applied sciences

MDPI

Article

On Sharing an FIB Table in Named Data Networking

Ju Hyoung Mun and Hyesook Lim *

Department of Electronic and Electrical Engineering, Ewha Womans University, Seoul 03760, Korea
* Correspondence: hlim@ewha.ac.kr; Tel.: +82-2-3277-3403

Received: 7 June 2019; Accepted: 2 August 2019; Published: 5 August 2019

Abstract: As a new networking paradigm, Named Data Networking (NDN) technology focuses on contents, and content names are used as identifiers for forwarding and routing, as opposed to IP addresses in the current Internet. NDN routers forward packets by looking up a Forwarding Information Base (FIB), each entry of which has a name prefix and output faces. An FIB should have the information to forward Interest packets for any contents. Hence, the size of an FIB would be excessively large in NDN routers, and the traffic for building an FIB would be significant. In order to reduce the traffic associated with building an FIB table and memory requirement for storing an FIB table, this paper proposes a new efficient method which combines the routing of network connectivity and the building of a forwarding engine using Bloom filters. We propose to share the summary of an FIB using a Bloom filter rather than to advertise each name prefix. The forwarding engine of the proposed scheme is a combination of Bloom filters, and hence the memory requirement of the forwarding can be much smaller than the regular FIB. Simulation results using ndnSIM under real network topologies show that the proposed method can achieve nearly the same performance as the conventional link state algorithm with 6–8% of the traffic for distributing the connectivity information and 5–9% of the memory consumption.

Keywords: Bloom filter; information-centric networking; Named Data Networking; routing and forwarding

1. Introduction

Distributing multimedia contents has become the main usage of the Internet and accounts for the majority of the Internet traffic today [1]. The current infrastructure based on a host-to-host communication model, however, is inefficient for distributing contents, since contents should be repeatedly transferred from content sources to multiple users. To overcome this inefficiency, content-oriented networking technologies have been introduced and Named Data Networking (NDN) [2] is a promising candidate as a future Internet architecture. Unlike the current Internet, NDN decouples the contents from their locations, and hence contents can be provided by any nodes which have the requested contents [2–4].

To support the content-oriented service, the NDN defines two types of packets, *Interest* and *Data*, to request and deliver contents, respectively. Packets are forwarded in an NDN router using three types of tables: Content Store (CS), Pending Interest Table (PIT), and Forwarding Information Base (FIB). The CS caches the contents so that any router which has the requested contents can transmit the requested content immediately. The PIT records the forwarded path of an Interest for *Data* to be forwarded backwards. The FIB is similar to a routing table which stores the information of content sources. *Interest* packets are forwarded by looking up FIBs [2,3,5].

An FIB should have the output face information for any contents in the network while the variety of contents is enormous. Furthermore, the lengths of content names are not fixed and theoretically unbounded, unlike the Internet protocol (IP) address. Even though content names can be aggregated,

it would be larger than the total number of domain names [6]. The size of an FIB table is expected to be excessively larger than an IP forwarding table, and consequently the traffic caused in building an FIB table can also be significant. Efficient routing and forwarding algorithms are essential for the successful realization of the NDN [3].

This paper has two primary aims: (1) to reduce the traffic associated with building an FIB table; and (2) to reduce the memory requirement for storing an FIB table. To reduce the traffic for building an FIB table at each router, this paper proposes a new efficient method to share the connectivity information of an NDN network. In other words, rather than advertising each name prefix, we propose to share the summary of a set of name prefixes by using a Bloom filter. We also propose to forward *Interest* packets based on querying the Bloom filters which are stored instead of FIB tables at each router, and hence the memory consumption is reduced.

More specifically, the contributions of this paper are as follows:

- We propose a novel method which combines the routing of network connectivity and the building of a forwarding engine using Bloom filters, while previous works focus on either the routing or the forwarding using Bloom filters. Since the Bloom filters used in the routing phase are also used for forwarding, the construction of forwarding engines is simplified to the union of Bloom filters.
- We also provide a method to reduce the impact of false positives, which is inevitable when using Bloom filters.
- Large scale simulations were conducted using four real topologies which have up to 279 nodes. One thousand consumers and producers were randomly allocated on the network topologies, equivalent to a network with up to 2279 nodes.
- The performance evaluation shows that the proposed scheme significantly reduces the network traffic for routing and the memory consumption for storing name prefixes in a forwarding engine.

The remaining part of this paper proceeds as follows. Section 2 briefly introduces related works. Our proposed FIB sharing algorithm is described in Section 3. The performance evaluation result is shown in Section 4. Section 5 concludes the paper.

2. Related Works

2.1. Routing in Named Data Networking

Conventional routing algorithms used in the Internet can also be applied for the NDN because the forwarding model of the NDN is a superset of the current IP network [2,7,8]. However, the NDN has fewer restrictions in routing. For example, while multi-source or multi-destination packets are prohibited in the Internet protocol in order to avoid looping, they are allowed in the NDN. Furthermore, while routers in the current Internet cannot identify changes that have occurred in a network without running routing protocols, NDN routers can detect network changes without running routing protocols due to the stateful forwarding, in which a router waits for the *Data* packet after transmitting an *Interest* packet [5]. Unlike the IP network, forwarding engines in NDN allow for multiple output faces, and thus NDN routers can rank the list of faces and alter the forwarding path when a link failure occurs. Since forwarding planes can adapt short-term changes, control planes in NDN can be more focused on long-term changes occurring in a network [7]. Therefore, new unconventional but effective routing algorithms can be applied [6,8–12].

As a routing protocol for NDN, Named-data Link State Routing (NLSR) protocol [8] updates routing information using *Interest* and *Data* packets and produces a list of ranked faces for each name prefix in order to support adaptive forwarding strategies. In the Coordinated Routing and Caching (CoRC) scheme [6], many off-path caching mechanisms are correlated with a hash-based routing algorithm. Because responsible routers (RRs) only have FIB tables, memory consumption can be reduced. However, every RR should perform the tunneling for a given designated set of producers, and every packet should be forwarded to the corresponding RR. Hence, the CoRC cannot provide the shortest-path delivery.

Another hash-based routing scheme, αRoute [11], employs a name-based distributed hash table (DHT) and a distributed overlay-to-underlay mapping. By partitioning name prefixes based on similarity, similar names are located in the same partition. The overlay-to-underlay mapping is utilized to support near-shortest path routing. This scheme claims to provide the near-optimal routing in theory, but this claim has not yet been fully evaluated by simulations. Moreover, because this scheme treats a name prefix as a set of characters, finding the longest matching name prefix at component levels is hard to achieve.

A Bloom Filter-based Routing (BFR) [13] has been introduced. Each content server broadcasts Content Advertisement Interest (CAI) message which contains the information of the content server with a Bloom filter. The received CAI messages are stored in PITs, and hence all nodes have CAI from all content servers in their PIT after the routing phase. Initially, the Interest is forwarded by looking up the Bloom filters stored in PIT, and then the FIB is populated when the Data is returned as the response. BFR reduces the traffic to build FIB tables by employing Bloom filters. However, the Bloom filters are only used for populating the FIB tables, and finding the longest matching prefix is not considered when looking up the CAI messages as well as no protection is provided for the delivery towards the wrong content server due to false positives. Furthermore, a router should search both FIB and CAI messages when the entry for the requested content has not been populated.

2.2. Forwarding in Named Data Networking

As in the IP network, content names in NDN are hierarchical and aggregatable for scalability. In forwarding an *Interest* packet, therefore, the face with the longest matching name prefix should be identified. Since finding the face of the longest matching name prefix in wire-speed is a challenging research issue, various lookup algorithms and data structures have been introduced. These algorithms can be categorized into three groups: (a) trie-based [14,15]; (b) hashing-based [6,16]; and (c) Bloom filter-based [14,17–19].

