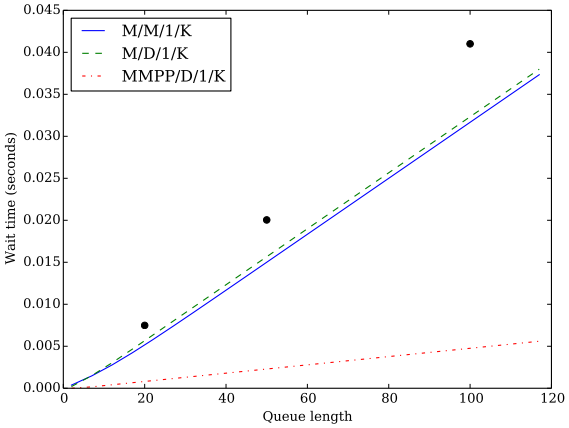


(c) 30 TCP flows (N=5613).

Figure 5.2: Steady-state queue length measurement results for TCP traffic for different numbers of concurrent flows. Each graph shows the normalised histogram of the observed queue lengths with the predicted values for the MMPP/D/1/K queue superimposed. While the 95th percentile interval of the predicted values is plotted, it is not discernible on the graphs. The test probability for a χ^2 test comparing the observations to the predicted values is shown in the plot legends. The values of N are the total number of observations for each measurement.



(a) Exponential service time.



(b) Deterministic service time.

Figure 5.3: Waiting time measurement as a function of queue length for the two different types of traffic. Each graph shows the measured average waiting time with 95th percentile interval (too small to be discernible on the graphs), as well as the predicted values for the different models (the predicted values are the same for both graphs). The parameters used for the measurements and model predictions are $\lambda = 3000, \mu = 2500$.

In summary, the packets are distributed quite evenly between the active queues, and to a very large extent they behave like totally isolated queues. However, the shared queue space makes the results deviate somewhat from what would be expected from completely independent queues, which fits well with the expectations.

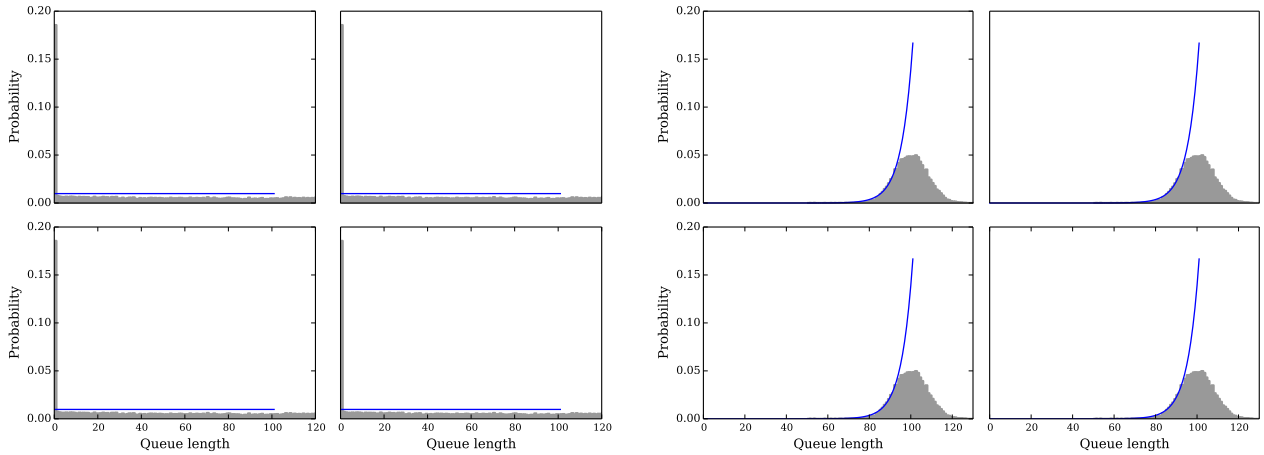
5.4 Summarising the results

The examined models' predictions of the actual observed behaviour of this real-world system show some correlation between predicted and observed values, even though the correlation is not statistically significant. For the TCP traffic, the correlation is non-existent, as expected from the MMPP model calculations' insensitivity to the chosen parameterisation.

Apart from the correlation with the model predictions, the results do clearly demonstrate various features of the observed traffic. This includes the efficiency of packet transmission in the Linux kernel, causing queues to be empty more frequently than predicted at lower speeds. It also includes the difference in behaviour between traffic with exponential and deterministic service times, and the fact that this difference is reflected in the model predictions at higher traffic rates. This, coupled with the discernible difference in observed values at different traffic rates, validates the measurement and traffic generation methodology.

Furthermore, the interaction between the TCP control loop and the queue is clearly visible, as is the effect of the aggregation of several TCP flows. Finally, waiting time, one of the primary measures of interest in network performance evaluation, is shown to be related linearly to queue length as the models predict, although the exact relation does not match the predicted values.

All things considered, the single queue experiments indicate that *for traffic that matches the assumptions of the models*, the queueing theory model predictions can be used as a reasonable qualitative indicator of the system behaviour. The simplifications inherent in the modelling process prevents the predictions from being statistically significant, but for applications where this is not a strict requirement, such as comparing different systems under the same assumptions, these types of models can be used to evaluate system behaviour with reasonable accuracy.



(a) Arrivals at 2500 pps.

(b) Arrivals at 3000 pps.

Figure 5.4: Queue length measurements for fairness queueing. Each figure shows four of the ten active queues during the measurement (the remaining six queues are left out for clarity, but look much the same). The traffic has exponential arrival and service times. The lines are the predicted queue lengths for the M/M/1/K models with parameters λ and μ set to be $1/10$th of the values for the total traffic. Notice the spike at queue length 1 on the graphs in (a), similar to the single queue results.

Finally, the results show that fairness queueing can to a large extent be thought of as a set of completely separate queues with service rate set as a proportional fraction of the overall system service rate. However, as expected in the Linux implementation, the shared queueing space makes the queue lengths be distributed differently from what a separate queue interpretation would suggest, due to the absence of a hard limit on queue lengths.

In summary, the results give a fairly good insight into the behaviour and limitations of queueing theory models when applied to a real-world system, and they highlight the importance of having the model assumptions match the actual parameters of the system being modelled.

6 Conclusions and further work

Three queueing theory models have been formulated to model the experimental setup featuring a real-world Linux-based system, and the numerical calculations of the model predictions have been compared to the observed values. The goal of the experiments has been to ascertain how well the models predict the observed queue length distributions, how well they predict the queue waiting times, and to what extent the fairness queueing implementation in the Linux kernel can be considered to have completely independent queues.

The model predictions for the three examined models have been compared to the experimental queue length and waiting time observations. For the M/M/1/K and M/D/1/K models, traffic was generated specifically to match the model assumptions (with exponentially distributed arrival intervals and exponential or deterministic service times). The results of this show some correlation between model predictions and observations for these models, even though it does not hold up to a rigorous statistical test. For TCP traffic (being compared to the MMPP/D/1/K model predictions) this correlation is almost completely absent; although this is most likely due to the fact that the numerical calculations for the MMPP model produce dubious results that are almost entirely independent of the choice of parameterisation. While every effort has been made to improve on the calculations, insufficient information has been available to do so, even though others have found the model to match well with observed traffic.

Setting aside the spurious model calculations, the observations of the system indicate that for traffic that matches the assumptions of the models, the queueing theory model predictions can be used as a reasonable qualitative indicator of the system behaviour. Furthermore, the experiments reveal some interesting features of the examined traffic. This includes the efficiency of packet transmission in the Linux kernel, which causes the queues to be more likely to be empty than predicted. Also included is the difference in behaviour between traffic with exponential and deterministic service times and between different traffic rates, which is to some extent matched by the model predictions and which validates the measurement and traffic generation methodology. Lastly, the interaction of the TCP control loop with the queue is apparent in the observations, corresponding well to what is expected from the theory of TCP operations.

