POLITECNICO DI TORINO

Master Degree course in Communications and Computer Networks Engineering

Master Degree Thesis

# Real-time flow monitoring based on time-space probabilistic data structures


**Supervisor**
Prof. Paolo Giaccone

**Candidate**
Simone Porcino


Academic Year 2022-2023

# Acknowledgements

In primis desidero ringraziare il mio relatore Paolo Giaccone che, sin dalla scelta dell'argomento, mi ha guidato costantemente nella stesura dell'elaborato. Non meno importante è stato l'aiuto di Alessandro Cornacchia, pronto ad assistermi quando ne ho avuto bisogno. La loro disponibilità ed i loro preziosi suggerimenti sono stati fondamentali per portare a termine questo lavoro.

Un ringraziamento speciale vorrei dedicarlo alla mia famiglia, fonte essenziale di supporto, forza ed incoraggiamento. Ringrazio in particolar modo i miei genitori, che mi hanno dato la possibilità di proseguire gli studi a Torino, in un'università prestigiosa come il Politecnico. Spero di ricambiare i loro sacrifici e l'impegno che hanno dedicato per farmi diventare quello che sono oggi. Questo traguardo non sarebbe stato possibile senza di loro, motivo per cui desidero dedicargli questa tesi.

Ci tengo a ringraziare affettuosamente anche la mia fidanzata Valentina, altra fonte di supporto, forza ed incoraggiamento. Aver remato nella mia stessa direzione è stato indubbiamente un'altra chiave di questo successo.

Ringrazio anche tutte quelle persone che, con parole e gesti, hanno contribuito a trasformare preoccupazione e ostacoli in sollievo e agevolazioni.

Un ultimo ringraziamento va a coloro che ho incontrato durante il percorso universitario, dai colleghi ai coinquilini: ciascuno ha lasciato un segno che porterò per sempre con me.

**Abstract**

Network monitoring is essential to ensure network performance, integrity and security. With the advent of Software-Defined Networking (SDN), network monitoring solutions can be implemented in programmable switches guaranteeing packet processing at link-rate. However, the limited hardware resources of programmable switches along with the growth of link rates and simultaneous number of flows make real-time traffic monitoring a challenge, requiring solutions that are memory efficient and, possibly, perform operations on a time scale very reduced. To this end, we propose a solution to passively and continuously monitor the flow round-trip time (RTT) through a probabilistic data structure composed of a set of Bloom filters that are evenly spaced across time slots over a given time window, with the aim of optimizing the memory footprint of the Bloom filter array while still guaranteeing accuracy in RTT measurements. A problem to be addressed is the management of false positive events typical of the Bloom filter, for which we propose different search policies to select one Bloom filter among those that return a true answer after being queried. Another problem concerns the management of the arrival time of the outgoing packets necessary to estimate the RTT, since it can not be stored within the Bloom filters. To evaluate such a system, for each search policy, we conducted simulations using two traffic patterns and several configurations in terms both of memory and time slot width to individuate the configuration that minimizes the error and guarantees the best accuracy. We show that the best space-time configuration changes with both the traffic characteristic and the search policy, but selecting the more recent Bloom filter in the past always provides the best accuracy, although it may introduce underestimated RTT samples due to an intrinsic problem of the system itself.

# Contents

# Chapter 1

# Introduction

Network monitoring refers to a set of actions (e.g. measurements, reports, analyses, etc.) that allow network administrators to assess the network health, which is crucial to ensure performance, integrity and security. In particular, network administrators must adopt solutions to proactively monitor the traffic network to identify, isolate and correct network defects, but also to avoid failures [4], whose consequences could be catastrophic because they can lead to business disruptions and to loss of reputation, resulting in loss of customers and revenue [2].

Network monitoring techniques can be *active*, *passive* or *hybrid*. First ones are carried out on end-hosts that actively send *probes* over the network, i.e. fictitious packets that are properly shaped to obtain desired info from the network. Second ones consist of passively observing traffic passing through a measurement node (either an end-host or, more frequently, a network device at a vantage point), which is in charge of collecting and processing data to estimate network characteristics. Last, hybrid monitoring measurements are based on the joint use of active and passive measurements.

The main drawback of active measurements is that they are network-invasive because require the generation of ad-hoc traffic, resulting in performance and management overheads at network nodes, altering the normal behaviour of the network. In addition, with active measurement tools it is only possible to collect end-to-end path samples, as opposed to passive monitors that allows you to observe the whole network traffic passing through the measurement point [13], providing a more accurate picture of network performance. On the other hand, passive approaches pays for this higher accuracy in terms of the volume of data to be collected and processed, requiring more performing devices. Furthermore, since hybrid measurements use both active and passive techniques, it inherits their pros and cons.

Regardless of the adopted measurement approach, administrators typically monitor networks with a special focus to *Quality of Service (QoS)*, i.e. a model according to which the quality of services offered by the network to end users can be classified through some parameters, such as throughput, packet losses and delay. One possible metric to assess network performance in terms of delay is the *Round-trip time (RTT)*. Indeed, since it represents the time that elapses from when a host transmits a packet to when it receives the related response, if it assumes a high value, it could be a sign of some anomaly within

the network. For instance, some RTT monitoring use-cases can be [6, 17]:

- detecting network congestion. When traffic crosses a congested path and starts to degrade network performance, the network can reroute this traffic to an alternate path which is less congested.
- guaranteeing QoE (Quality of Experience). Some applications such as video streaming and multiplayer cloud-gaming are latency-sensitive. Therefore, monitoring RTT can help to select an alternate path or server to better accommodate customers' demand.
- detecting routing attacks. Both BGP routing attacks and IP spoofing exhibit unexpected changes in the RTT, as the traffic is rerouted to pass through the malicious attacker. Hence, RTT can be used to potentially discover these attacks and to block the affected traffic.

With the advent of Software-Defined Networking (SDN), the way of performing network monitoring has changed. Thanks to its approach, which relies on a logical centralized control plane which is decoupled from the data plane, the controller collects information from every forwarding device in order to acquire a global network view, which is essential to keep it under control and intervene in real-time in case of some event. However, this decoupling involves a simpler data plane, represented by commodity switches which are cheaper and have limited amount of resources. The data plane can follow either a *stateless* or a *stateful* approach. In the first case, the switch does not keep any network state information, restricted only to forward the traffic according to the decisions taken by the controller. In the second case, some simple processing capabilities (e.g. finite state machines) are implemented in the switch in order to keep some state information useful to make decisions on the fly without necessarily going through the controller. Stateless switches, like OpenFlow, are not suited for monitoring large network data flows [7] because degrade performance. Their main limitation comes from the fact that all the flows must be monitored by the controller: the exchange of signaling information between data and control plane not only introduces some latency but also overloads some network links and the controller processing (which can in turn affect that of the data plane). Stateful switches overcome this limitation: not only they allow to balance the amount of processing between data and control plane, but they also improve network reactivity, given that they enable to take local decisions and to reduce signaling overhead. Stateful switches are also known as *programmable switches* because they introduce the possibility to program the data plane using an high-level language (e.g. P4[1]) to perform any network functionality you are interested to. These network applications ranges from traditional network functions (e.g. switching and routing) to traffic engineering (e.g. load balancing and QoS), from network monitoring to security domains (e.g. firewalling, network access control, DoS attack mitigation) [16]. For instance, network operators may be interested in monitoring flow RTT to trigger some traffic engineering: flows that experience high

[1] https://p4.org/

RTT values may go through a path with some congested node, therefore you would like to re-route them in real time along a different path in order to achieve some operational goal (e.g. QoS, QoE). Such an innovative and flexible technology has found particular interest in the field of networking, as the implementation of several applications directly in the switch fabric allows you to promptly react at line-rate packet processing in the presence of anomalies.

## 1.1 Motivation

The flexibility of programmable switches comes up against its hardware limitations in terms of computational and memory resources, whose efficiently use is a challenge for real-time network monitoring. In fact, today's networks are characterized by a large number of simultaneous flows at very high link-rate, therefore real-time traffic monitoring requires programmable switches that perform operations on a time scale very reduced and, possibly, with a small amount of memory to store information traffic related to the metric considered. To this end, programmable switches could be equipped with probabilistic data structure. Unlike traditional data structures, which always return true answers to queries about the stored data, probabilistic ones return approximate answers, in the sense that they can be true with a certain probability they are not. Despite this property, which involves the introduction of some errors in measurements, probabilistic data structures have the main benefit of being very performant, requiring low memory footprint and low query time. There are already solutions that passively and continuously monitor the RTT of data flows within the network. For instance, those based on Hash tables provide highly accurate estimates by storing the entire timestamp, thus requiring more memory than solutions based on Bloom filters. The latter, on the other hand, sacrifice some accuracy to save memory, as they are implemented by simple bit arrays or counter arrays. In this thesis work, we propose a solution that aims to passively and continuously monitor the flow RTT through a probabilistic data structure composed of a set of Bloom filters and to possibly individuate a trade-off between its accuracy in measuring the RTT samples and the amount of dedicated memory, by means of simulations of a simple testbed scenario.

## 1.2 Thesis structure

The remainder of this thesis is organized as follows. First, Chapter 2 provides the main theoretical concepts necessary for a better understanding of this work, namely the RTT and the probabilistic data structures, along with a tool they are based on (hash function) and a specific data structure on which our proposal is based (Bloom filter). In Chapter 3, we present in detail our idea applied in a specific scenario, explaining the problems that need to be managed when measuring the flow RTT. We also propose some possible solutions to them, accompanied by the relative pseudo-codes, in case they should be used for further analysis or in other contexts. Then, Chapter 4 first presents a quick overview of the methodology used to evaluate the accuracy of our solution, followed by the resulting outcomes. Finally, in Chapter 5 we draw our conclusions on the obtained results and propose some ideas that can be further explored in future works.

# Chapter 2

# Theoretical background

This chapter introduces the main pillars that are required for a better understanding of this work. Therefore, we will first define the RTT along with all the different types of delays that contribute to it. Next, we will go through probabilistic data structures, providing both a quick classification of them and a brief description of the main tool they are based on, namely hash functions. Finally, we will present one of the most popular probabilistic data structures, i.e. the Bloom filter, which is the main block for building the data structure on which our proposal is based.

## 2.1 Round-trip time (RTT)

The round-trip time (or RTT) is the amount of time it takes to transmit a packet from the source and receive a response from the destination:

$$RTT = \textit{(time when the response is received) - (time when the packet is sent)} \quad (2.1)$$

Behind (2.1) are hidden different types of delays that contribute to RTT, which are shown in Fig. 2.1 in a point-to-point communication with no networks in between. For the following definitions, we refer to [14].

**Processing delay ($\mathbf{t_{proc}}$)** It is the time required by a network node to process a packet. It is known that network nodes follow a layered protocol architecture (e.g. OSI model) to communicate with each other: whereas at the sender side the packet is received from an upper-layer protocol, at the receiver side the packet is received from a lower-layer protocol. What matters is that each layer introduces a processing delay to support a specific functionality for the packet, such as framing (or packetization), data compression, encryption/decryption, lookup forwarding tables, and so on. Processing delay depends on the device processor speed.

**Transmission delay ($\mathbf{t_{tx}}$)** It is the time that takes the sender to push all the packet's bits into the transmission medium. This delay is a function of the packet length $L$

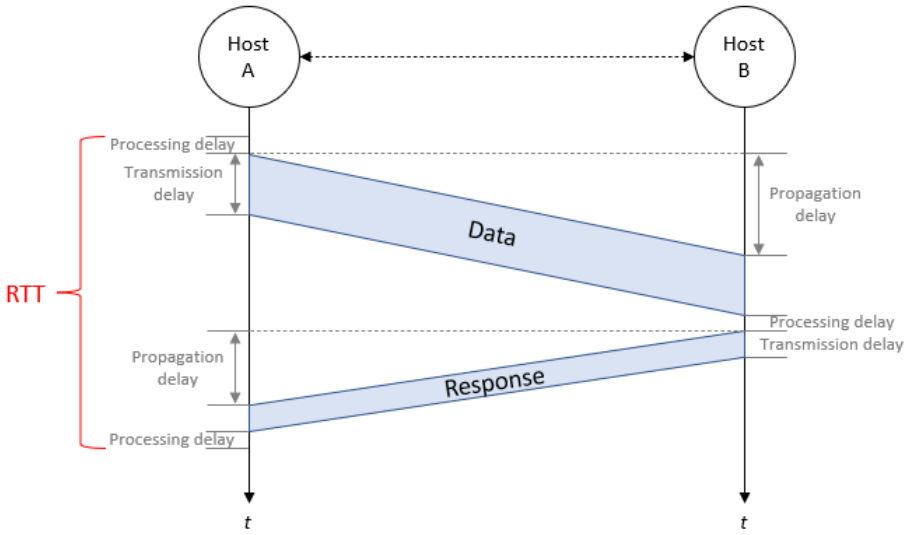



Figure 2.1: Delay components that contribute to RTT in a point-to-point communication with no networks in between

(in bits) and the transmission rate $R$ of the medium (in bits per second or bps):

$$t_{tx} = \frac{L}{R} \tag{2.2}$$

**Propagation delay ($t_{prop}$)** It is defined as the time elapsed to transmit a bit from the sender to the receiver. It is given by the ratio between the travelled distance $d$ and the speed of electromagnetic waves $s$ in the transmission medium:

$$t_{prop} = \frac{d}{s} \tag{2.3}$$

The speed of electromagnetic waves is $s = c/n$, so it is proportional to the speed of light in vacuum ($c \simeq 3 \times 10^8$ m/s) and it is inversely proportional to the index of refraction of the medium (for instance, both in copper wire and optical fiber $n \simeq 3/2$, so $s \simeq 2 \times 10^8$ m/s).

Each delay just discussed probably takes on different values depending on whether the packet is being transmitted from the source to the recipient (forwarding direction) or vice-versa (backward direction). Indeed, not only hardware capabilities may differ between end hosts, but also the size of the data packet is typically different from the size of the packet carrying the response. If we assume for simplicity that delays are equal regardless of the direction they refer to, the RTT in a point-to-point communication with no networks in between is:

$$RTT = 3 \times t_{proc} + 2 \times t_{tx} + 2 \times t_{prop} \tag{2.4}$$

Since the processing delay is normally negligible and the transmission delay is incorporated in the propagation delay, the round-trip time is often approximate as $RTT \simeq 2 \times t_{prop}$, although this may be inaccurate.

If the communication goes through some network, the RTT estimation becomes more difficult. Indeed, the physical distance between end-hosts becomes greater, as multiple links need to be traversed, increasing the propagation delay. Plus, several network nodes must also be traversed, each of which introduces a processing delay and a certain **queueing delay**, namely the time a packet needs to wait in a buffer before being processed and transmitted along the transmission medium. This delay usually refers to switching fabrics in the network, as they can only process one packet at a time. Indeed, while the packet at the head of the queue is being processed and transmitted, incoming packets are stored behind it in the queue. Queueing delay is quite variable because depends on the queue occupancy of each crossed node, which is in turn influenced by several factors including the scheduling algorithm adopted, the packet size, the network status and the user traffic behaviour.

In conclusion, since there are several factors that come into play for the RTT, you should know that its estimation is not as simple as one might think. Despite this, measuring RTT of data flows is particularly adopted by network operators because estimates above certain thresholds can signal some anomaly in the network path (e.g. congested node), which should be properly managed to safeguard the health of the network and the reputation of the company.

## 2.2 Probabilistic data structures

In computer science, data are used to describe a reality by means of a digital representation. A data structure is an entity used to store, organize and manage a dataset within a computer's memory using an appropriate method. Data structures can be classified according to several criteria, including the **space-speed efficiency**, i.e. how much memory it requires to store the elements of the dataset compared with how much time it takes to perform operations (writing, reading, updating and deleting) on the elements. In this context, you can distinguish between **deterministic data structures** and **probabilistic data structures**: the first ones rely on the concept of *key*, i.e. an identifier that uniquely matches one element of the dataset, whereas the seconds ones are based on different *hashing techniques* which, exploiting some randomness, may return the same identifier for different elements of the dataset.

This makes deterministic data structures more accurate than probabilistic data structures, as operations on the dataset always return an exact answer, at the price of requiring more memory and therefore being less scalable for some applications. On the other hand, in some scenarios, probabilistic data structures are preferred over deterministic ones because possible errors introduced by the returned approximate answers can be suitably controlled to achieve certain goals, with the main advantage of improving the performance in terms of space and speed. Indeed, probabilistic data structures are typically characterized by low memory requirements, constant query time and high scaling, which are critical especially for Big Data and for computer networks. In fact, in recent years, the speed of data transmission has soared, bringing two different challenges which are interdependent, one related to the packet processing time and another to the memory access time. The growth of the data rate implies the decrease of the transmission time, and

so of the processing time. However, the latter can not decrease at will because it strictly depends on the packet size and on the hardware capability, in particular on the number of clock cycles a CPU needs to elaborate a packet, which in turn depends on the specific data structure adopted to store all the packet processing rules. Therefore, to improve the processing time of a packet, in addition to having high performance hardware, you should minimize the number of memory accesses by using very efficient data structures such as the probabilistic ones.

### Classification

Probabilistic data structures can be grouped in five categories.

**Membership** It consists of deciding whether or not an element belongs to a dataset. There are two definitions of the same problem, which differ on the way they are solved. The following definitions have been taken by [11].

**Set membership problem**

Given a set of elements $S = \{s_1, ..., s_m\}$ on a universe $U$ (i.e. $S \subseteq U$) and given some element $x \in U$, determine whether $x \in S$:

- if $x \in S$, return TRUE always;
- if $x \notin S$, return FALSE always.

Since the *set membership problem* returns always a correct answer, it is solved using deterministic data structures, requiring a storage equal to $m \times L$, so it grows proportionally both to the number of elements to store ($m$) and the average size of each element ($L$). Then, depending on the adopted data structure, the problem is solved with different times, passing from $O(m)$ to $O(log\, m)$ simply keeping the dataset sorted in such a way as to perform a binary search, up to $O(1)$ using hash tables.

**Approximate set membership problem**

Given a set of elements $S = \{s_1, ..., s_m\}$ on a universe $U$ (i.e. $S \subseteq U$) and given some element $x \in U$, determine whether $x \in S$:

- if $x \in S$, return TRUE always;
- if $x \notin S$, return FALSE with high probability.

By relaxing the original problem, you get the approximate set membership problem, in which the result is probabilistic and may not give you a correct answer. Indeed, such a problem includes a particular type of event known as *false positive*, which occurs when the algorithm returns "true" even though the considered element does not belong to the set. Fortunately, this error is compensated by a low memory

requirement which is independent of the average size of the stored elements and simply grows as $\Theta(m)$. The most popular probabilistic data structures that are used to solve the approximate set membership problem are *Bloom filter*, *Cuckoo filter* and *Quotient filter*.

The set membership problem and its approximation find use in various applications of computer networks. For instance, packet processing can involve address lookup (which output port/interface to send an incoming packet to?), firewalling or intrusion detection schemes (which packet to accept or reject?); if instead you focus on network monitoring, you might be interested in evaluating some performance metric (e.g. RTT) by identifying packets belonging to the same flow.

**Cardinality** The problem consists of estimating the number of unique elements in a dataset (e.g. how many packets per flow). *Linear counting*, *LogLog* and *HyperLogLog* are probabilistic data structures that face the problem.

**Frequency** It relies on finding the frequency of some element of a dataset (e.g. detecting the most frequent flows in a data stream). Examples of data structures are *count sketch* and *count-min sketch*.

**Rank** The problem allows you to find a complete rank of the elements (e.g. estimating quantiles and percentiles in a data stream using only one pass through the data). *Random sampling*, *q-digest* and *t-digest* are possible solutions of the problem.

**Similarity** It is responsible for finding the nearest neighbor for large datasets using sub-linear in time solutions. Solutions are based on hashing techniques, such as *MinHash* and *SimHash*.

