NAMED DATA NETWORKING IN LOCAL AREA NETWORKS

by

Junxiao Shi

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2017

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Junxiao Shi
entitled Named Data Networking in Local Area Networks
and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

Beichuan Zhang — Date: 09 May 2017

Chris Gniady — Date: 09 May 2017

John Hartman — Date: 09 May 2017

Alon Efrat — Date: 09 May 2017

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.
I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Dissertation Director: Beichuan Zhang — Date: 09 May 2017

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: Junxiao Shi

ACKNOWLEDGEMENTS

First, I would like to sincerely and wholeheartedly thank my faculty advisor, Dr. Beichuan Zhang. He provided substantial technical guidance for my research and life advice for my future career.

Next gratitude goes to my dissertation committee members, Dr. Chris Gniady, Dr. John Hartman, and Dr. Alon Efrat. Their critical comments were helpful for improving this dissertation.

I am grateful to collaborators from the NDN project team, including Lixia Zhang, Lan Wang, Alexander Afanasyev, Davide Pesavento, Spyros Mastorakis, Yingdi Yu, and Michael Plass. I benefited from insightful design discussions with them.

I also would to like to thank my colleagues from the Network Research Lab, especially Teng Liang, Eric Newberry, Klaus Schneider, Mathias Gibbens, Shen Su, Ashiq Rahman, Chavoosh Ghasemi, Jerald Abraham, and Yifeng Li. They helped with fleshing out the designs, running the experiments, and reviewing the writing in my publications.

Many thanks to computer science lab staff Tom Lowry. He set up the system which allowed me to collect the NFS trace used in this dissertation.

Thanks also to the good friends I met during these years, including Debbie Schmidt, Jan Rydzak, Kayla Stack, Chris Sanchez, Gaurav Singh, Morris Zhou from the International Student Association, Youxi Li, Ashley Garcia, Ej Matchin from NorthPointe Apartments, and @TheTucsonHeat, @ElTucsonMonsoon, @AZFilmFest on Twitter. They made my PhD experience enjoyable and memorable.

Finally, I owe my utmost gratitude to my family who have given me their unconditional love and support throughout my whole life. Their trust and understanding helped me overcome many obstacles during the PhD process. None of my accomplishments would have been possible without them.

TABLE OF CONTENTS

TABLE OF CONTENTS – *Continued*

TABLE OF CONTENTS – *Continued*

LIST OF FIGURES

LIST OF FIGURES – *Continued*

LIST OF TABLES

ABSTRACT

The Named Data Networking (NDN) is a new Internet architecture that changes the network semantic from *packet delivery* to *content retrieval* and promises benefits in areas such as content distribution, security, mobility support, and application development. While the basic NDN architecture applies to any network environment, local area networks (LANs) are of particular interest because of their prevalence on the Internet and the relatively low barrier to deployment.

In this dissertation, I design NDN protocols and implement NDN software, to make NDN communication in LAN robust and efficient. My contributions include: (a) a forwarding behavior specification required on every NDN node; (b) a secure and efficient self-learning strategy for switched Ethernet, which discovers available contents via occasional flooding, so that the network can operate without manual configuration, and does not require a routing protocol or a centralized controller; (c) NDN-NIC, a network interface card that performs name-based packet filtering, to reduce CPU overhead and power consumption of the main system during broadcast communication on shared media; (d) the NDN Link Protocol (NDNLP), which allows the forwarding plane to add hop-by-hop headers, and provides a fragmentation-reassembly feature so that large NDN packets can be sent directly over Ethernet with limited MTU.

CHAPTER 1

INTRODUCTION

Named Data Networking (NDN) [1, 2] is data-centric communication architecture. In NDN, data is treated as a first-class entity: a file, a voice call, and a command to turn on the light, they are all data. NDN communication is centered on how to retrieve data. Compared to today's Internet Protocol (IP) architecture, NDN shifts the network service semantics from "delivering packet to given address" to "retrieving data of given name". This new model better fits the dominate communication pattern of today's Internet: content distribution and retrieval.

Local area networks (LANs), such as homes, offices, and enterprises, are predominant on the Internet, as they are the "first mile" of most communications. LAN communication is a long solved problem in IP (Section 2.1). It is also of particular interest to NDN because the relatively low barrier to deployment. If running NDN is easy and beneficial to applications, NDN can be deployed in LANs with no external coordination and much less effort compared to wide-area Internet. In the long run, as more and more LANs are NDN-enabled, the deployment may grow from the network edges towards the core, bringing more benefits to applications.

However, early NDN design and implementation [3] have been focused on wide area networks (WANs), and were not optimized for LAN communication. To retrieve data published by a server in the local area network, forwarding paths must be configured statically: either the consumers need to know the IP address of the server and connect to it, or every node must connect to a gateway router, forming a star topology. In both cases, NDN packets are unnecessarily encapsulated into UDP datagrams. Data are unicast to each consumer individually, and cannot take advantage of the multicast/broadcast capability of the underlying network.

This dissertation aims at addressing the above limitations and providing a complete solution to NDN deployments in LANs. Compared to previous work, the design

and implementation described in this dissertation allow NDN to: (a) discover available contents and forwarding paths through broadcast-based self-learning, without requiring the configuration of static routes or the overhead of a routing protocol; (b) consume less CPU and energy by filtering packets by their names in network interface card hardware, instead of processing every packet in the main CPU; (c) work natively in Ethernet environment, without relying on TCP/UDP tunnels for local communication.

First, I designed **NDN self-learning** (Chapter 4), a flood-and-learn forwarding procedure. NDN self-learning discovers available contents and forwarding paths in wired, switched Ethernet environments via occasional flooding, and does not need static configuration or routing protocols. In NDN self-learning, a node floods the first Interest across the network; if a Data packet returns, the node remembers the forwarding path, so that future Interests will only need unicast.

Although previous work [4] has proposed a similar flood-and-learning procedures, they overlooked the prefix granularity problem, i.e. what name prefix should be associated with a learned forwarding path. I explored various design options, and chose to attach a "prefix announcement" onto Data packets. Emulation with a file access traffic trace showed that having accurate prefix granularity significantly reduces Interest flooding.

Additionally, I proposed a method to reject malicious prefix announcements using NDN's data-centric security to reject malicious prefix announcements. The forwarding protocol was enhanced with NDN's stateful forwarding plane [5] to monitor the reachability of a learned forwarding path, so it can quickly switch to an alternate path if the current path stops working. I also designed an Interest diversion scheme to make use of off-path caching; this design was shown to reduce bandwidth on the WAN connection by up to 12.9%.

Second, I designed **NDN-NIC** (Chapter 5), a network interface card that can reduce CPU processing overhead in broadcast environments. On single-hop shared media such as traditional Ethernet (connected via repeaters) and most wireless net-

works, everyone can hear all signals transmitted within range. A node then accepts relevant packets, and discards those of no interest. Traditionally, the network interface card (NIC) performs packet filtering: it checks the destination MAC address of each packet, and admits the packet if the address is the host's own address. This hardware-based packet filtering can save CPU cycles and power consumption of the main system.

NDN can take advantage of the broadcast nature of shared medium: a consumer can broadcast an Interest without specifying a layer-2 destination address, and then anyone who has the Data can reply. This is particularly useful in wireless mobile networks, so that a node does not need to keep track of the addresses of its neighbors. However, implementing NDN broadcast in current systems may cause significant CPU overhead and power consumption, because the NIC is unable to filter NDN packets, and all NDN packets must be delivered to the main CPU for processing.

NDN-NIC addresses this problem by adding name-based packet filtering to the NIC hardware. The main challenge of NDN-NIC is to put hundreds of thousands of names from NFD's FIB, PIT, and CS tables into tens of kilobytes of memory available on the NIC. I used Bloom filters (BFs) to represent the names, and proposed two optimizations to reduce BF false positives. Simulation showed that NDN-NIC could reduce CPU usage by 93.96% compared to a regular NIC.

Third, I defined the **NDN Link Protocol (NDNLP)** (Chapter 6), a link adaptation protocol that enables NDN to become a universal network layer protocol for all applications and network environments [6]. NDN was envisioned to work on all kinds of underlying transports, including direct hardware connections as well as overlay tunnels over the existing IP network. However, early NDN implementation [3] only supported communication over UDP or TCP tunnels. A challenge of supporting more underlying transports is that, different transports provide different services: (a) TCP and WebSockets are reliable, while Ethernet and UDP can have packet loss; (b) Ethernet and Bluetooth limit packet size to Maximum Transmission Unit (MTU), while TCP and WebSockets have no practical packet size limit.

NDNLP hides the difference among underlying transports and provide a uniform best-effort delivery service to the NDN forwarding plane. This delivery service, known as the *face*, is then used by the forwarding plane to implement data retrieval service.

NDNLP adopts an extensible packet format based on Type-Length-Value structure. This packet format allows the forwarding plane to add auxiliary information, such as NDN self-learning's prefix announcements, as hop-by-hop headers onto Interest/Data packets. NDNLP also provides a fragmentation-reassembly feature that enables NDN to operate on network links with limited MTU.

Apart from a solution for deploying NDN in LAN, another contribution of this dissertation is a specification of **NDN forwarding behavior** (Chapter 3). Although the NDN team has published an NDN packet format specification [7], the NDN network layer protocol has been defined in terms of the behavior exhibited by the `ccnd` software [3], and there was no formal definition on the expected behavior of an NDN node's forwarding plane.

I summarize previous work about NDN architecture into data model, communication model, node model, and security model (Section 2.2). Along the way, I define the semantics of Interest, Data, and Nack, the three network layer packet types used in NDN forwarding. Then, I specify how an NDN node should process each network layer packet (Chapter 3), and explain how these procedures can prevent Interest loops. The forwarding procedures in the NDN Forwarding Daemon (NFD) (Section 2.3) has been designed according to my forwarding behavior specification.

CHAPTER 2

BACKGROUND

This chapter first briefly reviews how local area network (LAN) communication works in existing Ethernet/IP architecture. Then, I summarize the overall NDN architecture. Finally, I introduce a software implementation of NDN forwarder called the NDN Forwarding Daemon (NFD).

2.1 Ethernet/IP-based LAN Communication

Communication in local area networks, such as homes, offices, and enterprises, is a long solved problem under the IP architecture. As a concrete example, retrieving a file in an office network via Ethernet/IP involves three stages:

1. The client figures out the address of the server that has this file.
2. The client sends packets addressed to the server. The network delivers packets to the destination.
3. Network interfaces perform packet filtering: the intended destination host accepts and processes the packet, while other hosts discard the packet.

The next sections describe how each stage works, and point out the limitations of Ethernet/IP architecture in supporting content retrieval in LANs.

2.1.1 Mapping Application Identities to Host Addresses

Ethernet/IP provides a service to deliver packets to a given destination address. To retrieve a file using this service, the client must figure out the address of the server that has this file.

Starting with the file name, the client application must find out the host name of the server. There are several ways to obtain this information. Network File System (NFS) [8] relies on a **static configuration** where host names of NFS servers are pre-configured on the client. Hadoop Distributed File System (HDFS) [9] uses a **centralized resolution service**: the NameNode remembers where each file chunk is stored; clients can ask the NameNode to learn the host names of requested files. Dropbox LAN sync protocol [10] periodically **broadcasts announcements** about folders available from a Dropbox client, and other Dropbox clients on the LAN can listen for such announcements to learn the peer's addresses.

Then, the sender translates the host name into an IP address via Domain Name System (DNS) [11], and maps the IP address into a physical address via Address Resolution Protocol (ARP) [12].

2.1.2 Packet Forwarding and Loop Prevention

Ethernet employs a flood-and-learn procedure to forward packets to their destinations. If an Ethernet switch does not know the location of a packet's destination address, it floods the packet. Afterward, it remembers a mapping between the source address and the switch port, allowing subsequent packets to that address to be sent unicast.

Flooding a packet in Ethernet may cause bridge loops if the topology contains cycles. To prevent bridge loops, Ethernet employs the Fast Spanning Tree Protocol (FSTP) [13], which trims the topology to a spanning tree by disabling particular links. The remaining topology does not have any cycles, and therefore loop cannot happen. However, this severely reduces the available bandwidth in the network, causes traffic between some source-destination pairs to take a longer path, and negatively affects the performance of Ethernet.

To address the above limitation, larger enterprise LANs are often divided into multiple Ethernet segments, connected together via "L3 switches" or IP routers. Ethernet's flood-and-learn forwarding is used within each segments, while IP handles packet forwarding across segments. IP forwarding plane follows the forwarding paths

in the routing table, which is populated by a routing protocol. The routing protocol is required to be loop-free: the routing algorithm must assure that routing loops can never happen, not even transiently. This ensures there is no packet loops in IP forwarding.

On the other hand, routing protocols themselves often need to flood routing announcements, and the flooding mechanism must ensure those packets do not loop. Open Shortest Path First (OSPF), a commonly used routing protocol, prevents looping of routing announcements (called "LSAs") using locally maintained states [14]. Every LSA carries a sequence number assigned incrementally by the advertising router, and an identifier of the advertising router. When an LSA arrives, the router compares the LSA's sequence number with the last seen sequence number from the same advertising router in a local data structure called the "link state database", and processes the LSA only if the LSA's sequence number is newer. Then, the new sequence number is remembered in the link state database, so that future LSAs with same or older sequence numbers would be rejected.

2.1.3 Multicast

IP natively supports multicast, which can be used to send packets to multiple recipients simultaneously. However, setting up IP multicast to distribute data is a complicated process. First, the sender picks a multicast group address; preferably, the multicast group should not be in use by another data distribution so that it would not cause other nodes to receive irrelevant packets. Second, the sender must tell recipients the multicast group address through an out-of-band mechanism (such as unicast), and then the recipients join the group using Internet Group Management Protocol (IGMP) [15]. Third, the network switches setup a multicast distribution tree to deliver packets to correct recipients, typically via IGMP snooping [16].

IP multicast is not only difficult to use, but also have a few other limitations. First, IP multicast is only capable of real-time distribution; a recipient who joins late would not be able to receive previously delivered data. Second, there is no mechanism to recover from a packet loss in regular IP multicast. Third, today's IP

multicast deployment is limited to "islands" of network domains due to the difficulty in wide-area multicast routing [17]; consequently, data distribution from an Internet origin into a LAN over IP multicast does not work, and the sender has to resort to unicasting.

2.1.4 Address-based Filtering on NICs

The last hop of Ethernet may contain multiple end hosts. This was the case in traditional Ethernet (connected via repeaters or "HUBs"), and is still the case in today's wireless networks. Each end host performs packet filtering, accepts packets of interest, and discards the rest. In current systems, packet filtering is implemented on Ethernet network interface cards (NICs), to save CPU cycles and power consumption of the main system.

A conventional Ethernet NIC has an address-based filtering logic that matches the destination MAC address of an incoming packet against a list of acceptable unicast and multicast addresses. The operating system controls this list of addresses, which is populated with the host's own unicast address as well as any multicast group the host has joined.

Most NICs implement the list as an array of addresses and conduct exact match: a packet is delivered to the system if its destination address equals to any address in the array. The size of the address list varies among different NICs, e.g., according to driver implementations in Linux kernel 4.4, Broadcom BCM4401-B0 can store up to 32 addresses, while Intel PRO/Wireless 2200BG has space for only one address.

Some NICs also use a hash table to accommodate more addresses (for instance, Intel 82540EM has a 4096-bit hash table). Each acceptable address is mapped to a bit in the hash table, and a packet passes the filter when the hash value of its destination address matches a bit set to 1. False positives can occur in the hash table due to hash collisions.

Moreover, almost every NIC supports "ALLMULTI" mode, which causes the NIC to accept all Ethernet multicast packets by simply checking a bit in the destination address that identifies a multicast address.

2.2 Named Data Networking (NDN) Architecture

Named Data Networking (NDN) is a data-centric communication architecture. This section defines the concept of "named data", explains how data is retrieved from the network, defines the data structures in network nodes, and discusses how to secure and trust data.

2.2.1 Data Model: What is Data?

NDN treats *data* as a first-class entity. A file contains file chunk data. A phone call consists of voice data. A temperature reading from a sensor, a command to turn on the light, a traffic information report from a vehicle . . . they are all data.

This section introduces the concept of named data, and defines what is in a Data packet [1].

Data Naming

Data are *named*. By naming data, NDN enables applications to retrieve their desired data by names. Each piece of data is identified by a **name**, indicating what the content is. Each name consists of a sequence of **name component**s. For example, a piece of data carrying Arizona Computer Science homepage may be named `/edu/arizona/cs/www/index.html/v20170630/segment0`. There are seven name components in this name.

Names are *hierarchical*. Having hierarchical names facilitates name aggregation and provide application context. In the above example, starting the name with the prefix `/edu/arizona` indicates this piece of data comes from The University of Arizona. While NDN could work with non-hierarchical flat names, they are extremely inefficient because flat names cannot be aggregated in the data structures used in NDN nodes.

[1] This dissertation uses upper-case "Data" to refer to NDN's Data packet, and uses lower-case "data" to refer to the general concept of information.

Data are *immutable* [6]. Each piece of Data is uniquely named, and cannot be modified once created. This immutability allows disambiguation of coordination in a distributed system that may not be always connected. Consequently, names should be *expressive* enough in order to uniquely identify a piece of data.

Although Data packets are immutable, application can make changes to the communicated content by creating new versions of immutable Data packets. In the above example, the second to last name component `v20170630` is a version number [2]. Whenever Arizona Computer Science modifies its website homepage, the webmaster would generate a new version number and create a new name for the new Data.

Large documents are divided into small segments, each segment in a Data packet [19]. As I will show in Section 2.2.2, each Data packet can be individually retrieved. Data segmentation enables more efficient communication, because: (a) segments of a large document can be retrieved in parallel; (b) packet loss only affects some segments, not the entire document. In the above example, the last component `segment0` is a segment number [2]. It is worth noting that having version number and segment number at the end of a name is a *convention* [18] rather than a protocol requirement. NDN can still operate correctly when an application adopts a different naming scheme for its data.

Theoretically, there is no limitation on the number of name components in a name, or the length of each name component. Therefore, there are infinitely many potential NDN names. A total order called the **canonical order** is defined over the set of NDN names [7]. In the canonical order, under the same prefix, a name with a newer version number is greater than a name with an older version number; under the same version number, a name with a smaller segment number is less than a name with a greater segment number. This canonical order is used in multiple contexts including in-network caching (Section 2.2.3).

[2] Version number and segment number are encoded in a binary format, defined in [18]. A text format is used for presentation purpose.

The Data Packet

A **Data** packet contains the following fields [7]:

- the name
- meta information, such as the number of segments in a segmented document
- the content payload
- a cryptographic signature (Section 2.2.4)

2.2.2 Communication Model: How to Retrieve Data?

NDN communication is receiver-driven. The data receiver in NDN is called a **consumer**. The consumer requests the network to retrieve a Data by expressing an Interest, which is a request for the retrieval of a piece of data. The network is then responsible for locating a Data that can satisfy the Interest, and return it back to the consumer.

The Interest Packet

An **Interest** packet [7] contains the following fields:

- a name
- selectors (see below)
- a nonce (a random number) for loop detection purpose (Section 3.3)
- an InterestLifetime to indicate how long the Interest remains valid

There is no source address or destination address in the Interest packet.

The Interest **name** indicates what data is being requested. It can either be the exact name of the Data, or a prefix of the Data name. Being able to retrieve Data with incomplete names (i.e. prefixes) is crucial [6]. Recall that Data packets are immutable, and a producer must create new versions to make changes to communicated content. If a consumer wants to retrieve the latest version using exact name, it must first know the latest version number, which is infeasible in many environments. Therefore, the consumer should be able to express an Interest with an incomplete name, and *discover* the complete names from the network.

Selectors place additional constraints on what Data can satisfy the Interest. Those constraints include the length of Data name, the key which signed the Data, etc. Semantics of those selectors are defined in [7].

Data, Nack, or Timeout

Once the consumer expresses an Interest, the network attempts to retrieve a Data that can satisfy the Interest. The deadline of data retrieval is given in the InterestLifetime field. If the network successfully retrieves a Data that satisfies the Interest within the deadline, it would be returned to the consumer.

As a special case, the Data could be an attestation by a data producer that the requested content does not exist. This special Data must be signed by the data producer, and from network point of view it satisfies the Interest just like a regular Data. In case the requested content would be generated in the future, the packet payload may inform the consumer to re-express the Interest after waiting a certain period of time.

Each Interest packet brings back no more than one Data packet. If a consumer wants to retrieve a large collection of data requiring multiple Data packets, it needs to transmit multiple Interests. The consumer may transmit multiple Interests without waiting for Data in response to each one before sending the next, which achieves pipelining. However, this is only possible if the consumer knows the names of each Data so that it can construct different Interests in advance.

If the network is unable to locate a Data to satisfy the Interest, the consumer may be informed by a **Nack** (Negative ACKnowledgement) packet. The Nack packet carries the original Interest, and a reason code to indicate why the Interest cannot be satisfied. Possible reasons are “Duplicate”, “NoRoute” (including link failure), etc.

An Interest times out when InterestLifetime has elapsed but neither Data nor Nack comes back. This can happen, for example, when a packet is lost in the underlying transport or due to node failure.

If the consumer receives a Nack or experiences an Interest timeout, it may re-

transmit the Interest to ask the network to try again. The network would usually attempt to retrieve the data via alternate paths. The consumer should limit the number of retransmissions, increase the interval between retransmissions (exponential back-off), and eventually give up if the data is still unavailable after several attempts.

2.2.3 Node Model: How are Packets Processed in a Node?

The NDN network is a distributed system consisting of interconnected nodes. A node is any router, switch, or end host with NDN network stack. The network nodes collectively provide data retrieval service to applications.

This section first defines three data structures (or *tables*) in an NDN node, then gives an overview on how NDN forwarding plane works.

CS: In-Network Caching

Every NDN node can cache Data packets passing through the node. As a major benefit of NDN, in-network caching makes NDN more efficient: (a) Popular Data packets are more likely to be cached in a node near the consumer, so that they can be retrieved faster. (b) If a Data packet was lost in transit, a retransmitted Interest from the downstream can be satisfied by cached Data, instead of going all the way to the data producer.

The cache of Data packets is called the Content Store (CS). When an Interest arrives, the node queries the CS to see if there is a cached Data that can satisfy the Interest. Given an Interest, the CS is expected to return a Data that *satisfy* the Interest, i.e. the Data name must start with the Interest name, and the Data must not violate any selectors in the Interest. If the CS finds a Data that satisfies the Interest, the Data is returned to the downstream, and the Interest is not forwarded further. This saves upstream bandwidth, and shortens data retrieval delay. In case there are multiple Data that can satisfy the Interest, the CS selects the best match according to the ChildSelector in the Interest. The ChildSelector could be either

"leftmost", selecting the Data with smallest name (according to the canonical order, Section 2.2.1), or "rightmost", selecting the Data with greatest name.

The CS is an opportunistic cache with limited cache capacity. It should implement a cache replacement policy to regulate the cache size. When the cache is full, the policy may either evict existing Data to make room for new Data or stop admitting new Data. Most common cache replacement polices include First In First Out (FIFO), Least Recently Used (LRU), and Least Frequently Used (LFU). There are other policies that take into account the relation between Data packets and admit/evict all segments of an object together [20], or jointly make forwarding and caching decisions [21].

FIB: Guide Interest Forwarding

Interests not satisfied by the cache need to be forwarded. The Forwarding Information Base (FIB) is a table of Interest forwarding paths, which allows the node to determine where an Interest should be forwarded to retrieve the data.

FIB is organized as a name-based lookup table. Each FIB entry is identified by a name prefix, and have an ordered list of next hops. To forward an Interest, the node performs a **longest prefix match** lookup to find a FIB entry. The longest prefix match is performed name component by name component (not byte or bit). For example, Interest `/AA/BB` matches FIB entry `/AA` but not `/A`.

NDN forwarding plane generally does not modify the FIB, but requires the FIB to be populated already. Broadcast-based self-learning (Chapter 4) is one way to populate the FIB in local area networks. Control plane mechanisms such routing protocols [22] and management commands [23] can also create FIB entries.

FIB has a similar role as IP's routing table, but there are two differences: (a) FIB is indexed by name prefixes, while IP routing table is indexed by addresses. (b) Each IP routing entry usually has only one next hop, while each FIB entry can have multiple next hops. The forwarding procedures can choose between those next hops.

PIT: Get Data Back

After an Interest is forwarded, when the Data comes back, it must be returned to the consumer. In Information-Centric Networking (ICN), there are two methods to identify the return path of Data: (a) The network can create routing entries to consumers; for example, Data-Oriented Network Architecture (DONA) [24] sends Data to consumer via IP forwarding. (b) Or, the network can have states leading to the consumer.

NDN has adopted the second approach, maintaining states leading to the consumer. Return path of pending Interests are kept in the Pending Interest Table (PIT). When a node forwards an Interest, it inserts a PIT entry for the Interest, as well as the downstream node where the Interest came from. As an Interest is forwarded through the network, it leaves a PIT entry on every traversed node. When the Data comes back, each node queries the PIT to find what Interest(s) the Data can satisfy, sends the Data to the downstream nodes recorded in the PIT entries, and deletes satisfied PIT entries. This enables the Data to trace back to the consumer without requiring a return address in the packet.

Aside from establishing the return path of a Data, the PIT is also helpful in **Interest aggregation**. When a node receives an Interest requesting the same data as an existing pending Interest, instead of forwarding this Interest again, the node remembers the downstream node of this Interest in the existing PIT entry. Therefore, only one copy of the Interest is sent upstream, even if there are multiple downstream nodes wanting to retrieve the same data. When the Data comes back, both downstream nodes can receive it. This saves upstream bandwidth, and makes NDN scalable.

Interest aggregation is related to but different from answering the Interest with cached Data: although both mechanisms improve scalability and reduce bandwidth usage for popular content, Interest aggregation is effective when the Interest is still pending and before the node receives Data, while cached Data is useful after the Data has arrived.

To summarize, a PIT entry contains the following fields:

- Interest name and selectors, as the identifier of the PIT entry, used in Interest aggregation;
- records about downstream nodes, used for getting Data back;
- records about upstream nodes, to facilitate Nack processing (Section 3.2.3);
- a timer to delete the PIT entry when it expires after InterestLifetime.

Forwarding Procedures

When NDN architecture started, the packet processing procedures were defined in terms of the behavior exhibited by `ccnd` software [3], and there was no formal protocol specification. In `ccnd`, packet forwarding is hop-by-hop. When a node receives an Interest packet, the node may satisfy the Interest using a cached Data from the CS, or forward the Interest to the upstream in the FIB after confirming the Interest is not a duplicate. If multiple downstream nodes have sent the same Interest, the node only forwards one copy of the Interest and aggregate others in the PIT. When a Data packet comes back, a node can return the Data to downstream as recorded in the PIT, and add the Data to the CS. Different from IP where each packet has a fixed destination, NDN Interests can be forwarded to any node where the Data may be available, and Data can come from anywhere.

Cheng [5] extended the forwarding plane to be adaptive and stateful, and introduced the Nack packet type. One of the benefits brought by the PIT is that it makes Data take the reverse path of Interests, so that every node can observe whether and how soon the Data comes back after forwarding an Interest. If a node detects a reachability problem, it can forward the Interest onto an alternate path, or return a Nack. Upon receipt of a Nack, the downstream should take corrective actions.

As a contribution of this dissertation, NDN forwarding procedures are further developed in Chapter 3 and implemented in NFD (Section 2.3).

2.2.4 Security Model: How to Secure and Trust Data?

NDN secures data directly. Security is a property of data packets, staying the same whether the packets are in motion or at rest [6]. Securing data directly, as opposed to securing the communication channel like Transport Layer Security (TLS) [25] does, removes the requirement for direct channels between communication ends, and enables asynchronous production and consumption of named data. Applications running over NDN networks can utilize in-network storage, including caches and repositories, to achieve performance and scalability enhancements [26].

Every Data packet must be signed by a key that binds the data content to its name. The **signature** field contains a "key locator" that references the data producer's public key, as well as the bits generated by the signing algorithm. A consumer can retrieve the public key, and verify that the data originated from the data producer holding this key, regardless of from where the data was retrieved.

Signature verification only ensures the data originated from the producer, but does not answer the question about whether the producer is authorized to produce the data. To answer this question, the consumer must have an application-specific trust model that determines whether the key is authorized to sign the data. A common method of defining a trust model is through a **trust schema** [27]. A trust schema contains a set of name-binding rules that dictate which keys are authorized to sign each Data or sub-certificate, as well as a set of top-level certificates as trust anchors. With the trust schema, anyone can verify received Data packets against the trust model in an automated and consistent way. On the other hand, the trust schema can also be used on producer end for selecting an appropriate key to sign a specific Data packet.

2.3 NDN Forwarding Daemon (NFD)

NFD is a network forwarder that implements and evolves together with the NDN protocol [28]. The main design goal of NFD is to support diverse experimentation with NDN architecture. The design emphasizes modularity and extensibility to

allow easy experiments with new protocol features, algorithms, and applications.

I contributed to the architectural design of NFD, and in particular, its face system (Section 6.5) and forwarding plane (Chapter 3). NFD is also the primary platform where I experimented with the ideas and designs in this dissertation.

