SAMBA: Scalable Approximate Forwarding For NDN Implicit FIB Aggregation

Amir Esmaeili
School of Computing
Binghamton University
Binghamton, USA
aesmaeili@binghamton.edu

Abderrahmen Mtibaa
Department of Computer Science
University of Missouri St. Louis
St. Louis, USA
amtibaa@umsl.edu

2024 IEEE 13th International Conference on Cloud Networking (CloudNet) | 979-8-3503-7656-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/CLOUDNET62863.2024.10815897

***Abstract*—The rapid growth of data-driven applications has prompted a shift towards Information-Centric Networking (ICN) in the Internet landscape. Like TCP/IP's routing tables, ICN employs Forward Information Base (FIB) tables. However, unlike IP addresses, the URL-like naming scheme in ICN can cause FIB tables to grow exponentially, leading to delays in prefix lookups. Current solutions use computationally intensive explicit FIB aggregation method, or on-demand routing schemes, which use a discovery mechanism to help reduce the number of FIB records and thus have shorter lookup times, rely on flooding-based mechanisms and building routes for all requests, introducing further scalability challenges. In this paper, we propose *SAMBA*, an Approximate Forwarding-based Self Learning, that uses the nearest FIB trie record to the given prefix for reducing the number of discoveries thus keeping the FIB table small. By choosing the nearest prefix to a given name prefix, *SAMBA* uses Implicit Prefix Aggregation (IPA) which implicitly aggregates the FIB records and reduces the number of Self Learning discoveries required. Coupled with the approximate forwarding, *SAMBA* can achieve efficient and scalable forwarding. We demonstrate that *SAMBA* can help reduce lookup times by up to 45%. *SAMBA* also implements multipath discovery and consumer-controlled flooding mechanisms, which help minimize networking overhead. Our simulation results show that *SAMBA* reduces the FIB table size twenty fold compared to traditional Self Learning schemes.**

***Index Terms*—Name Data Networking, Approximate Forwarding, FIB Scalability.**

I. Introduction

In the evolving landscape of network architectures, Information-Centric Networking (ICN) paradigm and its most prominent architecture Named Data Networking (NDN) architecture represent a paradigm shift that prioritizes data retrieval based on content rather than traditional host-based addressing [1]. Central to the efficiency and effectiveness of ICN/NDN is the routing mechanism, which underpins the entire framework by finding and storing routes at each node. NDN consumers send *interests* to fetch any Data provided by a producer or stored in the network (*e.g.,* CDNs). ICN/NDN uses a Forward Information Base (FIB) to store routes computed by a routing algorithm, which can be used by the NDN forwarding plane to send and forward each interest message towards its producer [2].

Scalability is a significant challenge in NDN due to the nature of the Forwarding Information Base (FIB) tables. Unlike traditional IP routing tables, which manage routes based on relatively stable, hierarchical IP addresses, FIB tables, however, must handle a vast and dynamic range of content names [3]. These names are often hierarchical and much longer, leading to an exponential increase in the size and complexity of the FIB entries. As a result, the memory and processing power required to manage these tables grow significantly, making them less scalable. Additionally, the frequent addition and deletion of content names exacerbate the challenge, as FIB tables must dynamically adapt to these changes in real-time. Thus, developing scalable solutions for FIB management is critical to ensuring efficient and robust performance.

Researchers have been addressing this scalability challenge by proposing: (i) optimized data structures for faster FIB lookups [3]–[6], (ii) compression and FIB aggregation mechanisms to reduce name prefix sizes [7], [7]–[9], (iii) hardware-based FIB implementations for faster lookups [10], [11], (iv) on-demand routing schemes which discover routes as needed resulting in smaller FIB table sizes [9], [12], [13] when compared to proactive routing approaches [14], [15]. These solutions are not scalable due to collision (i), adding an extra layer of complexity (ii), requiring specific hardware (iii), perform an explicit offline aggregation (ii), or do not scale well under high demand.

In this paper, we propose *SAMBA*, a scalable NDN forwarding system that uses an Approximate Forwarding algorithm (AF) to forward interests based on a best-effort fashion, and when forwarding fails, an optimized Self Learning discovery is initiated to find and store as many routes to the producers on the reverse path. Therefore, *SAMBA* employs an Implicit Prefix Aggregation (IPA) to keep the size of the FIB table small, along with the Approximate Forwarding algorithm (AF) which performs a Longest Prefix Match (LPM) first, and when that fails sol chooses the first available FIB record on the subtrie. *SAMBA*'s forwarding system creates an Implicit Prefix Aggregation (IPA), which implicitly aggregates the FIB entries, unlike state-of-the-art explicit FIB aggregation schemes which incur an extra computation overhead to aggregate or compress the FIB records [7], [9], or predictive forwarding approaches [5], [16] that are unable to recover the correct path when wrong forwarding happens, *SAMBA* discovers correct path via broadcasting.

Our Contributions: In this paper, we make the following

contributions:

- We empirically measure the potential gains of reducing the FIB size on every router lookup operation.
- We propose *SAMBA*, Approximate Forwarding that uses the nearest FIB record to the given prefix to avoid unnecessary route discoveries, resulting in reducing the FIB size, thus it provides an IPA. It can also recover incorrect paths via broadcasting ensuring correct forwarding.
- *SAMBA* also implements multipath and a Stop-and-Wait feature, which help build alternate routes and reduce flooding in the network respectively.
- Our simulation results compare the performance of *SAMBA* to state-of-the-art Self Learning, and show major improvements, including reducing FIB size by up to 20×, overhead by 75%, and up to 50% more throughput during link failures.

The remainder of this paper is organized as follows. In Section II, we survey related work and existing schemes. Section III motivates the need for controlling the FIB table size. Section IV presents the detailed design of *SAMBA*. Section V evaluates *SAMBA*'s performance. Section VI concludes the paper and presents future research directions.