## 2.3 Hash function

To compactly represent a dataset, probabilistic data structures exploit some randomness using a hash function, namely a function that converts any input data of arbitrary size to a fixed-size value. Basically, given the universe set $U$, regardless of the size of each element, a hash function $h$ maps that set to a range of integer values $n$ that could represent the size of the probabilistic data structure you are interested in, i.e.:

$$h : U \to [1, n] \tag{2.5}$$

### Properties

A hash function is deterministic, meaning that for a given input value it must always generate the same hash value. However, it looks random if the following properties hold [11]:

- *uniform* $\to$ it is able to distribute all the elements of the universe across the data structure in a uniform way:

$$P(h(k) = j) = \frac{1}{n}$$

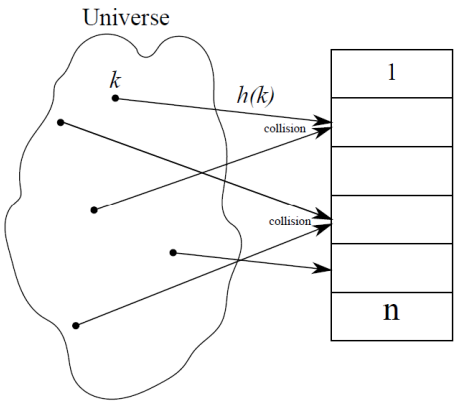



Figure 2.2: Hash function mapping. Figure reproduced from [11]

- *independence between two values* → if the previous property is satisfied, the probability that a hash function returns the same value for two different elements is $1/n$:

$$\forall k_1 \neq k_2, P(h(k_1) = h(k_2)) = \frac{1}{n}$$

### Probability of collision

The term **collision** refers to when two elements of the universe have the same hash value. The probability of a hash collision is a generalization of the *birthday problem* from mathematics. Therefore, if you consider $m$ elements chosen uniformly at random (with repetition), from a set of cardinality $n$ (with $m < n$), the probability $p(n)$ that at least one pair of equal elements has been chosen (collision) is:

$$p(n) = 1 - e^{-\frac{m(m-1)}{2n}} \simeq 1 - e^{-\frac{m^2}{2n}} \tag{2.6}$$

As a consequence, to observe at least one collision with probability $p$, it holds:

$$m \simeq \sqrt{2n \log\left(\frac{1}{1-p}\right)} \tag{2.7}$$

Furthermore, when $n$ is relatively large (i.e. $n \to +\infty$), it can be shown that the typical number of elements is sharply concentrated around its average:

$$E[m] = \sqrt{\frac{\pi}{2} \cdot n} \simeq 1.25\sqrt{n} \tag{2.8}$$

### Classification

Hash functions can be classified in *cryptographic* and *non-cryptographic*, depending on whether or not they have to provide certain security guarantees. Cryptographic hash functions are one-way (it is infeasible to generate the original element from its hash value unless all possible elements are tried), collision resistant (it is very hard to find

keys that collide) and use large number of bits to make brute force inversion harder. Such a complexity makes hash function very robust but also slow. On the other hand, non-cryptographic hash algorithms offer lower security guarantees in exchange for performance improvements [1], mostly focusing on high speed processing and low probability of collisions. Therefore, as long as you don't care so much about the security properties of cryptographic hash functions, the other family of functions is more suitable. That is the reason why, in the field of computer networking, non-cryptographic hash functions are more widespread.

## 2.4 Bloom filter

Bloom filter is a probabilistic data structure based on the approximate set membership, telling you whether or not an element is inside a set. It is known to answer membership queries with an efficient use of memory, as it is implemented by a bit array, i.e. an array that can only store a value of the set $\{0, 1\}$ in any of its positions. In initialization, a Bloom filter is an array of $n$ bits set to 0. Then, the probabilistic part comes into play, as it uses $k$ distinct hash functions for inserting/looking up an item in/from the array, meaning that a query can turn out a false positive due to probability of collision. Therefore, let's delve deeper into these aspects.

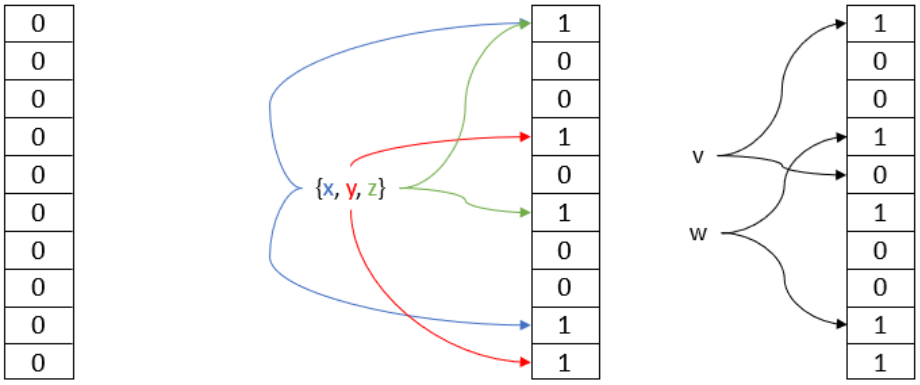
### 2.4.1 Operations

To **insert** an element $x$ to the set, you need to compute $k$ independent hash values $h_1(x)$, $h_2(x), \cdots, h_k(x)$, which will represent the positions of the array where you need to set the bit to 1. If an index is already set, nothing further is required.

For **looking up** an element $x$ (test whether it is in the set), you still need to compute $k$ hash values to get the positions of the array, but now you need to check the value of the bit: if any of them is 0, the element is definitely not in the set (there is no way of having *false negative*); otherwise (all bits set to 1), the element is *probably* present. In fact, there is no way to distinguish whether the element is actually in the set (*true*) or the bits have by chance been set to 1 during the insertion of other elements (*false positive*).

Fig. 2.3 shows an example of Bloom filter with $n = 10$ and $k = 2$. At the beginning the Bloom filter is initialized with all zeros (Fig. 2.3a). After a while the Bloom filter contains the set $\{x, y, z\}$ (Fig. 2.3b), where the colored arrows represent the hash values of each element, namely the positions in the bit array that each element is mapped to. Finally, Fig. 2.3c shows two different cases of query: the element $v$ is definitely not in the set $\{x, y, z\}$ because it hashes to one bit-array position containing 0, whereas $w$ is said to be an element of the Bloom filter although it represents a false positive, having never been entered before.

Bloom filter does not support **deletion** because otherwise you may have false negative, but this is not admissible. Indeed, deleting an element would imply to set bits from 1 to 0 at the positions generated by $k$ hash functions, which may cause the deletion of other elements in the set that share the same positions of the bit array. For instance, suppose that the element $w$ of Fig. 2.3c is inserted in the set, which becomes $\{x, y, z, w\}$. Now, if you suppose that the same element is deleted, the set becomes $\{x, y, z\}$ but it is




(a) Initialization (b) Insertion (c) Lookup

Figure 2.3: Bloom filter example ($n = 10$ and $k = 2$)

compromised. In fact, a probable lookup of the element $x$ or $y$ will return a *false negative*: since some of the positions in the array they were mapped to are now 0, the query tells you that the elements are not in the set even if they are.

### 2.4.2 Performance

The main drawback of Bloom filters is related to false positive events, whose probability can be calculated as:

$$P(FP) = \left(1 - \left(1 - \frac{1}{n}\right)^{mk}\right)^k \simeq \left(1 - e^{-\frac{mk}{n}}\right)^k \tag{2.9}$$

It is a function of the Bloom filter size $n$, the number of hash functions $k$ and the number of inserted elements $m$. Its behaviour is portrayed in Fig. 2.4. You can see that, apart from when $n \leq m$, the function is non-monotone. Intuitively, both few and lot of hash functions do not allow to distinguish among stored elements, resulting in a higher probability of false positive. As a consequence, there exists an optimal value of $k$ that minimizes the probability of false positive, i.e.:

$$k_{opt} = \frac{n}{m} \ln(2) \tag{2.10}$$

Hence, using the optimal value of hash functions, it is possible to optimally design the Bloom filter with a probability of false positive equal to:

$$P(FP) \simeq \left(1 - e^{-\frac{mk}{n}}\right)^{k_{opt}} = (0.6185)^{\frac{n}{m}} \tag{2.11}$$

For completeness, if you want a desired false positive probability $\epsilon$, you can derive the optimal required storage for the Bloom filter from (2.11):

$$n = m \cdot 1.44 \log_2\left(\frac{1}{\epsilon}\right) \quad bits \tag{2.12}$$

This means that, in order to maintain a constant false positive probability, the Bloom filter size must be proportionate to the number of inserted elements.

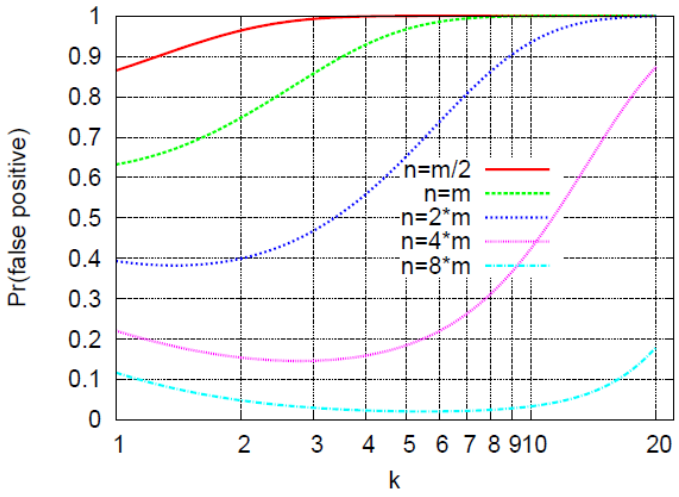



Figure 2.4: Probability of false positive as a function of $k$, for different values of $n$. Figure reproduced from [11]

### 2.4.3 Variants and applications

It is worth knowing that there are many variants of the Bloom filter with the aim of improving it by enabling deletion operation and by optimizing space and/or computational complexity. Since they are out of scope of this work, we only mention a few. *Counting Bloom filter* [9] is implemented by an array of counters instead of an array of bits. Hence, insertion and deletion are performed incrementing and decrementing the $k$ positions by 1, whereas the lookup is limited to check their values (any counter with 0 results in a $false$ answer, else in a $true$ with a certain probability of false positive). As in practice the counter consists of 4 or more bits, a counting Bloom filter requires at least 4 times more space than a standard Bloom filter. Another variant is the *Cuckoo filter* [8], which is implemented by a hash table that uses cuckoo hashing to store items' fingerprint. Thanks to only storing fingerprints and cuckoo hashing, which leads to high table utilization by solving collisions, Cuckoo filter is typically compact. According to the same authors, a properly designed Cuckoo filter is more efficient than conventional Bloom filter for applications that require low false positive rates, particularly for $\epsilon < 3\%$.

To conclude, we want to let you know that, in computer networking, Bloom filters are widely used in different tasks [15], such as monitoring and measurement (as in our case), packet routing and forwarding, caching of Web servers and storage servers, and so on.

# Chapter 3

# Real-time RTT monitoring

This chapter contains the heart of our proposal, describing the goal scenario and the operating principles of the system. In particular, after the description of the system model under examination, we will move on to its design, explaining sequentially the parameters involved, the way in which the system behaves when receiving outgoing and incoming packets, what are the problems to be faced in the flow RTT measurements and how they can be addressed, also reporting the relative pseudo-codes in case you want to use them for further study. Finally, since there is no state of the art for our purpose yet, we will make a high-level comparison between our solution and the most similar ones present in the scientific literature.

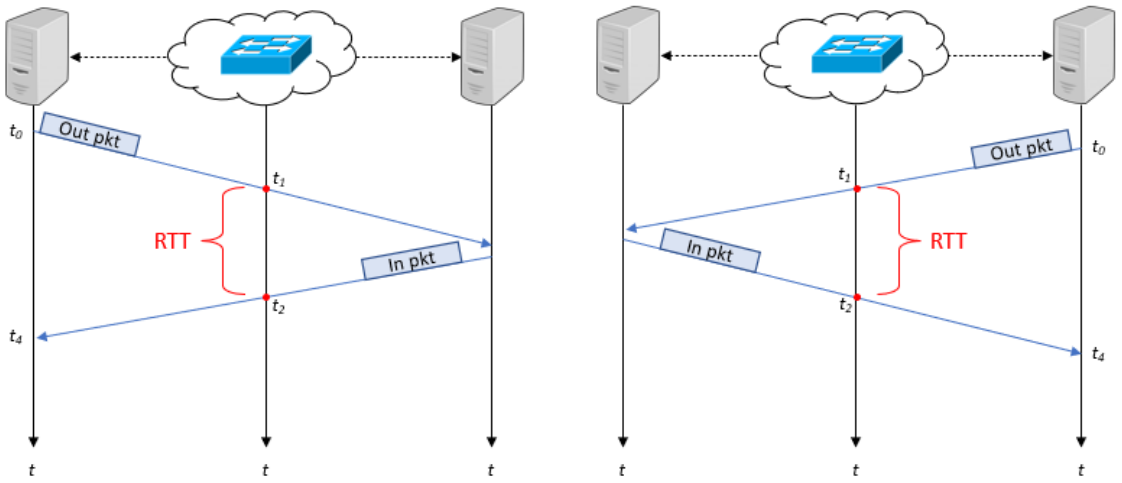
## 3.1 System model

For our study, we consider a system where the measurement point is a network device (e.g. a switch or a router) between source and destination. To measure RTT, the measurement point tracks packets passing through it in order to individuate the first pair of packets that share the same flow ID. The system just described is shown in Fig. 3.1.

Figure 3.1: System model

Since flows are typically bidirectional, from now on, regardless of the direction in which packets are transmitted, we will refer to as *outgoing packet* and as *incoming packet* respectively the first packet in the forwarding direction and the first packet in the backward one, both related to the same flow and observed from the measurement point. Fig. 3.2 shows that, although it is possible to be in two different scenarios, the measurement point is equally able to measure the flow RTT. Indeed, the latter is always given by the time difference between incoming and outgoing packets of the considered flow. For example, if the outgoing packet is observed at time $t_1$ and the incoming packet at time $t_2$, then $RTT = t_2 - t_1$. Obviously, RTT measurement is possible if the network device is somehow

able to record the time in which each outgoing packet is observed. As already said in Sec. 1.1, nowadays the number of flows in a network is quite large, so using traditional data structures would be expensive in terms both of memory and computation. This is where probabilistic data structures come to the aid, allowing us to adopt a solution that can answer a query with limited space and time.




(a) Case 1 (b) Case 2

Figure 3.2: RTT representation in the system model (for simplicity, network delays are not represented)

## 3.2 System design

To test our solution, we have designed a probabilistic data structure composed of a group of Bloom filters which are spaced in time. Basically, imagining time as a set of intervals (or time slots), each of these can be covered by a Bloom filter which is in charge of storing outgoing packets that arrive within it (Fig. 3.3a). Now, the question that arises is: how is it possible to cover the time horizon, which is infinite, through a data structure with finite memory? You have to imagine the data structure circularly, meaning that it has no real end and it can loop around itself (Fig. 3.3b). The working principle is the same as the clock: when the time reaches the slot that represents the tail of the data structure, at the next time slot you will start again from its head.

At this point, another question that can arise is: how much memory does the data structure need to store an indefinite number of flows over time? Contrary to before, this is a rather difficult question to answer because there are several factors that come into play, which intertwine with each other and which affect the accuracy of our final goal, namely the RTT measurement. The latter can never be a precise measurement, but rather an approximation, as we are dealing with a probabilistic data structure. Indeed, once an outgoing packet is stored in a Bloom filter, there is no longer any way of knowing the exact time it arrived in the system, but only a reference time associated with that

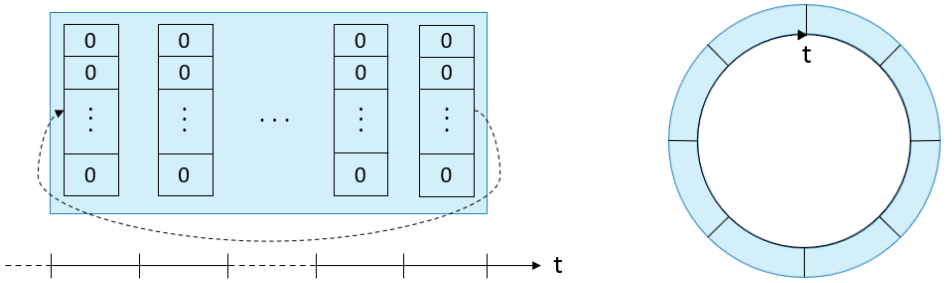
specific time slot. This obviously will introduce errors when calculating the RTT of each flow.

Before going into the detailed explanation of the system, we want to emphasize two points:

- flows are typically bidirectional, but the routing of outgoing and incoming packets may be asymmetric, meaning that only one of them can follow the path through the measurement point. In this work we don't deal with this event, assuming for simplicity that the routing is always symmetric in such a way that the measurement point can always observe a pair of outgoing-incoming packets belonging to the same flow.

- the proposed solution is protocol-agnostic: once defined what is meant by *flow*, and therefore the relative *flow ID*, it can be used in any network regardless of the adopted protocols. As a consequence, its working principles may be more suitable for some protocols and less suitable for others. Hence, if you wish to use it for a specific protocol, its working principles should be revisited accordingly.

The management of asymmetric routing and the definition of protocol-oriented solutions are left to future work.




(a) Linear representation (b) Circular representation

Figure 3.3: Matching probabilistic data structure over time

### 3.2.1 Parameter definition

The system makes use of a probabilistic data structure composed of $b$ Bloom filters. Let $N$ be the available amount of memory, then each Bloom filter will have a size of:

$$n = \frac{N}{b} \quad [bit] \tag{3.1}$$

Moreover, each Bloom filter covers an interval of $\Delta$ seconds and is temporally identified through a tag:

$$BF_i = i \cdot \Delta \qquad \text{with } i \in [0, b-1] \tag{3.2}$$

It is important to remember that, over time, there is always a $1:1$ mapping between time slots and Bloom filters, as we are dealing with a circular probabilistic data structure. Therefore, when an outgoing packet enters the system, it must be stored in the Bloom filter which is covering the current time slot at that instant. Then, it is necessary to establish how long the system can wait to observe a response of the packet just stored. We denote with $t_{max}$ the maximum amount of time that can elapse to receive a response of an outgoing packet before the system is no longer able to compute a RTT for them. This parameter is crucial for the system design because it determines, along with $b$, the duration of both the time slot $\Delta$ and the time window $T$ covered by the entire data structure, calculated respectively as:

$$\Delta = \frac{t_{max}}{b-1} \quad [sec] \tag{3.3}$$

$$T = t_{max} + \Delta \quad [sec] \tag{3.4}$$

The reason why the system has been designed with a time window $T$ which is $\Delta$ seconds wider than $t_{max}$ will be explained in Sec. 3.2.5.

Table 3.1 summarizes all these parameters, while Fig. 3.4 portrays the system as it was designed. The following sections, instead, are dedicated to describe in detail the behaviour of the system and how its accuracy in measuring the RTT would be affected by the parameter setting.

| Parameter | Description |
|---|---|
| $N$ | Data structure memory |
| $b$ | Number of Bloom filters |
| $n$ | Memory per Bloom filter |
| $BF_i$ | Identifier tag of the $i$-th Bloom filter |
| $\Delta$ | Time slot width |
| $T$ | Time window of the entire data structure |
| $t_{max}$ | Maximum threshold to compute RTT |

Table 3.1: System parameters

### 3.2.2 Reference time of outgoing packets

#### Bloom filter identification

When an outgoing packet enters the system, it is necessary to identify the tag of the Bloom filter in which to store it, which as we know is associated with a time slot. In particular, it is the time slot within which the packet is observed in the system.

Let $t_{out}$ be the timestamp of the outgoing packet, i.e. the time at which it enters the system, and let $t_{out_{circ}}$ be the time instant of the outgoing packet taking into account the circularity of the data structure (for simplicity, we rename it **circular time**). The latter is obtained as follows:

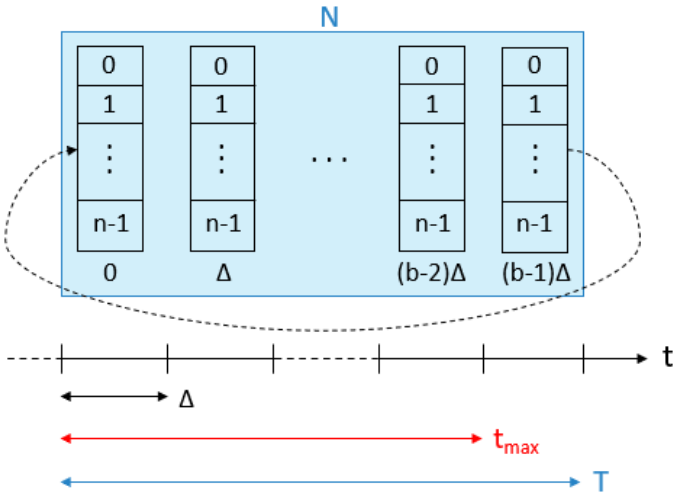



Figure 3.4: Reference of designed system

$$t_{out_{circ}} = t_{out} \mod T \tag{3.5}$$

where *mod* is the modulo operator, which returns the remainder of the division between two numbers, particularly useful for considering the "wrap-around" property typical of circular problems.

