Performance modeling of virtual switching systems

Guillaume Gallardo, Bruno Baynat, Thomas Begin

Performance modeling of virtual switching systems

Guillaume Artero Gallardo
ENS Lyon, Université Lyon 1,
Inria, CNRS, UMR 5668 -
Lyon, France
guillaume.artero@gmail.com

Bruno Baynat
Sorbonne Université
UPMC Univ Paris 06
CNRS, LIP6 UMR 7606
Paris - France
bruno.baynat@lip6.fr

Thomas Begin
Université Lyon 1, ENS Lyon,
Inria, CNRS, UMR 5668 -
Lyon, France
DIVA Lab, University of Ottawa
Ottawa, Canada
thomas.begin@univ-lyon1.fr

Abstract—**Virtual switches are a key elements within the new paradigms of Software Defined Networking (SDN) and Network Function Virtualization (NFV). Unlike proprietary networking appliances, virtual switches come with a high level of flexibility in the management of their physical resources such as the number of CPU cores, their allocation to the switching function, and the capacities of the RX queues, which gives the opportunity for an efficient sizing of the system resources. We propose a model for the performance evaluation of a virtual switch. Our model resorts to servers with vacation to capture the involved interactions between queues resulting from the implemented polling strategies. The solution to the model is found using a simple fixed-point iteration and it provides estimates for customary performance metrics such as the attained throughput, the packet latency, the buffer occupancy and the packet loss rate. In the tens of explored examples, the predictions of the model were found to be accurate, thereby allowing their use for the purpose of sizing problems.**

I. Introduction

Computer networks are expected to undergo major changes with the development of Software Defined Networking (SDN) and Network Function Virtualization (NFV). SDN introduces a new architecture in which a single or a small set of centralized controller(s) replaces the control plane, formerly distributed on each router in traditional IP networks. As for NFV, it enables common network functions (e.g., packet forwarding, firewall, caching) to be done via software on industry-standard hardware (e.g., x86 architecture). The expectations generated by combining SDN and NFV together are very high. Operators expect considerable gain in resource management flexibility (e.g., by dynamically re-programming their network), while reducing their cost of operations by replacing dedicated and proprietary appliances by standard off the shelf hardware.

Within SDN and NFV-based networks, nodes interconnecting the links are standard physical servers, with multiple CPU cores, running a virtual switch or virtual switch implementation. Analogously to routers in traditional IP networks, virtual switches are in charge of receiving, processing, switching, and transmitting packets flowing in the network. They handle incoming packets using a specialized library that largely defines their internal architecture. The DPDK library [1], which enables the processing of a packet in less than 80 CPU cycles, is becoming a prominent framework. Despite considerable progress, virtual switches are unlikely to be as fast their full hardware-based counterparts, and may be viewed as a critical spot when studying the performance of SDN/NFV networks.

However, unlike proprietary networking appliances, virtual switches come with a high level of flexibility in the management of their physical resources. Typically, the hypervisor running on the physical server oversees the sharing of the available resources among the different software hosted on this machine. For example, if a virtual switch faces an excessive workload, the hypervisor can allocate additional CPU cores to mitigate the effects of contention. Conversely, in case a virtual switch has low utilization of its allocated resource, the hypervisor can decide to de-allocate a fraction of its CPU cores. The de-allocated CPU cores may be turned off, thereby reducing the server power consumption, or provisioned to another task executed on the same physical server. Although these strategies appear appealing, their implementations require a method to determine on the fly the amount of resources to (de-)allocate. From the practical standpoint, such methods have to be computationally efficient, and ideally, with little or virtually no effect on the virtual switch performance. A fast analytical approach appears as a natural candidate to meet these constraints.

In this paper, we propose an accurate analytical queueing model to evaluate the performance of a virtual switch with several network interface cards (NIC) and several CPU cores. To circumvent the combinatorial growth of the state space and avoid dealing with multi-dimensional Markov chains, the proposed approach decouples the polling system associated with each CPU into several queues, and resorts to servers with vacation to capture the interactions between queues. The vacation of the server for a given queue represents the processing time devoted by the CPU to other queues. The model is solved using a simple fixed-point iteration, and provides estimates for customary performance metrics such as the attained throughput, the packet latency, the buffer occupancy and the packet loss rate.

The remainder of the paper is organized as follows. Section II discusses the context of virtual switching systems and the related state of the art. In Section III, we describe the internal architecture of a virtual switch. Section IV develops the proposed corresponding model and its analytical solution. In Section V, we explore numerical results to illustrate the accuracy and potential applications of the proposed model. Finally, Section VI concludes this paper.

II. Context and Related Work

a) Virtual switching: Virtual switches, implemented at the hypervisor stages, are mostly in charge of relaying data packets between NICs, or more specifically, between ports that may be either physical or virtual ports. There exist multiple strategies for mapping virtual ports to input ports: bridging, port aggregation, etc [2]. Their common objective is to provide the virtual switch with the capability of interconnecting a large number of virtual machines (VM) deployed on the same physical entity. For instance, in cloud computing, data packets may cross multiple VM located on the same machine before being routed outside the platform. The related forwarding rules can be programmed via standardized API such as OpenFlow [3]. One of the most popular open source implementation of a virtual multilayer switch supporting the OpenFlow protocol is Open vSwitch (OvS) [4]. It can be run on top of commodity hardware and used either in virtual or physical environments. This led several software switching solutions to compete on the market by proposing architectural schemes that improve the OvS switching performance [5]. Most of them use an optimized data-path library, such as the Intel's DPDK [1] and Netmap [6], that accelerates packet processing in the userspace. In particular, these schemes use a polling scheme to minimize scheduling and interrupt latency (see the Poll Mode Driver for DPDK). Note that, overall, the achievable performance highly depends on the way memory is handled within the switch [7]. OvS developpers proposed optimized caching techniques [4]. As a result, recent solutions such as CuckooSwitch, based on DPDK, followed the trend and proposed novel hashing and batching techniques to improve the packet throughput [8].

Several works have attempted to characterize the Open vSwitch performance from a network perspective. Most of them conducted measurements on an experimental testbed to demonstrate the enhancement provided by DPDK [7]. They also measured the impact of the number of NIC, the offered load and the packet size [8]. Other works investigated the impact of active flow monitoring [9], that might simply consists in sampling packets being forwarded across the virtual switch. However, increasing the sampling rate to gain accuracy consumes CPU resources, and in turn degrades the overall performance.