Finally, for the fairness queueing measurements, the results show that fairness queueing can to a large extent be thought of as a set of completely separate queues with service rate set to a proportional fraction of the overall system service rate. However, in the Linux kernel implementation, the shared queueing space makes the queue lengths be distributed differently from what a separate queue interpretation would suggest, due to the absence of a hard limit on queue lengths.

6.1 Further work

There are several areas of interest where further investigations might be worthwhile. The most important ones are summarised in the following.

An obvious point of further interest has been alluded to already: investigating the reason for the MMPP model's poor prediction performance, and what might be done to remedy this. It seems plausible that given a more thorough investigation than that which has been possible within the time constraints of this thesis, it would be possible to extract model predictions from the model that would be a better match for the real traffic, allowing the MMPP model to be more fully included in the comparison with real-world behaviour.

Similarly, for the simpler models, the qualitative correspondence between model and observation suggests that it might be possible to adapt the models to a better quantitative fit with the real-world results, or at least to quantify the discrepancies. Supposing this is viable, this quantification could be incorporated into the model predictions, in which case the results could be useful for some use cases. For example, as long as the discrepancy is bounded, the modelling approach could be used to compare different systems to each other; even if the results are not a completely reliable predictor of performance, it could still be used to evaluate which of several alternative systems best fits the requirements for a certain task.

Apart from these direct continuations of the work described by this thesis, several topics that impact overall network system performance have been excluded from the present analysis. One obvious area is the analysis of transient behaviour, rather than the steady-state measurements performed in this thesis. Since a large portion of traffic on the internet consist of short-lived connections, analysing the transient queue behaviour in the startup phase is an important element in the overall queue behaviour analysis; however, it also requires an entirely different way of working with the models, which is the reason it has been left out here.

Another area that has been deemed to be out of scope in this thesis, is the addition of queue management algorithms of various forms, such as the CoDel algorithm. This class of algorithm plays an important role in controlling the queueing latency, but they present a difficult modelling problem because of the more complex interactions between different parts of the system, compared to which implementing a queue management algorithm in a simulation is relatively straight-forward. With this in mind, doing a comparison between simulations and real-world behaviour similar to that presented here might be a more realistic approach to the inclusion of queue management algorithms.

Finally, this thesis has validated the statistics broadcast mechanism introduced in the kernel patch. Thus, continuing to work with the bufferbloat community to introduce a similar, but more widely applicable, mechanism to the Linux kernel is an obvious possibility for further investigation. This is corroborated by the stated interest of members of the community in such a mechanism. The starting point of this investigation could be finding a way to get the netlink packet construction out of the transmit fast path, for example by delegating it to a separate kernel thread. The end result could very well be a kernel patch that can be used in other contexts than the isolated experiments provided in this thesis, for example to instrument real-time configuration and parameter tuning in a Linux-based gateway.

Summing up, there are several areas that could be starting points for further investigation of the relationship between models, theory, simulation and practice, and many of them could conceivably play a part in getting to grips with the overall bufferbloat and latency issues affecting the internet.

7 References

[Ben06] Christian Benvenuti. *Understanding Linux Network Internals*. First edition. O'Reilly, 2006.

[BG00] Olivier Brun and Jean-Marie Garcia. 'Analytical Solution of Finite Capacity M/D/1 Queues'. In: *J. Appl. Prob.* 37 (2000), pp. 1092–1098.

[Blo89] C. Blondia. 'The N/G/1 Finite Capacity Queue'. In: *Commun. Statist.Stochastic Models* 5.2 (1989), pp. 273–294.

[Bol+06] Gunter Bolch et al. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. 2nd ed. Wiley, 2006.

[Bro06] Martin A. Brown. *Traffic Control HOWTO*. Online HOWTO. Oct. 2006. URL: `http://www.tldp.org/HOWTO/Traffic-Control-HOWTO/`.

[Cor11] Jonathan Corbet. *Network transmit queue limits*. LWN Article. Aug. 2011. URL: `https://lwn.net/Articles/454390/`.

[FM93] W. Fischer and K. Meier-Hellstein. 'The Markov-Modulated Poisson Process (MMPP) Cookbook'. In: *Performance Evaluation* 18.2 (1993), pp. 149–171.

[Get13] Jim Gettys. *Traditional AQM is not enough!* Blog post. July 2013. URL: `https://gettys.wordpress.com/2013/07/10/low-latency-requires-smart-queuing-traditional-aqm-is-not-enough/`.

[GN11] Jim Gettys and Kathleen Nichols. 'Bufferbloat: Dark Buffers in the Internet'. In: *Queue* 9.11 (Nov. 2011), 40:40–40:54. ISSN: 1542-7730. DOI: `10.1145/2063166.2071893`. URL: `http://doi.acm.org/10.1145/2063166.2071893`.

[Hem13] Stephen Hemminger. *Private email communcation*. June 2013.

[Her11] Tom Herbert. *bql: Byte Queue Limits*. Patch posted to the Linux kernel network development mailing list. Nov. 2011. URL: `http://article.gmane.org/gmane.linux.network/213308/`.

[HL86] Harry Heffes and David M. Lucantoni. 'A Markov Modulated Characterization of Packetized Voice and Data Traffic and Related Statistical Multiplexer Performance'. In: *IEEE Journal on Selected Areas in Communicaitons* SAC-4.6 (1986).

[HR11] Sangtae Ha and Injong Rhee. 'Taming the elephants: New TCP slow start'. In: *Computer Networks* 55.9 (2011), pp. 2092–2110. ISSN: 1389-1286. DOI: `10.1016/j.comnet.2011.01.014`. URL: `http://www.sciencedirect.com/science/article/pii/S1389128611000363`.

[HRX08] Sangtae Ha, Injong Rhee and Lisong Xu. 'CUBIC: a new TCP-friendly high-speed TCP variant'. In: *SIGOPS Oper. Syst. Rev.* 42.5 (July 2008), pp. 64–74. ISSN: 0163-5980. DOI: `10.1145/1400097.1400105`. URL: `http://doi.acm.org/10.1145/1400097.1400105`.

[Høi12] Toke Høiland-Jørgensen. *Battling Bufferbloat: An experimental comparison of four approaches to queue management in Linux*. Project Report, Roskilde University. Dec. 2012. URL: `http://rudar.ruc.dk/handle/1800/9322`.

[Høi13] Toke Høiland-Jørgensen. *Feedback request for kernel patch*. Mailing list posting on the 'bloat-devel' mailing list. June 2013. URL: `https://lists.bufferbloat.net/pipermail/bloat-devel/2013-June/000430.html`.

[ISOC] The Internet Society. *Workshop on Reducing Internet Latency*. 25th-26th Sept. 2013. URL: `http://www.internetsociety.org/latency2013`.

[Jon13] Rick Jones. *Netperf*. Open source benchmarking software. 2013. URL: `http://www.netperf.org/`.

[Lel+94] W. Leland et al. 'On the Self-Similar Nature of Ethernet Traffic (Extended Version)'. In: *IEEE/ACM Transactions on Networking* 2.1 (1994).

[Linux] *Linux kernel source v3.9.9*. Published at kernel.org. July 2013. URL: https://www.kernel.org/pub/linux/kernel/v3.0/linux-3.9.9.tar.bz2.

[McK90] P.E. McKenney. 'Stochastic fairness queueing'. In: *INFOCOM '90. Ninth Annual Joint Conference of the IEEE Computer and Communication Societies. 'The Multiple Facets of Integration'. Proceedings., IEEE*. (An updated and extended version of the paper is available at http://www.rdrop.com/users/paulmck/scalability/paper/sfq.2002.06.04.pdf.) June 1990, 733–740 vol.2. DOI: 10.1109/INFCOM.1990.91316.