Then, to identify the slot where the packet fell in, and therefore the tag of the Bloom filter where you will need to store it, you would follow function 1. Basically, for every time slot of the window $T$, you check whether or not the circular time falls inside it, stopping as soon as the condition is satisfied.

**Function 1** Identify Bloom filter tag

**Input:**

- $b$ := number of Bloom filters;
- $\Delta$ := time slot width;
- $t_{arr_{circ}}$ := circular arrival time of the considered packet (it can be either outgoing or incoming);

**Output:**

- $BF_i$ := BF tag associated to the time slot in which the considered packet is observed.

```
1: function GET_BF_TAG(b, Δ, t_arr_circ)
2:     for i = 0, ..., b − 1 do
3:         if (t_arr_circ ≥ i · Δ) and (t_arr_circ < (i + 1) · Δ) then
4:             BF_i ← i · Δ
5:             return BF_i
```

#### Reference time

Once the Bloom filter tag has been identified, the process for storing a packet within it is the one discussed in Sec. 2.4.1, where the input of the hash function is represented by the flow ID of the considered packet. Since the information you can store for every outgoing packet is simply the collection of bits that must be set to 1 in correspondence of the BF indexes that are returned by the hash values, any information about the exact instant it arrived in the system is lost. That is where the first issue arises: when an incoming packet will enter the system and look for the BF in which the associated outgoing packet was stored, which time instant of that slot do you should consider to compute the RTT? You have to make a design choice: for each time slot, you must properly choose a reference time which will represent the hypothetical arrival time of every outgoing packet. Let's indicate with $t_{ref_i}$ the reference time of the $i$-th slot, with $i \in [0, b-1]$. Then, a trivial choice is to set $t_{ref_i}$ to the intermediate value of the considered time slot, i.e.:

$$t_{ref_i} = \frac{i \cdot \Delta + (i+1) \cdot \Delta}{2} \tag{3.6}$$

From this equation, you can guess that $\Delta$ is a key parameter of the system. Intuitively, you would like to have $\Delta$ as small as possible in order to limit the distance between the real arrival time of outgoing packets in the system considering its circularity ($t_{out_{circ}}$) and their hypothetical arrival time ($t_{ref_i}$), which would translate in a more accurate estimation of the RTT. However, the counterpart of this choice is that, for the same amount of memory $N$ of the data structure, to cover the same time window $T$ with such a high granularity of time slots, you will have many Bloom filters with poor memory $n$ which, as we know from Sec. 2.4.2, implies a high probability of false positive. This is also confirmed by looking at (3.1) and (3.3): both $n$ and $\Delta$ are inversely proportional to the number of Bloom filters $b$ but, while in the second case this is a benefit, making the system potentially more accurate, in the first case it is not, increasing the false positive events from which the Bloom filter suffers. In short, you can control the accuracy of the system in evaluating the RTT by trading time slot width and Bloom filter memory, which from now on we will call **time-space trade-off**.

Now that we have consolidated how the system works with outgoing packets, we move to the next phase, which is related to incoming packets.

### 3.2.3 "Candidate" Bloom filter

When a incoming packet enters the system, it must not be stored in the data structure, unlike an outgoing packet. Instead, since packets that belong to the same flow share the same flow ID, and so the same hash values, you have to use them to perform a *lookup* in every Bloom filter of the data structure in order to individuate the one in which its related outgoing packet *may* have been stored. Note that we used the modal verb "may" in the previous sentence because the lookup operation in a Bloom filter can return a true answer even if it is wrong, due to the false positive event that characterizes the approximate set membership problem (see Sec. 2.4.1). Hence, we introduce the concept of **candidate**, which represents each Bloom filter that returns a true answer as a result of a lookup operation. Obviously, since there is no way to know which candidate represents

the Bloom filter where the outgoing packet was actually stored, you should select one of them to estimate the RTT. At this point, it starts to become more apparent how the space-time trade-off previously defined is crucial: a high time granularity is synonymous with better RTT estimations but also with higher probability of false positive, which implies a greater number of candidates and a higher probability of selecting the wrong Bloom filter (and vice-versa). And that's where a new problem comes into play: how do you choose among candidate Bloom filters while minimizing the RTT error? Again, this problem is not trivial. We though some possible solutions, which are shared in Sec. 3.3 and which were used to evaluate the accuracy of the RTT estimation under a specific system setting (discussion in Sec. 4.2.3).

### 3.2.4 RTT estimation

Regardless of the algorithm which is used to select a candidate Bloom filter, the way of measuring the flow RTT in the system just described is the same. Similarly to what was done with the outgoing packets, we indicate with $t_{in}$ and $t_{in_{circ}}$ the timestamp and the circular time of the incoming packet, respectively. Also, let $BF_{out}$ be the tag of the Bloom filter that has been chosen among candidates, and let $BF_{in}$ be the Bloom filter tag associated with the time slot where the incoming packet is observed in the system. Then, the RTT of $j$-th flow is estimated as follows:

$$\hat{RTT}_j = \begin{cases} (t_{in_j} - t_{ref_j}) \mod T & \text{if } BF_{out_j} \neq BF_{in_j} \\ t_{in_{circ_j}} - \hat{t}_{ref_j} & \text{otherwise} \end{cases} \tag{3.7}$$

Let's discuss the reason why there are two cases to distinguish. Basically, to get the flow RTT when outgoing packet and incoming packet enter the system at different time slots (i.e. $BF_{out} \neq BF_{in}$), you need to exploit the circularity of the data structure after subtracting the reference time of the outgoing packet from the timestamp of the incoming packet. However, the same principle is not always valid in the other case because it may violate the sequence in which outgoing and incoming packets should enter the system. Indeed, since we are considering the intermediate value of the slot as reference time of the outgoing packet, this assumes that the incoming packet can enter the system only after that time reference and by the end of the time slot under consideration, otherwise it is as if we were considering the outgoing packet in the future with respect to its incoming packet, which is impossible. Hence, a possible solution is to redefine the reference time of the outgoing packet, which we indicate with $\hat{t}_{ref_j}$ to distinguish it from the other case, again for $j$-th flow. So, we define it as the intermediate value between the left boundary of the time slot in question and the circular time of the incoming packet, i.e.:

$$\hat{t}_{ref_j} = \frac{BF_{out_j} + t_{in_{circ_j}}}{2} \tag{3.8}$$

In such a way, the new reference time of the outgoing packet is surely earlier than the time in which the incoming packet enters the system. Then, since both packets belong to the same time slot, their RTT can be estimated simply as the difference between the circular time of the incoming packet and the reference time of the outgoing packet just defined.

Finally, we would like to say that there is also another case in which the reference time should be managed properly, namely when the incoming packet arrives in the second half of the time slot and the chosen candidate is at the other extreme of the window $T$. In such a case, taking the intermediate value of that time slot will return a $\hat{RTT}_j > t_{max}$, which is inadmissible according to our design. To deal with this, the reference time should be set as the intermediate value of the residual time of the candidate time slot, i.e. the time that goes from $t_{in_{circ_j}} - t_{max}$ up to the right boundary of the slot. This case has not been covered in our work.

### 3.2.5 Bloom filter reset

As the amount of memory is finite while outgoing packets can enter the system without any time limit, it follows that the number of bits set to 1 of the various Bloom filters potentially tend to increase over time. To control this undesired event, which directly affects the probability of false positive when searching for candidates, each Bloom filter must be reset periodically to erase its content. Obviously, this should be done without jeopardising the possibility of computing flow RTT of already stored outgoing packets. This is the reason why the system has been designed by covering a time window $T$ with a margin of one time slot with respect to the maximum amount of time that can elapse between a pair of outgoing-incoming packets belonging to the same flow($t_{max}$), as you can see by looking at (3.4) and Fig. 3.4. Hence, given a Bloom filter with tag $BF_i$, the time in which it should be reset is given by:

$$t_{reset_{BF_i}} = w \cdot T + BF_i \qquad \text{with } i \in [0, b-1], w \in \mathbb{N}^+$$

In short, according to our system design, the reset period of each BF corresponds to the time window $T$ of the data structure. This allows flow RTT estimation for any outgoing packet already stored in the system, provided that its response arrives within $t_{max}$ seconds. In fact, for each outgoing packet, you are guaranteeing that $t_{max}$ seconds will have to elapse, from the moment it entered the system until it is deleted by the BF where it is stored.

## 3.3 Policies to choose a candidate Bloom filter

In this section we provide some possible solutions that can be used to choose among candidate Bloom filters, distinguishing between the case where the RTT distribution is known and the case it is not. Indeed, since the RTT can be modelled as a random variable, the knowledge of its statistical description can be exploited to increase the probability of choosing the true candidate, i.e. the one in which the outgoing packet related to the incoming packet under examination is actually stored.

### 3.3.1 Known RTT distribution

Let $X$ be the random variable associated to the flow RTT, we denote its probability density function (shortened to PDF) as $f_X(t)$ and its cumulative distribution function

(shortened to CDF) as $F_X(t)$. From probability theory, there exists a relationship between PDF and CDF, i.e.:

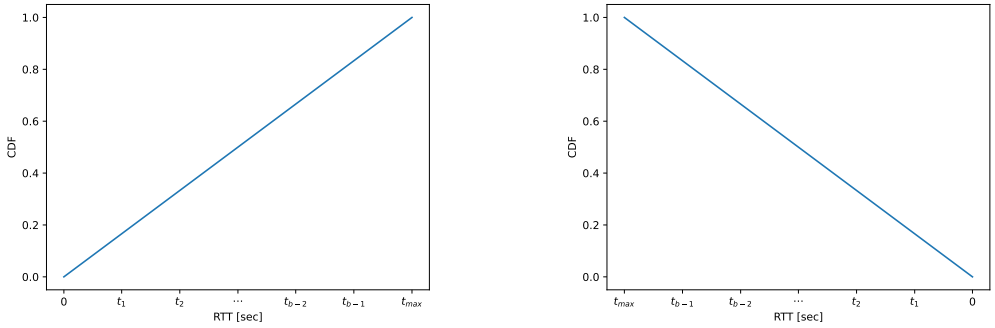
$$f_X(t) = \frac{d}{dt}F_X(t) \quad \leftrightarrow \quad F_X(t) = \int_{-\infty}^{t} f_X(y)dy$$

It means that, from the knowledge of one function, it is pretty straightforward to derive the other. Thus, assuming you know either the PDF or the CDF, this can be leveraged to do a clever search among candidates.

Because of its unpredictable nature, we can image the RTT to be described by a skewed distribution. This is actually confirmed by some sources [3, 5], reporting how TCP flows over the Internet are characterized by RTT measurements which, on average, are settles on low values. Such a type of behaviour is described by **right skewed distributions**, so-called because they have long tail on the right side of their peak (mode), meaning that most of the observations are concentrated on the left side, leaving on the other extreme only few observations, which are therefore relatively infrequent. Obviously, whichever skewed distribution is considered, its support must be truncated and upper bounded by $t_{max}$ to comply with our system design, i.e. it must hold $X \in [0, t_{max}]$. In the following, we provide further information on how to use the knowledge of the RTT distribution in our system, assuming to know its CDF.

When an incoming packet enters the system, you need to look backwards in time to select a Bloom filter among the candidates. To adapt this characteristic of the system to $F_X(t)$, it is sufficient to assume the time when the incoming packet enters the system as the lower extreme of the distribution (i.e. $F_X(t = t_{in}) = 0$) and to flip the latter horizontally, in such a way that its upper extreme is $t_{max}$ seconds in the past with respect to that instant. Fig. 3.6 allows you to visualize how the mapping between the RTT distribution and the time window system takes place, assuming for simplicity that you have a uniform CDF with support in $[0, t_{max}]$, reported in Fig. 3.5. Not only that, but from Fig. 3.6 you can also appreciate how the distribution is split into as many bins as the number of Bloom filters composing the data structure. It is the case to state a rightful premise, namely that not always a bin matches a whole time slot. Indeed, if you take a screenshot of the system at the instant an incoming packet arrives (i.e. at $t_{in}$) and you look back in time as already explained, the bin associated with that time slot will cover the interval $(0, t_1]$ - where $t_1 = t_{in_{circ}} - BF_{in}$ and represents the left boundary of the time slot where the incoming packet is observed -, which is for sure smaller than $\Delta$ seconds. Plus, since the support of the RTT distribution is finite over $(0, t_{max}]$ while the time window system $T$ is $\Delta$ seconds wider than that, also the bin associated to the oldest time slot of the current window will be smaller than $\Delta$, covering the interval $(t_{b-1}, t_{max}]$, where $t_{b-1} = t_{max} - (\Delta - t_{in_{circ}})$ and represents the residual part of the RTT distribution support. All the intermediate bins, instead, completely match the relative time slot of the time window system.

Having said that, we believe that knowing the RTT distribution can also be useful to cleverly set the reference time of the outgoing packet within a time slot. In fact, defining as $g_X(t)$ the probability density function of the random variable $X$ truncated over the support $(t_i, t_{i+1}]$ representing the $i$-th bin - i.e.




(a) Original (b) Reversing x-axis

Figure 3.5: CDF of the uniform random variable RTT with support $[0, t_{max}]$

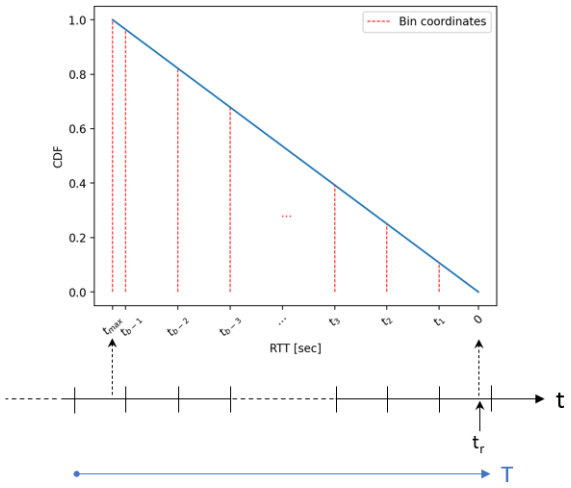



Figure 3.6: Mapping support of uniform RTT distribution in time window system

$$g_X(t) = f_X(t|t_i < X \leq t_{i+1}) = \frac{f_X \cdot I(\{t_i < X \leq t_{i+1}\})}{F_X(t_{i+1}) - F_X(t_i)}$$

- instead of considering the intermediate value of the time slot (as in (3.6)), it is reasonable to set the reference time as the expected valued of the random variable $X$ conditioned to the new support, i.e.:

$$t_{ref_i} = E[X|t_i < X \leq t_{i+1}] = \frac{\int_{t_i}^{t_{i+1}} y \cdot g_X(y)dy}{F_X(t_{i+1}) - F_X(t_i)} \tag{3.9}$$

Function 2 shows how the mean of the random variable $X$ defined over a generic support $(t_a, t_b]$ can be computed in terms of pseudo-code.

Finally, for our purpose we can exploit the knowledge of the RTT distribution either in terms of binning (i.e. probability associated within each bin) or in terms of central

tendency parameters (e.g. median or mean). We therefore propose three solutions: one based on the binning, one on the median, and another on the mean.

**Function 2** Get the expectation of the r.v. X over the truncated support $(t_a, t_b]$

**Input:**

- $t$ := x coordinates of the CDF describing the r.v. X;
- $F_X$ := y coordinates of the CDF describing the r.v. X;
- $t_a$ := lower limit of the truncated distribution;
- $t_b$ := upper limit of the truncated distribution;

**Output:**

- *mean_trunc_pdf* := expectation of the distribution truncated over the support $(t_a, t_b]$.

```
1:  function GET_MEAN_TRUNC_PDF(t, F_X, t_a, t_b)
2:      /* Initialization.                                                    */
3:      trunc_t ← new array    ▷ x coordinates of truncated func. → t ∈ (t_a, t_b]
4:      trunc_F_X ← new array                  ▷ y coordinates of truncated CDF
5:      trunc_f_X ← new array                  ▷ y coordinates of truncated PDF
6:      arr_temp ← new array                                ▷ Temporary array

7:      /* Use some interpolation method to derive the probability that X is
           smaller or equal than t_a and t_b.                                 */
8:      prob_a ← interpolate t_a with the function F_X(t)          ▷ P(X ≤ t_a)
9:      prob_b ← interpolate t_b with the function F_X(t)          ▷ P(X ≤ t_b)

10:     /* Compute the probability that X is within (t_a, t_b].             */
11:     prob_bin ← prob_b − prob_a                          ▷ P(t_a < X ≤ t_b)

12:     /* Compute the truncated arrays of t and F_X over the interval (t_a, t_b]. */
13:     trunc_t ← use some method to get evenly spaced numbers in (t_a, t_b]
14:     trunc_F_X ← interpolate any value of trunc_t with the function F_X(t)

15:     /* Compute the truncated array of the PDF.                          */
16:     insert 0 into trunc_f_X
17:     for i = 1, ..., length of trunc_F_X do
18:         p_i ← trunc_F_X[i] − trunc_F_X[i − 1]
19:         insert p_i into trunc_f_X

20:     /* At this point, we need to get the expectation of the truncated function
           over (t_a, t_b]. The analytical formula (3.9) can be split into 3
           steps: (1) product between items in the same positions of trunc_t
           and trunc_f_X; (2) sum the items of the array obtained from the
           previous step; (3) divide the value obtained from the previous step
           by the probability within (t_a, t_b].                            */
```

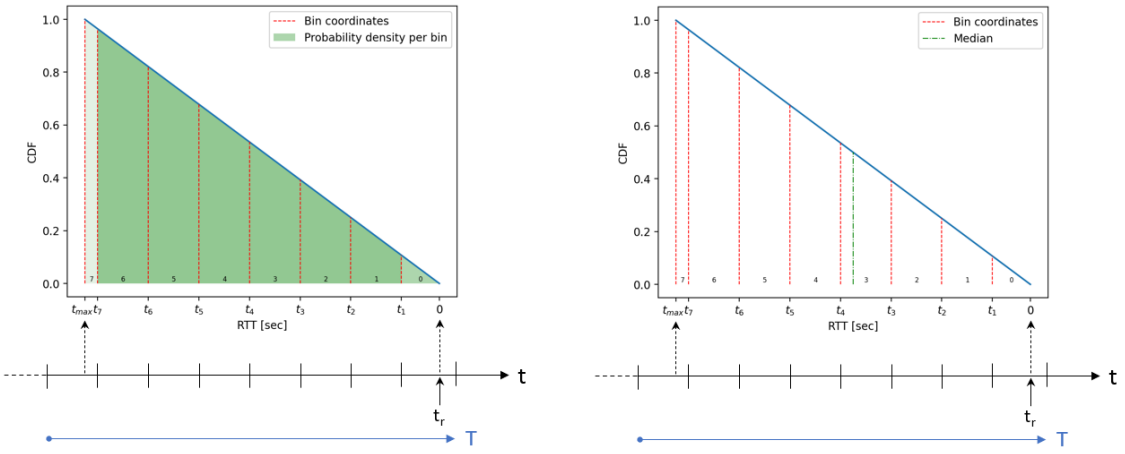
```
21:  for i = 0, ..., length of trunc_f_X do                    ▷ Step (1)
22:      product ← trunc_t[i] * trunc_f_X[i]
23:      insert product into arr_temp

24:  numerator_value ← sum items of arr_temp                   ▷ Step (2)

25:  mean_trunc_pdf ← numerator_value/prob_bin                 ▷ Step (3)

26:  return mean_trunc_pdf
```

### Search most likely candidate

Every time an incoming packet enters the system, the procedure consists first in mapping the distribution with the system in order to derive the bins and the probability density in each of them (pseudo-code in Function 3), and then in using this information to search for a candidate starting from the most likely bin to the least likely one (pseudo-code in Function 1). To better understand both the procedure and its implementation by pseudo-codes, you can consider Fig. 3.7a as example. It represents a snapshot of the system with 8 Bloom filters at the instant time an incoming packet enters the system, which is assumed to be at almost tree-quarters of a generic time slot. Focusing on the distribution, you can see how the area within each bin has different shades of the same color to indicate different probability densities: the brighter the color, the higher the probability density within the bin. In fact, since the distribution is uniform, the probability density is equal and maximum within each bin of width $\Delta$, whereas it is smaller within the first and the last bin, as both are narrower than $\Delta$. As a consequence, for candidate selection, we will first look for Bloom filters associated with bins having indexes $ind \in \{1, 2, 3, 4, 5, 6\}$ - in this case the order does not matter because all of them have the same probability -, then for Bloom filter associated with bin 0 and finally for the one with bin 7. Obviously, it is not necessary to visit all the BFs, but to stop as soon as the first candidate is found.