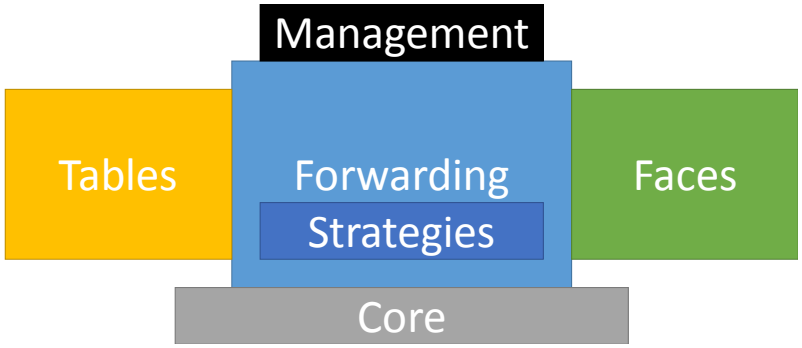


Figure 2.1: NFD architecture

The main functionality of NFD is to forward Interest, Data, and Nack packets among local applications and network links for the purpose of content retrieval. To do this, it abstracts lower-level network transport mechanisms into **faces**, maintains basic data structures as **tables**, and implements the packet forwarding logic in **forwarding**. In addition, it also exposes a **management** interface to configure, control, and monitor NFD.

As illustrated in Figure 2.1, NFD contains the following modules:

- **Core** provides various common services shared between different NFD modules. These include logging facility, hash computation routines, configuration file parser, and several other features.

- **Faces** implement a face abstraction on top of various lower level transport mechanisms. NFD can communicate with local application via Unix sockets, and communicate with remote nodes over Ethernet, UDP, TCP, and WebSockets. NDNLP (Chapter 6) is used to provide fragmentation-reassembly and other link adaptation services.

- **Tables** implement CS, FIB, PIT, and other data structures to support the forwarding plane (Section 2.2.3).
- **Forwarding** implements the forwarding plane as a set of packet processing pipelines. It also contains a framework to support different forwarding strategies (Section 3.1.3).
- **Management** implements the NFD Management Protocol [23], which works at application layer and allows utility programs to configure and control NFD.

NFD is widely used by the NDN community. NFD's mailing list, `nfd-dev`, was setup to facilitate exchanges among NFD users, and as of July 2017 it has 155 subscribers from multiple countries and has been generating a great amount of traffic. In addition to bug reports and feature requests, there are 64 public forks on GitHub, and 35 developers have contributed code to NFD.

NFD is deployed on the global NDN testbed, which has grown to 37 nodes and 100 links. It is also an essential part of the ndnSIM simulator [29] and the Mini-NDN emulator [30].

CHAPTER 3

NDN FORWARDING BEHAVIOR

Correct and efficient operation of Named Data Networking (NDN) requires a clearly defined forwarding plane. CCNx [3], the early prototype of NDN architecture, embedded many aspects of its forwarding behavior in the source code. The lack of a forwarding behavior specification hinders the adoption of NDN.

In this chapter, I specify the expected forwarding behavior of an NDN node, as a first step toward the standardization of the NDN architecture. I start with introducing the data structures and components of a node (Section 3.1). Then, I specify the procedures of processing Interest, Data, and Nack packets (Section 3.2). Finally, I zoom in on how the forwarding procedures can prevent persistent loops in the network (Section 3.3).

3.1 Components of a Node

In this section, I introduce the basic components of an NDN node.

3.1.1 Tables

Each node has three basic data structures (or tables): Content Store (CS), Forwarding Information Base (FIB), and Pending Interest Table (PIT). Definition and semantics of these tables have appeared in Section 2.2.3. Additionally, I proposed a fourth "dead nonce list" table in Section 3.3.4 to assist in loop prevention.

3.1.2 Face

Face is a generalization of the concept of *interface*. Incoming packets enter a node via faces, and get processed by the procedure in Section 3.2. Outgoing packets leave a node via faces, and are delivered to the next hop.

A face provides a best-effort delivery service over an underlying transport. A wide variety of underlying transports are supported by face implementations, including inter-process communication channels to applications on the same node, direct Ethernet connectivity with nodes in the local network, and UDP overlay tunnel connections to remote nodes. Faces internally handle the differences among underlying transports, and expose a ubiquitous interface to the forwarding plane.

It is worth noting that NDN forwarding treats connections to local application as faces. Therefore, discussion about "downstream" is applicable to local consumer applications; likewise, discussion about "upstream" or "next hop" is applicable to local producer applications.

More details about face and its implementations are discussed in Chapter 6.

3.1.3 Forwarding Strategy

The forwarding strategy is a component that controls various aspects of Interest forwarding behavior. The strategy decides where to forward an incoming Interest, if it cannot be satisfied by the local cache. It also decides on how to perform corrective actions when a Nack arrives. These decisions are based on inputs such as the next hops in the FIB entry, downstream and upstream records in the PIT entry, as well as collected *measurements* about recent data retrievals by Interests sharing a common prefix. The latter, measurements about recent retrievals, are collected by the strategy by observing the outcome of forwarded Interests to each next hop, including: successful retrieval ratio, traffic volume, and round trip time. These measurements are stored internally in data structures maintained by the forwarding strategy.

Some form of forwarding strategy exists in every NDN node. It could be a separate component, or be integrated in the overall forwarding plane. Examples of forwarding strategies include multicast, "best route" [28], broadcast-based self-learning (Chapter 4), and various designs in publications such as [5] and [31].

Experience with NDN applications has shown that different applications need different Interest forwarding behaviors. For example, a file retrieval application desires

forwarding paths with higher throughput, an audio chat application prefers paths with minimal delay, and a dataset synchronization library (such as ChronoSync [32]) needs to multicast Interests to all available neighbor nodes to reach its peers. This motivates to implement multiple forwarding strategies with different decision making algorithms, and dynamically choose a forwarding strategy based on application needs and network environments.

If a node implements multiple forwarding strategies, the effective forwarding strategy for an Interest can be indicated in the FIB entry. There are also other methods of choosing a forwarding strategy, but they are out of scope of this dissertation.

3.2 Packet Processing

In this section, I define how an NDN node should process Interests, Data, and Nacks, using the data structures described in Section 3.1.

Compared to CCNx, the forwarding behavior defined in this section incorporates the adaptive forwarding plane concept from [5], permits Interest retransmissions, and allows nodes to send and process Nack packets.

3.2.1 Interest Processing Steps

When a node receives an Interest from the downstream, it becomes responsible for retrieving the desired data and returning it to the downstream. The Data can either come from the local cache, or retrieved from an upstream node (or local application) via a face. Additionally, the node must prevent persistent Interest loops.

To fulfill these goals, the node processes the Interest according to the workflow in Figure 3.1, which takes one of the following five actions:

Return a cached Data. The node queries the CS with the incoming Interest. If the CS matching algorithm returns a Data that can satisfy the Interest, the node sends the Data back to the downstream, and the Interest is not forwarded. In this

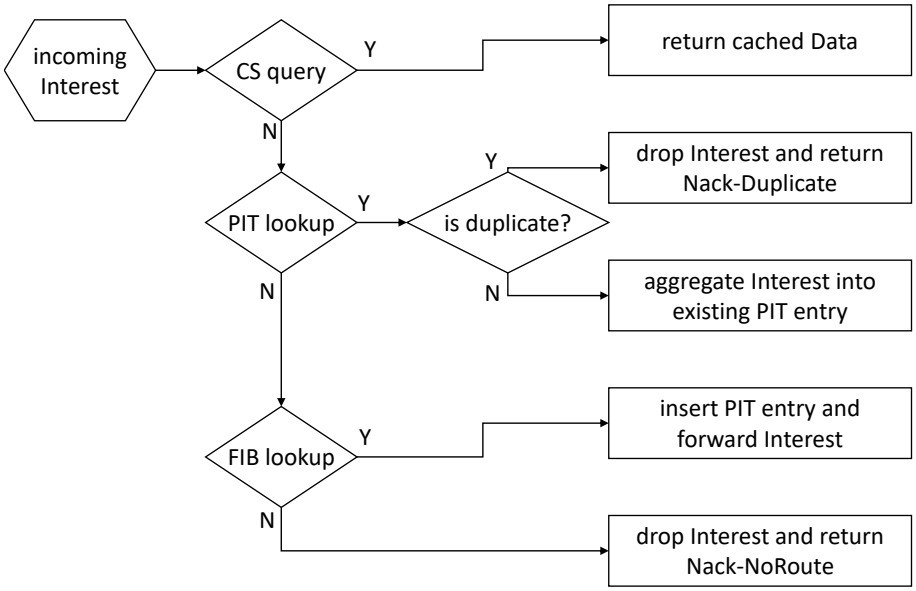


Figure 3.1: Interest processing steps

case, no PIT entry is created, because the Interest has been satisfied by the cached Data and is no longer "pending".

Drop the Interest if it is a duplicate. An incoming Interest is considered a duplicate if it has the same nonce as a previous Interest that was received from a different downstream. If the Interest is detected as a duplicate, the node informs the downstream to take corrective actions by returning a Nack with reason "Duplicate". This is part of the loop prevention mechanism, discussed in Section 3.3.2.

Aggregate the Interest. The node looks up the PIT to see whether an Interest with same name and selectors has been forwarded recently and is still pending. If so, the current Interest is aggregated into the existing PIT entry and not forwarded again. The downstream of the current Interest is recorded in the PIT entry, so that when Data comes back, a copy of it would be sent to this downstream.

Note that even if the Interest is still pending, if more than retransmission timeout (RTO) has elapsed after the Interest was last forwarded, the forwarding strategy may decide to forward the Interest rather than aggregating it. In this case, the previous

Interest or Data is probably lost, and retransmitting the Interest could increase the chance of a successful data retrieval.

Forward the Interest. The node looks up the FIB with the Interest name. A forwarding strategy decides which next hop(s) to use, based on the FIB entry and other information (Section 3.1.3).

The node then sends the Interest to each next hop chosen by the strategy, inserts a PIT entry (if it does not already exist) to remember the forwarded Interest, and records the downstream in the PIT entry. A timer is set to delete the PIT entry after InterestLifetime has elapsed.

Drop the Interest if it cannot be forwarded. If the forwarding strategy determines that the Interest cannot be forwarded to any next hop (e.g. when all next hops in the FIB match have link failures), the Interest is dropped, and the node informs the downstream to take corrective actions by returning a Nack with reason "NoRoute" (Section 3.2.3).

3.2.2 Data Processing Steps

When a node receives a Data packet, it returns the Data to downstream nodes that have requested this Data, and adds the Data to the local cache. Figure 3.2 outlines the workflow of processing a Data packet.

The node first queries the PIT to find all pending Interests that can be satisfied by the Data. If one or more PIT entries are found, the Data is added to the CS, a copy of the Data is sent to each downstream node indicated in the PIT entries, and PIT entries are deleted because they have been satisfied.

In case the incoming Data cannot satisfy any pending Interest, the Data is *unsolicited* and should be dropped. A node must not forward unsolicited Data, but may choose to cache the Data in CS [1].

[1] This is useful in vehicular networks where a "data mule" node can cache unsolicited Data to serve potential future Interests [33].

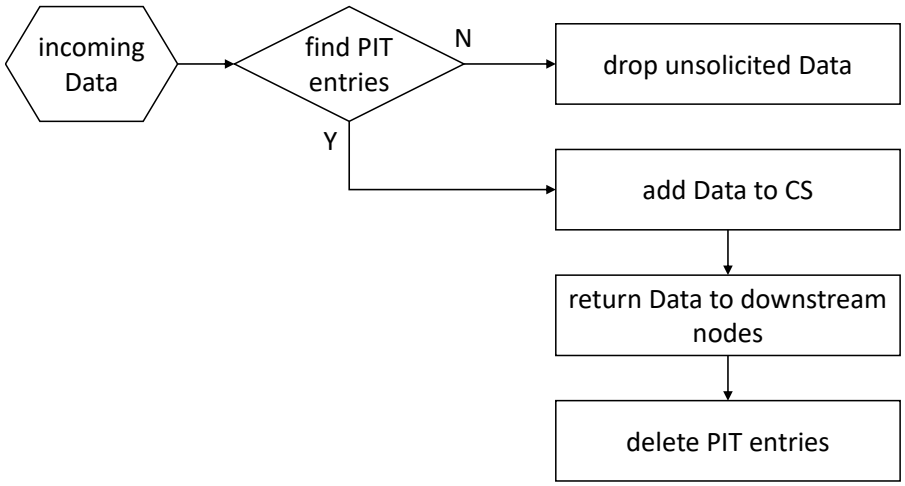


Figure 3.2: Data processing steps

3.2.3 Nack Processing Steps

Nack is a hop-by-hop message from upstream to inform the downstream that its Interest cannot be satisfied. As introduced in Section 2.2.2, a Nack packet contains the original Interest and a reason code such as "Duplicate" or "NoRoute". It triggers the downstream to take corrective actions such as forwarding the Interest onto an alternative path or sending Nacks to its downstream node(s). Figure 3.3 outlines the workflow of processing a Nack.

When a node receives a Nack, it first verifies that the Nack comes from an upstream node to which it has sent the Interest. In order to perform this check, the node queries the PIT to see whether the Interest carried in the Nack has a PIT entry, and checks whether the node transmitted the Nack is recorded as an upstream node in the PIT entry. If this check fails, the Nack is considered unsolicited and must be dropped.

The node also verifies that the Nack is against the latest Interest sent to the upstream. If the nonce in the Interest carried in an incoming Nack differs from the last nonce sent to the upstream (as recorded in the PIT entry), or in other words, the node has retransmitted the Interest but the Nack is against a previous Interest transmission, the Nack is considered outdated and should be dropped.

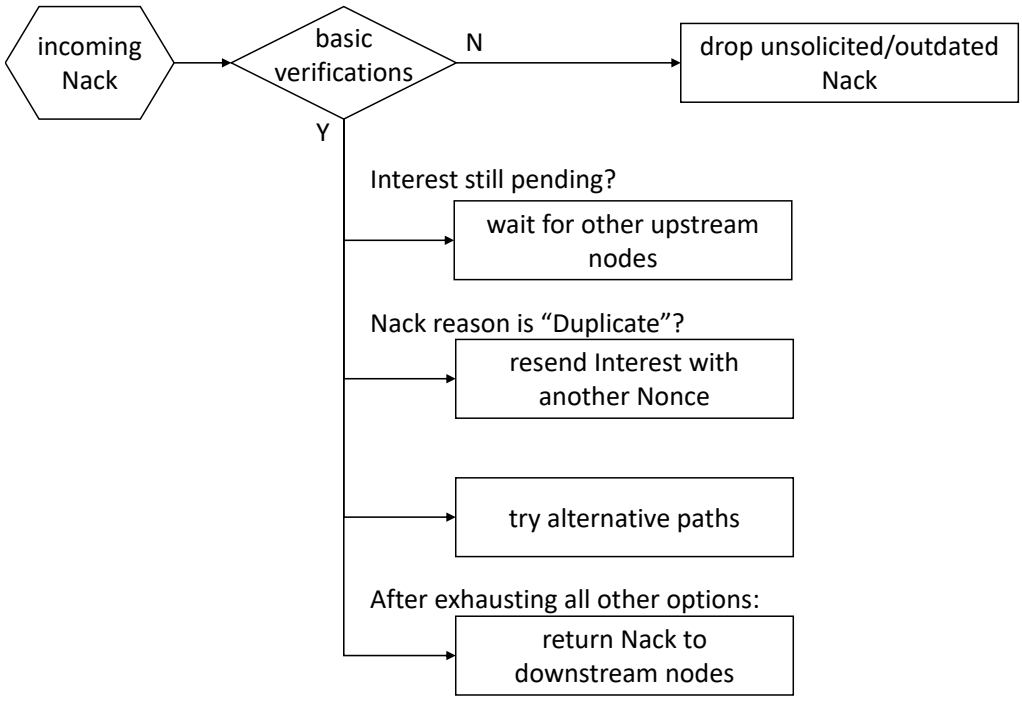


Figure 3.3: Nack processing steps

After verifying the Nack, the node should first exhaust all forwarding options, before giving up and sending Nacks to its downstream peers. Specifically, the node takes one of the following four actions:

Wait for other upstream nodes. If the Interest is still pending at other upstream nodes (according to records in the PIT entry), wait for them.

Resend the Interest with another nonce (for Nack-Duplicate only). A Nack with reason "Duplicate" comes back from an upstream node if the upstream has already received an Interest with the same nonce (Section 3.2.1). If the downstream has aggregated multiple incoming Interests in the PIT entry, receiving a Nack-Duplicate only means the nonce sent to the upstream is a duplicate, but other nonces may still be acceptable (see explanation in Section 3.3.3). Therefore, the node can resend the Interest to the same upstream with a different nonce selected among incoming Interests.

The node must ensure a nonce rejected by an upstream node through Nack-Duplicate is not sent again. For example, it may remember all rejected nonces in the PIT entry.

Try alternate paths. The node can forward the Interest to different next hops and attempt to retrieve data from there. The procedure of choosing alternate paths is similar to forwarding a new Interest (Section 3.2.1), which involves FIB lookup and strategy making decision. The strategy should ensure it does not use the same upstream multiple times for the same Interest. For example, it may keep track of which upstream it has tried and which upstream has returned a Nack in the PIT entry, and consider this information when choosing an alternate path.

Return Nacks to downstream. After exhausting all forwarding options, the node can give up by returning Nacks to downstream so they may take their own corrective actions.

In case a node has received Nacks from multiple upstream nodes with distinct reasons, it should choose the least severe reason to send back to downstream. Between the two defined Nack reasons, "Duplicate" is considered less severe than "NoRoute", because Nack-Duplicate allows a downstream to resend the Interest with a different nonce, while Nack-NoRoute does not allow that.

When returning a Nack to the downstream, the Nack packet to each downstream carries the last Interest from that downstream. Since each downstream usually sends the Interest with a different nonce, Nack packets sent to different downstream nodes would be different. This also means that the node is not forwarding the incoming Nack to downstream nodes, but generating new Nacks for each downstream; thus, Nack is a hop-by-hop message.

After returning Nacks, the node deletes the PIT entry because the Interest has been rejected and is no longer pending.

3.3 Loop Prevention

Loop is a common problem in computer networks. When the path to a particular destination forms a cycle among a group of nodes, when traffic for that destination arrives at any of these nodes, it will loop endlessly. The data plane of any computer network must be loop-free. In existing network protocols, Ethernet prevents loops by reducing the topology to a spanning tree, IP requires the routing protocol to provide loop-free paths, and OSPF prevents looping of routing announcements by checking sequence numbers (Section 2.1.2).

NDN shifts the responsibility of preventing loops to the forwarding plane. This section describes how NDN uses PIT states and the nonce field to prevent loops, and discusses the interaction with other parts of the NDN protocol.

3.3.1 Loop Prevention with PIT States

Forwarding state in the PIT is the primary method for loop prevention. When a node forwards an Interest, it inserts a PIT entry, which is kept until the Interest is satisfied by Data or expires after a timeout. This PIT entry is not only used for determining where to return Data, but also useful for preventing loops: if a pending Interest comes back, it will match the existing PIT entry, and the node would not forward it again. In case an Interest loops back after the original Interest has been satisfied and the PIT entry is erased, it will generally match a cached Data in CS, and thus will not be forwarded again. Data and Nack loops are naturally prevented, because they follow the reverse path of an Interest, so they cannot loop if Interests do not loop.

Relying on PIT state alone for loop prevention has three limitations:

- The node cannot distinguish whether an incoming Interest is looped back or is coming from a new consumer. It has to return the Data to every downstream. This wastes bandwidth in case the Interest was looped back.

- The node cannot retransmit the Interest to the same upstream, because re-

transmission can cause persistent loops.

- If a forwarded Interest comes back after it has been satisfied and the Data is also evicted from the CS, the node cannot detect the loop.

3.3.2 Duplicate Nonce Detection

To overcome the limitations of loop prevention using PIT states, NDN introduces a nonce field in every Interest packet to assist in loop detection. The **nonce** is a random number that identifies the Interest, typically generated by the consumer. Every node remembers recently seen nonces in the PIT and another data structure called the "dead nonce list" (Section 3.3.4). When an Interest arrives, its nonce is checked against the recently seen nonces, and the lookup result decides the next steps.

New nonce. If an incoming Interest has a new nonce, the Interest is not looping and should be forwarded normally.

Duplicate nonce from the same downstream. If an incoming Interest has the same nonce as a previous Interest, and the previous Interest comes from the same downstream node as the current Interest, it is safe to assume the Interest is not looping. The node treats the Interest as a retransmission, and forwards it normally.

Accepting such Interests as retransmissions has no danger of loops, because if there was a loop, the downstream node should have detected it. On the other hand, not requiring new nonces in retransmissions give downstream the freedom to retransmit an Interest without using a new nonce.

Duplicate nonce from a different downstream. If an incoming Interest has the same nonce as a previous Interest, and the previous Interest comes from a different downstream, the Interest must be dropped. This situation is either a loop, or a "multi-path arrival" described below.

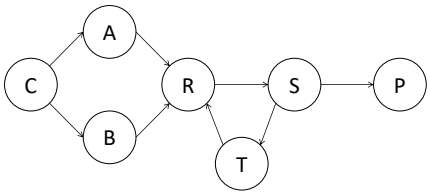


Figure 3.4: Loop vs multi-path arrival

NDN allows multi-path forwarding. In Figure 3.4 topology, a downstream node C sends an Interest with $nonce_C$ via two paths, $C-A-R$ and $C-B-R$. Those two Interests converge at the same upstream node R, and R would detect a duplicate nonce. Suppose R receives the Interest from A first, the Interest would be forwarded normally, and R remembers its nonce $nonce_C$. A short while later, R receives the Interest from B, and it would see the same $nonce_C$ on that Interest. This situation is called a multi-path arrival.

Unfortunately, without knowing the global topology, a node is unable to tell whether a duplicate nonce indicates a loop or is simply a multi-path arrival, if it has ever forwarded an Interest with that nonce. In the example, suppose R forwards the Interest from A toward S, but it loops back via $R-S-T-R$ path, R would see a duplicate nonce from T. Since R does not know the global topology, a duplicate nonce from T is no different as a duplicate nonce from B. To prevent Interest loops, R must drop the Interest in either situation.

3.3.3 Nack-Duplicate

When a node drops an Interest due to duplicate nonce, it should return a Nack with reason "Duplicate" to the downstream node. This helps cleaning up dangling PIT entries at the downstream, and avoids an undesired interaction with Interest aggregation.

Dangling PIT Entry

In Figure 3.4 example, when C's Interest is forwarded via both $C-A-R$ and $C-B-R$ paths, it leaves PIT entries on both A and B. Suppose R receives the

Interest from A first, R drops the later Interest from B because it has a duplicate nonce. At this moment, the PIT entry on B becomes "dangling", because it would not be satisfied by Data.

To clean up this dangling PIT entry, R should send a Nack to B. Upon receipt of this Nack, B runs through the Nack processing procedure (Section 3.2.3). Since does not have an alternative path in the example topology, B would promptly delete the dangling PIT entry and also return a Nack to C.

C would keep its PIT entry because it is still waiting for a response from A. If the same situation happens several times, the forwarding strategy at C can observe that sending the same Interest to both A and B leads to Nack-Duplicate from B, and it can stop doing so afterwards [34].

Duplicate Nonce Detection vs. Interest Aggregation

When an incoming Interest matches an existing PIT entry, a node can *aggregate* the Interest by adding the new downstream into the PIT entry, and not forward the Interest again (Section 3.2.1). This makes NDN scalable because each Interest is forwarded only once to the same upstream, but it has an unintended interaction with duplicate nonce detection: if the first Interest carries a nonce that is detected as duplicate at upstream, Interests from other downstream cannot be satisfied even if they are not duplicates.

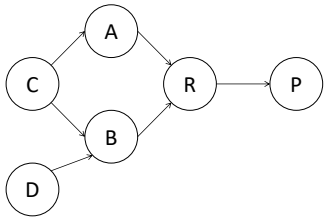


Figure 3.5: Interest aggregation interferes with duplicate nonce detection

In Figure 3.5 topology, C sends an Interest with $nonce_C$ via two paths to both A and B. The Interest via A reaches R first and is forwarded to P, and then the Interest via B arrives at R and is dropped because it has a duplicate nonce. Shortly

after that, D sends an Interest requesting the same data with $nonce_D$. B aggregates D's Interest into the existing PIT entry and does not forward it, but since R has already dropped the Interest from B, B and D would not receive the Data.

Sending a Nack from R to B can resolve this problem. If B receives the Nack before D's Interest, it deletes the PIT entry upon Nack, and D's Interest would be forwarded to R as a new Interest. If B receives the Nack after having aggregated D's Interest into the existing PIT entry, B would send up another Interest with $nonce_D$ right away. In either case, R would receive an Interest with $nonce_D$ and it would accept this Interest.

3.3.4 Storing Recently Seen Nonces

Duplicate nonce detection requires NDN nodes to remember recently seen nonces. A question is, how many nonces should a node remember, and where to store them?

The PIT is a natural choice for storing nonces, because it already has a name-based index, and a PIT entry is created for every incoming Interest unless it is satisfied by a cached Data. However, operational experience on the NDN testbed showed that storing nonces only in the PIT entry may lead to persistent loops. Although the NDN testbed had a loop-free routing protocol, in one incident, a routing misconfiguration created a network cycle with 400ms total latency, and an Interest was observed to be looping endlessly in the network. Log analysis revealed that the Interest had its InterestLifetime set to 100ms so that its PIT entry was deleted after 100ms. When the packet came back to the same node, NFD failed to detect a duplicate nonce because the PIT entry was gone, and forwarded the Interest into the cycle again, causing a persistent loop.

One potential solution is to ensure the minimum lifetime of a PIT entry to be at least the expected maximum length of a network cycle. That would ensure the PIT entry still exists when the Interest comes back, and therefore the node can detect the duplicate nonce. However, keeping PIT entries longer for the sole purpose of loop detection is wasteful, because a PIT entry contains much more information than the nonces, and retaining it requires a significant amount of memory.

Instead, I introduce Dead Nonce List (DNL) which is a compact data structure that stores name+nonce combinations. When a PIT entry is erased, nonces in the PIT entry are copied into the DNL. The nonce on an incoming Interest is queried not only in the matching PIT entry, but also in the DNL. This allows the node to store many more nonces in the same amount of memory.

CHAPTER 4

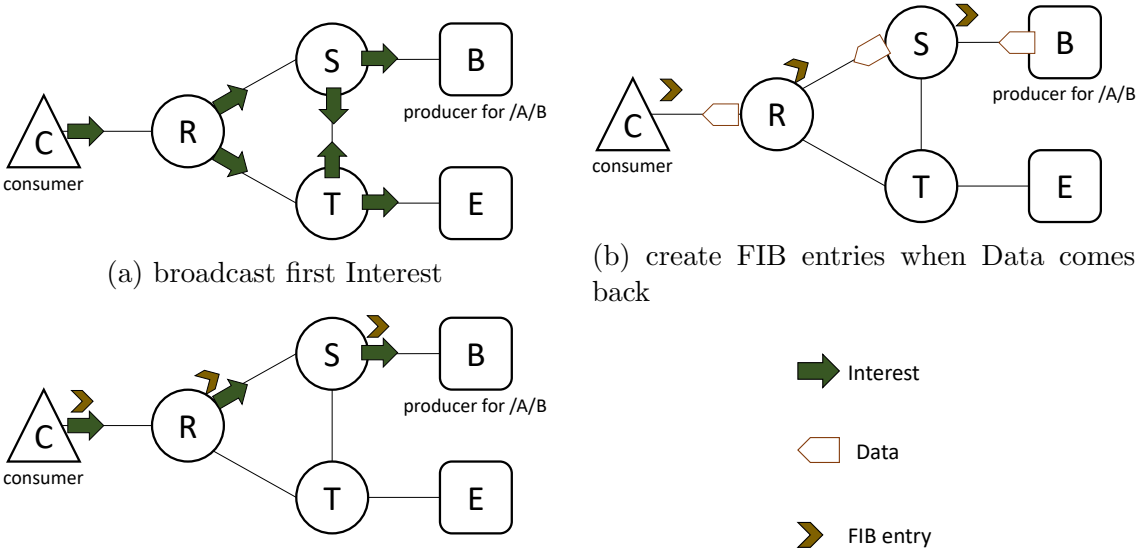
BROADCAST-BASED SELF-LEARNING IN NDN

Broadcast-based self-learning is a common mechanism in network designs. Self-learning broadcasts the first packet with an unknown path across the network. When, and if, a response packet returns, a forwarding table entry is created toward the destination, so that future packets will only need unicast. Forms of broadcast-based self-learning have been implemented in both switched Ethernet and Ad hoc On-Demand Distance Vector routing (AODV) [35].

Broadcast-based self-learning allows networks to adapt to changing environments and allows for server/producer mobility, without using any routing or other control protocols. These benefits are particularly notable in mobile ad hoc and wireless network environments, where no fixed infrastructure has been established and periodic routing announcements would cost undue computation time and energy. This mechanism also fits the NDN architecture well because it does not require prior knowledge of the location of data.

The high-level idea of broadcast-based self-learning is straightforward, illustrated in Figure 4.1: Node C sends the first Interest, which is broadcasted. When the Interest reaches the producer node B, a Data is returned on the reverse path of the Interest. Network nodes create FIB entries leading to B, so that future Interests will be sent via unicast.