[NJ12] Kathleen Nichols and Van Jacobson. 'Controlling queue delay'. In: *Commun. ACM* 55.7 (July 2012), pp. 42–50. ISSN: 0001-0782. DOI: 10.1145/2209249.2209264. URL: http://doi.acm.org/10.1145/2209249.2209264.

[Sta12] CACM Staff. 'BufferBloat: what's wrong with the internet?' In: *Commun. ACM* 55.2 (Feb. 2012), pp. 40–47. ISSN: 0001-0782. DOI: 10.1145/2076450.2076464. URL: http://doi.acm.org/10.1145/2076450.2076464.

[Sti13] Martin Stiemerling. *Proposed Charter for "Active Queue Management and Packet Scheduling" (aqm) WG*. Aug. 2013. URL: https://datatracker.ietf.org/doc/charter-ietf-aqm/.

[SV96] M. Shreedhar and G. Varghese. 'Efficient fair queuing using deficit round-robin'. In: *Networking, IEEE/ACM Transactions on* 4.3 (1996), pp. 375–385. ISSN: 1063-6692. DOI: 10.1109/90.502236.

[SVA91] Efstathios D. Sykas, Konstantinos M Vlakos and Nikolaos G Anerousis. 'Performance evaluation of statistical multiplexing schemes in ATM networks'. In: *Computer Communications* 14.5 (June 1991), pp. 273–286.

[Tah13] Dave Taht. *Comments on kernel patch*. Mailing list posting on the 'bloat-devel' mailing list. June 2013. URL: https://lists.bufferbloat.net/pipermail/bloat-devel/2013-June/000433.html.

[TG12] Dave Taht and Jim Gettys. *Best practices for benchmarking Codel and FQ Codel*. Wiki page on bufferbloat.net web site. Dec. 2012. URL: https://www.bufferbloat.net/projects/codel/wiki/Best_practices_for_benchmarking_Codel_and_FQ_Codel.

[Uni+13] Univ. of Tennessee et al. *LAPACK - Linear Algebra PACKage*. 2013. URL: http://www.netlib.org/lapack/.

[WB13] Keith Winstein and Hari Balakrishnan. 'TCP ex Machina: Computer-Generated Congestion Control'. In: *Proceedings of the ACM SIGCOMM 2013 conference*. Aug. 2013.

[RFC793] J. Postel. *Transmission Control Protocol*. RFC 793 (Internet Standard). Updated by RFCs 1122, 3168, 6093, 6528. Internet Engineering Task Force, Sept. 1981. URL: http://www.ietf.org/rfc/rfc793.txt.

[RFC2001] W. Stevens. *TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms*. RFC 2001 (Proposed Standard). Obsoleted by RFC 2581. Internet Engineering Task Force, Jan. 1997. URL: http://www.ietf.org/rfc/rfc2001.txt.

[RFC2581] M. Allman, V. Paxson and W. Stevens. *TCP Congestion Control*. RFC 2581 (Proposed Standard). Obsoleted by RFC 5681, updated by RFC 3390. Internet Engineering Task Force, Apr. 1999. URL: http://www.ietf.org/rfc/rfc2581.txt.

[RFC3168] K. Ramakrishnan, S. Floyd and D. Black. *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC 3168 (Proposed Standard). Updated by RFCs 4301, 6040. Internet Engineering Task Force, Sept. 2001. URL: http://www.ietf.org/rfc/rfc3168.txt.

[RFC3390] M. Allman, S. Floyd and C. Partridge. *Increasing TCP's Initial Window*. RFC 3390 (Proposed Standard). Internet Engineering Task Force, Oct. 2002. URL: http://www.ietf.org/rfc/rfc3390.txt.

[RFC3549] J. Salim et al. *Linux Netlink as an IP Services Protocol*. RFC 3549 (Informational). Internet Engineering Task Force, July 2003. URL: http://www.ietf.org/rfc/rfc3549.txt.

[RFC5681] M. Allman, V. Paxson and E. Blanton. *TCP Congestion Control*. RFC 5681 (Draft Standard). Internet Engineering Task Force, Sept. 2009. URL: http://www.ietf.org/rfc/rfc5681.txt.

[RFC6582] T. Henderson et al. *The NewReno Modification to TCP's Fast Recovery Algorithm*. RFC 6582 (Proposed Standard). Internet Engineering Task Force, Apr. 2012. URL: http://www.ietf.org/rfc/rfc6582.txt.

[RFC6817] S. Shalunov et al. *Low Extra Delay Background Transport (LEDBAT)*. RFC 6817 (Experimental). Internet Engineering Task Force, Dec. 2012. URL: http://www.ietf.org/rfc/rfc6817.txt.

[RFC6928] J. Chu et al. *Increasing TCP's Initial Window*. RFC 6928 (Experimental). Internet Engineering Task Force, Apr. 2013. URL: http://www.ietf.org/rfc/rfc6928.txt.

