On Simulating Bloom Filters in the ndnSIM Open Source Simulator

M. Tortelli*, G. Piro, L.A. Grieco, G. Boggia

DEI, Dep. of Electrical and Information Engineering, Politecnico di Bari – v. Orabona 4 – 70125, Bari, Italy

Abstract

Name Data Networking (NDN) is a novel information-centric architecture, based on networking primitives that are driven by hierarchical content names. It promises to ease the development of content sharing services, simplify the management of mobile applications across heterogeneous wireless technologies, and enable native multicast and multipath communication. In this context, simulation tools play an important role because they offer the possibility to easily evaluate, even in large scale scenarios, the performance of new protocols, algorithms, and design methodologies for the NDN architecture. Among the several simulators that are available nowadays, ndnSIM, which is a module of the well known NS-3 open source framework, can be considered as the most complete one due to its accurate representation of all the facets of the NDN architecture. To further broaden its scope, an extension that models and simulates Bloom filters is proposed in this manuscript. As a matter of fact, the adoption of Bloom filters in NDN represents an active research branch, for which an open source simulation platform is still missing. The proposed extension allows the simulation of several types of Bloom filters for different purposes, such as membership check of locally cached contents, and/or name lookup in forwarding strategies designed for information-centric architectures. This manuscript provides all the details of the new software module (from the design criteria to an accurate de-

*Corresponding author

Email addresses: m.tortelli@poliba.it (M. Tortelli), g.piro@poliba.it (G. Piro), a.grieco@poliba.it (L.A. Grieco), g.boggia@poliba.it (G. Boggia)

Preprint submitted to Elsevier

May 1, 2015
scription of the code), as well as results gathered from the simulation of several use cases and from a scalability test.

*Keywords:* Bloom filter, NDN, NS-3, ndnSIM, Simulation and Modeling

1. Introduction

The Information-Centric Networking (ICN) paradigm aims at evolving the classic host-centric Internet design to better support nowadays applications [1]. In ICN systems, networking primitives are driven by *content names* rather than IP addresses. This conceptually simple change of perspective allows ICN architectures to natively support seamless user mobility, content sharing services, multipath routing, and multicast communications [2].

The term ICN encompasses several architectures, such as Data-Oriented Network Architecture (DONA), Publish Subscribe Internet Technology (PURSUIT), Network of Information (NetInf)/Scalable and Adaptive Internet Solutions (SAIL), CONVERGENCE, Content-Centric Networking (CCN)/Name Data Networking (NDN), COntent Mediator architecture for content-aware nETworks (COMET), and MobilityFirst [2][39]. Despite some distinctive differences (e.g., content naming schema, security-related aspects, routing strategies, and cache management), they share a receiver-driven communication model, which is based on content names [3]. ICN-related topics are being widely investigated to assess pros and cons of the different architectures, as well as to conceive optimized protocol stacks. In this field, the CCN/NDN is the one that has been attracting the greatest attention of the scientific community. This is testified by the large number of projects and research instruments related to CCN/NDN, which include softwares for experimental evaluations, such as CCNx [6], NDN Forwarding Daemon (NFD) [7], CCN-lite [12], CCN-Joker [13], a real testbed [5], and network simulators, such as ndnSIM [14], ccnSim [15], and Icarus [34]. Regarding CCN/NDN simulators, they differ from each other not only for the adopted programming language, but also for their level of fidelity, modularity, and scalability [37]. Unfortunately, despite the availability of a wide set of
features, they do not include software components that simulate Bloom filters [16].

Bloom filters are space-efficient probabilistic data structures, able to compactly represent large data sets with a tunable false positive probability [16]. They have already been exploited in IP networks for several purposes, such as implementing efficient longest prefix matching operations [24], filtering out unnecessary accesses to slow off-chip memories in order to speed up lookup procedures [30], and keeping a summary of contents that are stored in a distributed system of web proxies [22]. More recently, many scientific contributions (i.e., [35, 18, 19, 25, 17, 36, 20, 38]) explored the adoption of Bloom filters in ICN systems, with the intent of making both the representation of locally cached contents and name-based routing protocols more efficient (details are reported in Sec. 2.3).

By simulating Bloom filters inside NDN nodes, the ndnSIM extension proposed herein enables the cross-comparison of currently available and future solutions. To the best of the authors’ knowledge, this contribution represents the first open source tool modeling Bloom filters in NDN. At the present stage, this tool\(^1\) already provides: (i) the possibility to use several types of Bloom filters, (ii) an easy way to instantiate them in a NDN node, and (iii) efficient functionalities which allow users to adopt Bloom filters for different purposes, such as the representation of locally cached contents and/or the creation of Bloom filter-based Forwarding Information Base (FIB) tables.

To demonstrate the consistency and the flexibility of the proposed module, two different use cases are investigated: the first one is dedicated to the impact that the sizing of the filter has on its false positive probability; the second one, instead, is focused on the implementation of a name-based routing protocol for NDN. In this latter case, a comparison with respect to other routing strategies that do not use Bloom filters, i.e., flooding and shortest path, is presented. Obtained results show that the use of Bloom filters can dramatically reduce

\[^1\]Available at http://telematics.poliba.it/BFndnSIM.
the network overhead. In addition, they highlight how a Bloom filter-based forwarding strategy can provide the same performance of a shortest path-based algorithm, even without filling FIB tables with preemptively calculated routes toward seed copies. To provide a further insight, we, also, carry out a strong scalability test which assesses computational needs, expressed in terms of simulation time and memory usage. It is proved that the limited computational cost, along with the flexibility of this new module, can make it very useful for the NDN research community.