Nevertheless, making the scheme efficient and secure requires addressing a number of technical issues: (a) NDN uses hierarchical names and FIB entries are associated with name prefixes instead of host addresses. Given the names of the broadcast Interest and the returned Data, what prefix granularity should be used in the FIB entry for a learned path? Section 4.1 examines FIB prefix granularity problem. (b) Broadcast-based self-learning lacks authentication, and is vulnerable to attacks such as ARP spoofing. Section 4.2 investigates how such attacks can be



(a) broadcast first Interest

(b) create FIB entries when Data comes back

(c) unicast subsequent Interests

Figure 4.1: Broadcast-based self-learning in NDN

prevented with NDN's data-centric security, and proposes a trust model for prefix announcements.

After examining these issues common to all networks, I developed **NDN self-learning**, a forwarding scheme applying self-learning to local area networks, and switched Ethernet in particular, that (a) builds forwarding tables in the data plane with low overhead; (b) recovers quickly from link failures and other path problems; (c) makes use of off-path caches for Internet contents. The protocol of NDN self-learning strategy is described in Sections 4.3 and 4.4.

I conducted evaluations using both real and synthetic traffic. As shown in Section 4.5, compared to Reactive Optimistic Name-based Routing (RONR) [4] and Ethernet, NDN self-learning can: (a) more accurately learn FIB prefixes from producers, leading to less packet flooding; (b) recover from link failures faster, without waiting for control plane convergence; (c) divert Interests requesting Internet contents to off-path caches in order to reduce the load on the Internet connection.

4.1 Prefix Granularity

The NDN Forwarding Daemon (NFD) [28] determines where to forward an Interest by doing a longest prefix match on the Forwarding Information Base (FIB), a name-based lookup table of forwarding paths. With self-learning, NFD floods Interests that do not match an entry in the FIB, and learns forwarding paths through observation of Data return paths, which are then inserted into the FIB. However, this still leaves unresolved the issue of how to determine the name prefix of an inserted FIB entry from the Data name. Having an accurate FIB prefix is critical for the performance of NDN self-learning.

In Figure 4.1, node B serves the prefix `/A/B`. After flooding an Interest `/A/B/C/1` and receiving a Data with the same name from B, network nodes can see the names of the Interest and Data, but have no knowledge of B's true prefix (`/A/B`). If they use a FIB prefix that is too specific (too long, such as `/A/B/C`), a subsequent Interest `/A/B/D/1` would be flooded unnecessarily. Conversely, if the FIB prefix is too generic (too short, such as `/A`), an Interest that cannot be served by the producer (e.g. `/A/E/F/1`) would be incorrectly forwarded to B.

I identified three potential solutions to the prefix granularity problem, which are described below.

4.1.1 k-shorter Prefix

The simplest solution to this problem is to derive the FIB entry prefix from the Data name, removing the last k components. For example, if the Data name is `/A/B/C/1` and $k = 2$, the FIB entry prefix would be `/A/B`.

However, the number of name components to remove (value of k) is highly dependent on the naming scheme used by the application. For example, *ndnputchunks*, a file transfer application, generates Data names in the format `/network-prefix/file-name/version2/segment0`, while *NdnCon*, a real-time conferencing application, generates Data names in the format `/network-prefix/user-name/camera3/1920x1080/key/frame500/data/segment0` [36]. The correct k value for each ap-

plication is 2 and 6, respectively.

In a general-purpose network, it is impossible for forwarder nodes to understand the naming scheme of every application and adjust k accordingly. Consequently, there would always be some FIB entries with prefixes that do not match the producer's prefix, negatively affecting network performance. Since unnecessary flooding does less harm than mis-forwarding Interests to the wrong producer, k is often conservatively set to a small value. RONR adopts this solution with $k = 1$.

4.1.2 FIB Aggregation

Another solution is to aggregate related prefixes into more generic prefixes over time. In this solution, NFD initially sets FIB prefixes conservatively, such as the Data name minus one component. Then, if most FIB entries under a common prefix point to the same next hop, they can be aggregated into a single entry at the shorter common prefix. As an example, if `/A/B/C`, `/A/B/D`, and `/A/B/E` all point to the same next hop, they are aggregated into an `/A/B` entry pointing to that next hop.

This mechanism is similar to IP prefix aggregation, but is necessarily different because the NDN namespace is infinite (Section 2.2.1). Even after a next hop is observed to serve many sub-prefixes under a common prefix, NFD cannot conclude with absolute certainty that the same next hop is able to serve all data under this prefix, because there are infinitely many sub-prefixes under it. Therefore, aggregations are optimistic, and it is necessary to deaggregate in case an aggregation is found to be incorrect.

For this method to work efficiently, the time complexity of the aggregation algorithm must be bounded by the length of the name. It is unacceptable for the algorithm to traverse a large subtree for deciding whether to aggregate an FIB entry. However, I was unable to design an efficient algorithm that satisfy the above requirements.

4.1.3 Prefix Announcements

The third and final solution is to have the producer explicitly inform the network of the prefix it serves. The producer attaches a **prefix announcement** on a Data packet sent in response to a flooded Interest. The prefix announcement is itself a Data packet, containing the prefix served by the producer. It is signed by the producer, allowing its authenticity to be verified (see Section 4.2.1).

The announcement is attached as a link-layer hop-by-hop header on the Data (Section 6.3). It is not part of the Data packet itself, and is not covered by the Data's signature. This separation allows a producer to serve previously-generated Data without re-signing. For example, a repository can collect video frames generated by a camera, and serve them to the network. The repository's prefix granularity is likely shorter than the camera's. By keeping the announcement out of the Data, the repository can replace the announcement with its own while keeping the camera's signature intact.

This solution is straightforward and allows FIB prefixes to adapt to applications' differing granularities. One drawback of this approach lies in the bandwidth overhead of transmitting announcements; however, this is compensated for by the reduction in computational overhead compared to FIB aggregation.

There are two variants of this solution: (a) Instead of flooding the first application Interest and attaching the announcement onto the application Data, the network can use an ARP-like procedure: the consumer floods an Interest asking "who has `/A/B/C/1`?", receiving the announcement in response, and then unicasts application Interests. However, this incurs an extra round-trip time before the first Data is retrieved. (b) Instead of letting the producer attach an announcement onto the Data, the producer prefix can be attached onto the Interest by the consumer. This was applied to vehicular networks in *Navigo* [37], but it would not work in general-purpose networks because consumers may not know a producer's prefix granularity.

4.2 Trust of Announcements and Data

ARP spoofing [38] is a common attack in IP-based local area networks. In this attack, a malicious node transmits an ARP message with a forged sender IP address, allowing it to capture, manipulate, or even block traffic intended for another host. A similar **producer spoofing** attack can occur with NDN self-learning. To perform this attack, a malicious node responds to flooded Interests with bogus Data, causing subsequent Interests to be forwarded toward itself. This attack is particularly effective when combined with prefix announcements: by announcing a very short prefix, the resulting FIB entry can attract Interests under a large namespace.

In this section, I propose a mitigation of producer spoofing attacks using NDN's mandatory Data authentication, which requires all applications to sign and verify every Data packet. This is done by applying two tiers of trust models: (a) a universal, network-wide trust model for authenticating prefix announcements, and (b) application-specific trust models carried in every prefix announcement for validating packets under the announced prefix. Announcement trust and application data trust are separated because there is not a universal trust model that covers all application data; instead, each application can define its own trust model. Conversely, a LAN is under the control of one entity, so it is feasible to define a universal trust model for prefix announcements.

4.2.1 Trust Model for Announcements

In a routing-based NDN network, all prefix announcements are authorized by the network administrator. A trust relationship among routers prevents other nodes from announcing arbitrary prefixes [22]. In broadcast-based self-learning, applying a similar principle, the network administrator can authorize prefixes announced by producers.

When a content-producing node joins the network, its operator will request an **announcement signer certificate** from the network administrator. Part of the

name of this certificate indicates which prefix the producer is allowed to announce [1]. The administrator then signs the certificate with the LAN's certificate authority. After the certificate has been issued, the producer can sign its prefix announcements with this certificate.

The network administrator configures every switch/router in the network with a trust model for authenticating prefix announcements. Upon receiving a prefix announcement, NFD can verify that: (a) the announcement has a valid signature matching the public key in the announcement signer certificate, (b) the announced name prefix matches the allowed prefix encoded as part of the certificate name, and (c) the certificate is issued by the LAN's certificate authority. If all three conditions are satisfied, NFD accepts the prefix announcement and inserts a FIB entry for the announced prefix.

One issue with this procedure is that, if a received prefix announcement is signed by a certificate not in NFD's cache, NFD will need to retrieve the certificate from the producer. NFD cannot return the Data to downstream until the certificate has been retrieved, the prefix announcement has been verified, and the FIB entry has been inserted. Otherwise, the downstream may send subsequent Interests, but NFD would not be able to forward them because the relevant entry has not yet been inserted into the FIB. Instead, NFD must hold the Data in a buffer during certificate retrieval. However, this adds to the forwarding delay, and may cause the downstream to choose a suboptimal, slower path where there was no significant certificate retrieval delay.

As a trade-off between forwarding performance and security, NFD may elect to temporarily insert the FIB entry during certificate retrieval, allowing it to return the Data immediately, so the consumer can resume sending Interests right away. In the case that a prefix announcement is found to be bogus, NFD should delete the FIB entry and penalize the producer. This penalty is applied by requiring, for a limited time, that announcements received from this producer be verified before they

[1] To produce content under multiple prefixes, each prefix needs a separate announcement signer certificate.

are inserted into the FIB, disabling the temporarily insertion mechanism described earlier.

4.2.2 Trust Schema for Application Data

While an announcement trust model ensures that no one can announce unauthorized prefixes, a malicious node can still collect a prefix announcement from a legitimate producer, and attach it onto a bogus Data packet. This is a form of replay attack, as the malicious node is replaying an old announcement. A standard countermeasure to replay attacks is challenge-response. In this case, a forwarder would generate a random piece of information as a "challenge", and the producer should include the challenge in the prefix announcement, signed by the announcement signer certificate, which proves that the announcement is fresh (not replayed). However, using challenge-response would require that every flooded Interest be processed at the producer and prevent the use of in-network caches, as they are unable to sign the challenge.

To prevent producer spoofing, while keeping the benefits of NDN's in-network caching, I propose a different solution: Every prefix announcement may carry the trust model for Data packets under their announced prefix, encoded as a *trust schema* (Section 2.2.4). In-network caches are able to respond to flooded Interests with legitimate Data, but malicious nodes cannot poison the network with bogus Data that does not comply with the trust schema.

NFD is normally not required to verify application Data because it would be prohibitively expensive. Instead, if a consumer detects a bogus Data during its validation, it can send a report to the network [39], which then triggers NFD to verify the application Data against the trust schema. If the Data is found to be bogus, NFD (a) deletes the FIB next hop record that brought back bogus Data; (b) if the upstream is an end host, stops flooding Interests toward it for a period of time as a penalty; (c) if the upstream is a switch, forwards the report to upstream so that it can block the malicious producer. On the other hand, if a downstream node has sent a false report and the Data turns out to be valid, NFD would ignore future

reports from that downstream for a period of time to prevent Denial of Service (DoS) attacks.

4.3 Self-Learning on Switched Ethernet

After examining two issues common to all network environments where NDN self-learning may be used, this section describes a specific design that applies NDN self-learning to switched Ethernet in wired LAN environments. I explain how self-learning builds forwarding tables in the data plane without any other control protocols and with low overhead, and how it can quickly recover from link failures and other reachability problems. This design primarily targets wired networks with hundreds of nodes, such as those found in an office building or a university department.

4.3.1 Building Forwarding Tables in the Data Plane

NDN self-learning determines the location of content by flooding the first Interest, and observing the return path of the corresponding Data packet. Nodes build their forwarding tables using the prefix announcements carried on Data. This happens entirely in the data plane, and does not require a routing protocol or any other control plane protocol.

This process is similar to the flood-and-learn process in switched Ethernet (Section 2.1.2). Unlike Ethernet, NDN has built-in loop freedom (Section 3.3) and does not require the Fast Spanning Tree Protocol (FSTP) to prevent bridge loops. This not only eliminates a control plane protocol, but also enables NDN to utilize all available links.

After a successful Interest flooding, NFD inserts a FIB entry for the prefix listed in the returned Data's prefix announcement. The next hop of this entry points to the upstream from which the Data was received. Following this FIB entry, NFD forwards subsequent Interests under this producer's prefix along the known path via unicast, and does not flood them again.

If there are multiple paths to reach the producer, self-learning would learn the

fastest path. This is a consequence of forwarding plane's duplicate nonce detection mechanism (Section 3.3.2): when two Interests with the same nonce arrive at the same node via multiple paths, only the first copy is accepted, while other copies are discarded. Later, Data reply is returned only via the fastest path traversed by the Interest at the time of flooding, and the network learns this fastest path.

The number of dynamic FIB entries created by self-learning is subject to a capacity limit. When the FIB is full, the Least Recently Used (LRU) entry is erased.

4.3.2 Minimizing Flooding Overhead

Interest flooding consumes network bandwidth, and incurs CPU processing overhead at end hosts that receive the flooded Interests but do not have the content. The key to minimizing overhead is to flood less often and in smaller regions.

In keeping with this goal, only the consumer can initiate Interest flooding. Flooding should be initiated for any Interest with an unknown path or when the previously learned path has failed (see Section 4.3.4). A *discovery tag* field on every Interest indicates whether the Interest is being flooded or not. The consumer tags an Interest as "discovery" if it wishes to flood the Interest; otherwise, the Interest is tagged as "non-discovery". The inclusion of this tag allows the consumer to regulate the flooding frequency.

When NFD receives a discovery Interest but already knows a path, NFD retags the Interest as "non-discovery" and forward it along the known path via unicast, effectively "absorbs" Interest flooding. This mechanism has two benefits: (a) when multiple downstream nodes concurrently request data under the same name prefix, each of them may send a discovery Interest if it does not know the prefix. If NFD knows the prefix and path, it can turn them into unicast instead of flooding them. (b) If a malicious downstream sends all Interests as discovery, NFD can retag most of them as non-discovery, stopping a potential DoS attack [2]. After retagging an

[2] A malicious consumer can still launch a DoS attack by sending discovery Interests for non-existent prefixes, and a different countermeasure such as [40] would be needed.

Interest as non-discovery and unicasting it following a FIB entry, when the Data packet comes back, the original prefix announcement that was used to create the FIB entry is put back onto the Data packet, allowing the downstream to learn the producer's prefix.

4.3.3 Fast Reaction to Link Failures

Link failures are infrequent but inevitable in wired switched Ethernet environments. NFD constantly monitors the availability of learned paths, and informs the consumer when a link failure is detected so that it may initiate another flooding.

NFD uses a variant of Bidirectional Forwarding Detection (BFD) [41] at the link layer to detect link failures between adjacent nodes. NDN-BFD treats any incoming NDN packet on a link as an indication that the link is up. If a node has not sent any NDN packets on a link during a specific period of time (e.g. 5ms), it transmits an "idle packet", which informs the neighbor that the link is still up, even though there is no current activity. If no NDN packets or idle packets arrive within a longer period of time (e.g. 50ms), NDN-BFD declares a link failure to the forwarding plane.

A link failure affect all pending Interests already forwarded via the failed link, as well as all future Interests that would have used the failed link as their next hop. For already forwarded Interests, although NFD could attempt to retransmit them on an alternate path, it is computationally expensive to do so, because the Pending Interest Table (PIT) is organized by name prefix and NFD cannot easily enumerate all affected PIT entries. Thus, as a trade-off between recovery speed and processing overhead, self-learning relies on consumer retransmission (see Section 4.3.4) to retransmit these Interests.

NFD will react to link failure when it receives another Interest (which could be a new Interest or a retransmission) matching a FIB entry pointing to a failed link. It does not forward the Interest onto the failed link. Instead, if the FIB entry has a known alternate path, NFD forwards the Interest on that path. If no alternative path is known and the incoming Interest is tagged as "discovery", NFD floods the

Interest. Otherwise, since there is no alternative path and the Interest cannot be flooded, NFD informs the downstream about the link failure by returning a *Nack* with reason "NoRoute" against the non-discovery Interest (Section 2.2.2).

Upon receiving the Nack, the downstream node sets a flag in the relevant FIB entry indicating that the current path to this prefix has a link failure. It then runs through the same procedure as the node adjacent to the failure (listed above). Eventually, if no functioning alternate path is found, the Nack will be returned, hop by hop, to the consumer. Then, the consumer can retransmit the Interest, tagged "discovery", to initiate a new flooding, which may find another path to the producer.

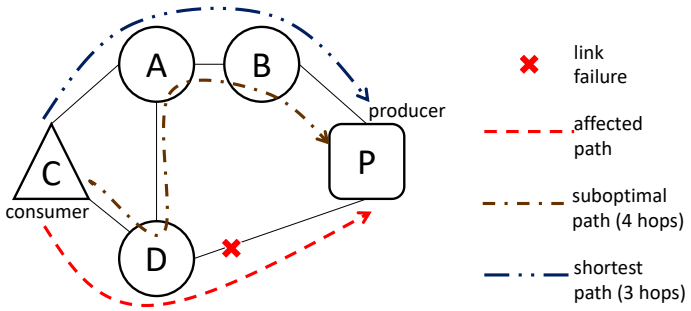


Figure 4.2: Flooding from node D learns suboptimal path

The NDN self-learning protocol requires flooding to be initiated from the consumer, instead of allowing NFD to flood a (non-discovery) Interest when there is a link failure, because flooding from an in-network node might cause the network to learn suboptimal paths. In Figure 4.2, consumer C initially retrieves Data from P via $C-D-P$ path. When link $D-P$ fails, if D floods the Interest, the path learned would be $C-D-A-B-P$, which is one hop longer than the shortest path $C-A-B-P$ which could be learned if flooding was initiated from the consumer. This is also the motivation of introducing a "discovery tag" into the Interest packet to indicate whether an Interest is being flooded.

4.3.4 Consumer Retransmission

When the consumer believes or is informed that an Interest or Data has been lost, but still wishes to retrieve the content, it should retransmit the Interest (Sec-

tion 2.2.2). The retransmitted Interest has the same name, but carries a new nonce, to avoid being detected as a duplicate. The consumer can either send a non-discovery Interest, allowing the network to explore known alternate paths, or send a discovery Interest, initiating a new round of flooding.

There are two situations when a consumer should retransmit an Interest:

Nack As mentioned in the previous section, when the network returns a Nack-NoRoute to the consumer, the consumer should retransmit a discovery Interest to re-trigger flooding. Note that the consumer should always tag the retransmitted Interest as "discovery". It would be useless to retransmit a non-discovery Interest, because the network has already exhausted all known alternate paths prior to returning the Nack to the consumer.

Retransmission timer Once the consumer has learned the prefix granularity from the prefix announcement, it can measure the round trip time for data retrievals, and use it to calculate the retransmission timeout (RTO) using TCP's algorithm [42]. After expressing an Interest, if no Data comes back within the RTO, it can retransmit the Interest.

NFD can distinguish a retransmitted Interest from a normal Interest because there is a PIT entry indicating a previous Interest with the same name is not yet satisfied. Retransmitted Interests are processed slightly different from normal Interests: for a non-discovery Interest retransmission, NFD would try a known alternate path other than the previously used path, unless there is only one path in which case the retransmission would be sent that way; for a discovery Interest retransmission, NFD would bypass FIB next hops and flood the Interest right away.

This consumer retransmission mechanism is primarily used to recover from a link failure. If the retransmission was triggered by the retransmission timer, the lack of a Data reply could also be caused by packet loss. But in NFD, packet losses due to link error are greatly reduced through link layer retransmissions [43]. In this work, it is also assumed that there is no congestion, as most wired LANs are over-provisioned.

If a congestion does occur and causes some packets to be dropped, retransmitting discovery Interests and flooding them will likely worsen the congestion. Thus, this design could be used in conjunction with a congestion control scheme [44] for better results.

4.3.5 Handling Producer Mobility

Mobility is handled using the same procedures as link failure: a node that has moved is treated as a link failure between the previous hop and the moved node. When an Interest reaches the previous hop, a Nack will be returned so that the network can discover the new location of the Data. After finding a new path, the new next hop is added to the FIB entry and will be used for subsequent Interests. The old path is retained in the FIB entry as an alternate. Retaining the old path is especially useful when a node is moving back and forth between several attachment points. In this case, the forwarder at the branching point could switch between the alternative paths without repeated flooding.

One side effect of retaining multiple alternate paths is that traffic may concentrate on the most stable path. To avoid this, NFD probes all known paths at a regular interval and switches the primary path to the fastest found during testing.

4.4 Caching of Internet Contents

One of NDN's benefits is in-network caching. Every node opportunistically caches Data packets passing through it in the Content Store (CS), allowing them to satisfy future Interests requesting the same Data (Section 2.2.3). Both discovery and non-discovery Interests can be satisfied with cached Data. When answering a discovery Interest, NFD attaches the original prefix announcement onto the Data. It is possible for an Interest flooding may learn a path toward a cache, instead of a path toward the producer. If a subsequent Interest is forwarded to a cache but cannot find a match in CS, NFD should forward or flood (if the Interest is tagged "discovery") the Interest as it normally would, or return a Nack if there is no available upstream. Returning

a Nack would cause the network to find a path to another cache or producer.

In-network caching allows popular contents requested by multiple consumers to be satisfied from a nearby cache, resulting in bandwidth savings and latency reduction. However, for most local contents produced by a node within the local area network (LAN), these improvements are marginal, as internal links have abundant bandwidth and low latency. On the other hand, most LANs are connected to the Internet via wide area network (WAN) connections that often have limited bandwidth and high latency. Maximizing the utilization of in-network caches can significantly improve the performance of Internet content retrieval.

4.4.1 Internet Contents Retrieval

Internet contents are retrieved via a gateway router. The network uses normal Interest flooding to discover paths to the gateway router, which announces itself as the "producer" of the "/" prefix. One issue is that, a FIB entry at the "/" prefix will match not only Interests requesting Internet contents, but also Interests requesting local (internal) contents. To avoid this problem, the gateway's announcement should carry a *local prefix list*, containing a list of administrator-defined prefixes internal to the LAN. If an Interest matches this root FIB entry, but falls under one of the local prefixes, it will not be forwarded toward the gateway, but instead flooded or Nacked as if there were no FIB match.

Since this FIB entry matches every Interest requesting Internet contents, the network will unicast all Interests for Internet content toward the gateway, allowing only caches on the path between consumers and the gateway to be utilized. Although these on-path caches can satisfy some redundant Interests, requiring less content to be retrieved from the Internet, the caches in network switches and the gateway router have limited capacity and handle a high volume of traffic, and therefore can only offer limited assistance in this regard. However, there is abundant cache capacity on other, off-path nodes, especially on end hosts. Retrieved Data can be cached for longer on these nodes due to the less frequent replacement of old entries. Therefore, the next section extends the design to utilize off-path caches for Internet traffic by

diverting Interests to them, in order to reduce WAN connection bandwidth usage.

4.4.2 Diverting Interests to Off-Path Caches

To utilize off-path caches, all nodes remember where each Data packet they processed was forwarded to and, therefore, where they may be cached. After these Data packets have been evicted from the on-path cache, Interests requesting them can be forwarded to potential off-path caches.

Information about potential off-path caches is stored in the CS. When a Data is forwarded downstream, the downstream node is recorded as a potential off-path cache in the Data's CS entry. When a CS entry must be evicted, instead of deleting it altogether, it is converted to a "stub" entry, containing the Data name and the locations of potential off-path caches, but not the Data payload. Since a name is much smaller than a Data payload, a node can store many more stub entries than regular CS entries. These stub entries are subject to a separate capacity limit and can be evicted once that limit is exceeded.

When an incoming Interest matches a CS stub entry, the forwarder examines potential off-path cache locations, and decides whether to divert the Interest by predicting whether the Data is still cached on the off-path cache. This prediction uses a simple heuristic: a Data is less likely to have been evicted if less Data packets have been sent to that downstream in the meantime. This heuristic is implemented in three steps: (a) NFD maintains an outgoing Data counter for each downstream peer. This counter is incremented every time a Data packet is sent to the downstream. (b) When a downstream is recorded as a potential off-path cache, the current value of that downstream's counter is recorded in the CS entry. (c) To determine whether an Interest should be diverted to an off-path cache, NFD calculates the number of other Data packets sent to this cache after the requested Data. This is done by subtracting the value recorded in the CS entry from the current counter value. If the difference is less than or equal to a pre-determined *diversion threshold*, the Interest is diverted to the off-path cache, and not forwarded further toward the gateway. In case there are multiple eligible off-path caches, the Interest is diverted

to the one with the least "other Data count", as it is most likely to still have the Data.

When an off-path node receives a diverted Interest [3], the forwarder on that node queries its CS to look for a match. If a regular CS entry is found, the Data packet is returned to the diverting node, which returns the Data to the downstream. If a stub CS entry is found, the Interest is passed to the downstream that is most likely to still have the Data, chosen with the same heuristic as above; however, it is not subject to the diversion threshold criteria. If there is no match in the CS, most likely because the stub entry has been evicted, the off-path node returns a Nack to the diverting node, which then forwards the Interest toward the gateway. This Interest is also tagged "no diversion", disallowing other on-path nodes from diverting it again, in order to limit the extra latency introduced by Interest diversion.

4.5 Evaluation

NDN self-learning was implemented in NFD, and evaluated in the Mininet network emulator [45], with both real world and synthetic topologies and traffic traces. Three additional forwarding schemes were implemented for comparison.

- The "broadcast" scheme reused NFD's multicast forwarding strategy, which floods Interests to every neighbor except the downstream, and returns Data via the shortest path.
- RONR was implemented according to [4]. FIB granularity is derived from the Data name, removing the last component. The FIB expiration timer is set to 5 seconds.
- "Ethernet-style self-learning" adapted Ethernet's self-learning algorithm to NDN. Interests and Data are restricted to the spanning tree, calculated by an

[3] A node can tell that the Interest is already diverted because it is coming from an upstream in the direction of the gateway.

FSTP implementation from the VDE switch emulator [46]. FIB granularity comes from prefix announcements, and FIB entries expire after 15 seconds.

4.5.1 Low Bandwidth Usage

NDN self-learning can learn producers' accurate prefixes from prefix announcements. These announcements are used to build forwarding tables in the data plane with low overhead. I evaluated the performance of NDN self-learning using the NFS trace captured at Arizona Computer Science department network and the actual topology of this network. Appendix A describes how I collected the traffic trace, and the `nfs-trace-client` and `nfs-trace-server` programs that replay the traffic trace in an NDN-based file access protocol.

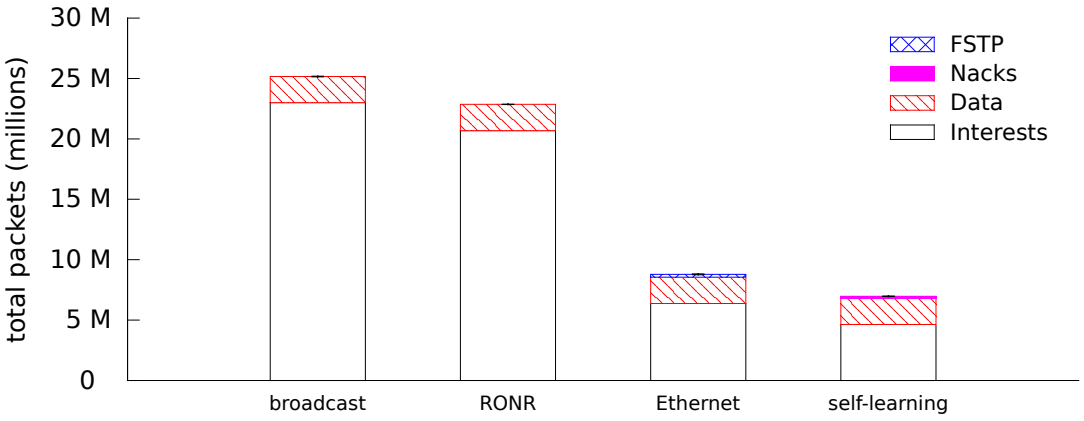


Figure 4.3: NFS experiment, total packets

I selected a subset of the NFS trace (Section A.4), which contains 428443 NFS operations in 24 hours. Each hour is emulated separately; reported numbers are the total of 24 hours, averaged among four trials with error bars showing standard deviation where applicable [4]. Figure 4.3 shows the number of packet transmissions needed to complete the NFS operations with each forwarding scheme. The FIB capacity of NDN self-learning was set to 50 entries per node; RONR and Ethernet used a max-

[4]Error bars may not be visible because the standard deviation is very small for most experiments.

imum of 900 and 63 FIB entries per node, respectively. The results demonstrate that NDN self-learning transmitted the least number of packets. Ethernet-style self-learning, which also uses prefix announcements to determine FIB granularity, transmitted 26% more packets than self-learning. 3% can be attributed to FSTP, while the rest is caused by increased flooding because Ethernet's FIB entries expire at 15 seconds. RONR uses a 1-shorter prefix for its FIB entry granularity, which required one flooding per file for most NFS operations, because Data names usually end with version number and segment number components (Section A.3). This pushed RONR's bandwidth usage close to that of broadcasting.