8 Appendix: source code

8.1 Kernel patch

This includes the parts of the kernel patch that alters the statistics gathering of the `fq_codel` qdisc, and the part that adds the statistics broadcast. The full patch is available at `https://git.tohojo.dk/qstatsc/tree/kernel-patch/broadcast-stats.patch`.

```
diff --git a/net/core/dev.c b/net/core/dev.c
index c9eb9e6..dc2c6c3 100644
--- a/net/core/dev.c
+++ b/net/core/dev.c
@@ -2636,6 +2636,7 @@ static inline int __dev_xmit_skb(struct sk_buff *skb, struct Qdisc *q,
         kfree_skb(skb);
         rc = NET_XMIT_DROP;
     } else if ((q->flags & TCQ_F_CAN_BYPASS) && !qdisc_qlen(q) &&
+       !dev_net(dev)->ipv4.sysctl_qdisc_disable_bypass &&
            qdisc_run_begin(q)) {
         /*
          * This is a work-conserving queue; there are no old skbs
@@ -2660,6 +2661,10 @@ static inline int __dev_xmit_skb(struct sk_buff *skb, struct Qdisc *q,
     } else {
         skb_dst_force(skb);
         rc = q->enqueue(skb, q) & NET_XMIT_MASK;
+#ifdef CONFIG_NET_SCH_BROADCAST_STATS
+       qdisc_broadcast_stats(q);
+#endif
+
         if (qdisc_run_begin(q)) {
             if (unlikely(contended)) {
                 spin_unlock(&q->busylock);
diff --git a/net/sched/sch_fq_codel.c b/net/sched/sch_fq_codel.c
index 5578628..61f0c0b 100644
--- a/net/sched/sch_fq_codel.c
+++ b/net/sched/sch_fq_codel.c
@@ -55,6 +55,7 @@ struct fq_codel_sched_data {
     struct tcf_proto *filter_list;  /* optional external classifier */
     struct fq_codel_flow *flows;    /* Flows table [flows_cnt] */
     u32     *backlogs;  /* backlog table [flows_cnt] */
+   u8      *qlens;     /* qlen table [flows_cnt] */
     u32     flows_cnt;  /* number of flows */
     u32     perturbation;   /* hash perturbation */
     u32     quantum;    /* psched_mtu(qdisc_dev(sch)); */
@@ -63,6 +64,10 @@ struct fq_codel_sched_data {
     u32     drop_overlimit;
     u32     new_flow_count;

+#ifdef CONFIG_NET_SCH_BROADCAST_STATS
+   void        *stats_buffer;
+#endif
+
     struct list_head new_flows; /* list of new flows */
     struct list_head old_flows; /* list of old flows */
 };
@@ -159,6 +164,7 @@ static unsigned int fq_codel_drop(struct Qdisc *sch)
     skb = dequeue_head(flow);
     len = qdisc_pkt_len(skb);
     q->backlogs[idx] -= len;
+   q->qlens[idx]--;
     kfree_skb(skb);
     sch->q.qlen--;
     sch->qstats.drops++;
@@ -187,6 +193,7 @@ static int fq_codel_enqueue(struct sk_buff *skb, struct Qdisc *sch)
     flow = &q->flows[idx];
     flow_queue_add(flow, skb);
     q->backlogs[idx] += qdisc_pkt_len(skb);
+   q->qlens[idx]++;
     sch->qstats.backlog += qdisc_pkt_len(skb);

     if (list_empty(&flow->flowchain)) {
@@ -224,6 +231,7 @@ static struct sk_buff *dequeue(struct codel_vars *vars, struct Qdisc *sch)
```

```
        if (flow->head) {
            skb = dequeue_head(flow);
66          q->backlogs[flow - q->flows] -= qdisc_pkt_len(skb);
+           q->qlens[flow - q->flows]--;
            sch->q.qlen--;
        }
        return skb;
71  @@ -379,7 +387,11 @@ static void fq_codel_destroy(struct Qdisc *sch)

        tcf_destroy_chain(&q->filter_list);
        fq_codel_free(q->backlogs);
+       fq_codel_free(q->qlens);
76      fq_codel_free(q->flows);
+#ifdef CONFIG_NET_SCH_BROADCAST_STATS
+       fq_codel_free(q->stats_buffer);
+#endif
 }
81
 static int fq_codel_init(struct Qdisc *sch, struct nlattr *opt)
@@ -413,6 +425,22 @@ static int fq_codel_init(struct Qdisc *sch, struct nlattr *opt)
            fq_codel_free(q->flows);
            return -ENOMEM;
86      }
+       q->qlens = fq_codel_zalloc(q->flows_cnt * sizeof(u8));
+       if (!q->qlens) {
+           fq_codel_free(q->flows);
+           fq_codel_free(q->backlogs);
91 +           return -ENOMEM;
+       }
+#ifdef CONFIG_NET_SCH_BROADCAST_STATS
+       q->stats_buffer = fq_codel_zalloc(sizeof(struct tc_fq_codel_xstats) +
+                         q->flows_cnt * sizeof(struct tc_fq_codel_flow_stats));
96 +       if (!q->stats_buffer) {
+           fq_codel_free(q->flows);
+           fq_codel_free(q->backlogs);
+           fq_codel_free(q->qlens);
+           return -ENOMEM;
101 +       }
+#endif
        for (i = 0; i < q->flows_cnt; i++) {
            struct fq_codel_flow *flow = q->flows + i;

106 @@ -464,6 +492,9 @@ static int fq_codel_dump_stats(struct Qdisc *sch, struct gnet_dump *d)
        .type               = TCA_FQ_CODEL_XSTATS_QDISC,
    };
    struct list_head *pos;
+#ifdef CONFIG_NET_SCH_BROADCAST_STATS
111 +   struct tc_fq_codel_flow_stats *buf; int ret; unsigned int i; struct tc_fq_codel_flow_stats fst;
+#endif

    st.qdisc_stats.maxpacket = q->cstats.maxpacket;
    st.qdisc_stats.drop_overlimit = q->drop_overlimit;
116 @@ -476,7 +507,26 @@ static int fq_codel_dump_stats(struct Qdisc *sch, struct gnet_dump *d)
    list_for_each(pos, &q->old_flows)
        st.qdisc_stats.old_flows_len++;

+#ifdef CONFIG_NET_SCH_BROADCAST_STATS
121 +   st.qdisc_stats.act_flows_count = 0;
+   buf = q->stats_buffer + sizeof(st);
+   for(i = 0; i < q->flows_cnt; i++) {
+       if(q->qlens[i] > 0) {
+           st.qdisc_stats.act_flows_count++;
126 +           fst.flow_id = i;
+           fst.qlen = q->qlens[i];
+           fst.backlog = q->backlogs[i];
+           fst.delay = codel_time_to_us(q->flows[i].cvars.ldelay);
+           memcpy(buf++, &fst, sizeof(fst));
131 +       }
+   }
+   memcpy(q->stats_buffer, &st, sizeof(st));
+   ret = gnet_stats_copy_app(d, q->stats_buffer,
+                 sizeof(st) + st.qdisc_stats.act_flows_count * sizeof(fst));
136 +   return ret;
```

```
+#else
     return gnet_stats_copy_app(d, &st, sizeof(st));
+#endif
 }

 static struct Qdisc *fq_codel_leaf(struct Qdisc *sch, unsigned long arg)
@@ -527,7 +577,6 @@ static int fq_codel_dump_class_stats(struct Qdisc *sch, unsigned long cl,

     if (idx < q->flows_cnt) {
         const struct fq_codel_flow *flow = &q->flows[idx];
-        const struct sk_buff *skb = flow->head;

         memset(&xstats, 0, sizeof(xstats));
         xstats.type = TCA_FQ_CODEL_XSTATS_CLASS;
@@ -545,10 +594,7 @@ static int fq_codel_dump_class_stats(struct Qdisc *sch, unsigned long cl,
                 codel_time_to_us(delta) :
                 -codel_time_to_us(-delta);
         }
-        while (skb) {
-            qs.qlen++;
-            skb = skb->next;
-        }
+        qs.qlen = q->qlens[idx];
         qs.backlog = q->backlogs[idx];
         qs.drops = flow->dropped;
     }
diff --git a/net/sched/sch_generic.c b/net/sched/sch_generic.c
index 2022408..ede9916 100644
--- a/net/sched/sch_generic.c
+++ b/net/sched/sch_generic.c
@@ -28,6 +28,7 @@
 #include <net/sch_generic.h>
 #include <net/pkt_sched.h>
 #include <net/dst.h>
+#include <net/netlink.h>

 /* Main transmission queue. */

@@ -101,6 +102,88 @@ static inline int handle_dev_cpu_collision(struct sk_buff *skb,
     return ret;
 }

+#ifdef CONFIG_NET_SCH_BROADCAST_STATS
+static inline u64 qdisc_stats_time(void)
+{
+   u64 ns = ktime_to_ns(ktime_get());
+   do_div(ns, NSEC_PER_USEC);
+   return ns;
+}
+
+int qdisc_broadcast_stats(struct Qdisc *q)
+{
+   struct tcmsg *tcm;
+   struct nlmsghdr *nlh;
+   struct gnet_dump d;
+   struct sk_buff *skb;
+   struct net *net;
+   unsigned char *b;
+   u64 time;
+
+   if(!q->dev_queue || !q->dev_queue->dev)
+       return 0;
+
+   net = dev_net(qdisc_dev(q));
+
+   if(!netlink_has_listeners(net->rtnl, RTNLGRP_TC_STATS))
+       return 0;
+
+   time = qdisc_stats_time();
+   if(time < q->last_stats_broadcast +
+       net->ipv4.sysctl_qdisc_stats_broadcast_interval)
+       return 0;
+
```

```
+   skb = alloc_skb(NLMSG_SPACE(1024), GFP_ATOMIC);
+   if(!skb)
+       return -ENOBUFS;
+   b = skb_tail_pointer(skb);
+
+   nlh = nlmsg_put(skb, 0, 0, RTM_QDISC_STATS, sizeof(*tcm), NLM_F_MULTI);
+   if (!nlh)
+       goto out_free;
+
+   tcm = nlmsg_data(nlh);
+   tcm->tcm_family = AF_UNSPEC;
+   tcm->tcm__pad1 = 0;
+   tcm->tcm__pad2 = 0;
+   tcm->tcm_ifindex = qdisc_dev(q)->ifindex;
+   tcm->tcm_parent = q->parent;
+   tcm->tcm_handle = q->handle;
+   tcm->tcm_info = atomic_read(&q->refcnt);
+
+   if (nla_put_string(skb, TCA_KIND, q->ops->id))
+       goto nla_put_failure;
+
+   if (gnet_stats_start_copy(skb, TCA_STATS2, NULL, &d) < 0)
+       goto nla_put_failure;
+
+   if (q->ops->dump_stats && q->ops->dump_stats(q, &d) < 0)
+       goto nla_put_failure;
+
+   q->qstats.qlen = q->q.qlen;
+   if (gnet_stats_copy_basic(&d, &q->bstats) < 0 ||
+       gnet_stats_copy_queue(&d, &q->qstats) < 0)
+       goto nla_put_failure;
+
+   if (gnet_stats_finish_copy(&d) < 0)
+       goto nla_put_failure;
+
+   nlh->nlmsg_len = skb_tail_pointer(skb) - b;
+
+   nlmsg_notify(net->rtnl, skb, 0, RTNLGRP_TC_STATS, 0, 0);
+
+   q->last_stats_broadcast = time;
+
+   return 0;
+
+nla_put_failure:
+out_free:
+   kfree_skb(skb);
+   return -1;
+
+}
+#endif
+
 /*
  * Transmit one skb, and handle the return status as required. Holding the
  * __QDISC_STATE_RUNNING bit guarantees that only one CPU can execute this
@@ -115,6 +198,9 @@ int sch_direct_xmit(struct sk_buff *skb, struct Qdisc *q,
            spinlock_t *root_lock)
 {
    int ret = NETDEV_TX_BUSY;
+#ifdef CONFIG_NET_SCH_BROADCAST_STATS
+   qdisc_broadcast_stats(q);
+#endif

    /* And release qdisc */
    spin_unlock(root_lock);
@@ -565,6 +651,9 @@ struct Qdisc *qdisc_alloc(struct netdev_queue *dev_queue,
    sch->enqueue = ops->enqueue;
    sch->dequeue = ops->dequeue;
    sch->dev_queue = dev_queue;
+#ifdef CONFIG_NET_SCH_BROADCAST_STATS
+   sch->last_stats_broadcast = qdisc_stats_time();
+#endif
    dev_hold(dev);
    atomic_set(&sch->refcnt, 1);
```