### Search candidate around median (or mean)

The procedure we propose is the same irrespective of using the median or the mean, therefore we explain it considering one of them. Let $t_{med}$ be the median value of the distribution. To exploit it in the system, you need to go back $t_{med}$ seconds with respect to the circular time of the incoming packet, i.e. $t_x = t_{in_{circ}} - t_{med}$, and to identify the Bloom filter tag associated with the time slot that includes $t_x$. Indeed, since the Bloom filter just identified is located at the median of the distribution with respect to the considered incoming packet, it is reasonable to take it as reference to start selecting a candidate. Therefore, the idea we propose is the following: if the reference Bloom filter just individuated is a candidate, you select it and stop the search immediately; otherwise you start looking for a candidate by jumping from one side to the other around the reference, stopping as soon as the first candidate is found. Note that when the reference Bloom filter is not a candidate, since we are dealing with right-skewed distributions, the




(a) Most likely bin

(b) Search around median

Figure 3.7: Example of policies applied to a system with $b = 8$ and RTT uniformly distributed in $[0, t_{max}]$, assuming that the incoming packet arrives at almost 3/4 of a time slot

number of BFs to visit from the reference BF to the BF where the incoming packet is observed is smaller than the number of BFs to visit in the opposite side. Therefore, we need to split the search around the reference BF into two phases:

1. you need to check in pairs the BFs next to the reference one which have the same distance from it and to stop when the pair that includes the BF associated with the incoming packet is reached;

2. if no candidate was found in the previous phase, you need to check all the remaining BFs in increasing order of distances from the reference one.

Again, taking Fig. 3.7b as an example can help to understand. Hence, once the incoming packet enters the system, you individuate the bin which includes the median value of the distribution. In our case, it is located in bin 3, which means that the Bloom filter associated with it is taken as reference. Then, you have to check if it is a candidate: if it is, you can stop right away; otherwise you have to start a search around it considering the two phases described above. Indeed, even if the uniform distribution has null skewness, the number of BFs that should be visited after bin index 3 are 3 in one way and 4 in the other way, i.e. bin indexes are $\{2, 1, 0\}$ and $\{4, 5, 6, 7\}$ respectively. Thus, the two phases consists in searching for a candidate as follows:

1. 3 pairs of BFs having the same distance from the reference one can be distinguished, i.e. BFs associated with bin indexes $\{2, 4\}$ at distance 1, $\{1, 5\}$ at distance 2 and $\{0, 6\}$ at distance 3. Therefore, the algorithm consists in searching for a candidate according to the following order of bin indexes: $\{2, 4, 1, 5, 0, 6\}$.

2. if no one of the previous BFs was a candidate, there is only one BF left to check, i.e. the one associated with bin index 7 at distance 4.

The implementation in pseudo-code is reported in Algorithm 2.

**Function 3** Get list of bins to visit in descending order of probability

**Input:**

- $b$ := number of Bloom filters;
- $\Delta$ := time slot width;
- $t_{max}$ := maximum RTT threshold;
- $t_{in_{circ}}$ := circular arrival time of incoming packet;
- $t$ := x coordinates of RTT distribution;
- $F_X$ := y coordinates of RTT CDF;

**Output:**

- *array_ind_bins* := array of bins sorted by descending probability.

```
1:  function GET_BINS_TO_VISIT(b, Δ, t_max, t_in_circ, t, F_X)
2:      /* Initialization.                                                    */
3:      t_bins ← new array          ▷ x coordinates of bins → t ∈ [0, t_max]
4:      F_bins ← new array          ▷ y coordinates of bins → CDF ∈ [0, 1]
5:      f_bins ← new array                ▷ y coordinates of bins → PDF
6:      sort_f_bins ← new array ▷ Prob. density associated with each bin,
        sorted in descending order of prob. per bin
7:      array_ind_bins ← new array ▷ Bin indexes in descending order of prob.

8:      /* Compute time coordinates of bins, taking into account that the first
           and the last bins have different width with respect to others.  */
9:      for i = 0, ..., b − 1 do
10:         if (t_in_circ ≥ i · Δ)  and  (t_in_circ < (i + 1) · Δ) then
11:             t_1 ← t_in_circ − (i · Δ)                ▷ First bin = (0, t_1]
12:             insert t_1 into t_bins
13:             break
14:     for i = 1, ..., b − 2 do
15:         t_{1+i} ← t_1 + (i · Δ)                ▷ Intermediate bins = (t_i, t_{1+i}]
16:         insert t_{1+i} into t_bins
17:     insert t_max into t_bins                          ▷ Last bin = (t_{b−1}, t_max]

18:     /* Use some interpolation method to derive the probability that the
           RTT is smaller or equal than each time coordinate within t_bins. */
19:     F_bins ← interpolate t_bins with the function F_X(t)

20:     /* Compute the probability density inside each bin.                */
21:     p_0 ← F_bins[0]                                          ▷ First bin
22:     insert p_0 into f_bins
23:     for i = 1, ..., b − 1 do
24:         p_i ← F_bins[i] − F_bins[i − 1]                   ▷ Residual bins
25:         insert p_i into f_bins
```

```
26:     /* Get the list of bin indexes sorted in descending order of probability. */
27:     sort_f_bins ← sort f_bins in descending order
28:     for each p ∈ sort_f_bins do
29:         ind_bin ← array index of p associated with f_bins
30:         insert ind_bin into array_ind_bins

31:     return array_ind_bins
```

**Algorithm 1** Search most likely candidate

**Input:**

- $b$ := number of Bloom filters;
- $\Delta$ := time slot width;
- $t_{max}$ := maximum RTT threshold;
- $t_{in_{circ}}$ := circular arrival time of incoming packet;
- $t$ := x coordinates of RTT distribution;
- $F_X$ := y coordinates of RTT CDF;
- $BF_{in}$ := Bloom filter tag where incoming packet is observed;
- $array\_BF\_tags$ := array containing all the Bloom filter tags;
- $array\_candidates$ := array containing only candidate Bloom filter tags.

**Output:**

- $BF_{out}$ := Bloom filter tag where the outgoing packet is supposed to be stored.

```
1:  /* Initialization.                                                    */
2:  ind_BF_in ← BF_in/Δ
3:  is_found ← False
4:  array_ind_bins ← GET_BINS_TO_VISIT(b, Δ, t_max, t_in_circ, t, F_X)

5:  /* Search most likely candidate.                                      */
6:  for each ind_bin ∈ array_ind_bins do
7:      ind_BF_to_check ← (ind_BF_in − ind_bin) modulo b
8:      tag_BF_to_check ← array_BF_tags[ind_BF_to_check]
9:      for each tag_BF_candidate ∈ array_candidates do
10:         if tag_BF_to_check == tag_BF_candidate then
11:             BF_out ← tag_BF_candidate
12:             is_found ← True
13:             break
14:     if is_found == True then
15:         break

16: return BF_out
```

**Algorithm 2** Search candidate around median

**Input:**

- $b$ := number of Bloom filters;
- $\Delta$ := time slot width;
- $t_{in_{circ}}$ := circular arrival time of incoming packet;
- $t_{med}$ := median value of the distribution;
- $BF_{in}$ := Bloom filter tag where incoming packet is observed;
- $array\_BF\_tags$ := array containing all the Bloom filter tags;
- $array\_candidates$ := array containing only candidate Bloom filter tags.

**Output:**

- $BF_{out}$ := Bloom filter tag where the outgoing packet is supposed to be stored.

```
/* Initialization.                                                    */
is_found ← False                                  ▷ Boolean variable

/* Get time associated to the median of the distribution with respect to
   the circular time of the current incoming packet. Then get the BF tag
   associated with the time slot which includes that time.            */
t_x ← t_in_circ − t_med                           ▷ Median time wrt t_in_circ
BF_med ← GET_BF_TAG(b, Δ, t_x)                    ▷ Tag

/* Check whether BF_med is a candidate.                               */
for each tag_BF_candidate ∈ array_candidates do
    if BF_med == tag_BF_candidate then
        BF_out ← tag_BF_candidate
        is_found ← True
        break

/* If BF_med is not a candidate, we need to start checking among the
   remaining BFs, performing a search around it through two phases:
   (1) Check in pairs the BFs next to the reference one which have the
   same distance from it. If no one of them is a candidate, you still need
   to stop this type of search when the pair that includes the BF associated
   with the incoming packet is reached. (2) Check all the remaining BFs
   in increasing order of distances from the reference one.           */
```

```
/* Phase 1                                                                    */
if is_found == False then
    /* Before checking in pairs the BFs next to the reference one which
       have the same distance from it, we need to find the stopping con-
       dition of this type of search, which is given by the number of BFs
       that separate BF_med from BF_out.                                      */
    BF_in ← GET_BF_TAG(b, Δ, t_in_circ)   ▷ BF tag of time slot including t_in
    ind_BF_in ← BF_in/Δ                   ▷ Index associated to BF_in
    ind_BF_med ← BF_med/Δ                 ▷ Index associated to BF_med
    num_BFs_shift ← (ind_BF_in − ind_BF_med) modulo b   ▷ Stop cond.

    /* At this point we can start searching a candidate looking at BFs in
       pairs next to the reference.                                           */
    if num_BFs_shift! = 0 then
        for i = 0, . . . , num_BFs_shift − 1 do
            /* Focus on the BF to the right of the median one (i.e. closer
               to t_in_circ)                                                  */
            ind_right_BF ← (ind_BF_med + (i + 1)) modulo b      ▷ Index
            tag_right_BF ← array_BF_tags[ind_right_BF]          ▷ Tag
            /* Now, check whether tag_right_BF is a candidate.             */
            for each tag_BF_candidate ∈ array_candidates do
                if tag_right_BF == tag_BF_candidate then
                    BF_out ← tag_BF_candidate
                    is_found ← True
                    break

            /* If tag_right_BF is not a candidate, we turn our attention
               on the BF to the left of the median one (i.e. farther from
               t_in_circ)                                                  */
            if is_found == False then
                ind_left_BF ← (ind_BF_med − (i + 1)) modulo b   ▷ Index
                tag_left_BF ← array_BF_tags[ind_left_BF]        ▷ Tag
                /* Check whether tag_left_BF is a candidate.              */
                for each tag_BF_candidate ∈ array_candidates do
                    if tag_left_BF == tag_BF_candidate then
                        BF_out ← tag_BF_candidate
                        is_found ← True
                        break

            /* If a candidate has been found, the current loop for can be
               terminated.                                                */
            if is_found == True then
                break
```

```
45: /* Phase 2: if you enter the next if statement, it means that no candidate
       was found in the previous phase. */
46: if is_found == False then
47:     for i = 0, ··· , b − (2 ∗ num_BFs_shift + 1) do
48:         /* Focus on the BF to the left of the median one (i.e. farther from
               t_in_circ), next to all BFs already visited. */
49:         ind_left_BF ← (ind_BF_med − num_BFs_shift − (i+1)) modulo
            b                                                    ▷ Index
50:         tag_left_BF ← array_BF_tags[ind_left_BF]             ▷ Tag

51:         /* Check whether tag_left_BF is a candidate. */
52:         for each tag_BF_candidate ∈ array_candidates do
53:             if tag_left_BF == tag_BF_candidate then
54:                 BF_out ← tag_BF_candidate
55:                 is_found ← True
56:                 break

57:         /* If a candidate has been found, the current loop for can be ter-
               minated. */
58:         if is_found == True then
59:             break

60: return BF_out
```

### 3.3.2 Unknown RTT distribution

In the following we will propose some candidate selection algorithms that can be applied in any circumstance, especially when the RTT distribution is unknown.

#### Search closest candidate

Given the list of candidates, when a incoming packet enters the system, the idea is to compute the time distance between the BF tag associated with the time slot where the incoming packet is observed and the BF tag of each candidate. The candidate that returns the smallest time difference will be considered as the closest one. Algorithm 3 shows the implementation in the form of pseudo-code.

**Algorithm 3** Search closest candidate

**Input:**
- $T$ := time window system;
- $BF_{in}$ := Bloom filter tag where incoming packet is observed;
- $array\_candidates$ := array containing only candidate Bloom filter tags.

**Output:**
- $BF_{out}$ := Bloom filter tag where the outgoing packet is supposed to be stored.

1: */* Initialization. */*
2: $temp\_array \leftarrow$ new array

3: */* Compute time distance between time slot where incoming packet is observed and time slot associated with each candidate BF. */*
4: **for each** $tag\_BF\_candidate \in array\_candidates$ **do**
5: $\quad time\_dist \leftarrow (BF_{in} - tag\_BF\_candidate)$ modulo $T$
6: $\quad$ insert $time\_dist$ into $temp\_array$

7: */* Take the closest candidate, i.e. the one that returned the smallest time difference. */*
8: $ind\_clos\_BF \leftarrow$ array index of the smallest value in $temp\_array$
9: $BF_{out} \leftarrow array\_candidates[ind\_clos\_BF]$

10: **return** $BF_{out}$

### Search farthest candidate

It works similarly to the closest policy, with the only difference that, after computing the time distance between the BF tag associated with the time slot where the incoming packet is observed and the BF tag of each candidate, the chosen candidate will be the one that returned the largest time difference. Hence, the pseudo-code, reported in Algorithm 4, changes from that of the closest policy only for lines 8 and 9.

**Algorithm 4** Search farthest candidate

**Input:**
- $T$ := time window system;
- $BF_{in}$ := Bloom filter tag where incoming packet is observed;
- $array\_candidates$ := array containing only candidate Bloom filter tags.

**Output:**
- $BF_{out}$ := Bloom filter tag where the outgoing packet is supposed to be stored.

```
1: /* Initialization. */
2: temp_array ← new array

3: /* Compute time distance between time slot where incoming packet is
      observed and time slot associated with each candidate BF. */
4: for each tag_BF_candidate ∈ array_candidates do
5:     time_dist ← (BF_in − tag_BF_candidate) modulo T
6:     insert time_dist into temp_array

7: /* Take the farthest candidate, i.e. the one that returned the largest time
      difference. */
8: ind_far_BF ← array index of the highest value in temp_array
9: BF_out ← array_candidates[ind_far_BF]

10: return BF_out
```

#### Take random candidate

In this case there is no criterion, in the sense that a candidate is selected at random from the list of candidates. Thus, it would have been superfluous to report the pseudo-code.

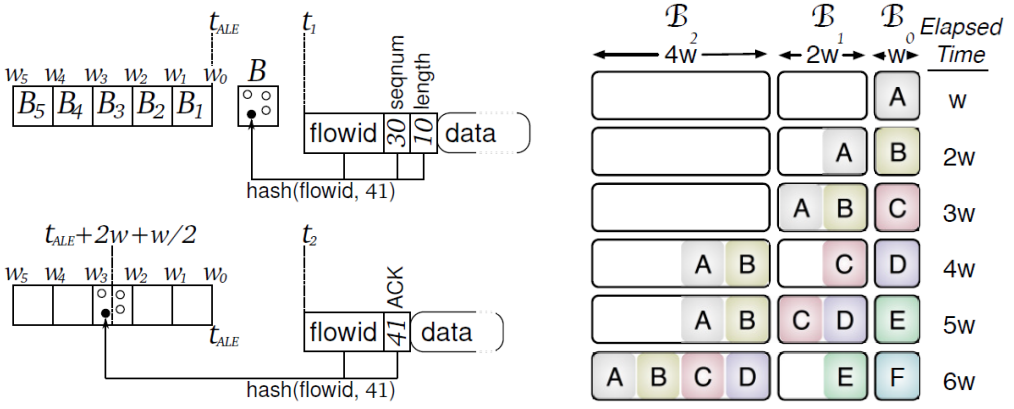
## 3.4 Comparison with other proposals

Network monitoring covers a large research area, therefore it is possible to find a plethora of related works. In this section we report some of them that have dealt with traffic monitoring using time-dependent metrics. In the end, we will make a qualitative comparison between our idea and the discussed proposals, since there was no way to validate them quantitatively.

#### ALE (Approximate Latency Estimator)

Gangam et al. [10] proposed *ALE*, a method for estimating the latency of TCP flows in terms of RTT in real-time. The implementation consists in discretizing the past time with intervals of length $w = w_i - w_{i+1}$ over a sliding window $W = w_0 - w_n$, meaning that $w = w_0 - w_1$ always represents the most recently elapsed interval. The $i$-th interval

(also called bucket $B_i$) is associated with a Counting Bloom Filter (CBF). ALE also uses an additional CBF out of the sliding window to hold the state of the current present (it is indicated simply as $B$). Then, every $w$ seconds, the content of each CBF is moved to the CBF in the bucket on the left, i.e. to its older neighbor, which implies to discard the content of the oldest CBF, since it falls off the sliding window; meanwhile, the bucket holding the present state is reset. Moreover, the hypothetical arrival time of a flow is considered at the middle of the bucket but, to avoid of keeping track of each of the interval boundaries, ALE uses a single reference time $t_{ALE}$ that points to the end of the sliding window (i.e. corresponds to $w_0$) and which is incremented by $w$ after every $w$ seconds. Therefore, an entry stored in $B_i$, would have arrived in the time interval $[t_{ALE} - w \cdot i, t_{ALE} - w(i+1)]$. Fig. 3.8a shows the system just described through an example of insertion when an outgoing packet arrives (upper half) and of lookup when an ACK arrives (lower half). The insertion is performed by hashing both flow ID and expected ACK number (which is one more than the last sequence number in the packet being processed) and incrementing the appropriate counters. For the lookup you need to look backwards in buckets (including $B$) and find the first bucket where the corresponding segment was recorded. In general, when a match is located in $B_i$, the appropriate counters are decremented and the RTT is calculated as $t - t_{ALE} + (2i+1) \cdot w/2$, where $t$ indicates the arrival time of the ACK in the system. Similarly to our solution, the sliding window $W$ should be set to best fit the RTT distribution, in order to collect most of the RTT samples, as well as the interval width $w$ determines the accuracy of the estimate.




(a) Uniform intervals (ALE-U)

(b) Exponential intervals (ALE-E)

Figure 3.8: ALE. Figures reproduced from [10]

The same authors also proposed a variation of ALE with the aim of covering the same window span $W$ with a smaller number of CBFs, considering intervals with amplitude that increases exponentially as you move into the past. To differentiate the two implementations, ALE-U indicates the one that uses buckets of uniform size, while ALE-E the one that uses buckets of exponential size. ALE-E follows the same general idea of moving contents of CBF to its older neighbor, but now this does not involve all buckets simultaneously, rather the content of each bucket is shifted according to an exponential time.

In general, since the $i$-th interval has size $2^i w$, bucket $B_i$ is shifted into $B_{i+1}$ every $2^i w$ seconds. Moreover, when bucket $B_i$ is shifted, its content is merged with the content of bucket $B_{i+1}$ by adding up the corresponding counters, resulting in bitmaps that get more "crowded" and prone to false positives. This is illustrated in Fig. 3.8b. It is immediately evident how ALE-E provides a relative accuracy: estimation of longer latency samples are more inaccurate with respect to smaller latency samples. However, the other side of the coin is the reduced amount of memory needed to cover the same window $W$, since now the number of required buckets is only $1 + \lfloor \log_2(W/w) \rfloor$.

### Dart (Data-plane Actionable Round-trip Times)

Sengupta et al. [17] presented *Dart*, a system that continuously monitor RTTs of TCP flows in real-time. It uses a pipeline of two hash tables to achieve two goals: (1) track packets that can lead to useful RTT samples, since packet retransmissions and packet reordering can lead to RTT ambiguities; (2) identify the synergy between per-flow and per-packet state for efficient memory utilization, since packets that never receive a matching ACK and packets with large RTTs are expensive to keep in memory.