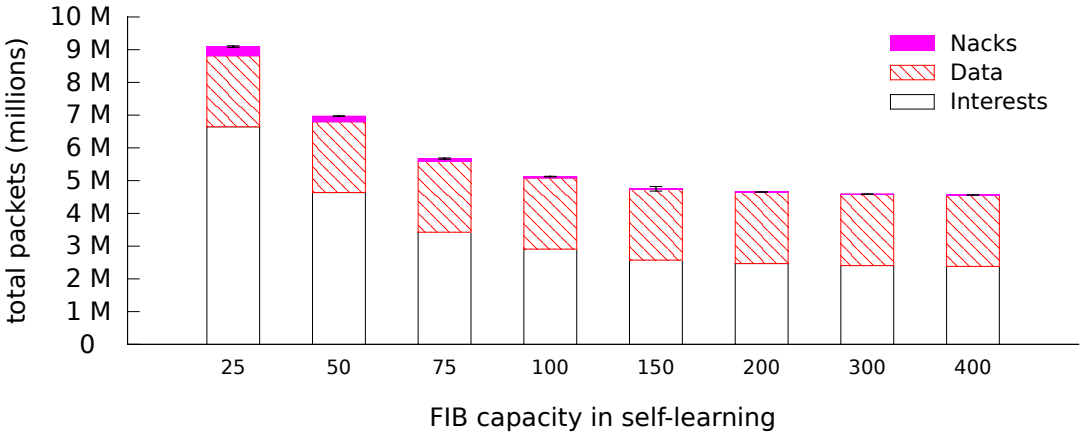


Figure 4.4: Effect of FIB capacity in NFS experiment

I studied the effect of FIB capacity on NDN self-learning, with results shown in Figure 4.4. It can be seen that NDN self-learning performs better with a higher FIB capacity limit, because every node is able to remember more forwarding paths for a longer period of time, resulting in less flooding. However, the improvement was marginal when the FIB capacity was greater than 150.

4.5.2 Fast Reaction to Link Failure

Each forwarding scheme was evaluated for their reaction to link failure on the topology in Figure 4.5. The topology contains 12 NDN switches and 12 end hosts. Node P is a producer, serving content under `/P`. Node C is the consumer executing

`ndnping` that continually sends Interests for `/P/ping/<sequence-number>` at a 1-second interval. The consumer application will retransmit an Interest at most twice upon receiving a Nack, but will not retransmit after a 4-second timeout, because a Data arriving more than 4 seconds late is no longer useful to the consumer.

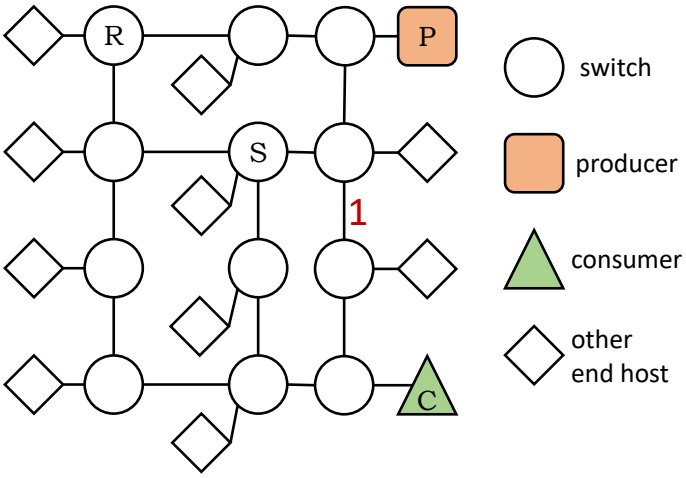


Figure 4.5: Topology for link failure experiment

The experiment lasted 60 seconds. 30 seconds into the experiment, link 1 experienced a failure. Since link 1 sits on the shortest path between C and P, it is expected that traffic will be affected, and therefore the forwarding scheme should find an alternate path.

I ran this experiment for the broadcast, Ethernet, RONR, and NDN self-learning schemes. Since the location of the root bridge in Ethernet affects which links are on the spanning tree, I ran Ethernet experiment twice, with R and S selected as the root bridge. I also ensured that link 1 was on the initial spanning tree.

Emulation revealed that, although every forwarding scheme can eventually detect a link failure and switch to an alternate path, the bandwidth usage and recovery speeds varied greatly. Figure 4.6 shows the total number of packet transmissions across the network by each scheme, averaged among seven trials with error bars showing standard deviation where applicable. Among the 60 end-to-end Interest-Data exchanged, there were 5.0 losses with RONR, 1.0 loss with Ethernet-R, 0.7 loss with Ethernet-S, and zero loss with broadcast or NDN self-learning. RONR

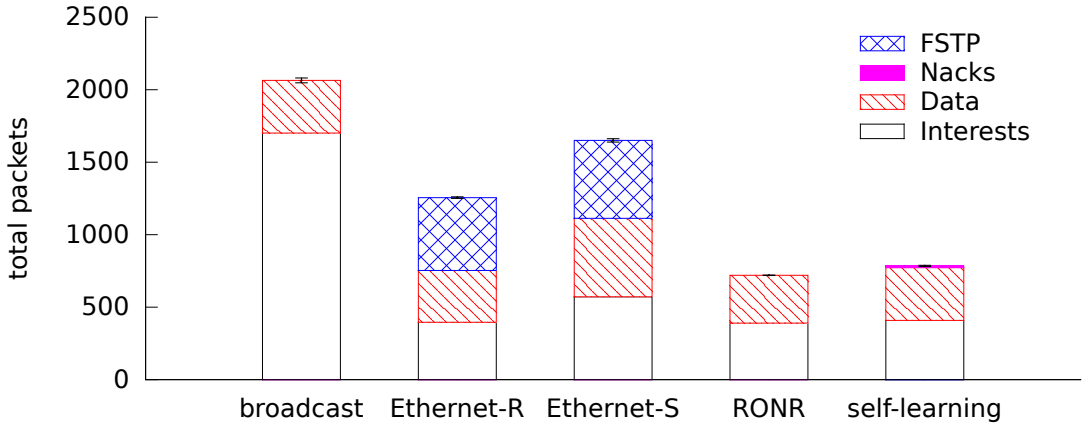


Figure 4.6: Link failure experiment, total packets

demonstrated 8.2% less bandwidth usage compared to NDN self-learning, but its end-to-end loss rate is the highest. NDN self-learning had the second least packet transmissions to complete the communication.

Within Ethernet scenarios, the number of packet transmissions varied significantly depending on the choice of root bridge, or equivalently, the location of the consumer and producer nodes in relation to the root bridge. In this particular communication, having R as the root bridge resulted in 23.9% less packet transmissions compared to having S as the root bridge. This is due to the difference in end-to-end path lengths, both before and after the link failure.

I can also see that FSTP accounted for a large portion of the packet transmissions, because the application traffic volume is low in this experiment. Higher application traffic volume would reduce the proportion of FSTP overhead. However, even after excluding FSTP overhead, Ethernet-S still transmitted 41.8% more Interest/Data packets than NDN self-learning's total bandwidth usage.

I also compared the number of Interest transmissions in RONR and NDN self-learning for each Interest (shown in Figure 4.7). Both schemes flooded `/P/ping/0` and learned the shortest path. Link 1 failed after `/P/ping/30` was retrieved. The sharp increase in Interest transmissions indicates that NDN self-learning detected the link failure immediately and responded by promptly flooding `/P/ping/31`. On

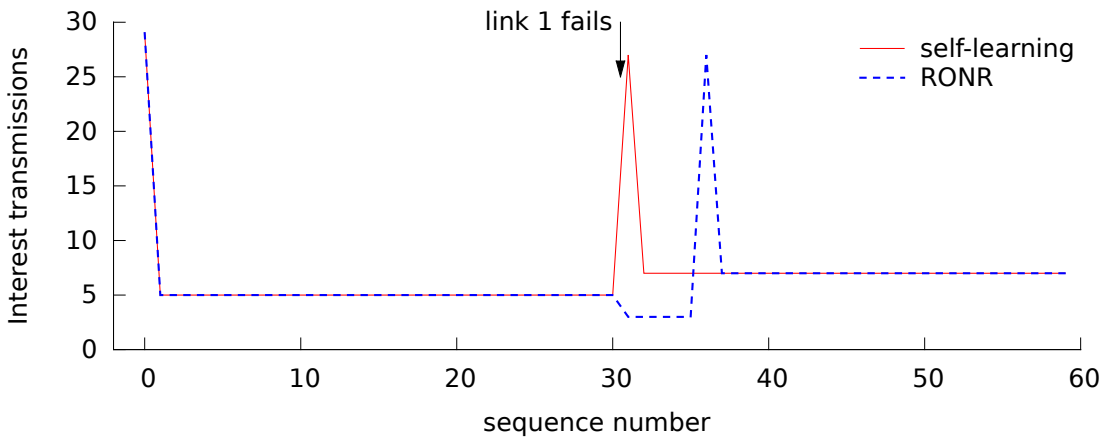


Figure 4.7: Link failure experiment, Interest transmissions

the other hand, RONR was stuck on the failed path for 5 seconds until the FIB entry expired, causing packet loss during these 5 seconds, and then discovered a new path by flooding `/P/ping/36`.

4.5.3 Off-path Cache Utilization

The Interest diversion design was also evaluated for its effectiveness in utilizing off-path caches for Internet content retrieval. This experiment used the tree topology shown in Figure 4.8. In this experiment, each end host ran a consumer program that retrieved 5000 Data packets from the Internet over a period of 1000 seconds, with a 200ms interval between Interests. Interest names followed a Zipf distribution with the parameters $N = 200000$ and $s = 0.955$, representing a distribution seen in Internet streaming services [47]. The end hosts also retrieved Data from each other at regular intervals, such that local traffic accounted for either 20%, 50%, or 80% of the total traffic volume (number of expressed Interests) of the network, depending upon the scenario. The CS capacity on each node was set to 2000 Data packets and 4000 stub entries, both using the First In First Out (FIFO) replacement policy.

The main benefit of diverting Interests to off-path caches is to decrease the amount of traffic on the WAN connection, which is often the source of congestion and latency when accessing Internet content. Figure 4.9 shows the number of pack-

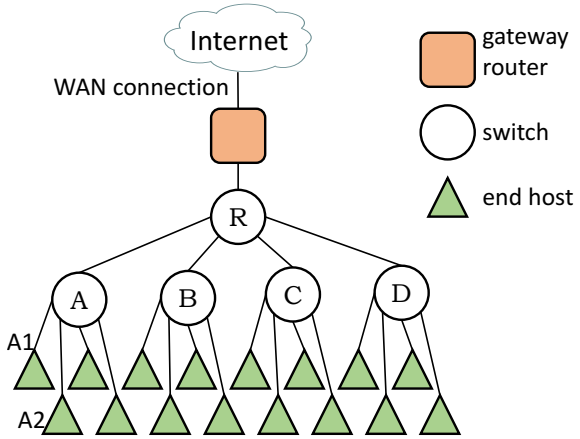


Figure 4.8: Topology for the Internet content retrieval experiment

ets traversing the WAN connection when running self-learning without diversion ("nodivert") and with diversion, under three different traffic ratios, averaged over five trials with error bars showing standard deviation where applicable. I can see that the Interest diversion mechanism can effectively reduce utilization of the WAN connection by up to 12.9%. The total amount of bandwidth saved by this mechanism was affected by what percentage of the LAN's traffic volume consists of Internet content. It was more effective when Internet traffic made up a greater share of the LAN's total traffic, as demonstrated in the lower curves.

The diversion threshold parameter (X-axis), used by the forwarder to determine whether a potential off-path cache may still have the Data (mechanism described in Section 4.4.2), had a significant impact on the effectiveness of the Interest diversion mechanism. When the diversion threshold was set to a small value (e.g. 250), no diversions occurred in these experiments, so that there was no change in performance. Larger diversion thresholds tended to achieve greater benefit, as long as they were set below the CS capacity ($\leq$ 2000). Setting the threshold to be greater than the CS capacity (e.g. 3000) caused a slight reversal in the trend of decreased WAN connection utilization, but still allowed for better performance than self-learning without diversion.

The processing overhead of diverted Interests is shown in Figure 4.10. In this plot, "hits" indicate how many diverted Interests were satisfied by an off-path cache,

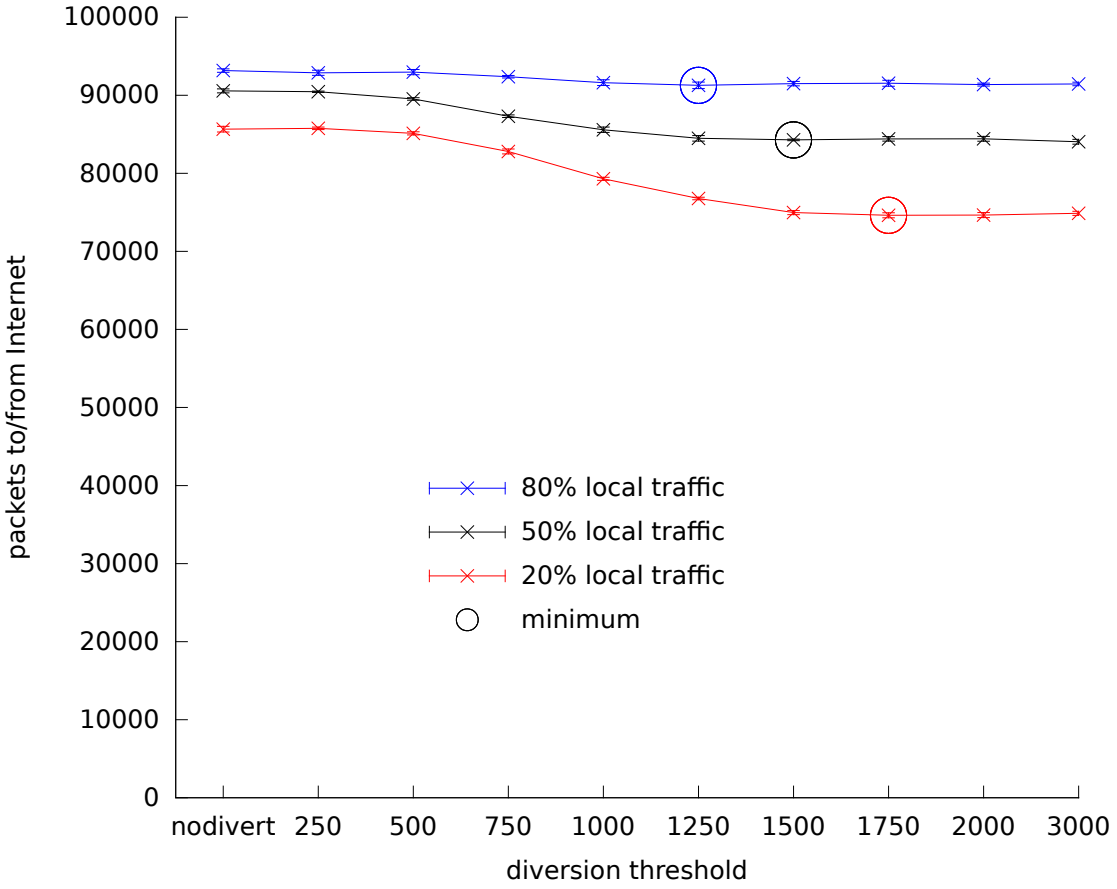
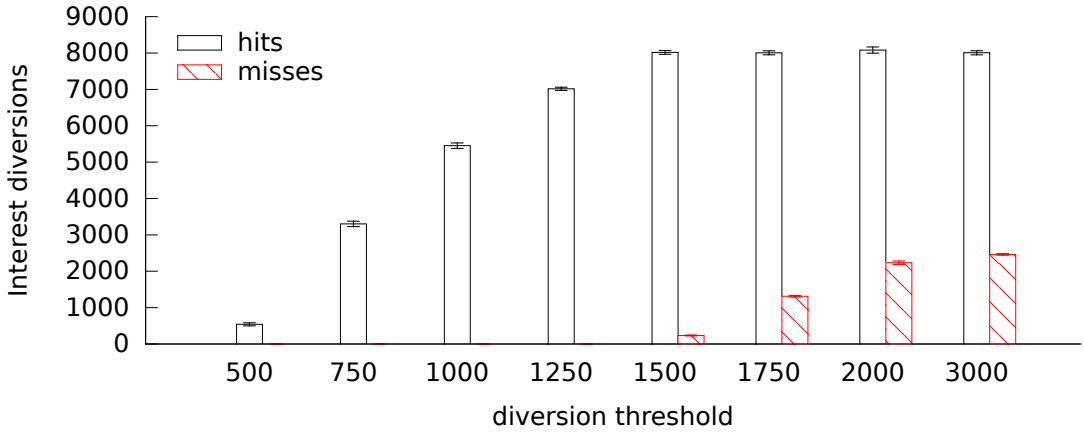


Figure 4.9: WAN connection utilization

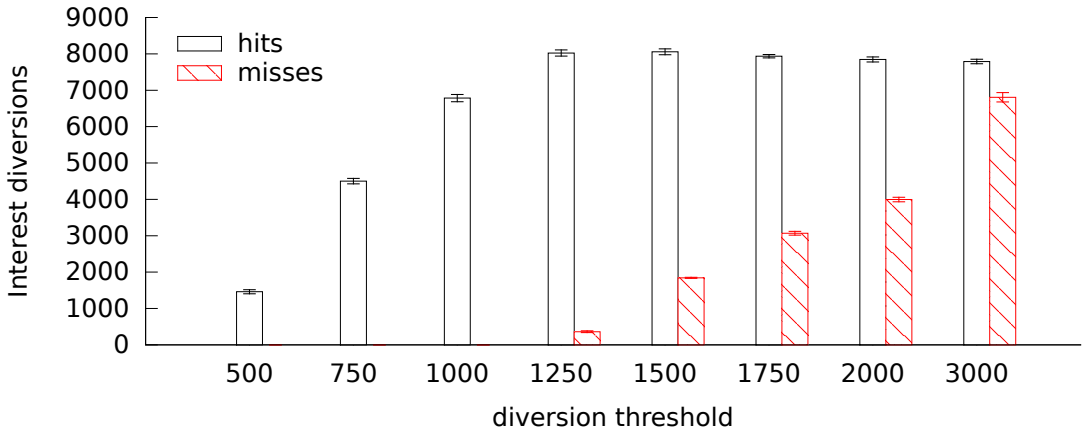
while "misses" indicate how many diverted Interests missed an off-path cache and were Nacked. A "hit" is desirable because, while it incurs a small processing overhead within the LAN, the significantly larger delay of sending an Interest onto the Internet is avoided. A "miss", on the other hand, incurs internal processing overhead and adds a small, but undesirable, delay before Data is retrieved from the Internet.

In scenarios with smaller diversion thresholds, the number of cache misses were negligible. When the diversion threshold approached the CS capacity 2000, a significant number of cache misses began to appear, because it was more likely that some Internet Data has been evicted from the CS by a local Data. For example, if $A1$ fetches a local Data from $A2$ and causes an Internet Data to be evicted from A's CS,

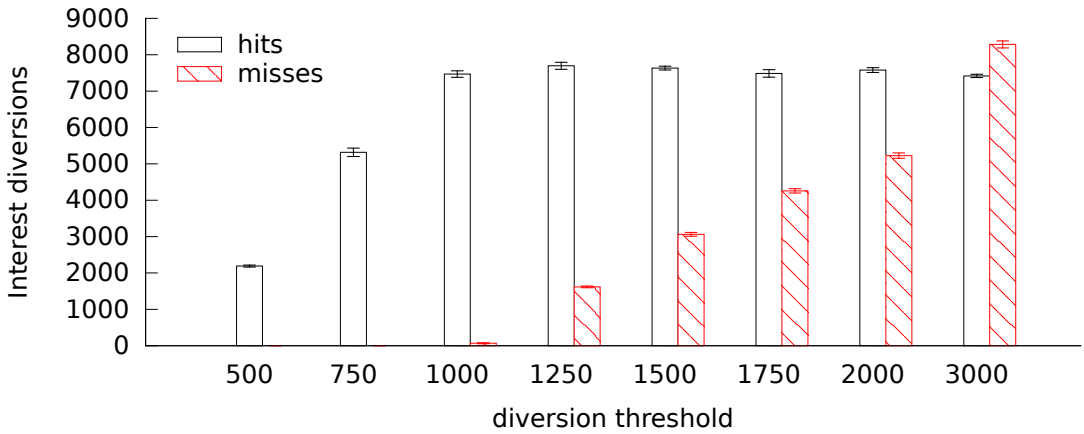
R will not know of this eviction and may still divert an Interest toward A, causing a cache miss. This is evident when comparing cache misses among different local traffic ratios: there were more cache misses when the network had more local traffic. When the diversion threshold was set to 3000, exceeding the CS capacity, there was a significant increase in cache misses. Therefore, taking both WAN connection utilization and internal processing overhead into account, for best performance, the diversion threshold should be set to a value around 60$\sim$75% of the CS capacity.



(a) 20% local traffic



(b) 50% local traffic



(c) 80% local traffic

Figure 4.10: Interest diversion hits and misses

CHAPTER 5

NDN-NIC: NAME-BASED FILTERING ON NETWORK INTERFACE CARD

In Named Data Networking (NDN), Interest packets carrying content names are sent to retrieve contents rather than to a particular destination identified by an address. This model fits shared media particularly well, where a single transmission is received by every node within range. For example, when a node requests a web page, any node with the content can reply, without the requester having to figure out or specify a particular receiver. It makes content retrieval more efficient, more scalable, and more fault-tolerant.

In shared media, however, there are also many packets that a receiver wants to filter out because they are not of interest to the receiver. In wireless networks, for instance, transmitted signals are heard by every node within radio range. Traditionally packet filtering is conducted at the network interface card (NIC) based on the destination MAC address (Section 2.1.4), which is not applicable to NDN traffic because NDN packets do not carry source or destination address. The current implementation of NDN over Ethernet is to multicast NDN traffic to all NDN nodes, and let NDN Forwarding Daemon (NFD) conduct name-based filtering in user space. Compared to filtering on NIC, user-space filtering not only incurs significant overhead to the main CPU, but also increases system power consumption because the system can be awoken by irrelevant packets. Although it is possible to treat the shared medium as a group of point-to-point links, put unicast MAC addresses into NDN packets, and utilize NIC's address filtering to reduce the overhead, doing so will lose the benefits of NDN and fall back to the address-based network architecture.

In this chapter, I propose **NDN-NIC**, a network interface card that can filter out irrelevant NDN packets based on their names, in order to reduce CPU overhead and system power consumption while keeping NDN's benefits The main technical

challenge is how to support scalable name-based filtering using only a small amount of on-chip memory. A typical NDN end host may have hundreds of thousands name rulesets due to the content it serves, the Interests it sends, and the content cache it maintains. In NFD they are stored as table entries in Forwarding Information Base (FIB), Pending Interest Table (PIT), and Content Store (CS), which could take up gigabits of memory [48]. On the other hand, a conventional, low-cost NIC only has tens of kilobytes of on-chip memory. It is infeasible to fit all these rulesets as-is into such as small amount of memory.

I propose to store these rulesets as a compact representation in three Bloom filters (BFs) on NDN-NIC and conduct packet filtering based on these BFs. A unique property of BF is the guarantee of no false negative, which means NDN-NIC will never drop any packet that should be accepted. However, as BF has false positives, some unwanted packets may pass the filter and be delivered to the system and later dropped by NFD. It is essential to choose carefully which names or name prefixes should be stored in the BF, to keep NDN-NIC's false positives as low as possible. It is also important to minimize the CPU overhead in maintaining the data structures, to maximize the overall savings in CPU usage and power consumption.

I have designed and evaluated a number of techniques that can reduce the number of names stored in BFs but may make name matching less accurate. The overall result is reduced memory footprint with acceptable false positives and no false negative.

I evaluated the design of NDN-NIC by running a real-world file access traffic trace, and simulating packet filtering and BF maintenance at each end host. Using an NDN-NIC equipped with 2.53KBytes memory, my design can reduce CPU usage by 93.96% compared to a regular NIC.

5.1 Background

This section first reviews the three tables used in NDN forwarding, including their name matching rules and implementation, where the challenges of name-based fil-

tering stem from. Then, I briefly introduce Bloom filters, the primary data structure used in NDN-NIC design.

5.1.1 Name Matching in NDN

The NDN Forwarding Daemon (NFD) makes forwarding decisions based on the name rulesets stored in FIB, PIT, and CS (Section 2.2.3). Both Interest and Data forwarding procedures [1] perform name matching in these tables, as summarized below.

For an incoming Interest with name `/A/b`: (a) NFD queries the CS for any Data whose name starts with `/A/b`. This could match `/A/b`, `/A/b/1`, `/A/b/1/2`, `/A/b/3`, etc (Section 2.2.2). (b) NFD inserts a PIT entry for the Interest if it does not already exist, which involves an *exact match* for `/A/b`. (c) NFD performs a *longest prefix match* lookup on the FIB. This could match FIB entries `/A/b`, `/A`, or `/`.

For an incoming Data with name `/A/b/1`: (a) NFD queries the PIT to find all pending Interests this Data can satisfy. This would match any PIT entry whose name is a prefix of `/A/b/1`, such as `/`, `/A`, `/A/b`, and `/A/b/1`. (b) If the Data satisfies a PIT entry, NFD puts the Data into the CS, which involves an *exact match* lookup.

In NFD implementation, FIB, PIT, and CS are stored on the **name tree**, a tree structure that follows the name hierarchy (Figure 5.1). Each node is identified by a name, and can attach FIB/PIT/CS entries with that name; each edge goes from a shorter name to a longer name with one more name component. Having the name tree allows the three tables to share a common index structure, which not only reduces memory consumption, but also saves computation overhead during table lookups. For example, after finding/inserting a PIT entry for an Interest, to perform a longest prefix match on the FIB for the same Interest, NFD can start from the name tree node containing the PIT entry, walk up the tree until finding a name tree node that has a FIB entry, instead of starting over from the root node.

[1] NDN-NIC does not currently support Nack packets.

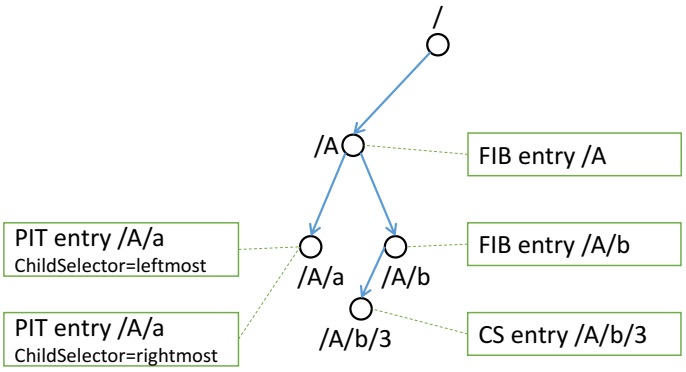


Figure 5.1: NDN-NIC overall filtering accuracy

5.1.2 Bloom Filter

Bloom filters (BFs) [49] are widely used in network processing to implement constant-time set membership testing with efficient memory usage.

A Bloom filter is defined by allocating an array of m zero-initialized bits, and selecting k independent hash functions each takes a key as input and returns a number within range $[0, m-1]$. To add a key to the BF, the algorithm computes k hashes of the key using the k hash functions, gets k positions in the array corresponding to the hashes, and sets the bits at these positions to "1". To query whether a key is in the BF, the algorithm computes k hashes of the key using the same hash functions, and checks the bits at chosen positions. If any of these bits is "0", the key is definitely not in the set; otherwise, the key is considered to be in the set. However, these bits could have been set to "1" due to other added keys, resulting in a **false positive**.

After adding n keys into an m-bit BF with k hash functions, its false positive probability [50] is:

$$p \approx \left(1 - \mathrm{e}^{-kn/m}\right)^k \tag{5.1}$$

This equation tells us that, given a BF with fixed size, when more keys are added (higher n), there would be more false positives. On the other hand, with same number of keys, a larger BF (larger m) has less false positives than a smaller BF.

One cannot remove keys from a standard BF, because it is unknown how many keys have caused a bit to be set to "1". Counting Bloom filter (CBF) [50] overcomes

this shortcoming by replacing each bit in the BF's array with a counter. Initially all counters are initialized to zero. The algorithm increments the counter when adding a key hashed to an array position, and decrements the counter when removing that key. To query whether a key is in the CBF, it suffices to check whether the counters at array positions chosen by the hash functions are all non-zero. A CBF with m counters has the same false positives as an m-bit BF, but it requires more memory than the standard BF: if using 8-bit integers as counters, the memory footprint is eightfold.

5.2 Design Overview

Name-based filtering in NDN brings three new challenges compared to address-based packet filtering. First, unlike most IP hosts that only have a handful of unicast/multicast addresses, an end host on the NDN testbed can have hundreds of FIB and PIT entries, and hundreds of thousands of CS entries; a consumer grade NIC has only tens of kilobytes of memory, and cannot fit a copy of all those names. Second, IP and Ethernet addresses of an end host are relatively stable, but name rulesets in the three tables, especially PIT and CS, are changing frequently, which requires NDN-NIC to be updated accordingly. Last but not least, Ethernet packet filtering only needs exact match, but NDN adopts different name matching rules for each table (Section 5.1.1), which requires NDN-NIC to support the same semantics on name-based filtering.