The rest of the paper is organized as in the following: in Section 2, key aspects of both NDN and Bloom filters are discussed. Then, in Section 3, design criteria of the presented tool, along with a detailed description of its set of functionalities, are presented. Two significative use cases, as well as a scalability test, are examined in Section 4. In the end, conclusions and future upgrades of the proposed module are discussed in Section 5.

2. Background

2.1. NDN Architecture

In NDN, each content is divided into chunks that are uniquely identified by hierarchical names, which, in turn, directly guide packet forwarding, thus avoiding the use of IP addresses [11]. Nodes interact through a receiver-driven communication model, which means that contents are generated (and delivered) only in response to the relative requests. In particular, only two types of messages are exchanged between NDN users: Interest and Data packets. Their processing inside NDN nodes is handled by three main structures:

- Content Store (CS);
- Forwarding Information Base (FIB);
- Pending Interest Table (PIT).

The CS is a cache memory where received/forwarded contents can be stored. Different replacement policies can be adopted, such as Random, Least Recently
Used (LRU), and Least Frequently Used (LFU). This distributed in-network caching capability may reduce the load of the original content providers and allow users to retrieve nearby cached copies. The FIB is similar to the one used in IP nodes, except for the fact that its entries contain hierarchical content names with variable lengths, instead of IP addresses with a fixed length.

To natively support multipath routing and efficiently fetch data from multiple sources at the same time, multiple faces can be associated to a single FIB entry. The PIT is used, instead, to keep track of Interest packets that have been previously forwarded toward content sources and that still remain unsatisfied. Its information are adopted to build the reverse paths through which Data packets are forwarded to their requesters. It emerges that routing and forwarding operations are executed only for Interest packets. Furthermore, NDN easily supports multicast communications. In fact, requests for the same contents can be aggregated at each node into one PIT entry (i.e., the one created after the forwarding of the first Interest), keeping trace of the respective incoming faces.

An example of a NDN communication session is reported in Fig. 1. A consumer generates an Interest packet to retrieve a particular content. When the Interest is received by a NDN node, the CS is firstly looked up to verify if the request can be locally satisfied. If so, the requested Data will be sent back; otherwise, the PIT is looked up to check if the same Interest has been previously forwarded but it is still unsatisfied. In this case, the received Interest is discarded, and its arrival face is added to the incoming faces of the matched PIT entry. On the contrary, if a matching entry is not found inside the PIT, the FIB is examined in order to find potential routes to forward the Interest through. To this end, a Longest Prefix Match (LPM) operation is executed on the hierarchical name of the requested content. This means that the Interest packet will be forwarded through the interface that is associated with the prefix providing the highest number of matched components. If any entry is not found after the FIB lookup, instead, the received Interest is definitively discarded. Once the Interest packet reaches a node that can satisfy such a request, a corresponding Data packet will be generated and forwarded back toward the
Figure 1: A sample NDN communication session.

Description:
(1) The Consumer generates an Interest for the /domain/content.
(2) Router A does not find the content in the CS, forwards the request to Router B according to the FIB, and updates the PIT.
(3) Router B does not find the content in the CS, forwards the request to the Publisher according to the FIB, and updates the PIT.
(4) The Publisher finds the requested data in its CS and generates the Data packet.
(5) Router B caches the data and forwards it to Router A, according to the PIT.
(6) Router A caches the data and forwards it to the Consumer, according to the PIT.
(7) The Consumer receives the requested content.
(8) A new Consumer generates an Interest for the same /domain/content.
(9) Router A finds the content in the cache and immediately generates the Data packet.
requester by following the reverse path (i.e., by exploiting traces left by Interest packets inside the PIT tables of traversed nodes).

2.2. Theoretical description of Bloom filters

Bloom filters are probabilistic data structures conceived to efficiently perform membership queries on large data sets [21].

To ease the comprehension of the notions presented in the following, a summary of all the adopted symbols is reported in Tab. 1.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>Data set represented by the Bloom filter.</td>
</tr>
<tr>
<td>$M$</td>
<td>Cardinality of $S$.</td>
</tr>
<tr>
<td>$x$</td>
<td>Generic element of $S$.</td>
</tr>
<tr>
<td>$y$</td>
<td>Element for which a membership query is executed.</td>
</tr>
<tr>
<td>$m$</td>
<td>Filter size, expressed in number of bits.</td>
</tr>
<tr>
<td>$k$</td>
<td>Number of hash functions used to obtain a footprint.</td>
</tr>
<tr>
<td>$h_i(x)$</td>
<td>$i$-th hash function executed on element $x$.</td>
</tr>
<tr>
<td>$p_{fp}$</td>
<td>False positive probability.</td>
</tr>
<tr>
<td>$k_{opt}$</td>
<td>Optimal number of hash functions.</td>
</tr>
<tr>
<td>$d$</td>
<td>Number of bits per cells.</td>
</tr>
<tr>
<td>$p$</td>
<td>Number of cells decremented by 1 in a Stable Bloom filter.</td>
</tr>
</tbody>
</table>

In general, a Bloom filter is a $m$-bits long vector that can be used to map the IDs of the elements belonging to a data set $S$, supposed with cardinality $M$, to their “footprints”. By using $k$ independent hash functions, two basic operations are defined: element mapping and membership check. They aim to insert elements inside the Bloom filter, and to test whether unknown elements are members of the represented data set, respectively. A simple example, describing how element mapping and membership check operations are implemented, is reported in Fig. 2.
In detail, the *element mapping* operation is handled by means of two consecutive steps. First of all, the footprint of an element \( x \) is computed by executing \( k \) hash functions, that are \( h_1(x), h_2(x), \ldots, h_k(x) \). Then, the bits of the vector at positions \( h_1(x) \mod m, h_2(x) \mod m, \ldots, h_k(x) \mod m \), are set to 1. It is worth noting that footprints can overlap with each other in one, or more, of their \( k \) positions, thus generating collision regions.