II. Related Work

Routing and forwarding are two important modules in the ICN/NDN, while routing finds available routes for destinations and fills the FIB table, forwarding selects the next hop for a given incoming interest from FIB table [17]. A key function in forwarding that operates over the FIB table for every incoming interest is the lookup function. The performance of the lookup function depends on the number of FIB entries, making FIB scalability a major challenge.

FIB scalability in NDN has been extensively explored across five research categories:: (i) optimized data structures for faster FIB lookups [3]–[6], (ii) compression and FIB aggregation mechanisms to reduce name prefix sizes [7], [8], (iii) hardware-based FIBs [10], [11], (iv) on-demand routing schemes [9], [12], [13] which discover routes as needed resulting in smaller FIB table sizes when compared to proactive routing approaches [14], [15], [18], and (v) predictive forwarding [5], [16] which speculatively forwards an interest when a longest prefix match is unavailable.

However, while optimized data structures (i) such as hash-based solutions [3] are not scalable due to collisions, hardware-based methods (ii), though fast in operation, are complex to implement and incur an additional cost. On the other hand, prefix compression, and coding solutions [7], [8] operate for each interest received, incurring a complex computation at the routers. Aggregation-based methods [4], however, are run periodically on the entire FIB tables, which can be complex, slow, and inefficient. We, in this paper, propose *SAMBA* that uses an implicit aggregation method which reuses "similar" FIB records keeping the FIB tables small in size.

On-demand routing (iv) and speculative forwarding (v) are critical areas of research that closely align with the focus of our work. On-demand schemes (iv) [9], [12], [13], [19], [20], on the other hand, compute paths reactively (*i.e.,* on-demand, when needed), by broadcasting a discovery interest to establish a route at the network for the requested producer. On-demand schemes keep the FIB size "smaller" and "manageable" by storing only necessary routes. Self Learning [9], one of the most popular on-demand routing schemes uses a *discovery mechanism* to find new routes (when "regular" interests fail to find one) and store these routes on all nodes on the path. While Self Learning approaches [9], [12] reduce the number of FIB entries in the routers' FIB tables, these tables can still grow as the number of requests scale. Finally, forwarding based on speculation is another approach to keep the FIB smaller. Fuzzy forwarding [16] exploits semantic similarity to find corresponding records which is highly complex to construct in vector space, and speculative forwarding [5] uses a token-relaxed Patricia trie as a data structure and forwards interests speculatively. This method, however, fails to address the challenge of finding a route when speculative forwarding fails to reach the producer.

We propose *SAMBA*, which leverages features in both Self Learning and Approximate Forwarding (AF) to forward interests to the closest FIB record for any given prefix, however when forwarding fails, an updated Self Learning discovery is initiated to identify as many routes as possible. *SAMBA* is designed to minimize the number of "unnecessary" discoveries to maintain the FIB tables as small as possible.

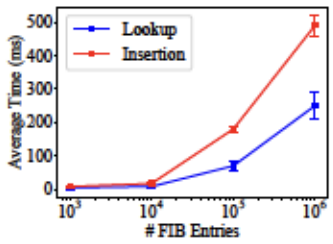


Fig. 1: Scalability assessment of FIB lookup time; average of 100 FIB lookup and insertion times for as the FIB table size scales from 1k to 1M entries (x axis in logscale)

III. Why FIB Size Matters?

To quantify the impact of the increasing number of FIB entries at the routers on lookup and insertion operation delays, we conduct a series of benchmarking experiments using a desktop machine equipped with an Intel Core i5 CPU and 8 GB of RAM. We implement the FIB data structure as a trie [21][1]. We generate tries that consist of FIB entries for prefixes of lengths up to 50 characters. We implement our trie such that each node represents a single character. This design allows our trie to store prefixes with heights of up to 50.

We vary the size of the trie from 1K to 1M entries and perform an average of 100 randomly generated prefix lookups and insertions. Each lookup and insertion operation is repeated 20 times and averaged. Results are shown in Figure 1.

[1]Tries are known to perform faster lookups than hierarchical hash tables [4]

Figure 1 illustrates that as the number of FIB entries increases, the lookup and insertion time increases almost linearly. unlike binary trees, tries have a larger lookup time and does not scale well as the number of FIB entries grows [22]. Note that while the exact lookup and insertion time may vary depending on the implementation, the hardware, the language, etc., we are more interested in the trend showing the potential impact of a reduction in FIB table size on the performance. For instance, the figure shows that a FIB size reduction of 10×, say from 1M to 100K entries can reduce the lookup time by up to 45%. This time saved will be applied to every single interest processed by the router resulted in major performance improvement across the network and a much better quality of experience for the end consumers. This paper aims to utilize the minimum number of FIB entries while maintaining the same (*i.e.,* or similar) routing/forwarding performance.

IV. *SAMBA*

In this section, we present *SAMBA*, a scalable approximate forwarding that employs an enhanced Self Learning-based routing scheme that reduces the number of discoveries while keeping the FIB tables sizes as small as possible. We first argue why self learning schemes are more suited to be coupled with an approximate forwarding for faster forwarding at scale.

A. Why LPM is Not Suited for Self Learning Schemes?

Longest prefix matching (LPM) is commonly used by most ICN/NDN routers to determine the list of next hops for any given prefix (*i.e.,* producer). While LPM is very efficient and fast for most routing schemes, in Self Learning, routes are set on-demand. This means there may not be a route for all legitimate prefixes. In this case, LPM will fail to find a list of next hop faces, triggering a discovery for a new route. *We argue that this process is sub-optimal and can lead unnecessary route discoveries, resulting in larger FIB tables and network overhead.* In fact, often producers implement multiple services, or a server may host multiple tenant and/or multiple stakeholders services, therefore a partial name prefix of the producer or the server can suffice to serve many prefixes without the need to discover all or/and store them all in the FIB [9].