NameFilter [19] lookup structure consists of two stages: the first stage for ascertaining the longest matching length and the second stage for finding the corresponding face. Both stages are based on multiple Bloom filters. Each of the Bloom filters in the first stage is dedicated to a specific length storing name prefixes in that specific length. The second stage consists of Bloom filters for each face. When looking up, both stages are queried sequentially: the longest matching length is found in the first stage, and then the output face is determined by querying the Bloom filter of the matching length in the second stage. NameFilter is a memory-efficient lookup architecture, but it requires a number of memory instances to support the parallel query and may return false faces because of false positives of the Bloom filters.

A Name Lookup engine with Adaptive Prefix Bloom filter (NLAPB) [18] is a hybrid lookup structure of Bloom filters and name prefix tries. In this hybrid data structure, a name prefix is divided into two parts: B-prefix and T-suffix. Bloom filters are built for the valid lengths of B-prefixes while tries are created for T-suffixes. The length of B-prefixes is adaptively chosen based on popularity. NLAPB reduces the memory consumption compared to the name prefix trie and the false return rate as opposed to Bloom filter-based structure [20].

Name prefix trie with a Bloom filter (NPT-BF) and NPT-BF-Chaining [14] have the same architecture which contains two parts: an on-chip Bloom filter and an off-chip name prefix trie. The Bloom filter is employed to filter out unnecessary accesses to the name prefix trie. Usually, the access time to off-chip memory takes much longer than to on-chip memory, and thus reducing the number of off-chip memory is crucial for enhancing the lookup performance. The lookup procedure of NPT-BF is as follows. The Bloom filter programmed by every node in the trie is queried first and then the trie is accessed only if the query result is positive. To find the longest matching prefix, this process is continued while increasing the number of name components. To further reduce the off-chip memory accesses, NPT-BF-Chaining continues to query the Bloom filter while increasing the number of name components to find the best matching length (BML). After finding the BML, the corresponding node

on BML is accessed. If the input matches with the node, then the output face is returned immediately. Otherwise, back-tracking to upper levels should be performed until the matching node is found. These algorithms guarantee to return the right face and when the on-chip Bloom filter is sufficiently large, the lookup can be done with only one off-chip memory access on average.

2.3. Bloom Filter

Due to their simplicity, Bloom filters [21] have been widely used in many applications [22–24] and a set of variant structures has also been introduced for different purposes [25–27]. The programming involves storing the given set into an m-bit vector by setting the bits indicated by hash indexes acquired from k hash functions for each element of the set, while the querying involves checking the membership information of an input by referring to the bits indicated by the same hash functions used in the programming. The query result can be identified as either *positive* or *negative*. If any of the referenced bits is 0, then the query result is *negative*, meaning that the input does not belong to the given set. Otherwise, if all of the referenced bits are 1, the query result is *positive*, meaning that the input is an element of the given set. The negative results are always correct, but the positive results may be false. Let n be the number of elements programmed to an m-bit Bloom filter. The false positive rate can be obtained from Equation (1).

$$f = (1-p)^k = \left(1 - (1 - \frac{1}{m})^{kn}\right)^k \quad (1)$$

The valuable characteristic of a Bloom filter is that the Bloom filter for the union of plural sets is easily obtained. When the Bloom filter parameters are identical [22], the Bloom filter of the union set is obtained by performing the bitwise-OR operation of Bloom filters for each set, as shown in Figure 1: $BF(S_A \cup S_B) = BF(S_A) \mid BF(S_B)$. This characteristic is effectively used in our proposed method described in later sections.

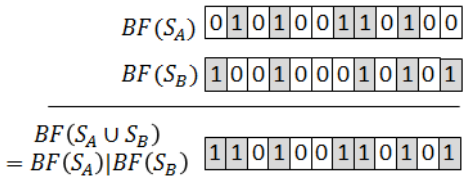


Figure 1. An example of two Bloom filter union, where the size of the Bloom filters, the number of hash indices, and the used hash functions are identical.

3. Proposed Work

Since the volume of name prefixes would be much larger than that of IP prefixes, building an FIB table would consume the significant amount of network bandwidth. The goal of this paper is to reduce the traffic associated with building an FIB table and the memory consumption for storing an FIB table. To reduce the traffic for building the FIB table, this paper proposes sharing the connectivity information using a Bloom filter. Additionally, to reduce the memory consumption for the FIB table, this paper proposes storing the Bloom filter at each face instead of the FIB table and forwarding *Interest* packets according to the query result of the Bloom filter.

3.1. Sharing Connectivity Information

Figure 2 illustrates the proposed scheme, where r_i (for $i = 1, ..., 7$) represents a router in the network. It is assumed that nodes r_3, r_6, and r_7, which are directly connected to other networks, receive prefix announcements from content publishers. As shown in the figure, they build an FIB table and create a summary of the FIB table in the form of a Bloom filter, named a summary Bloom filter (S-BF). The S-BF is constructed by programming every name prefix in the FIB table one by one. After building the S-BF, the router propagates its connectivity information to its neighboring nodes by flooding

the S-BF. Hence, a set of name prefixes programmed in the S-BF is shared in the network instead of actual name prefixes and the link state information. Because transmitting the S-BF consumes much less bandwidth than transmitting actual name prefixes, the traffic for propagating the connectivity information is greatly reduced.

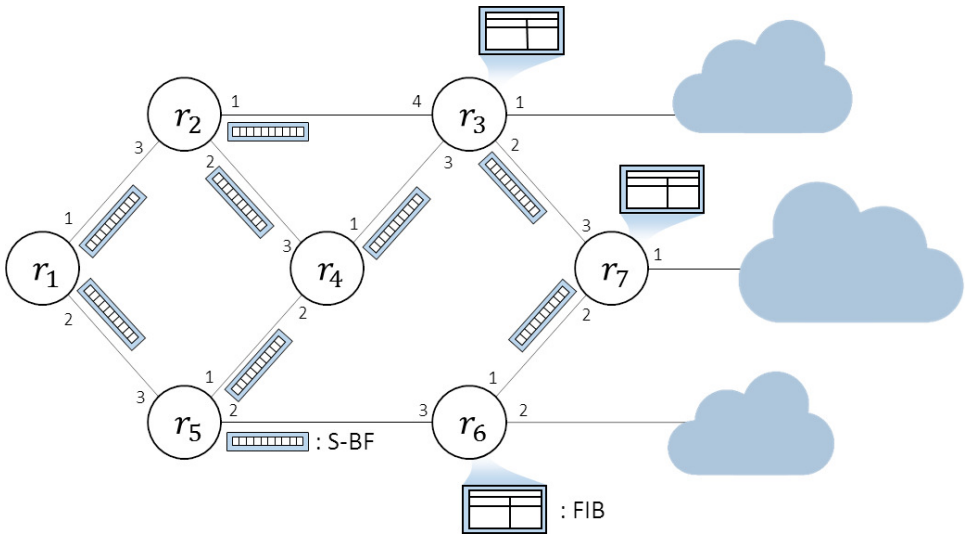


Figure 2. An example of the proposed scheme. The nodes connected to other networks, r_3, r_6, and r_7, only have the FIB table and the connectivity information is carried in S-BF.

Each node builds a forwarding Bloom filter (F-BF) by combining the received S-BFs. Figure 3 shows an example of building F-BFs using the received S-BFs in r_4 in Figure 2. A node has n F-BFs, where n is the number of faces, and the F-BF which dedicated for face i is denoted as F-BF[i]. The F-BFs are constructed by bitwise-ORing the received S-BFs, as described in Figure 1. If we assume that all links have the same delay and r_3, r_6, and r_7 send the S-BF at the same point, the S-BF from r_3 arrives first through face 1. The F-BF[1] for face 1 is formed by bitwise-ORing the S-BF from r_3 and existing F-BF[1]. Later, the S-BF from r_7 also arrives through face 1 and combines into F-BF[1]. As a result, S-BF from r_3 and r_7 are bitwise-ORed to form the F-BF[1] for face 1. The S-BF from r_6 arrives first through face 2, and forms the F-BF[2] in face 2. Even though the same S-BFs may arrive later through other faces, the proposed work builds F-BFs only using the S-BFs arrived first to forward packets through the shortest path.