The first hash table is named *Range Tracker (RT)* table. It stores per-flow state in the form of hashed flow ID and measurement range, respectively as key and value. The range is nothing but a sequence number range (in terms of bytes) that can potentially produce correct RTT samples. It is individuated imaging the flow sequence number space divided into three components: bytes already ACKed, bytes in-flight (i.e. transmitted but not ACKed), bytes not seen yet. Then, the measurement range is *[maximum byte ACKed, maximum byte transmitted]*. In this way, if SEQ packets appear in order, the right extreme of the range is moved forward; similarly, if ACK packets arrive in increasing order, the left extreme of the range is moved forward. If instead there are ambiguities, retransmissions are detected when a data packet arrives with bytes smaller than the maximum byte transmitted, while reordering is detected when an ACK packet arrives with bytes smaller than the maximum byte ACKed. When at least one of the ambiguities occurs, the measurement range becomes *affected*, therefore it is collapsed moving the left extreme to the right one, becoming *[maximum byte affected, maximum byte transmitted]* (note that even though the two extremes have different notations, their values are equal to the highest byte transmitted). After the collapse, the measurement range updates regularly, as explained above, which means that in general it is *[maximum byte ACKed OR affected, maximum byte transmitted]*. This mechanism allows to avoid taking RTT samples of flows affected by ambiguities and to ensure collecting only valid samples.

Hence, when a data packet arrives, it is first stored in the RT table, either updating or collapsing the measurement range depending on which case it falls. Then, if the packet is valid, it goes through the second hash table, which is named *Packet Tracker (PT)*. In this case, the key is represented by the hash value of the flow ID combined with the expected ACK, while the value is the associated timestamp. Finally, when an ACK comes back, the corresponding RTT sample is calculated subtracting its timestamp from the timestamp recorded in the entry of the PT table.

In addition, Dart tries to improve memory efficiency by handling unmatched entries that can come from cumulative ACKs and packets with large RTTs. The idea is to

distinguish between the two cases, evicting unmatched entries related to cumulative ACKs (these packets will most likely not receive any ACK), but keeping for a longer time those that may result in large RTTs. To do that, Dart employs a *lazy eviction* strategy that relies on a recirculation for a second chance: *lazy* because the process is triggered only by a hash collision in the PT table, and "second chance" because packets that actually have large RTTs have a second chance to live in the PT table. Basically, every time a new data packet collide with an entry already stored in the PT table, the new entry gets inserted, the old entry is evicted, and its associated packet is sent back to the ingress pipeline to check against the RT table whether it has become stale. If the associated measurement range has been incremented with respect to that packet, then the packet is considered stale and is dropped; otherwise the packet is treated as if it were new, repeating the above process. Finally, to prevent infinite eviction loops, Dart compares the new entry with the currently evicted entry before trying re-insertion and sets a maximum number of recirculations per SEQ packet.

Fig. 3.9 illustrates the working of Dart through an example. Firstly, the new packet for flow F2 arrives with expected ACK 500 at time $t = 80$ (event 1) and, since it is valid, the associated measurement range is updated from $[300, 400]$ to $[300, 500]$ (event 2). Then, when the corresponding packet record with key $(F2, 500)$ and timestamp value 80 arrives at the PT table, it collides with an existing entry $(F3, 600, t = 40)$ (event 3): the new entry gets stored (event 4) while the old entry is recirculated to the RT table for revalidation (event 5).

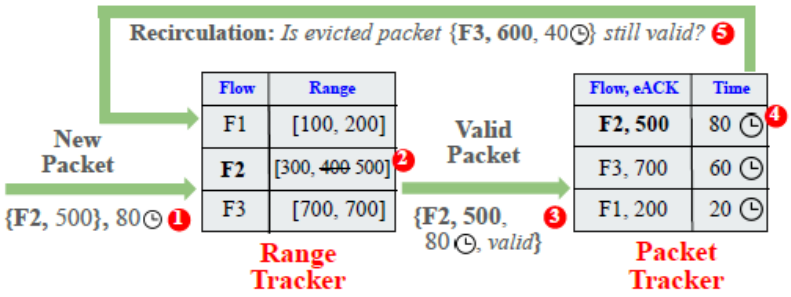



Figure 3.9: Working of Dart. Figure reproduced from [17]

### Other proposals

*BLOOMTIME* [15] is a method to monitor time-dependent metrics of network traffic using basic Bloom filters. Its architecture is similar to that of ALE-U, except for the bucket which holds the state of the current present: instead of using an additional bucket, BLOOMTIME directly stores it into the first (and most recent) bucket of the sliding window. Then, every $w$ seconds, the content of each bucket is moved to its older neighbor, the content of the first bucket is reset, and the content of the last one is discarded. Being implemented with basic Bloom filters, BLOOMTIME does not support deletion in exchange for a smaller memory footprint.

*DDBF (Double Deletion Bloom Filter)* [12] is an extension of the original Bloom filter designed for flow RTT measurements. It consists of extending each cell of the Bloom filter

from one bit to $s_t$ bits to record the SYN arrival time plus $s_c$ bit to count the number of stored SYN packets. In this way, on the arrival of a SYN packet, its timestamp will overwrite the content of each hashed cell and the appropriate counter will be increased by 1. Then, at the arrival of an ACK packet, records can be deleted in two ways according to the difference between its arrival time $t_{sak}$ and the oldest timestamp $t_{ts}$ in the hashed cells: if it is less than a given time span $vt$, it is considered as the RTT of the flow, therefore all the corresponding cells are decremented by 1 (explicit deletion); otherwise the ACK is discarded and only the cells with timestamp greater than $t_{sak} - vt$ are decremented by 1 (implicit deletion). This double-deletion mechanism enables to efficiently delete out-of-data records. The time span $vt$ should be set in order to observe most of the normal RTT samples. Moreover, DDBF provides more accurate estimations of RTT samples at the price of more memory: accuracy and space efficiency can be balanced by properly setting $s_t$ and $s_c$.

Chen et al. [6] proposed an algorithm based on multi-stage hash table that passively and continuously monitors the RTT of TCP traffic. The main limit of a single stage is that each packet has only one chance to be inserted into the table, being discarded if it collides with an entry already stored and not expired yet (only if the entry is older than a given threshold $T\_Expire$ is overwritten). A multi-stage hash table overcomes this limit, giving each packet as many chances as the number of stages to be stored. Each stage is essentially a hash table that stores the hashing of the flow ID combined with the expected ACK (obtained summing the SEQ number with the payload size) as the key, and the timestamp as the value. Note that, to reduce the impact of hash collision, each stage uses a different hash function to calculate the location where storing the record. Hence, every time an outgoing TCP packet arrives, it is stored in the first feasible hash table, checking through the different stages. For instance, if the insertion attempt fails for the first stage (the intended location is already occupied by an entry that has not yet expired), you try again the insertion in the following stages until you find one admissible location. If instead all locations are occupied and unexpired, the considered packet will not be recorded. Then, for each incoming TCP packet, the same hash-based addresses are used to look up every stage until a match is found. If this occurs, you compute the RTT sample subtracting the timestamp of the current packet from the timestamp recorded in entry of the hash table, also deleting the record from it; otherwise no RTT sample is produced.

### Performance comparison

Since authors used different notations to describe their own proposal, we rename the involved parameters with a unique notation in order to make a more natural comparison among them.

For algorithms that relies on the discretization of time, we denote by $b$ the number of data structures associated with intervals of uniform size and $b_e$ the number of data structures associated with intervals of exponential size. Remember that, to cover the same sliding window $W$ and assuming that the smallest interval has width $w$, then $b_e = \lfloor \log_2(W/w) \rfloor < b = W/w$. Also, ALE needs an additional CBF out of the sliding window to hold the state of the current present. Furthermore, for hash table based solutions, $b$

indicates the number of stages it is split into.

Then, we denote by $n$ the number of cells/counters per data structure, $s_c$ the number of bits per cell/counter, $s_t$ the number of bits to store a timestamp, $s_f$ the number of bits to store a fingerprint, $s_{mr}$ the number of bits to store the measurement range. Finally, $k$ indicates the number of hash functions.

For better guidance through the parameters, you should know as follows: $s_c = 4$ bit allows to keep the probability of overflow low for practical values of $n$ [9]; $s_t \in \{20, 32\}$ bit to have a microsecond and a nanosecond timestamp precision, respectively; $s_f$ is set according to the hash probability you would like (more bits to reduce it).

A qualitative comparison among the discussed proposals can be appreciated looking at Table 3.2. For simplicity, it is assumed that RT and PT tables of Dart have the same number of buckets and are not spread over multiple stages. The table confirms that to achieve high accuracy levels along with low time complexities, it is required more space. Therefore, if you are willing to tolerate some inaccuracy for the RTT estimations and to accept higher time complexities, then you could opt for some solution that relies on Bloom filters or derivatives. Moreover, although it is not reported in the table, you should be aware that, with respect to our solution, BLOOMTIME and ALE require an additional time complexity due to shifting buckets every $w$ seconds.

| **Proposal** | **Estimation** | **Memory [bit]** | **Insertion** | **Lookup** | **Deletion** |
|---|---|---|---|---|---|
| Our | Approx. | $b \cdot n$ | $O(k)$ | $O(k \cdot b)$ | $No$ |
| BLOOMTIME [15] | Approx. | $b \cdot n$ | $O(k)$ | $O(k \cdot b)$ | $No$ |
| ALE-E [10] | Approx. | $(1 + b_e) \cdot n \cdot s_c$ | $O(k)$ | $O(k \cdot b_e)$ | $O(k \cdot b_e)$ |
| ALE-U [10] | Approx. | $(1 + b) \cdot n \cdot s_c$ | $O(k)$ | $O(k \cdot b)$ | $O(k \cdot b)$ |
| DDBF [12] | Exact | $(s_t + s_c) \cdot n$ | $O(k)$ | $O(k)$ | $O(k)$ |
| Chen et al. [6] | Exact | $(s_f + s_t) \cdot b \cdot n$ | $O(b)$ | $O(b)$ | $O(b)$ |
| Dart [17] | Exact | $(2s_f + s_{mr} + s_t) \cdot n$ | $O(1)$ | $O(1)$ | $O(1)$ |

Table 3.2: Comparison of different solutions (time complexities in the average case)

# Chapter 4

# Methodology and numerical evaluations

The goal of the following chapter is to test and validate the idea introduced in the previous chapter to passively and continuously monitor the flow RTT. Hence, in Sec. 4.1 we will present our methodology, providing an overview of the simulator implemented in Python. Subsequently, in Sec. 4.2 we will go through the analysis of different results: starting from an evaluation of the scalability of the system, we will move on to its performance when varying the amount of memory and the time slot width, up to the real flow RTT accuracy it can provide.

## 4.1 Overview of the simulator

For our analysis, we employed a discrete-event simulator built on Python[1], a high-level programming language that, among its many uses, includes system testing. In a discrete-event simulation it is crucial to identify *events* and *time steps* in order to represent the system only at specific instants of time. Therefore, time advances through discrete steps, at which an event will change the system state.

We used the *event scheduling* approach to model our discrete simulator. It relies on a data structure called *event list*, which includes all events whose execution time is already known. In our model, the arrival of both outgoing packets and incoming packets represents an event. Therefore, any time a packet is generated according with some distribution (see Sec. 4.1.1), it must be scheduled in the system by adding the sample value drawn by the distribution to the current simulation time. The scheduled time is also called **timestamp** and represents the time when an event must be simulated in the system. Actually, it is also useful to implement the event list by a priority queue, as it allows us to store events in ascending timestamp order. Finally, as a timestamp refers to the time of a simulated environment, it does not have any correlation with the real time. As a consequence, the virtual time advances through the timestamp of the last

[1] https://www.python.org/

event popped from the priority queue.

### 4.1.1 Traffic generation

When you want to analyse a network system, whether it is already existing or not, the choice of the traffic model to use plays a fundamental role. This is because network traffic represents the input to the system, so its mathematical description will impact the final results.

Most traffic models are *open-loop*, this means that the traffic is simply generated by a source but, since our system also expects the response from the destination, a *closed-loop* model would be more appropriate. However, closed-loop models are difficult to setup and manage, as they also require the implementation of a window protocol. To make this easier, we considered the traffic generation as the cascade of two open-loop models: the generation of an outgoing packet also triggers the generation of the associated incoming packet. Essentially, if you suppose to be at the simulation time $t_{now}$ and an outgoing packet of flow $j$ is generated with a sample time $t_x$ drawn from some distribution, you schedule it with the timestamp $t_{out_j} = t_{now} + t_x$ but meanwhile you generate and schedule its incoming packet accordingly. For instance, if the distribution draws a sample time $t_y$ for the incoming packet, it is scheduled at $t_{in_j} = t_{out_j} + t_y$.

As regards the mathematical representation of the traffic, it is harder than it sounds because its generation is very diverse, by varying with the type of network, services and user behaviour. Since in general not all the traffic can be represented by an exact mathematical model, input traffic is often defined through a general description which should be conveniently set up in such a way to generate data that complies with traffic requirements. In the following, we discuss the two open-loop models used for our system.

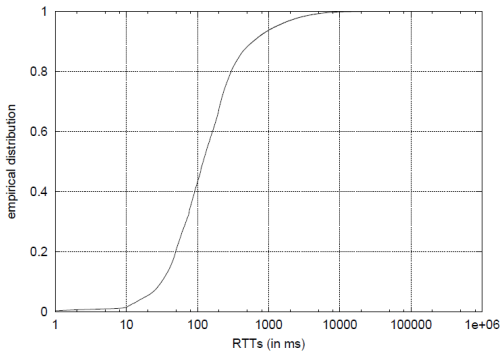
#### Generation model for outgoing packets

The generation of outgoing packets was modelled according to a Poisson process with intensity $\lambda$. This allows us to generate events which are independent of each other (*memoryless* property) and with a constant average inter-arrival time equal to $1/\lambda$.

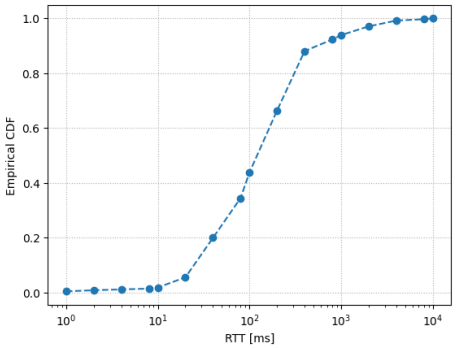
#### Generation model for incoming packets

The model to generate incoming packets must describe the flow RTT. At the beginning we used the uniform distribution to test the system with a smooth traffic. Later, we also used some distribution to test the system with a traffic more in line with reality, namely that takes into account all the aspects involved in a packet transmission over a real network (see Sec. 2.1). For this purpose, there are several empirical distributions that can be drawn from the literature, such as [3,5]. We tested our system with the distribution of Aikat et al., derived by RTT samples empirically collected from over 1 million TCP connections and reported in Fig. 4.1a. Obviously, being an empirical distribution, it is not available in any Python library. Hence, to generate samples from that distribution, we used the CDF inversion technique. As it requires the mathematical knowledge of the empirical distribution $F_X(t)$, we picked some coordinates from Fig. 4.1a to approximately derive it. Picked values are reported in Fig. 4.1 and was used to approximately reconstruct the

original empirical CDF (Fig. 4.1b). At this point, the inverse CDF technique consists of generating a value $u = U \sim (0, 1)$ and to find the corresponding sample $t_k$ of the empirical CDF such that the condition $F_X(t_{k-1}) < u \leq F_X(t_k)$ is satisfied. To be noticed that the support of the empirical RTT distribution is $[0, 10]$, meaning that $t_{max} = 10$ seconds, which is fundamental to properly set other system parameters, as we know from Sec. 3.2.1.




(a) Original. Figure reproduced from [3]




(b) Approximated

Figure 4.1: Empirical CDF

| **X** *[ms]* | 1 | 2 | 4 | 8 | 10 | 20 | 40 | 80 | 100 |
|---|---|---|---|---|---|---|---|---|---|
| $\mathbf{F_X(t)}$ | 0.005 | 0.009 | 0.012 | 0.015 | 0.019 | 0.056 | 0.2 | 0.345 | 0.438 |

| **X** *[ms]* | 200 | 400 | 800 | 1000 | 2000 | 4000 | 8000 | 10000 |
|---|---|---|---|---|---|---|---|---|
| $\mathbf{F_X(t)}$ | 0.664 | 0.881 | 0.923 | 0.939 | 0.971 | 0.992 | 0.997 | 1 |

Table 4.1: Empirical CDF

### 4.1.2 Packets and Bloom filter implementation

Firstly, we used Python's built-in *list* data structure for packet implementation. In particular, for each generated packet, it was sufficient to create a list of type *[flow ID, timestamp, bool]*, where:

- *flow ID* represents the flow identifier, in whatever form it is defined. Since flows are typically bidirectional, it is the same for both the outgoing packet and the associated incoming packet. Furthermore, since it is taken as input of the hash functions, we set it as an integer that is increased by one unit each time an outgoing packet is generated. Obviously, this is just a simplification.
- *timestamp* is the time when the packet must be simulated in the system. It is obtained as explained in Sec. 4.1.1.

- *bool* is a boolean variable that allows to distinguish between outgoing packet and incoming packet of the same flow.

Regarding the entire data structure composed of Bloom filters, it was implemented with the *dictionary* data structure built into Python. It stores data in $\{key : value\}$ pairs, which fits well with the probabilistic data structure we have defined. Indeed, the *keys* of the dictionary were set according to BF tags and each key was associated as value with the relative Bloom filter, each implemented by using the *bitarray* object of the *bitarray*[2] Python library. When a packet needs to be simulated, we compute $k_{opt}$ hash functions to hash the associated flow ID and get $k_{opt}$ Bloom filter positions, which are used for insertion or for lookup operations according to whether the packet is outgoing or incoming: in the first case it is necessary to mark the bits in the positions of the Bloom filter which is covering the interval which includes the timestamp of the considered outgoing packet; in the second case it is necessary to check the values of the bits in the positions of each Bloom filter to identify the candidates associated with the considered incoming packet. If the lookup of a specific Bloom filter returns a *True* answer, the associated tag (dictionary *key*) is inserted into a list that collects all the candidates of the current flow, which is later used for selecting a candidate (according to some algorithm proposed in Sec. 3.3) and calculating accordingly the flow RTT.

Also, the *bitarray* object provides the *count()* method, which was used to get the amount of bits set to 1 within the Bloom filter (i.e. $n_1$) before being reset and to empirically calculate its probability of false positive as follows:

$$P(FP)_{emp} = \left(\frac{n_1}{n}\right)^{k_{opt}} \tag{4.1}$$

### Generating $k_{opt}$ hash functions

For generating hashes, we used the MD5 algorithm available in the Python *hashlib* module[3]. It takes as input a bytes-like object and outputs a 128 bit hash value (also called *digest*). Although MD5 is a cryptographic hash function, it has been found to suffer from vulnerabilities, so nowadays it is mainly used for non-cryptographic purposes due to its speed in calculating hashes.

In our system, after encoding the flow ID using *encode("utf-8")*, we send it to *md5()* to generate a MD5 hash object. Then, for generating $k_{opt}$ independent hashes, we exploit the *update()* method of the hash object. Basically, each time the method is called, the digest is updated simply concatenating the new string. To perform this operation repeatedly, we enter a *for* loop where, at each iteration, the MD5 hash objected is updated with the encoded version of the current loop counter $i$. For instance, if $flowID = 5$ and the loop counter $i = 0$, then the new MD5 object will be computed on the encoded string 50; at the next iteration, since the loop counter becomes $i = 1$, the MD5 objected will be updated through the encoded string 501; and so on. Obviously, after each update of

[2]https://pypi.org/project/bitarray/

[3]https://docs.python.org/3.11/library/hashlib.html

the MD5 hash object, this must be first converted to hexadecimal format and then to integer, from which the remainder must be calculated via the modulo operation to obtain a position of the Bloom filter in the range $[0, n-1]$.

### 4.1.3 Parameter setting

#### Packet generation

The first parameter to setup is certainly the intensity rate of the Poisson process, as it represents the rate according with are generated packets. We initially simulated the system with different rates, i.e. $\lambda \in \{500, 1000, 2000\}$ packets/second, and with a rather large time granularity variation, i.e. $b \in \{4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048\}$. This helped us to verify how much the system scales under ideal conditions, namely when, for each flow, it is possible to identify only the real Bloom filter where the outgoing packet was stored. This condition was simulated by considering a relatively large amount of memory $N$.