By nature, modeling approaches are non intrusive, and represent a nice alternative to evaluate the performance of virtual switches. In particular, a model for estimating the packet loss probability and the average sojourn time of OpenFlow architectures is provided in [10]. This model assumes that all the packets arrive at the same queue before being forwarded accross the switch. As stated by the authors, it cannot capture the effect of more sophisticated processing strategies such as polling. As polling highly contributes to packet switching acceleration, it requires a particular attention for accurately modeling the virtual switch performance.

b) Polling models: Strategies of polling have been extensively used in computer networks and telecommunication systems. For instance, the IEEE 802.5 Token Ring introduced in the early 1980's, used this scheme in its medium access method. Nonetheless, the analysis of polling systems started even before in the late 1950's with the patrolling repairman model for the British cotton industry. Reference surveys on polling models were published in the early 1990's by Takagi to provide a classification of polling systems and related research advances [11], [12]. These studies underscore that the performance of a polling system depend, in general, on many factors including the number and the capacity of queues, the arrival and service rates, as well as the switch-over time (the time spent by the server to switch from one queue to the following one).

Polling systems can be classified according to service policies, that might be exhaustive or gated, and unlimited or M-limited. **Exhaustive:** Once the server polls a given queue, it serves the queue until its complete exhaustion, and then it switches to the next queue. This implies that any request arriving in a queue while the server is currently processing another request of the same queue, will be served before the server moves to the next queue. **Gated:** Unlike the exhaustive policy, the server does not process (in the current round) requests that may enter the queue while the server is already serving this same queue. Additionally, for both aforementioned policies, one can set an upper limit on the number of requests that the server can process for a same queue before switching to the next one. **M-Limited:** On each turn, the server can serve at most M requests for each queue. As discussed hereafter in Section III, the polling scheme implemented in a virtual switch corresponds to a gated M-limited.

Unfortunately, the general solution to polling systems is not known. However their analysis is no less important, and therefore, several approximations have been developed. Tran-Gia proposed an analytical framework for computing the performance of a gated 1-limited polling system with non-zero switch-over time [13]. The modeling approach consists of solving a fixed-point problem to evaluate the state probabilities of an embedded Markov chain. In particular, the analysis of each involved queue is carried out at polling instants, i.e., ends of vacation. It requires the computation of Laplace-Stieltjes transforms as well as the use of Laplace inversion procedures or two-moment approximation techniques. As stated by the authors, such model is accurate only for large switch-over times and small values of the queue capacities (less than 10 requests). The fixed-point approach developed by Tran-Gia has then been extended to the case of exhaustive M-limited systems in [14]. In this work, the authors leverage the techniques provided by Lee to study *M/G/1/K* queues with server vacation [15], [16]. It consists in decomposing the polling system in individual *M/G/1/K* queues with server vacation. Each queue is then studied at polling instants. To reduce the amount of modeling assumptions introduced in the previous works, a more general framework is presented in [17]. When conducting the analysis of each queue, it eliminates the hypothesis that the busy period, i.e., the time the server is not in vacation, and that the vacation time are independent. This approach relies on solving a system of several numerical equations. However, as stated by the author, the complexity of the involved expressions may require to use a symbolic computation software. Broadly speaking, most of these approaches address a different policy than that implemented in virtual switches, and/or they involve complex arithmetic operations such as Laplace-Stieltjes transforms that may not scale with the number of queues or with their capacity. In this context, a different modeling strategy based on continuous time Markov chains has been proposed in [18]. The analysis of each queue is not conducted at particular time instants anymore, and relies on a detailed decomposition of the server vacation. However, solving each Markov chain cannot be done straightforwardly and requires using numerical methods such as successive over-relaxation (SOR) or Gauss-Seidel.

III. System Description

c) Switch architecture: A virtual switch comprises several NICs (network interface cards) that altogether provide a total of N I/O ports. It also contains a set of C CPU cores (physical or logical) that are in charge of processing the packets coming from the different I/O ports. Figure 1 illustrates this internal architecture. The use of modern NIC enables each port to perform load balancing by spreading its incoming trafic load among all the CPU cores. Incoming packets from a single port are dispatched to C logical queues, one per core, also referred to as RX queues. Each core is thus assigned to N independant RX queues, one per port, that are processed in a polling fashion. The way the packets are routed results from a specific hashing function, such as the Receive Side Scaling (RSS) used in DPDK. This technique aims at accelerating packet processing by directing packets belonging to the same flow to the same RX queue. In addition, to avoid synchronization issues, a given RX queue cannot be processed by multiple CPU cores. When a CPU core polls a specific RX queue, it only processes the first-in-line packet before polling the following queues. Packets are thus served in a round robin fashion. Optionally, the virtual switch can be set in a mode in which it processes a batch of packets from a RX queue before moving to the next RX queue. By doing so, the number of some time-consuming system calls is expected to be reduced. When the batch size is set to M, the core can prefetch a maximum of M packets in one RX queue before processing them in a run-to-completion manner and polling the next RX queue. Note that the core must know in advance the number of packets to poll before starting its fetching. Hence, as detailed in Section II, such a polling strategy corresponds to a gated M-limited policy. Because the RX queues are logical entities, assigning the core to a new RX queue is a straightforward operation. This simply requires to update the memory address of a pointer, and thus the switch-over times are negligible.

Processing a packet at least consists of reading its headers and extracting the destination address before transferring it to the adequate output port. The processing thread can additionally apply some flow-specific operations such as deep packet inspection (DPI), encryption, or QoS monitoring, taken as examples. While all the cores may have the same clock frequency, packets directed to different RX queues can thus present peculiar flow characteristics that impact their processing time. Once a packet is processed by a CPU core, it is (logically) forwarded from its RX queue to a logical TX queue associated with the appropriate destination output port. At this stage, the packet is pending for transmission on the next hop and does not hold any core processing resource anymore. The remainder in processing a packet is performed by another independant process. To achieve zero loss performance, transmission resources and TX queue sizes are usually over-sized. Thus, the bottleneck of a virtual switch is likely to occur during the processing of packets in RX queues due to the limited CPU resources[1]. If the system is overloaded, there is a chance of achieving 100% CPU core utilization. As a consequence the CPU cores cannot process the packets as fast as required, eventually leading to buffer overflows and packet rejections at the input ports. Predicting such behavior can be of great interest for calibrating the parameters of the switch, e.g., the total number C of CPU cores allocated to the switch. We thus concentrate our efforts on evaluating the switch performance at RX queue level.