To execute the *membership check* of an unknown element \( y \), instead, its footprint is firstly computed with the same group of hash functions used for the mapping phase. Then, the \( k \) identified positions are checked: if even a single bit is found to be 0, \( y \) is claimed to be not in the set, with a complete certainty (i.e., classical Bloom filters do not generate *false negatives*). Otherwise, in case all the \( k \) bits are found to be 1, the element \( y \) is considered as a member of the data set \( S \), although with a fixed *false positive probability*, \( p_{fp} \). Indeed, the presence of collision regions may generate a positive match for a membership check of an unknown element.

---

\( h_i(x) \), with \( i = 1, \ldots, k \), is the \( i \)-th hash function of the element \( x \)
element that is, actually, not present inside the set (i.e., false positive).

As discussed in [16], $p_{fp}$ is influenced by the length of the filter. Hence, the higher is the number of bits per element, the lower is the number of collisions generated during the membership check. However, it has been shown that $p_{fp}$ can be minimized by optimally choosing the length of the filter, according to the following formula:

$$m = -\frac{M \ln p_{fp}}{(\ln 2)^2}. \quad (1)$$

Under this hypothesis, the optimal number of hash functions to be used is equal to:

$$k_{opt} = \frac{m}{M} \ln 2. \quad (2)$$

Several Bloom filter variants have been proposed so far [16]. Differently from the original implementation (i.e., the one discussed before), they also introduce the possibility to delete elements, thus enabling a dynamic management of the filter. To this end, the single element of the vector is identified by a cell made up of $d$ bits (with $d \geq 1$).

An example is represented by the Counting Bloom filter [22]. In this case, at every element insertion, the $k$ cells associated to the footprint are incremented by 1 (the maximum value of the counter is $2^d - 1$). On the contrary, when an element needs to be deleted, the $k$ cells of the footprint are decremented by 1. In case of overflow/underflow of a cell, the respective counter is left to its maximum/minimum value.

Another particular characterization of the standard Counting Bloom filter is the Stable Bloom filter [23], which introduces the concept of stability: after a certain number of insertions and deletions, the filter reaches a stable point where the ratio of zeros and ones is kept constant. In detail, to achieve the stability, at every insertion of an element, $p$ random cells are firstly decremented by 1; then, the $k$ cells of the footprint are set to their maximum value.

2.3. From theory to practice: Bloom filters in IP and NDN networks

Bloom filters have been adopted for several purposes in networking systems. In IP networks, for example, they have been employed as a software solution
for the lookup of IP addresses, owing to their space efficiency, constant access
time, and better scalability with respect to other approaches based on Ternary
Content-Addressable Memory (TCAM) [24]. Bloom filters have, also, been con-
sidered as a way to speed up lookup operations by filtering out unnecessary
accesses to slow off chip memories [30]. Furthermore, the adoption of Bloom fil-
ters has been proposed also in the context of communication protocols related
to web caching. For example, the solution presented in [22], namely Summary
Cache, introduces the use of web proxies that keep a summary of the cache
directory of each participating proxy by using Counting Bloom filters. This al-

dows to check the summaries for potential hits before sending any queries, thus
reducing the communication overhead.

More recently, some scientific contributions (i.e.,[35, 18, 19, 25, 17, 36, 20,
38]) demonstrated that Bloom filters can bring important advantages also in
ICN architectures. Indeed, they can be used for designing new communica-
tion protocols and/or data structures able to overcome some of the well-known
problems that could arise in case of a large scale development of the NDN archi-
tecture (such as network overhead, forwarding speed, and excessive growth of
routing tables). The main aspects covered by the aforementioned works, along
with the respective improvements achieved with the adoption of Bloom filters,
are summarized in Tab. 2.

In particular, Bloom filters have been considered in the design and imple-
mentation of real high-speed content-based routers aimed to support forwarding
tables with one billion content prefixes [17]. In [36], instead, Bloom filters are
used to efficiently represent data stored in both FIB and PIT tables. The
work presented in [38] proposes to build a lookup engine for NDN nodes which
jointly combines Bloom filters and small-scale tree data structures. The pro-
posed scheme guarantees a lower false positive rate and a lower processing time
with respect to other solutions that are fully based on Bloom filters. It, also, re-
quires less memory space when compared with common tree-based approaches.
In addition, Bloom filters have been proposed as a key component of routing
protocols that assume the collection of information about temporary cached