The detailed procedure when a router receives an S-BF is shown in Algorithm 1. Initially, a node initializes all F-BFs corresponding to each face. If a node receives a new S-BF, the face combines the received S-BF with the F-BF, as shown in Figure 3; the S-BF is then forwarded to faces other than the incoming face. If a node receives a duplicate S-BF through different faces, it means that multiple paths to the publisher are available. The proposed work builds an F-BF using the firstly received S-BFs; however, the time difference between multiple S-BFs can be used to rank the faces. Since the proposed scheme works in a fully decentralized way, each node terminates the sharing the connectivity information right after flooding the S-BF, and this is repeated whenever a new S-BF arrives. When the process for propagating the connectivity information has been completed, FIB tables are only stored at edge nodes, which are connected to the outside of a network. Internal nodes only have an F-BF at each face. It is not possible for a node to know whether the connectivity information spreads to all other nodes in the network; however, the packet forwarding is possible even in the middle of the procedure sharing the connectivity information.

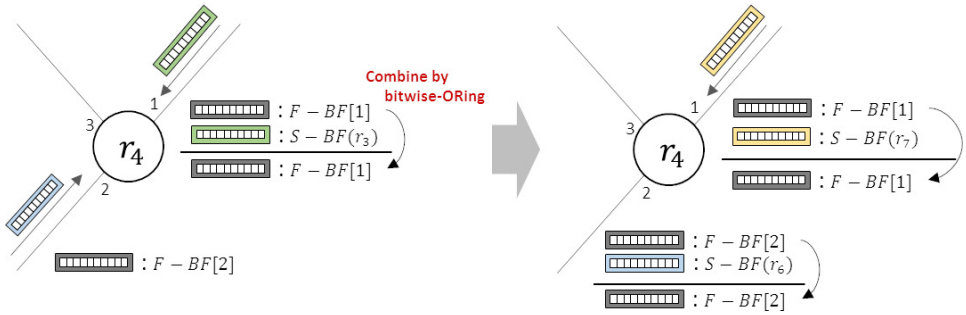


Figure 3. An example of building an F-BF. A node has n F-BFs, where n is the number of faces, and the F-BF which dedicated for face i is denoted as F-BF[i]. Whenever a new S-BF arrives, the F-BF of the incoming face is updated by bitwise-ORing the incoming S-BF and the existing F-BF.

Algorithm 1: Procedure for receiving Bloom filters.

```
Function OnS-BF (S-BF, inFace)
    if S-BF is new then
        Combine( S-BF, F-BF[inFace] );
        foreach face except inFace do
            Forward( S-BF, face );
        end
    end
end
```

3.2. Packet Forwarding Procedure

In the proposed scheme, *Interest* packets are forwarded according to the query result of F-BFs instead of searching through an FIB table. When an *Interest* packet arrives at a node, the node looks up every F-BF stored at each face of the node. To support the longest name prefix matching in our proposed scheme, the querying to an F-BF is serially performed while decreasing the number of name components, starting from the entire name of the *Interest* packet. The detailed F-BF querying procedure is shown in Algorithm 2. Even though the querying to a Bloom filter is time-efficient, querying multiple F-BFs can be time-consuming, particularly when a node has many faces. To reduce the querying time, multiple F-BFs are merged into one functional Bloom filter [19,26], as shown in Figure 4. Each column of the functional Bloom filter represents an F-BF from each face. Cells designated by hash indexes in querying are bitwise-ANDed to obtain a result vector. Set bits in the result vector represent positive faces. In Figure 4, cells located by hash indexes 1, 4, and 8 are examined, and the F-BF stored in the nth face produces a positive result. Hence, by merging all of the F-BFs into one functional Bloom filter, the querying is performed only once for each *Interest* packet.

Bloom filters never produce false negatives, and thus the proposed scheme guarantees that *Interest* packets are always forwarded towards content sources. However, due to false positives of Bloom filters, the query results of F-BFs can be false. When the F-BFs produce positives in multiple faces, if the *Interest* is forwarded to only one of the matching faces, it might be forwarded to the false path, and packets may be lost or detoured. To secure the packet delivery, it is possible to forward packets to all matching faces; however, this wastes network bandwidth. To avoid such inefficiency, we propose using an auxiliary FIB table. The proposed scheme initially forwards *Interest* packets to all of the positive faces. When an *Interest* packet has been forwarded to multiple faces and the corresponding *Data* packet arrives at one of the faces, an entry is added to the auxiliary FIB table with the face of the first arrived *Data* packet. The auxiliary FIB table is only used when an F-BF produces positives in multiple faces. In such a case, the auxiliary FIB table is searched as well, and the *Interest* packet is forwarded accordingly. The detailed procedure for the *Interest* packet forwarding is shown in Algorithm 3.

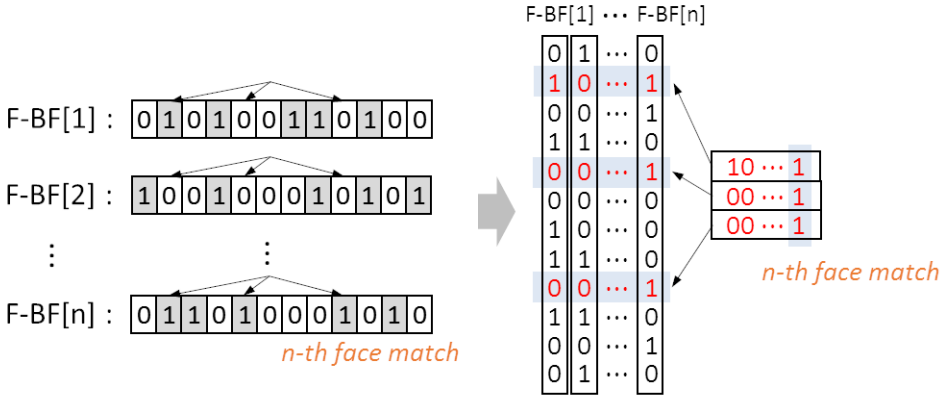


Figure 4. An example of F-BFs merging-to reduce the time for querying F-BF[i], where $i = 1 \cdots n$. F-BFs are merged into one functional Bloom filter [19,26].

Algorithm 2: Procedure for querying F-BFs.

```
Function F-BF.Query (Name)
    matchFace = ∅;
    for i = 1 to all.Face do
        BloomFilter = F-BF[i];
        (c_1, c_2, ··· , c_L) = Decompose( Name );
        subStr = "";
        matchLen = 0;
        for i = L to 1 do
            subStr = Concatenate(c_1, ··· , c_i);
            if BloomFilter.Query(subStr) then
                matchFace = matchFace ∪ Face[i];
                continue;
            end
        end
    end
    return matchFace;
end
```

The false positive rate of each F-BF differs since the number of programmed elements is different. The false positive rate for each face is $f_f = \left(1 - (1 - \frac{1}{m})^{ksn_s}\right)^k$, where s is the number of combined S-BFs in a face and n_s is the number of name prefixes in an S-BF. Since it is difficult to estimate the number of S-BFs to be combined at every node, it is not possible to derive the exact formula for the entire topology. However, the false positive rate of the proposed work is lower than $f_f \leq \left(1 - (1 - \frac{1}{m})^{kn_t}\right)^k$, where n_t is the total number of name prefixes.

Figure 5, which is adopted from [5], shows the packet handling procedure of the proposed scheme. The *Interest* packet handling procedure is illustrated in the upper part of Figure 5. The proposed scheme differs from the conventional one in several ways as follows. The F-BFs are first queried to identify matching faces. The auxiliary FIB is only looked up when the F-BF query produces positives in multiple faces. In addition, the auxiliary FIB is much smaller than an ordinary FIB table, because it only has entries for name prefixes which produce positives in multiple faces, while the conventional FIB table has entries for all name prefixes. The memory consumption for storing F-BFs is also much smaller than that for storing an FIB table.

