mathematics

Article

Optimality of a Network Monitoring Agent and Validation in a Real Probe

Luis Zabala [1], Josu Doncel [2,*] and Armando Ferro [1]

[1] Department of Communications Engineering, University of the Basque Country, UPV/EHU, 48013 Bilbao, Spain
[2] Department of Mathematics, University of the Basque Country, UPV/EHU, 48940 Leioa, Spain
* Correspondence: josu.doncel@ehu.eus

Abstract: The evolution of commodity hardware makes it possible to use this type of equipment to implement traffic monitoring systems. A preliminary empirical evaluation of a network traffic probe based on Linux indicates that the system performance has significant losses as the network rate increases. To assess this issue, we consider a model with two tandem queues and a moving server. In this system, we formulate a three-dimensional Markov Decision Process in continuous time. The goal of the proposed model is to determine the position of the server in each time slot so as to optimize the system performance which is measured in terms of throughput. We first formulate an equivalent discrete-time Markov Decision Process and we propose a numerical method to characterize the solution of our problem in a general setting. The solution we obtain in this problem has been tested for a wide range of scenarios and, in all the instances, we observe that the optimality is close to a threshold type policy. We also consider a real probe and we validate the good performance of threshold policies in real applications.

Keywords: network monitoring; performance evaluation; Markov decision process; Linux network subsystem

MSC: 90B15; 60K30; 93E20

Citation: Zabala, L.; Doncel, J.; Ferro, A. Optimality of a Network Monitoring Agent and Validation in a Real Probe. *Mathematics* **2023**, *11*, 610. https://doi.org/10.3390/math11030610

Academic Editor: Kagan Eugene

Received: 20 December 2022
Revised: 13 January 2023
Accepted: 20 January 2023
Published: 26 January 2023

1. Introduction

The complexity of modern communication networks has dramatically increased in the last years. As a result, traffic monitoring of such complex systems, which is required by network operators to cope with their users' needs to guarantee their satisfaction, has become an extremely challenging task.

Different applications can be related to traffic monitoring systems, ranging from real-time multimedia traffic monitoring [1] to intrusion detection systems [2] and network antiviruses [3], to mention a few. Furthermore, nowadays, with the proliferation of cloud services, traditional network and data center environments must be complemented with cloud network monitoring tools in order to keep track of performance indicators. This is accomplished by deploying some lightweight monitoring software agents that collect statistics on physical or virtual servers and physical or virtual network devices [4]. Then, each of these agents will send their data to a repository that typically provides big data-based analytics for alerting, diagnostics and displaying summarized information.

It has been shown that the advances of commodity hardware allow software to attain good performance while the main advantages of flexibility and modularity still hold, see [5,6]. This can be achieved, for instance, with an efficient use of the kernel modules.

Within the data monitoring systems based on commodity hardware, Linux-based network end systems have been widely deployed [7,8]. From a network performance perspective, Linux represents an opportunity since it is amenable to optimization and tuning network stack, due to its open source support. Whenever we want to know how our network behaves, it is important to have a packet capturing device able to achieve rates similar to those on the network, to obtain the most comprehensive view.

We represent in Figure 1 the evolution of packets since entering the Linux end system [9] to their delivery to the monitoring application. This is achieved in three different stages:

- The packet is transferred from Network Interface Card (NIC) to the ring buffer. Since this is a Direct Memory Access (DMA) transfer, all of the actions related to this packet movement are completed without consuming any CPU's resources.
- The packet is transferred from the ring buffer to the analysis buffer, driven by a software interrupt request (*softirq*) [10]. This is the capturing stage in Figure 1.
- Finally, the packets are treated by the monitoring application and they are pulled out from the analysis buffer. This is the analyzing process in Figure 1.

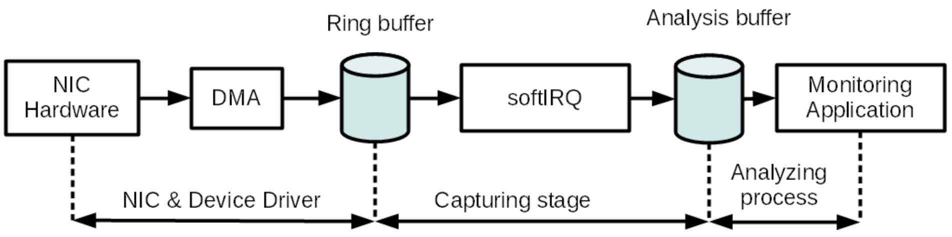


Figure 1. Packet receiving process with Linux networking subsystem.

It is remarkable that, while the DMA transfer does not have any CPU consumption, both the capturing process and the analyzing process do have it. Additionally, if there is only one processor, both processes will not be running at the same time. As a consequence, in order to design traffic monitoring systems optimally, it is interesting to build mathematical models which allow us to study their performance theoretically. Specifically, there is an increasing demand for analytical models to aid probe designers in predicting how effective and efficient is the network probe when subjected to high-speed traffic. In addition, modeling and analyzing the performance of network probes can be extremely useful in gaining a deeper understanding of the probe's behavior and characteristics. Probe designers and system administrators can identify bottlenecks and key parameters that impact its performance, and then perform the necessary tuning for optimal performance. Analysis can provide quick answers to numerous design and operational questions.

Our objective is to develop a mathematical model that describes the behavior of a network traffic monitoring agent and to gain insights about how to optimize its performance. For this purpose, we consider an analytical queuing model developed for the study and analysis of the performance of a single-processor network traffic probe built on off-the-shelf hardware. We represent the two main stages (capturing and analysis) of the traffic monitoring system using a single-server tandem queue. However, as we represent a single-processor system, the two servers of the tandem queue cannot be active simultaneously and, for this reason, we propose a Markov Decision Process (MDP) to choose which server has to be active at each time and obtain the optimum scheduling policy that allows us to achieve a better performance in terms of network analysis throughput. The model proposal is evaluated and, finally, the results obtained from the model are compared to those that we obtain in a real Linux-based network probe.

The performance of traffic analysis solutions has been studied experimentally under multi-Gigabit network environments [11–13]. However, few of these solutions have been modeled and studied analytically in order to assess the performance and behavior of system performance under heavy network loads. For instance, in [14], the Linux system's packet receive process is represented by the token bucket algorithm and queuing processes. The authors of [15] present a finite queuing system with multiple stages of service to represent the behavior of a rule-based network firewall. Another approach is the one presented in [16] which describes an $M^x/G^B/1/K$ queue with vacations to formulate high-speed packet I/O frameworks. In [17,18], the analytical models, based on $M/M/1$ and $M/M/c/K$ queues, estimate the computing resources required to satisfy the Quality of Service (QoS) targets of some cloud monitoring applications.

There are some researches to measure power consumption of packet processing engines [19,20]. Among them, we would like to mention [19]. This work proposes an $M^x/D/1$ queue with a setup period and it adds the resolution of an optimization problem to study the trade-off between energy consumption and network performance indexes.

To finish the related work review, we would like to highlight the application of queuing models in the field of Fifth-Generation (5G) networks. Here, multiple software components, referred to as Virtual Network Functions (VNF), are connected and they process packets to provide a service chain. Network monitoring can be one of those service chains. This type of model represents each VNF with a queue. For instance, in [21], the authors formulate an open queuing network of G/G/m queues to predict the performance of generic softwarized network services in terms of response time. Other examples of this research line can be [22], which addresses a problem of VNF placement by using an open queuing network of $M/M/1$ queues, and [23], which explains how to model the building block of a 5G node.

As far as we know, most of the previous studies presented in the literature do not take account of the relation between the capturing stage and the analysis stage of a network monitoring system. For this reason, the main contribution of this article is to study the application of a novel analytical model based on classical concepts of MDPs, considering processing and buffering limitations of the capture-analysis system and optimizing the performance in terms of analysis throughput.

The remainder of the paper is organized as follows. The queuing model for the capturing and analysis system is presented in Section 2. Then, in Section 3, the optimization problem based on MDP is detailed including its parameters and the solution. The model validation and the obtained results are discussed in Sections 4 and 5. Finally, we conclude the paper in Section 6.

2. Model Description

In this section, we represent the packet receiving process of a traffic monitoring system (see Figure 1) using an open queuing network, as drawn in Figure 2. The system described in Section 1 is modeled by a tandem queue, and computer consumptions are related to service capacities of the queuing network.

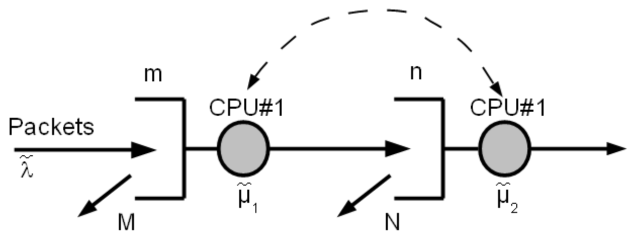


Figure 2. Queuing system representation for the case of one single processor.

As can be seen in Figure 2, packets arrive at the first queue of the model. This packet arrival corresponds to the DMA transfer from the NIC to the ring buffer without any CPU consumption. Thus, the first queue represents the ring buffer of the capturing stage where the packets are stored before being processed by the *softirq*. The model assumes that the first queue, the *capturing queue* henceforth, has got a bounded capacity of M packets. If a packet arrives when the *capturing queue* is full, then the packet is dropped out of the system without being processed. Next, the first server treats the packets of the *capturing queue*. This represents the *softirq* processing. For this reason, the service time of the first server is related to the processing time required by the CPU for the *softirq*.

Once a packet is served by the first server, i.e., the *softirq* processing of the packet is completed, it moves to the second stage of the queuing network. This is related to the analysis stage where packets are waiting in the analysis buffer until they are processed by the monitoring application of the network traffic monitoring system (see Figure 1). In the model, the second waiting queue and the second server represent the analysis buffer and the analysis processing, respectively. Just as the *capturing queue*, the second queue, referred to as the *analysis queue* henceforth, is also considered a finite queue. It has got a capacity of N packets and, when it is full, the arriving packets cannot enter it. Finally, as soon as the second server completes the analysis processing of a packet, this packet leaves the network, meaning that the monitoring application has completed the task related to this packet.

In order that the analytical solution does not become unmanageable, we assume that packets arrive at the *capturing queue* following a Poisson process with parameter $\tilde{\lambda}$. We also suppose that both the time needed by the processor to complete the capturing process of each packet and the analysis time of each packet in the second stage follow an exponential random variable with parameters $\tilde{\mu}_1$ and $\tilde{\mu}_2$, respectively. It is proven that packet arrivals do not follow a Poisson process for some types of traffic such as Ethernet network and that they are rather bursty [24], but the case we are dealing with is slightly different. In our case, the packet arrival is not directly the traffic of the Ethernet network, but it is the incoming packets from the network card's buffer to the kernel memory area via DMA. Regarding service rate modeling, although the program's code has a quite deterministic behavior, some randomness is introduced by Poisson incoming traffic, variable length of packets and kernel scheduler uncertainty. Despite of these limitations, we will keep working with these assumptions for simplicity of the analysis and, as will be demonstrated later, the results obtained from our model were closely matched to results obtained from real experimental measurements.