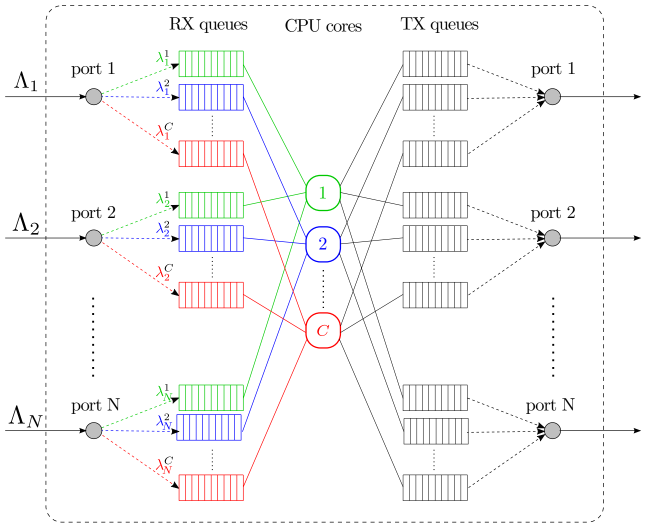


Fig. 1. Internal architecture of a virtual switch with N I/O ports and C CPU cores.

d) System decomposition: As illustrated in Figure 1, the general switch architecture can be decomposed in C independant subsystems, each of them composed of one CPU core polling N independant RX queues. Every subsystem is identified with a distinct color in Figure 1. As a consequence these subsystems can be studied independantly from each others. For the sake of clarity, we now refer to such a subsytem simply as a polling system. A given polling system j, associated with CPU core j, $j = 1, ..., C$, is characterized by the capacity of each of its assigned RX queues, denoted by K, the mean arrival rate of packets at RX queue i, denoted by λ_i^j, as well as the average processing time of packets stored in RX queue i, denoted by $1/\mu_i^j$. The input parameters λ_i^j and μ_i^j can be statistically estimated by measurements. In particular, the

[1]Given the current transfer rates of SDRAM, the accesses to the memory are much faster than the operating speeds of NICs and CPU cores.

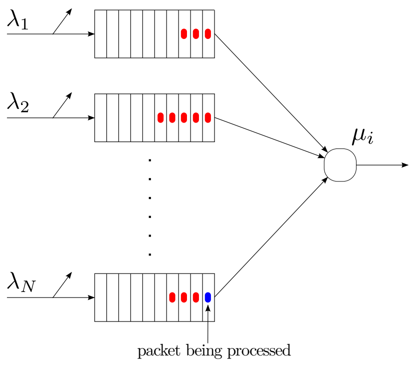


Fig. 2. Polling system associated with a given CPU core.

former can be obtained by using the aggregate packet arrival rate at port i, referred to as Λ_i, plus the statistics of the hashing functions. It follows that: $\Lambda_i = \sum_{j=1}^{C} \lambda_i^j$.

In Section IV, we describe an accurate and scalable modeling framework to derive all performance parameters of such a polling system: the j-th CPU core utilization rate U^j, the loss rate b_i^j, the average sojourn time of a packet $\bar{r}_i^j$ and the buffer occupancy $\bar{q}_i^j$, relative to the i-th RX queue $(i = 1, ..., N)$ attached to the j-th CPU core $(j = 1, ..., C)$.

e) Global system performance: Finally, characterizing the global performance of the virtual switch consists in aggregating the individual performance of every sub-system (each representing the polling of a CPU core accross N queues). For instance, the global CPU core utilization rate, denoted by U, is simply the arithmetic mean of all core utilizations:

$$U = \frac{\sum_{j=1}^{C} U^j}{C} \tag{1}$$

Similarly, system metrics relative to a given port can be computed by averaging the performance results of its attached RX queues. The packet rejection probability at port i, referred to as B_i, can be computed as:

$$B_i = \frac{\sum_{j=1}^{C} b_i^j \lambda_i^j}{\sum_{j=1}^{C} \lambda_i^j} \tag{2}$$

Note that the global CPU utilization and blocking probability are performance parameters that capture and summarize the overall level of congestion in a virtual switch, and therefore represent metrics of direct interest for network operators.

IV. MODELING FRAMEWORK

As discussed above, throughout this section we only consider the model associated with a given CPU core j and its N associated RX queues. Therefore, for the sake of clarity, we drop superscript j in all subsequent notations and equations. Figure 2 represents the polling system associated with the considered CPU core.

A. 1-Limited scenario

We first consider the case of a 1-Limited polling system. Under such configuration, the considered CPU core processes, in a round-robin fashion, a single packet from each of its non empty input RX queue.

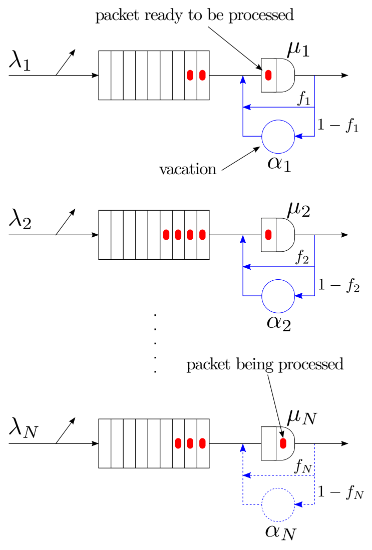


Fig. 3. Decomposition of a sub-system into N queues.

1) Principle of the decomposition: The idea of the model is to decompose the original polling system into a set of independent queueing models with server vacations. This decomposition is illustrated in Figure 3. The buffer of queue i in the decomposed model represents the i-th RX queue associated with the j-th considered CPU core (whose superscript j has been dropped). The server of queue i in the decomposed model aims at reproducing the way packets of RX queue i are processed by the CPU core. Because the core performs a polling on each of its input RX queue, between the processing of two successive packets of a given queue i of the model, there is an in-between time that corresponds to the processing of one packet of all other non empty queues. In the model, this time will be referred to as a *vacation* time. As an illustration, in Figure 3, the server of queue N is in process, meaning that the CPU core is currently processing a packet in RX queue N, and all other queues are in vacation. In this particular example, when queue N ends its processing, it goes in vacation, the first in line packet of queue N is put on a hold, and at the same time queue 1 ends its vacation and start the processing of its first in line packet. In the general case, when a given queue i ends its processing, and when at that precise time all other queues are empty, it skips vacation and immediately start the processing of the next-in-line packet (if one). In our model, we represent the likelihood of this event by a probability f_i. With a probability $1 - f_i$, when queue i ends its processing, it thus goes in vacation and stays unavailable for an average time $1/\alpha_i$ (see Figure 3).

2) Markov chain model associated with each RX queue: In order to derive a simple decomposed model, we make the following Markovian assumptions. First, we assume that the arrival of packets at the entrance of queue i follows a Poisson process of rate λ_i. Then, we assume that the processing time of one packet from queue i is exponentially distributed with rate μ_i. Finally, we assume that the vacation time of queue i is exponentially distributed with rate α_i. The robustness of these Markovian assumptions are evaluated in the next section.

Under these assumptions we can associate with each queue i of the decomposed model, the continuous-time Markov chain depicted in Figure 4. A state (k, P) of this chain, $k = 1..., K$, corresponds to queue i with currently k packets and the first-in-line packet being processed (P), i.e., the CPU core is assigned to RX queue i. A state (k, R) of this chain, $k = 1..., K$, also corresponds to queue i with k packets, but with the first-in-line packet that is not currently in process and that is ready (R) to be processed, i.e., the CPU core is assigned to another RX queue. State $(0, E)$ corresponds to an empty (E) queue i but not a full empty system, i.e., the CPU core is currently assigned to another non empty RX queue. Finally state $(0, F)$ corresponds to a full (F) empty system, meaning a system where all queues are empty and the CPU core is idle. With this description, lower green and red states of the chain ($(0, E)$ and (k, R)) correspond to a queue with a server in vacation, blue states (k, P) to a queue with a processing service, and the black state $(0, F)$ to an empty system.