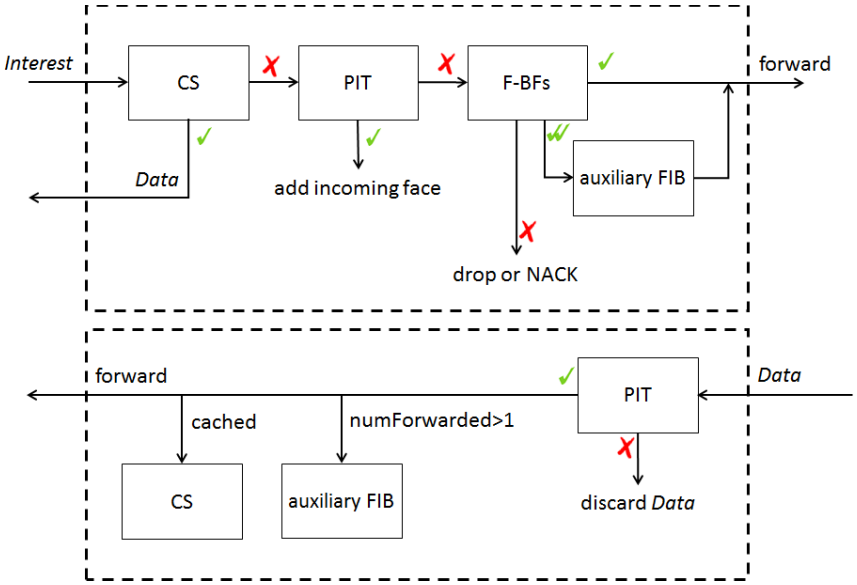


Figure 5. Packet handling procedure [5], where the upper part illustrates the *Interest* packet handling procedure and the lower part describes the *Data* packet handling procedure.

Algorithm 3: Procedure for forwarding *Interest* packets.

```
Function OnInterest (Interest)
    cachedData = ContetStore.Find( Interest );
    if cachedData then
        Forward( cachedData, incomingFace );
        return;
    else
        PIT.Add( Interest );
    end
    faceFBF = F-BF.Query( Interest );
    if faces.cnt==1 then
        Forward( Interest, faceFBF );
    else
        faceFIB = FIB.Find( Interest );
        Forward( Interest, faceFIB );
    end
end
```

The *Data* packet handling procedure is shown in Algorithm 4 and in the lower part of Figure 5. When a *Data* packet arrives as the response to an *Interest* packet, the PIT is looked up. If the matching entry is found in the PIT, then the *Data* is cached in the CS and forwarded to the faces in the matching PIT entry. The *Data* packet handling procedure is the same as the conventional one, except that the auxiliary FIB table is updated according to the first arrived *Data* if the *Interest* has been forwarded to multiple faces as explained earlier.

Algorithm 4: Procedure on handling *Data* packets.

```
Function OnData (Data)
    PITentry = PIT.Find( Data );
    if PITentry then
        CS.Add( Data );
        Forward( Data, face );
        // update FIB table if there are multiple-matching faces
        if PITentry.numForwarded > 1 then
            FIB.Add( Data, incomingFace );
        end
    else
        // actions based on policy
    end
end
```

4. Performance Evaluation

The proposed scheme was evaluated using ndnSIM, which is an NS-3 based Named Data Networking simulator [5]. We used four real topologies acquired from Rocketfuel [28]. Detailed information about the topologies is shown in Table 1. *Total nodes* represents the total number of nodes in each topology. A node in a network can be classified as backbone, gateway, or leaf, and the number of each type is also shown. *Links* represents the total number of links which connect the nodes and all links have the same delay. One thousand producers and consumers were randomly placed among backbone routers and leaf routers, respectively. Thus, the topologies used for the evaluation were equivalent to a network with more than 2000 nodes in total, including the producers and the consumers, which is the largest scale based on our knowledge. While producers were evenly connected to ten backbone nodes, the number of consumers in a single node was randomly chosen. We assumed that each name prefix was responsible for ten contents and each consumer generates *Interest* packets for a single name prefix. Real URL data acquired from the ALEXA [29] were used as name prefixes. The performance evaluation was conducted under various Zipf parameters [30], which indicate the popularity of contents. The run time for each experiment was 60 seconds, and each consumer generates ten *Interest* packets per second according to Poisson distribution. Thus, 600 thousand Interest packets in total were issued.

Table 1. Topology information.

AS	Name	Total Nodes	Backbone	Gateway	Leaf	Links
1755	Ebone	163	23	68	72	366
6461	Abovenet	176	13	33	130	289
3936	Exodus	192	39	58	95	422
1221	Telstra	279	65	45	169	731

4.1. Performance Metrics

In evaluating the performance of routing schemes, we used a set of metrics: the average hop count, the total number of *Interest* packets, and the memory consumption. The average hop count is the average number of nodes passed until the corresponding *Data* packet arrives after an *Interest* packet is sent out from a customer. Hence, the average hop count is defined as Equation (2), where C is the number of consumers, and R_i is the number of *Interests* created by customer i.

$$AvgHop = \frac{\sum_{i=1}^{C} \sum_{j=1}^{R_i} \left(Hop\left(Interest_{i,j}\right) + Hop\left(Data_{i,j}\right) \right)}{\sum_{i=1}^{C} R_i} \quad (2)$$

The total number of *Interest* packets is defined as Equation (3), where N_r is the number of routers in the network. If an *Interest* is forwarded by three routers, the total number of *Interest* packets is increased by three. This metric measures how many *Interest* packets are forwarded towards multiple faces.

$$TotalIntNum = \sum_{i=1}^{N_r} R_i \tag{3}$$

Since the lengths of name prefixes are not fixed, the memory consumption is calculated in two ways, as shown in Equations (4) and (5). The Mem_1 in Equation (4) assumes that each variable-length name prefix is stored in the FIB and the Mem_2 in Equation (5) assumes that a name prefix is stored as a 128-bit signature, where $|FIB_i|$ is the total number of FIB entries in node N_i, $Size(FIB_i[j])$ is the size of the jth entry to store a name prefix into $FIB_i[j]$, $Size(face)$ is the memory requirement for storing output faces, and $Size(signature)$ is the memory requirement for storing the signature of a name prefix. Note that output faces are represented by a fixed-length vector, of which the number of bits is equal to the number of output faces. Although an FIB table also keeps the ranks of possible faces, the memory requirement for ranking is ignored for simplicity.

$$Mem_1 = \sum_{i=1}^{N_r} \left(\sum_{j=1}^{|FIB_i|} (Size(FIB_i[j]) + Size(face)) \right) \tag{4}$$

$$Mem_2 = \sum_{i=1}^{N_r} (|FIB_i| \cdot (Size(signature) + Size(face))) \tag{5}$$