The queuing network consists of two servers, however, as we are going to consider the case of a single-processor system, this single CPU has to divide its efforts among the two tasks (capturing and analysis). For this reason, in the model, the two servers of the queuing network (both of them named as #*CPU1* in Figure 2) cannot be active at the same time. Therefore, the system consists of a two-stage tandem queue attended by a moving server. According to the preferences of the system scheduler, labeled as the *decision maker* henceforth, the processor is allocated to the *capturing queue* or the *analysis queue*.

The way the *decision maker* chooses the right action for the dynamic processor at any time has a big impact in the performance of the network, measured in terms of analysis throughput. Allocating it too much time to the *analysis queue* may lead to packet loss in the *capturing queue*, whereas allocating it too much to the *capturing queue* may decrease the total capacity for packet analysis. In this paper, we aim to provide insights into how the *decision maker* should proceed with the goal of optimizing the performance of the system. Our analysis is based on a three-dimensional MDP. This allows us to obtain a dynamic policy that tells the *decision maker* where to allocate the processor at any time. This decision will depend on the number of packets waiting in each queue and the *position* of the processor when the decision is made.

An Equivalent Discrete Time Model

Next, we build a discrete time model, which describes the dynamics of the continuous time model introduced above. This is done via uniformization techniques. The discrete time model allows us to test its accuracy when explaining the performance of the traffic monitoring system. On the other hand, it provides an easier optimization framework, which helps to find the protocol that optimizes the analysis throughput.

Using uniformization arguments, we define the following quantities:

$$\lambda = \frac{\tilde{\lambda}}{\tilde{\lambda} + \tilde{\mu}_1 + \tilde{\mu}_2}. \quad (1)$$

$$\mu_1 = \frac{\tilde{\mu}_1}{\tilde{\lambda} + \tilde{\mu}_1 + \tilde{\mu}_2}. \tag{2}$$

$$\mu_2 = \frac{\tilde{\mu}_2}{\tilde{\lambda} + \tilde{\mu}_1 + \tilde{\mu}_2}. \tag{3}$$

Thus, we move from a continuous to a discrete system where:

- At any time slot, and given that the *capturing queue* is not full (with M packets in its buffer), a new packet arrives to the network with probability λ;
- If the processor is working in the *capturing queue*, and the queue is not empty, the processor captures a packet within one time slot with probability μ_1;
- If the processor is working in the *analysis queue*, and this queue is not empty, the processor finishes the analysis stage of a packet within one time slot with probability μ_2.

The probabilities given in Equations (1)–(3) are independent from one time slot to another. The length of each time slot, in units of time, is given by

$$t_s = \frac{1}{\tilde{\lambda} + \tilde{\mu}_1 + \tilde{\mu}_2}. \tag{4}$$

3. MDP Formulation

In this section, we present an optimization problem which leads to the maximization of the analysis throughput in our discrete time model described in Section 2. This optimization problem falls in the framework of the MDPs. An MDP consists of a state space, an action space, and a reward structure. Whenever the *decision maker* chooses among the available actions, it earns a reward which depends on the current state of the system, and after receiving it, the system evolves its state to a new one. The goal of the *decision maker* is to find the right sequence of actions, maximizing the expected total reward earned from the system.

It is easy to see that the model can be seen as a a three-dimensional discrete time MDP. We consider a time-slotted system $t \in \mathcal{T} := \{0, 1, 2, \dots\}$, at which the system can make decisions. The time epoch t corresponds to the beginning of the time period t. At any t, the *decision maker* has to choose among two actions: either allocate the processor to the *capturing queue* or the *analysis queue*. The state of the system is represented by the number of packets waiting in both queues and the position of the processor before the decision is made.

We provide below the elements that describe the evolution of the system:

- $\mathcal{A} := \{C, A\}$ is the *action set* of the system. At any time, selecting action C represents allocating the processor to the first queue for packet capturing, whereas action A represents allocating the processor to the second queue for packet analysis;
- $\mathcal{S} := \mathcal{M} \times \mathcal{N} \times \mathcal{A}$ is the *state space* of the system. If the state of the system is $s = (m, n, a)$, with $m \in \mathcal{M} := \{0, 1, \dots, M\}$, $n \in \mathcal{N} := \{0, 1, \dots, N\}$ and $p \in \mathcal{A}$, this means that there are *m packets queuing* in the *capturing queue*, *n packets queuing* in the *analysis queue*, and that the previous action taken by the system has been p, representing the current *position* of the processor;
- R_s^a is the *one-period expected reward* earned by the system if action $a \in \mathcal{A}$ is chosen when the system is in state $s \in \mathcal{S}$;
- $P^a := \left(p_{s,s'}^a\right)$ is the state-transition matrix if action $a \in \mathcal{A}$ is decided at the beginning of a period. If $s = (m, n, p)$ and $s' = (m', n', p')$, then the transition probability of moving from state s to state s' is $p_{s,s'}^a$.

3.1. Reward Structure

Since the goal of the *decision maker* is to maximize the analysis throughput, we establish that it earns 1 unit of reward whenever a packet leaves the *analysis queue* due to its completion. Moreover, we also consider that the *decision maker* incurs a cost whenever it decides to change the position of the processor. This cost represents the time that the processor spends changing its position, effectively affecting the performance of the system. We denote this switching cost by SC. The one-period expected rewards R_s^a, depending on the current state $s = (m, n, p)$ and the action a chosen by the *decision maker*, can be expressed as follows:

$$R^a_{s=(m,n,p)} = \begin{cases} 0, & p = C, a = C, 0 \leq n \leq N, \\ -SC, & p = C, a = A, n = 0, \\ \mu_2 - SC, & p = C, a = A, 1 \leq n \leq N, \\ -SC, & p = A, a = C, 0 \leq n \leq N, \\ 0, & p = A, a = A, n = 0, \\ \mu_2, & p = A, a = A, 1 \leq n \leq N, \end{cases} \quad 0 \leq m \leq M \tag{5}$$

The reward is zero when (i) the processor stays in the capturing queue or (ii) the processor stays in analysis queue and the analysis queue is empty. The reward is minus the switching cost when (i) the processor moves from the capturing queue to the analysis queue and the analysis queue is empty or (ii) the processor moves from the analysis queue to the capturing queue. The reward is equal to μ_2 when the processor stays at the analysis queue and this queue is not empty. Finally, the reward is μ_2 minus the switching cost when the processor moves from the capturing queue to the analysis queue and the analysis queue is not empty.

Observe that this problem of minimizing costs is not equivalent to maximizing the throughput, but, as we will see later, the obtained results fulfill the goal of this work.

3.2. Transition Probabilities

As mentioned before, $p^a_{s,s'}$ is the probability of moving from the state $s = (m, n, p)$ to the state $s' = (m', n', p')$ when action a is taken at the beginning of the transition period. Thus, the value of p, the position of the processor at the state $s = (m, n, p)$, coincides with the action a, i.e., $p = a$ at the state s. On the other hand, the position p' of the state s' will be the value of the action decided at the beginning of the next transition period. The value of p' does not affect the transition probability.

The transition probabilities $p^a_{s,s'}$ can be separately obtained by using the two cases of a. Firstly, for action $a = C$, state $s = (m, n, p)$ with $p = a = C$, and state $s' = (m', n', p')$,

$$p^{a=C}_{s,s'} = \begin{cases} 1-\lambda & m = 0,\ 0 \leq n \leq N,\ m' = 0,\ n' = n \\ \lambda & 0 \leq m \leq M-1,\ 0 \leq n \leq N,\ m' = m+1,\ n' = n \\ 1-\lambda-\mu_1 & 1 \leq m \leq M-1,\ 0 \leq n \leq N,\ m' = m,\ n' = n \\ \mu_1 & 1 \leq m \leq M,\ 0 \leq n \leq N-1,\ m' = m-1,\ n' = n+1 \\ \mu_1 & 1 \leq m \leq M,\ n = N,\ m' = m-1,\ n' = N \\ 1-\mu_1 & m = M,\ n = 0,\ m' = M,\ n' = 0 \\ 1-\lambda-\mu_1 & m = M,\ 1 \leq n \leq N,\ m' = M,\ n' = n \\ \lambda & m = M,\ 1 \leq n \leq N,\ m' = M,\ n' = n \end{cases} \tag{6}$$

In Equation (6), the probabilities assume that

$$\lambda(1-\mu_1) \approx \lambda \tag{7}$$

$$(1-\lambda)\mu_1 \approx \mu_1 \tag{8}$$

$$(1-\lambda)(1-\mu_1) \approx 1-\lambda-\mu_1 \tag{9}$$

The above expressions represent the transition probabilities when the processor is in the capture node. We describe a few of them here. When the capture queue is not full, it receives a new packet with probability λ. When this queue is empty, the probability that the state is not modified is $1-\lambda$, whereas when it is not empty, this probability is $1-\lambda-\mu_1$. When the capturing queue is full, the probability that the state does not change is λ.

Secondly, for action $a = A$, state $s = (m, n, p)$ with $p = a = A$, and state $s' = (m', n', p')$,

$$p_{s,s'}^{a=A} = \begin{cases} 1-\lambda & 0 \le m \le M-1,\ n=0,\ m'=m,\ n'=0 \\ \lambda & 0 \le m \le M-1,\ 0 \le n \le N,\ m'=m+1,\ n'=n \\ 1-\lambda-\mu_2 & 0 \le m \le M,\ 1 \le n \le N,\ m'=m,\ n'=n \\ \mu_2 & 0 \le m \le M,\ 1 \le n \le N,\ m'=m,\ n'=n-1 \\ 1 & m=M,\ n=0,\ m'=M,\ n'=0 \\ \lambda & m=M,\ 1 \le n \le N,\ m'=M,\ n'=n \end{cases} \tag{10}$$

Just as the case of action $a = C$, in Equation (10), the probabilities assume that

$$\lambda(1-\mu_2) \approx \lambda \tag{11}$$
$$(1-\lambda)\mu_2 \approx \mu_2 \tag{12}$$
$$(1-\lambda)(1-\mu_2) \approx 1-\lambda-\mu_2 \tag{13}$$

The above expressions represent the transition probabilities when the processor is in the analysis node. As it can be seen, the transition probabilities are analogous to those of the case where the processor is in the capturing node.