From any state of this chain (except (K, P) and (K, R)) we can reach the state immediately on the right with some rate λ_i corresponding to the arrival of a new packet in queue i. We can exit a state (k, P), $k = 1..., K$, after a processing time of rate μ_i. At this instant, with some probability f_i, all other queues $j \neq i$ are empty and the CPU core is instantaneously reassigned to queue i, i.e., queue i does not go on vacation. In this case the Markovian process switches to state $(k - 1, P)$ ($(0, F)$ if $k = 1$). With a probability $1 - f_i$, at least one of the other queues is not empty and the server of queue i goes in vacation. In this case the Markovian process switches to state $(k-1, R)$ ($(0, E)$ if $k = 1$). From any state (k, R), $k = 1..., K$, we can reach state (k, P) with some rate α_i corresponding to the end of vacation of queue i. From state $(0, E)$, we can reach state $(0, F)$ with a rate γ_i that has to be carefully characterized. Indeed, this transition corresponds to a system becoming fully empty, starting from a state where queue i is empty (with no more information about other queues except that they are not all empty). Finally, starting from state $(0, F)$ corresponding to a full empty system, we can reach state $(1, P)$ if the first arriving customer enters queue i, i.e., with a rate λ_i, or we can can reach state $(0, E)$ if the first arriving customer enters another queue $j \neq i$, i.e., with a rate $\sum_{j \neq i} \lambda_j$.

As a conclusion of this subsection, three parameters are left to be estimated in order to fully characterize the Markov chain associated with queue i, namely probability f_i and rates α_i and γ_i. Subsection IV-A4 will provide estimates of this missing parameters. Assuming these parameters are known, we explain in Appendix 1 how to directly and efficiently derive the stationary probabilities of the Markov chain.

3) Performance parameters: Conditioned by the fact that all Markov chains (associated with all queues $i = 1, ..., N$) have been solved in stationary regime, we can derive the steady-state performance parameters of the system as follows. First, we can easily obtain the average number of packets in queue i from the stationary probabilities of the chain:

$$\bar{q_i} = \sum_{k=1}^{K} k \times (\pi_i(k, P) + \pi_i(k, R)), \tag{3}$$

as well as the loss rate at the entrance of queue i:

$$b_i = \pi_i(K, P) + \pi_i(K, R). \tag{4}$$

The average sojourn time of an accepted packet in queue i is obtained using Little's law:

$$\bar{r_i} = \frac{\bar{q_i}}{\lambda_i(1 - b_i)}. \tag{5}$$

In order to calculate the CPU core utilization, we first define the global input rate and the global output rate:

$$\lambda_{in} = \sum_{i=1}^{N} \lambda_i, \tag{6}$$

$$\lambda_{out} = \sum_{i=1}^{N} \lambda_i(1 - b_i). \tag{7}$$

We can derive from these two quantities the proportion of packets of queue i that are processed by unit of time:

$$p_i = \frac{\lambda_i(1 - b_i)}{\lambda_{out}}, \tag{8}$$

and then calculate the CPU core utilization:

$$U = \left(\sum_{i=1}^{N} \frac{p_i}{\mu_i} \right) \lambda_{out} = \sum_{i=1}^{N} \frac{\lambda_i(1 - b_i)}{\mu_i}. \tag{9}$$

4) Estimating the Markov chain parameters: Conditioned by the fact that all performance parameters are known, we can estimate the missing probabilities and transition rates of Markov chains associated with all queues. In the following we give the expressions of probability f_i and rates α_i and γ_i, associated with a given queue i.

First, we can estimate f_i as the probability that all other queues ($j \neq i$) are empty, knowing that a packet of queue i is being processed:

$$f_i = \prod_{j \neq i} \mathcal{E}_j, \tag{10}$$

where $\mathcal{E}_j$ is the probability to be in state $(0, E)$ of Markov chain j, knowing that it can neither be in state $(0, F)$ (because the system is not totally empty) nor in any state (k, P) (because the CPU core is currently processing a packet of queue i):

$$\mathcal{E}_j = \frac{\pi_j(0, E)}{\pi_j(0, E) + \sum_{k=1}^{K} \pi_j(k, R)}. \tag{11}$$

Instead of considering α_i, we estimate $1/\alpha_i$, the mean vacation time conditioned by the fact that queue i goes on vacation (i.e., that vacation of queue i is not null). Because the mean vacation time of queue i corresponds to the mean processing time of one packet of each non empty queue $j \neq i$, we have:

$$\frac{1}{\alpha_i} = \sum_{j \neq i} \frac{1}{\mu_j} \times \frac{1 - \mathcal{E}_j}{1 - f_i} = \frac{1}{1 - f_i} \sum_{j \neq i} \frac{1}{\mu_j}(1 - \mathcal{E}_j). \tag{12}$$

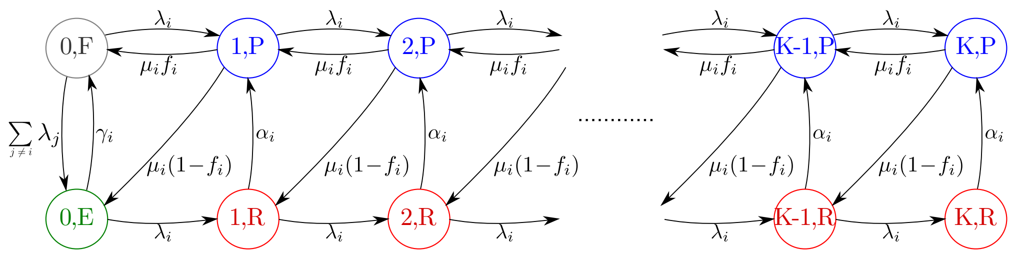


Fig. 4. Continuous-Time Markov Chain associated with queue i.

Finally, in order to estimate γ_i, namely the transition rate between state $(0, E)$ and state $(0, F)$ of Markov chain i, we introduce in the resolution a set of new equations which state that the stationary probability of state $(0, F)$ of any Markov chain must correspond to the probability that the CPU core is idle. We thus add the following equations: $\pi_i(0, F) = 1 - U$ for all $i = 1, ..., N$, and by doing that, we impose that the stationary probabilities of states $(0, F)$ of all Markov chains are equal. If we use these equations together with the stationary equation associated with state $(0, F)$ of Markov chain i, we obtain:

$$\gamma_i = \frac{\lambda_{in}(1-U) - \mu_i f_i \pi_i(1, P)}{\pi_i(0, E)}. \tag{13}$$

5) Fixed-point solution: The parameters of a Markov chain associated with a given queue i (f_i, γ_i and α_i) depend at the same time on the stationary solution of the other Markov chains (through relations 10 and 12) and on the CPU core utilization U (relation 13). In turn, the CPU core utilization (as well as all performance parameters) depends on the stationary solution of all Markov chains (relation 9). As a result, the resolution of the model relies on a fixed-point iterative technique that is summarized by Algorithm 1. The main loop of the algorithm is repeated until a given convergence criterion is reached, e.g., the maximum relative difference of varying parameters between two successive iterations is very small (e.g., less that 10^{-4}).