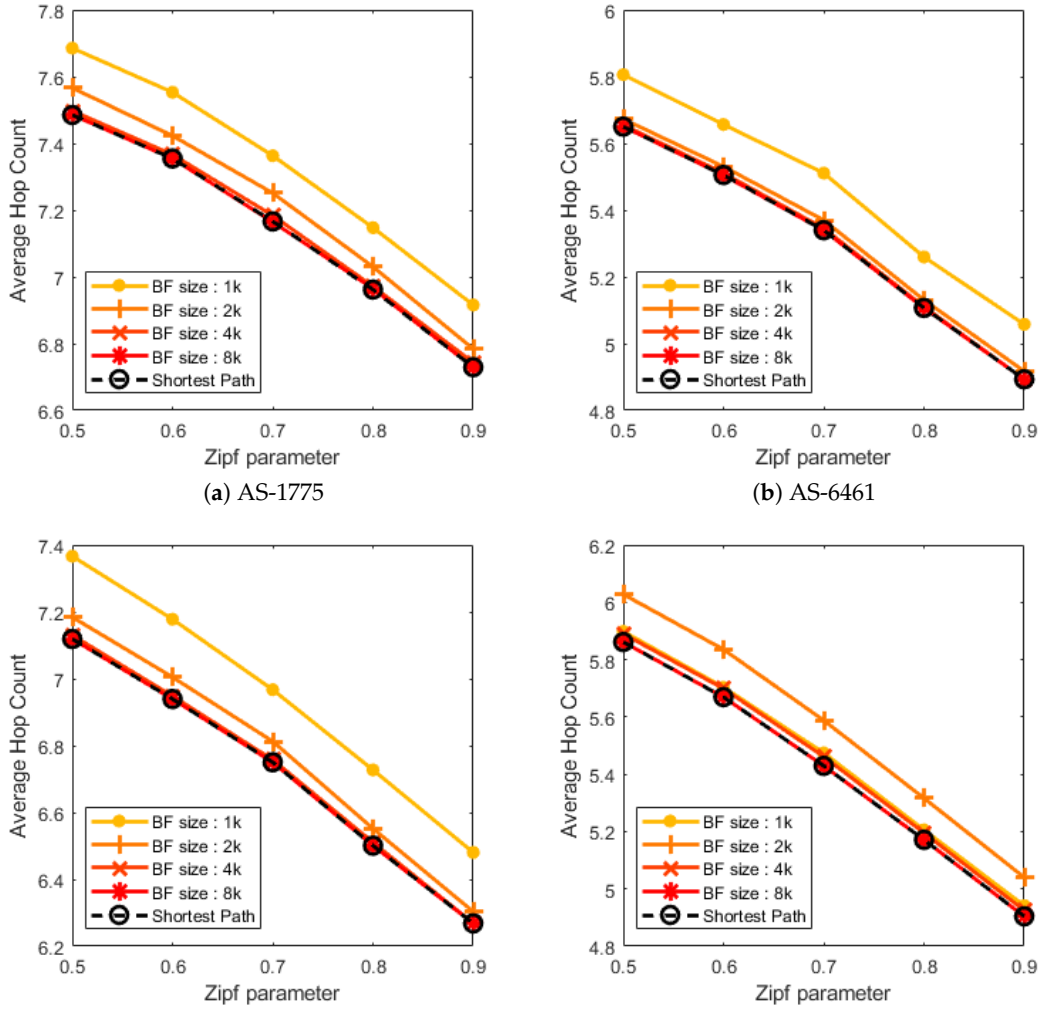
4.2. Performance of Proposed Work

We evaluated the performance of our proposed work using the various sizes of Bloom filters: 1024, 2048, 4096, and 8192 bits. The average hop count, the total number of *Interest* packets and the memory consumption were evaluated according to the Bloom filter sizes. During the experiments, it was confirmed that the consumers received the requested *Data* packet for every *Interest* packet.

Figure 6 shows the average number of hops according to the various sizes of the Bloom filter, and the performance of the proposed scheme is compared to the *shortest path* scheme, which ensures the shortest path. In Figure 6, the black dotted line is the result of the *shortest path* scheme. Across all topologies, it is shown that as the size of the Bloom filter gets larger, the average hop count converges to the *shortest path*.

Figure 7 shows the total number of *Interest* packets according to the various sizes of the Bloom filter. The *shortest path* scheme forwards an *Interest* packet only to the face with the minimum cost, while our proposed scheme forwards an *Interest* packet to all matching faces in order to avoid the delay caused by the false positives of Bloom filters. The black dotted line shows the result of the *shortest path* scheme. In all topologies, as the size of Bloom filter grows, the total number of *Interest* packets converges to the *shortest path*.

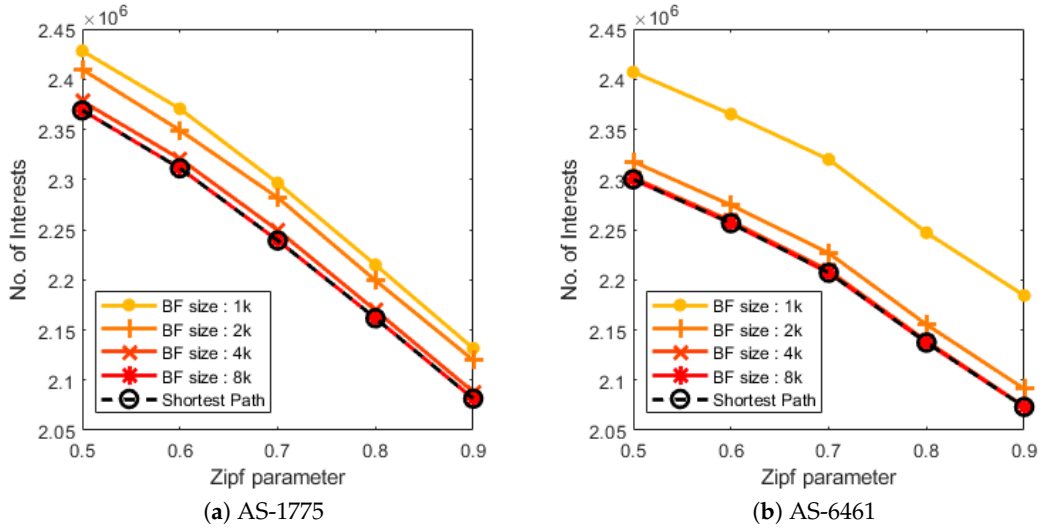
Figure 8 shows the memory consumption based on Equations (4) and (5). The memory consumption for the proposed scheme consists of the memory consumptions for Bloom filters and auxiliary FIB tables. As the size of the Bloom filter grows, the memory requirement for the Bloom filter naturally increases, but the number of auxiliary FIB entries decreases. As shown in the figure, the memory usages for all topologies are minimal when the size of the Bloom filter is 2048 bits.



(**a**) AS-1775 (**b**) AS-6461

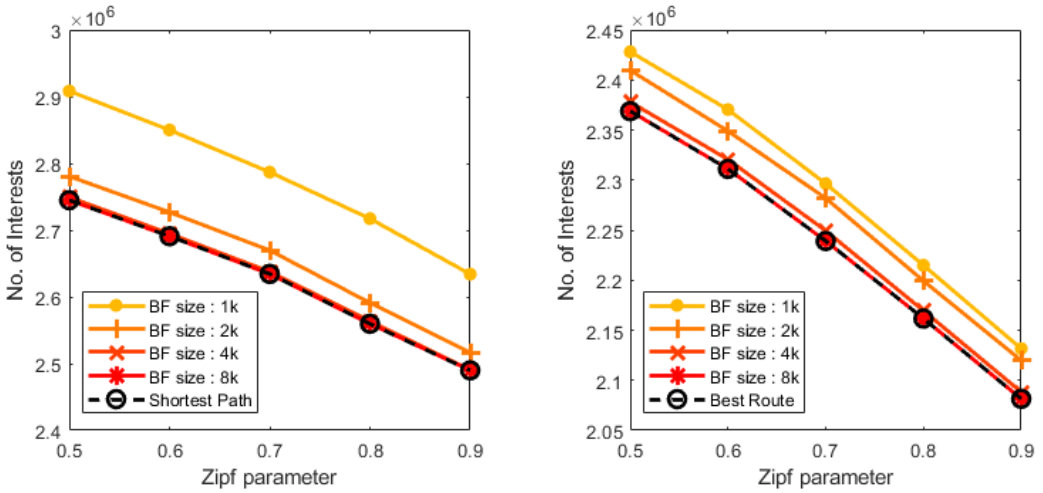
(**c**) AS-3967 (**d**) AS-1221

Figure 6. The average number of hop count under the various sizes of Bloom filters. As the size of Bloom filters grows, the data retrieval is performed along the optimal path.