For simplicity, let us assume that a given prefix p, of length k, has the following form: "`/<domain1/domain2/.../domaink`". p is stored in the FIB as a branch in the trie of height k, where the leaf, "`domaink`" includes a pointer to the outgoing interface(s) for any interests matching p. Self Learning schemes, similar to proactive routing schemes, set records in the FIB for all prefixes, up to "`domaink`", which creates a very dense FIB trie structure as shown in Fig. 2 (subfigure in the left), while *SAMBA* implicitly aggregates FIB records and re-utilizes existing similar records for new incoming interests which creates a sparse and easy to search trie structure as per the right subfigure in Fig. 2. For instance, if a node receives a new interest "`/<domain1/domain2'/...`", it will first try the route set by the record "`/<domain1/domain2/.../domaink`", however, if that routes fails, then and only then the node initiates a new discovery and stores a new route. This simple IPA mechanism reduces the trie size considerably by employing an implicit FIB aggregation mechanism which is lightweight compared to an explicit offline aggregation which operates on the entire FIB and can be very costly [9].

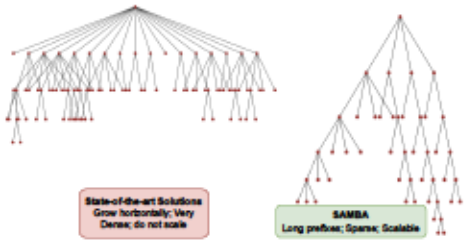


Fig. 2: Comparing the FIB properties using state-of-the-art solutions [9] (left) and *SAMBA* (right). *SAMBA* inserts very few prefix entries, often longer but act as an implicit aggregate/summarization for the entire trie branch.

Besides this simple idea, throughout the paper, we will address, analyze, and discuss the following research questions:

1) What is the complexity introduced by the new AF algorithm compared to the well-studied and optimized longest prefix matching algorithm [9]?
2) What additional delay costs are incurred by *SAMBA* when it fails to reuse FIB records?
3) Can IPA achieve compression performance comparable to that of explicit FIB aggregation?

B. Approximate Forwarding (AF) & Implicit Prefix Aggregation (IPA)

Consider a trie, a string-indexed look-up data structure, consisting of a set of nodes. All the children of a node have a common prefix of the string associated with that parent node. The trie is rooted "/" (refer to the trie example consisting of 6 leaf nodes in figure 3). To determine the next hops for any given prefix p, The AF algorithm works as follows: (i) First it operates similar to the longest prefix matching (LPM) algorithm by searching from the root of the trie "/" until it reaches a leaf node, *i.e.,* a node in the trie which stores next hop faces for the given prefix, or a normal trie node, which we will refer to as stopping node (Sn). (ii) if a leaf node is reached then AF returned exactly the same next hop than LPM, however AF operates differently when Sn is reached– *i.e.,* LPM fails to find a next hop face. AF performs a simple lightweight depth-first search (DFS), searching for a next hop in the sub-trie rooted at "`Sn`". Note that the DFS does not search the strings but simply checks if any next hops exists in the sub-trie. (iii) If DFS successfully found a list of next hop faces, say f, then AF returns f as a forwarding face unless f is a `local` face, then AF returns a `NOROUTE NACK` (Alg. 1). This message triggers a new discovery message in consuemr side to discover the correct route.

Thus, the interest will follow an approximate path towards a given producer; if the correct producer is reached then data is sent back to consumer(s), otherwise a given producer receiving

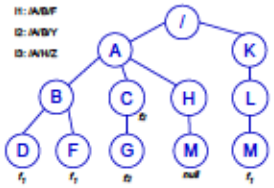


Fig. 3: Trie example: (I1) AF performs similar to LPM and returns f_1; (I2), AF runs a DFS on the sub-trie rooted at B and returns f_1 as it finds leaf node D; (I3) AF runs DFS on the sub-trie rooted at H which fails to find a next hop, thus AF returns a NOROUTE.

the interest while unable to satisfy it (*i.e.,* when DFS returns f which is a `local` face, the interest is about to be sent to the incorrect producer application), sends a NOROUTE NACK to the consumer. This AF's NACK will trigger a *discovery interest* by the consumer and a new prefix p for the producer will be added to all nodes in the path, similar to the Self Learning mechanism.

To illustrate how AF works, we depict in Figure 3 a sample trie at a given node, u, consisting of six leaf nodes (*i.e.,* nodes with a next hop face, $f_i \neq null$). If u receives the interest "`/A/B/F`" (I1), AF performs a longest prefix match (LPM) and finds a next hop face, $f1$, at node F. However, if u receives interest "`/A/B/Y`" (I2), LPM fails, thus AF runs a DFS at node B and finds a forwarding face $f1$, at node D. If f_1 is not a *local* face then AF forwards the interest to f_1, otherwise it sends a NOROUTE NACK to the consumer. Finally, if u receives interest "`/A/H/Z`" (I3), AF runs DFS at node H, which fails to find any leaf in the sub-trie, resulting in sending a NOROUTE NACK to the consumer.

Note that AF uses DFS when longest prefix matching fails. While this functionality may introduce an extra delay overhead, we argue that the DFS search of a leaf is fast with a major benefit consisting of preventing unnecessary discovery flooding. The AF algorithm will create a FIB with that prefixes are implicitly aggregated (IPA).

Algorithm 1 Approximate Forwarding (AF) algorithm when receiving an interest with name prefix p_i

Require: p_i
1: $Sn \leftarrow LPM('/')$
2: **if** $IsLeaf(Sn)$ **then**
3: Return $Sn.f$ // LPM found a face
4: **else if** $f \leftarrow DFS(Sn)$ **then**
5: **if** $isLocal(f)$ **then**
6: Return NOROUTE // DFS successful but producer
7: **else**
8: Return f // DFS successful, forwarder
9: **end if**
10: **else**
11: Return NOROUTE // AF: No face found in sub-trie
12: **end if**

C. Multipath Discovery: Finding Alternate Routes

As described in the previous section, IPA achieves an implicit aggregation of the FIB and maintains a smaller FIB size by trying approximate faces (*i.e.,* routes). However, when these faces fail to reach the correct producer (*i.e.,* either because the wrong producer is reached or a node with no face is encountered), a NOROUTE NACK received by the consumer initiates an interest discovery to store a new, and more accurate, route at the FIB.