Algorithm 1: Fixed-point iterative technique

```
Input  : System parameters $K$, $\mu_i$, $\lambda_i$ for each queue $i$
Output : Stationary probabilities $\pi_i$ and performance metrics for
         each queue $i$
Initialize $\pi_i$, $f_i$, $\alpha_i$ and $\gamma_i$ for each queue $i$;
Initialize $U$;
while convergence criterion not satisfied do
    foreach queue $i \in [\![1, N]\!]$ do
        Compute probability $f_i$ using Eq. 10;
        Compute transition rate $\alpha_i$ using Eq. 12;
        Compute transition rate $\gamma_i$ using Eq. 13;
        Solve the Markov chain associated with queue $i$ and
        compute the stationary probabilities $\pi_i$;
    end
    Compute server utilization rate $U$ using Eq. 9;
end
Compute all performance metrics of interest from Eq. 3 to 9;
```

B. Extension to the M-Limited scenario

A simple way of providing an extension of the previous model to the case of a M-Limited system with $M > 1$, consists in keeping the same Markov chain model associated with a single RX queue (as depicted in Figure 4), and modifying its parameters. Indeed, from any state (k, P), $k = 1..., K$, after a processing time of rate μ_i, we can still either reach state $(k-1, P)$ ($(0, F)$ if $k = 1$), or reach state $(k-1, R)$ ($(0, E)$ if $k = 1$). The first case happens either if the current packet in process is not the last packet of a batch, or if it is the last one but all other queues $j \neq i$ are empty and the CPU core is then instantaneously re-assigned to queue i (like before). We then need to redefine the probability f_i and the vacation rate α_i to take into account these two possibilities. All remaining parameters of the Markov chain model keep the same expression as in the case of a 1-Limited system.

V. Performance Evaluation

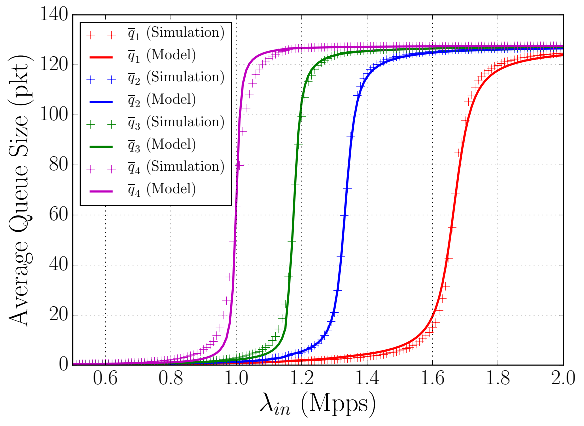
In this section, we explore a number of cases to study the accuracy of our proposed approach for different performance parameters. We also investigate the robustness of the Poisson assumption for the arrival process as well as the use of an exponential distribution for modeling the packet processing time. Then we give an example demonstrating how, analogously to the famous Erlang formulas, our model can deliver dimensioning curves that can be used to manage the resources of a virtual switch.

In the vast majority of explored cases, the convergence of the fixed-point iteration involved in the solution of our model is rapidly found, say less than several tens of iterations, making its execution time small. We use a home-made discrete-event simulator as comparison basis to assess the accuracy of our approach. The simulator reproduces the behavior of a real polling system implementing the Gated 1-Limited policy. Each simulation is run for 5 seconds of simulated time, which typically corresponds to several millions of packet completions, resulting in very small confidence intervals that will not be shown on figures.

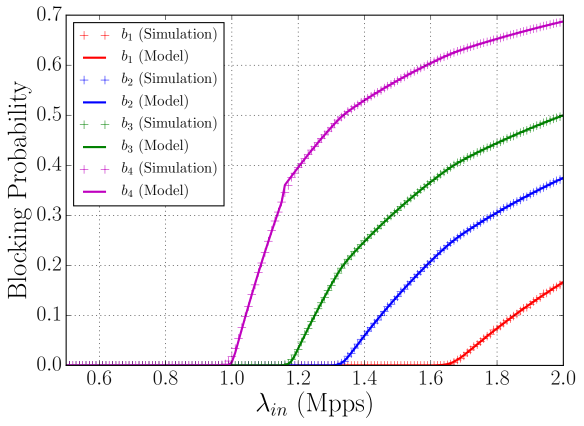
For the sake of clarity, we consider in our examples that the capacities of the RX queues are equally set to $K = 128$, and that the dispatching function, which determines the mapping between the packets arriving on a given port and the CPU cores, is evenly-balanced.

A. Homogeneous CPU cores

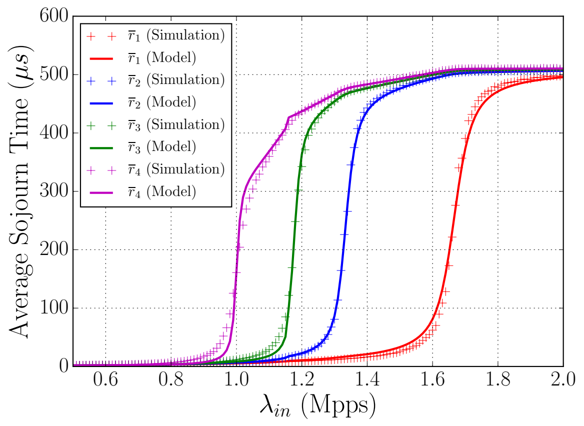
We start with an example in which we study the performance from the perspective of a CPU core. Let us consider a virtual switch comprising 4 input ports served by a set of CPU cores



(a) Average queue size.



(b) Blocking probability.



(c) Average sojourn time.

Fig. 5. Homogeneous CPU cores with $N = 4$ ports receiving respectively 15%, 20%, 25% and 40% of the total traffic.

whose service rates are set to 1 Mpps (million packets per second) each. We consider that the CPU cores are homogeneous and statistically identical. Since we assume that the dispatching function is well-behaved, it follows that every CPU core undergoes the same performance. Hence, we can restrict our analysis to only one of them. The global input rate of packets bounded to the selected CPU core, λ_{in}, is supposed to be unevenly spread among the 4 ports as follows: 15% on port 1, 20% on port 2, 25% on port 3 and 40% on port 4. We let λ_{in} vary from 0.5 Mpps to 2 Mpps in order to explore the whole spectrum of possible loads.