8.2 Statistics reader

This includes the part of the statistics reader that communicates with the netlink interface and reads the incoming data, as well as the qdisc-specific handler for the `codel` qdisc. The full source code for the statistics reader is available at `https://git.tohojo.dk/qstatsc/tree/`.

```
/**
 * netlink_comm.c
 *
 * Toke Høiland-Jørgensen
 * 2013-06-04
 */

#include <dlfcn.h>
#include <stdio.h>

#include <netlink/msg.h>
#include <netlink/netlink.h>
#include <netlink/attr.h>
#include <linux/gen_stats.h>

#include "netlink_comm.h"
#include "formatter.h"

static struct nla_policy tca_policy[TCA_MAX+1] = {
        [TCA_KIND] = { .type = NLA_STRING,
                       .maxlen = IFNAMSIZ },
        [TCA_STATS2] = { .type = NLA_NESTED },
};

static struct nla_policy tca_stats_policy[TCA_STATS_MAX+1] = {
  [TCA_STATS_BASIC] = { .minlen = sizeof(struct gnet_stats_basic)},
  [TCA_STATS_QUEUE] = { .minlen = sizeof(struct gnet_stats_queue)},
};

static struct qdisc_handler *qdisc_handler_list;

static int netlink_msg_handler(struct nl_msg *msg, void *arg);

struct nl_sock *create_socket()
{
    return nl_socket_alloc();
}

void destroy_socket(struct nl_sock *sk)
{
    nl_socket_free(sk);
}

int setup_socket(struct nl_sock *sk, struct options *opt)
{
    int ret;
    nl_socket_disable_seq_check(sk);

    if((ret = nl_socket_modify_cb(sk, NL_CB_VALID, NL_CB_CUSTOM, netlink_msg_handler, opt)) < 0)
        return ret;

    if((ret = nl_connect(sk, NETLINK_ROUTE)) < 0)
        return ret;

    if((ret = nl_socket_add_memberships(sk, RTNLGRP_TC_STATS, 0)) < 0)
        return ret;
    return 0;
}


static int netlink_msg_handler(struct nl_msg *msg, void *arg)
{
    struct nlmsghdr *hdr;
    struct tcmsg *tcm;
    struct nlattr *attrs[TCA_MAX+1];
    struct nlattr *stat_attrs[TCA_STATS_MAX+1];
```

```
    struct qdisc_handler *h;
    char qdisc[IFNAMSIZ] = {0};
    int qdisc_l = 0;
    char ifname[IFNAMSIZ] = {0};
    int ifname_l = 0;
    struct timeval current_time = {0};
    double time;

    char *ret = NULL;

    struct gnet_stats_basic *sb;
    struct gnet_stats_queue *q;

    struct options *opt = arg;

    struct recordset rset = {0};

    hdr = nlmsg_hdr(msg);
    tcm = nlmsg_data(hdr);
    if(!has_iface(opt, tcm->tcm_ifindex))
        return NL_SKIP;

    if((ret = rtnl_link_i2name(opt->cache, tcm->tcm_ifindex, ifname, IFNAMSIZ)) == NULL)
        return NL_SKIP;

    // No length checking in netlink library.
    if((ifname_l = 1 + strnlen(ifname, IFNAMSIZ)) >= IFNAMSIZ) {
        ifname[IFNAMSIZ-1] = '\0';
        ifname_l = IFNAMSIZ;
    }

    gettimeofday(&current_time, NULL);
    time = (double)current_time.tv_usec / 1000000 + (double) current_time.tv_sec;
    add_record_double(&rset, "time", sizeof("time"), time);
    add_record_str(&rset, "iface", sizeof("iface"), ifname, ifname_l);

    nlmsg_parse(hdr, sizeof(*tcm), attrs, TCA_MAX, tca_policy);

    if(attrs[TCA_KIND]) {
        qdisc_l = nla_len(attrs[TCA_KIND]);
        memcpy(qdisc, nla_get_string(attrs[TCA_KIND]), qdisc_l);
        add_record_str(&rset, "qdisc", sizeof("qdisc"), qdisc, qdisc_l);
    }

    add_record_hex(&rset, "handle", sizeof("handle"), tcm->tcm_handle >> 16);

    if(attrs[TCA_STATS2]) {
        nla_parse_nested(stat_attrs, TCA_STATS_MAX, attrs[TCA_STATS2], tca_stats_policy);
        if(stat_attrs[TCA_STATS_BASIC]) {
            sb = nla_data(stat_attrs[TCA_STATS_BASIC]);
            add_record_uint(&rset, "bytes", sizeof("bytes"), sb->bytes);
            add_record_uint(&rset, "packets", sizeof("packets"), sb->packets);
        }

        if(stat_attrs[TCA_STATS_QUEUE]) {
            q = nla_data(stat_attrs[TCA_STATS_QUEUE]);
            add_record_uint(&rset, "drops", sizeof("drops"), q->drops);
            add_record_uint(&rset, "qlen", sizeof("qlen"), q->qlen);
            add_record_uint(&rset, "backlog", sizeof("backlog"), q->backlog);
            add_record_uint(&rset, "overlimits", sizeof("overlimits"), q->overlimits);
            add_record_uint(&rset, "requeues", sizeof("requeues"), q->requeues);
        }
        if(stat_attrs[TCA_STATS_APP]) {
            h = find_qdisc_handler(qdisc);
            if(h)
                h->parse_stats(stat_attrs[TCA_STATS_APP], &rset);
        }
    }
    opt->formatter->format(opt->formatter, &rset);
    clear_records(&rset);
    if(!opt->run_length || current_time.tv_sec < opt->start_time.tv_sec + opt->run_length)
        return NL_OK;
    else
```

```
        return NL_STOP;
}

struct qdisc_handler *find_qdisc_handler(const char *name)
{
    char buf[128] = {0};
    void *dlh;
    struct qdisc_handler *h;

    for (h = qdisc_handler_list; h; h = h->next)
        if(strcmp(h->id, name) == 0)
            return h;

    dlh = dlopen(NULL, RTLD_LAZY);
    snprintf(buf, sizeof(buf), "%s_qdisc_handler", name);
    h = dlsym(dlh, buf);
    if(h) {
        h->next = qdisc_handler_list;
        qdisc_handler_list = h;
    }
    return h;
}

/**
 * qdisc_codel.c
 *
 * Toke Høiland-Jørgensen
 * 2013-07-03
 */

#include <linux/pkt_sched.h>

#include "formatter.h"
#include "netlink_comm.h"

static int codel_parse_stats(struct nlattr *attr, struct recordset *rset)
{
    struct tc_codel_xstats *st;

    if(nla_len(attr) < sizeof(*st))
        return -1;

    st = nla_data(attr);

    add_record_uint(rset, "maxpacket", sizeof("maxpacket"), st->maxpacket);
    add_record_uint(rset, "drop_count", sizeof("drop_count"), st->count);
    add_record_uint(rset, "drop_lastcount", sizeof("drop_lastcount"), st->lastcount);
    add_record_uint(rset, "drop_overlimit", sizeof("drop_overlimit"), st->drop_overlimit);
    add_record_uint(rset, "ecn_mark", sizeof("ecn_mark"), st->ecn_mark);
    add_record_uint(rset, "ldelay", sizeof("ldelay"), st->ldelay);
    add_record_uint(rset, "drop_next", sizeof("drop_next"), st->drop_next);
    add_record_uint(rset, "dropping", sizeof("dropping"), st->dropping);

    return 0;
}

struct qdisc_handler codel_qdisc_handler = {
    .id = "codel",
    .parse_stats = codel_parse_stats,
};
```