10
Table 2: *Bloom filters* in CCN/NDN literature: review and identified advantages.

<table>
<thead>
<tr>
<th>Work</th>
<th>Summary</th>
<th>Identified advantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>[18]</td>
<td><em>Bloom filters</em> used to represent contents cached in the neighborhood.</td>
<td>Reduction of network traffic and server load.</td>
</tr>
<tr>
<td>[19]</td>
<td><em>Bloom filters</em> used to address cached contents in the control plane.</td>
<td>Reduction of network traffic and hit distance.</td>
</tr>
<tr>
<td>[25]</td>
<td><em>Bloom filters</em> used to store the digest of the suffixes of the locally cached contents.</td>
<td>Relieving of the <em>suffix hole</em> problem.</td>
</tr>
<tr>
<td>[17]</td>
<td><em>Bloom filters</em> for name-based lookup operations in high-speed content routers.</td>
<td>Support of forwarding tables containing $10^9$ content prefixes with unlimited characters.</td>
</tr>
<tr>
<td>[38]</td>
<td>Combined use of <em>Bloom filters</em> and tree structures in a Name Lookup engine.</td>
<td>Reduction of false positive rate, processing time, and memory requirements.</td>
</tr>
</tbody>
</table>
copies in neighboring nodes. In solutions discussed in [35, 18, 19, 25], for example, each node periodically sends a Bloom filter-based representation of its cached contents to all of its neighbors, which, in turn, update their FIB tables with consistent information. In [20], instead, a different approach is defined: nodes individually update their Bloom filter-based FIBs according to retrieved contents, thus avoiding signaling messages.

3. Bloom filter extension for ndnSIM

The Bloom filter module presented herein is developed within ndnSIM [14], i.e., an open source NDN simulator based on the NS-3 network simulator [26]. Following the philosophy of NS-3, ndnSIM implements an independent and complete NDN stack that can be instantiated inside a simulated node. This stack encompasses all the basic structures which characterize a NDN node, such as CS, PIT, and FIB, along with communication primitives (i.e., Interest and Data) and network entities (like NDN faces). In addition, ndnSIM provides a series of applications, helper classes, and traffic generators that can be used to facilitate the creation of a complete simulation scenario [14]. A basic C++ class, associated to each level of the NDN stack, can be further specialized. This means, for example, that different forwarding strategies (such as Flooding, Best Route), cache replacement policies (i.e., LRU, LFU, Random), and application layers (i.e., Consumer, Producer) can be chosen. Owing to these features, ndnSIM represents a good candidate for developing and testing Bloom filters, as well as to devise and simulate new communication protocols based on them.

3.1. Design criteria and code description

The implemented module is completely written in C++, and its code is freely available under the GPLv2 license. Fig. 3 shows the UML diagram of the classes that are added or modified from the original version of ndnSIM. It is important to remark that the diagram reports the most important data members and functions.
Figure 3: UML Diagram of the tool.
Strategic design criteria related to key aspects, such as the choice of the data structure used to implement Bloom filters, the possibility to further specialize their implementations, and the integration of the tool within the entire ndnSIM module, are carefully taken into account in order to guarantee an effective usage of the proposed module.

First, Bloom filters need to be implemented by means of a data structure that guarantees an easy and direct access to its bits, thus efficiently enabling insertions, deletions, and membership check operations. To reach this goal, two well-known C++ data structures can be used: the dynamic_bitset, which belongs to the Boost C++ libraries [27], and the std::bitset [28]. Despite both of them permit the access to every single using simple indices (i.e., without relying on bitwise operations with predefined bit-masks), the dynamic_bitset is selected as the most suitable solution for our implementation. In fact, it allows to define, at run-time, the size of the filter. Differently, the std::bitset requires the size of the filter to be specified at compile-time through an integer template parameter. Hence, thanks to the highest flexibility of the dynamic_bitset, the proposed implementation offers the possibility to easily resize the filter according to users’ needs.

Our module is developed with the idea to support the simulation of different kind of Bloom filters and further upgrade their features. First of all, a basic class, namely BloomFilterBase, is created to define all the parameters and functionalities that are common among the different Bloom filters. At the time of this writing, the tool offers a complete implementation of both Counting [22] and Stable [23] Bloom filters. As reported in Fig. 3, they are implemented in the BloomFilterCounting and BloomFilterStable classes, respectively, which inherit from the BloomFilterBase class. In this way, in line with the philosophy of ndnSIM and NS-3, which are based on modularity and flexibility, different data structures can be integrated by simply extending the BloomFilterBase class, and by customizing the implementation of specific operations, like footprint insertion, initialization, and so on.

ForwardingStrategy, PitImpl, and StackHelper classes, although already avail-
able in the original version of ndnSIM, are properly extended (see Fig. 3) to integrate Bloom filters. In summary, the ForwardingStrategy class handles the transmission and reception of Interest and Data packets; the PitImpl class is responsible for the lookup and the creation of PIT entries; the StackHelper class is used to set up and aggregate the several components of a NDN node (i.e., CS, PIT, FIB, Forwarding Strategy) within a single stack. These classes are modified in order to support the definition and the adoption of Bloom filters for mapping locally cached contents, and/or for representing FIB tables, thus making the tool suitable for the implementation of enhanced forwarding strategies for NDN.

Finally, users can select the type and the role of Bloom filters directly from the configuration file of the simulated scenario, by exploiting the SetBfFib() and SetBfCache() methods defined within the StackHelper class.

3.2. Offered functionalities

The main functionalities the implemented Bloom filter extension are thoroughly described in the following (a summary is provided in Fig. 4).