It turned out that a good value of $\lambda$ to evaluate system performance is 1000 pkts/sec. Although a detailed explanation is given in Sec. 4.2.1, we can summarize saying that a low intensity combined with a system made up of many Bloom filters involves a waste of memory resources, while an increase of the intensity does not allow them to be exploited to the fullest, but rather it leads to a higher probability of false positive per Bloom filter.

#### System configuration

As we are dealing with a time-based system, the setting of $\lambda$ has a strong impact on its behaviour. Indeed, it determines the average number of flows $E[m]$ observed in each time slot, whose width $\Delta$ will in turn impact on a series of parameters that depend on each other, namely the optimal number of hash functions $k_{opt}$, the average occupancy (in terms of bits set to 1) per Bloom filter $E[n_1]$ and its probability of false positive $P(FP)$:

$$E[m] = \lambda \cdot \Delta \quad \text{[pkts]} \tag{4.2}$$

$$k_{opt} = \frac{n}{E[m]} \ln(2) \tag{4.3}$$

$$E[n_1] = n - E[n_0] = n - n \cdot e^{-\frac{E[m] \cdot k_{opt}}{n}} \quad \text{[bits]} \tag{4.4}$$

$$P(FP) = (0.6185)^{\frac{n}{E[m]}} \tag{4.5}$$

We denoted with $n_0$ and $n_1$ the number of bits respectively equal to 0 and 1 of the Bloom filter. Moreover, the expression of $E[n_0]$ - which represents the expected number of bit set to 0 inside a Bloom filter - within (4.4) was derived from *Appendix A* of [18].

As you can see from (4.2), once the intensity rate $\lambda$ is fixed, the average number of flows per time slot depends on $\Delta$. Hence, to simulate the behaviour of the system in various contexts, measuring the corresponding performance and accuracy, we run out one simulation for each combination of $\Delta$ and $N$. In particular, to evaluate system performance when $\lambda = 1000$ pkts/sec, we considered:

$$b \in \{4, 8, 16, 32, 64, 128, 256, 512\}$$
$$N \in \{2^{14}, 2^{15}, 2^{16}, 2^{17}, 2^{18}, 2^{19}, 2^{20}\} \quad [bit]$$

We always considered a data structure with a power of 2 memory, varying from $2^{14} \simeq 16$ kbit to $2^{20} \simeq 1$ Mbit. As a consequence, to perfectly distribute the total amount of memory between Bloom filters, it would make sense to have an even number of them. This premise allows us to explain why the minimum number of Bloom filters simulated is 4: starting from a smaller number (i.e. $b = 2$) would imply $\Delta = t_{max}$, so an insignificant system for our aim. Similarly, an amount of memory smaller than $N = 2^{14}$ bit would imply less than 1 hash function per flow, which would once again be meaningless.

Looking at the other extreme, a further increase in memory does not bring any benefit, since simulations showed that system performance for $N = 2^{20}$ bit is already optimal, in the sense that it guarantees at most one candidate per flow. Furthermore, a greater thinning of time does nothing but increasing the probability of false positive per Bloom filter, worsening system performance especially for low values of memory $N$.

**Simulation length**

Finally, we need to establish the simulation time $t_{sim}$. It was tuned according with the maximum threshold to compute the RTT $t_{max}$, which should be set according to the support of the distribution used for generating incoming packets. Therefore, we always set $t_{max} = 10$ seconds, referring so to the upper extreme of the empirical RTT distribution of Fig. 4.1b even when the uniform one is used. Note that this implies to have a uniform distribution with the same support as the empirical one. This is done in order to fairly compare the system when it is tested by using different RTT distributions. As a consequence, we set $t_{sim} = 10 \cdot t_{max} = 100$ seconds because we though it might be sufficient to observe the system in a steady state for a relatively long time. In fact, with this setting, after an initial transient phase of $T$ seconds, the system will enter its steady state which will last for the remaining $t_{sim} - T \simeq 90$ seconds.

For the sake of completeness, but without any impact on what has been already explained, from the intensity rate $\lambda$ and the simulation time $t_{sim}$ you can derive the average total number of flows $E[M]$ injected in the system, i.e. $E[M] = \lambda \cdot t_{sim} = 10^5$.

### 4.1.4 Evaluation metric

We evaluate RTT accuracy on the basis of the Average Absolute Error (AAE) and of the Average Relative Error (ARE):

$$AAE = \frac{1}{M} \sum_{i=1}^{M} AE_i = \frac{1}{M} \sum_{i=1}^{M} |\hat{RTT}_i - RTT_i| \quad [sec] \tag{4.6}$$

$$ARE = \frac{1}{M} \sum_{i=1}^{M} RE_i = \frac{1}{M} \sum_{i=1}^{M} \frac{AE_i}{RTT_i} \tag{4.7}$$

where:

$$RTT_i = t_{in_i} - t_{out_i} \quad [sec] \tag{4.8}$$

$$\hat{RTT}_i = \begin{cases} (3.7) & \text{if RTT distribution is unknown} \\ (3.9) & \text{otherwise} \end{cases} \tag{4.9}$$

Basically, the AAE is defined as the sum of the absolute error of $i$-th flow ($AE_i$) - each given by the absolute difference between the estimated value ($\hat{RTT}_i$) and the true one ($RTT_i$) - divided by all observed flows. Similarly, the ARE is defined as the sum of the relative error of $i$-th flow ($RE_i$) - each given by the absolute errors ($AE_i$) divided by the true value ($RTT_i$) - all averaged over the observed flows.

Remember that the accuracy of the estimated value $\hat{RTT}_i$ depends both on the time slot width $\Delta$ (the shorter it is, the higher the error) and on whether or not the selected candidate Bloom filter is the true one. Moreover, note that when the RTT distribution is known, $\hat{RTT}_i$ corresponds exactly to the reference time of (3.9) because the latter is calculated assuming $t_{in_i}$ as the lower extreme of the distribution, as explained in Sec. 3.3.1.

### 4.1.5 Statistics collection

Once the simulation is run, we started collecting statistics only after the end of the transient phase, i.e. after the first $T$ seconds, since it is the time necessary to insert flows in every Bloom filter of the system at least once. Furthermore, apart from collecting per-flow RTT metrics, we also collected statistics related to each Bloom filter, in order to evaluate system performance. In particular, before resetting each Bloom filter, we stored its context in terms of occupancy (number of injected flows and number of bits set to 1) and probability of false positive. Once the simulation was completed, for each system configuration and for each statistic, we calculated its overall average through two phases: firstly, we obtained the average associated with each Bloom filter, averaging the values stored before each reset for the entire duration of the simulation; secondly, the average values of the previous step were in turn averaged to obtain the overall average of the considered statistic. For the sake of completeness, we also tried to obtain the overall average through an alternative procedure, again made up of two phases: we first averaged the values stored before each reset of every Bloom filter in each time window $T$, and finally we averaged those values. It turned out that both procedures lead to the same results.

## 4.2 Result analysis

Before discussing the accuracy of the system in evaluating the flow RTT, it was appropriate to perform a preliminary simulation which led us to reasonably set the parameters for our objective. It allowed us to see how scalable the system is in terms of flows, necessary to select a single arrival rate and to reduce the range of $N$ and $\Delta$ with which to test the system in the subsequent phases. Actually, the two preliminary phases also revealed some unexpected aspects of the system itself.

### 4.2.1 Scalability evaluation

The first step concerns the test of the system against different intensity rates in terms of flows injected per second, in order to see how much it scales. This is accomplished by imposing regular traffic on the system - namely generating incoming packets according to a uniform distribution - and using an amount of memory large enough to guarantee a single candidate per flow - namely the Bloom filter where the outgoing packet was actually stored - even with a high time granularity. After several tries, it was found that the minimum amount of memory to satisfy our requirements is $N = 2^{20}$ bit (1 Mbit), as you can see looking at Fig. 4.2e. Fig. 4.2 groups all the statistics of the system obtained by using an amount of memory equal to 1 Mbit and by varying intensity rates, i.e. $\lambda \in \{500, 1000, 2000\}$ pkts/sec. Apart from the average number of candidates, for all the other statistics both empirical and analytical results are reported, in order to verify whether the system complies with the theory.

Firstly, the analytical optimal number of hash functions to minimize the probability of false positive - reported in (4.3) - must be rounded to the nearest integer. Furthermore, developing the equation, it becomes:

$$\begin{aligned} k_{opt} &= round\left[\frac{n}{E[m]}\ln(2)\right] \\ &= round\left[\frac{\frac{N}{b}}{\lambda \cdot \Delta}\ln(2)\right] \\ &= round\left[\frac{\frac{N}{b}}{\lambda \cdot \frac{t_{max}}{b-1}}\ln(2)\right] \\ &= round\left[\frac{N}{\lambda} \cdot \underbrace{\frac{(b-1)}{b}}_{[0.75,0.99]} \cdot \underbrace{\frac{\ln(2)}{t_{max}}}_{0.069}\right] \end{aligned} \tag{4.10}$$

The trend of the curves in Fig. 4.2a is in line with (4.10), since the optimal number of hash functions is inversely proportional to the arrival rate. Once the latter is fixed, since both $N$ and $t_{max}$ are given, the optimal number of hashes per flow is almost constant, except when the time granularity is thick, as the ratio $(b-1)/b$ progressively decreases with it, until it becomes 0.75 when $b$ is minimum.

Secondly, Fig. 4.2b confirms (4.2), according with the average number of flows per Bloom filter linearly increase with the arrival rate. At the same time, once the latter is fixed, flows stored in each Bloom filter decrease as the time granularity of the system becomes thinner. Surprisingly, the same cannot be said for the average occupancy per Bloom filter. In fact, although it is always decreasing with the growth of the time granularity, it does not vary with the intensity rate of the Poisson process. This behaviour can be explained by analysing (4.4). At first glance, it would seem to depend on the Bloom filter size, the average number of flows per Bloom filter and the optimal number of hash functions used to store them. However, substituting the expression of $k_{opt}$ of (4.3) into (4.4), we obtain:

$$
\begin{aligned}
E[n_1] &= n - n \cdot e^{-\frac{E[m]}{\cancel{n}} \cdot \frac{\cancel{n}}{E[m]} ln(2)} \\
&= n - n \cdot e^{-ln(2)} \\
&= n - n \cdot \frac{1}{2} = \frac{n}{2} = \frac{N}{2 \cdot b}
\end{aligned} \tag{4.11}
$$

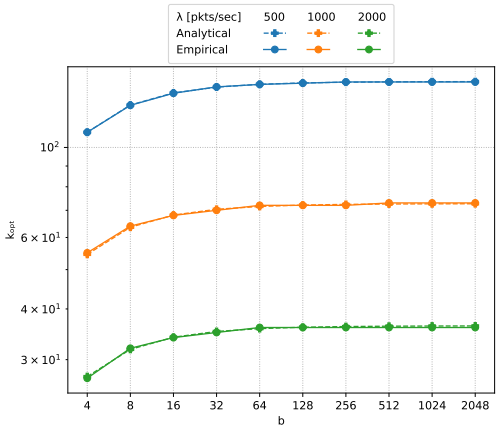
This result is very interesting. It tells us that, by using the optimal number of hash functions per flow, the average occupancy of the Bloom filter is always equal to half of its memory, regardless of how many items you store inside it. But not only that: since $n = N/b$ and $b = 2^x$ - with $x \in [2, 11]$ -, once the amount of memory $N$ is fixed, the average occupancy per Bloom filter is inversely proportional to $2^{x+1}$.

Another interesting result comes from the average probability of false positive. Looking at Fig. 4.2d, you can see how empirical curves - obtained through (4.1) - are not consistent with analytical ones - obtained through (4.5) -, assuming a non monotonic behaviour which becomes more marked with the reduction of the intensity rate. This is due to the correlation between the process of arrival flows and the time granularity of the system. Indeed, although the Poisson process guarantees on average an inter-arrival time equal to $1/\lambda$, its infinitesimal definition states that the probability of an event in a small time interval is proportional to its length. Therefore, since an increase in the number of Bloom filters results in a reduction of the time slot duration $\Delta$, the probability of observing flows within it is consequently reduced, especially by decreasing the intensity rate. This phenomenon, along with variable hash collisions, results in having Bloom filters that are not loaded uniformly. As a consequence, the overall average probability of false positive is biased, being more sensitive to extreme values relative to the more loaded Bloom filters. Nevertheless, as it possible to see from Fig. 4.3, the empirical curves never exceed those relating to their upper limit, which are computed by imaging that all flows within a time window $T$ are always stored inside a single BF, i.e.:

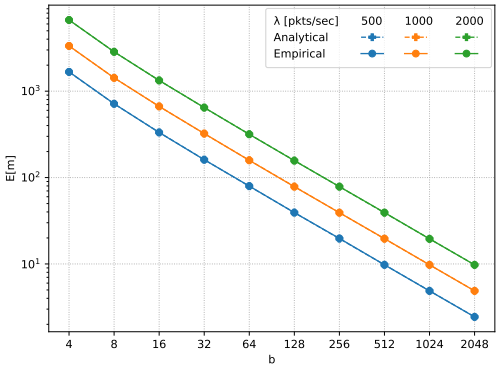
$$
P(FP)_{UB} = (0.6185)^{\frac{n}{\lambda \cdot T}}
$$

Returning to the Poisson process property, it can be appreciated looking at Fig. 4.4, which portrays the Bloom filter contexts during the entire simulation. In particular, it refers to the third Bloom filter of the data structure (BF tag $= 2 \cdot \Delta$) under different time granularities. Firstly, you can see how the reset described in Sec. 3.2.5 works. Once the transient phase is over, the BF is first reset and then starts storing flows for the following $\Delta$ seconds. Subsequently, its context is frozen for the next $T$ seconds, at which time the procedure just described is repeated until the end of the simulation. And secondly, you can see how, between the various time windows, the load of the Bloom filter becomes increasingly unbalanced as the time granularity increases. The apex is reached when $\Delta$ is minimum, since there are some time windows during which no flows are stored in the Bloom filter - i.e. time windows $[40, 50]$, $[60, 70]$ and $[90, 100]$ seconds -, resulting in a waste of memory resources for the system.

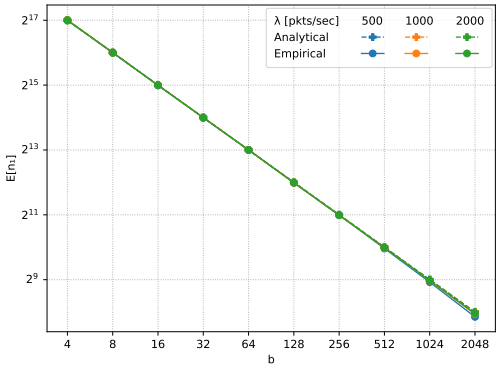
In the light of what has been observed, we can state that, in general, the system is not suitable when the time is too finely fragmented due to an increase in the probability



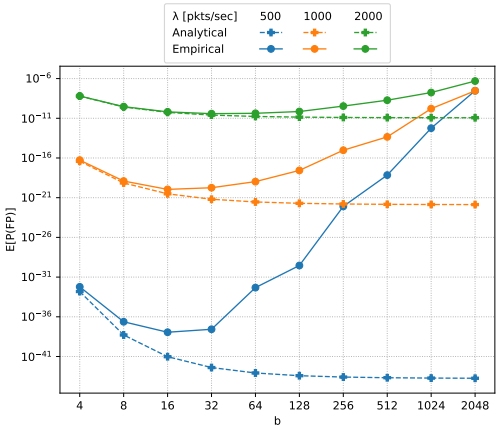

(a) Optimal number of hash functions per flow



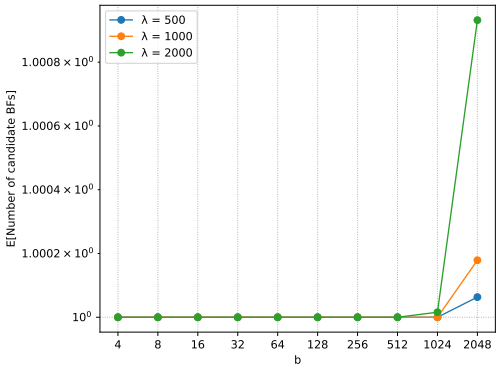

(b) Average number of flows per BF




(c) Average occupancy per BF




(d) Average probability of false positive per BF




(e) Average number of candidates per flow

Figure 4.2: System statistics when $N = 2^{20}$ bit and $\lambda \in \{500, 1000, 2000\}$ pkts/sec.

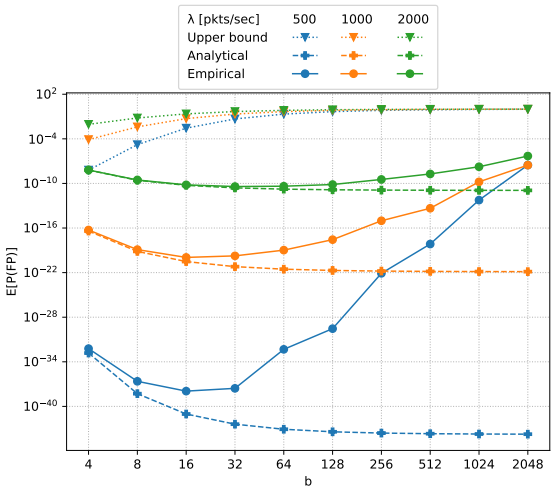
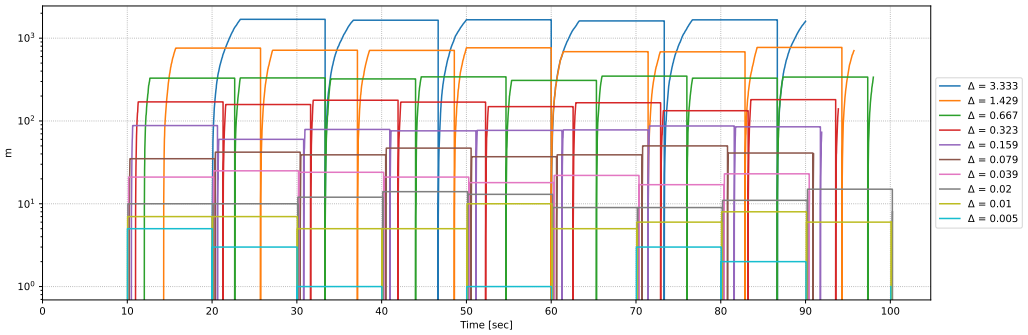
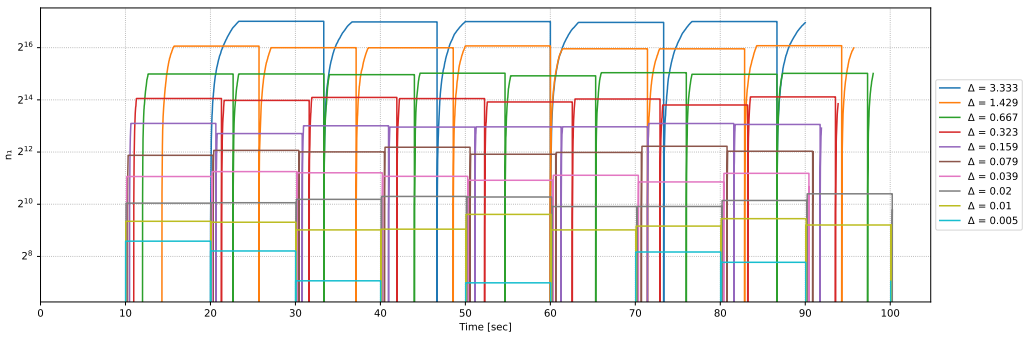



Figure 4.3: Average probability of false positive per BF: analytical upper bound (dotted lines) vs analytical average (dashed lines) vs empirical average (solid lines)




(a) Number of flows




(b) Number of bits set to 1

Figure 4.4: Occupancy over time of BF with tag $= 2 \cdot \Delta$, when $N = 2^{20}$ bit and $\lambda = 500$ pkts/sec

of false positive per Bloom filter, to which is added a waste of memory when flows enter the system with a low intensity. However, avoiding working under the above conditions, the system can be considered to be scalable, as it can track up to 2000 pkts/sec with only 1 Mbit of memory, providing a relatively small probability of false positive per BF.