8.3 Traffic generator

This includes the part of the traffic generator that schedules and sends out the packets. The full source code is available at `https://git.tohojo.dk/traffic-gen/tree/`.

```
/**
 * sender.c
 *
 * Toke Høiland-Jørgensen
```

```
 * 2013-07-01
 */

#include <stdlib.h>
#include <math.h>
#include <sys/socket.h>
#include <netdb.h>
#include "sender.h"
#include "util.h"

static void set_port(struct addrinfo *addr, unsigned short port)
{
    if(addr->ai_family == AF_INET) {
        struct sockaddr_in *saddr = (struct sockaddr_in *)addr->ai_addr;
        saddr->sin_port = htons(port);
    } else if(addr->ai_family == AF_INET6) {
        struct sockaddr_in6 *saddr = (struct sockaddr_in6 *)addr->ai_addr;
        saddr->sin6_port = htons(port);
    }
}

static unsigned short gen_port(range)
{
    return PORT_START + rand() % range;
}

static double exp_distrib(double mean)
{
    // Draw a random value and create an exponentially distributed value
    // from it. Calculated by -ln(r)/mean where r is a random number between
    // 0 and 1
    //
    // Ref: https://en.wikipedia.org/wiki/Exponential_distribution#Generating_exponential_variates
    double r;
    do {
        r = (double)rand() / (double) RAND_MAX;
    } while(r == 0.0); // log(0) is undefined
    return -log(r)/mean;
}

static unsigned int exp_wait(unsigned int pps)
{
    // Exponential wait is an exponentially distributed value with mean set
    // to achieve pps packets/sec.
    return (unsigned int) 1000000 * exp_distrib(pps);
}

static unsigned int scale_payload(unsigned int size, unsigned int overhead)
{
    double scale = exp_distrib(1.0);
    return min(max(0, (size * scale)-overhead), MAX_PAYLOAD);
}

static void schedule_next(unsigned int pps, char poisson, struct timeval *now, struct timeval *next)
{
    int delay;
    if(poisson)
        delay = exp_wait(pps);
    else
        delay = 1000000/pps;
    next->tv_sec = now->tv_sec;
    next->tv_usec = now->tv_usec + delay;
    if(next->tv_usec > 1000000) {
        next->tv_sec += next->tv_usec / 1000000;
        next->tv_usec %= 1000000;
    }
}

void send_loop(struct options *opt)
{
    struct timeval now, next, stop;
    int ret;
    char msg[MAX_PAYLOAD] = {0};
```

```
    int overhead = OVERHEAD(opt->dest->ai_family);
    int payload = opt->pkt_size - overhead;
    char destaddr[INET6_ADDRSTRLEN];

    if(payload < 0) {
        fprintf(stderr, "Minimum packet size for selected host and address family is %d bytes\n",
            overhead);
        return;
    }

    if((ret = getnameinfo(opt->dest->ai_addr, opt->dest->ai_addrlen,
                    destaddr, sizeof(destaddr),
                    NULL, 0,
                    NI_NUMERICHOST)) != 0) {
        fprintf(stderr, "Error creating address string: %s\n", gai_strerror(ret));
        return;
    }

    printf("Sending %d pps of size %d (%d+%d) bytes to %s for %d seconds (%s).\n",
        opt->pps, opt->pkt_size, payload, overhead,
        destaddr,
        opt->run_length,
        opt->poisson_interval ? "poisson" : "deterministic"
        );

    gettimeofday(&now, NULL);
    stop.tv_sec = now.tv_sec + opt->run_length;
    stop.tv_usec = now.tv_usec;
    srand(now.tv_sec ^ now.tv_usec);
    do {
        schedule_next(opt->pps, opt->poisson_interval, &now, &next);
        while(now.tv_sec < next.tv_sec || now.tv_usec < next.tv_usec) {
            if(next.tv_usec - now.tv_usec > USLEEP_THRESHOLD)
                usleep(USLEEP_THRESHOLD);
            gettimeofday(&now, NULL);
        }
        set_port(opt->dest, gen_port(opt->port_range));
        if(opt->poisson_packets)
            payload = scale_payload(opt->pkt_size, overhead);
        sendto(opt->socket, msg, payload, 0, opt->dest->ai_addr, opt->dest->ai_addrlen);
    } while(now.tv_sec < stop.tv_sec || now.tv_usec < stop.tv_usec);
}
```

8.4 Probability calculator for the M/M/1/K queue

```
import numpy as np

class PiCalc(object):
    def __init__(self, kmax, lmbda, mu):
        self.kmax = kmax
        self.lmbda = lmbda
        self.mu = mu
        self.a = self.lmbda/self.mu

    def calc_pival(self, k):
        if k > self.kmax:
            return 0
        if self.lmbda == self.mu:
            return 1 / (self.kmax+1)
        return ((1-self.a)*self.a**k)/(1-self.a**(self.kmax+1))

def calc_probs(l,m,N):
        pc = PiCalc(N, l, m)
        xvals = range(0,N+1)
        return np.asarray(list(map(pc.calc_pival, xvals)))

def calc_wait(l,m,K):
    if l == m:
        return K/(2*l)
    a = l/m
    return 1 / (m*(1-a)) - ((K+1)/(1-a**(K+1)))*(a**(K+1)/l)
```

```
if __name__ == "__main__":
    from mpl_setup import plt
    fig = plt.figure()
    ax = fig.add_subplot(111)
    linestyles = ["-", "--", "-.", ":"]

    for l in (2375, 2500, 3000):
        xvals = range(0,101)
        vals = calc_probs(l, 2500, 100)

        ax.plot(xvals, vals, label="$\lambda={:n}$".format(l),
                linestyle=linestyles.pop(0))

        ax.legend()
        ax.set_xlabel("Queue length")
        ax.set_ylabel("Probability")

        plt.tight_layout()
        fig.savefig("predicted-pi-vals.pdf")
```