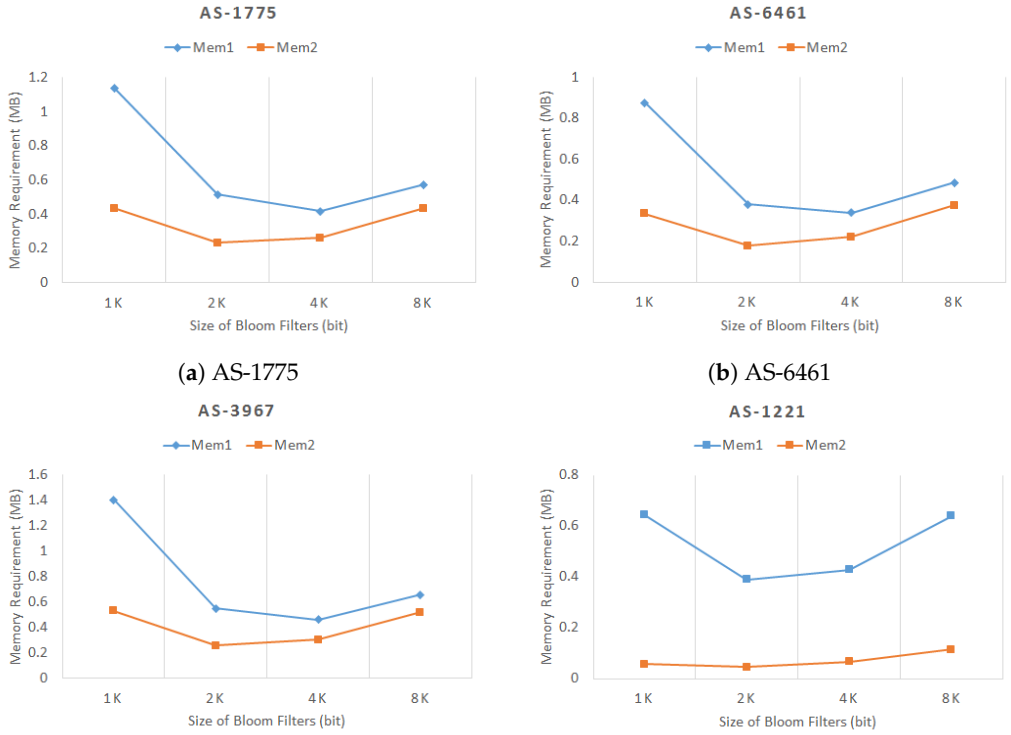
(**a**) AS-1775 (**b**) AS-6461

Figure 7. *Cont.*



(**c**) AS-3967 (**d**) AS-1221

Figure 7. The number of *Interest* packets under the various sizes of Bloom filters. As the size of the Bloom filters gets larger, the number of multicasts decreases.



(**a**) AS-1775 (**b**) AS-6461

(**c**) AS-3967 (**d**) AS-1221

Figure 8. Memory requirements based on Equation (4) and (5). As the size of the Bloom filter grows, the memory requirement for the Bloom filter naturally increases, but the number of auxiliary FIB entries decreases. In all topologies, the memory usages are minimal when the size of the Bloom filter is 2048 bits.

4.3. Performance Comparison with Other Schemes

The performance of the proposed algorithm was compared to those of *shortest path* and *flooding*. The size of a Bloom filter of the proposed work in this comparison was fixed as 2048 bits, which has the minimum memory consumption for all topologies. The *shortest path* in ndnSIM uses Dijkstra's algorithm, which guarantees the packet delivery along the optimal path using the minimum amount of Interest packets. The performance of the three schemes was compared based on the *shortest path*.

The *flooding* scheme forwards an *Interest* packet to all faces other than the incoming face of the *Interest* if no matching entry is found in an FIB table. When a *Data* packet arrives as a response, the FIB table is updated using the incoming face of the *Data* packet as the output face of the longest matching name prefix for the *Interest*.

Figure 9 compares the average hop count when the Zipf parameter is 0.7. As shown in the figure, the average hop count of our proposed scheme is very close to the *shortest path* algorithm in all four topologies, which means that the proposed scheme can achieve the near optimal packet delivery. We also conducted the simulation under various Zipf parameters, from 0.5 to 0.9, but those results are omitted for simplicity since they show a similar tendency. Both the proposed scheme and *flooding* ensure that the *Interest* packets are delivered to content sources at all times and can deliver packets along the shortest path with the same hop counts as the *shortest path* in an ideal case. However, since in-network caching is a default function, duplicated *Interest* packets in these schemes may cause in-network caches to store unnecessary contents, thus degrading the cache performance. The degraded in-network cache performance can generate more cache misses and lead to longer packet delivery paths.

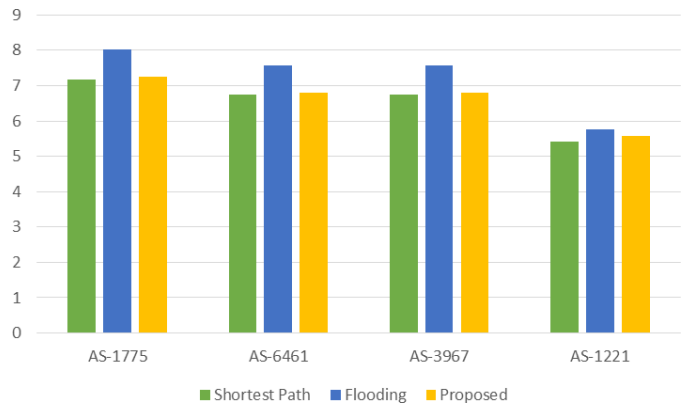


Figure 9. Comparison in average hop count. The proposed scheme can achieve the near optimal packet delivery in all four topologies.

Figure 10 compares the normalized number of *Interest* packets, which is the total number of *Interests* divided by the number of *Interests* in the *shortest path* scheme, when the Zipf parameter is 0.7. The *shortest path* scheme generates the smallest number of *Interest* packets by forwarding an *Interest* packet only to the best matching face. The proposed scheme uses only about 1% more *Interest* packets than the *shortest path* scheme, regardless of the topologies. Since the number of *Interest* packets of the proposed work is influenced by the size of a Bloom filter, it can be further reduced by increasing the size of Bloom filters. Since the *flooding* scheme forwards packets toward multiple faces, it consumes many more *Interests*. Particularly, in a large topology such as AS-1221, the *flooding* consumes almost twice the number of *Interest* packets compared to the *shortest path* scheme. Therefore, the simple *flooding* is not scalable and inappropriate for the large topology.

Tables 2–5 compare the memory consumption. For all four topologies, the *shortest path* scheme has the largest number of FIB entries, because every router in the network has the complete knowledge of all name prefixes. Meanwhile, the *flooding* scheme has fewer entries than the *shortest path*, because routers only have the name prefix entries for requested contents. Therefore, in the *flooding* scheme, if a router has never been requested for any contents, it has no entries in its FIB table. The proposed scheme has the smallest number of FIB entries in all topologies, since it updates the auxiliary FIB entries only when the *Interest* packet is forwarded to multiple faces. Overall, these results indicate that the proposed algorithm is highly scalable.

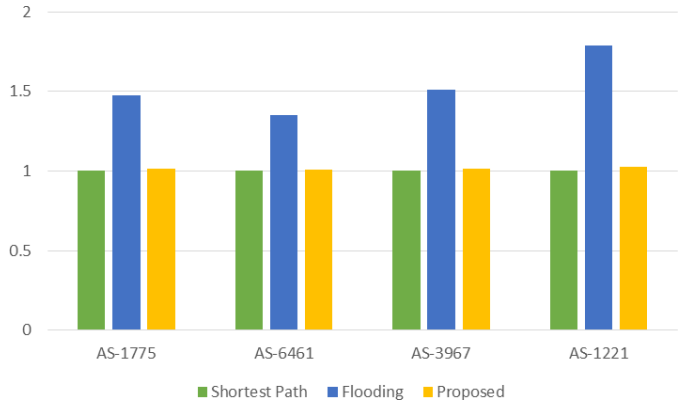


Figure 10. Comparison in the normalized number of *Interest* packets. The proposed scheme uses only about 1% more *Interest* packets than the *shortest path*, while simple *flooding* consumes up to almost twice the number of *Interest* packets.