SAMBA implements a multipath feature allowing any node to exhaust all forwarding faces prior to forwarding the NOROUTE NACK back towards the consumer, thus reducing the overhead of initiating a new broadcast-based discovery. Note that storing multiple faces at the leaf nodes does not increase the size of the FIB, but simply adds only a few bytes to the leaf nodes.

SAMBA implements a NACK handling algorithm, as described in Alg. 2, which checks the FIB for another alternative route for the same given prefix p_i and if say a node u successfully finds a face, f, for p_i, u informs all nodes downstream and the consumer of the existence of an alternative route which was not tested. Consumer then can quickly (*i.e.,* without waiting for a timeout), re-transmits the same interest which will follow the same path until reaching node u, which can try the alternative route towards the producer. This process can continue until all nodes exhaust all available faces leading to producer or abruptly interrupted by consumer, after few attempts, to send a discovery interest and store a new route to producer. Alg. 2 allows a given node u to change the NACK reason to ALT_ROUTE to inform all nodes downstream of the existence of an alternative, untested route.

This multipath feature reduces the need to broadcast many interest discovery messages which will be flooded, thus adding a major overhead and unnecessary congestion. We will evaluate the impact of this feature on latency and overhead in the evaluation section.

Algorithm 2 OnReceiveNACK: *SAMBA*'s multipath feature $< NACK_{p_i} >$ from face $oFace$

Require: $NACK_{p_i}$, $oFace$
1: **if** $Reason(Nack_{p_i}) ==$ NOROUTE **then**
2: $RemoveFaceFromLeaf(p_i, oFace)$ // Remove failed face and check if another one exists
3: **if** $f \leftarrow ExistAnotherFace(p_i)$ **then**
4: $SetNackReason(NACK_{p_i}$, ALT_ROUTE) // If another face exists, inform downstream
5: $Send(NACK_{p_i}, iFace)$
6: **else**
7: $RemoveFIBEntry(p_i)$
8: $Forward(NACK_{p_i})$
9: **end if**
10: **else if** $Reason(Nack_{p_i}) ==$ ALT_ROUTE **then**
11: $Forward(NACK_{p_i})$ // Nodes receiving ALT_ROUTE NACK keep their face to test alternative route
12: **end if**

Now, we describe how *SAMBA* discovers and stores multiple routes for the same prefix. Note that discovery interests are broadcasted and do not follow LPM or AF mechanisms. We use a similar approach to Shi *et al.* 's approach which implements a temporary data structure to save the PIT record for a given time allowing multiple Data messages are to be received for the same interest discovery [12]. While Shi *et al.* 's approach works well for paths with two or more egress links, it fails when two paths have a node with two ingress links. This failure is due to the loop prevention mechanism which prevents any node of accepting two interests with the same nonce [23]. *SAMBA* overwrites this rule, if and only if it receives a discovery interest. *SAMBA* allows multiple interest discoveries to be received, their incoming faces added to PIT, however only the first interest is forwarded/broadcasted to the upstream faces. This change will not create looping interests (*i.e.,* interests forwarded indefinitely in the network), but yet allows a data received by the node to be sent to all incoming faces which help discover multiple routes instead of one. To allow that, when an interest discovery is received by a forwarder, and it is already in the PIT table, the forwarder node adds the interest incoming face (iFace) to the corresponding PIT record and silently discards the received duplicate interest (Alg. 3).

Algorithm 3 OnReceiveInterestDiscovery $< I >$ from $iFace$

Require: I, iFace
1: **if** $ExistsInPIT(I) == 0$ **then**
2: $InsertPIT(I, iFace)$ // First discovery interest is saved and broadcasted to all faces except iFace
3: $BroadcastToAllFaces(I, oFaces)$ // Broadcast discovery to all outgoing faces, oFaces
4: **else**
5: $AddIncomingFace(I, iFace)$ // Append incoming face for duplicate discovery interests
6: $DiscardInterest(I)$ // Duplicate discovery interests are dropped
7: **end if**

On receiving a discovery Data message (*i.e.,* a Data with a discovery tag), a forwarder node, u, forwards the message to all downstream paths, iFaces, which create multiple ingress paths when $|iFaces| > 1$. To help create multiple egress path, u forwards the first Data message D downstream and removes all incoming faces, iFaces, as well as the outgoing face where the Data is received from, oFace, and set an expiry time for the PIT record. Note that *SAMBA* keeps the PIT alive and all outgoing faces which the node did not receive any Data from yet, for potential alternative route announcement on those outgoing faces. This mechanism, similar to the one proposed by Shi *et al.* [9], allows discovering multiple egress paths. Alg. 3 and Alg. 4 provide details on how interest discoveries and Data discoveries are handled by *SAMBA* respectively.