Thus, the dynamics of the system is captured by the state process $s(\dot{})$ and the action process $a(\dot{})$, which correspond to state $s(t) \in \mathcal{S}$ and action $a(t) \in \mathcal{A}$ at all time epoch $t \in \mathcal{T}$. As a result of choosing action $a(t)$ at the beginning of time slot $t \in \mathcal{T}$ while the system is in state $s(t)$, the *decision maker* earns a reward, and the system evolves its state for the next time slot, $t+1$.

3.3. Optimization Problem

Now, we describe, in more detail, the problem we consider in this paper, and we give its mathematical formulation. We denote by $\Pi_{s,a}$ the set of randomized and non-anticipating policies, available for the *decision maker*. These policies assume that, at time $t \in \mathcal{T}$, the *decision maker* has exact information about the state process until t, i.e., it knows $s(0), s(1), \ldots, s(t)$, and it also knows all the decisions that have been made before, i.e, $a(0), a(1), \ldots, a(t-1)$.

The problem is to find a policy $\pi \in \Pi_{s,a}$ that maximizes the expected total future rewards considering an infinite time horizon, i.e.,

$$\max_{\pi \in \Pi_{s,a}} \mathbb{E}_0^{\pi} \left(\sum_{t=0}^{\infty} R_{s(t)}^{a(t)} \right), \tag{14}$$

where $\mathbb{E}_0$ denotes the expectation over the evolution of the system, conditioned to the information available at $t = 0$.

3.4. Numerical Solution

Problem (14) is a standard MDP, for which it is well known that there exists an optimal policy which is deterministic (non-randomized), stationary (Markovian) and independent of the initial state. In particular, this implies the existence of an optimal policy which, at any time t, only depends on the state of the system $s(t)$. An optimal Markovian solution of the MDP can be obtained by solving the Bellman equation (see [25,26]). However, the fact that our MDP lies on a three-dimensional state space, together with the existence of switching costs, makes the Bellman equation intractable from an analytic point of view, and thus, we are forced to solve the MDP using numerical methods. In particular, our solutions are obtained following the value iteration algorithm, which we briefly describe below:

Let us denote by $\mathbb{V}_s^{\pi}$ the value function of the MDP starting from state $s \in \mathcal{S}$ at time $t = 0$ and following policy $\pi \in \Pi_{s,a}$, i.e.,

$$\mathbb{V}_s^{\pi} = \mathbb{E}^{\pi}\left(\sum_{t=0}^{\infty} R_{s(t)}^{a(t)} | s(0) = s\right). \tag{15}$$

The value iteration algorithm proceeds as follows:

- Fix an arbitrary initial value $\mathbb{V}_s^0$ for all $s \in \mathcal{S}$.
- For $i = 1, 2, \ldots, n$, calculate recursively:
  - $\mathbb{V}_s^i = \max\limits_{a \in \mathcal{A}}\{R_s^a + \sum\limits_{s' \in \mathcal{S}} p_{s,s'}^a \mathbb{V}_{s'}^{i-1}\}$
  - $a_s^i = \arg\max\limits_{a \in \mathcal{A}}\{R_s^a + \sum\limits_{s' \in \mathcal{S}} p_{s,s'}^a \mathbb{V}_{s'}^{i-1}\}$
- Let n grow

This iterative algorithm converges to the optimal solution of the MDP, providing in the limit two important values:

- $\lim\limits_{i \to \infty} \mathbb{V}_s^i = \mathbb{V}_s^{\pi^*}$, and
- $\lim\limits_{i \to \infty} a_s^i = a_s^{\pi^*}$,

where $\mathbb{V}_s^{\pi^*}$ is the value function following an optimal policy and starting from state s, and $a_s^{\pi^*}$ represents the action chosen by the optimal policy whenever the system is in state s.

The goal is, therefore, to obtain numerically the optimal policy of the three-dimensional state space MDP, by using a set of parameters which reflect the experiments carried out in a real traffic monitoring system. Once we obtain this policy, we will be able to compare the performance of the real system and the one obtained by the Markovian model which follows the optimal policy. This improvement might be seen as the potential improvement of the performance in the traffic monitoring system when using a dynamic allocation policy which depends on the amount of packets in capturing and analysis buffers.

4. Performance Evaluation of the Model

In this section, we solve the optimization problem described above for a given set of parameters. To do so, first of all, we explain how to obtain those input parameters. Next, we describe the evaluation of the MDP model. To conclude the section, we present the results of the simulation that follows the optimal policy obtained from the MDP model.

4.1. Estimation of Parameters

In order to set the model's input parameters ($\tilde{\lambda}$, $\tilde{\mu}_1$, $\tilde{\mu}_2$, M and N), we perform some tests on a Linux-based network probe, which is a prototype called Ksensor [27]. This real probe is installed inside a testbed platform [28], as can be seen in Figure 3.

Inside the testing platform, the traffic generator is in charge of sending data packets to the network (or to a potential receiver). The probe has to capture and process as many network packets as possible. The test manager, with the help of software agents installed in the generator and probe, measures the metrics of interest. Therefore, the test manager provides us with some statistical results of the experimental tests from which we can estimate values for the input parameters of the mathematical model. Such experimental results are, for example, the number of captured packets, CPU consumption in capturing and analysis tasks, the test's duration, etc.

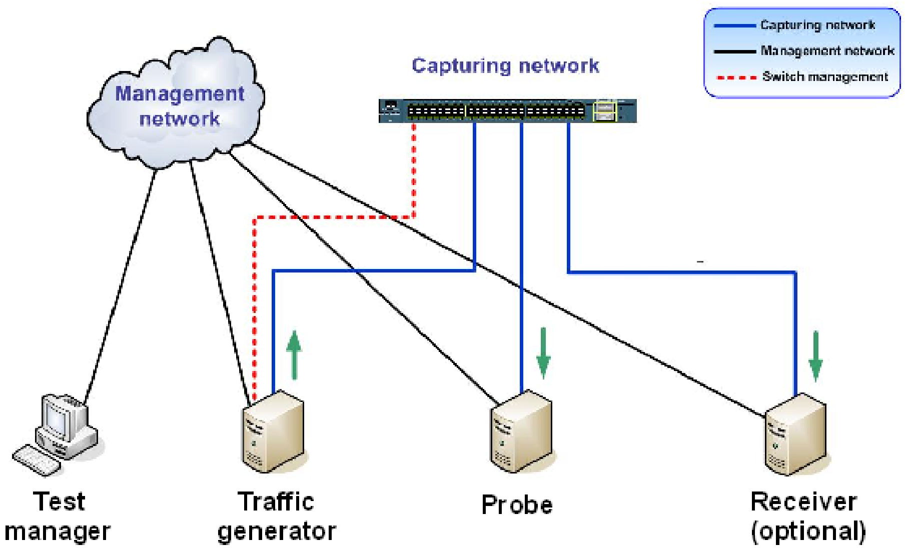


Figure 3. Testbed platform for experimental measurements.

Regarding the hardware and software details of the experimental setup, on the one hand, the traffic generator, the test manager and the receiver are installed on the same kind of physical machine: a computer with two Intel Xeon 5110 CPUs at 1.66 GHz, 2 GB of RAM and running Debian 7 GNU/Linux. In order to inject synthetic network traffic, the traffic generator has an Endace DAG 4.3GE card installed [29]. The packets generated are minimum sized IP packets (40 bytes) for 1 Gbps Ethernet network. This means that the probe will receive the maximum number of packets possible. On the other hand, the probe is installed on a machine with two Intel Quad Xeon 5420 CPUs at 2.5 GHz, with 4 cores each, 4 GB of RAM and running a modification of the kernel Linux 2.6.23 with a module that implements the analysis task. In our MDP model-oriented experiments, the probe is configured to use a single CPU core. Lastly, as Figure 3 shows, all the machines are connected to the same 1 Gbps Ethernet switch.

Some experimental results allow us to set the values of the input parameters of the model as follows:

- $\tilde{\lambda}$ is related to the network packet rate. In the testing scenario, the traffic generator allows us to set the packet injection rate into the network and we choose 21 rates varying from 50,000 to 1,500,000 pps. In addition, the test manager gathers the packet rate from the network switch to which the probe is connected. These values of network traffic rate are directly used as the input parameter $\tilde{\lambda}$ of the model. As we keep the same values of the experimental tests, it will be possible to compare the theoretical results of the numerical evaluation of the model with the experimental ones. Due to the fact that the tests are performed for each one of the rates produced by the traffic generator, there are experimental results for each one of those network traffic rates;
- $\tilde{\mu}_1$ is the packet capturing rate. We estimate it by considering the total amount of packets captured by the *softirq* processing during the test (which we denote as n_{cap}) and the time spent by the CPU in the *softirq* (denoted as T_{cap}). Both n_{cap} and T_{cap} are provided by the test manager since, during the test, the number of packets captured by the polling process of the *softirq* and the number of cycles consumed by the CPU in the capturing stage are counted. A simple conversion from CPU cycles to seconds allows us to obtain T_{cap}. Thus, $\tilde{\mu}_1$ is estimated as follows:

$$\tilde{\mu}_1 = \frac{n_{cap}}{T_{cap}}. \tag{16}$$

- $\tilde{\mu}_2$ is the packet analysis rate and, if n_{an} and T_{an} denote the total amount of packets processed by the analysis stage during the test and the total time devoted by the CPU to analysis purposes, respectively, we obtain the following estimation for $\tilde{\mu}_2$:

$$\tilde{\mu}_2 = \frac{n_{an}}{T_{an}}. \tag{17}$$

  Similarly to T_{cap}, the term T_{an} is derived from the number of cycles consumed by the CPU in the analysis stage during the test. It is also worth mentioning that the probe's performance is measured under different analysis loads. Three scenarios are implemented to simulate a low analysis load, a medium analysis load and a high analysis load. For this reason, there are different values assigned for the parameter $\tilde{\mu}_2$, one for each analysis load. As each analysis load is implemented with a loop that takes a different number of cycles in each case (in particular, 0, 300 and 1000 cycles for low, medium and high loads, respectively), from now on, we will assign *null* to the low load scenario, *0.3 k* to the medium load scenario and *1 k* to the high load scenario;
- M is the capacity of the capturing buffer. In the case of the probe used in the testbed, its typical value is 200. For this reason, $M = 200$ for the numerical evaluation;
- N is the capacity of the analysis buffer. The typical value is 4096 in the probe of the test. Therefore, $N = 4096$.

Once we have estimated values for $\tilde{\lambda}$, $\tilde{\mu}_1$, $\tilde{\mu}_2$, M and N, we obtain λ, μ_1 and μ_2 following Equation (1)–(3), respectively. We also calculate the length of each time slot of the simulated discrete time model, t_s, invoking (4).