Table 2. Memory consumption of AS-1775.

	Shortest Path	**Flooding**	**The Proposed**
Number of FIB entries	162,000	126,798	8769
Number of total characters	7,943,346	6,218,337	438,261
Number of Bloom filters	-	-	370
Memory using Equation (4) (MB)	7.73	5.89	0.52
Ratio	1	0.76	0.07
Memory using Equation (5) (MB)	2.63	2.01	0.23
Ratio	1	0.76	0.09

Table 3. Memory consumption of AS-6461.

	Shortest Path	**Flooding**	**The Proposed**
Number of FIB entries	175,000	75,394	6133
Number of total characters	8,580,775	3,694,688	308,835
Number of Bloom filters	-	-	327
Memory using Equation (4) (MB)	8.35	3.60	0.38
Ratio	1	0.43	0.05
Memory using Equation (5) (MB)	2.83	1.22	0.17
Ratio	1	0.43	0.06

Table 4. Memory consumption of AS-3967

	Shortest Path	**Flooding**	**The Proposed**
Number of FIB entries	191,000	91,008	9054
Number of total characters	9,365,303	4,452,560	449,800
Number of Bloom filters	-	-	457
Memory using Equation (4) (MB)	9.11	6.88	0.55
Ratio	1	0.75	0.06
Memory using Equation (5) (MB)	4.00	2.34	0.26
Ratio	1	0.75	0.08

Table 5. Memory consumption of AS-1221.

	Shortest Path	Flooding	The Proposed
Number of FIB entries	278,000	169,358	5700
Number of total characters	13,631,174	8,299,488	283,712
Number of Bloom filters	-	-	465
Memory using Equation (4) (MB)	13.26	8.07	0.39
Ratio	1	0.61	0.03
Memory using Equation (5) (MB)	4.51	2.75	0.21
Ratio	1	0.61	0.05

Figure 11 graphically compares the memory consumption, which is normalized by the memory consumption of the *shortest path* for the Mem_2 case shown in Equation (5). It is shown that the proposed work consumes only 5–9% of the memory consumed by the *shortest path*.

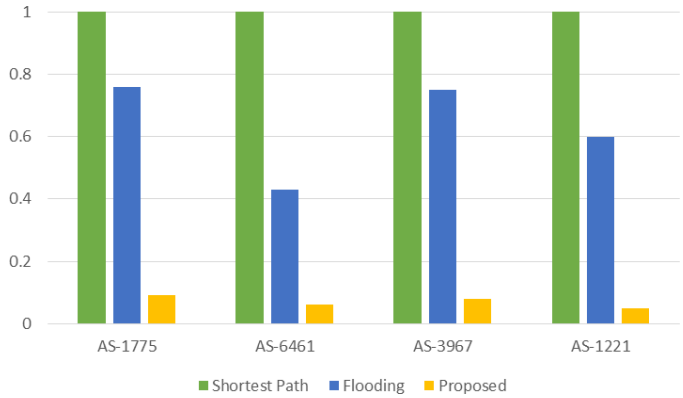


Figure 11. Comparison in memory consumption (Mem_2). The proposed work consumes only 5–9% of the memory consumed by the *shortest path*, of which routers have the complete knowledge of all name prefixes.

In our proposed work, every producer and every node which receives name prefix broadcasting from other networks needs to propagate an S-BF which contains the connectivity information. The traffic caused in propagating the connectivity information is shown in Equation (6), where N_{pro} is the number of producers, S_{BF} is the size of the S-BF, and N_l is the number of the entire links in a network.

$$T_P = N_{pro} \cdot S_{BF} \cdot N_l \tag{6}$$

For link state algorithms, the pair of name prefix and its cost should be propagated in order to create an FIB table. Equation (7) shows the traffic, where N_{pfx} is the number of name prefixes and S_{pfx} is the size of each name prefix.

$$T_{LS} = \sum_{i=1}^{N_{pfx}} S_{pfx} \cdot N_l \tag{7}$$

Figure 12 compares the traffic amounts for routing. Because the traffic amounts for routing are related to the number of links in the network, more traffic is required for the networks with more complicated connections. Hence, the amount of consumed traffic is minimum for AS-6461. While the *shortest path* scheme propagates the individual connectivity information, the proposed scheme spreads the summary of the connectivity information in the form of a Bloom filter. Therefore, the traffic used for the proposed work is much smaller than *shortest path*, which is about 6–8% of the traffic of the *shortest path*.

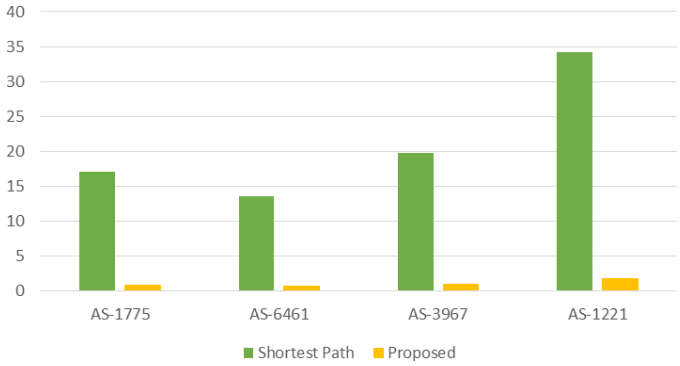


Figure 12. Comparison in traffic for routing (MB). The traffic used for the proposed work is much smaller than *shortest path*, which is about 6–8% of the traffic of the *shortest path*.

On the other hand, the *flooding* scheme does not cause any overhead traffic for routing, since it updates the connectivity information when a corresponding *Data* packet arrives after an *Interest* packet is flooded. However, since a *Data* packet has to be sent back as a response for every *Interest* packet, a lot of bandwidth is wasted to deliver *Data* packets for the flooded *Interest* packets. Moreover, the size of a *Data* packet, which carries an actual content, is usually much larger than the size of the packet carrying the connectivity information. Even though the *flooding* scheme does not lead to any traffic for routing, a huge amount of network bandwidth is wasted to retrieve *Data* packets for flooded *Interest* packets.

In the BFR [13], a Bloom filter is only used for propagating the connectivity information and the FIB is populated whenever a new *Interest* is issued. As a result, the full FIB is built for all requested contents similar to *flooding*. However, our proposed work utilizes the Bloom filter for both sharing the connectivity information and building the forwarding engine. The proposed scheme builds the lookup architecture simply by combining the receiving S-BFs, and hence it is not necessary to store the full FIB table. While there is no protection for the extra traffic and the delivery towards the wrong origin due to false positives in the BFR, the proposed scheme adapts the auxiliary FIB in order to reduce the impact of the false positives. The BFR may have to search both a FIB and Bloom filters in the case of no matching prefix in the FIB.

5. Conclusions

For the successful realization of NDN, efficient routing and forwarding algorithms are essential. This paper proposes a novel method, which combines routing and forwarding using Bloom filters in order to reduce the traffic and the memory for constructing FIB tables. The connectivity information is shared in the form of a summary Bloom filter (S-BF). Nodes collect these S-BFs and then form a forwarding Bloom filter (F-BF). Interest packets are delivered by querying F-BFs. To prevent extra traffic caused due to false positives of Bloom filters, we also propose to use an auxiliary FIB table. The performance evaluation was conducted using ndnSIM using real topologies acquired from Rocketfuel, which range from 2163 to 2279 nodes with 1000 consumers and producers. The simulation results show that the proposed scheme can achieve the nearly optimal performance as compared to the link state algorithm, while significantly reducing the network traffic for routing with the substantially smaller amounts of memory. The proposed scheme does not require any prior knowledge about network topologies and works in a fully distributed fashion.