I believe that Bloom filter (BF) is an ideal data structure for NDN-NIC. Benefit from its efficiency in memory, hundreds of thousands of variable-length names can be stored in a BF that only takes a fixed amount of memory that is present on the NIC. BF guarantees there is no false negative, so that every non-matching packet can be safely dropped. On the other hand, BF brings possibility of false positives. In case a packet passes the hardware filter due to BF false positive, NFD will drop the packet because it does not match any table. This incurs some CPU overhead, but causes no correctness concern.

BFs do not support key removal (other than clearing the BF and rebuilding it from scratch), which is necessarily when a table entry is deleted. Counting Bloom filters (CBFs), in contrast, support key removal, but consume more memory because every single bit is replaced with a multi-bit counter. NDN-NIC utilizes a hybrid usage of BFs and CBFs: NDN-NIC maintains CBFs in software, and mirrors their contents into standard BFs in hardware. Therefore, the design can support key removal without increasing memory requirements on the NIC.

BF queries are exact match membership tests. NDN name matching not only uses exact match queries, but also has two types of prefix match queries:

- Entry-key prefix match (EPM): a name in the table is a prefix of the name of an incoming packet;
- Search-key prefix match (SPM): the name of an incoming packet is a prefix of a name in the table.

As introduced in Section 5.1.1, EPM is used to query incoming Interest names on the FIB and to query incoming Data names on the PIT; SPM is used to query incoming Interest names on the CS. Furthermore, incoming Interests are matched against FIB and CS but not PIT, while incoming Data are matched against PIT but not FIB or CS. Therefore, each of the three tables needs a separate BF, and each BF needs a different lookup method. BF-FIB and BF-PIT store FIB or PIT entry names, and implement EPM by querying the BF with every name prefix of an incoming Interest/Data. BF-CS stores all prefixes of CS entry names, and implements SPM by querying the BF with the incoming Interest's name. This technique allows us to support different name matching rules as per NDN semantics.

The overall architecture of an NDN host equipped with NDN-NIC is depicted in Figure 5.2. The NDN-NIC design consists of two components: the **NDN-NIC hardware**, a network interface card that filters incoming NDN packets (Section 5.3), and the **NDN-NIC driver**, an NFD module that notifies hardware what to filter (Section 5.4).

The NDN-NIC hardware performs name-based filtering based on the packet filtering logic and three BFs, BF-FIB, BF-PIT, and BF-CS. When a packet arrives,

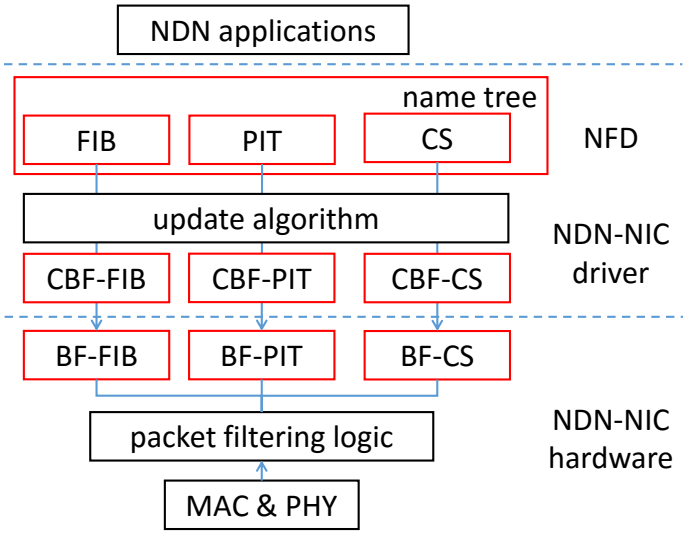


Figure 5.2: Overall architecture

the packet filtering logic queries the BFs with its name, and makes a deliver/drop decision. The NDN-NIC driver maintains three CBFs in response to changes in FIB, PIT, and CS, and then updates hardware BFs accordingly.

Note that it is unnecessary and invalid to add all table names into the BFs.

- The FIB contains both entries pointing to local producers, and entries pointing to the network (most notably, the `/` entry which represents the default route). It is unnecessary for NDN-NIC to accept an Interest matching a FIB entry in the second category, because an Interest received from a network interface would be forwarded out of the same network interface. Therefore, a FIB entry name is added to the NIC only if it has a next hop other than NDN-NIC itself. In case of a multi-homed host with multiple NICs, a FIB entry name should be added to an NDN-NIC if it has a next hop pointing to either a local producer or another NIC.

- The PIT contains both entries representing Interests expressed by local consumers and forwarded to the network, and Interests received from the network requesting Data from local producers. Only the former category of PIT entry names need to be added to the NIC. In case of a multi-homed host with mul-

tiple NICs, a PIT entry name should be added to an NDN-NIC if the Interest has been forwarded out of this NDN-NIC, and it is not yet satisfied or timed out.

- Unlike FIB and PIT, every CS entry name should be added to the NIC, because the cached Data can be used to satisfy an Interest from the network, regardless of whether this Data was previously received from the network or generated by a local producer.

After knowing what names to put on NIC, an update algorithm generates a compact representation of those names to be downloaded to the NIC. The update algorithm must be able to incrementally adapt to changes in table entries and update the CBFs, whose contents are mirrored into the BFs. I propose three update algorithms: *Direct Mapping*, *Basic CS*, and *Active CS* (Section 5.4). Direct Mapping simply adds every name in a table to the corresponding BF. Basic CS and Active CS were designed to reduce the number of names in the BFs so as to reduce BF false positives and improve overall filtering accuracy.

This overall architecture is a logical design. In practice, the name-based filter module, i.e., the BFs and packet filtering logic can either be deployed on the NIC itself or as a separate module between a conventional NIC and the host system. In this chapter, I take the former case to explain my design.

5.3 Name Filter on Hardware

The NDN-NIC hardware is a network interface card (NIC) that helps to offload NDN packet filtering. In addition to all the components of a regular NIC (PHY, MAC, USB/PCI, etc.), the NDN-NIC has three Bloom filters and a packet filtering logic to support name-based packet filtering.

NFD's FIB, PIT, and CS tables are represented as three BFs on the NDN-NIC (Figure 5.2). For each incoming packet, the packet filtering logic queries BFs differently according to the packet type. As introduced in Section 5.2, for an incoming Interest (e.g. `/A/b`), the NIC queries **BF-FIB** with all prefixes of the Interest

name (i.e. `/`, `/A`, and `/A/b`) and queries **BF-CS** with the complete Interest name (i.e. `/A/b`). For an incoming Data, the NIC queries **BF-PIT** with all prefixes of the Data name. The packet is delivered to NFD if any BF query finds a match; otherwise the NIC drops the packet.

The packet filtering accuracy of NDN-NIC depends on the characteristics of the Bloom filters, in particular: size of each BF, number of hash functions, and choice of hash functions. They are not only related to false positive probability, but also constrained by the hardware complexity and manufacturing cost. Section 5.5.3 evaluates different BF sizes and the number of hash functions.

NDN-NIC chooses to use *H3* hash functions [51], because they can be computed efficiently in hardware logic without requiring a specialized digest chip, and they are also simple enough for computing in software. H3 hash functions are incremental: in a hardware hashing circuit, after processing `/A` and computing its hash value, it is easy to compute the hash of `/A/b` by updating the existing hash value and internal register states with input `b`, without re-processing `/A`. This means, NDN-NIC can use k hashing circuits to compute k hash functions in parallel, and query the relevant BFs after reading each name component.

Although I did not build the NIC hardware, I anticipate that, with careful circuit design, the hardware can achieve high speeds with reasonable manufacturing cost. The BFs are compact enough to fit in fast SRAM or block RAM of a FPGA chip. Since NDN-NIC design requires frequent BF updates in order to keep in sync with NFD's tables, updates to the BFs should be implemented in the data plane as hardware logic, rather than in the control plane. NDN-NIC driver sends BF update commands over PCI/USB in the same way as sending outgoing packets, so that the hardware logic can process BF update commands as fast as they arrive.

5.4 NDN-NIC Driver

The NDN-NIC driver is a software module that keeps BFs on hardware up to date with NFD's FIB, PIT, and CS. Its main components are three counting Bloom filters

(CBFs) and an update algorithm.

Whenever NFD inserts or deletes an entry in FIB, PIT, or CS, the update algorithm adds or removes names in the CBFs as necessary. Updates to CBFs are then downloaded into BFs on hardware: a non-zero CBF counter value sets the corresponding BF bit to "1", and a zero CBF counter value clears the BF bit to "0". In this way, the BFs have exactly same matching results as the CBFs but smaller memory footprints. Updates to the BFs are applied incrementally: during a CBF update, the driver keeps track of which CBF counters have been incremented from zero to non-zero (corresponding BF bits should be set to "1") and which CBF counters have been decremented to zero (corresponding BF bits should be cleared to "0"), and then sends BF update commands to the hardware. The driver first performs 0-to-1 changes on BFs, and then performs 1-to-0 changes. This allows the hardware to continue processing incoming packets on partially updated BFs without danger of false negatives.

One concern is that frequent table updates potentially cause frequent hardware updates. Update commands would compete with outgoing traffic because the host cannot send outgoing traffic to the NIC when an update command is being processed. However, with the optimized algorithms proposed below, not every table update would generate a CBF update, and not every CBF update would result in a BF update on hardware. Section 5.5.6 evaluates BF update overhead.

In the following subsections, I propose one simple update algorithm, Direct Mapping, and two optimizations, Basic CS and Active CS.

5.4.1 Direct Mapping

Direct Mapping (DM) is straightforward: entries in each table are directly mapped to its corresponding BF. FIB entry names go into BF-FIB, PIT entry names go into BF-PIT, CS entry names and their prefixes go into BF-CS [2].

DM's filtering accuracy is reasonable when tables have a small number of en-

[2]More precisely, the names are updated into CBFs first and then downloaded into BFs on hardware. I omitted the CBF step to simplify the explanation.

tries compared to BF sizes, but it degrades quickly with larger tables. According to Equation 5.1, with perfect hash functions, a 65536-bit Bloom filter has a false positive probability of 4.33% after adding 10000 keys (names). This number quickly grows to 35.96% with 30000 keys, and exceeds 50% with 50000 keys. In practice, the false positive probability could be even higher due to imperfect hash functions.

As observed on the NDN testbed, a typical end host has small FIB and PIT, but a large CS. FIB and PIT are expected to have no more than a few hundred entries each. Suppose there are 100 programs running on the host, each serving 5 name prefixes, there would be 500 FIB entries. If 20 programs are actively being used, each having 40 pending requests (similar to today's web browsers), there would be 800 PIT entries. On the other hand, if the CS is allocated 1GB memory, assuming an average Data packet size of 4KB, the CS could contain 262144 entries. All these names need to be added to BF-CS; furthermore, in order to overcome BF's exact-match-only limitation, all their prefixes are added as well, which would require more than 500000 names in BF-CS. If BF-CS false positive probability is to be kept under 10% with so many names, 2404160 bits would be required, which will not fit in most NICs.

DM's computation overhead mainly comes from hash function computation. To reduce this overhead, the hashes can be computed once when it is added to a BF for the first time, and remembered on the name tree node, trading memory for CPU. DM also incurs high BF update overhead on hardware: CBF is updated on every table change, and unless the BF bits happen to be the same, almost all table changes require BF updates.

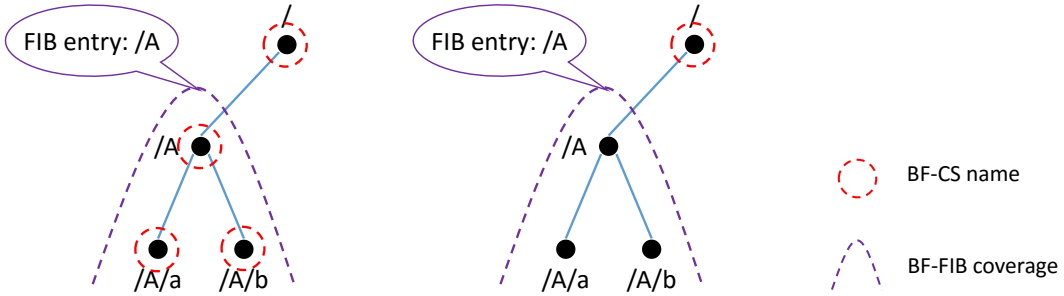
Observing that most names come from the CS, I propose two novel optimizations, Basic CS and Active CS, which add less names to BFs while still ensuring no false negative, so as to improve filtering accuracy and reduce BF update overhead.

5.4.2 Basic CS

Basic CS reduces false positives in BF-CS by not adding names under existing FIB entries, because Interests with those names are already being accepted because they

match BF-FIB.

As illustrated in Figure 5.3, suppose there is a FIB entry `/A`, and two CS entries, `/A/a` and `/A/b`. Both DM and Basic CS places `/A` into BF-FIB. Then, DM adds four names, `/A/a`, `/A/b`, `/A`, and `/` to BF-CS. Basic CS only adds `/` to BF-CS, because the other three Interest names match `/A` in BF-FIB and are already being accepted by NDN-NIC hardware.



(a) Direct Mapping (b) Basic CS

Figure 5.3: BF-CS usage, DM vs Basic CS

Basic CS is effective when most Data in CS came from local producers, since those Data names fall under FIB prefixes of these local applications which have been added to BF-FIB. In this case, only the prefixes shorter than the FIB entry name (such as `/` in the above example) are added to BF-CS, regardless of how many Data packets were cached. However, the CS also caches Data that were requested by local applications and received from the network; those Data would not match FIB entries in BF-FIB, and Basic CS would be ineffective in this case.

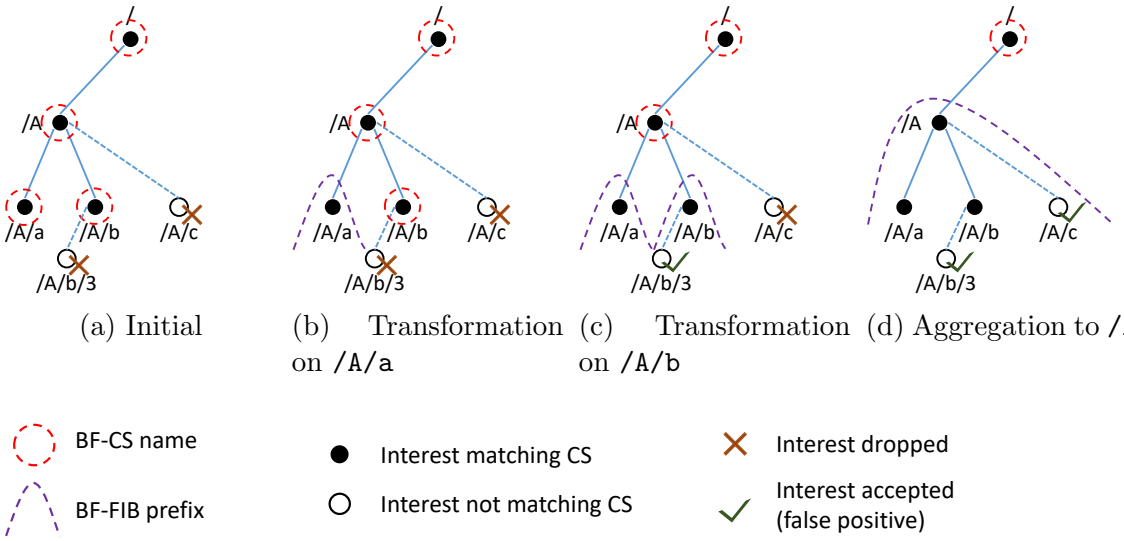
The extra computation overhead of Basic CS, compared to DM, is minimal. When NFD inserts a CS entry, it already needs to search the name tree to find the node where the CS entry should be stored. To implement Basic CS, the NDN-NIC driver just needs to check each name tree node along the way and see whether it also has a FIB entry, and stop adding names into BF-CS once it finds a FIB entry.

When a producer quits, its FIB entry would be deleted but Data generated by this producer remains valid in the cache. This can cause a larger overhead: at that time, the CS still contains Data packets under the former FIB prefix, but that prefix

is no longer in BF-FIB. Basic CS would have to re-add every CS entry name under the former FIB prefix into BF-CS. However, this occurs infrequently; when this situation does occur, names can be re-added to BF-CS asynchronously, and removal of the FIB prefix from BF-FIB should be delayed until re-adding is completed.

5.4.3 Active CS

Unlike Basic CS that passively waits to reuse FIB entries, Active CS actively creates the opportunity to reduce BF-CS usage by adding appropriate prefixes into BF-FIB, without requiring corresponding FIB entries.



(a) Initial (b) Transformation on `/A/a` (c) Transformation on `/A/b` (d) Aggregation to `/A`

Figure 5.4: Two basic operations in Active CS

Active CS has two basic operations: (a) **Transformation** replaces a single BF-CS name with a BF-FIB prefix, reducing BF-CS usage and increasing BF-FIB usage by the same amount; (b) **Aggregation** aggregates multiple BF-FIB prefixes up to a common shorter prefix in BF-FIB, decreasing BF-FIB usage. By repeatedly applying Transformation and Aggregation operations, the algorithm reduces BF-CS usage at the cost of a smaller BF-FIB usage increment. For example, in Figure 5.4, a CS with two entries `/A/a` and `/A/b` initially required four names in BF-CS. Transformations on `/A/a` and `/A/b` remove two names from BF-CS, and add two prefixes

to BF-FIB. Aggregation to `/A` replaces two BF-FIB prefixes with one shorter prefix in BF-FIB.

There is no false negative before or after any of these operations, because all CS entry names and their prefixes are matching at least one of BF-CS and BF-FIB. However, as shown in Figure 5.4, Interests `/A/b/3` and `/A/c` were initially rejected by the NIC's filter (Figure 5.4a), but they pass the filtering after Transformation and Aggregation have created BF-FIB prefixes `/A/b` and `/A` (Figure 5.4d), but NFD would drop them because they do not match any CS entry. The reason is that a BF-FIB prefix (using prefix matching) is less precise than BF-CS names (using exact matching). Therefore, while Active CS can reduce BF false positives, it introduces a new type of false positive, **prefix match false positive**, which is a semantic false positive caused by broad coverage of BF-FIB prefixes.

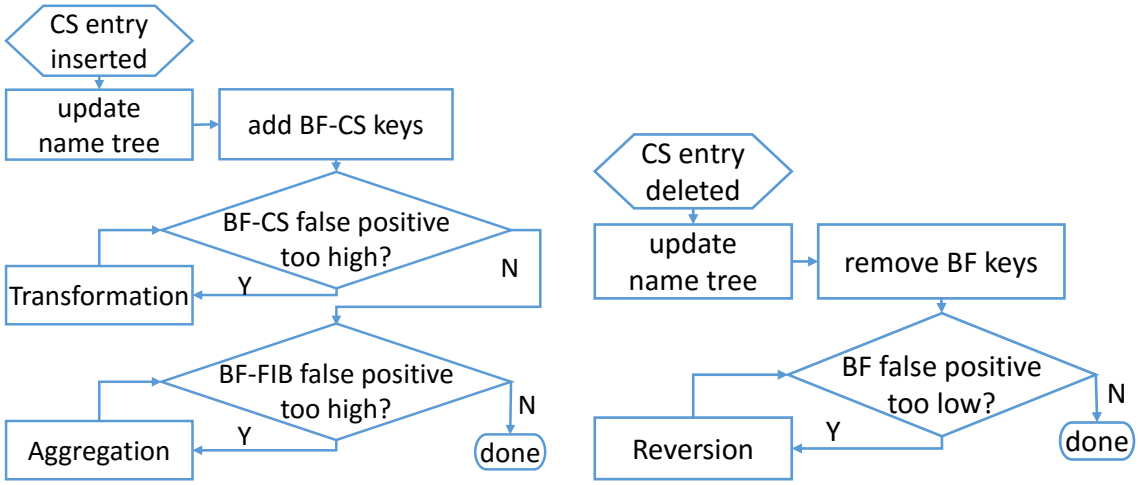
The goal of Active CS is to minimize overall false positives of both types. While BF false positive probability can be estimated with Equation 5.1, prefix match false positives are dependent on traffic and hard to predict. Active CS works around this problem with three strategies that answer three questions:

- When to do operations? It imposes an upper bound threshold on BF-CS/BF-FIB false positives, keeps as many names in BF-CS/BF-FIB as allowed, and doesn't do Transformation or Aggregation until BF false positives are too high.

- Where to do operations? Transformation and Aggregation are performed on the deepest eligible name tree node, to minimize the increment of prefix match false positives, under the assumption that a longer prefix in BF-FIB would match less irrelevant Interests. A name tree node is eligible for Transformation if its name is in BF-CS. A name tree node is eligible as a target of Aggregation (i.e. its name becomes the new prefix added to BF-FIB, while its descendants are removed from BF-CS/BF-FIB) if it has two or more descendants whose names are in BF-FIB.

- When to undo operations? When BF-CS/BF-FIB false positive probability falls below a lower bound, the algorithm reverses the two operations in an

effort to reduce prefix match false positives.



(a) CS entry insertion (b) CS entry deletion

Figure 5.5: Active CS flowchart

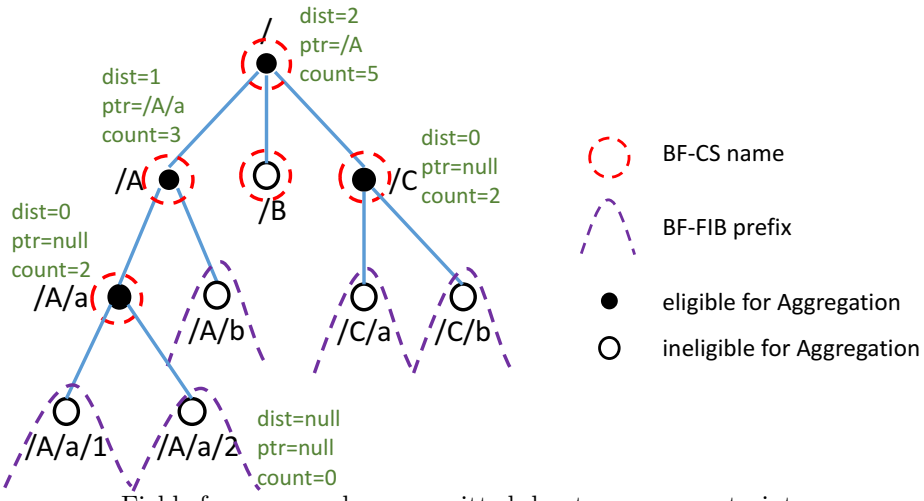
The workflow of Active CS is shown in Figure 5.5. When a CS entry is inserted, the name tree is updated accordingly, and names not already matched by BF-CS and BF-FIB are added to BF-CS as necessary [3]. Then, the NDN-NIC driver checks false positive probabilities of both BF-CS and BF-FIB. If either BF's false positive probability is over the upper bound, Transformation or Aggregation is applied repeatedly until both BF false positives fall under the upper bound.

When a CS entry is deleted, the name tree is updated accordingly; if a name tree node is deleted, its name is removed from BF-FIB and BF-CS. After that, the driver checks BF false positive probabilities. If both BF's false positive probabilities are below the lower bound threshold, the algorithm undoes Transformation and Aggregation in a **Reversion** operation. The shortest BF-FIB prefix is chosen as the target to maximize the potential reduction of prefix match false positives, because I assumed longer BF-FIB prefixes suffer from less prefix match false positives. It

[3]More precisely, the driver only updates CBF-CS in this step. BF updates are sent to NIC hardware at the end, so that names added to CBF-CS in this step but removed later by Transformation/Aggregation do not incur BF update overhead.

replaces the target BF-FIB prefix with a BF-CS name, and adds BF-FIB prefixes for the target's children in the name tree. To prevent oscillation between Transformation/Aggregation and reversion, the lower bound threshold should be configured to have sufficient distance from the upper bound.

Active CS is considerably more complex than DM and Basic CS. It needs an efficient algorithm design so that its computation overhead does not outweigh the savings on packet processing overhead. To quickly locate where to perform Transformation, Aggregation, or Reversion, Active CS maintains additional information in the name tree. For Transformation, every name tree node remembers the distance to the deepest eligible descendant, and a pointer to the child containing that descendant. For Aggregation, every node remembers similar distance and pointer fields to the deepest eligible descendant, and also keeps track of how many of its descendants are in BF-FIB to indicate whether a node itself is eligible (Figure 5.6 shows an example of fields maintained for Aggregation). For Reversion, every node maintains the distance to the shallowest descendant having a BF-FIB prefix, and a pointer to the child containing that descendant.



Fields for some nodes are omitted due to space constraint.

Figure 5.6: Name tree node fields for Aggregation

With those additional maintained information, Active CS can locate the desired node by walking down the name tree from the root following the pointers, which

is an procedure of $O(h)$ time complexity, where h is height of the name tree. To maintain this information, every time a CS entry is inserted or deleted, in the worst case, it is necessary to walk up the name tree and update all ancestors. At each node, the algorithm needs to find the maximum (for Transformation/Aggregation) or minimum (for Reversion) among distance fields on this node's children, and calculate the node's own distance and pointer fields accordingly. The time complexity of these updates after a CS insertion/deletion is $O(kh)$, where k is the degree of a name tree node.

While h is relatively small because it is bounded by the number of name components in a name, k represents the fan-out in the name hierarchy which can potentially be large. To keep the CPU overhead low, I introduce **degree threshold**: if the degree of a name tree node is over a certain threshold, a BF-FIB prefix is added for this name; its descendants are then covered by the BF-FIB prefix, and will not trigger any more updates in the distance fields and pointers. This change allows k to be bounded by the degree threshold instead.

However, imposing a degree threshold may add a name prefix with high irrelevant Interest traffic into BF-FIB, and significantly increase prefix match false positives. Therefore, the degree threshold is a trade-off between name tree updating cost and prefix match false positives. Section 5.5.5 evaluates two types of degree threshold settings.

5.5 Evaluation

I evaluated NDN-NIC with different Bloom filter settings and update algorithms using the NFS trace captured at Arizona Computer Science department network, and estimated the overhead in CPU usage and hardware BF updates. Simulation showed that, with 4096-bit BF-FIB, 16384-bit BF-CS, and 256-bit BF-PIT (2.53KBytes total size), NDN-NIC was able to reduce CPU usage by 93.96% compared to a regular NIC. The above saving was achieved with a small hardware overhead of less than 6% clock cycles spent on BF updates.

5.5.1 Simulation Setup

NDN-NIC simulation was conducted in two stages. First, I replayed a file access traffic trace on an emulated shared medium, and collected packet arrivals and table changes from each end host. Second, traffic and table traces were processed through a NDN-NIC simulator, to collect statistics of false positives, CPU usage, BF update overhead, etc.

In the first stage, I used the NFS trace captured at Arizona Computer Science department network. Appendix A describes how I collected the traffic trace, and the `nfs-trace-client` and `nfs-trace-server` programs that replay the traffic trace in an NDN-based file access protocol. All NFS servers and clients where placed on a shared medium where every node can hear from every other node; this differs from self-learning evaluation (Section 4.5.1) which used the real department topology. Each emulated host runs a modified version of NFD, which writes a *traffic and table trace* that logs every packet received from the shared media and whether it was accepted or dropped by NFD, and every change to the FIB, PIT, and CS tables.

I replayed 24 hours of NFS trace on a network emulated by Mininet [45]. Each hour of the trace was replayed individually. There were 2 NFS servers and between 16 and 46 NFS clients (see Section A.4), all placed on the same shared medium. This emulation step was executed only once, gathering traffic information as the input for NDN-NIC simulation. During the emulation, on each host, the FIB had only 1 entry (pointing to the local NFS server or client application, see Section A.3), the PIT had up to 1589 entries, and the CS had up to 28883 entries [4]. 41914117 packet arrivals and 2615648 table changes were logged among all hosts; among those received packets, 1924713 packets (4.59%) were accepted by NFD.

In the second stage, I processed the traffic and table trace from each host with a NDN-NIC simulator. This Python-based simulator contains two main modules:

- Simulated NDN-NIC hardware, which performs BF lookups upon packet arrival and decides whether to deliver or drop the packet. This decision is then

[4] Only table entries that should go into BFs are counted toward these numbers, see Section 5.2.

compared with NFD's decision: if the hardware delivers a packet but NFD dropped it, the simulator logs a false positive.

- NDN-NIC driver, which processes table change records from the trace, and performs BF updates as necessary. To quantify the overhead of such updates, the simulator logs how many bits are changed from "0" to "1" or from "1" to "0" during each table change.

Trace from each host in each hour was simulated in the NDN-NIC simulator separately. All simulations were repeated five times using different random polynomial terms for H3 hash functions. The plots in this section show the total across all hosts in 24 hours, averaged among five trials with error bars showing standard deviation where applicable.

This simulation setup does not directly measure CPU usage. Since the workload is mostly memory bound, I estimated CPU usage in terms of memory accesses: I counted how many name tree nodes were accessed during packet processing and name tree updating, and used that number to represent CPU usage.