- **Initialization**: all the aspects related to the creation of a Bloom filter object inside a NDN node are covered. As anticipated before, two methods of the StackHelper class, that are SetBfFib() and SetBfCache(), can be used to configure the scope of the filter, as well as to set all the attributes needed to define its size. In particular, the size of the filter can be defined through “optimal” or “custom” techniques. The “optimal” scheme calculates the size of the filter by using Eq. 1. The “custom” approach, instead, allows users to directly specify the desired length of the filter and the number of hash functions to be used. Once the type and the role of the filter is chosen, the MakeBfInitialization() function is called to start the initialization of the corresponding data structure. According to the selected initialization technique, either the InitBloomFilterOptimal() or the InitBloomFilterCustom() method will be called to effectively create
Figure 4: Structure and Functions of the tool.
the desired filters. When the optimal dimensioning is selected, users must define the content catalog cardinality, \( M \), the target probability of false positives, \( p_{fp} \), and the number of bits per cell, \( d \). Then, the \textit{ExtractOptimum()} method is invoked to compute the size of the filter, \( m \), and the optimal number of hash functions, \( k \). The current implementation makes use of the non-cryptographic MurmurHash3 hash function \cite{29}, which is very suitable for general lookup operations, thanks to its speed and low collision rate. The \( k \) different hash functions are obtained by seeding the same hash function with different seeds, which are randomly generated through the \textit{GenerateSeedsHash()} method.

- **Insertion**: the element mapping phase is implemented by the \textit{ComputeIndicesFilter()} and \textit{InsertFootprint()} methods. The first one is called to generate, by using the aforementioned \textit{MurmurHash3()} function, the indices of the \( k \) cells of the filter, whose bits will be effectively set by the \textit{InsertFootprint()} method. Obviously, the implementation of the \textit{InsertFootprint()} function depends on the type of the selected filter. Indeed, when a \textit{Counting Bloom filter} is chosen, the identified \( k \) cells are simply incremented by 1 (this operation is executed by the \textit{IncrementFilterCell()} method). In case of a \textit{Stable Bloom filter}, instead, \( p \) random cells are firstly decremented by 1, using the \textit{DecrementFilterCell()} method, and then, the \( k \) cells are set to their maximum value through the \textit{SetMaxFilterCell()} method. For example, the \textit{InsertFootprint()} method can be called when a new content is cached by a NDN node, in order to update the \textit{Bloom filter} that stores the footprints of the cached contents.

- **Deletion**: both \textit{Counting} and \textit{Stable Bloom filters} support the deletion of elements. Hence, knowing the footprint of the target element, which is calculated by the \textit{DecrementFootprintCells()} method, the \textit{DecrementFilterCell()} function is executed to decrease the identified \( k \) cells by 1 unit. In addition, it is also possible to reset the counter of a cell by setting all its bits to 0. This operation is executed by the \textit{ResetFootprint()} method.
• **Membership Check:** by using the `LookupFilter()` function, the presence of an element into the data set represented by the filter is verified. As for the *element mapping* operation, the footprint of the target element is firstly computed. Then, the `CountCell()` method is used to evaluate the counters of the $k$ cells of the footprint. At the end, the element will be declared as part of the data set only if all the $k$ counters are found to be greater (or equal) than 1.

• **Interface Ranking:** this latest functionality aims at supporting the implementation of *Bloom filter*-based forwarding strategies. It is assumed that a NDN node may collect, within dedicated *Bloom filters*, information about contents that are available through each network interface. Hence, when an *Interest* is received, and the requested content is not in the cache, its name is looked up inside all the available *Bloom filters*. Then, network interfaces whose *Bloom filter* provides a positive match are ordered in a ranked list according to specific metrics (e.g., name length, routing cost). This operation is executed by the `LookupOrderedBloomFilters()` and `MakeCustomLookupBloomFilters()` functions, which compute the routing cost of each interface. Note that, in the current implementation, the routing cost is evaluated by considering the number of name components for which a *Bloom filter* provides a match. For example, the higher is the number of matched name components, the lower is the routing cost associated to the corresponding interface. Interfaces with lower routing costs will be, then, preferred when forwarding *Interest* packets.

4. **Module evaluation**

4.1. *Analyzing the consistency of the tool through use cases*

The consistency of the proposed tool is demonstrated through two representative use cases. The first one is inspired by the work presented in [30], which states that unnecessary accesses to off-chip memories allocated inside routers
can be avoided by filtering them out using on-chip Bloom filters. Querying the filter before effectively looking up the cache can, in fact, improve the lookup speed because of the constant access time of Bloom filters.

The second use case, instead, is inspired by the work in [20], which proposes the use of Bloom filters to collect routing information from retrieved contents. This information, indeed, can be exploited to forward successive requests for the same contents, or for contents that share a prefix with the names that are already mapped inside the Bloom filters.

It is worth noting that the scope of the proposed tool is not bound to the use cases presented herein. However, investigated examples can be considered as valuable starting points to develop novel Bloom filter-based enhancements for the NDN architecture.

**Use case 1: Content Store Membership Check.**

The scope of this scenario is twofold: it certifies the accuracy of the proposed tool, and it shows how Bloom filters can be used to represent a set of locally cached contents.

To this end, a simple network scenario is set up, where two NDN nodes, one client and one producer, are connected by a point-to-point link. During the simulation, the client issues a single request for all the $M = 104450$ unique contents of the entire catalog [20]. These requests are received by the producer, which stores only a fraction of the catalog inside its cache. In particular, during the initialization phase, $L < M$ contents are randomly chosen and made available at the producer side. Their names are inserted into a dedicated Counting Bloom filter, which is characterized by a number of bits per cell equal to $d = 1$.