Another key aspect of the model is that the traffic monitoring system wastes time whenever the processor changes its task. The processor needs a non negligible amount of time to switch from the packet capturing task to the packet analysis task, and the same thing happens when it moves from the analysis to the capturing. Let us denote this switching time by t_{sw}. Our estimation of t_{sw} is also derived from a testing scenario, in particular, from one with high packet traffic rate. Under this condition, we assume that there are no time intervals in which the processor is idle.

First, we calculate the time spent by the system switching the position of the processor (T_{sw}). If the test's duration is T_{tot} (value registered by the test manager), we have that

$$T_{sw} = T_{tot} - T_{cap} - T_{an}, \tag{18}$$

Dividing T_{sw} by the number of switches, n_{sw}, we obtain the time that the processor needs to make each switch:

$$t_{sw} = \frac{T_{sw}}{n_{sw}}. \tag{19}$$

The number of switches is derived from the number of *softirqs* counted during the test. The number of *softirqs* is also registered by the test manager. If there are no idle periods under high network packet rate, we assume that every *softirq* process is followed by an analysis process and vice versa. The CPU is in use almost all the time for those two processes, except for switching context. Therefore, we can consider that there are two context switches per each *softirq*, one at the transition *softirq-analysis* and the other at the transition *analysis-softirq*. In this way, we set the value of n_{sw}.

Last, we calculate the value of the switching cost, which allows us to define a meaningful reward structure for our optimization problem. Since we fix 1 unit of reward whenever the analysis of a packet is completed, it is reasonable to establish the switching cost as the proportion of time lost during the switch compared to the total time that is needed to process a packet since it arrives at the system until it leaves when the analysis is completed. Thus, we calculate the switching cost as follows:

$$SC = \frac{t_{sw}}{\frac{1}{\tilde{\mu}_1} + \frac{1}{\tilde{\mu}_2}}. \quad (20)$$

From the scenario with the highest network traffic rate, we estimate the switching time and the switching costs. The switching time is fixed as $t_{sw} = 850$ ns for all the scenarios, whilst the switching cost depends on the value of $\tilde{\mu}_1$ and $\tilde{\mu}_2$ according to Equation (20).

Table 1 shows the values of the input parameters which are estimated from the testing scenario. In order not to make too big the size of the table, the 21 values assigned to $\tilde{\lambda}$ are not written, but the smallest and the biggest ones are. The table also displays the average of $\tilde{\mu}_1$ and three average values of $\tilde{\mu}_2$, one for every analysis load (*null*, *0.3 k* and *1 k*). Both the values of $\tilde{\lambda}$ and those of $\tilde{\mu}_1$ and $\tilde{\mu}_2$ are in packets per second (pps). Similarly, the particular values used for M, N and SC appear in the table. As a result of the combination of all the possible values taken for $\tilde{\lambda}$, $\tilde{\mu}_1$, $\tilde{\mu}_2$, M, N and SC, we can state that we will evaluate the model for 63 different scenarios.

Table 1. Set of values for the input parameters of the numerical evaluation.

$\tilde{\lambda}$ (pps)	$\tilde{\mu}_1$ (pps)	$\tilde{\mu}_2$ (pps)	M	N	SC
50,000		902,798 (*null*)			0.4 (*null*)
···	1,188,786	289,119 (*0.3 k*)	200	4096	0.2 (*0.3 k*)
1,500,000		116,755 (*1 k*)			0.1 (*1 k*)

4.2. Structure of the Optimal Policy

Having estimated the values of the input parameters, in this section, we evaluate numerically the MDP model and we analyze the structure of the optimal policy, obtained by following the value iteration algorithm for each scenario proposed in Table 1. For a given scenario, the optimal policy consists of two matrices. The first matrix, $C = (c_{i,j})_{i=1,\ldots,M,j=1,\ldots,N}$, describes the action followed by the optimal policy when the processor is allocated to the *capturing queue*, and there are i and j packets queued in the *capturing* and the *analysis queues*, respectively. The second one, $A = (a_{i,j})_{i=1,\ldots,M,j=1,\ldots,N}$, represents the optimal choice when the processor is allocated to the *analysis queue*. As will be shown later, matrices C and A will be represented in two-dimensional diagrams by using the blue color to indicate the set of states where the optimal policy is to allocate the processor to the *capturing queue*, whereas the white area will indicate that, in these states, it is optimal to dedicate the processor to analysis purposes. Based on our simulations, we see that the shape of the optimal policy depends on the set of parameters of each scenario, and conclude that the optimal policy is always given in terms of a switching curve. However, we see that the shape of this switching curve follows an specific pattern which depends on the saturation of the system. Let us see it by considering some examples that distinguish three ranges of network traffic rates.

Firstly, Figure 4 shows the shape of the optimal policy when the network traffic rate is low, 100,000 pps, the analysis load is *null*, $M = 200$ and $N = 4096$. According to the optimal policy, if the processor is allocated to the *capturing queue*, there is no incentive to move it to the *analysis* stage unless the analysis buffer is full ($n = 4096$). On the other hand, if the processor is already positioned in the *analysis queue*, whether to keep it or to switch it to the *capturing queue* depends on the values of m and n, i.e., the number of packets that are waiting in the buffers, as shown in Figure 4b. In particular for this case, the optimal policy tells us that, while the analysis buffer is not empty, there is not any switch from analysis to capture until the capturing buffer is almost full and the number of packets in the analysis buffer is less than a given value. For the set of parameters of Figure 4, it is optimal to keep the processor in the *analysis queue* if there are less than 198 packets in the *capturing queue* ($m < 198$) and the analysis queue is not empty ($n \neq 0$). If $m < 198$ and the analysis buffer is totally emptied ($n = 0$), then the processor switches to the capturing stage. However, when there are more than 197 packets in the capturing stage ($m \geq 198$), the optimal policy also depends on how many packets are in the analysis buffer, i.e., the processor switches from analysis to capture when the pair (m, n) reaches a given value. In this way, the switch is done if n reaches 4092 packets and m is equal to 200 (full capture buffer) or if n is decreased to 4065 packets and m is equal to 199 (almost full capture buffer) or if n reaches 3875 and m is equal to 198 (almost full capture buffer too). In this first example, due to the low traffic rate, there is no risk of dropping packets from the *capturing queue* if the processor is allocated to analysis purposes with less than 198 packets in the *capturing queue*. Thus, the optimal policy is capable of analyzing the 100% of the packets that arrive to the system, minimizing, at the same time, the total number of switches.

It is easy to interpret why the optimal policy behaves in this way. Maximizing analysis throughput is equivalent to minimizing the total packet loss. Thus, whenever the *analysis queue* is not full, it is good to switch the processor to packet capturing if there is a big risk of dropping packets due to the lack of buffer in the *capturing queue*. On the other hand, if the number of packets in the *capturing queue* is not large enough, then there is no reason to switch processor's *position*, and it can be kept working in the *analysis queue*, minimizing the time lost in switches. Nevertheless, as this case corresponds with a low traffic rate, the system has got enough resources to carry all the incoming traffic and, although the policy is not obviously detrimental, it does not have a real positive effect.

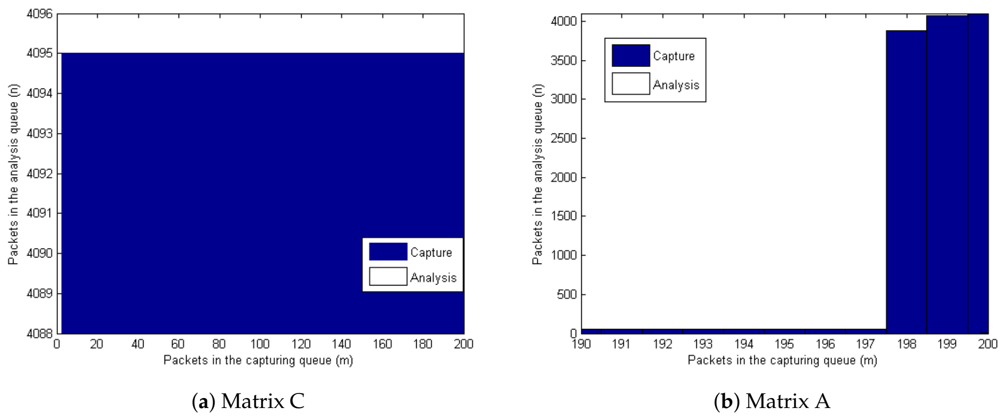
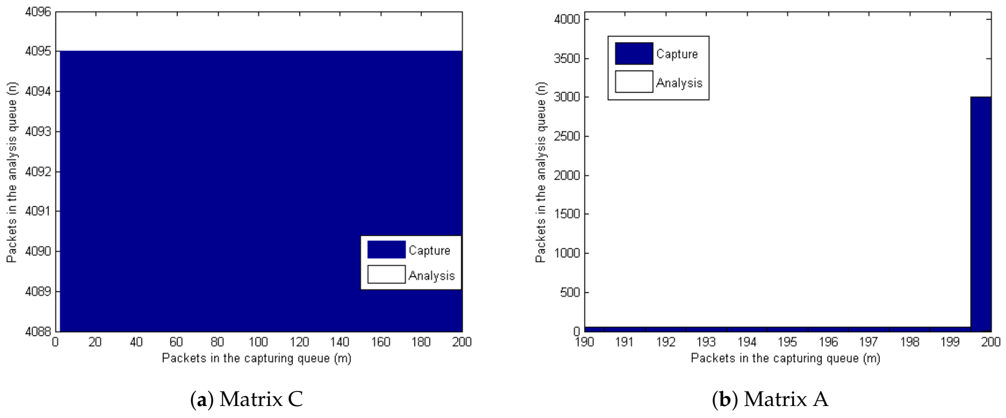


Figure 4. Optimal policy with network traffic rate of 100,000 pps and *null* analysis load.