For each level of input rate λ_{in}, we evaluate the average queue size $\bar{q}_i$, the mean sojourn time (of an accepted packet) $\bar{r}_i$, and the blocking probability b_i, associated with each individual port ($i = 1, \ldots, 4$) as given by our model, and we compare them to those delivered by simulation. The corresponding results are shown in Figure 5. The magenta curves pertain to the performance of port 4, which is expected to be the first to saturate since it captures the largest fraction of incoming packets. On the other hand, the red curves are associated to port 1 (the last to saturate), and thus, for a same value of λ_{in}, correspond to a lower level of load.

Figure 5(a) shows the evolution of the average queue size $\bar{q}_i$ as a function of the global input rate λ_{in}. While all these curves are monotonically increasing from 0 to 128 (the capacity of queues), they exhibit highly nonlinear behavior. As expected, port 4 is the first to experience resource contentions as λ_{in} nears 0.9. Conversely, the average queue size for the other ports remains small for larger values of λ_{in}, before rapidly increasing to a value of 128. Looking at the accuracy of the model, we observe that the curves predicted by the model are generally close to those delivered by the simulation. More precisely, it appears that the largest errors occur on the most congested port (port 4) near the tipping points. However, the model accurately captures the saturation threshold of all ports (including port 4), as well as the average queue size for a wide range of loads.

We now discuss Figure 5(b) and the accuracy on the blocking probability b_i. We expect the blocking probability to increase from 0 to 1 as the global input rate λ_{in} increases. Again, because it receives the largest fraction of incoming packets, port 4 (magenta curves) is the first to experience packet losses. Overall, we observe that the model and the simulation coincide over the whole range of considered loads.

In Figure 5(c), we consider the average sojourn time $\bar{r}_i$ spent by packets in each queue. Similarly to the average queue size and the blocking probability, we observe that the mean sojourn time of each port starts to rise at different levels of load λ_{in}. The figure also shows that for any port, the model is able to accurately forecast the value of the average sojourn time over the whole spectrum of loads.

As mentioned in Section IV, the derivation of the model relies on two Markovian assumptions. First, the arrival of packets at a given queue is assumed to follow a Poisson process. Then, the processing time of one packet is supposed to be exponentially distributed. Figures 6 and 7 report the evolution of

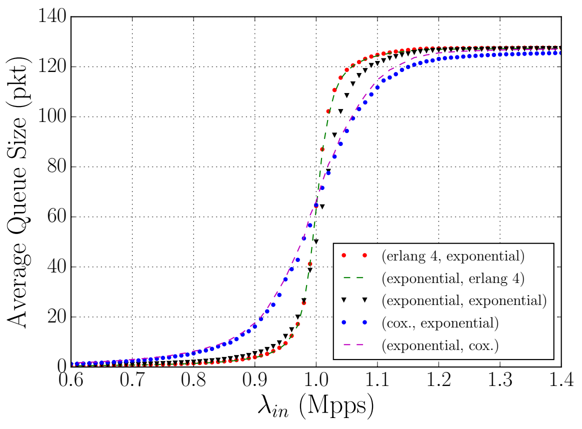


Fig. 6. Homogeneous CPU cores with $N = 4$ ports: Evolution of the average queue size $\overline{q}_4$ when using different distributions of the packet inter-arrival time and packet processing time. Each data sample is associated with a couple of distributions (inter-arrival time distrib., packet processing time distrib.).

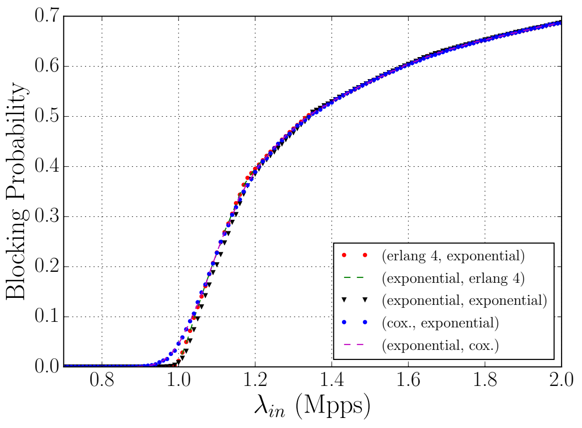


Fig. 7. Homogeneous CPU cores with $N = 4$ ports: Evolution of the blocking probability b_4 when using different distributions of the packet inter-arrival time and packet processing time. Each data sample is associated with a couple of distributions (inter-arrival time distrib., packet processing time distrib.).

the average queue size and the blocking probability at the most congested port, namely port 4, as a function of the global input rate λ_{in} when we release these assumptions. In particular, we run several simulations where we only change the distribution of the packet inter-arrival time (the same set for every queue) as well as the distribution of the one packet processing time. The followed distributions are either the Erlang-4 whose squared coefficient of variation is $1/4$, the exponential or a Coxian-2 with a squared coefficient of variation equal to 5. Figure 6 shows that such distributions affect differently the average queue size near the tipping points. When either the packet inter-arrival time or the packet processing time exhhibit a low variability, the behavior of the average queue size tends to be more steep. On the other hand, when these distributions are replaced by more variable ones, the rise of the average queue size usually occurs at lower levels of loads. Figure 7 then shows

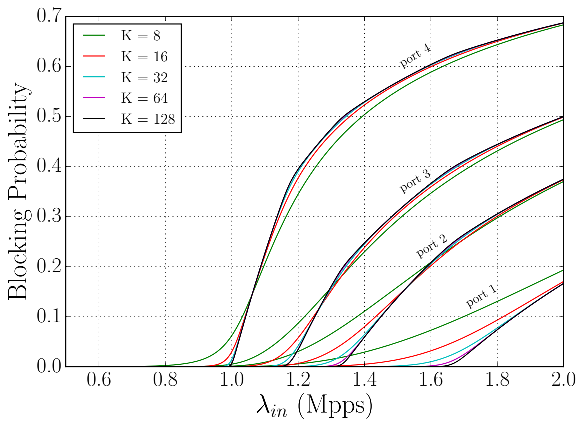


Fig. 8. Homogeneous CPU cores with $N = 4$ ports: Evolution of the blocking probabilities, b_1, b_2, b_3 and b_4, for different values of K.

that the blocking probability is almost insensitive to the selected distribution for the service or inter-arrival times. Indeed, only slight differences appear for loads at around 1 Mpps. Overall, it appears that Markovian assumptions are justified for a wide range of distributions when the size of batches is limited to 1.

Finally, we investigate the influence of the capacity K of RX queues. Figure 8 shows the blocking probability on each port, namely b_1, b_2, b_3 and b_4, as obtained by the model, for queue capacities ranging from $K = 8$ to 128. As can be seen on the figure, from $K = 32$ the capacity of RX queues has very low impact on the blocking probability. The result is also true for other performance parameters, highlighting that this system parameter does not need to be carefully dimensioned as long as it exceeds a few dozens.

B. Heterogeneous CPU cores

In our second example, we release the assumption that all the CPU cores are identical. Instead, we consider a virtual switch with $N = 4$ ports and $C = 8$ CPU cores, each processing at a different speed. The service rate of the j-th core ($j = 1, \ldots, 8$) is set to $1+(j-1)/8$ Mpps. We assume that the packet arrival rates are identical on each port, i.e., $\Lambda_1 = \Lambda_2 = \Lambda_3 = \Lambda_4 = \Lambda_{in}/4$, where Λ_{in} is the total submitted traffic.