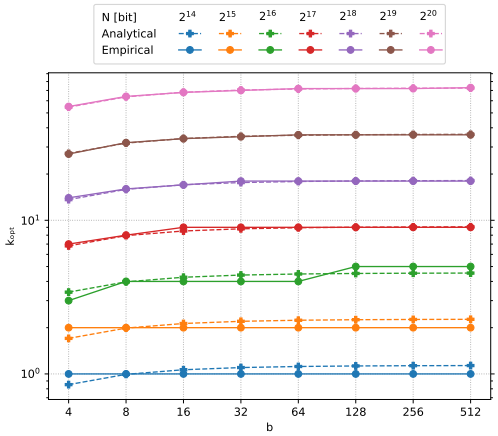
### 4.2.2 Time-space evaluation

Next, we started to evaluate the system by playing with different combinations of time-space setting. Since the results above analysed refer to a system with a memory sufficient to guarantee at most one candidate per flow, we expect that as memory decreases, the system may be affected with an increase in the probability of false positives and candidates per flow. Therefore we believe that $\lambda = 1000$ pkt/sec could be the right compromise to test the system not only in terms of performance based on the time-space variation, but also in the next phase, in which it will be evaluated in terms of accuracy in measuring the flow RTT. Furthermore, on the basis of what said in Sec. 4.1.3, we have narrowed the range of time-space values to $b \in \{4, 8, 16, 32, 64, 128, 256, 512\}$ and $N \in \{2^{14}, 2^{15}, 2^{16}, 2^{17}, 2^{18}, 2^{19}, 2^{20}\}$ bit, respectively. The obtained results are grouped in Fig. 4.5.

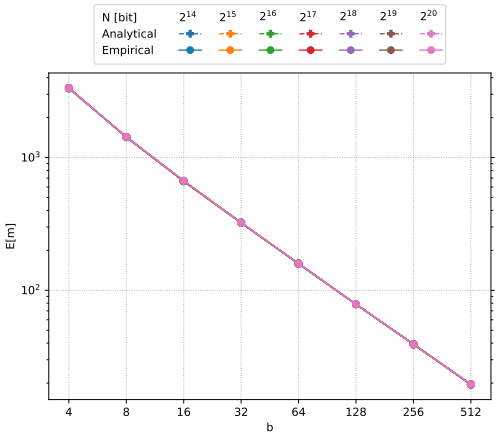
Starting from the optimal number of hash functions (Fig. 4.5a), the obtained curves are a further confirmation of (4.10). Indeed, since now $\lambda$ is given while $N$ varies, $k_{opt}$ grows linearly with the latter. Furthermore, it is still true that, for the same value of $N$, it decreases slightly as the time granularity becomes thicker.

Let's move on the average results related per Bloom filter (Fig. 4.5b to 4.5d). Firstly, the average number of flows injected into each BF does not vary with the amount of memory $N$, but only depends on the time granularity of the system, as confirmed by (4.2). Secondly, the average number of bits equal to 1 are directly proportional to $N$ and inversely proportional to $b$, perfectly in agreement with (4.11). By the same equation, once $N$ is fixed, the average occupancy halves every time the number of Bloom filters doubles. As regards the probability of false positives, it emerges once again that the empirical results are not always consistent with the analytical ones: they are pretty consistent for $N \leq 2^{17}$, then they differ progressively as both memory and number of Bloom filter increase, assuming a non-monotonic trend. This is due to the fact that the optimal number of hashes and the relative collision probability are proportional to the amount of memory which, in addition to the correlation between the Poisson process and the time granularity of the system (explained above), results in having unbalanced Bloom filters and therefore more skewed probability of false positives. It is also interesting to notice that, in those cases, it is possible to individuate the optimal number of Bloom filter to use in order to minimize the probability.

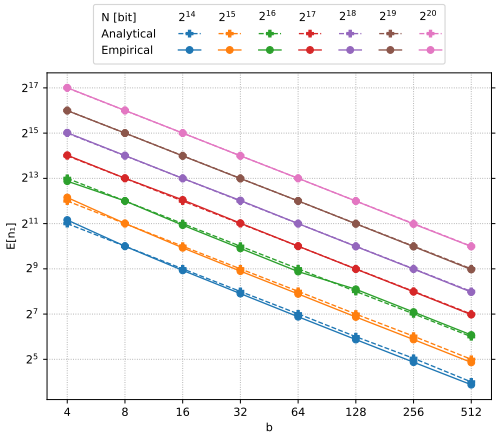
Finally, Fig. 4.5e shows the average number of candidates per flow. Essentially, the smaller the memory footprint, the more candidates there are due to a higher probability of false positive per Bloom filter. It is worth noting that, for the least available memory, the number of candidates is nearly half of the available number of Bloom filters, therefore we expect it to return the worst accuracy for RTT measurements. On the other hand, since an increase in memory implies a reduction in candidates, we expect lower RTT measurement errors.



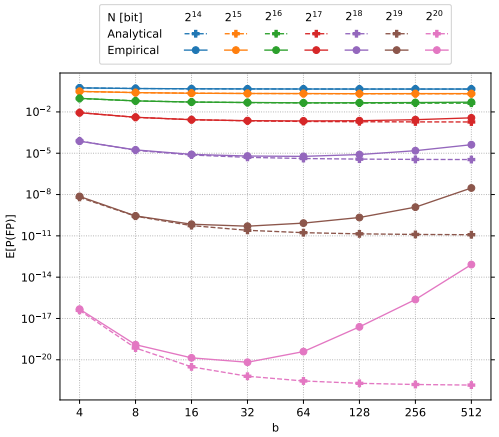

(a) Optimal number of hash functions per flow



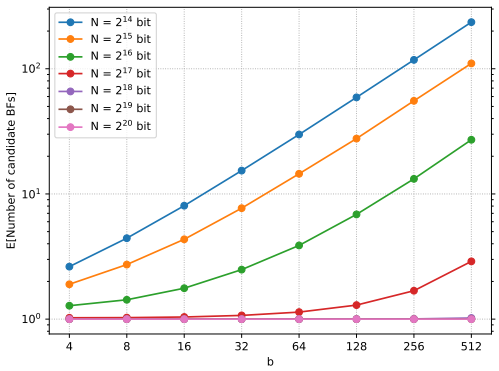

(b) Average number of flows per BF




(c) Average occupancy per BF




(d) Average probability of false positive per BF




(e) Average number of candidates per flow

Figure 4.5: System statistics with time-space variability: $N = \{2^{14}, 2^{15}, \cdots, 2^{20}\}$ bit and $b = \{2^2, 2^3, \cdots, 2^9\}$

### 4.2.3 RTT accuracy

To evaluate the RTT measurement accuracy, each configuration system was run 10 times using different seeds to test the system with different populations of timestamps. In the end, we calculated the confidence interval at the 95% confidence level, by exploiting the Python's *SciPy* library to compute Student's t distribution[4] values. This procedure was repeated twice, once generating incoming packets with the uniform distribution, another with the empirical distribution of Fig. 4.1b.

Furthermore, for each system configuration, the flow RTT was calculated choosing a candidate according to each policy explained in Sec. 3.3 in order to establish which one provides the best accuracy. Obviously, to have a yardstick of comparison, we also estimated the flow RTT taking the real Bloom filter where the associated outgoing packet was actually stored. It essentially represents the systematic error of the system.

In the following, we report a comparison of flow RTT obtained using the uniform and the empirical distribution, both in terms of AAE and ARE. Note that, to avoid having graphs that are particularly full of curves and difficult to read, we have restricted the comparison to 3 intermediate memory values, i.e. $N \in \{2^{15}, 2^{17}, 2^{19}\}$ bit. We excluded the two extremes of the original range of $N$ because of what we have seen in the previous subsections: for $N = 2^{14}$ bit, the system suffers from a high probability of false positive, making it difficult to apply in real applications; for $N = 2^{20}$ bit, the system performs optimally. Last remark: for each error metric, the y-axis of the different graphs was set with the same coordinate limits for a better comparison.

#### Absolute average errors (AAE)

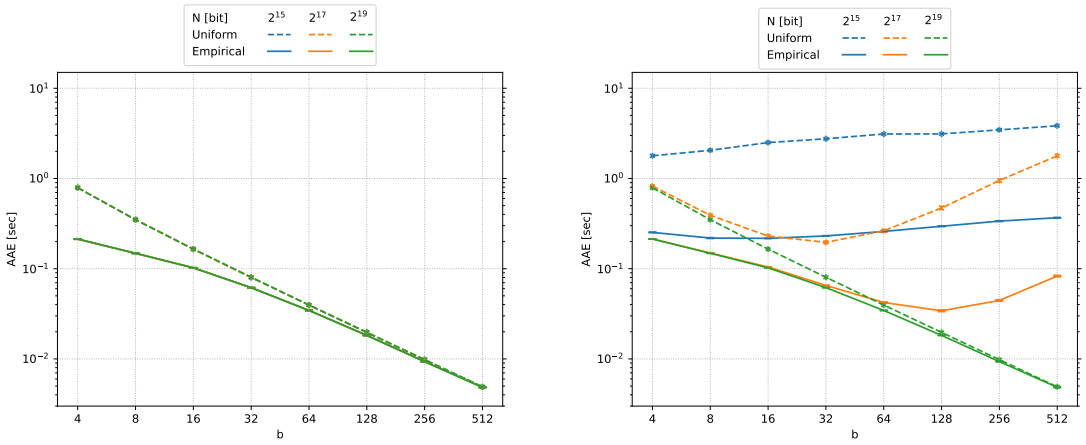
Let's start the discussion from Fig. 4.6a, which shows the absolute errors when you always know (ideally) which BF actually contains the outgoing packet of the considered flow. The first observation is that, when using the uniform distribution, the average error is $\frac{\Delta}{4}$, regardless of the memory footprint. Indeed, since the uniform distribution is symmetric, its skewness is null, therefore the hypothetical arrival time of outgoing packets is supposed to be at half of the time slot. As a consequence, the RTT measure is $\frac{\Delta}{2}$ if the outgoing packet arrives at one of the time slot borders and it is 0 if it arrives exactly in the middle of it. By averaging these two cases, the error is exactly $\frac{\Delta}{4}$. This result is also confirmed by ALE-U [10]. Instead, the absolute errors obtained with the empirical distribution are less than $\frac{\Delta}{4}$ for thick time granularity, due to a reference time that varies according to the truncated distribution relative to each flow to fit with the asymmetrical nature of the distribution itself. For instance, for $b = 4$, the empirical distribution provides on average an error smaller than 580 ms with respect to the uniform one. However, this effect is mitigated as the time granularity becomes finer. Not surprisingly, regardless of the RTT distribution, the accuracy is improved by increasing the number of Bloom filters, since the time slots become thinner.

By analysing the results obtained selecting a candidate with the exploitation of statistical properties of the distribution (Fig. 4.6b to 4.6d), there are no significant differences

[4] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.t.html

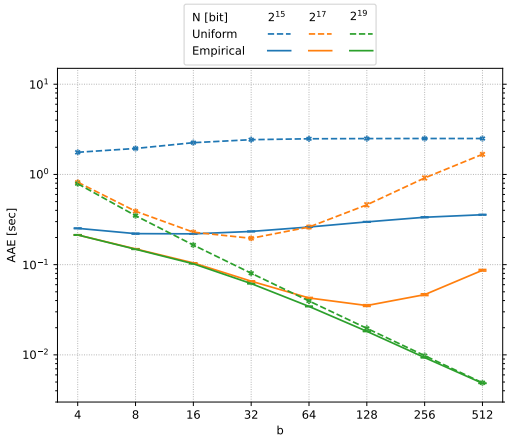
in the use of one policy rather than another, except when using the *search around mean* policy which, for $N \in \{2^{15}, 2^{17}\}$ bits and a finer time granularity system, gives slightly worse error if RTT samples are drawn by the empirical distribution. However, regardless of the policy adopted, the key point is that, given the amount of memory, the empirical distribution guarantees a smaller absolute error. Another interesting aspect is that curves for $N = 2^{17}$ bit are non-monotonic, meaning that you can individuate the minimum number of Bloom filters that minimizes the average error. In particular, the RTT measured with the empirical distribution deviates almost 35 ms from the true value when using 128 Bloom filters, while it deviates roughly 190 ms with 32 Bloom filters and uniform traffic. If instead you want to even reduce the discrepancy of the measurement from the true value, you need to increase the amount of memory (e.g. $N = 2^{19}$ bits) and to distribute it among a number of Bloom filters that varies according to the RTT distribution (e.g. $b \geq 64$ for empirical samples).

Let's move on to the policies that can be used in all circumstances (Fig. 4.6e to 4.6g). It is worth noting that generally, if the RTT is uniform, the average error does not suffer from the policy you adopt and from the amount of memory you use. The only exception is $N = 2^{15}$ bit: choosing at random among a large set of candidates typically leads to a slightly minor error. Instead, for RTT modelled according to real traffic, choosing the candidate closest in the recent past always yields smaller errors, due to the right-skewness characteristic of the distribution. Suffice is to say that 44% of the RTT samples are lower than 100 ms, which becomes lower than 400 ms by doubling the frequency of observation of the samples, while only 0.06% of them are in the range $[1, 10]$ seconds. As a consequence, the selection of the farthest candidate implies greater errors, while a random selection provides intermediate errors. Finally, we are surprised that the average errors obtained with the *search closest* policy follow the ones obtained with the *search most likely* and the *search around median* policies, even here except for the uniform case with $N = 2^{15}$ bit, in which they are slightly higher. However, for a natural comparison between policies, and therefore to establish which supplies the best accuracy, we need to analyse the relative errors.

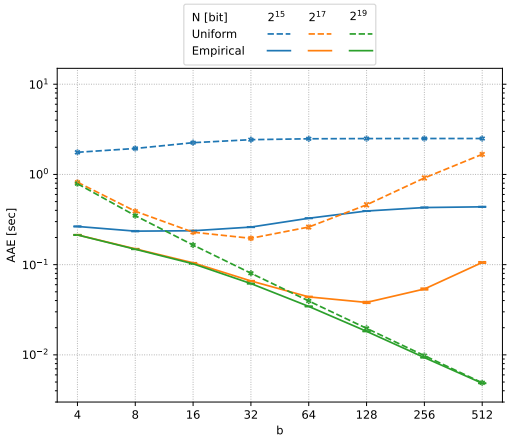



(a) Take true BF

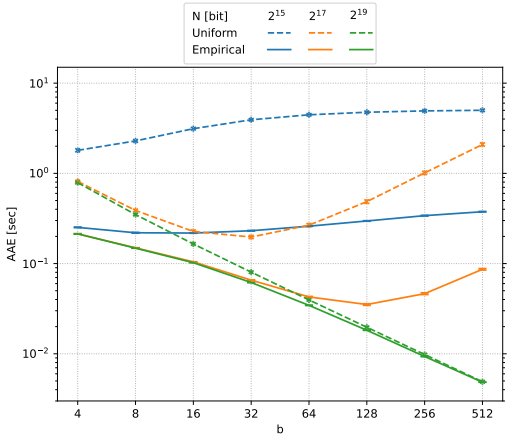
(b) Search most likely BF



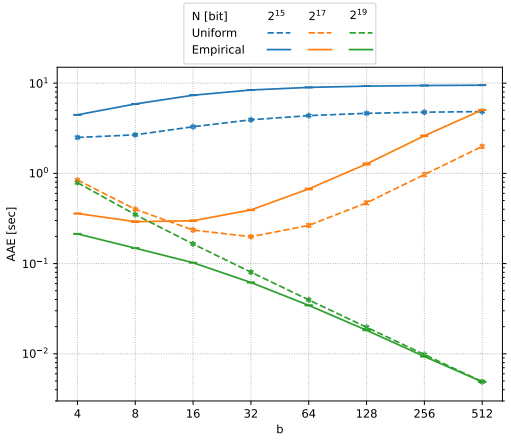

(c) Search BF around median



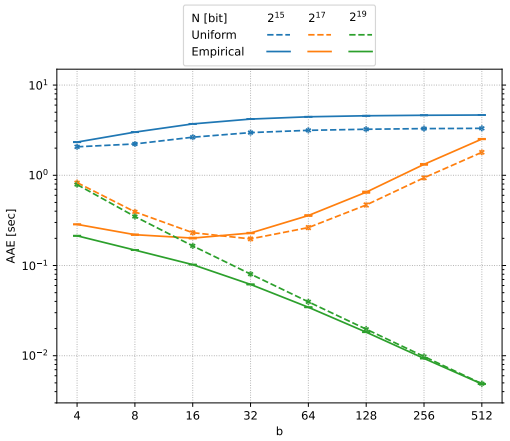

(d) Search BF around mean




(e) Search closest BF




(f) Search farthest BF




(g) Take random BF

Figure 4.6: Average absolute error with confidence intervals at the 95% confidence level

### Average relative errors (ARE)

In this last section, we identify the best policy that guarantees the best RTT sample accuracy.

Results are grouped in Fig. 4.7. First off all, what immediately catches the eye is the instability of the system against uniform RTTs, confirmed by large confidence intervals for each configuration and for each policy adopted to select a candidate. This behaviour is dictated by a larger variance of the uniform distribution over the support considered, which reflects on a higher standard error of the sample means. Despite this, analysing the results obtained in the ideal case (Fig. 4.7a), the accuracy results better with the uniform population. Intuitively, this should be traced back to the characteristics of the two distributions. Indeed, based on the definition of ARE in (4.7), we would expect that the relative error of each flow is greater for empirical samples, since the true RTT value is on average 11.7 times smaller than that of the uniform case (i.e. $E[X]_{emp} = 0.427$ sec; $E[X]_{unif} = 5$ sec). Therefore, being the true RTT value in the denominator of the formula, this will lead to a higher ARE.

Secondly, it is not surprising that more memory availability implies better accuracy, regardless of the RTT distribution and how a candidate is chosen. Indeed, looking at curves obtained with $N = 2^{19}$ bit, since the number of candidates is essentially reduced to one, the accuracy only depends on the time granularity of the system. Given to assure this point, what instead is really interesting is the outcome of the closest policy (Fig. 4.7e), whose accuracy exceeds that of any other policy. To justify this, we further analysed the results of our simulations, identifying another metric of comparison. We define the *Average Percentage of Failure (APF)* as the sum of the ratio between the number of flows for which a policy chooses a wrong candidate ($M_w$) and the total number of observed flows ($M$), all averaged over the number of times the simulation was run ($N_{sim}$):

$$APF = \frac{1}{N_{sim}} \sum_{i=1}^{N_{sim}} \frac{M_w}{M}$$

Results are illustrated in Fig. 4.8 and show that the general trend is a function that increases with the number of candidates. Notably, for the same amount of memory $N$ and the same RTT distribution, the behaviour of the *search most likely* and *search closest* policies exhibits no significant differences (Fig. 4.8a vs Fig. 4.8d), confirming they are the most precise policies in finding the true BF. This result is understandable for the empirical RTT distribution, as its probability density is mostly concentrated around the left tail, meaning that searching in increasing order of probability associated with each BF typically degenerates in searching the BF in the more recent past, especially when the number of candidates is reduced. With the increase of the latter, instead, the policy that searches for the most likely BF becomes progressively more precise than the closest one, thanks to the exploitation of statistical properties. This is why, if you carefully compare the two, the search most likely policy significantly outperforms the other when going from $b = 256$ to $b = 512$ for $N = 2^{15}$ bits, decreasing by about 4.7%. Plus, being the empirical distribution right-skewed, searching a candidate around the median is more appropriate than searching around the mean, as you can see by comparing Fig. 4.8b to Fig. 4.8c. In fact, the median is much less sensitive to extreme values than the mean, which instead

is more biased towards the right tail because of them. As a consequence, selecting the oldest candidate in the past turns out to be the worst option (Fig. 4.8e), followed by a random selection (Fig. 4.8f).

In the light of what we have just said, we can now move on to explain why selecting the closest candidate provides a better RTT accuracy than selecting the most likely one. It was found that the system suffers from a *border effect*, which can be defined as follows (note that, for convenience, we will use as subscripts $pol_1$ and $pol_2$ to refer to results obtained by two different policies):

**Border effect**

**IF**:

1. outgoing packet and incoming packet of the same flow arrive in two adjacents time slots
2. BFs associated to those time slots are both candidates and are chosen by two different policies
3. $\hat{RTT}_{pol_1} < \hat{RTT}_{pol_2}$

**THEN**: the BF that minimizes the error is the one chosen by policy 1, regardless whether it is true, i.e. $RE_{pol_1} < RE_{pol_2}$

In other words, when the true BF precedes another candidate (false positive) which is the most recent in the past with respect to the incoming packet, it is chosen by the closest policy (i.e. policy 1) by leading to underestimate the error if the real arrival time of the outgoing packet is closer to the reference time of the wrong candidate rather than to the one of the true BF. By analysing one simulation instance in the worst case in terms of number of candidates per flow, i.e. when the system is deployed with $N = 2^{15}$ bit, it turned out that the *search closest* policy is more affected by the border effect than the *search most likely* one. In particular, by adopting the *search closest* policy, we observed that the number of flows affected by this effect progressively reduces when the system goes from being composed of 4 to 512 Bloom filters and the RTT samples are uniformly distributed, whereas it follows an opposite trend when they are drawn from an empirical distribution. However, in this thesis work we have not further explored this aspect, therefore we cannot justify this trend.