[1]Same prefix and same nonce, which we refer to as interest (I) in Alg. 3

Algorithm 4 OnReceiveDiscoveryData $< D >$ from interface $oFace$

Require: D, $oFace$
1: **if** $iFaces \leftarrow PIThasFace(D)$ **then**
2: $FIBinsert(D, oFace)$ // Data from first path arrived
3: $SendDataDownstream(D, iFaces)$
4: $RemoveFacePIT(iFaces, oFace)$ // Keep outgoing faces which did not send data yet
5: $PitRecordExipreTimer(D, tmp)$ // Set a timer tmp for the PIT record to expire
6: **else if** $tmp > 0$ **then**
7: $FIBinsert(D, oFace)$ // Insert an alternative egress path
8: $RemoveFacePIT(D, oFace)$
9: **else**
10: $sendNACK($`UNSOLICITED_DATA`$)$
11: **end if**

D. SAMBA's Consumer Stop-and-Wait Mechanism

Most applications send multiple messages to the same producer. To avoid sending numerous interest discovery requests as soon as an application sends an interest to a given producer, *SAMBA* implements a Stop-and-Wait consumer mechanism. This mechanism queues interests for any given prefix that has an ongoing discovery. Using this mechanism, consumers do not send multiple interest discoveries for the same prefix name, thus avoiding unnecessary overhead caused by sending multiple "redundant flooding messages" to discover the same route.

SAMBA's Stop-and-Wait mechanism, as depicted in the flowchart in Figure 4, aims to queue all interests sharing the same prefix p with an ongoing current discovery. The queued interests wait until: (1) a new route is discovered and stored in the FIB, or (2) a timer expiration triggers a new discovery process for the same prefix p.

SAMBA's consumer initiates a route discovery for prefix p as soon as it receives `NOROUTE` NACK, and starts a timer for the requesting interest with prefix p. Any new interest received by the consumer app sharing the same interest p, will follow the Stop-and-Wait mechanism described above (Figure 4). When a new route is discovered, all queued interests are sent with the highest priority (*i.e.,* prior to new interests).

V. Evaluation

In this section, we compare the performance of *SAMBA* against state-of-the-art Self Learning algorithm [21]. First, we describe our evaluation methodology, metrics, and then we study the impact of various parameters. on *SAMBA*'s performance.

A. Simulation Setup

We use ndnSIM [24], a module of ns-3 [25], to implement and evaluate *SAMBA*. We perform our simulation on a Desktop machine with a 4-core i7 Intel CPU and 8 GB memory.

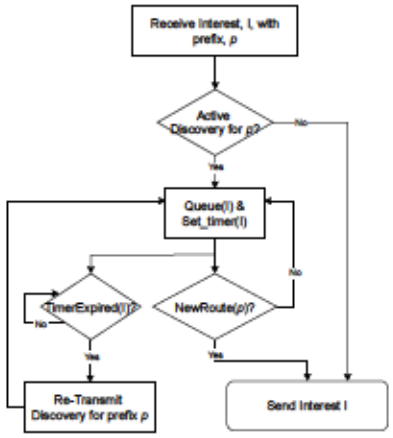
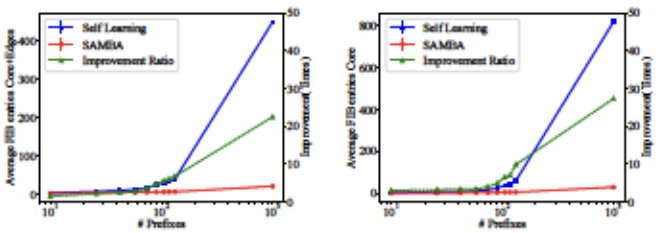


Fig. 4: Flowchart diagram of *SAMBA*'s Stop-and-Wait mechanism. Interests are queued waiting for the creation of a new route or a timeout of a current active discovery.



(a) Avg. FIB records in core+edge routers (b) Avg. FIB records in core routers

Fig. 5: Comparison of *SAMBA*'s and Self Learning's average FIB table size as the number of prefixes, C, increases. The improvement ratio, in the right y-axis, increases as the number of prefixes per producer increases (x axis in logscale).

Network Topology: We created an ISP-like network topology using NetScaNDN [26] topology maker module, consisting of N nodes, including R routers, C Prefixes (consumers), and P Producer Connection Points. The R routers consist of R_c core and R_e edge routers. While the core routers are connected to three other core routers, edge routers, however are connected randomly to 1, 2, or 3 core routers (and to none of the edge routers). Edge routers are also used as access nodes for consumers and producers. We connect randomly each consumer and producer to one and only one edge router. While we vary C and P in our simulation, we fix $R_e = 16$ and $R_c = 21$. In our topology the average path length between any consumer and producer is roughly 3.2 hops.

Implementation: We implement AF forwarding strategy at each node in the network, including forwarders, producers, and consumers. Consumers also implement *SAMBA*'s Stop-and-Wait mechanism as described in section IV-D. Consumers' apps, starting at a random time spanning from beginning of simulation and 50 seconds after, send requests at a rate of 8 interest per second. Each consumer i sends interests with the same prefix p_i for a given producer u. interests have the following name format: "$/p_i/\texttt{seq}$", where seq is the sequence number for consumer i. We set all producers to produce a total of $M \geq P$ prefixes, such that each producer can produce one or more prefixes. In our simulation, we disable in-network caching to evaluate the features introduced by *SAMBA* without any unknown parameter such as caching.

We perform twenty simulation runs for each experiment in which we vary the topology and the producer prefix association. Each simulation run is set to 60 seconds.

B. Evaluation Metrics

We use the following three metrics to evaluate *SAMBA*:

1) *Average number of FIB entries*: We measure the number of FIB entries for a given router as the number of leaves in its FIB trie. This metric is proportional to the number of nodes in the trie. We measure the average number of FIB entries for all routers as well as for core-only.
2) *Network overhead*: Measured as the number of broadcasted discovery interests (*i.e.,* interests with a discovery tag).
3) *Average number of redundant paths*: We measure the number of disjoint paths for each prefix at each router and compute the average per prefix and per router.

C. Results and Analysis

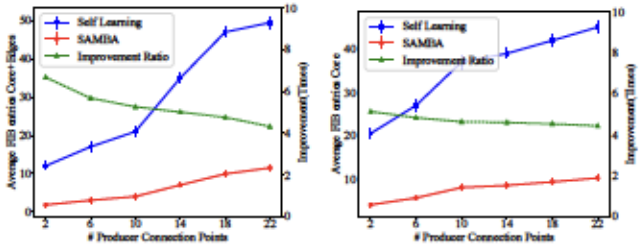
We design *SAMBA* to reduce the FIB size and thus reduce the lookup latency as shown in Figure 1. We first compare the FIB sizes of *SAMBA* and Self Learning.