In this example, the performance of each CPU core are different since they all differ. Instead of showing detailed per port performance parameters, we chose to give in this subsection global system performance metrics, namely the global CPU utilization rate of the switch U (given by relation 1), and the blocking probability obtained at each input port B_i (relation 2). The global CPU utilization takes values between 0 and 1, and reflects the proportion of used resources among the 8 cores. The blocking probability at port i represents the fraction of packets being dropped due to a lack of buffer space in any of the 8 subsequent RX queues. Note that in our example, since all Λ_i are equal, so do the B_i.

Figure 9 reports the evolution of the global CPU utilization U and the blocking probability at port i, B_i, as a function of the

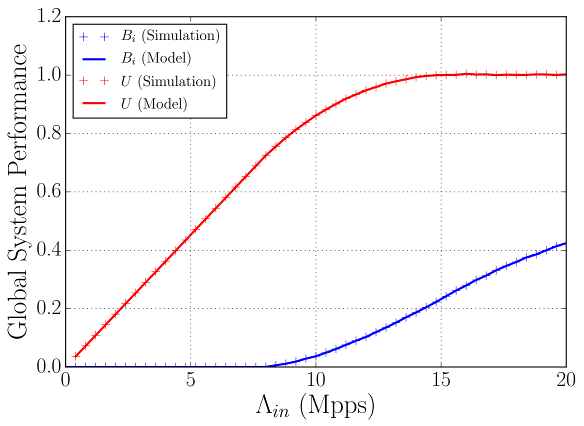


Fig. 9. Heterogeneous CPU cores with $C = 8$ cores and $N = 4$ ports: Evolution of the global CPU utilization U and the blocking probability at port i, B_i.

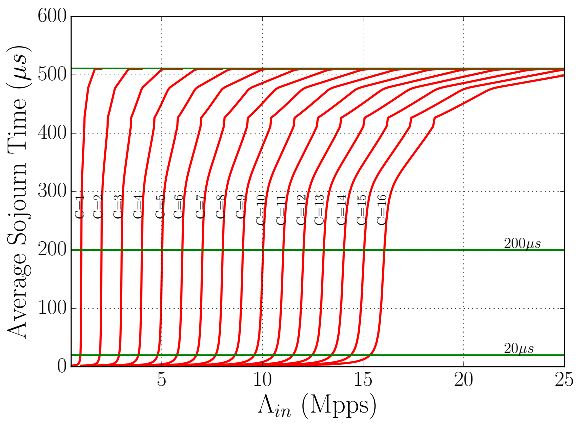


Fig. 10. Sizing the number of CPU cores in a virtual switch according the predicted average sojourn time.

total submitted traffic Λ_{in}. First, we observe that U gradually increases with increasing values of Λ_{in} until it reaches its maximum value of 1. As for B_i, packet losses are virtually null for levels of load Λ_{in} less than 8 Mpps. Then, losses become more frequent, and near 20% when Λ_{in} comes close to 15 Mpps. As shown in the Figure, the performance predicted by our model fully coincide with those delivered by simulation.

C. Sizing the number of CPU cores

To explore the field of possible application of our model, we study the problem of determining the right amount of CPU cores in a virtual switch so as to meet a given QoS criterion. We illustrate it by choosing for the QoS criterion, a maximum tolerable delay for packet switching. For the sake of saving space, we reconsider the virtual switch described in Section V-A, but we let the number of CPU cores C unspecified. For values of C ranging from 1 to 16, we run our model under various level of loads Λ_{in}, and we calculate the mean sojourn time spent by packets in the queue of the most congested port, namely port 4.

Figure 10 illustrates the corresponding results. Assuming a load of 5 Mpps, and a QoS objective set to 200 μs, it suffices to provision 5 CPU cores to satisfy this QoS level. On the other hand, if the maximum tolerable delay is 20 μs, then a total of 6 CPU cores are needed to maintain the QoS level.

A similar study can be carried out for the mean queueing size of the most congested RX queue. We see from Figure 11 that if ones assumes a load of 5 Mpps but with a QoS objective set to 50% of buffer occupancy, 3 CPU cores are enough to satisfy this QoS level.

Overall, these dimensioning curves, easily and quickly obtained through our model, indicate the right amount of CPU cores to provision to a virtual switch according to a certain level of load, and therefore may ensure a QoS level or save computing resources for other purposes.

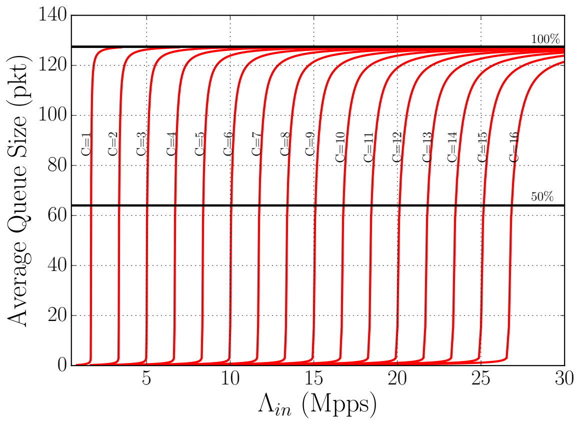


Fig. 11. Sizing the number of CPU cores in a virtual switch according the predicted mean queue size.

VI. Conclusion

This paper presents an analytical queueing model to evaluate the performance of a virtual switch with several network interface cards and several CPU cores. Polling systems, in which a set of servers sequentially poll packets from a set of queues, appears as a natural representation for modeling the behavior of a virtual switch. However, to circumvent the combinatorial growth of the state space associated with these models, the proposed approach decouples the polling system associated with each CPU into several queues, and resorts to servers with vacation to capture the interactions between queues. In this paper, we limited our analysis to the case in which packets are handled sequentially, i.e., by batches of size 1.

We carried out tens of examples to assess the accuracy of our proposed model for various performance parameters such as the attained throughput, the packet latency, the buffer occupancy and the packet loss rate, that may be of interest when sizing the resources of a virtual switch. The accuracy of our approximation is typically very good. Additionally, the proposed approximation is simple to implement and its execution speed is fast. This allows us to use the model to provide dimensioning curves analogous to the famous Erlang curves, in order to instantaneously estimate the minimum number of CPU core to allocate to the switch in order to satisfy a given QoS criterion.

Future works will be devoted to extending our model to let it handle the case of batches of arbitrary sizes. We will also investigate the issue of automatically and efficiently discovering the minimum number of CPU cores to meet a given QoS criterion.

Ackowledgment

This work is partly funded by the French ANR REFLEXION under the “ANR-14-CE28-0019” project.