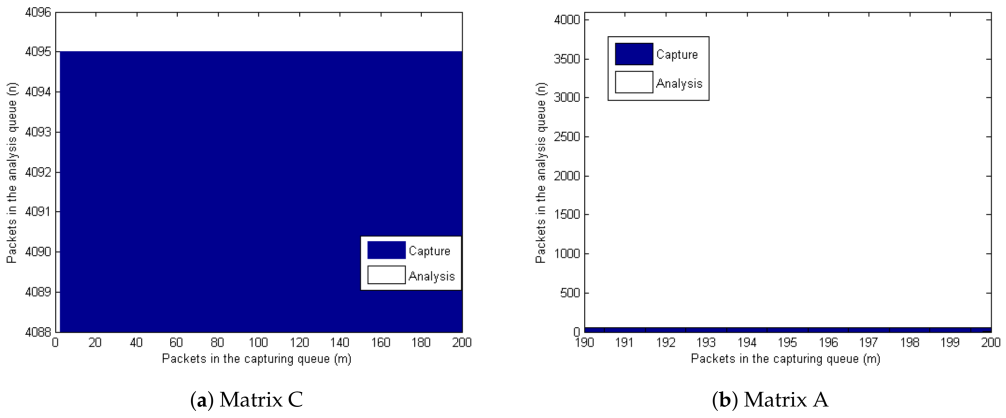
Secondly, Figure 5 shows the structure of the optimal policy when the data traffic rate is intermediate. In particular, it is the case of a network traffic rate of 620,000 pps, *null* analysis load, $M = 200$ and $N = 4096$. When the processor is allocated to the *capturing queue*, as can be see in matrix C of Figure 5a, it is optimal, again, to keep it in that position until the *analysis queue* is full. On the other hand, if the processor's *position* is the *analysis queue*, it is always optimal to keep it analyzing packets unless the *capturing queue* is full ($n = 4096$). If this is the case, the optimal policy is characterized by a threshold value, which tells to the *decision maker* whether the optimal action is to switch the processor's position or not. Although Figure 5 shows the particular case of 620,000 pps, similar structures are obtained with other intermediate traffic rates (between 600,000 and 1,000,000 pps).

As traffic rate increases, the policy defined by matrix C remains and, also, the threshold policy that characterizes the optimal policy defined by matrix A for intermediate traffic rates, but the value of the threshold goes to 0 packets. Figure 6 exhibits this fact for the particular scenario with a traffic rate of 1,100,000 pps, *null* analysis load, $M = 200$ and $N = 4096$. Again, according to matrix C (see Figure 6a), the processor does not switch from capture to analysis until the analysis buffer is full ($n = 4096$). Additionally, matrix A of Figure 6b indicates that it is optimal to keep the processor analyzing packets until the analysis buffer is completely emptied ($n = 0$). The reason is the following: the traffic rate is so high that it is impossible to empty the two buffers at the same time. Thus, the optimal policy is a non-idling policy that minimizes the number of switches because this maximizes the total time that the processor spends either capturing or analyzing packets.



(**a**) Matrix C (**b**) Matrix A

Figure 5. Optimal policy with network traffic rate of 620,000 pps and *null* analysis load.



(**a**) Matrix C (**b**) Matrix A

Figure 6. Optimal policy with network traffic rate of 1,100,000 pps and *null* analysis load.

4.3. Performance of the Optimal Policy

Once we obtain the optimal policy for each scenario, we simulate the dynamics of the queuing network in order to see the analysis throughput obtained following the optimal policy. Results are shown in Figures 7 and 8. On the one hand, Figure 7 shows the analysis throughput obtained from a real network probe based on Linux, if the capturing and the analysis stages are executed according to the standard scheduler of the system. On the other hand, Figure 8 indicates that the performance of the traffic monitoring system can be significantly improved if the *decision maker* allocates the processor according to the optimal policy that we obtain when solving the MDP derived from our tandem queue model. It is observed that the decrease in throughput starting about 500,000 pps with *null* analysis load, 200,000 pps with *0.3 k* and 100,000 pps with *null*, disappears. We even see that the throughput achieves higher values with low and medium traffic rates. Moreover, the analysis throughput remains steady for high values of traffic rate, as opposed to the original traffic monitoring system.

The results presented in Figure 8 are acquired from a discrete event simulation program. Trajectories of packet arrivals, capture, analysis and queuing processes are simulated by the program which receives the following input parameters: the input data rate ($\tilde{\lambda}$), the packet processing rate in the capturing stage ($\tilde{\mu}_1$), the packet processing rate in the capturing stage ($\tilde{\mu}_2$), the size of the capturing buffer (M) and the analysis buffer (N), the switching time (t_{sw}) and the content of the matrices C and A that, according to the optimal policy, point out the action that should be taken in every moment of decision.

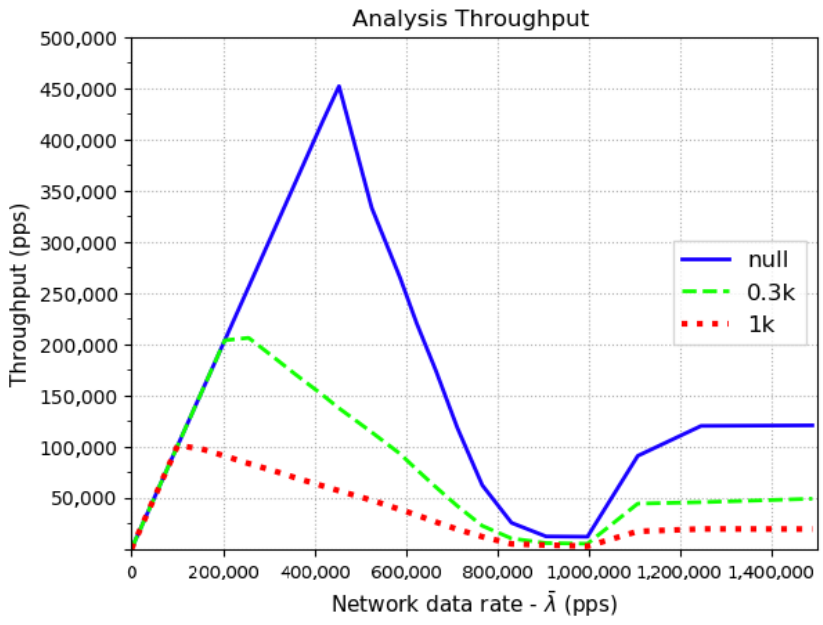


Figure 7. Performance achieved in the real traffic monitoring system.

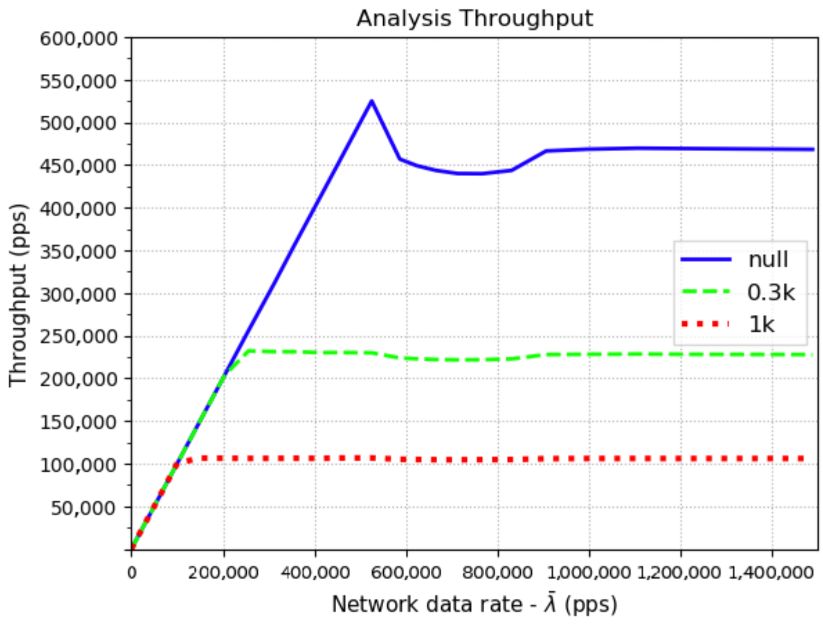


Figure 8. Simulated performance of the optimal policy.

According to the equivalent discrete time model presented in Section 2, the simulation program assumes time slots of length t_s (defined in Equation (4)). The program assumes that the current state $s = (m, n, p)$ is known and, later, in a time slot of t_s, the system goes to the state $s' = (m', n', p')$ according to the transition probabilities of Equations (6)–(10). The program generates events which happen according to the transition probabilities of the model. When the system goes to a state $s' = (m', n', p')$, m' and n' depend on the event generated by the simulator and p' will take the value of the action determined by the optimal policy. If $p = C$, the value of the action will be taken from matrix C and if $p = A$, the value of the action will be taken from matrix A.

Another aspect which is taken into account by the simulator is the time spent in context switches. When the processor is switching from the capture to the analysis or vice versa, i.e., a context switch is taking place, the processor is not in any of the two stages. Therefore, packets are not captured, nor are they analyzed during that time. However, it is possible that new packets arrive at the system. Those packets enter the capture buffer or, in the worst case, if the buffer is full, those packets are dropped and the system throughput is penalized. As mentioned before, the simulation program has got the switching time, t_{sw}, that comes from an experimental measurement. If the switching time is greater than the slot time, $t_{sw} > t_s$ the program divides the switching time into slots of duration t_s and, as before, it studies what happens between the start and the end of the slot, taking into account that the only possible event is the arrival of packets in that slot with probability λ. If $t_{sw} > t_s$, the probability of a new arrival during the switching time is estimated with $\lambda_{switch} = \lambda \cdot \frac{t_{sw}}{t_s}$. If $t_{sw} > t_s$ and t_{sw} is not a multiple of t_s, the remaining time $t_{rem} = t_{sw} - kt_s$ is defined, being k a positive integer, the probability of a new arrival during that remaining time is $\lambda_{rem} = \lambda \cdot \frac{t_{rem}}{t_s}$.

The simulation program controls every variable of the system: m, n, p, M, N. In addition, several counters manage the number of packets analyzed, the number of packet arrivals, the number of time slots and the number of context switches. After simulating what happens in a sufficiently large number of slots, the analysis throughput is computed by dividing the number of the analyzed packets by the simulation's total time. The simulation's total time is the sum of the time of all the slots and the time of all the switches, i.e., $t_{total} = Number\_of\_slots \cdot t_s + Number\_of\_switches \cdot t_{sw}$.

5. Implementation and Validation in a Network Probe

In the previous section, the results of the simulation showed that the policy obtained from the MDP model brings along with it an improvement in terms of analysis throughput. Now, the next step is to implement that optimal policy in a real probe. To this end, we select the same probe which was used in the experimental study (see Figure 3).

5.1. Policy Based on a Threshold Value

As explained in Section 3, the optimal policy depends on the number of packets in the capture stage (m), the number of packets in the *analysis queue* (n) and the current *position* of the processor (i.e., if the processor is running the capture stage or the analysis one at that time). Keeping track of all these variables is not easy in a real Linux-based probe. While the number of packets in the *analysis queue* can be controlled in a simple way, on the other hand, it is not the same in the case of the control of the number of packets in the *capture queue*.