D. SAMBA's FIB Table Size

We vary the number of prefixes per producer to evaluate how these two algorithms, *SAMBA* and Self Learning, construct their FIB tries and which one scales better.

We first fix the number of producers in the network, $P = 4$, and vary the number of prefixes, C. As C increases, producers serve more and more prefixes (*e.g.,* a Google server can serve Gmail, Drive, Calendar, and Photo services). We plot, in Figure 5, the average FIB size of all nodes in the network (Fig. 5a) and for core nodes only (Fig. 5b), using *SAMBA* and Self Learning, as we increase the number of requested prefixes, C. We show that *SAMBA* outperforms Self Learning and keeps the size of the FIB smaller as C increases. *SAMBA*'s FIB size does not exceed 22 and 35 prefixes on average for all routers and core routers respectively, while Self Learning's FIB size increases exponentially, exceeding 420 and 805 prefixes respectively.

Additionally, we measure the improvement ratio is measured as the fraction of average number of FIB records in Self Learning to average number of FIB records in *SAMBA*, and show that *SAMBA*'s improvement increases as the number of prefixes per producer increases. In fact, *SAMBA* re-uses the same saved route for the producer, thus the more prefixes served by unique producer, the better performance achieved



(a) Avg # FIB records in core+edge routers when P changes (b) Avg # FIB records in core routers when P changes

Fig. 6: Comparison of *SAMBA*'s and Self Learning's average FIB table size as the number of number of producer connection points), P, increases. The improvement ratio increases as the number of prefixes per producer increases.

by *SAMBA*. Self Learning, however, re-discovers all paths to all prefixes and thus the size of its FIB trie (*i.e.,* or table) increases exponentially. The improvement ratio increases from $2\times$ and $4\times$ when $C = 10$, to $20\times$ and $30\times$ when $C = 1k$ for all routers and core routers respectively. This improvement gain increases show how *SAMBA* can scale better compared to Self Learning, thus reduces the trie search and lookup times as shown in Figure 1.

We also fixed the number of prefixes, $C = 100$, and vary the number of producers connection points, P, to investigate how *SAMBA* scales as the number of producers increases. Our results in Figure 6 show that while *SAMBA* keeps the FIB size small compared to Self Learning, the improvement ratio decreases as P increases. The improvement ratio for all routers decreases from $6\times$ to $6\times$ as P increases from 2 to 22. However, this improvement decreases much less for core routers (from $5\times$ to $4.3\times$), which are more important in any networking architecture, thus keeping their FIB smaller is more critical than edge routers. As the number of prefixes per producer connection points decreases–*i.e.,* P increases, *SAMBA*'s AF algorithm will fail to use accurate approximate routes resulting in more and more discoveries. However, we note that these discoveries remain smaller than those initiated by Self Learning.

E. Overhead Analysis

In addition to *SAMBA*'s main objective reducing the FIB size, we argue that *SAMBA* also uses much less message overhead when compared to other state-of-the-art routing schemes. While proactive solutions have much higher overhead, we compare *SAMBA* to the Self Learning reactive approach to quantify when and how *SAMBA* utilizes fewer discoveries to achieve the same delivery performance.

However, we measure the network overhead as number of interest discovery messages that flooded over the network, in both experiments shown in section V-D, *SAMBA* has lower overhead compared to Self Learning. Figure 7a shows that *SAMBA* is able to reuse FIB records around 20 times more than Self Learning. It, therefore, generates less overhead than Self Learning, especially when the number of requested different prefixes, C, increases. For instance when $C = 1k$, Self Learning flooded 150k interests to discover new routes, however, *SAMBA* did not send more than 8,087 interests and reused most of the pre-existing FIB records to forward consumers' interests. After reusing pre-existing FIB records, *SAMBA* failed in only 10% interests and ended up sending new discovery interests to add new routes.

Also, the number of discovery messages in the second experiment when number of producer connection points P is varying, increases for both *SAMBA*, and Self Learning, however, *SAMBA* has better results in decreasing the number of discovery flooding in the network as it is depicted in Figure 7b.

Other overheads of *SAMBA* is the delay and additional discovery interest due to wrong forwarding. While the wrong forwarding happens just in the first interest of a connection and the delay of finding a new path is less than a Round Trip Time (RTT), so these overheads are not considerable and it is still fewer than the number discovery messages in Self Learning.

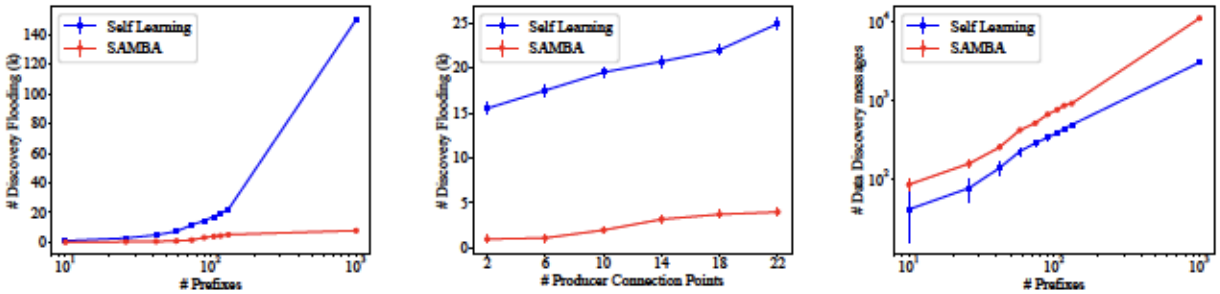
We also evaluate the overhead of *SAMBA* for detecting the multipaths by measuring the number of data discovery messages that return on the router link. While in Self Learning just a path is discovered, and duplicate discovery interests from multiple paths dropped, *SAMBA* discovers all available paths, consequently number of data discovery messages increases (Figure 7c). This additional overhead of *SAMBA* is doubled compared to Self Learning. However, we will show how alternative paths can increase the *SAMBA* performance in presence of link failure.