Author Contributions: Conceptualization, J.H.M. and H.L.; methodology, H.L.; software, J.H.M.; validation, J.H.M.; formal analysis, J.H.M.; investigation, H.L.; resources, J.H.M.; data curation, J.H.M.; writing—original draft preparation, J.H.M.; writing—review and editing, H.L.; visualization, J.H.M.; supervision, H.L.; project administration, H.L.; and funding acquisition, H.L.

Funding: This research was funded by the National Research Foundation of Korea (NRF), NRF-2017R1A2B4011254.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. *Cisco Visual Networking Index: Forecast and Methodology, 2016–2021*; Cisco: San Jose, CA, USA, 2017.
2. Jacobson, V.; Smetters, D.K.; Thornton, J.D.; Plass, M.F.; Briggs, N.H.; Braynard, R.L. Networking Named Content. In Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies (CoNEXT '09), Rome, Italy, 1–4 December 2009.
3. Liang, Y.; Chen, Z.; Li, J.; Ding, S. Reality Check of CCN Routing Using Search Processor. In Proceedings of the 2012 Third International Conference on Networking and Distributed Computing, Hangzhou, China, 21–24 October 2012; pp. 16–20.
4. Mun, J.; Lim, H. Cache sharing using Bloom filters in named data networking. *J. Netw. Comput. Appl.* **2017**, *90*, 74–82. [CrossRef]
5. Yi, C.; Afanasyev, A.; Moiseenko, I.; Wang, L.; Zhang, B.; Zhang, L. A case for stateful forwarding plane. *Comput. Commun.* **2013**, *36*, 779–791. [CrossRef]
6. Choi, H.; Yoo, J.; Chung, T.; Choi, N.; Kwon, T.; Choi, Y. CoRC: Coordinated routing and caching for named data networking. In *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '14), Los Angeles, CA, USA, 20–21 October 2014*; ACM: New York, NY, USA, 2014; pp. 161–172.
7. Yi, C.; Abraham, J.; Afanasyev, A.; Wang, L.; Zhang, B.; Zhang, L. On the role of routing in named data networking. In Proceedings of the 1st ACM Conference on Information-Centric Networking (ACM-ICN '14), Paris, France, 24–26 September 2014; ACM: New York, NY, USA, 2014.
8. Hoque, A.K.M.M.; Amin, S.O.; Alyyan, A.; Zhang, B.; Zhang, L.; Wang, L. NLSR: Named-data link state routing protocol. In Proceedings of the 3rd ACM SIGCOMM Workshop on Information-Centric Networking (ICN '13), Hong Kong, China, 12 August 2013; ACM: New York, NY, USA, 2013; pp. 15–20.
9. Lehman, V.; Gawande, A.; Zhang, B.; Zhang, L.; Aldecoa, R.; Krioukov, D.; Wang, L. An experimental investigation of hyperbolic routing with a smart forwarding plane in NDN. In Proceedings of the IEEE/ACM 24th International Symposium on Quality of Service (IWQoS), Beijing, China, 24–25 June 2016; pp. 1–10.
10. DiBenedetto, S.; Papadopoulos, C.; Massey, D. Routing policies in named data networking. In Proceedings of the ACM SIGCOMM Workshop on Information-Centric Networking (ICN '11), Toronto, ON, Canada, 19 August 2011; pp. 38–43.
11. Ahmed, R.; Bari, M.F.; Chowdhury, S.R.; Rabbani, M.G.; Boutaba, R.; Mathieu, B. αRoute: A name based routing scheme for Information Centric Networks. In Proceedings of the IEEE INFOCOM, Turin, Italy, 14 April 2013; pp. 90–94.
12. Eum, S.; Nakauchi, K.; Murata, M.; Shoji, Y.; Nishinaga, N. Potential based routing as a secondary best-effort routing for Information Centric Networking (ICN). *Comput. Netw.* **2013**, *57*, 3154–3164. [CrossRef]
13. Marandi, A.; Braun, T.; Salamatian, K.; Thomos, N. BFR: A bloom filter-based routing approach for information-centric networks. In Proceedings of the IFIP Networking Conference (IFIP Networking) and Workshops, Stockholm, Sweden, 23 June 2017; pp. 1–9.
14. Lee, J.; Shim, M.; Lim, H. Name Prefix Matching Using Bloom Filter Pre-Searching for Content Centric Network. *J. Netw. Comput. Appl.* **2016**, *65*, 36–47. [CrossRef]
15. Seo, J.; Lim, H. Bitmap-based Priority-NPT for Packet Forwarding at Named Data Network. *Comput. Commun.* **2018**, *130*, 101–112. [CrossRef]
16. So, W.; Narayanan, A.; Oran, D. Named data networking on a router: fast and dos-resistant forwarding with hash tables. In Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '13), San Jose, CA, USA, 21–22 October 2013; pp. 215–225.
17. Lee, H.; Nakao, A. Improving Bloom Filter Forwarding Architectures. *IEEE Commun. Lett.* **2014**, *18*, 1715–1718. [CrossRef]
18. Quan, W.; Xu, C.; Guan, J.; Zhang, H.; Grieco, L.A. Scalable Name Lookup with Adaptive Prefix Bloom Filter for Named Data Networking. *Comput. Netw.* **2014**, *18*, 102–105. [CrossRef]
19. Wang, Y.; Pan, T.; Mi, Z.; Dai, H.; Guo, X.; Zhang, T.; Liu, B.; Dong, Q. NameFilter: Achieving fast name lookup with low memory cost via applying two-stage Bloom filters. In Proceedings of the IEEE INFOCOM, Turin, Italy, 14 April 2013; pp. 95–99.
20. Dharmapurikar, S.; Krishnamurthy, P.; Taylor, D. Longest prefix matching using Bloom filters. *IEEE/ACM Trans. Netw.* **2006**, *14*, 397–409. [CrossRef]

21. Bloom, B. Space/time trade-offs in in hash coding with allowable errors. *Commun. ACM* **1970**, *13*, 422–426. [CrossRef]
22. Tarkoma, S.; Rothenberg, C.E.; Lagerspetz, E. Theory and Practice of Bloom Filters for Distributed Systems. *IEEE Commun. Surv. Tutor.* **2012**, *14*, 131–155. [CrossRef]
23. Broder, A.Z.; Mitzenmacher, M. Network Applications of Bloom Filters: A Survey. *Internet Math.* **2004**, *1*, 485–509. [CrossRef]
24. Fan, L.; Cao, P.; Almeida, J.; Broder, A.Z. 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), Vancouver, BC, Canada, 31 August–4 September 1998; ACM: New York, NY, USA, 1998; pp. 254–265.
25. Yang, T.; Liu, A.X.; Shahzad, M.; Zhong, Y.; Fu, Q.; Li, Z.; Xie, G.; Li, X. A shifting bloom filter framework for set queries. *Proc. VLDB Endow.* **2016**, *9*, 408–419. [CrossRef]
26. Bonomi, F.; Mitzenmacher, M.; Panigrah, R.; Singh, S.; Varghese, G. Beyond Bloom filters: From approximate membership checks to approximate state machines. In Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '06), Pisa, Italy, 11–15 September 2006; pp. 315–326.
27. Sisi, X.; Yanjun, Y.; Qing, C.; Tian, H. kBF: A Bloom Filter for key-value storage with an application on approximate state machines. In Proceedings of the IEEE INFOCOM, Toronto, ON, Canada, 27 April–2 May 2014; pp. 1150–1158.
28. Spring, N.; Mahajan, R.; Wetherall, D. Measuring ISP topologies with rocketfuel. In Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '02), Pittsburgh, PA, USA, 19–23 August 2002; pp. 133–145.
29. Alexa the Web Information Company. Available online: http://www.alexa.com (accessed on 5 August 2019).
30. Breslau, L.; Cao, P.; Fan, L.; Phillips, G.; Shenker, S. Web Caching and Zipf-like Distributions: Evidence and Implications. In Proceedings of the IEEE INFOCOM, New York, NY, USA, 21–25 March 1999; pp. 126–134.