5.5.2 Overall Filtering Accuracy

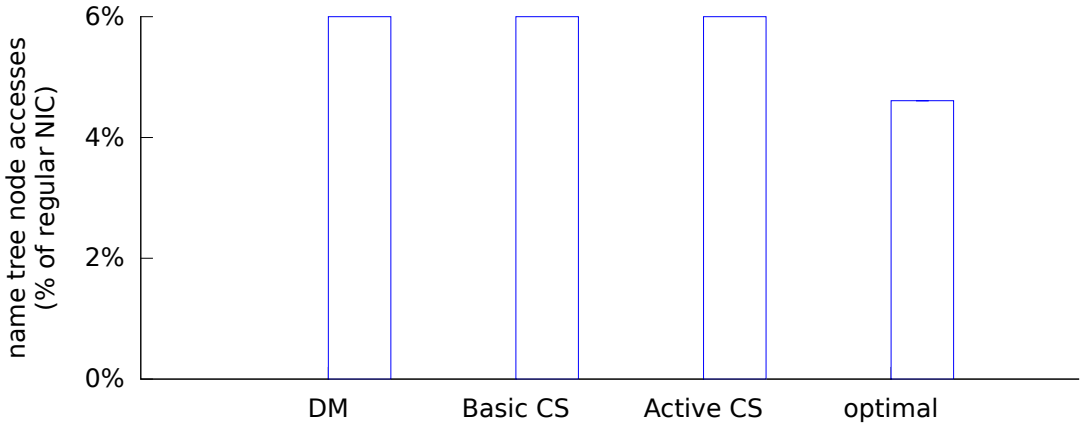


Figure 5.7: NDN-NIC overall filtering accuracy

Figure 5.7 shows the estimated CPU usage under three CS algorithms, compared to regular NIC at 100%. BFs were set to reasonable target sizes for cheap NICs:

4096 bits for BF-FIB, 16384 bits for BF-CS, and 256 bits for BF-PIT; each BF uses three H3 hash functions with random polynomial terms. Active CS parameters were set to achieve the best result for the NFS trace [5]. Reported CPU usage includes both packet processing and name tree updating cost. I chose CPU usage as the metric instead of percentage of delivered packets, because the ultimate goal of NDN-NIC is to reduce overhead of the main system, so the name tree updating cost of Active CS algorithm should not be neglected.

As shown in the plot, NDN-NIC reduced CPU usage by 92.72% with Direct Mapping, 93.28% with Basic CS, and 93.96% with Active CS. As a comparison, an optimal NIC would reduce CPU usage by 95.39%. No false negatives were found in these simulations.

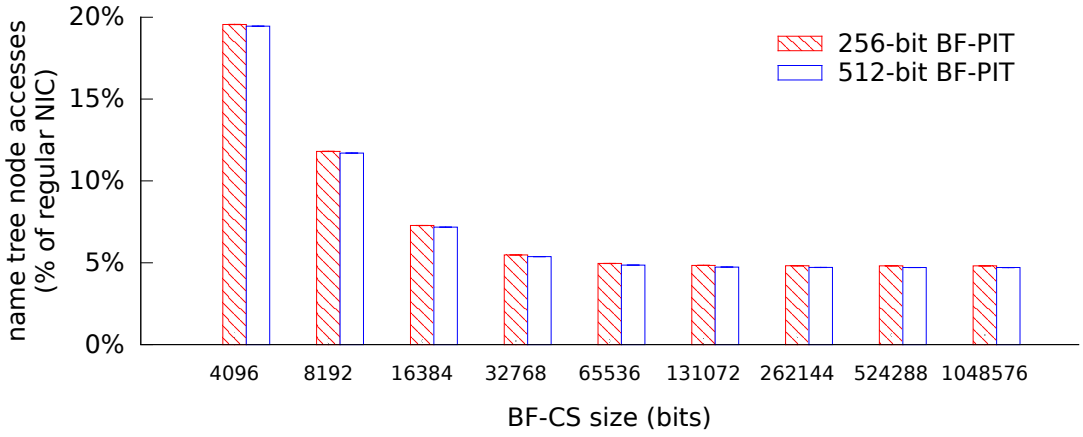
5.5.3 Bloom Filter Characteristics

Bloom filter size directly affects filtering accuracy of NDN-NIC. Figure 5.8 shows the CPU usage with different BF sizes using Direct Mapping algorithm. Each BF used three random H3 hash functions. With Direct Mapping, having larger BFs improved filtering accuracy significantly, because there were less BF false positives with the same number of names added. Since CS is the largest among the three tables, increasing BF-CS size gave the most benefit. A similar trend was observed with Basic CS algorithm.

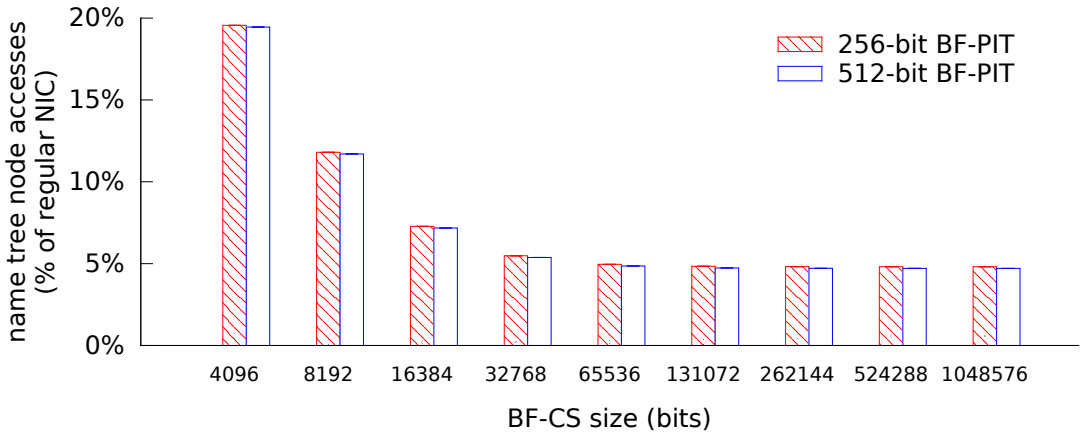
Figure 5.9 shows the same metric with Active CS algorithm. BF false positive threshold was set to 0.1% upper bound, and degree threshold was set to "/64/32/16" (see Section 5.5.5). While having larger BF sizes still improved filtering accuracy and reduces CPU usage, the gain of increasing BF-CS size beyond 65536 bits becomes marginal because many name tree nodes have exceeded the degree threshold.

The number of hash functions also affects BF false positive probability. Figure 5.10 shows the CPU usage when there were between 1 and 10 random H3 hash functions, under two CS algorithms and three BF size settings. The result illustrates that three hash functions was the best choice in my simulations. Having only one

[5] Active CS parameters: "/64/32/16" degree threshold, 0.1% BF false positive upper bound



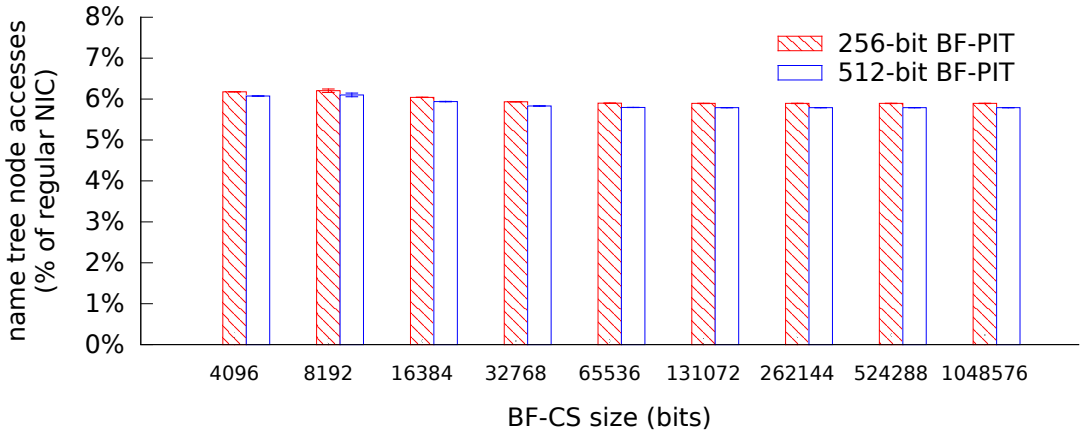
(a) 4096-bit BF-FIB



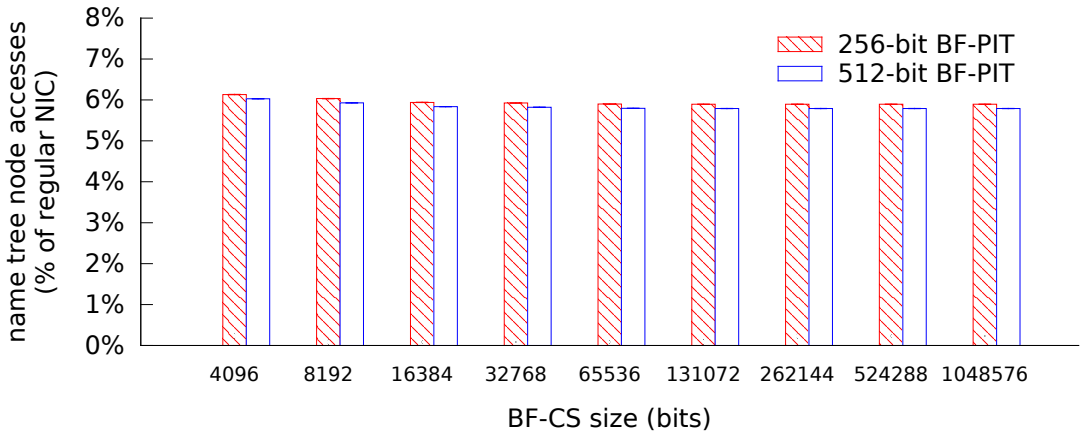
(b) 65536-bit BF-FIB

Figure 5.8: Effect of Bloom filter size, Direct Mapping

hash function degraded a BF into a hash table, and the filtering accuracy worsened significantly. Having more than three hash functions gave at most a marginal improvement on filtering accuracy, but that step would require more logic resources in hardware and more CPU time in software to compute these hashes (note that the CPU time for hash computation was not reflected in the "name tree node accesses" metric used as an estimation of CPU usage in this section). Although techniques such as double hashing [52] and segmented hashing [53] can keep the number of actual hash functions low but derive more hash values from the actual hash functions, the total entropy of those hash values is no more than the number of output bits



(a) 4096-bit BF-FIB



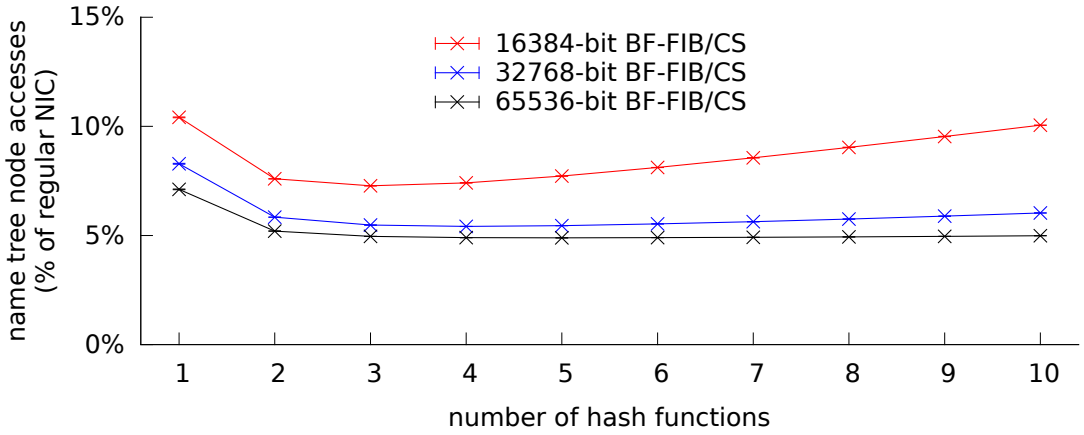
(b) 65536-bit BF-FIB

Figure 5.9: Effect of Bloom filter size, Active CS

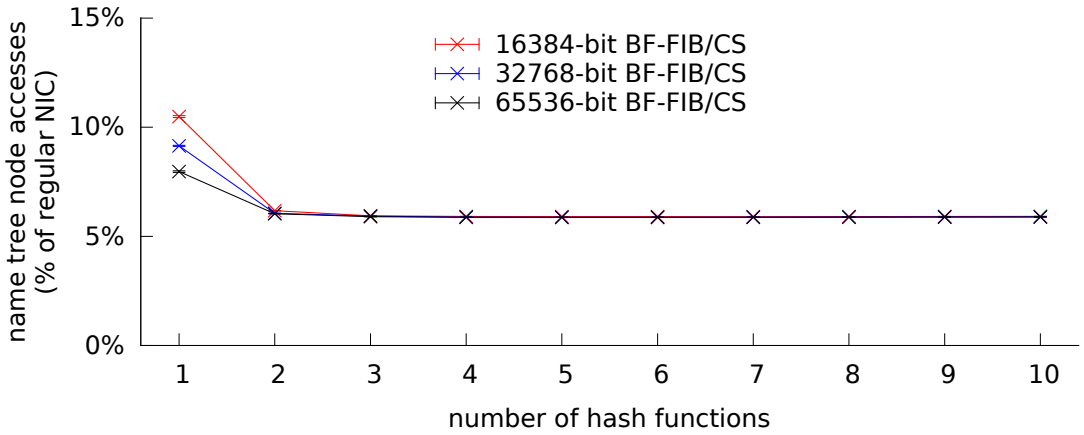
from the actual hash functions. Since each H3 hash function outputs just enough bits for indexing into the Bloom filter array (e.g. H3 outputs 16 bits for a 65536-bit BF), applying those techniques on H3 hash functions would create correlation among hash values and increase BF false positives.

5.5.4 Update Algorithms

In this section, I compare the CPU usage of regular NIC (no filtering), NDN-NIC with DM, Basic CS, and Active CS. Reported CPU usage includes both packet processing and name tree updating cost.



(a) Direct Mapping, 256-bit BF-PIT



(b) Active CS, 256-bit BF-PIT

Figure 5.10: Effect of number of hash functions

Figure 5.11 shows the estimated CPU usage with different BF-CS sizes; BF-FIB and BF-PIT were set to 4096 bits and 256 bits, respectively. Basic CS performed only slightly better than DM, because in NFS traffic trace, few CS entries were covered by FIB entries registered by local producers. Active CS performed much better than DM and Basic CS when BF-CS were small. However, with BF-CS size larger than 16384, Active CS had higher CPU usage than DM and Basic CS, because BF false positives were already low with little room for further reduction, and as a result, name tree updating cost exceeded the potential saving on packet processing. Considering the limited memory resources on NIC, Active CS was better

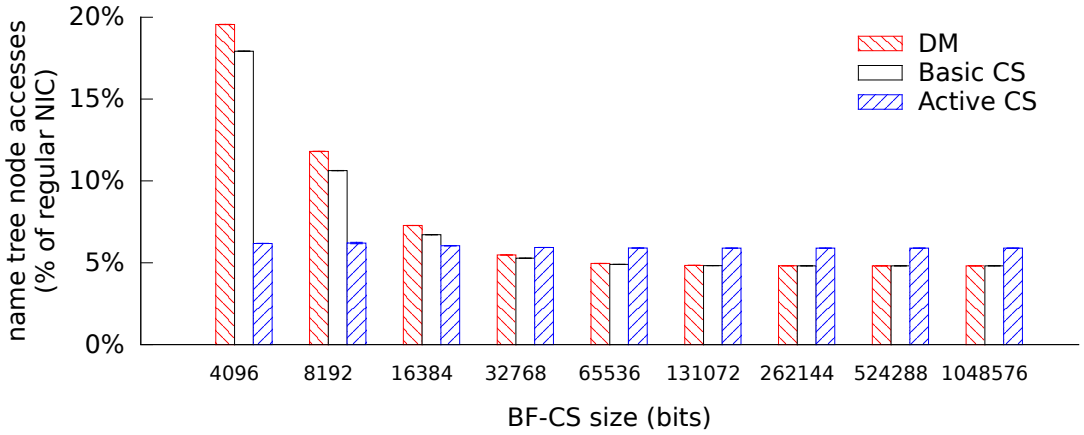


Figure 5.11: Comparison among update algorithms

than others.

5.5.5 Active CS Parameters

Active CS algorithm has two parameters, BF false positive thresholds and degree threshold. BF false positive thresholds control the BF false positive probability that can be tolerated in exchange for decreased prefix match false positives. Degree threshold limits the overhead of updating name tree nodes, but may increase prefix match false positives. I study the effect of these two parameters in this section Simulations this section used 16384-bit BF-FIB, 16384-bit BF-CS, 256-bit BF-PIT, and three random H3 hash functions in each BF.

Figure 5.12 shows the effect of degree threshold. In this test, BF false positive thresholds were set to 0.1% upper bound and an appropriate lower bound that prevents oscillation [6].

I have tried two types of degree threshold settings: (a) "fixed" settings use the same degree threshold for all names; (b) "name-length dependent" settings provide different degree thresholds at different name lengths.

Among fixed settings, higher degree threshold incurred somewhat lower packet

[6]The lower bound is set to be low enough so that at least two Reversions can be performed from a BF at upper bound before its false positive probability would reach the lower bound.

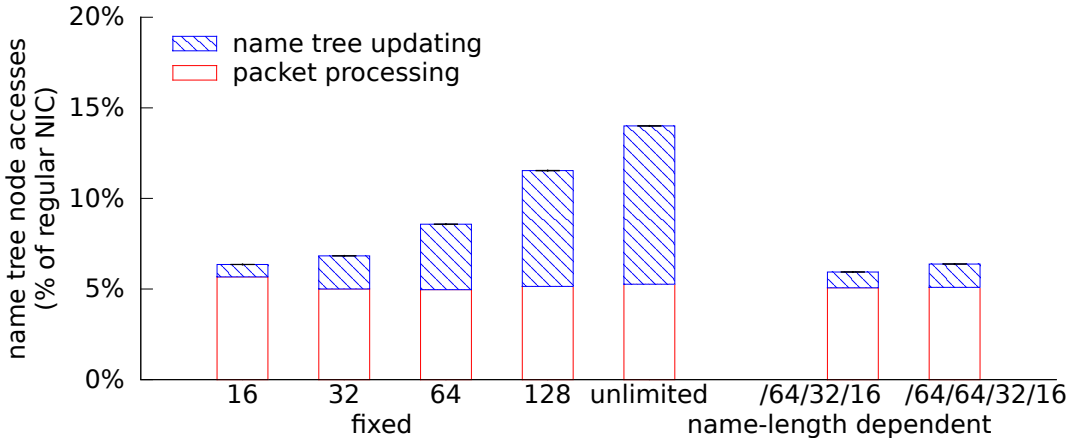


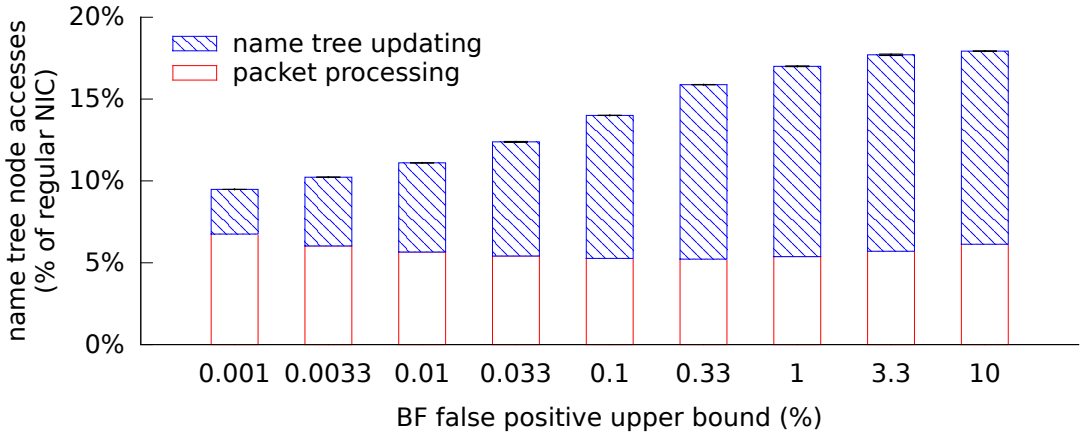
Figure 5.12: Effect of degree threshold

processing cost, because less BF-FIB prefixes were added due to exceeding degree threshold and thus less irrelevant Interests were accepted. On the other hand, it significantly increased name tree updating cost, since there were more nodes needing updates.

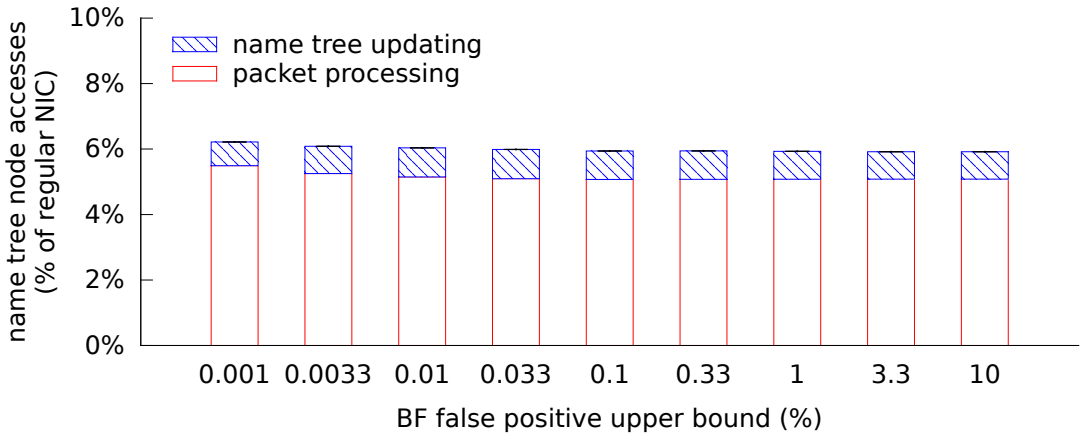
In name-length dependent settings, shorter prefixes were given a larger degree threshold, because adding a short prefix to BF-FIB causes more prefix match false positives than longer prefixes. For instance, "/64/32/16" assigns degree threshold "64" to the first name component, "32" to the second name component, and "16" to all other components. As shown in the plot, "/64/32/16" and "/64/64/32/16" had similar packet processing cost compared to "32" and "64" respectively, but their name tree updating costs were lower than fixed settings.

Figure 5.13 shows the CPU usage under different upper bound thresholds of BF false positives, under two degree threshold settings. There are two trends with unlimited degree threshold (Figure 5.13a):

- Name tree updating cost increased with higher BF false positive thresholds, because more names were kept in BF-CS, resulting in more nodes to update.
- Packet processing cost was minimized at 0.33% BF false positive threshold. Lower BF false positive thresholds caused more CS entry names to be aggregated onto shorter BF-FIB prefixes which increased prefix match false posi-



(a) Unlimited degree threshold



(b) Degree threshold set to /64/32/16

Figure 5.13: Effect of BF false positive thresholds

tives; higher BF false positive thresholds suffered from more BF false positives.

The trends are different with "/64/32/16" degree threshold (Figure 5.13b):

- Name tree updating cost stayed stable, because it is bounded by the degree threshold.
- Packet processing cost was minimized at 0.1% BF false positive threshold, although the differences from other settings were small.

Based on these results, I conclude that the degree threshold has bigger effect on the CPU usage of NDN-NIC, in which a name-length dependent setting is effective in reducing CPU usage; BF false positive thresholds, on the other hand, are less

important.

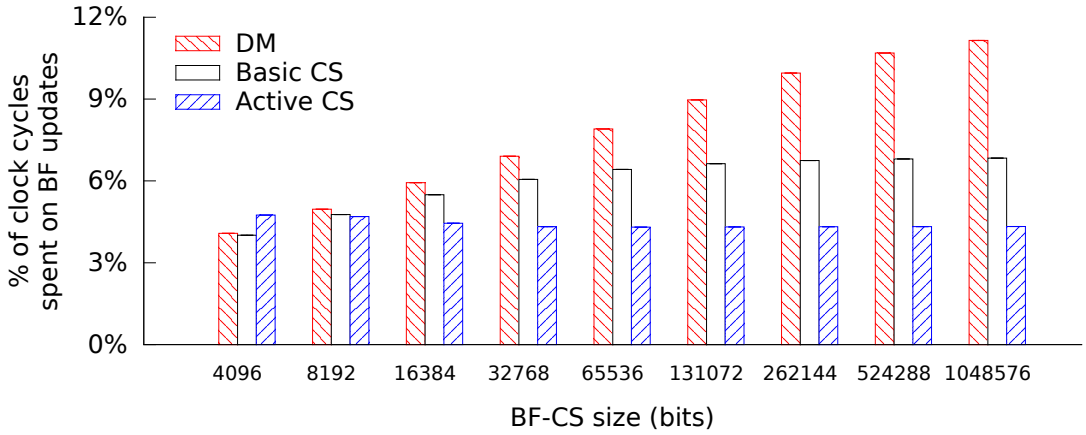
5.5.6 BF Update Overhead on Hardware

In this section, I quantify the impact of Bloom filter updates on hardware packet processing. I anticipate that BF updates are implemented as hardware logic in the data plane, rather than through control plane software which is much slower. A regular NIC can normally process one byte of outgoing traffic per clock cycle [54]. BF update commands share these clock cycles and compete with outgoing traffic, thus reduce outgoing bandwidth. Incoming traffic is processed in a separate logical unit and would not be affected by BF updates.

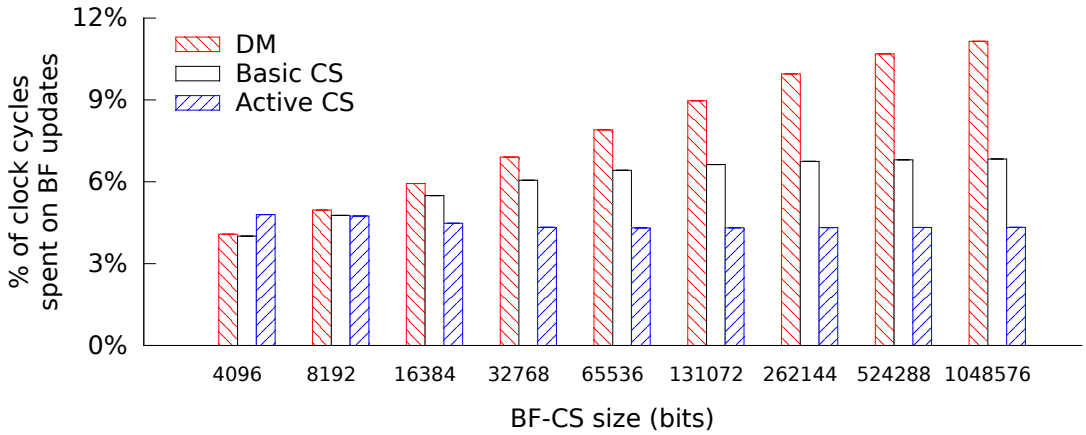
In the NFS trace, NICs sent a total of 1407885 Interest/Data packets. The total size of those outgoing packets was 1078MB and they consumed a total of 1078 million clock cycles. To estimate the extra clock cycles consumed by BF updates, I assumed each BF update needs 8 bytes (and thus 8 clock cycles), including 2 bytes to differentiate a BF update command from an outgoing packet and select which BF to update, 2 bytes as an address within the BF, and 4 bytes as the new value of the memory word [7]. I further assumed the worst case that every bit updated during a table change is in a different memory word, and therefore requires a separate update command.

Figure 5.14 shows the percentage of clock cycles spent on BF updates with different BF-FIB/BF-CS sizes and update algorithms. I can see that BF updates are responsible between 4.0% and 11.1% of consumed clock cycles, depending on BF sizes and the update algorithm. With DM or Basic CS, larger BF sizes tended to require more updates, because there were more bits in BFs so that when a name was added, the bit positions chosen by hash functions were less likely to be "1" already. Active CS incurred less BF update overhead than DM and Basic CS in most cases, because many CS entry insertions and deletions occurred under BF-FIB prefixes and did not require updates to BF-CS. As the only exception, when BF-CS was set

[7] A faster implementation is possible, but I assumed 8-clock-cycle BF updates to show an upper bound of BF update overhead.



(a) 4096-bit BF-FIB and 256-bit BF-PIT



(b) 65536-bit BF-FIB and 256-bit BF-PIT

Figure 5.14: BF update overhead: 3 hash functions, varying BF sizes

to 4096 bits, Active CS had higher BF update overhead than DM and Basic CS. Log analysis indicated that in this case, BF-CS quickly reached its false positive upper bound, causing Active CS to remove names from BF-CS and add them into BF-FIB, requiring updates in both BFs. Unlike DM and Basic CS, Active CS required less updates with larger BF sizes, because under the same BF false positive threshold setting, having larger BF allowed BF-CS to accommodate more names before BF false positive reaches the threshold and some names have to be moved/aggregated to BF-FIB.