Based on the NDN literature [20, 31], the cache to catalog ratio, i.e., the ratio between the cache size and the cardinality of the catalog, is chosen in the range $[0.01\% − 1\%]$. The size of the filter is obtained according to the formula reported in Eq. 1, by setting the probability of false positive, $p_{fp}$, to $5\%$. As discussed in Sec. 2.2, however, the optimal dimensioning technique supposes the exact knowledge of the content catalog cardinality, $M$. Unfortunately, since
the content cardinality is not available a priori in a real scenario, $M$ can be only estimated. As a consequence, the measured probability of false positive may differ from the expected value. To demonstrate this effect through simulations, the filter is intentionally under-dimensioned considering, as the real content catalog, only the small number of contents that can be stored inside the cache (i.e., $L$ instead of $M$).

Every time the producer receives a new query from the client, it executes the membership check operation to verify the presence of the requested content inside its cache, and the relative result is classified. For the sake of completeness, besides the empirical false positives ratios (FP), both true positives ratios (TP), and true negatives ratios (TN) are evaluated (note that false negatives cannot happen in a Bloom filter with no deletion operations). Obtained results have been averaged over 10 different runs; mean values, along with their respective 95% confidence intervals, are reported in Tab. 3.

<table>
<thead>
<tr>
<th>Cache To Catalog Ratio [%]</th>
<th>Empirical FP [%]</th>
<th>Empirical TP [%]</th>
<th>Empirical TN [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.01 ± 0.14</td>
<td>1.00 ± 0.0</td>
<td>92.98 ± 0.14</td>
</tr>
<tr>
<td>0.50</td>
<td>5.72 ± 0.14</td>
<td>0.50 ± 0.0</td>
<td>93.78 ± 0.14</td>
</tr>
<tr>
<td>0.10</td>
<td>5.56 ± 0.24</td>
<td>0.10 ± 0.0</td>
<td>94.33 ± 0.24</td>
</tr>
<tr>
<td>0.01</td>
<td>7.44 ± 1.43</td>
<td>0.01 ± 0.0</td>
<td>92.55 ± 1.43</td>
</tr>
</tbody>
</table>

First of all, results highlight how the non-optimal sizing of the filter leads to an expected discrepancy between the empirical false positive ratio and the desired value: the former quantity is always higher than 5%. In conducted tests, in fact, the filter is dimensioned considering the size of the cache, which, in terms of number of contents, is much smaller than the real cardinality of the entire catalog. As a consequence, the filter experiences an increment of the collision regions, which brings to a consequently growth of the empirical
\( p_{fp} \) values. Furthermore, as a proof of the accuracy of the proposed tool, it is possible to observe that empirical true positive ratios reflect the values of the different cache to catalog ratios (i.e., there are no false negatives), and that empirical true negative ratios are equal to the difference, in percentage, between 100 and the sum of empirical false positives and empirical true positives.

**Use Case 2: Content-driven Routing.**

This use case intends to demonstrate how the proposed tool can be, also, used to develop a Bloom filter-based forwarding strategy for NDN. In particular, it is inspired by the work in [20], which suggests to populate FIBs according to retrieved contents: nodes are equipped with as many Bloom filters as their network interfaces, thus having the possibility to map the names of retrieved contents. Since this process is content-driven, Bloom filters are initially empty (e.g., warm-up phase); in this case, all the Interest packets are sent in flooding due to the absence of matching interfaces. Then, every time a new Data packet is received, the Bloom filter associated to the respective interface is updated accordingly. After having collected enough information inside the filters, flooding operations will be limited, and Interest packets will be forwarded only toward the interface(s) whose filter(s) has provided a match (either complete or partial).

In order to evaluate the impact of both the type of the filter and the technique used to define its size, four different Bloom filter-based forwarding strategies are simulated, which are based on Counting Bloom filters, both with optimal and custom sizes, and Stable Bloom filters, both with optimal and custom sizes. Moreover, to remark pros and cons of the aforementioned approach, two additional routing schemes are considered for the performance comparison: Flooding and Shortest Path. When the Flooding scheme is used, each node forwards received Interest packets toward all the available faces. Regarding the Shortest Path approach, instead, the Dijkstra algorithm is used to compute the shortest paths toward the seed copies, and to populate the FIBs accordingly. In this case, Interest packets are sent only toward the shortest paths.
All the parameters needed to size the filters are reported in Tab. 4. Eq. 1 is used to dimension the filter, with a desired probability of false positive equal to 5%. For the optimal dimensioning case, the parameter $M$ is set to its real value (i.e., 104450). For the custom one, instead, it is supposed that the used value of $M$ is, actually, underestimated. According to the analysis presented in the previous sub-section, this assumption should highlight the impact that a wrong dimensioning has on the performance of Bloom filter-based strategies in real scenarios, i.e., where the catalog cardinality is not known a priori. In particular, the estimated catalog cardinality considered for the custom case is equal to $M/6$, and the number of hash functions, $k$, is set to 3.

The simulated network reproduces the GEANT topology, which is composed of 22 nodes organized as depicted in Fig. 5. In each run, two random nodes, selected as repositories, share a catalog of 104450 unique contents. All the remaining nodes, instead, act as clients and request contents according to a Zipf’s like probability distribution, that is $P(X = i) = i^{-\alpha}/\sum_{j=1}^{M} j^{-\alpha}$. Two distinct values of the Zipf’s exponent are considered, i.e., $\alpha \in \{1.0, 1.2\}$. Each node is also equipped with a cache memory properly configured for ensuring a cache to catalog ratio equal to 0.1%.

The performance evaluation of all the considered forwarding strategies is done by monitoring three metrics: content retrieval cost, which expresses the amount of traffic needed to retrieve a single content, hit ratio, and retransmitted Interest ratio.