8.5 Probability calculator for the M/D/1/K queue

```
import numpy as np
import math

m = 2500
K = 100

def calc_bs(l,m,K):
    p = l/m
    alpha = np.zeros(K+1)
    a = np.zeros(K)
    b = np.zeros(K)

    a[0] = b[0] = 1

    alpha[0] = math.exp(-p)
    for k in range(1,K+1):
        alpha[k] = alpha[k-1]*p/k

    # Relation (2) of Brun and Garcia (2000)
    expp = math.exp(p)
    a[1]=expp-1
    for n in range(2,K):
        sumterm = 0
        for i in range(1,n):
            sumterm += alpha[i]*a[n-i]
        a[n] = expp*(a[n-1]-sumterm-alpha[n-1]*a[0])

    # section 4 of Brun and Garcia (2000)
    b = np.cumsum(a)
    return b

def calc_probs(l,m,K):
    p = l/m
    b = calc_bs(l,m,K)
    P = np.zeros(K+1)
    # Relation (3) of Brun and Garcia (2000)
    P[0] = 1/(1+p*b[K-1])
    P[K] = 1-b[K-1]/(1+p*b[K-1])
    for j in range(1,K):
        P[j]=(b[j]-b[j-1])/(1+p*b[K-1])

    assert P.sum() - 1.0 < 0.0001

    return P


def calc_wait(l,m,K):
    p = l/m
```

```
    b = calc_bs(l,m,K)
    return (K/l - 1/m - b[:K-1].sum()/(l*(1+p * b[K-1])))

if __name__ == "__main__":
    from mpl_setup import plt
    linestyles = ["-", "--", "-.", ":"]

    for l in 2375,2500,3000:
        plt.plot(range(K+1), calc_probs(l,m,K), label="$\lambda={:n}$".format(l),
                 linestyle=linestyles.pop(0))

    plt.legend()
    plt.gca().set_xlabel("Queue length")
    plt.gca().set_ylabel("Probability")
    plt.tight_layout()
    plt.savefig("predicted-m-d-1-n.pdf")
```

8.6 Probability calculator for the MMPP/D/1/K queue

```
import numpy as np
from numpy import linalg as LA
import math

import warnings
warnings.simplefilter('ignore', np.ComplexWarning)

def calc_probs(l,m,N,r):

    qsize=N
    mu=1/m # In Blondia, mu is duration of service period

    r = np.asarray(r)
    alpha = np.asarray([1.0,0.0]) # initial probabilities - Blondia pg 282
    lmbda = np.asarray(l)

    Lmbda = np.asmatrix(np.diag(lmbda))
    T = np.matrix([[-r[0], r[0]],
                   [0, -r[1]]]) # Blondia pg 281
    Tline0 = np.matrix([[0],
                        [r[1]]])
    T0 = np.matrix([[0,0],
                    [r[1],r[1]]])
    A0 = np.asmatrix(np.diag(alpha))
    Qstar = T + T0*A0
    e = np.matrix([[1],
                   [1]])

    # Blondia pg 290
    def R(z):
        return (z-1)*Lmbda+Qstar

    # Blondia appendix
    eta = np.ma.diag(-R(0)).max()
    gamma = np.zeros(200)
    gamma[0] = math.exp(-eta * mu)
    N_0 = None
    epsilon = 10**-10 # Threshold for truncation
    for n in range(1, len(gamma)):
        gamma[n] = (eta * mu/n)*gamma[n-1] # Heffes and Lucatoni pg 867
        if 1-gamma[:n].sum() < epsilon and N_0 is None:
            # Truncation as per Blondia pg 292
            N_0 = n
            gamma_remainder = 1 - gamma[:n].sum()
    if N_0 is None:
        N_0 = n
        gamma_remainder = 0

    # Definitions at the bottom of pg 291, Blondia
    P = 1/eta * T + np.identity(2)
    P0 = 1/eta * Tline0

    K = np.empty((N_0+1, qsize), dtype=object)
```

```
    for n in range(qsize):
        K[0,n] = np.zeros((2,2))
    K[0,0] = np.identity(2)
    for l in range(N_0):
        K[l+1,0] = P*K[l,0]
    for l in range(N_0):
        for n in range(1,qsize):
            K[l+1,n] = P*K[l,n]+P0*alpha*K[l,n-1]

    # Blondia pg 292; using truncation
    A = np.empty(qsize, dtype=object)
    for n in range(qsize):
        A[n] = np.zeros((2,2))
        for l in range(n, N_0+1):
            A[n]+=gamma[l]*K[l,n]
        A[n]+=gamma_remainder * K[N_0,n]

    U =  LA.inv(Lmbda - Qstar) * Lmbda # Heffes and Lucantoni
    # B_n defined @ Blondia pg 283
    B = np.empty(qsize, dtype=object)
    for n in range(qsize):
        B[n] = U*A[n]

    # Construct Q matrix
    Q = np.matrix(np.zeros((2*qsize, 2*qsize)))

    for j in range(qsize):
        s, e = j*2, (j+1)*2
        Q[0:2,s:e] = B[j] # First row is B-values
        for i in range(qsize-j-1):
            i_s,i_e = (i+1)*2,(i+2)*2
            j_s,j_e = (j+i)*2,(j+i+1)*2
            Q[i_s:i_e,j_s:j_e] = A[j] # Diagonal of A_j values

    # Fill last column of A's
    for i in range(qsize):
        s,e = i*2, (i+1)*2
        Q[s:e,(qsize-1)*2:] = A[qsize-i:].sum()

    # We want the left eigenvector (Blondia pg 282)
    w,v = LA.eig(Q.transpose())


    # Find eigenvalue closest to 1 with no imaginary part
    diff = 100.0
    idx = -1
    for i in range(len(w)):
        if abs(w[i]-1.0) < diff and w[i].imag == 0:
            idx = i
            diff = abs(w[i]-1.0)
    assert idx > -1

    # We've made sure the eigenvalue has no imaginary part, should be safe
    # to convert to float represenation
    pivals = v[:,idx].astype('float')

    # Impose normalisation constraint on eigenvector
    pivals /= pivals.sum()

    # Each queue length is really two entries in the eigenvector, representing
    # each of the model phases. We're only interested in the total probabilities
    # for each queue length, so sum the values for both phases
    pi = np.zeros(qsize)
    for i in range(qsize):
        pi[i] = pivals[i*2:(i+1)*2].sum()
    return pi

def calc_wait(l,m,K):
    r = (8.0,10.0)
    pi = calc_probs(((l-200)/1000, l/1000),m/1000,K,r)
    mu=1/(m/1000)
    wait = 0.0
    for k in range(1,K-1):
```

```
            wait += ((k-1)*mu - mu/2)*pi[k]
        return wait/1000

if __name__ == "__main__":
    from mpl_setup import plt
    linestyles = ["-", "--", "-.", ":"]

    m = 2.500
    N = 100
    r = (8.0,10.0)


    for l in (2.3,2.8),(2.400,2.600),(2.800,3.000):
        probs = calc_probs(l,m,N,r)
        plt.plot(range(len(probs)), probs, label="$\lambda={:s}$".format(str(l)),
                 linestyle=linestyles.pop(0))

    plt.legend(loc="upper left")
    plt.gca().set_xlabel("Queue length")
    plt.gca().set_ylabel("Probability")
    plt.tight_layout()
    plt.savefig("predicted-mmpp-d-1-n.pdf")
```

8.7 Results analysis and graphing tools

These tools perform the data parsing, analysis and graphing of the data as exhibited in section 5.