If we take a look at the architecture of the Linux networking subsystem [9,30], parameter m measures the number of packets in the capturing queue. The packet capturing process in Linux starts when the first packet arrives to the NIC. In this moment, the NIC fires a hardware interrupt request (*hardirq*). The *hardirq*'s Interrupt Service Routine (ISR) disables the *hardirqs* and schedules a *softirq*. Meanwhile, the first packet and those that are arriving are being copied to the capturing queue by the DMA without any processor consumption. The *softirq*'s ISR takes the packets from the DMA queue to the *capturing queue* of the probe.

In accordance with the explanation given in the previous paragraph, the number of packets that are in the DMA, measured by parameter m, is continuously changing. On the one hand, the *softirq* takes packets from this queue. This action can be controlled easily because it is performed by a software executed by the processor. On the other hand, the DMA copies the packets that arrive to the NIC to the DMA queue without any processor intervention. This action cannot be controlled because it is performed by a hardware resource that works independently from the processor. The state of the queue is known but the measure is not precise because when this value is read, the state of the queue can be different.

Due to the difficulties that appear when monitoring parameter m in the real system, it is interesting to consider the analysis of the structure of the optimal policy in order to propose an affordable implementation which is the most consistent possible with the optimal policy. The conclusion extracted from that study (see Section 4.2) is that the more the network data rate increases, the more the optimal policy approaches a threshold value.

At this point, we can define several threshold based policies for the case when the processor is in the analysis stage, such as:

- Policy based on a threshold T_m which depends on m. It works as follows:

```
if $m \geq T_m$ then
    $action \leftarrow CAPTURE$
else
    $action \leftarrow ANALYSIS$
end if
```

- Policy based on a threshold T_n which depends on n. It works as follows:

```
if $n \leq T_n$ then
    $action \leftarrow CAPTURE$
else
    $action \leftarrow ANALYSIS$
end if
```

- Step-wise policy which depends on m and n. It works as follows:

```
if $(n = 0)$ or $(m \geq T_m$ and $n \leq T_n)$ then
    $action \leftarrow CAPTURE$
else
    $action \leftarrow ANALYSIS$
end if
```

The simulation program used to evaluate the performance of the optimal policy can also foresee the results of the threshold based policies. For example, Figure 9 shows the analysis throughput obtained when an m-based threshold policy with value $T_m = 196$ is used. It is visible in the graph that the policy with T_m is not beneficial, on the contrary, it is detrimental, since, from a network rate of around 500,000 pps, the throughput is very low.

Figure 10 shows an example of an n-based threshold policy with $T_n = 3000$. In contrast with the m-based threshold policy, this policy achieves a considerable throughput improvement whose form is similar to the optimal policy's one, i.e., it prevents the throughput decrease in the range of intermediate rates, and it manages to increase the throughput at the highest rates.

It is also possible to simulate the behavior of a step-wise policy. Figure 11 presents an example of this type of policy with $T_m = 198$ and $T_n = 2048$. The results are positive too, but its implementation will be more difficult compared to the n-based policy.

From the point of view of the implementation, the policies based on a threshold value are feasible, in particular, those ones whose threshold value depends on the parameter n. If the processor is allocated to the capturing task, there is no switch until the *analysis queue* is full or the *capturing queue* is empty. This happens for all the values of network data rate, thus, here, we identify a threshold policy for the values $n = N$ and $m = 0$. However, if the processor is allocated to the analysis process, the moment when the processor switches to the capturing task is different depending on the network data rate and the number of packet in both buffers (m and n). Let us see it in the context of three ranges of network traffic rate.

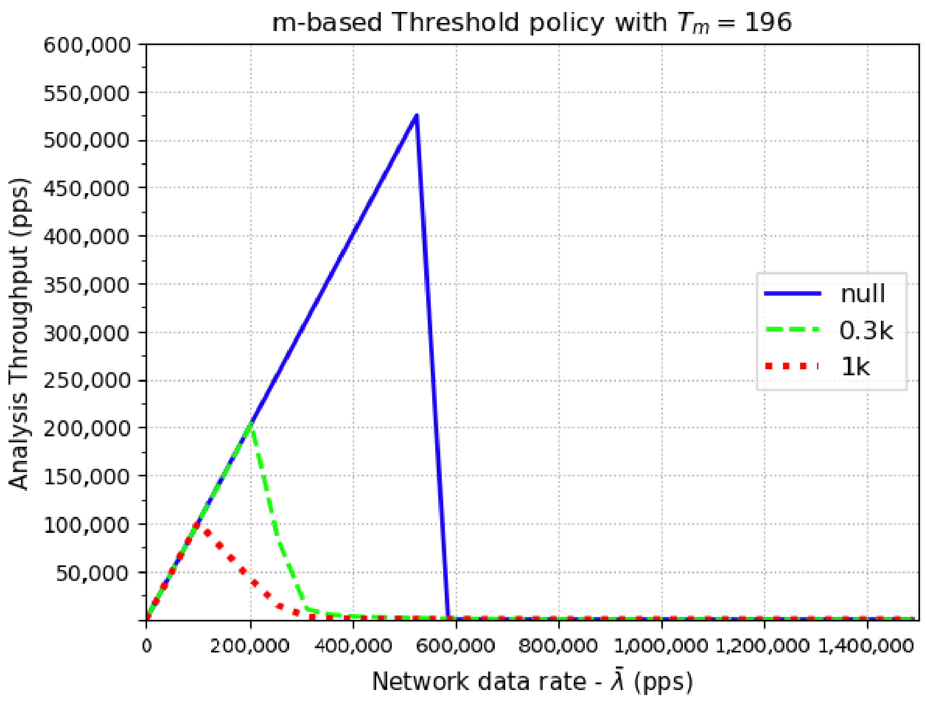


Figure 9. Performance simulation for m-based threshold policy.

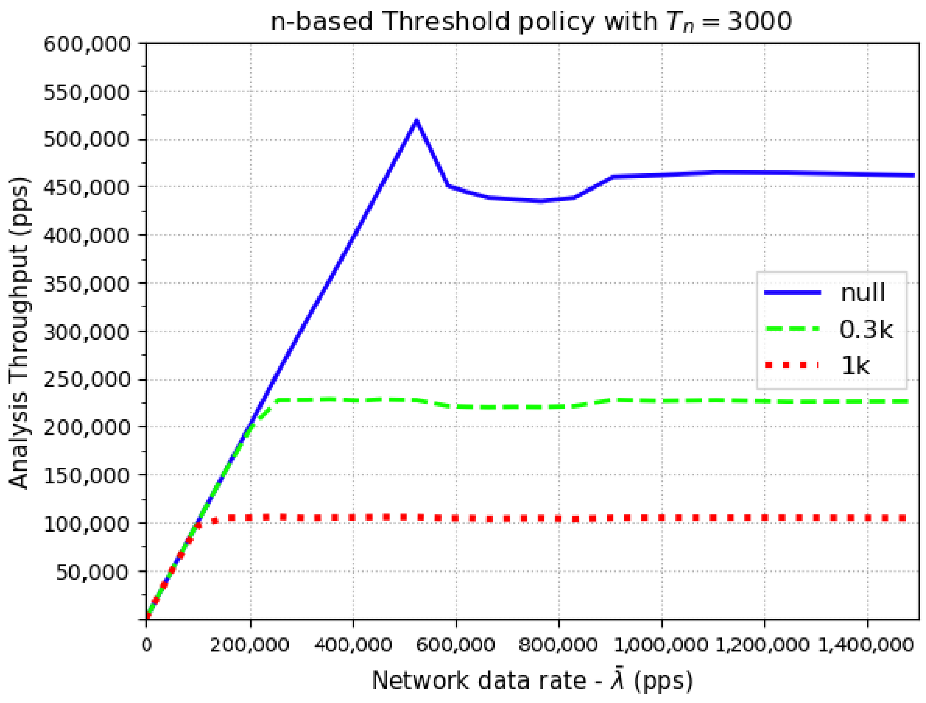


Figure 10. Performance simulation for n-based threshold policy.

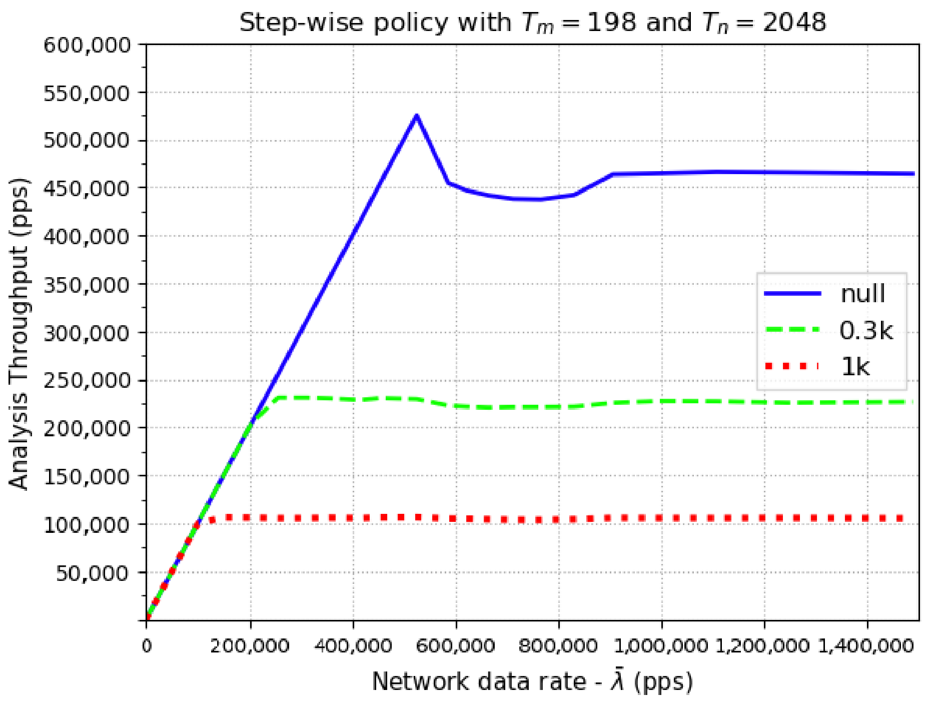


Figure 11. Performance simulation for step-wise policy.

First, with low traffic rates, the optimal policy tells us that while the analysis buffer is not empty, there is no switch from analysis to capture until the capturing buffer is almost full and the number of packets in the analysis buffer is less than a given value. As seen in Figure 4, for the network data rate of 100,000 pps and *null* analysis load, the probability of $m > 197$ is very low and, as a result, it is possible to set the threshold value of $n = 0$ which is the common value proposed by the optimal policy for every $m < 197$. However, as data network rate increases, the probability of having full or almost full capture buffer also grows. In these cases, the previous threshold value of $n = 0$ could not be appropriate because it would not avoid the risk of dropping packets at the entrance of the capture buffer.