<table>
<thead>
<tr>
<th>Dimensioning Technique</th>
<th>$M$</th>
<th>$p_{fp}$ [%]</th>
<th>$d$</th>
<th>$k$</th>
<th>Cells of the Filter [kBytes]</th>
<th>$m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>104450</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>651269</td>
<td>238</td>
</tr>
<tr>
<td>Custom</td>
<td>104450</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>108545</td>
<td>40</td>
</tr>
</tbody>
</table>

The content retrieval cost evaluated in scenarios with $\alpha = 1$ and $\alpha = 1.2$
Figure 5: GEANT Network.
is reported in Figs. 6 and 7, respectively. It is possible to note that Bloom filter-based forwarding strategies dramatically reduce the communication overhead with respect to the Flooding approach. This is due to the fact that each node gradually learns (and stores) new routes toward remote contents inside its Bloom filters, thus limiting the number of flooded Interest packets. However, the lowest communication cost is always associated with the Shortest Path strategy. Although this advantage, it is important to remark that with the Shortest Path strategy a routing algorithm should be executed to preemptively calculate routes toward all the available contents, thus becoming less scalable than Bloom filter-based approaches.

By comparing the different variants of the Bloom filter based forwarding strategies, instead, two main results are worth to be highlighted: when the optimal dimensioning technique is used, both Counting and Stable Bloom filters provide similar results, as reported in Figs. 6(a) and 7(a). In case of the custom dimensioning, instead, the use of Stable Bloom filters entails a slightly increment of the content retrieval cost, as it can be noticed from Figs. 6(b) and 7(b). This means that the decrement of $p$ random cells by 1 unit expected by Stable Bloom filters at each insertion, may generate the erasure of important routing information when the filters are under-dimensioned, thus forcing nodes to send more Interest packets in flooding.

The gradual learning process characterizing all the Bloom filter-based strategies is further confirmed by the measured hit ratio. Results reported in Figs. 8 and 9 indicate an increment of the hit ratio in the first part of the simulation, i.e., where information are progressively stored inside the filters. Then, the hit ratio converges to a plateau. This demonstrates that the number of Interest packets that are sent in flooding, as well as the number of unsatisfied requests, decrease as soon as nodes collect information in their Bloom filters. Also in this case, the difference in performance between Counting and Stable Bloom filters is evident when the custom dimensioning technique is adopted. Indeed, the slightly lower hit ratio associated to the Stable Bloom filter variant, reported in Figs. 8(b) and 9(b), is caused by the higher number of Interest packets that
Figure 6: Chunk Retrieval Cost evaluated when $\alpha = 1$ and Bloom filters are dimensioned using (a) the optimal and (b) the custom technique, respectively.
Figure 7: Chunk Retrieval Cost evaluated when $\alpha = 1.2$ and Bloom filters are dimensioned using (a) the optimal and (b) the custom technique, respectively.
Simulation time $[s]$ & Memory Usage $[MBytes]$
\hline
\alpha = 1 & \alpha = 1.2 & \alpha = 1 & \alpha = 1.2 \\
\hline
Flooding & $1026 \pm 45$ & $541 \pm 23$ & $481 \pm 18$ & $457 \pm 8$ \\
Shortest Path & $240 \pm 7$ & $190 \pm 3$ & $7335 \pm 4$ & $7331 \pm 2$ \\
Counting Bloom Filter, Optimal & $608 \pm 4$ & $333 \pm 3$ & $553 \pm 2$ & $529 \pm 8$ \\
Stable Bloom Filter, Optimal & $614 \pm 5$ & $334 \pm 2$ & $553 \pm 2$ & $529 \pm 8$ \\
Counting Bloom Filter, Custom & $606 \pm 5$ & $331 \pm 2$ & $494 \pm 2$ & $470 \pm 8$ \\
Stable Bloom Filter, Custom & $716 \pm 10$ & $360 \pm 3$ & $494 \pm 2$ & $470 \pm 8$ \\
\hline

Table 5: Scalability Analysis

are transmitted in flooding. As expected, the Shortest Path forwarding strategy always guarantees the highest hit ratio, due to the pre-computed routes that allow to drastically limit the forwarded Interest packets.

In the end, all the considered forwarding strategies report similar results in terms of retransmitted Interest ratio (see Figs. 10 and 11).

In summary, the conducted tests highlight three main aspects related to Bloom filter-based strategies, that are (i) the learning process of the nodes, which is made possible by leaving traces of retrieved contents inside Bloom filters, (ii) the similar behavior of Counting and Stable Bloom Filters when optimally dimensioned, and (iii) the effects of the false negatives introduced by Stable Bloom filters when the custom dimensioning is adopted.