Appendix 1: Markov chain solution

The continuous-time Markov chain associated with a single queue i of the decomposition is illustrated in Figure 12 with some cuts that will be useful in the resolution.

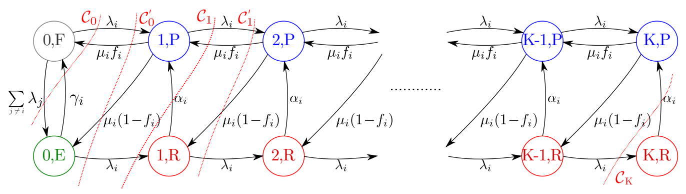


Fig. 12. Continuous-Time Markov Chain associated with queue i.

If we first consider frontier equations associated with cuts $\mathcal{C}_0$ and $\mathcal{C}'_0$, we obtain the two following stationary equations:

$$\begin{aligned}\mathcal{C}_0: &\quad \gamma_i\pi_i(0,E)+\mu_i f_i\pi_i(1,P)=\lambda_{in}\pi_i(0,F)\\ \mathcal{C}'_0: &\quad \mu_i\pi_i(1,P)=\lambda_i(\pi_i(0,E)+\pi_i(0,F))\end{aligned}$$

From these two equations, we can obtain $\pi_i(0,E)$ and $\pi_i(1,P)$ as a function of $\pi_i(0,F)$:

$$\pi_i(0,E)=\frac{\pi_i(0,F)}{\gamma_i+f_i\lambda_i}\left(\lambda_{in}-f_i\lambda_i\right) \tag{14}$$

$$\pi_i(1,P)=\frac{\lambda_i}{\mu_i}(\pi_i(0,E)+\pi_i(0,F)) \tag{15}$$

We apply the same principle to cuts $\mathcal{C}_1$ and $\mathcal{C}'_1$:

$$\begin{aligned}\mathcal{C}_1: &\quad \alpha_i\pi_i(1,R)+\mu_i f_i\pi_i(2,P)=\lambda_i(\pi_i(1,P)+\pi_i(0,E))\\ \mathcal{C}'_1: &\quad \mu_i\pi_i(2,P)=\lambda_i(\pi_i(1,P)+\pi_i(1,R))\end{aligned}$$

giving:

$$\pi_i(1,R)=\frac{\lambda_i}{\alpha_i+f_i\lambda_i}\left[\pi_i(0,E)+(1-f_i)\pi_i(1,P)\right] \tag{16}$$

$$\pi_i(2,P)=\frac{\lambda_i}{\mu_i}(\pi_i(1,R)+\pi_i(1,P)) \tag{17}$$

We can generalize this result for all $k\in[\![2,K-1]\!]$ and obtain:

$$\pi_i(k,R)=\frac{\lambda_i}{\alpha_i+f_i\lambda_i}\left[\pi_i(k-1,R)+(1-f_i)\pi_i(k,P)\right] \tag{18}$$

$$\pi_i(k+1,P)=\frac{\lambda_i}{\mu_i}(\pi_i(k,P)+\pi_i(k,R)) \tag{19}$$

Finally, the last cut $\mathcal{C}_K$ gives:

$$\pi_i(K,R)=\frac{\lambda_i}{\alpha_i}\pi_i(K-1,R) \tag{20}$$

Finally, all stationary probabilities of the chain have been expressed from $\pi_i(0,F)$. This last remaining unknown is of course obtained by normalization.

References

[1] G. Pongrácz, L. Molnár, and Z. L. Kis, “Removing roadblocks from SDN: openflow software switch performance on intel DPDK,” in *2nd European Workshop on Software Defined Networks, EWSDN 2013, October 10-11, 2013*, 2013, pp. 62–67.

[2] R. Jain and S. Paul, “Network virtualization and software defined networking for cloud computing: a survey,” *IEEE Communications Magazine*, vol. 51, no. 11, pp. 24–31, November 2013.

[3] N. McKeown, T. Anderson, H. Balakrishnan, G. M. Parulkar, L. L. Peterson, J. Rexford, S. Shenker, and J. S. Turner, “OpenFlow: enabling innovation in campus networks,” *Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[4] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, “The Design and Implementation of Open vSwitch,” in *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, May 4-6, 2015*, 2015, pp. 117–130.

[5] M. Honda, F. Huici, G. Lettieri, and L. Rizzo, “mswitch: A highly-scalable, modular software switch,” in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, 2015, pp. 1–1:13.

[6] L. Rizzo, “netmap: A novel framework for fast packet I/O,” in *2012 USENIX Annual Technical Conference, June 13-15, 2012*, 2012, pp. 101–112.

[7] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, “Performance characteristics of virtual switching,” in *3rd IEEE International Conference on Cloud Networking, CloudNet 2014, October 8-10, 2014*, 2014, pp. 120–125.

[8] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, “Scalable, high performance ethernet forwarding with cuckooswitch,” in *Conference on emerging Networking Experiments and Technologies, CoNEXT ’13, December 9-12, 2013*, 2013, pp. 97–108.

[9] V. Mann, A. Vishnoi, and S. Bidkar, “Living on the edge: Monitoring network flows at the edge in cloud data centers,” in *Fifth International Conference on Communication Systems and Networks, COMSNETS 2013, January 7-10, 2013*, 2013, pp. 1–9.

[10] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia, “Modeling and performance evaluation of an openflow architecture,” in *23rd International Teletraffic Congress, ITC 2011, September 6-9, 2011*, 2011, pp. 1–7.

[11] H. Takagi, “Queuing analysis of polling models,” *ACM Computing Surveys*, vol. 20, no. 1, pp. 5–28, 1988.

[12] ——, “Analysis of finite-capacity polling systems,” *Advances in Applied Probability*, vol. 3, no. 2, pp. 373–387, jun 1991.

[13] P. Tran-Gia and T. Raith, “Performance Analysis of Finite Capacity Polling Systems with Nonexhaustive Service,” *Performance Evaluation*, vol. 9, no. 1, pp. 1–16, 1988.

[14] M. Lang and M. Bosch, “Performance analysis of finite capacity polling systems with limited-m service,” in *13rd International Teletraffic Congress, ITC 1991*, 1991, pp. 731–735.

[15] T. T. Lee, “M/G/1/N queue with vacation time and exhaustive service discipline,” *Operations Research*, vol. 32, no. 4, pp. 774–784, Aug. 1984.

[16] ——, “M/G/1/N queue with vacation time and limited service discipline,” *Performance Evaluation*, vol. 9, no. 3, pp. 181–190, 1989.

[17] D. Kofman, “Blocking probability, throughput and waiting time in finite capacity polling systems,” *Queueing Systems*, vol. 14, no. 3-4, pp. 385–411, 1993.

[18] A. Sohail, “Performance evaluation of a non-exhaustive polling system with asymmetrical finite queues,” in *14th International Conference on Computer Modelling and Simulation, 2012 UKSim, Cambridge, United Kingdom, March 28-30, 2012*, 2012, pp. 613–617.