Figure 5.14 shows the percentage of clock cycles spent on BF updates with

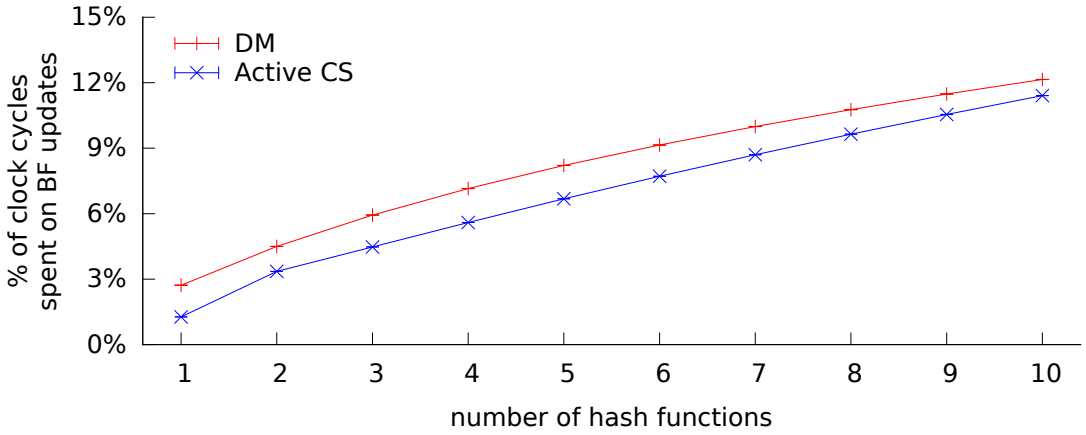


Figure 5.15: BF update overhead: 16384-bit BF-FIB, 16384-bit BF-CS, and 256-bit BF-PIT, varying number of hash functions

different number of hash functions. As expected, BF update overhead increased when more hash functions are used, because a larger number of hash functions would select more distinct bits to set to "1" when a name was added.

CHAPTER 6

NDN LINK PROTOCOL

In this chapter, I present the NDN Link Protocol (NDNLP), a link adaptation protocol that hides the difference among different underlying transports, and provides the forwarding plane a uniform transport service known as the *face*. I first introduce the motivation and design goals of NDNLP. Then, I define the NDNLP packet format, and present its two main features: hop-by-hop headers and fragmentation-reassembly. Finally, I illustrate the architecture of NFD's face system in which NDNLP is implemented.

6.1 Motivation and Design Goals

NDN is a universal network protocol for all applications and network environments [6]. NDN can work over all kinds of underlying transports, from physical links such as Ethernet and High-Speed Serial Interface (HSSI) cables, to logical tunnels over the existing Internet such as UDP, TCP, and WebSockets. Those underlying transports have different characteristics, for example:

- TCP and WebSockets can transport large messages reliably, while Ethernet and UDP only provide best-effort delivery and have limited packet size.
- WebSockets over TLS can guarantee the integrity of delivered packets and prevent packet injection, while others do not provide such guarantee.

NDNLP is a link adaptation protocol that hides the difference among underlying transports, to meet the needs of NDN forwarding plane. NDNLP provides best-effort delivery of network layer packets of reasonably large size. This delivery service, known as the *face*, is then used by NDN forwarding plane to implement content retrieval service.

Additionally, NDNLP offers the ability to attach hop-by-hop headers onto any

network layer packet. This capability is used to encode Nack packets, which are hop-by-hop in nature. It also allows NFD to accept special instructions from local applications as hop-by-hop headers, eliminating the need to define a separate set of "socket options".

NDNLP is designed to be a **unified protocol** that can be used on all kinds of underlying transports. Since different transports need different features, NDNLP ensures **all features are optional** and can be turned on or off per face. For example, Ethernet faces need the fragmentation-reassembly feature; TCP tunnels can already deliver large message, so that fragmentation-reassembly is turned off to avoid processing overhead.

Being a unified protocol, NDNLP must be extensible and allow different designs of a feature to be adopted per face. For example, a reliability improvement feature based on Automatic Repeat reQuests (ARQ) [55] is suitable for stationary wired networks, but Forward Error Correction (FEC) [56] may work better in highly mobile vehicular networks.

6.2 Packet Format

As a unified protocol, NDNLP needs a packet format that is extensible and works on all kinds of underlying transports. NDNLP adopts a Type-Length-Value (TLV) packet format, similar to how Interest and Data packets are encoded [7].

The NDNLP packet, or **LpPacket**, is structured as follows:

```
LpPacket ::= LP-PACKET-TYPE TLV-LENGTH
               LpHeaderField*
               Payload?

Fragment ::= FRAGMENT-TYPE TLV-LENGTH
               byte+
```

The **Fragment** field carries (part of) a network layer packet. If fragmentation-reassembly feature is disabled, this field must carry a complete Interest or Data

packet. Otherwise, this field is constructed and interpreted according to the rules defined in the fragmentation-reassembly feature.

LpHeaderField represents a header field. An LpPacket can have zero or more header fields, each encoded as a TLV element. NDNLP features can define new header fields by extending the definition of LpHeaderField. For example, Section 6.4 defines SeqNo, FragIndex, and FragCount headers for use in fragmentation-reassembly feature.

TLV-TYPE numbers of the header fields are allocated from a centralized registry, so that each header can be interpreted unambiguously. The least significant bit of a TLV-TYPE number indicates whether a header field is "critical" or "non-critical". A header field is critical if that bit is "1", otherwise it is non-critical. A node can safely ignore an unrecognized non-critical field in a received LpPacket, but must drop the LpPacket if there is an unrecognized critical field. For example, (a) fragmentation related fields are critical, because a node cannot correctly reassemble the network layer packet if it does not recognize the fragmentation scheme; (b) link-layer acknowledgement fields are non-critical, because a node can still processing the network layer packet contained in an LpPacket after ignoring the acknowledge, although it would not benefit from reliability improvements provided by the acknowledgement. This design provides backwards compatibility: when a new feature is designed, it may choose non-critical TLV-TYPE numbers for its header fields, so that old nodes that do not recognize these fields can still processing the rest of LpPackets.

Since all NDNLP features are optional, all header fields are optional as well. When no NDNLP header field is needed on an Interest or Data packet, it can be transmitted directly without encapsulating into an LpPacket. This reduces encapsulation overhead, and provides backwards compatibility with other nodes or applications that do not support NDNLP.

The Fragment field is also optional, so that information in header fields can be sent even when there is no network layer packet to transmit. This is particularly useful for sending link-layer acknowledgements.

6.3 Hop-by-Hop Headers

In several occasions, NDN forwarding plane needs to attach hop-by-hop headers onto network layer packets. Adding hop-by-hop headers into Interest/Data packets at network layer is undesirable or even unfeasible. An intermediate node could add hop-by-hop headers into an Interest, but it has to re-encode the Interest packet, resulting in performance overhead. An intermediate node cannot add or remove fields in a Data, because the Data is protected by a signature and any modification would invalidate this signature (Section 2.2.4).

LpPacket is a good place for attaching hop-by-hop headers. NDNLP headers are not covered by the Data signature, so that changes in hop-by-hop headers would not cause signature validation errors. LpPacket is always encoded and decoded at every hop, so that re-encoding does not incur extra overhead.

LpPacket currently can carry the following hop-by-hop headers:

- **Nack** indicates Nack reason (Section 6.3.1).
- **CongestionMark** signals congestion [44].
- **Ack** acknowledges receipt of an LpPacket [43].
- **PrefixAnnouncement** announces producer's prefix in NDN self-learning (Section 4.1.3).
- **NoDiversion** prevents an Interest from being diverted (again) to an off-path cache (Section 4.4.2).
- **IncomingFaceId**, **NextHopFaceId**, and **CacheControl** allow local applications to have advanced control on packet forwarding and caching behavior (Section 6.3.2).

These fields are encoded as follows:

```
LpHeaderField ::= .. | Nack | CongestionMark | PrefixAnnouncement |
                       IncomingFaceId | NextHopFaceId | CacheControl
```

```
Nack ::= NACK-TYPE TLV-LENGTH
           NackReason?

NackReason ::= NACK-REASON-TYPE TLV-LENGTH
                 nonNegativeInteger

CongestionMark ::= CONGESTION-MARK-TYPE TLV-LENGTH
                     nonNegativeInteger

Ack ::= ACK-TYPE TLV-LENGTH
          nonNegativeInteger

PrefixAnnouncement ::= PREFIX-ANNOUNCEMENT-TYPE TLV-LENGTH
                         Data

NoDiversion ::= NO-DIVERSION-TYPE TLV-LENGTH(=0)

IncomingFaceId ::= INCOMING-FACE-ID-TYPE TLV-LENGTH
                     nonNegativeInteger

NextHopFaceId ::= NEXT-HOP-FACE-ID-TYPE TLV-LENGTH
                    nonNegativeInteger

CachePolicy ::= CACHE-POLICY-TYPE TLV-LENGTH
                  CachePolicyType

CachePolicyType ::= CACHE-POLICY-TYPE-TYPE TLV-LENGTH
                      nonNegativeInteger
```

6.3.1 Nack

Nack is a network layer packet, flowing from upstream to downstream, that indicates an Interest cannot be satisfied (Section 2.2.2). A Nack packet contains the Interest that cannot be satisfied and optionally a reason. Nack packets are hop-by-hop in nature: a node does not forward a received Nack, but can generate its own Nacks toward downstream after receiving Nacks from upstream.

This hop-by-hop nature fits NDNLP's ability of carrying hop-by-hop headers pretty well. Therefore, instead of introducing a new encoding format at network layer, NDNLP attaches a **Nack** field to turn an Interest into a Nack.

The Nack reason, if supplied, is nested inside the Nack field. NDNLP uses a nested **NackReason** TLV element rather than putting the reason code directly in the Nack field, so that a future protocol revision may allow Nack to carry other information as additional elements nested under the Nack field.

As an example, a Nack with reason "NoRoute" against Interest `/A/B` is encoded as:

```
LpPacket TLV-LENGTH=27
  Nack TLV-LENGTH=5
    NackReason TLV-LENGTH=1
      NoRoute
  Fragment TLV-LENGTH=16
    Interest TLV-LENGTH=14
      Name TLV-LENGTH=6
        NameComponent TLV-LENGTH=1
          'A'
        NameComponent TLV-LENGTH=1
          'B'
      Nonce TLV-LENGTH=4
        0xf3684f82
```

6.3.2 Local Fields

At the end host, TCP/IP network stack treats local applications and network links differently: local applications are "above" the network stack and communicates with the stack via a set of function calls known as the *socket API*; network links are "below" the network stack and communicates with the stack via device drivers. In NDN, end hosts are full NDN nodes and have the same forwarding capability as routers. From forwarding point of view, there is no fundamental difference between a local application and a network link: they are both treated as faces.

However, certain local applications may want advanced controls on the forwarding and caching behavior of their packets. In TCP/IP network stack, such controls are achieved via *socket options*. For example, the `IP_MULTICAST_LOOP` option gives an application control on whether packets sent to a multicast group should be looped back to the local sockets. In NDN stack, the same controls are encoded as hop-by-hop headers in LpPacket, where "hop" means the app-NFD connection. Having these fields in NDNLP eliminates the need of define a separate set of socket options, and allows NFD to treat the connection to a local application as a face.

There are three fields defined for advanced forwarding and caching control:

- **IncomingFaceId** tells application on which face did the packet arrive.
- **NextHopFaceId** allows consumer to control where an Interest should be forwarded.
- **CacheControl** allows producer to prevent a Data from being cached in local Content Store (CS).

They are used in the implementation of NFD Management protocol [23], among other applications. These fields are only permitted between applications and local NFD; putting them on a network link would cause the LpPacket to be dropped.

6.4 Fragmentation-Reassembly

Most underlying transports have Maximum Transmission Unit (MTU) limits. For example, Ethernet frames are typically limited to 1500 octets; UDP packets are limited to 65507 octets due to the 16-bit IPv4 total length field. To send packets larger than MTU size, NDNLP provides a hop-by-hop fragmentation-reassembly feature.

6.4.1 Why Hop-by-Hop Fragmentation?

In IPv4 [57], the sender can transmit packets up to 65536 octets (the maximum possible value in the 16-bit total length field). When a packet encounters a link with small MTU, the router (re-)fragments it into multiple smaller packets. The fragments are reassembled by the receiver at the final destination.

In IPv6 [58], the sender must first discover the smallest MTU over the path to the destination (Path MTU), and fragment data from higher layers (in TCP or application) to fit the Path MTU. IPv6 routers do not re-fragment packets enroute. The fragments are reassembled at the final destination.

Unlike IPv4 or IPv6, NDNLP uses hop-by-hop fragmentation and reassembly. When a packet encounters a link with small MTU, the node fragments it into multiple LpPackets. The fragments are immediately reassembled at the next hop, rather than at the final destination. Reassembling at each hop is required because (a) an Interest cannot be partially processed, because a node must know the entire name to make forwarding decisions; (b) a node must have the full Data in order to match it with pending Interests; (c) caches should hold full Data packets to satisfy Interests from consumers with disparate MTU sizes.

The concept of Path MTU does not apply to NDN due to caching and asynchronous consumption of Data packet. The only way for a producer to pre-size its Data to fit every link MTU is to constrain the Data size to the smallest possible MTU size among all networks. Obviously, doing so is extremely inefficient when the Data does not actually traverse links with the smallest MTU. Therefore, it is

recommended that applications should be aware of the prevalent MTU sizes in the network and produce Data segments accordingly, but the task of sending Data over links with smaller MTUs should be handled by NDNLP's hop-by-hop fragmentation and reassembly [59].

6.4.2 Header Fields

The fragmentation-reassembly feature defines three new header fields:

```
LpHeaderField ::= .. | SeqNo | FragIndex | FragCount

SeqNo ::= SEQ-NO-TYPE TLV-LENGTH
            fixed-width unsigned integer

FragIndex ::= FRAG-INDEX-TYPE TLV-LENGTH
                nonNegativeInteger

FragCount ::= FRAG-COUNT-TYPE TLV-LENGTH
                nonNegativeInteger
```

When a network layer packet is fragmented into multiple LpPackets, those LpPackets are assigned consecutive sequence numbers. The relative position of a fragment and the total number of fragments in the network layer packet are reflected by FragIndex and FragCount fields. The receiver can use these three fields to reassemble the packet.

6.4.3 Sender Operation

To send a packet of L octets on a link with MTU, the sender first decides maximum payload length MPL, such that the total length of a resulting LpPacket will not exceed MTU. Then, the packet is sliced into $N = \lceil \frac{L}{MPL} \rceil$ fragments. N LpPackets are created and assigned consecutive sequence numbers, FragIndex from 0 to $N-1$,

and have N as the FragCount. The payload itself is placed in the "Fragment" field. Each of the first $N-1$ LpPackets contains MPL octets of payload, and the last LpPacket contains the remaining part.

For example, with $MPL = 1000$, a 2500-octet packet is sent as the following 3 LpPackets:

SeqNo	FragIndex	FragCount	Fragment
600	0	3	0-999
601	1	3	1000-1999
602	2	3	2000-2499

If the Interest/Data packet needs to carry hop-by-hop headers, these headers are added to the first LpPacket, so that the receiver can process them upfront. Hop-by-hop headers occupy space in the first LpPacket and reduce the number of bytes available for the payload. To accommodate those headers, the sender reduces the size of payload in the first LpPacket. As an example, a 2500-octet packet with 300-octet hop-by-hop headers may be fragmented as:

SeqNo	FragIndex	FragCount	Fragment
700	0	3	0-699
701	1	3	700-1699
702	2	3	1700-2499

Note that the current protocol does not support fragmenting hop-by-hop headers. If there were so many hop-by-hop headers that they themselves exceed the MTU size, NDNLP would not be able to send them.

6.4.4 Receiver Operation

The receiver maintains a **partial message store** data structure. Each entry in the store represents a partially received network layer packet. Entries are indexed by *message identifier*, which is the sequence number of the first LpPacket, and can be calculated from any LpPacket by subtracting FragIndex from SeqNo. Each entry

contains a list of received fragments. It also has a timestamp of latest LpPacket arrival, so an incomplete message without new LpPackets coming can be cleaned out after some time.

When an LpPacket with SeqNo S, FragIndex I, and FragCount N is received, the receiver first calculates message identifier $M = S - I$. If entry M does not exist in the store yet, it is created. The current fragment is added into the entry, unless a fragment with the same FragIndex I is already in the entry. When the entry has all the fragments (as indicated by FragCount), the receiver reassembles the network layer packet from the received fragments and delivers it to upper layer, and removes the entry from the store.

For example, if the packets in Section 6.4.3 are received in the order of 601, 600, 601, 602, the following will happen:

1. **601 arrival** $M = S - I = 601 - 1 = 600$. Create an entry in partial message store with message identifier 600. Save the payload of the LpPacket in the entry.

2. **600 arrival** $M = S - I = 600 - 0 = 600$. An entry with message identifier 600 exists in the partial message store. Save the payload of the LpPacket in the entry.

3. **601 arrival** $M = S - I = 601 - 1 = 600$. An entry with message identifier 600 exists in the partial message store. Since fragment $I = 1$ has already arrived, this is a duplicate and is dropped.

4. **602 arrival** $M = S - I = 602 - 2 = 600$. An entry with message identifier 600 exists in the partial message store. Save the payload of the LpPacket in the entry. Now the partial message has all the fragments, so the network layer packet is reassembled and delivered to upper layer, and the entry is removed from the store.

As a performance improvement, if $N = 1$, the network layer packet was not

fragmented, so it can be delivered right away without going through the partial message store.

6.5 NFD Face System Architecture

NDNL is implemented in NFD face system. NFD face system provides a face abstraction upon which the forwarding plane can send and receive network layer packets. This face abstraction is implemented on top of various underlying transports, including (a) physical links such as Ethernet and Bluetooth, (b) overlay tunnels such as UDP and WebSockets, and (c) inter-process communication channels such as Unix sockets. Similarity and differences among these transport types influence the design of face system architecture, as described below.

```
FaceSystem
|
|--ProtocolFactory for Ethernet
|  |--Face on eth0
|  |--Face on eth1
|
|--ProtocolFactory for WebSockets
|  |--Channel on ws://192.0.2.1:9696/
|     |--Face to 192.0.2.2:6961
|     |--Face to 192.0.2.3:27933
|
...
```

Figure 6.1: NFD face system architecture

The face system adopts a four-tier hierarchy, illustrated in Figure 6.1:

1. The top tier, **FaceSystem**, interfaces with the forwarding plane and NFD management. It is the entry point of the face system.

2. A **ProtocolFactory** controls faces of a particular underlying protocol. For example, there is a ProtocolFactory for Ethernet, and one for WebSockets.

3. A **Channel** owns a local endpoint, which can be put in listening mode in order to hear packets from new remote nodes and create unicast faces accordingly. This tier is present only if the underlying transport is unicast-only.

4. A **Face** implements the face abstraction for use in the forwarding plane.

Each individual **Face** is composed of two parts:

- The "upper" part, **LinkService**, implements NDNLP to translate between network layer packets and LpPackets.

- The "lower" part, **Transport**, wraps the underlying communication mechanism (such as sockets) and exposes a best-effort delivery service for LpPackets.

This architecture makes NFD face system extensible. An experimenter can easily add a new link service feature by only modifying LinkService while keeping other parts intact. Similarly, one can add a new transport type by implementing new subclasses of ProtocolFactory, Channel, and Transport; it does not require modifications in LinkService, forwarding, or management.

However, this extensibility comes at the cost of complexity. First, the separation between LinkService and Transport necessitates copying packet buffers, which degrades performance. Second, it is difficult for the forwarding plane to utilize unique features in a particular transport type (e.g. measuring the receiving signal strength of a wireless interface), because the Transport abstraction only exposes features common to all transport types.

CHAPTER 7

RELATED WORK

This chapter summarizes related work on Future Internet Architectures which NDN is one of them, designs of NDN forwarding and caching, broadcast-based self-learning, NDN table lookup algorithms, and Bloom Filters in networking.

7.1 Future Internet Architectures

Named Data Networking (NDN) is one of five Future Internet Architecture (FIA) [60] research efforts to develop, explore, and evaluate "new and improved" network architectures to carry the Internet into the future. Each of these five efforts explore a key architecture thrust.

- NDN [1] makes content as first-class.
- MobilityFirst [61] builds mobility from the ground up. They cleanly separate names from addresses, and rely upon a distributed name service to bind names and addresses. MobilityFirst also enhances security by representing both names and addresses in an intrinsically verifiable manner.
- NEBULA [62] enhances cloud computing. They build ultrareliable routers with an extensible control plane that can enforce arbitrary policies, to provide resilient networking services that are dependable, secure, flexible, and extensible.
- eXpressive Internet Architecture (XIA) [63] focuses on built-in extensibility. They unite multiple clean-slate network architectures with different principals (such as host, content, services, or users), and can evolve to accommodate new, as yet unforeseen, principals over time.

- ChoiceNet [64] factors in economic incentives. They develop an "economy plane" for the Internet that enables network providers to offer new network-based services (QoS, storage, etc.) for sale to customers, and enables users to select among service alternatives.

NDN is one of many proposed Information-Centric Networking (ICN) [65] architectures. ICN aims at evolving the Internet infrastructure to directly support data-centric and location independent communications by introducing uniquely named data as a core Internet principle. Major ICN architectures include Data-Oriented Network Architecture (DONA) [24], NDN, Content Centric Networking (CCN) [2], Publish Subscribe Internet Technology (PURSUIT) [66], Network of Information (NetInf) [67], COntent Mediator architecture for content-aware nETwork (COMET) [68], and CONVERGENCE [69]. Research challenges in ICN [70] include naming, security, routing scalability, mobility, wireless networking, congestion control, in-network caching, network management, and applications.

7.2 NDN Forwarding and Caching

7.2.1 Forwarding

Cheng [71] defined a stateful and adaptive forwarding plane for NDN. NDN routers maintain state information of pending Interests. This state information, coupled with the symmetric exchange of Interest and Data, enables NDN routers to detect loops, observe data retrieval performance, and explore multiple forwarding paths, all at the forwarding plane. Cheng presented a concrete design of NDN's forwarding plane to make the network resilient and efficient. Most importantly, Cheng invented the Nack packet type, which is a useful signal for an upstream to inform the downstream the unavailability/unreachability of requested content. Cheng's work has influenced the NDN forwarding behavior specification presented in Chapter 3 of my dissertation, and the fast failure recovery mechanism in NDN self-learning (Section 4.3.3).

7.2.2 Broadcast-based Self-Learning

The idea of self-learning existed long before ICN. It dates back to "Hot-Potato Heuristic Routing" [72] proposed by Paul Baran in 1964: a network node observes traffic, and estimates distance back to the source node of each packet based on hop count carried in the packet; a packet is forwarded along the shortest available path toward its destination. In modern networks, a similar idea was implemented in switched Ethernet: switch learns the location of hosts by observing traffic; it floods frames if it doesn't know the location of the destination.

Listen First, Broadcast Later (LFBL) [73], *sCDN* [74], and *Reactive Optimistic Name-based Routing (RONR)* [4] are early works that incorporate self-learning in ICN. All three are designed for wireless networks with various degree of dynamics. LFBL, despite being a data-centric protocol, learns the location of an address rather than a name prefix; an Interest would be flooded unless it carries the producer's address and the network knows where this address is. sCDN protocol exposes a publisher/subscriber interface; flooding happens in the control plane, while the data plane follows the forwarding paths computed by the control plane. RONR protocol is closer to my NDN self-learning design which uses name-based forwarding and data plane flooding, but it does not have accurate granularity knowledge and thus incurs unnecessary flooding, and cannot recover from a link failure until learned FIB entry expires.

7.2.3 Off-Path Cache Utilization

Caching is a unique feature of ICN. Normally, only on-path caches are utilized. Methods of utilization off-path cache have been proposed for Internet service provider (ISP) networks.

CCN-FOH [75] explores one-hop neighbors of every on-path node in addition to forwarding the Interest via the known path. This scheme is simple: it does not require remembering any information about potential off-path caches. However, it has high overhead because the Interest is forwarded to every node one hop away

from on-path nodes. It is effective in reducing latency when Data is found in an off-path cache, but it cannot find Data cached in an off-path node two or more hops away. Also, this scheme does not stop the Interest from being forwarded toward the original path, so it cannot save wide area network (WAN) connection bandwidth when applied to local area network (LAN).

In [76], when an incoming Interest matches the record of off-path caches, it is forwarded to a potential off-path cache. This scheme has lower overhead than *CCN-FOH* at the expense of recording which nodes are potential off-path caches, and it can also utilize off-path caches more than one hop away. However, this design cannot save WAN connection bandwidth when applied to LAN, because in most cases, the Interest is still forwarded toward the FIB next hop, without waiting for an answer from the off-path cache. On the other hand, in order to limit bandwidth overhead, their design attaches some *quotas* on the Interest and lets every node decide whether to spend them. This idea is present in NDN self-learning design as the "no diversion" tag.

S-BECONS [77] lets on-path nodes remember which downstream has most recently retrieved a file along with a timestamp of that retrieval, and decides whether to divert an incoming query or not based on whether the timestamp is recent enough. This scheme is very similar to NDN self-learning, but I argue that the traffic volume sent to a downstream is a better predictor than when the downstream last retrieved the Data, hence NDN self-learning's diversion threshold is based upon "other Data" count.

Fetching the Nearest Replica [78] requires routers to upload the names of their most popular content as Bloom filters (BFs) to a centralized Tracker Server. Consumer downloads those BFs; if the requested content name is in a BF, the Interest is encapsulated and sent to the corresponding replica. However, this method requires deploying a centralized Tracker Server and downloading BFs to every consumer, which significantly increases the protocol complexity.

7.3 NDN Table Lookup Algorithms

In order for NDN to be commercially viable, the forwarding engine must support wire-speed operations. This includes fast table lookup of variable-length names, efficient data structures to store millions to billions of names, and fast packet processing.

Cisco Systems designed a NDN router with hash tables [79]. The design is optimized for a software router, and they optimize the memory layout of hash buckets to be friendly to data cache in Intel CPUs. *SipHash* was chosen because of its extremely low hash collision probability. Their SipHash implementation can compute all name prefix hashes with one pass while simultaneously parsing the name components. This approach is similar to NDN-NIC's incremental hash computation (Section 5.3).

Cisco's design also includes a 2-stage Forwarding Information Base (FIB) lookup algorithm to improve the average FIB lookup time and make the worst-case FIB lookup time bounded and independent of input names. Their design, however, only support exact matching in Pending Interest Table (PIT) and Content Store (CS) lookup because they believe prefix matching is infeasible at scale.

Other than hash tables, FIB can be organized as a tree, and name components can be compressed or encoded to reduce memory consumption [80]. On high-end hardware routers, FIB can also be distributed onto multiple line-cards in order to support more entries [81].

Yuan [82] proposed a PIT compression method by replacing variable-length names in the PIT with fixed-length fingerprints. The PIT of a 100Gbps router was shown to be compressed down to as little as 37MiB which can fit into SRAM chips on a high-end router. This compression method is similar to NDN-NIC's BF-PIT (Section 5.3) which stores the hash values (i.e. fingerprints) of PIT entry names. However, Yuan's design does not support Interest selectors (Section 2.2.2). Although NDN-NIC's BF-PIT also does not process Interest selectors, BF-PIT is allowed to be imprecise because it is only a pre-filter: the worst case is a false posi-

tive. On the other hand, not processing Interest selectors in NFD's PIT may cause incorrect forwarding decision if Interests of same name but different selectors were received from multiple downstream nodes.

7.4 Bloom Filters in Networking

BFs [49] are building blocks widely used in IP networking, such as peer-to-peer network applications, resource/packet routing and measurements [83]. Fan et al. [50] exploited a hybrid usage of counting Bloom filter (CBF) and BF, applying it to web cache sharing, i.e., each web cache tracks its own cache contents with a CBF, and broadcasts the corresponding standard BF to other caches, dramatically reducing both local updating cost and message traffic. Inspired by their approach, I maintain CBFs in NDN-NIC driver but download standard BFs into NDN-NIC hardware.

Dharmapurikar et al. [84] are the first to use BF to implement longest prefix match in IP forwarding. In NDN, longest prefix match is also needed for variable-length names in the FIB. So et al. [85] proposed to use hash tables combined with BF, and data pre-fetching for implementing the FIB. Wang et al. [86] proposed NameFilter, a two-stage BF-based scheme for fast longest name lookup in FIB. Quan et al. [87] proposed Adaptive Prefix Bloom filter, a scalable name lookup engine making hybrid use of BF and trie. Perino et al. [81] proposed a prefix Bloom filter scheme using GPU for FIB. Beyond FIB, Li et al. [88] proposed an enhanced implementation of PIT using mapping Bloom filter, reducing on-chip memory consumption. While most work focuses on NDN name lookup in software, NDN-NIC uses Bloom filters on hardware to achieve name-based filtering.

CHAPTER 8

CONCLUSIONS

This dissertation proposes a complete solution for Named Data Networking (NDN) deployment in local area networks (LANs).

I defined the NDN forwarding behavior, which specifies how each NDN node should process Interest, Data, and Nack packets, in order to implement content retrieval service. This protocol has been implemented in the forwarding plane of NDN Forwarding Daemon (NFD).

In this forwarding plane, I proposed a broadcast-based self-learning strategy to discover contents and forwarding paths securely and efficiently through occasional flooding in a switched Ethernet environment. I discussed two previously overlooked issues, namely, the FIB granularity problem and the trust model for prefix announcements. NDN self-learning can build forwarding tables in the data plane with low overhead, recover quickly from link failures, and efficiently utilize off-path caches for Internet data.