4.2. Scalability analysis

As a final step, the impact that the proposed Bloom filter extension has on the overall performance of the ndnSIM simulator is provided. In particular, a comprehensive scalability analysis is performed, by measuring both the simulation time and the memory usage. In order to guarantee the consistency of the presented results, tests are conducted considering the same simulation scenario presented for the second use case, and by using a machine with Intel Xeon at 3.6 GHz CPU, 32 GBytes of RAM, and Linux Ubuntu 12.04 operating system.
Figure 8: Hit Ratio evaluated when $\alpha = 1$ and Bloom filters are dimensioned using (a) the optimal and (b) the custom technique, respectively.
Figure 9: Hit Ratio evaluated when $\alpha = 1.2$ and Bloom filters are dimensioned using (a) the optimal and (b) the custom technique, respectively.
Figure 10: Retransmitted Interest Ratio evaluated when $\alpha = 1$ and Bloom filters are dimensioned using (a) the optimal and (b) the custom technique, respectively.
Figure 11: Retransmitted Interest Ratio evaluated when $\alpha = 1.2$ and Bloom filters are dimensioned using (a) the optimal and (b) the custom technique, respectively.
Obtained results, summarized in Tab. 5, indicate that the presence of Bloom filters inside a simulated node does not have a noticeable impact on the memory usage. In fact, when an optimal dimensioning technique is used, only 72 MBytes of additional memory are allocated with respect to the Flooding strategy. On the contrary, the Shortest Path approach requires a very considerable amount of memory, i.e., 7.4 GBytes, due to the storage of the pre-filled FIBs inside each node.

The simulation time is strictly influenced by the number of events that are handled during the simulation, which are, in this case, correlated to the number of Interest packets that are forwarded. For this reason, the Shortest Path and Flooding strategies are associated with the lowest and the highest simulation time, respectively. Bloom filter-based approaches, instead, report intermediate values, that are around 600 seconds, with $\alpha = 1$.

Considering these results, it is possible to state that the proposed tool does not limit the scalability of ndnSIM because it does not require significant computational and memory capabilities.

4.3. Expected impact on the research community and lessons learned

From the literature review (discussed in Sec. 2.3), we learned that certain issues characterizing NDN networks, such as forwarding speed and scalability of routing protocols, can be alleviated by using Bloom filters.

Thanks to the presented work, the research community can converge toward better standardized simulation procedures, thus broadening the spectrum of cross-comparisons among new proposals. As a consequence, the proposed tool may have a not negligible impact on the several research activities focusing on NDN. In fact:

- it represents the first open source tool that models Bloom filters within a widely used network simulator (i.e., NS-3);
- it offers the possibility to test and evaluate the effectiveness of novel protocols, algorithms, and methodologies based on Bloom filters, as well as
the possibility to compare them with other solutions already proposed in the literature.

After having identified strategic design criteria for key development aspects, and tested consistency and flexibility of the proposed implementation, the following important lessons learned can be reported:

- The implemented module is flexible, in the sense that it allows users to select between several types of Bloom filters; furthermore, it provides the possibility to vary the target usage of Bloom filters themselves (e.g., membership check of locally cached contents and/or design of name-based forwarding strategies).

- The consistency of the proposed tool, evaluated by means of significant use cases, strengthens the role that the presented implementation may assume in current research activities involving the use of Bloom filters in NDN.

- The scalability of the ndnSIM simulator is not compromised by the integration of the Bloom filter extension. This makes the tool very suitable to evaluate NDN-based proposals also in medium and large-scale scenarios.

- The implemented module is expandable, meaning that its modularity permits to easily add new features and/or new Bloom filter types.

- The developed extension is fully integrated within ndnSIM, as well as within the whole NS-3 simulation framework. Therefore, it can be exploited in several contexts that differ from the presented use cases, or that may go beyond the simulation of NDN networks.

- The availability of a comprehensive open source simulator may reduce the number of published works whose results, obtained by using proprietary or custom tools, are not reproducible [37].
5. Conclusions

In this work, an open source extension for the ndnSIM simulator, modeling Bloom filters, has been presented. At the current stage, it allows to choose between two types of Bloom filters, that are Counting and Stable. In addition, two types of dimensioning techniques, i.e., Optimal and Custom, as well as different scopes, can be selected for the simulated Bloom filters. The effectiveness of the proposed implementation has been evaluated through the study of two significant use cases. The first one relates to the adoption of Bloom filters in membership check operations for locally cached contents. The second one, instead, demonstrates the benefits that Bloom filters may bring to the design of advanced named-based routing protocols for NDN. In the end, a comprehensive scalability analysis has been carried out for highlighting the impact that the implemented tool has on both the simulation time and the memory usage of the ndnSIM simulator. The conducted analysis confirmed the consistency, the accuracy, and the flexibility of the presented extension, thus strengthening the impact that it may have on the research community.

In the near future, we plan to improve the proposed Bloom filter extension by adding new features, such as the possibility to resize the filters according to network states (i.e., number of nodes, effective number of contents in the network, and so on), the introduction of Bloom filter-based signaling messages that can be exchanged among nodes to update routing information, and the implementation of new operations regarding Bloom filters themselves, such as merging and splitting.

6. Acknowledgments

This work was supported by the PON projects (RES NOVAE, DSS-01-02499 and EURO6-01-02238) funded by the Italian MIUR and by the European Union (European Social Fund).
References


   URL http://www.ccn-lite.net


[14] A. Afanasyev, I. Moiseenko, L. Zhang, ndnSIM: NDN simulator for NS-3,


   bloom filters for distributed systems, IEEE Communications Surveys Tutor-


[18] M. Lee et al., SCAN: Scalable Content Routing for Content-Aware Net-

[19] Y. Wang et al., Advertising cached contents in the control plane: Necessity
   and feasibility, in: Proc. of IEEE INFOCOM WKSHPS, Orlando, (FL),
   USA, 2012.

   domain Routing in NDN, in: Proc. of IEEE CCNC, Special Session on
   Research and Case Study for Designing and Deploying Information-centric
   Networks, Las Vegas, NV, USA, 2014.

[21] B. H. Bloom, Space/time trade-offs in hash coding with allowable errors,


URL http://www.nsnam.org/.


38