In addition to this phenomenon, we also observed that the *search most likely* policy produces a set of outliers which becomes progressively larger with respect to that of the *search closest* policy as the time granularity becomes thinner, especially for uniform RTT samples. These types of events lead to an increase in the ARE of the *search most likely* policy with respect to that of the *search closest* one. Also in this case, the reason for this behavior has not been studied in this thesis work.

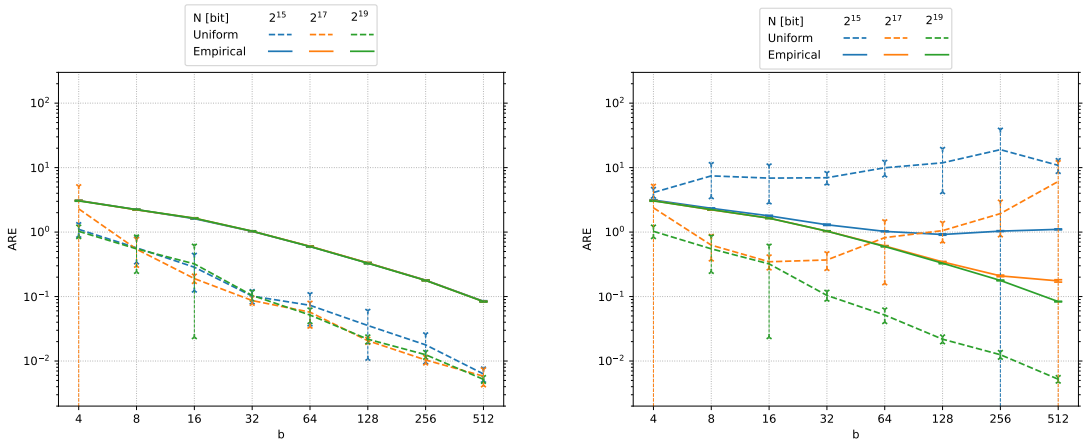
Therefore, if on the one hand the precision between the two policies in finding the true BF can be considered nearly comparable, on the other hand the closest policy provides a better RTT accuracy because of the two aforementioned phenomena.

Now that you are aware of the altered accuracy of the *search closest* policy, deflated by underestimated RTT samples, we can quantify how far it deviates from the accuracy of the *search most likely* policy, focusing on the most relevant points (Fig. 4.7b vs Fig. 4.7e). For empirical distribution, the curves for $N = 2^{17}$ bit follow the same path till $b = 128$, then the ARE of the search most likely policy increases, deviating from the search closest one by a factor of 1.49 when $b = 512$. As regards the curves for $N = 2^{15}$ bit, the first sign of weakness begins to appear with $b = 16$; for $b \geq 128$ the two policies provide almost parallel curves, but the best gain is obtained when $b = 256$, with a factor of 1.93. Plus, it can be seen that both policies give a little nod of best accuracy when using $b = 128$ (most likely) and $b = 256$ (closest).

Moving on to the results obtained with the uniform distribution, you can see that choosing the closest candidate gives a non-monotonic ARE for $N \in \{2^{15}, 2^{17}\}$ bit, optimizing the accuracy respectively for $b \in \{32, 128\}$, while choosing the most likely one does the same only for $N = 2^{17}$ bit and $b = 16$ but loosing in accuracy by a factor of 1.74 if you take the same system configuration and you adopt the closest policy. Furthermore, for $b \in \{8, 16, 32\}$, the closest policy is able to provide the best accuracy by using only $N = 2^{17}$ bit.

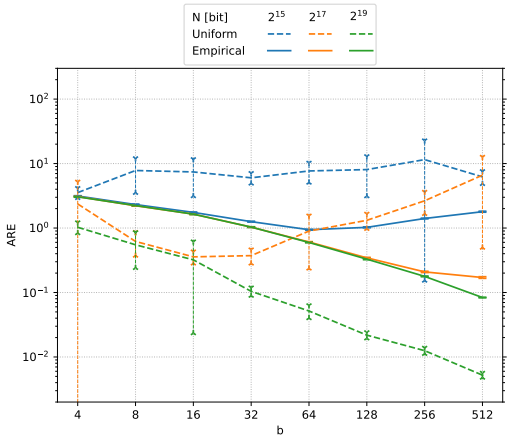
Other interesting points derive from the direct comparison between the two distributions for the same policy, as it is possible to identify some points in which the ARE obtained with one distribution exceeds that obtained with the other. In particular, when the closest candidate is chosen, the empirical ARE becomes smaller than the uniform one for $N = 2^{15}$ bit and $b \geq 64$ (the best gain is when $b = 256$ with a factor of 1.74), but also for $N = 2^{17}$ bit and $b = 512$ by a factor of 2.82. Similarly, if you search for the most likely BF, the accuracy obtained with empirical samples becomes better than the accuracy obtained with uniform ones for $N = 2^{17}$ bit and $b \geq 64$ (the best gain is when $b = 512$ with a factor of 34.85).

Finally, we believe it is superfluous to discuss the results of other policies, given that they perform worse. They are reported just for a matter of completeness.

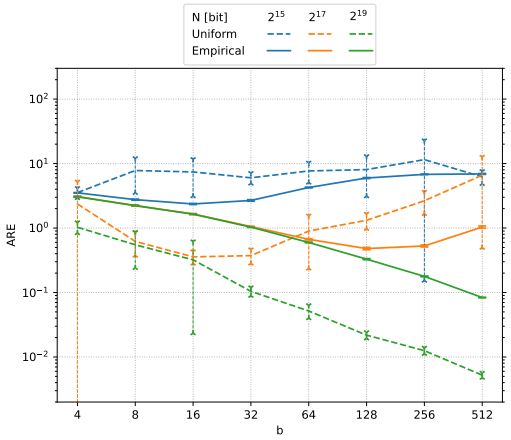



(a) Take true BF

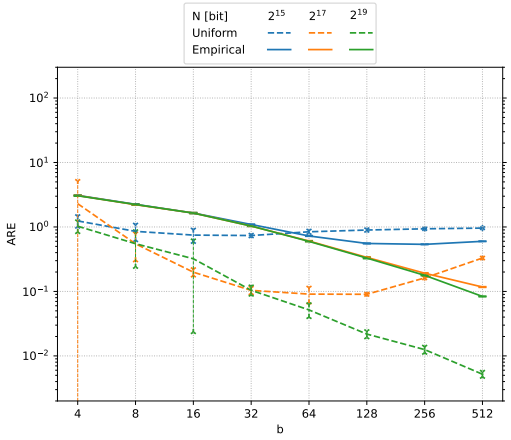
(b) Search most likely BF



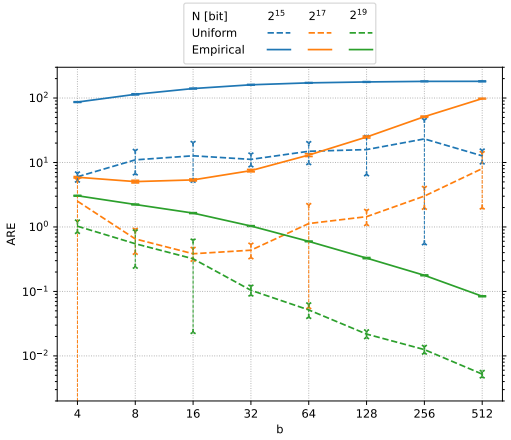

(c) Search BF around median



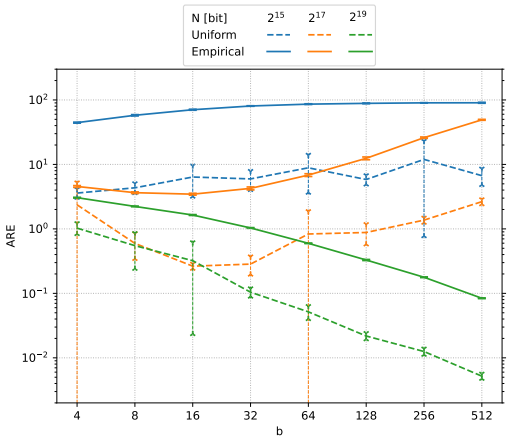

(d) Search BF around mean




(e) Search closest BF

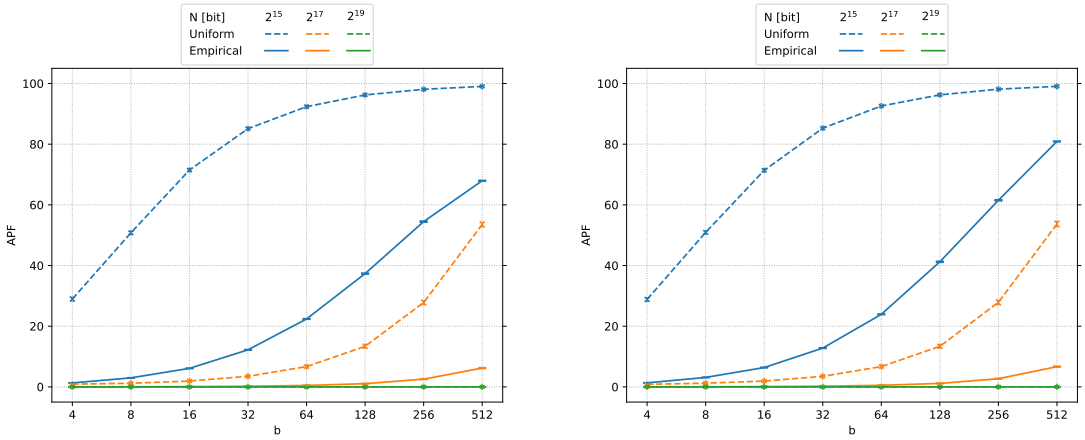



(f) Search farthest BF




(g) Take random BF

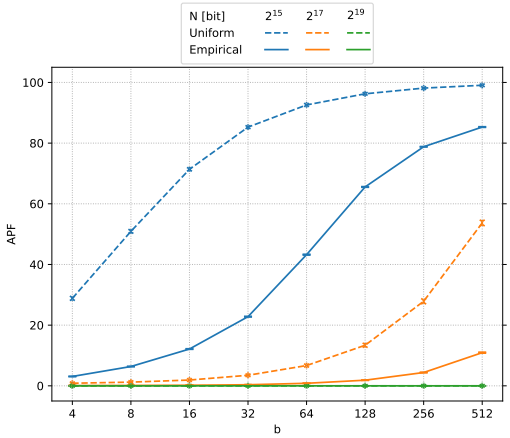
Figure 4.7: Average relative error with confidence intervals at the 95% confidence level



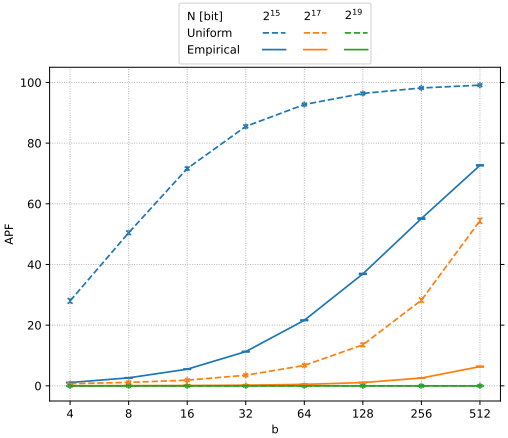

(a) Search most likely BF



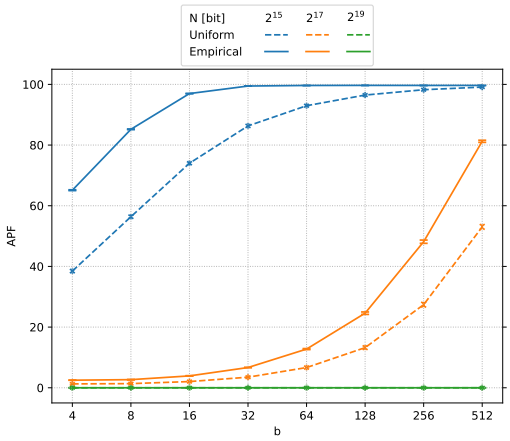

(b) Search BF around median



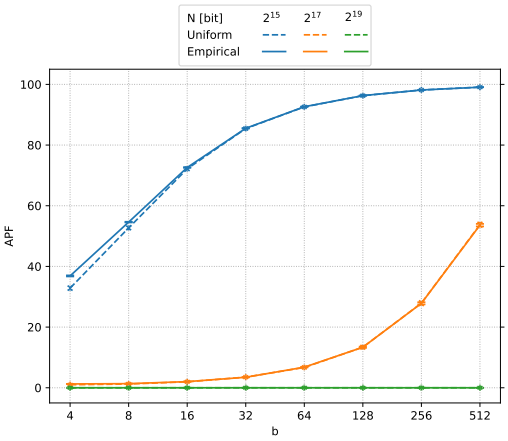

(c) Search BF around mean




(d) Search closest BF




(e) Search farthest BF




(f) Take random BF

Figure 4.8: Average percentage of failure with confidence intervals at the 95% confidence level

# Chapter 5

# Conclusion

The advent of SDN has introduced the possibility of implementing real-time network monitoring solutions by exploiting data-plane programmable switches. However, the main challenges arise from the limited resources of programmable switches, which are facing with processing and storing of a huge number of simultaneous flows at the link-rate of today's networks. To this end, equipping programmable switches with probabilistic data structures could be one possible solution, as they typically require low memory and reduced computational time.

In this thesis work we proposed a solution for passively monitoring flow RTT using an array of Bloom filters evenly spaced across time slots over a given time window. Although the conditions in which the system was tested were rather stringent, assuming flows always passing through the measurement point and ignoring aspects related to network protocols, the solution showed good potential.

First of all, it was observed that the solution is relatively scalable. If the arrival rate of the packets at the measurement node is too low, it is essential that the system does not have a too fine time granularity to fully exploit the available memory. At the same time, the arrival rate must not be particularly excessive because otherwise we could run into Bloom filters with a high probability of false positives.

Secondly, it was observed that, by fixing the arrival rate, distributing a greater amount of memory to a given number of Bloom filters results in a decrease in false positive events, and therefore in a decrease in the number of candidates to choose for the calculation of the flow RTT.

To identify the average error introduced by the system and evaluate its accuracy in calculating the RTT, for each search policy, we conducted simulations with several space-time configurations and two RTT distributions, on the basis of which we set the reference arrival time of outgoing packets within a time slot. We demonstrated that the systematic error is on average 4 times smaller than the time slot width when the reference time of the outgoing packet corresponds to its intermediate value, but it can be reduced by setting the reference time on the fly according to the arrival time of the incoming packet and the RTT distribution. Moreover, the best time-space configuration that minimizes the average absolute error is dictated by both the traffic pattern and the search policy. For instance, adopting either one of the policies based on statistical properties or the

search closest policy, the system can minimize the absolute error distributing $2^{17}$ bit among 128 Bloom filters if the RTT samples follow an empirical distribution. However, selecting the most recent candidate in the past turns out to be the best in terms of accuracy regardless of the traffic pattern, although this result is affected by a border effect from which the system suffers, which may introduce underestimated RTT samples. For example, subjecting the system to an empirically derived RTT distribution and using the search closest policy, you can guarantee the same accuracy of the ideal case using only $2^{15}$ bit, provided that the time configuration is set with at most 32 Bloom filters; to further improve the accuracy without deviating from the ideal case, the system needs more memory and finer time granularity, such as $2^{17}$ bit distributed among 128 Bloom filters or $2^{19}$ bit among a number of Bloom filters equal or greater than 256.

## 5.1 Future works

The solution we proposed is in a primitive state, as it was tested through a simulation methodology under stringent hypotheses that neglected some typical characteristics of networks, such as asymmetric routing and protocols. It also showed some signs of weakness, including the border effect and some outliers produced in case the search most likely policy is adopted. Therefore, in the future, these aspects might be investigated further before implementing the solution in a programmable switch and evaluating it in a real packet network.

Furthermore, the proposed solution is versatile and applicable in different use cases. For instance, it might be used to measure any time-dependent metric in the field of network monitoring or for smart mobility applications (e.g. tracking the movements of smart devices through sensors distributed over a geographical area to achieve a specific road traffic target).

# Bibliography


[1] Cryptographic and non-cryptographic hash functions. URL: https://dadario.com.br/cryptographic-and-non-cryptographic-hash-functions/.

[2] What is network monitoring? URL: https://www.ibm.com/topics/network-monitoring.

[3] Jay Aikat, Jasleen Kaur, F. Donelson Smith, and Kevin Jeffay. Variability in tcp round-trip times. In *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, IMC '03, page 279–284, New York, NY, USA, 2003. Association for Computing Machinery. doi:10.1145/948205.948241.

[4] Jawad Alkenani and Khulood Nassar. Network monitoring measurements for quality of service: A review. *Iraqi Journal for Electrical and Electronic Engineering*, 18:33–42, 12 2022. doi:10.37917/ijeee.18.2.5.

[5] Mark Allman. A web server's view of the transport layer. *SIGCOMM Comput. Commun. Rev.*, 30(5):10–20, oct 2000. doi:10.1145/505672.505674.

[6] Xiaoqi Chen, Hyojoon Kim, Javed Aman, Willie Chang, Mack Lee, and Jennifer Rexford. Measuring tcp round-trip time in the data plane. pages 35–41, 08 2020. doi:10.1145/3405669.3405823.

[7] Damu Ding, Marco Savi, Federico Pederzolli, and Domenico Siracusa. Design and development of network monitoring strategies in p4-enabled programmable switches. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, page 1–6. IEEE Press, 2022. doi:10.1109/NOMS54207.2022.9789848.

[8] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, page 75–88, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2674005.2674994.

[9] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '98, page 254–265, New York, NY, USA, 1998. Association for Computing Machinery. doi:10.1145/285237.285287.

[10] Sriharsha Gangam, Jaideep Chandrashekar, Italo Cunha, and Jim Kurose. Estimating tcp latency approximately with passive measurements. volume 7799, pages 83–93, 03 2013. doi:10.1007/978-3-642-36516-4_9.

[11] Paolo Giaccone. Efficient data structures for high speed packet processing, November 2020. https://www.telematica.polito.it/app/uploads/2018/07/

probabilistic-data.pdf.

[12] Xinjie Guan, Xili Wan, Ryoichi Kawahara, and Hiroshi Saito. An online framework for flow round trip time measurement. In *Proceedings of the 2013 25th International Teletraffic Congress (ITC)*, pages 1–4, 2013. doi:10.1109/ITC.2013.6662968.

[13] Hao Jiang and Constantinos Dovrolis. Passive estimation of tcp round-trip times. *SIGCOMM Comput. Commun. Rev.*, 32(3):75–88, jul 2002. doi:10.1145/571697.571725.

[14] Ivan Marsic. *Computer Networks: Performance and Quality of Service*. Rutgers University, 2013.

[15] Racyus Pacífico, Lucas Bragança, Gerferson Coelho, Pablo Silva, Alex Borges, Marcos Vieira, Ítalo Cunha, Luiz Vieira, and José Miranda Nacif. Bloomtime: space-efficient stateful tracking of time-dependent network performance metrics. *Telecommunication Systems*, 74, 06 2020. doi:10.1007/s11235-020-00653-1.

[16] Davide Sanvito. *Traffic Management in Networks with Programmable Data Planes*, pages 13–23. Springer International Publishing, Cham, 2021. doi:10.1007/978-3-030-62476-7_2.

[17] Satadal Sengupta, Hyojoon Kim, and Jennifer Rexford. Continuous in-network round-trip time monitoring. pages 473–485, 08 2022. doi:10.1145/3544216.3544222.

[18] Kyu-Young Whang, Brad Vander Zanden, and Howard Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Trans. Database Syst.*, 15:208–229, 06 1990. doi:10.1145/78922.78925.