F. SAMBA's Performance in Presence of Link Failures

In this section, first, we compare the level of path redundancy in *SAMBA* and Self Learning. To this end, we measure the Average number of Paths per Prefix (APP) in the core and edge routers respectively. In this experiment we use the network topology of section V-D, fix the number of prefixes $C = 30$, and producers $P = 4$. Then, we vary the number of core and edge routers link k from one to 10.

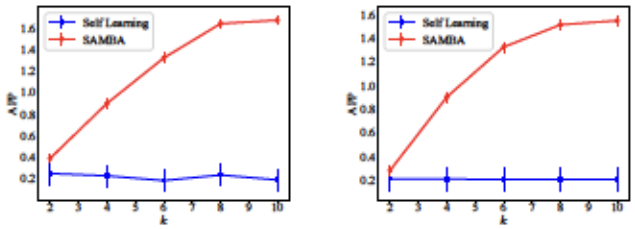
While Self Learning can detect only one path per discovery and each router just has a next hop per prefix, *SAMBA* can discover all existing paths. With a maximum of $k = 10$ for each router *SAMBA* can discover 1.67 APP in core routers (Figure 8a), while Self Learning can only discover on average almost 0.25 APP for all k values, also Self Learning can discover 1.53 APP in core and edge routers (Figure 8b), and Self Learning can discover 0.2 APP.

Toy Scenario with Link Failure In this experiment, We simulate link failure scenario with a simple topology shown in Figure 9a with two disjoint paths towards producer P1, and each link delay is 10ms. We run the simulation for 12 seconds and simulate an R3-R4 link failure at time 8 seconds using the $LinkDown$ function in ndnSIM. In this experiment, we implement consumer C1 requests prefix "`/P1`" from P1 and runs congestion control mechanism using AIMD [27] algorithm, also we set starting window size to one.To find



(a) Interest Discovery when C changes (b) Interest Discovery sent to the Network when P changes (c) Network overhead of multipaths when C changes

Fig. 7: Network overhead comparison of *SAMBA* and Self Learning.

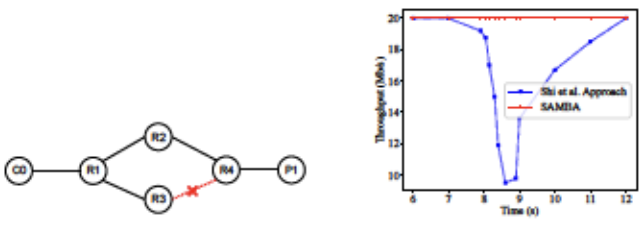


(a) Avg # discovered paths for 30 consumers in core routers (b) Avg # discovered paths for 30 name consumers in core and edge routers

Fig. 8: In this experiment, the level of redundancy increases by adding a degree of k in core and edge routers. While detected paths in Self Learning are almost the same, in *SAMBA*, it increases by elevating redundant links in core and edge routers.

failed links, we exploit BFD [28] protocol which is a periodic bidirectional way in link layer, with time interval $Int = 5ms$, and dead interval of 3.

Figure 9b shows the throughput measured at C1 node over time. We compare *SAMBA* with Shi's approach [12] that can discover multipath through different producers, while in Shi's approach consumer throughput decreases by up to 50% when the link fails. It took the consumer more than 1.5 seconds to re-discover a new path and recover its maximum throughput.



(a) C0 requests /P1 (b) Consumer throughput

Fig. 9: Link Failure scenario when R3-4 Link fails in second 8, *SAMBA* uses the alternate route immediately.

Consumer's congestion window got decreased and the delay to recover is a result of a slow start/congestion avoidance mechanism. Whereas, *SAMBA*'s consumer does not get impacted by the link failure and maintains a maximum throughout during simulation. *SAMBA* uses its multipath feature to set two disjoint paths (when available, *e.g.,* at R1) resulting in a seamless switching to the second path C0-R1-R2-R4-P1 as soon as R1 receives a `NOROUTE NACK` to divert traffic towards R2 instead of R3, and R1 sends `ALT_ROUTE` to C1, so the consumer will not activate the slow start state. Note that in case the second path fails, *SAMBA* will then initiate a route discovery and set new paths towards P1 (this scenario is not simulated in our experiment). In addition to the throughput gains, *SAMBA* will also reduce the number of discovery messages sent because of alternative paths while Shi's approach has exactly 195 discovery messages to recover from the link failures, which *SAMBA* avoided.

VI. Conclusion and Future Work

In NDN, many routing approaches face scalability issues due to the increasing number of FIB records affecting lookup times. To address this, we developed *SAMBA*, which AF and IPA to maintain small FIB tables by finding the nearest prefix for a given name. *SAMBA* also features multipath discovery for route redundancy and a Stop-and-Wait mechanism to minimize redundant discoveries. Evaluations show SAMBA achieves 20 times fewer FIB records and 5 times fewer flooding messages compared to Self Learning. Future work will focus on implementing *SAMBA* on a real testbed and enhancing its multipath management.

Acknowledgment

This material is based upon work supported by the National Science Foundation under grant #2148358 and is supported in part by funds from OUSD R&E, NIST, and industry partners as specified in the Resilient & Intelligent NextG Systems (RINGS) program.

References

[1] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, K. Claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, "Named data networking," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 66–73, 2014.

[2] A. Tariq, R. A. Rehman, and B.-S. Kim, "Forwarding strategies in ndn-based wireless networks: A survey," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 1, pp. 68–95, 2019.