8.7.1 Queue length measurements

```

import json, gzip, sys, os, re, math
import matplotlib
from matplotlib import pyplot as plt
from matplotlib import rc
from itertools import repeat
from math import sqrt
from scipy import stats
import numpy as np

import m_d_1_n, m_m_1_n, mmpp_d_n_k2

kmax = 101
mu = 2500
fformat = "pdf"
flows = None
normalise = True
upper_bound = 0.3

rc('font', family='serif', serif="Gentium Basic")
histstyle = dict(color="#999999", linewidth=2, edgecolor="#999999",)

if len(sys.argv) < 2 or not os.path.exists(sys.argv[1]):
    print("Usage: %s <filename>" % sys.argv[0])

filename = sys.argv[1]

linestyles = ["-", "--", "-.", ":"]
linecolors = ["blue", "green", "red", "orange"]

match = re.search("([0-9]+)pps", filename)
if match:
    pps = int(match.group(1))
    probvals = [
        ("M/M/1/K",  m_m_1_n.calc_probs(pps, mu, kmax)),
        ("M/D/1/K",  m_d_1_n.calc_probs(pps, mu, kmax)),
    ]
else:
    match = re.search("([0-9]+)flow", filename)
    probvals = [
        ("MMPP/D/1/K", mmpp_d_n_k2.calc_probs((2.600,2.800), mu/1000, kmax+1, (8,10)))
```

```
        ]
    if match:
        flows = int(match.group(1))
        upper_bound = 0.13

if filename.endswith(".gz"):
    fp = gzip.open(filename, "rt")
    outfile = filename.replace("json.gz", fformat)
else:
    fp = open(filename)
    outfile = filename.replace("json", fformat)
data = json.load(fp)
keys = data[0].keys()


def get_vars(pvals, n):
    vals = []
    var_min = []
    var_max = []
    for p in pvals:
        p = max(0,p)
        if normalise:
            v = p
            var = p*(1-p)/n
        else:
            v = n*p
            var = n*p*(1-p)
        var = sqrt(var)
        vals.append(v)
        var_min.append(max(0,v-var*2))
        var_max.append(max(0,v+var*2))
    return vals,var_min,var_max

def codel(data):
    qlens = np.asarray([i['qlen'] for i in data[200:-200]])
    n = len(qlens)

    fig = plt.figure()
    ax1 = fig.add_subplot('111')
    xvals = range(kmax+1)
    bins = range(kmax+2)
    ret = ax1.hist(qlens, bins=bins, normed=normalise, **histstyle)

    for name,pvals in probvals:

        vals, var_min, var_max = get_vars(pvals, n)

        hist,bin_edges = np.histogram(qlens, bins=bins)
        chi2, pval = stats.chisquare(hist[2:], pvals[2:]*n)

        ax1.plot(xvals, vals, label="%s; p=%.2f" % (name, pval),
                 linestyle=linestyles.pop(0),
                 color=linecolors.pop(0),
            )

        ax1.fill_between(xvals, var_min, var_max, alpha=0.3, zorder=2)#, color="palegreen")
        ax1.set_ybound(lower=0, upper=upper_bound)
        ax1.set_xbound(upper=100)

    ax1.legend(prop=dict(family=['monospace']))
    ax1.set_xlabel("Queue length")
    ax1.set_ylabel("Probability")
    ax1.tick_params(bottom='off', top='off', left='off', right='off')
    plt.savefig(outfile)

def fq_codel(data):
    xvals = range(kmax+1)
    if pps > 2500:
        bins = range(50,151)
        ybounds = (0,0.2)
        xbounds = (0,130)
    else:
        bins = range(201)
```

```
            ybounds = (0,0.2)
            xbounds = (0,120)

        flow_ids = set()
        for d in data:
            if 'flows' in d:
                flow_ids.update([i['id'] for i in d['flows']])
        flows = dict(zip(flow_ids, repeat([])))
        for d in data[100:]:
            keys = list(flows.keys())
            if 'flows' in d:
                for f in d['flows']:
                    flows[f['id']].append(f['qlen'])
                    keys.remove(f['id'])
            for k in keys:
                flows[k].append(0)
        length = len(flows)
        pvals = m_m_1_n.calc_probs(pps/length, mu/length, kmax)
        values = list(flows.values())
        fig = plt.figure()
        for i in range(4):
            ax = fig.add_subplot(2, 2, i+1)
            n,bins,patches = ax.hist(values[i], normed=True, bins=bins,
                                     **histstyle)
            vals,var_min,var_max = get_vars(pvals,len(values[i]))
            ax.plot(xvals, vals, label="Predicted", linewidth=1.2, color="blue")

            if i < 2:
                ax.set_xticks([])
            else:
                ax.set_xlabel("Queue length")
                for t in ax.get_xticklabels():
                    t.set_size(9)
            if i in (1,3):
                ax.set_yticks([])
            else:
                ax.set_ylabel("Probability")
                for t in ax.get_yticklabels():
                    t.set_size(10)
            ax.set_xbound(*xbounds)
            ax.set_ybound(*ybounds)
        plt.tight_layout()
        plt.savefig(outfile)

    if 'ldelay' in keys:
        codel(data)
    else:
        fq_codel(data)
```

8.7.2 Wait time measurements

```
    from matplotlib import rc
    import gzip, json, sys, re
    import numpy as np
    from scipy import stats

    import m_m_1_n, m_d_1_n, mmpp_d_n_k2

    def parse_wait(filename):
        if filename.endswith(".gz"):
            fp = gzip.open(filename, "rt")
        else:
            fp = open(filename)
        data = json.load(fp)
        keys = data[0].keys()
        xval = 0

        match = re.search("qlen([0-9]+)", filename)
        if match:
            xval = int(match.group(1))

        if not 'qlen' in keys:
```

```
        raise RuntimeError("Missing qlen in file %s" % filename)

    waitvals = []
    small_count = 0
    small_threshold = 1
    count_threshold = 1

    averages = []

    # The data file contains the results of many measurements, separated by
    # idle time. Divide the sample into the individual measurements by splitting
    # it at the idle periods (with empty or nearly empty queue), and compute the
    # sample average and standard deviation of the measured delays.
    for i in range(len(data)):
        ql  = data[i]['qlen']
        wait = data[i]['ldelay']
        if ql <= small_threshold:
            small_count += 1
            if small_count >= count_threshold:
                if len(waitvals) > 20:
                    # Chop off first 20 measurements to get to steady-state
                    averages.append(sum(waitvals[20:])/len(waitvals[20:]))
                waitvals = []
                small_count = 0
        else:
            waitvals.append(wait)
    if len(averages) > 50:
        # Too many samples, discard the lowest ones
        averages = sorted(averages)[-50:]

    # CoDel measures delay in microseconds, we need seconds
    averages = np.asarray(averages)/10**6
    return xval, averages.mean(), np.sqrt(averages.var(ddof=1)/len(averages))



if __name__ == "__main__":
    measures = []
    if len(sys.argv) > 1:
        for f in sys.argv[1:]:
            measures.append(parse_wait(f))
    rc('font', family='serif', serif="Gentium Basic")

    fig = plt.figure()
    ax = fig.add_subplot(111)
    linestyles = ["-", "--", "-.", ":"]

    mu = 2500
    lmbda = 3000
    xvals = range(2,120,5)
    funcs = [(m_m_1_n.calc_wait, "M/M/1/K"),
             (m_d_1_n.calc_wait, "M/D/1/K"),
             (mmpp_d_n_k2.calc_wait, "MMPP/D/1/K")]

    for f in funcs:
        ax.plot(xvals, list(map(lambda k: f[0](lmbda, mu, k), xvals)),
            label=f[1], linestyle=linestyles.pop(0))

    if measures:
        for m in measures:
            ax.errorbar(m[0], m[1], yerr=m[2]*2)
        ax.plot([m[0] for m in measures], [m[1] for m in measures], "o")

    ax.legend(loc='upper left')
    ax.set_xlabel("Queue length")
    ax.set_ylabel("Wait time (seconds)")
    ax.set_ybound(upper=0.045)
    plt.savefig("wait-times.pdf")
```