Thus, secondly, with intermediate traffic rates, it is necessary to consider the case when the processor is analyzing packets, the capture buffer is almost full and the processor must change from analysis to capture in order to analyze the 100% of the incoming traffic without dropping any packet at the capturing entrance. For example, for the network data rate of 525,000 pps and *null* analysis load, the obtained policy indicates that the processor has to change from analysis to capture when m is greater than or equal to 194; otherwise, the analysis task does not finish until the analysis buffer is empty. If $m \geq 194$, the n-value of the switching point (m, n) differs from one case to another. Specifically, we have the following pairs: $(m = 194, n = 3609)$, $(m = 195, n = 3840)$, $(m = 196, n = 3910)$, $(m = 197, n = 3951)$, $(m = 198, n = 3978)$, $(m = 199, n = 3996)$, $(m = 200, n = 4010)$. As seems logical, the greater the occupation of the capture buffer, the sooner the switch should take place (that is, with a greater value of n). However, as mentioned previously, the implementation of a policy based on both parameters, m and n, is difficult. For this reason, we opt for fixing a policy based on a threshold whose value depends only on n.

Then, with higher intermediate traffic rates, for example, 620,000 pps, as can be seen in Figure 5, the optimal policy is characterized by a threshold value because the processor only switches from the analysis to the capture when the capture buffer is full. Thus, the threshold value is $m = M$, but, as before, given the difficulties to have monitored the occupation of the capture buffer, it can be translated to an n-based threshold value.

Finally, with the highest data network rates, the optimal policy is a non-idling policy that can be totally expressed in terms of n-based threshold values. For example, as can be seen in Figure 6 for the network data rate of 1,100,000 pps, the processor switches from analysis to capture when the analysis buffer is emptied. Thus, in this case, the threshold value is $T_n = 0$ for all the values of m which are in the range $0 \leq m \leq M$.

5.2. Comparing Optimal and Real Analysis Throughputs

Given what we have just seen, a control mechanism is introduced into the real probe. It works as follows: every time a new packet is introduced into the *analysis queue* (the buffer of the monitoring application), the system checks the number of packets in this queue and it does not switch to analyze packets as long as the *analysis queue* is not full and the capturing buffer is not empty. This follows exactly the result of the optimal policy contained in the matrix C. When the *analysis queue* is full, this means that the capture threshold is reached and the system disables the packet capturing. To do that, *hardirqs* and NIC polling are disabled. Hence, in that moment, the analyzing instance will process the packets in the queue. This will happen until the packet amount reaches the analysis threshold that has been previously estimated as T_n. In that moment, the packet capture task is enabled again so that it can start again.

When implementing this control mechanism, we can state that two thresholds have to be set: the maximum threshold, T_{max}, as the maximum size of the *analysis queue* ($T_{max} = N$) and the minimum threshold as $T_{min} = T_n$.

Our test setup (see Figure 3) allows us to implement the threshold policy based on $T_{max} = N$ and $T_{min} = T_n$ due to the fact that we can control the network data rate managing properly the traffic generator. Figure 12 presents the analysis throughput of the real probe with the implemented control mechanism for different network data rates and analysis loads. We can see that the obtained result is very close to the simulated performance of the optimal policy (Figure 8).

In Figure 13, we show the difference between the optimal policy and the implemented one in terms of throughput efficiency, assuming that the optimal policy achieves 1.0 (100%). The result tells us that even if the implemented policy does not follow the optimal one so closely, the difference between both of them is not bigger than 2% in all of the range of network data rates. Figure 13 also shows results of step-wise, T_n- and T_m-based policies assessed by the simulation tool. In the case of T_n-based policy, it is also very close to the real probe's one. Thus, we can conclude that the real implementation is acceptable, as well as that the simulation tool reproduces the behavior of the real system.

Another aspect considered for the implementation in the real probe is the variability of the proposed threshold, T_n, with the network data rate. As mentioned previously, in our experimental scenario with a source of synthetic traffic, it is feasible to conduct tests varying the parameter T_n according to a *controlled* network data rate. However, in a real environment, it can be complex to control the data flow speed and adapt the n-threshold policy simultaneously. For that reason, we implement a threshold policy with the same value of T_n for all the network data rates and analysis loads. In this way, we test different values of T_n such as 512, 1024, 2048 and 3000. Figure 14 presents a comparison in terms of throughput efficiency with respect to the optimal policy for those settings. The most significant result is the performance improvement achieved when introducing the control mechanism over network data rates that overload the system. Obviously, the optimal policy is the best and, although the implemented policy in the real system does not reach it, the obtained result is very acceptable. The performance is improved until 80% for some network data rates. In addition, the reliability of the developed simulation tool for the cases which have been able to achieve a real implementation is remarkable.

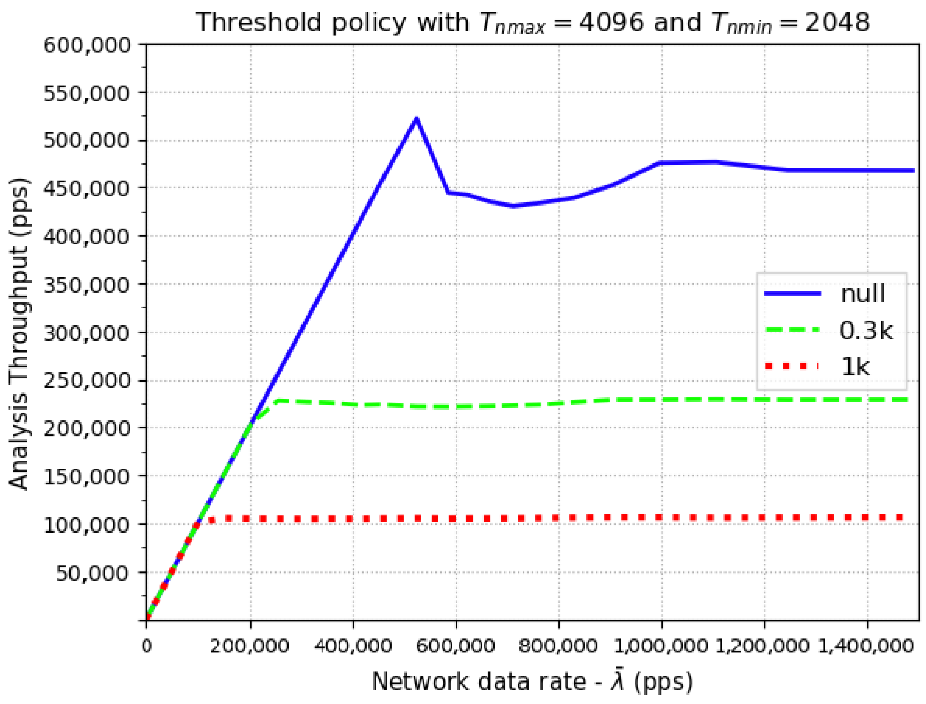


Figure 12. Performance of the real probe with the control mechanism.

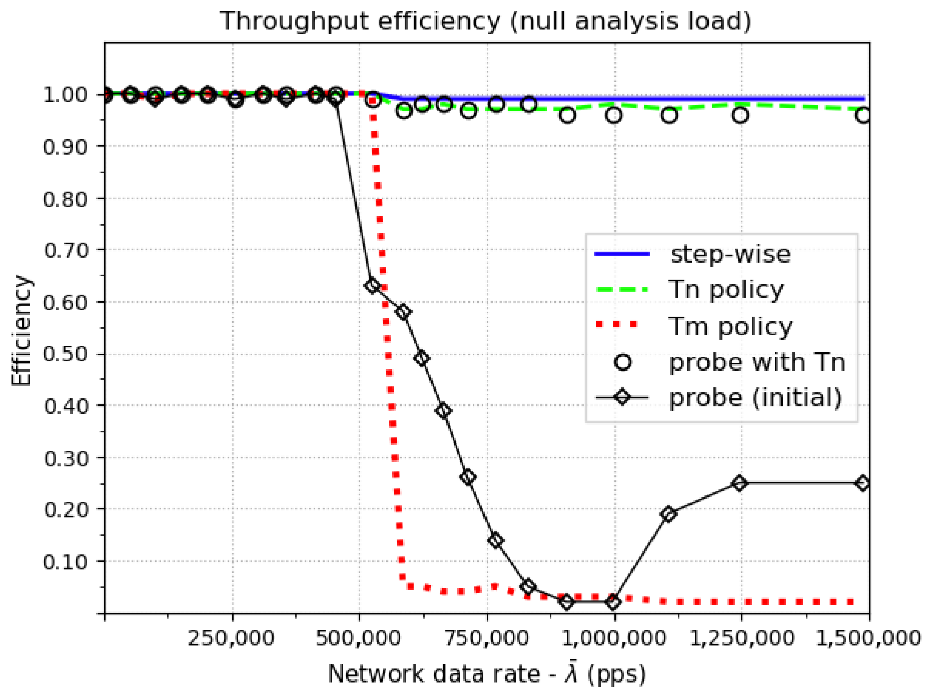


Figure 13. Comparison in terms of throughput efficiency with respect to the optimal policy.

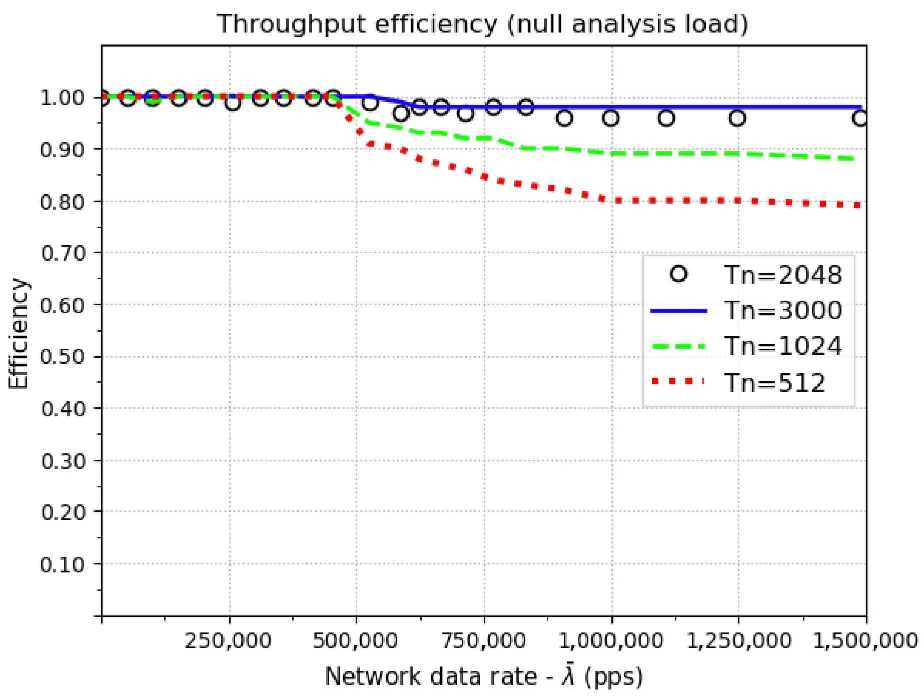


Figure 14. Performance comparison between implemented scenarios on the real probe with n-based threshold policies.