[3] R. Shubbar and M. Ahmadi, "Efficient name matching based on a fast two-dimensional filter in named data networking," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 34, no. 2, pp. 203–221, 2019.

[4] O. Karrakchou, N. Samaan, and A. Karmouch, "Fctrees: A front-coded family of compressed tree-based fib structures for ndn routers," *IEEE Transactions on Network and Service Management*, vol. 17, no. 2, pp. 1167–1180, 2020.

[5] T. Song, H. Yuan, P. Crowley, and B. Zhang, "Scalable name-based packet forwarding: From millions to billions," in *Proceedings of the 2nd ACM conference on information-centric networking*, pp. 19–28, 2015.

[6] J. Hu and H. Li, "A composite structure for fast name prefix lookup," *Frontiers in ICT*, vol. 6, p. 15, 2019.

[7] T. Niu and F. Yang, "An entropy-based approach: Compressing names for ndn lookup," *IEEE Access*, vol. 9, pp. 109833–109846, 2021.

[8] Y. Zhang, Z. Xia, A. Afanasyev, and L. Zhang, "A note on routing scalability in named data networking," in *2019 IEEE International Conference on Communications Workshops (ICC Workshops)*, pp. 1–6, IEEE, 2019.

[9] J. Shi, E. Newberry, and B. Zhang, "On broadcast-based self-learning in named data networking," in *2017 IFIP Networking Conference (IFIP Networking) and Workshops*, pp. 1–9, IEEE, 2017.

[10] O. Karrakchou, N. Samaan, and A. Karmouch, "Endn: An enhanced ndn architecture with a p4-programmabie data plane," in *Proceedings of the 7th ACM Conference on Information-Centric Networking*, pp. 1–11, 2020.

[11] R. Miguel, S. Signorello, and F. M. Ramos, "Named data networking with programmable switches," in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pp. 400–405, IEEE, 2018.

[12] T. Liang, J. Pan, M. A. Rahman, J. Shi, D. Pesavento, A. Afanasyev, and B. Zhang, "Enabling named data networking forwarder to work out-of-the-box at edge networks," in *2020 IEEE International Conference on Communications Workshops (ICC Workshops)*, pp. 1–6, IEEE, 2020.

[13] M. Meisel, V. Pappas, and L. Zhang, "Listen first, broadcast later: Topology-agnostic forwarding under high dynamics," in *Annual conference of international technology alliance in network and information science*, p. 8, 2010.

[14] L. Wang, A. Hoque, C. Yi, A. Alyyan, and B. Zhang, "Ospfn: An ospf based routing protocol for named data networking," tech. rep., Technical Report NDN-0003, 2012.

[15] A. M. Hoque, S. O. Amin, A. Alyyan, B. Zhang, L. Zhang, and L. Wang, "Nlsr: Named-data link state routing protocol," in *Proceedings of the 3rd ACM SIGCOMM workshop on Information-centric networking*, pp. 15–20, 2013.

[16] K. Chan, B. Ko, S. Mastorakis, A. Afanasyev, and L. Zhang, "Fuzzy interest forwarding," in *Proceedings of the 13th Asian Internet Engineering Conference*, pp. 31–37, 2017.

[17] D. Mansour, H. Osman, and C. Tschudin, "Load balancing in the presence of services in named-data networking," *Journal of Network and Systems Management*, vol. 28, pp. 298–339, 2020.

[18] I. V. Brito, "Ndvr: Ndn distance vector routing," *arXiv preprint arXiv:2102.13584*, 2021.

[19] L. Wang, V. Lehman, A. M. Hoque, B. Zhang, Y. Yu, and L. Zhang, "A secure link state routing protocol for ndn," *IEEE Access*, vol. 6, pp. 10470–10482, 2018.

[20] E. Baccelli, C. Mehlis, O. Hahm, T. C. Schmidt, and M. Wählisch, "Information centric networking in the iot: Experiments with ndn in the wild," in *Proceedings of the 1st ACM conference on information-centric networking*, pp. 77–86, 2014.

[21] M. Liu, T. Song, Y. Yang, and B. Zhang, "A unified data structure of name lookup for ndn data plane," in *Proceedings of the 4th ACM Conference on Information-Centric Networking*, pp. 188–189, 2017.

[22] M. Al-Suwaiyel and E. Horowitz, "Algorithms for trie compaction," *ACM Transactions on Database Systems (TODS)*, vol. 9, no. 2, pp. 243–263, 1984.

[23] A. Afanasyev, J. Shi, B. Zhang, L. Zhang, I. Moiseenko, Y. Yu, W. Shang, Y. Huang, J. P. Abraham, S. DiBenedetto, *et al.*, "Nfd developer's guide," *Dept. Comput. Sci., Univ. California, Los Angeles, Los Angeles, CA, USA, Tech. Rep. NDN-0021*, vol. 29, p. 31, 2014.

[24] A. Afanasyev, I. Moiseenko, L. Zhang, *et al.*, "ndnsim: Ndn simulator for ns-3," *University of California, Los Angeles, Tech. Rep*, vol. 4, pp. 1–7, 2012.

[25] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena, "Network simulations with the ns-3 simulator," *SIGCOMM demonstration*, vol. 14, no. 14, p. 527, 2008.

[26] A. Esmaeili and M. Fazli, "Netscandn: A scalable and flexible testbed to evaluate ndn on multiple infrastructures," *arXiv preprint arXiv:2409.17128*, 2024.

[27] K. Schneider, C. Yi, B. Zhang, and L. Zhang, "A practical congestion control scheme for named data networking," in *Proceedings of the 3rd ACM Conference on Information-Centric Networking*, pp. 21–30, 2016.

[28] D. Katz, D. Ward, S. Pallagatti, and G. Mirsky, "Bidirectional forwarding detection (bfd) multipoint active tails," tech. rep., ietf, 2019.