6. Conclusions

This paper presents an MDP model for a network traffic monitoring system based on commodity hardware in order to improve the performance in terms of analysis throughput. First, we identified the performance problem observed in a real network traffic probe when the network data rate increases. It happened for the different analysis loads that the probe supports in the preliminary empirical performance evaluation (see Figure 7).

Then, we propose a model that consists of a two-stage tandem queue attended by a moving server. In this framework, a three-dimensional MDP is formulated in which the objective is to determine the position of the server in each time slot to improve the throughput of the queuing system. We have analyzed the structure of the optimal policy for a wide range of scenarios where we show that it has a threshold-type behavior.

We have also considered a real probe so as to validate our theoretical findings. First, we analyzed several threshold type policies and we also simulated the optimal policy using parameters from the real probe system. These experiments show the improvement of the optimal policy in terms of throughput. Furthermore, we considered the real system in which we investigate the performance of the threshold-based policy, which is close to the optimal one, and allows to demonstrate the improvement derived from our mathematical model.

As future work, we aim to adapt some of the fundamentals applied in the modeling of this work to 5G network environments. We believe that it is appropriate to represent with queuing models the computational consumption of the different processing-stages or VNFs. In addition, it is common to approach limited-resource applications in real information systems on current 5G mobile networks. For instance, more and more multimedia applications require intensive edge computing in network nodes where there are limited resources. The proposal presented in this work identifies which strategy could be optimal to adapt the available resources to the state of the traffic capture task and the analysis needs. Additionally, there is a current trend towards distributed computing resources that is applied to network monitoring systems, the infrastructure of networks themselves and new virtualized services. Therefore, within these contexts, there are proposals that determine how to conduct resource placement in a dynamic and efficient way (see, for instance, [22]). In this sense, we would like to extend this work based on an MDP model in that research line, by considering ubiquity issues on distributed computing systems, virtualized network services and edge computing on current telecommunication infrastructure.

Author Contributions: Conceptualization, L.Z., J.D. and A.F.; methodology, L.Z., J.D. and A.F.; software, L.Z. and A.F.; validation, L.Z. and A.F.; formal analysis, L.Z., J.D. and A.F.; investigation, L.Z., J.D. and A.F.; resources, L.Z., J.D. and A.F.; data curation, L.Z., J.D. and A.F.; writing—original draft preparation, L.Z., J.D. and A.F.; writing—review and editing, L.Z., J.D. and A.F.; visualization, L.Z., J.D. and A.F.; supervision, L.Z., J.D. and A.F.; project administration, A.F.; funding acquisition, A.F. All authors have read and agreed to the published version of the manuscript.

Funding: This research was partially supported by the Department of Education of the Basque Government, Spain through the Consolidated Research Groups NQaS (IT1635-22) and MATHMODE (IT1456-22), by the Marie Sklodowska-Curie, Spain grant agreement No 777778, by the Spanish Ministry of Science and Innovation, Spain with reference PID2019-108111RB-I00 (FEDER/AEI), by grant PID2020-117876RB-I00 funded by MCIN/AEI (10.13039/501100011033) and by Grant KK-2021/00026 funded by the Basque Government.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Pramanik, A.; Sarkar, S.; Maiti, J. A real-time video surveillance system for traffic pre-events detection. *Accid. Anal. Prev.* **2021**, *154*, 106019. [CrossRef] [PubMed]
2. Liao, H.J.; Lin, C.H.R.; Lin, Y.C.; Tung, K.Y. Intrusion detection system: A comprehensive review. *J. Netw. Comput. Appl.* **2013**, *36*, 16–24. [CrossRef]
3. Abdel-Gawad, H.I.; Baleanu, D.; Abdel-Gawad, A.H. Unification of the different fractional time derivatives: An application to the epidemic-antivirus dynamical system in computer networks. *Chaos Solitons Fractals* **2021**, *142*, 110416. [CrossRef]
4. Aktas, M.S. Hybrid cloud computing monitoring software architecture. *Concurr. Comput. Pract. Exp.* **2018**, *30*, e4694. [CrossRef]
5. Schneider, F.; Wallerich, J.; Feldman, A. Packet Capture in 10-Gigabit Ethernet Environments Using Contemporary Commodity Hardware. In Proceedings of the 8th International Passive and Active Measurement Conference, PAM 2007, Louvain-la-neuve, Belgium, 5–6 April 2007; Springer: Berlin/Heidelberg, Germany, 2007; pp. 207–217.
6. Ntop Project. Available online: http://www.ntop.org (accessed on 19 December 2022).
7. Pereira, R.I.; Dupont, I.M.; Carvalho, P.C.; Jucá, S.C. IoT embedded linux system based on Raspberry Pi applied to real-time cloud monitoring of a decentralized photovoltaic plant. *Measurement* **2018**, *114*, 286–297. [CrossRef]
8. Jo, E.; Yoo, H. Implementation of cloud monitoring system based on open source monitoring solution. In *Software Engineering in IoT, Big Data, Cloud and Mobile Computing*; Springer: Berlin/Heidelberg, Germany, 2021; pp. 181–190.
9. Freitas, E.; de Oliveira Filho, A.T.; do Carmo, P.R.; Sadok, D.; Kelner, J. A survey on accelerating technologies for fast network packet processing in Linux environments. *Comput. Commun.* **2022**, *196*, 148–166. [CrossRef]
10. Bovet, D.; Cesati, M. *Understanding the Linux Kernel, Third Edition*; O'Reilly Media: Sebastopol, CA, USA, 2005.
11. Fusco, F.; Deri, L. High Speed Network Traffic Analysis with Commodity Multi-core Systems. In Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement, IMC'10, Melbourne, Australia, 1–3 November 2010; ACM: New York, NY, USA, 2010; pp. 218–224. [CrossRef]
12. Moreno, V.; Del Rio, P.M.S.; Ramos, J.; Garcia-Dorado, J.L.; Gonzalez, I.; Arribas, F.J.G.; Aracil, J. Packet storage at multi-gigabit rates using off-the-shelf systems. In Proceedings of the 2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS), Paris, France, 20–22 August 2014; IEEE: Piscataway Township, NJ, USA, 2014; pp. 486–489.
13. Trevisan, M.; Finamore, A.; Mellia, M.; Munafo, M.; Rossi, D. Traffic Analysis with Off-the-Shelf Hardware: Challenges and Lessons Learned. *IEEE Commun. Mag.* **2017**, *55*, 163–169. [CrossRef]
14. Wu, W.; Crawford, M.; Bowden, M. The performance analysis of Linux networking – packet receiving. *Comput. Commun.* **2007**, *30*, 1044–1057. [CrossRef]
15. Salah, K.; Elbadawi, K.; Boutaba, R. Performance modeling and analysis of network firewalls. *IEEE Trans. Netw. Serv. Manag.* **2012**, *9*, 12–21. [CrossRef]
16. Li, X.; Ren, F.; Yang, B. Modeling and analyzing the performance of high-speed packet I/O. *Tsinghua Sci. Technol.* **2021**, *26*, 426–439. [CrossRef]
17. El Kafhali, S.; Salah, K. Performance analysis of multi-core VMs hosting cloud SaaS applications. *Comput. Stand. Interfaces* **2018**, *55*, 126–135. [CrossRef]

18. El Kafhali, S.; Salah, K. Performance modelling and analysis of Internet of Things enabled healthcare monitoring systems. *IET Netw.* **2019**, *8*, 48–58. [CrossRef]
19. Bolla, R.; Bruschi, R.; Carrega, A.; Davoli, F. Green networking with packet processing engines: Modeling and optimization. *IEEE/ACM Trans. Netw.* **2014**, *22*, 110–123. [CrossRef]
20. Ibrahim, A.G.M.; Khedr, M.E.; Shaheen, M. Power Consumption of Packet Processing Engines and Interfaces of Edge Router: Measurements and Modeling. In Proceedings of the ICNS 2016: The Twelfth International Conference on Networking and Services, Lisbon, Portugal, 26–30 June 2016.
21. Prados-Garzon, J.; Ameigeiras, P.; Ramos-Munoz, J.J.; Navarro-Ortiz, J.; Andres-Maldonado, P.; Lopez-Soler, J.M. Performance modeling of softwarized network services based on queuing theory with experimental validation. *IEEE Trans. Mob. Comput.* **2019**, *20*, 1558–1573. [CrossRef]
22. Agarwal, S.; Malandrino, F.; Chiasserini, C.F.; De, S. VNF placement and resource allocation for the support of vertical services in 5G networks. *IEEE/ACM Trans. Netw.* **2019**, *27*, 433–446. [CrossRef]
23. Faraci, G.; Lombardo, A.; Schembra, G. A building block to model an SDN/NFV network. In Proceedings of the 2017 IEEE International Conference on Communications (ICC), Paris, France, 21–25 May 2017; pp. 1–7.
24. Leland, W.; Taqqu, M.; Willinger, W.; Wilson, D. On the self-similar nature of Ethernet traffic. *IEEE/ACM Trans. Netw.* **1994**, *2*, 1–15. [CrossRef]
25. Puterman, M.L. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*; John Wiley & Sons: Hoboken, NJ, USA, 2014.
26. Bertsekas, D. *Dynamic Programming and Optimal Control: Volume I*; Athena Scientific: Nashua, NH, USA, 2012; Volume 1.
27. Munoz, A.; Ferro, A.; Liberal, F.; Lopez, J. A Kernel-Level Monitor over Multiprocessor Architectures for High-Performance Network Analysis with Commodity Hardware. In Proceedings of the 2007 International Conference on Sensor Technologies and Applications (SENSORCOMM 2007), Valencia, Spain, 14–20 October 2007; pp. 457–462.
28. Pineda, A.; Zabala, L.; Ferro, A. Network architecture to automatically test traffic monitoring systems. In Proceedings of the Mosharaka International Conference on Communications and Signal Processing (MIC-CSP2012), Barcelona, Spain, 6–8 April 2012.
29. Endace Ltd. Available online: https://www.endace.com (accessed on 11 January 2023).
30. Benvenuti, C. *Understanding Linux Network Internals*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2006.