On a smaller LAN of a single-hop shared medium, I proposed NDN-NIC, a network interface card (NIC) that performs name-based packet filtering. NDN-NIC filters incoming packets on hardware by querying Bloom filters (BFs) maintained by a software driver, and drops irrelevant packets before they are delivered to NFD for software processing. This saves CPU cycles and power consumption in the main system. Although I did not build the NIC hardware, my design tackles NDN-NIC's main research challenge: using the limited amount of on-chip memory to support packet filtering based on a large number of rulesets.

Finally, I defined the NDN Link Protocol (NDNLP). It allows NFD to add hop-by-hop headers onto Interest/Data packets, and provides a fragmentation-reassembly feature to allow NDN to operate directly over Ethernet. I also presented an architectural design of NFD's face system in which NDNLP is implemented on

top of underlying transports including Ethernet and TCP/UDP tunnels, which provides the communication channel used by NFD's forwarding plane.

I hope this dissertation can encourage the adoption of NDN in local area networks.

APPENDIX A

THE NFS TRACE

Designs of NDN communication in local area networks (LANs) need to be evaluated with a traffic trace that represents LAN communication patterns. While there are a handful of publicly available traffic traces, they are unsuitable for the evaluations in this dissertation:

- University data center trace [89] only contains source and destination IP addresses, but does not indicate what content is being requested. NDN is a data-centric architecture, and requires a trace that indicates each piece of content.
- UC Berkeley Home IP Web trace [90] contains the content being requested, but the content identifier is represented as a digest without hierarchy. NDN uses hierarchical naming, and would not work efficiently with flat names such as digests.
- Boston University web client trace [91] gives complete hierarchical names, but the traffic volume is too low for my evaluations.

Therefore, I decided to collect my own traces.

Network File System (NFS) [8] is a common protocol in office networks of a university department. I captured 24 hours of NFS traffic in University of Arizona Computer Science department, and derived NDN-based file access traffic from the captured file names and packet timing. These traffic traces were used in the evaluations of NDN self-learning (Section 4.5.1) and NDN-NIC (Section 5.5.1).

A.1 NFS Traffic Capturing

The topology of Arizona Computer Science department network, as of Nov 2014, is shown in Figure A.1. There are two main file servers, *zuni* and *titan*. They host the

home folders of each computer science student, as well as folders for the department website and other projects. Each folder is served from only one server, and the client machines are configured such that they know which server is hosting the requested folder.

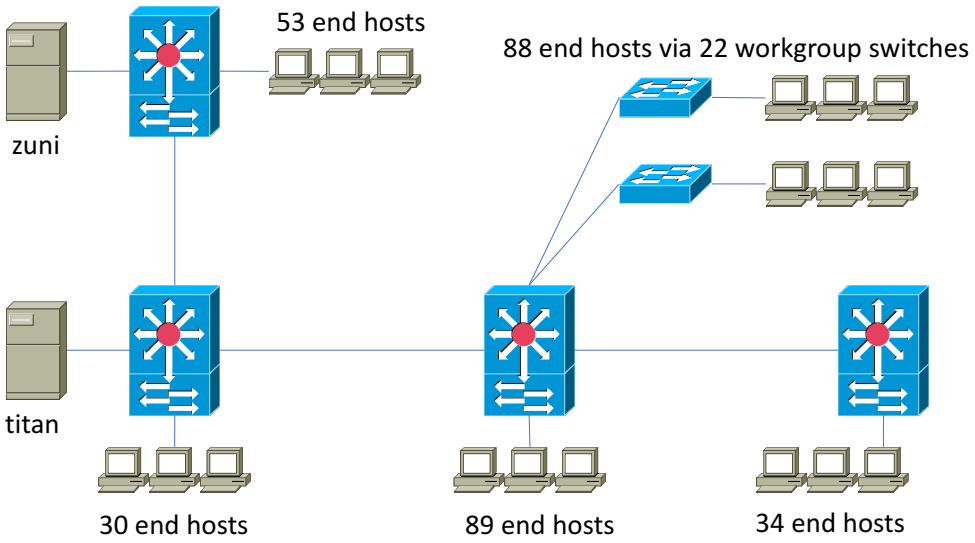


Figure A.1: Arizona Computer Science topology

With the help of lab staff, I captured traffic of *zuni* on Nov 12, 2014, and traffic of *titan* on Dec 03, 2014. Lab staff configured the Ethernet switch to copy the traffic on the file server's port onto a mirroring port, and directly wired that mirroring port to my computer. I was told that traffic mirroring is a low priority job and may skip packets if the switch is busy; the extent of such packet skipping was unknown. I `tcpdump`'ed all traffic coming from the mirroring port to pcap files, rotating to a new file hourly for *zuni* and every 15 minutes for *titan*. After a pcap file was written completely, a second program `nfsdump` [92] was invoked to parse the pcap file and extract any NFS operations seen in the traffic. Several modifications of `nfsdump` were needed to reassemble TCP segments, and to recognize two additional NFSv3 [93] operations used in our network.

The output from `nfsdump` contains the following information:

- Timestamp.
- Source IP address and port number.

- Destination IP address and port number.
- Transport protocol: TCP or UDP.
- NFS Remote Procedure Call (RPC) direction: call or response.
- NFS protocol version.
- NFS RPC function, such as `GETATTR`, `READDIRPLUS`, `WRITE`, etc.
- Additional fields from RPC message, such as file attributes, payload size, etc.

To respect privacy, the payload (file contents) was discarded, and file/directory names were anonymized as SHA-1 digests. The anonymization procedure replaces each name component as a SHA-1 digest, and preserves the name hierarchy. For example, the path `/home/shijunxiao/NDN/paper.pdf` would be anonymized as `/E83249BD3BA79932E16FB1FB5100DAFADE9954C2/B0AE086059D1B3E3C4BADC837FD0AC0249E3E257/868661A2D1D8AB8F27550D56D2A4279462EFBD54/866BB7B7664570903D8A4FF82E022374E169CB18`. Raw pcap files were deleted as soon as `nfsdump` finished processing; only the output described above is retained.

A.2 NFS Name Hierarchy Reconstruction

NFS [93] associates a *filehandle* with each file or directory. To read a file, the client first figures out the filehandle for the file, and then requests the file contents using the filehandle. For example, to access the file `/home/shijunxiao/NDN/paper.pdf`, the client makes four NFS procedure calls:

1. MOUNT the directory `/home/shijunxiao`, obtaining a filehandle for the mount point.
2. LOOKUP the subdirectory `NDN` under the mount point's filehandle.
3. LOOKUP the file `paper.pdf` under the subdirectory's filehandle.
4. READ the file using its filehandle.

The last READ procedure call, as seen on the network, only contains the filehandle, but does not carry the file name. In order to know which file is being read, it is necessary to find out where the filehandle comes from by inspecting previous procedure calls, and then reconstruct the full name.

I wrote a series of scripts to reconstruct the full names for each NFS operation.

1. `fhparent.js` extracts filehandle-name-parent relations from `nfsdump` output. From the above example, this script may output [1]:

   ```
   fh1, /home/shijunxiao, MOUNTPOINT
   fh2, NDN, fh1
   fh3, paper.pdf, fh2
   ```

2. `fullpath.cc` derives full path for each filehandle. It reads `fhparent.js` output as a directed graph representing a forest, where each filehandle becomes a node, each MOUNTPOINT record becomes the root node of a tree labelled with the mount point path, and each non-MOUNTPOINT record becomes an edge from the parent filehandle to the child filehandle labelled with the file name. Then, each tree is traversed so that every node is labelled with the full path constructed from the mount point's path and the file names on every edge.

   From the above example, this program may output:

   ```
   fh1, /home/shijunxiao
   fh2, /home/shijunxiao/NDN
   fh3, /home/shijunxiao/NDN/paper.pdf
   ```

3. `operations.js` reconstructs file access operations from `nfsdump` and `fullpath.cc` output. It creates a separate output file for each client IP. The last modified time of the file, if known from file attributes in NFS messages, is recorded as a "version" number, which will be used to derive NDN Data names.

   From the above example, this script may output:

[1] The actual trace contained SHA-1 anonymized names, but clear-text names are used here for illustration.

```
2014-11-30 12:00:00, lookup, /home/shijunxiao/NDN,
  version=1417373419
2014-11-30 12:00:01, lookup, /home/shijunxiao/NDN/paper.pdf,
  version=1417373073
2014-11-30 12:00:02, read, /home/shijunxiao/NDN/paper.pdf,
  version=1417373073, offset=0, count=3072
2014-11-30 12:00:03, read, /home/shijunxiao/NDN/paper.pdf,
  version=1417373073, offset=3072, count=2048
```

Note that MOUNT operation is skipped, because explicit mounting is unnecessary in NDN. LOOKUP operations, however, are preserved, because they allow the client to learn the file attributes.

4. `rwmerge.js` joins read/write operations on consecutive ranges of the same file into a single operation, and converts byte offsets into segment units where each segment is 4096 octets. This is because, in NDN, files are divided into fixed-size segments, and each segment is encoded as a Data packet. It would be inefficient to generate separate Data packets for each requested byte range, because such Data packets cannot be cached to satisfy future requests.

   From the above example, this script may output:

```
2014-11-30 12:00:00, lookup, /home/shijunxiao/NDN,
  version=1417373419
2014-11-30 12:00:01, lookup, /home/shijunxiao/NDN/paper.pdf,
  version=1417373073
2014-11-30 12:00:02, read, /home/shijunxiao/NDN/paper.pdf,
  version=1417373073, start=0, count=2
```

NFS clients can also locally cache filehandles of previously accessed files and directories. Many client computers in our department network are kept on 24x7, which means some filehandles for commonly accessed folders may come from a procedure call hours or days before the client READing the file. Since I was only able to

capture 24 hours of traffic from each file server, this also means that these procedure calls may not be in my trace. To solve this problem, I modified `fullpath.cc` so that it can identify tree roots that are not associated with a MOUNTPOINT node, assign paths like `/UNRESOLVED-fh8` to those roots, and then derive full paths of the nodes within those trees based on the root's path.

Finally, I overlapped NFS operations initiated by the same client on both file servers together. As mentioned earlier, traffic of *zuni* was captured on Nov 12, 2014, and traffic of *titan* was captured on Dec 03, 2014. I was unable to capture from both servers simultaneously due to hardware limitation, but I chose two Wednesdays on normal school weeks (not holidays), hoping that the file access traffic of these two days are representative. For each distinct client IP address, the `timeofday.sh` script sorts the NFS operations initiated by that client by time of day, and outputs a single file showing all operations regardless of which server. For example, if a client reads `/home/shijunxiao/NDN/paper.pdf` from *titan* at Nov 12 12:00:02, and writes `/cs/www/people/shijunxiao/index.html` to *zuni* at Dec 03 12:05:00, the output would look like (LOOKUP operations are omitted):

```
12:00:02, read, /home/shijunxiao/NDN/paper.pdf,
  version=1417373073, start=0, count=2
12:05:00, write, /cs/www/people/shijunxiao/index.html,
  version=1417633500, start=0, count=2
```

A.3 Deriving NDN-based File Access

I designed a NDN-based file access protocol that can support major operations from NFS. The operations are mapped into NDN as outlined in Table A.1.

With the exception of WRITE command, all Interests are expressed by the client and replied by the server. Data payload sizes were estimated from response sizes seen in NFS. The WRITE command, on the other hand, was designed as a three step process:

1. The client expresses an Interest `/NFS/<path>/./write/<client-host>:`

NFS operation	Interest/Data name	Data payload size (octets)
GETATTR LOOKUP	I: /NFS/<path>/./attr D: /NFS/<path>/./attr/<version>	84
READLINK	I: /NFS/<path> D: /NFS/<path>/<version>	160
READ	/NFS/<path>/<version>/ One Interest-Data exchange per segment.	4096
WRITE	see description in text	
READDIR-PLUS	First I: /NFS/<path>/./dir Subsequent I & D: /NFS/<path>/./dir/<version>/ Each Interest-Data exchange fetches up to 32 directory entries.	174 per entry
SETATTR CREATE MKDIR SYMLINK REMOVE RMDIR RENAME	/NFS/<path>/./<command>/<args>/<signature> where <args> is 32 octets representing command arguments, and <signature> has 4 name components totaling 115 octets.	248

Table A.1: NFS operations in NDN

<version>:<first-segment>:<last-segment>/<signature> to start writing. The server replies with Data containing 108 octets to acknowledge this command.

2. The server expresses Interests to retrieve segments from the client. The Interest name looks like /<client-host>/NFS/<path>/<version>/. The client replies to those Interests with Data containing 4096 octets.
3. After the client has received enough Interests, it sends another Interest /NFS/<path>/./commit/<client-host>:<version>:<first-segment>:<last-segment>/<signature> to finish writing. The server replies with Data containing 100 octets to acknowledge this command.

I implemented this protocol in a pair of programs, nfs-trace-client and nfs-trace-server. Instances of the server program were installed on two nodes in a network emulated by Mininet [45]. The server program takes as input a list

of name prefixes it should serve. This list was generated from the file paths served by *zuni* and *titan* as observed in the captured traffic. The emulated *zuni* serves 154099 prefixes, and the emulated *titan* serves 193829 prefixes. A server instance responds to an Interest only if the `<path>` portion of the Interest name falls under one of the prefixes served by this server; otherwise, it would remain silent. In NDN self-learning experiments, the server also attaches prefix announcements when replying to discovery Interests. The server program is otherwise stateless: the client indicates how the server should reply to an Interest using a reserved header field in the Interest, and the server responds accordingly.

Instances of the client program were installed on other nodes in the emulated network. The client program reads as input the file access operations initiated by this client, which came from `timeofday.sh` script. For each file access operation in the input, the client expresses the first Interest at the specified timestamp, and expresses subsequent Interests after receiving replies to earlier Interests. Each Interest may be retransmitted up to 3 times upon Nack or 4-second timeout; the file access operation is marked as failed if the retransmission limit is exceeded. The client program writes a log of each file access operations executed, including begin and end timestamps, full file/directory path, whether the operation was successful, and how many retransmissions were needed. In NDN self-learning experiments, when responding to a segment-fetching Interest during a WRITE command, the client program attaches a prefix announcement contain `/<client-host>/NFS` prefix if the Interest is tagged "discovery".

A.4 Statistics of NFS Trace

The number of file access operations and active clients in each hour of the day is shown in Table A.2.

Due to CPU and memory limitation of the emulation server, I selected a subset of client hosts to use in evaluations. A client was selected if it initiated between 200 and 2000 file access operations within the hour. Non-selected clients were not included

in the emulated network; in NDN self-learning experiments, a workgroup switch was included in the emulated network only if at least one client was connected to it. The number of selected clients and their traffic volume is shown in the "selected" columns in Table A.2.

Hour	NFS operations		Active clients	
	total	selected	total	selected
00-01	1955210	21034	140	40
01-02	2294409	12187	101	23
02-03	1943218	11723	93	21
03-04	2447286	8563	88	16
04-05	2196724	9069	92	18
05-06	1530638	10086	91	18
06-07	1015299	9587	89	17
07-08	839633	12050	137	21
08-09	912082	16298	102	25
09-10	1833637	14873	104	23
10-11	4099730	25401	104	30
11-12	4035717	11196	106	23
12-13	4114151	24163	121	30
13-14	3483430	28690	115	35
14-15	3434448	41790	116	46
15-16	1059079	21981	108	27
16-17	1946491	22206	105	30
17-18	4300076	20810	112	32
18-19	4856492	21611	108	33
19-20	4710906	21427	94	28
20-21	4326474	20807	112	30
21-22	5751690	15005	97	26
22-23	4672958	15200	106	29
23-24	5049715	12686	102	24
TOTAL	72809493	428443		

Table A.2: NFS operations counts

GLOSSARY

Active CS

is an update algorithm in NDN-NIC that actively creates BF-FIB names to reduce false positives in BF-CS.

AODV

Ad hoc On-Demand Distance Vector routing.

ARP

Address Resolution Protocol.

ARQ

Automatic Repeat reQuests.

Basic CS

is an update algorithm in NDN-NIC that does not add names to BF-CS if they are under a FIB entry.

BF

Bloom filter.

BFD

Bidirectional Forwarding Detection.

CBF

counting Bloom filter.

CCN

Content Centric Networking.

COMET

COntent Mediator architecture for content-aware nETwork.

consumer

is an NDN party that expresses Interests.

Content Store

is a cache of Data packets.

CS

Content Store.

Data

is an NDN network layer packet that carries a piece of content.

Dead Nonce List

is a compact data structure that stores name+nonce combinations.

Direct Mapping

is an update algorithm in NDN-NIC that directly maps entries in a table to the corresponding BF.

DM

Direct Mapping.

DNL

Dead Nonce List.

DNS

Domain Name System.

DONA

Data-Oriented Network Architecture.

DoS

Denial of Service.

EPM

entry-key prefix match.

FEC

Forward Error Correction.

FIA

Future Internet Architecture.

FIB

Forwarding Information Base.

FIFO

First In First Out.

Forwarding Information Base

is a table of Interest forwarding paths.

FSTP

Fast Spanning Tree Protocol.

HDFS

Hadoop Distributed File System.

HSSI

High-Speed Serial Interface.

ICN

Information-Centric Networking.

IGMP

Internet Group Management Protocol.

Interest

is an NDN network layer packet that requests the retrieval of a piece of content.

IP

Internet Protocol.

ISP

Internet service provider.

LAN

local area network.

LFBL

Listen First, Broadcast Later.

LFU

Least Frequently Used.

local area network

is a computer network that spans a small geographic area, usually administered by a single organization or individual, and does not contain leased lines.

LRU

Least Recently Used.

MTU

Maximum Transmission Unit.

multicast

is group communication where information is addressed to a group of destination computers simultaneously.

Nack

is a hop-by-hop packet that indicates an Interest cannot be satisfied.

NDN

Named Data Networking.

NDNLP

NDN Link Protocol.

NetInf

Network of Information.

NFD

NDN Forwarding Daemon.

NFS

Network File System.

NIC

network interface card.

OSPF

Open Shortest Path First.

Pending Interest Table

is a table of pending Interests.

PIT

Pending Interest Table.

producer

is an NDN party that produces Data in reply to Interests.

PURSUIT

Publish Subscribe Internet Technology.

retransmission timeout

is a timer value calculated using TCP's algorithm [94].

RONR

Reactive Optimistic Name-based Routing.

RPC

Remote Procedure Call.

RTO

retransmission timeout.

shared medium

is a medium or channel of information transfer that serves more than one user at the same time.

SPM

search-key prefix match.

TLS

Transport Layer Security.

TLV

Type-Length-Value.

WAN

wide area network.

XIA

eXpressive Internet Architecture.

REFERENCES

[1] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, k. claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, "Named Data Networking," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 66–73, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2656877.2656887

[2] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking Named Content," in *CoNEXT*, 2009.

[3] Palo Alto Research Center, "CCNx 0.7.2," 2013. [Online]. Available: {https://github.com/ProjectCCNx/ccnx/tree/ccnx-0.7.2/doc}

[4] 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 *ICN'2014*.

[5] C. Yi, A. Afanasyev, I. Moiseenko, L. Wang, B. Zhang, and L. Zhang, "A Case for Stateful Forwarding Plane," *Comput. Commun.*, vol. 36, no. 7, Apr. 2013.

[6] NDN Project Team, "NDN Protocol Design Principles," http://named-data.net/project/ndn-design-principles/.

[7] ——, "NDN Packet Format Specification (version 0.2-2)," 2017. [Online]. Available: https://named-data.net/doc/ndn-tlv/

[8] T. Haynes and D. Noveck, "Network File System (NFS) Version 4 Protocol," RFC 7530 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–323, Mar. 2015, updated by RFC 7931. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7530.txt

[9] A. S. Foundation, "Hdfs architecture," 2015. [Online]. Available: http://hadoop.apache.org/docs/r2.7.1/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[10] D. Mehndiratta, "How does Dropbox LAN Sync work and protect users' privacy when they enable this over a shared LAN?" 2013. [Online]. Available: http://qr.ae/RXt5lK

[11] P. Mockapetris, "Domain names - concepts and facilities," RFC 1034 (Internet Standard), RFC Editor, Fremont, CA, USA, pp. 1–55, Nov. 1987, updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936, 8020. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1034.txt

[12] D. Plummer, "Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware," RFC 826 (Internet Standard), RFC Editor, Fremont, CA, USA, pp. 1–10, Nov. 1982, updated by RFCs 5227, 5494. [Online]. Available: https://www.rfc-editor.org/rfc/rfc826.txt

[13] R. Perlman, "An Algorithm for Distributed Computation of a Spanningtree in an Extended LAN," *SIGCOMM Comput. Commun. Rev.*, vol. 15, no. 4, pp. 44–53, Sep. 1985. [Online]. Available: http://doi.acm.org/10.1145/318951.319004

[14] J. Moy, "OSPF Version 2," RFC 2328 (Internet Standard), RFC Editor, Fremont, CA, USA, pp. 1–244, Apr. 1998, updated by RFCs 5709, 6549, 6845, 6860, 7474, 8042. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2328.txt

[15] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan, "Internet Group Management Protocol, Version 3," RFC 3376 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–53, Oct. 2002, updated by RFC 4604. [Online]. Available: https://www.rfc-editor.org/rfc/rfc3376.txt

[16] M. Christensen, K. Kimball, and F. Solensky, "Considerations for Internet Group Management Protocol (IGMP) and Multicast Listener Discovery (MLD) Snooping Switches," RFC 4541 (Informational), RFC Editor, Fremont, CA, USA, pp. 1–16, May 2006. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4541.txt

[17] B. Zhang, S. Jamin, and L. Zhang, "Host multicast: a framework for delivering multicast to end users," in *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, 2002.

[18] Y. Yu, A. Afanasyev, Z. Zhu, and L. Zhang, "NDN Technical Memo: Naming Conventions," NDN, Tech. Rep. NDN-0023, 2014.

[19] I. Moiseenko, L. Wang, and L. Zhang, "Consumer / Producer Communication with Application Level Framing in Named Data Networking," in *ACM-ICN*, 2015.

[20] Y. Thomas, G. Xylomenos, C. Tsilopoulos, and G. C. Polyzos, "Object-Oriented Packet Caching for ICN," in *ACM-ICN*, 2015.

[21] G. Rossini and D. Rossi, "Coupling Caching and Forwarding: Benefits, Analysis, and Implementation," in *ACM-ICN*, 2014.

[22] V. Lehman, A. K. M. M. Hoque, Y. Yu, L. Wang, B. Zhang, and L. Zhang, "A Secure Link State Routing Protocol for NDN," NDN, Tech. Rep. NDN-0037, 2016.

[23] NDN Project Team, "NFD management protocol," https://redmine.named-data.net/projects/nfd/wiki/Management.

[24] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica, "A Data-oriented (and Beyond) Network Architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 4, pp. 181–192, Aug. 2007. [Online]. Available: http://doi.acm.org/10.1145/1282427.1282402

[25] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–104, Aug. 2008, updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5246.txt

[26] K. Shilton, J. Burke, C. Duan, and L. Zhang, "A World on NDN: Affordances & Implications of the Named Data Networking Future Internet Architecture," 2014.

[27] Y. Yu, A. Afanasyev, D. Clark, k. claffy, V. Jacobson, and L. Zhang, "Schematizing Trust in Named Data Networking," in *ACM-ICN*, 2015.

[28] A. Afanasyev *et al.*, "NFD Developer's Guide," NDN, Technical Report NDN-0021, Revision 7, 2016.

[29] S. Mastorakis, A. Afanasyev, I. Moiseenko, and L. Zhang, "ndnSIM 2.0: A new version of the NDN simulator for NS-3," NDN, Technical Report NDN-0028, January 2015.

[30] A. Afanasyev, J. Burke, D. Pesavento, B. Zhang, and L. Zhang, "Tutorial: Exploring NDN research through real world problem solving," in *Proceedings of ACM Information-Centric Networking Conference (ICN'2016)*, September 2016, experimenting with NDN Apps using Mini-NDN.

[31] V. Lehman, A. Gawande, R. Aldecoa, D. Krioukov, B. Zhang, L. Zhang, and L. Wang, "An experimental investigation of hyperbolic routing with a smart forwarding plane in ndn," NDN, Tech. Rep. NDN-0042, 2016.

[32] Z. Zhu and A. Afanasyev, "Let's ChronoSync: Decentralized dataset state synchronization in Named Data Networking," in *Proceedings of the 21st IEEE International Conference on Network Protocols (ICNP 2013)*, Goettingen, Germany, October 2013. [Online]. Available: http://icnp13.informatik.uni-goettingen.de/index.html

[33] G. Grassi, D. Pesavento, L. Wang, G. Pau, R. Vuyyuru, R. Wakikawa, and L. Zhang, "Vehicular inter-networking via named data," *CoRR*, vol. abs/1310.5980, 2013. [Online]. Available: http://arxiv.org/abs/1310.5980

[34] K. Schneider and B. Zhang, "How to Establish Loop-Free Multipath Routes in Named Data Networking," NDN, Tech. Rep. NDN-0044, 2017.

[35] C. Perkins, E. Belding-Royer, and S. Das, "Ad hoc On-Demand Distance Vector (AODV) Routing," RFC 3561 (Experimental), RFC Editor, Fremont, CA, USA, pp. 1–37, Jul. 2003. [Online]. Available: https://www.rfc-editor.org/rfc/rfc3561.txt

[36] P. Gusev, Z. Wang, J. Burke, L. Zhang, T. Yoneda, R. Ohnishi, and E. Muramoto, "Real-Time Streaming Data Delivery over Named Data Networking," *IEICE TRANSACTIONS on Communications*, 2016.

[37] G. Grassi, D. Pesavento, G. Pau, L. Zhang, and S. Fdida, "Navigo: Interest forwarding by geolocations in vehicular named data networking," in *WoWMoM 2015*.

[38] V. Ramachandran and S. Nandi, "Detecting arp spoofing: An active technique," in *ICISS*, 2005.

[39] S. DiBenedetto and C. Papadopoulos, "Mitigating poisoned content with forwarding strategy," in *INFOCOM 2016*.

[40] A. Afanasyev, P. Mahadevan, I. Moiseenko, E. Uzun, and L. Zhang, "Interest Flooding Attack and Countermeasures in Named Data Networking," in *IFIP Networking 2013*, May 2013.

[41] D. Katz and D. Ward, "Bidirectional Forwarding Detection (BFD)," RFC 5880 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–49, Jun. 2010, updated by RFCs 7419, 7880. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5880.txt

[42] V. Paxson, M. Allman, J. Chu, and M. Sargent, "Computing TCP's Retransmission Timer," RFC 6298 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–11, Jun. 2011. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6298.txt

[43] S. Vusirikala, S. Mastorakis, A. Afanasyev, and L. Zhang, "Hop-by-hop best effort link layer reliability in Named Data Networking," NDN, Tech. Rep. NDN-0041, 2016.

[44] K. Schneider, C. Yi, B. Zhang, and L. Zhang, "A Practical Congestion Control Scheme for Named Data Networking," in *ICN 2016*.

[45] B. Lantz, B. Heller, and N. McKeown, "A Network in a Laptop: Rapid Prototyping for Software-defined Networks," in *HotNets-IX*.

[46] M. Goldweber and R. Davoli, "VDE: An Emulation Environment for Supporting Computer Networking Courses," *SIGCSE Bull.*, vol. 40, no. 3, Jun. 2008.

[47] Y. Liu, F. Li, L. Guo, B. Shen, S. Chen, and Y. Lan, "Measurement and analysis of an internet streaming service to mobile devices," *IEEE Transactions on Parallel and Distributed Systems*, 2013.

[48] H. L. Mai, N. T. Nguyen, G. Doyen, A. Ploix, and R. Cogranne, "On the readiness of ndn for a secure deployment: The case of pending interest table," in *IFIP WG 6.6 International Conference on Management and Security in the Age of Hyperconnectivity*, 2016.

[49] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Commun. ACM*, vol. 13, no. 7, Jul. 1970.

[50] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *Transactions on Networking*, vol. 8, no. 3, 2000.

[51] M. Ramakrishna, E. Fu, and E. Bahcekapili, "A Performance Study of Hashing Functions for Hardware Applications," in *Int. Conf. on Computing and Information*, 1994.

[52] A. Kirsch and M. Mitzenmacher, "Less Hashing, Same Performance: Building a Better Bloom Filter," *Random Struct. Algorithms*, vol. 33, no. 2, Sep. 2008.

[53] C. E. Rothenberg, C. A. B. Macapuna, M. F. Magalhães, F. L. Verdi, and A. Wiesmaier, "In-packet Bloom filters: Design and networking applications," *Computer Networks*, vol. 55, no. 6, pp. 1364 – 1378, 2011.

[54] *NetFPGA-1G-CML Reference Manual*, https://reference.digilentinc.com/_media/netfpga-1g-cml:netfpga-1g-cml_rm.pdf, 2016.

[55] G. Fairhurst and L. Wood, "Advice to link designers on link Automatic Repeat reQuest (ARQ)," RFC 3366 (Best Current Practice), RFC Editor, Fremont, CA, USA, pp. 1–27, Aug. 2002. [Online]. Available: https://www.rfc-editor.org/rfc/rfc3366.txt

[56] M. Luby, L. Vicisano, J. Gemmell, L. Rizzo, M. Handley, and J. Crowcroft, "The Use of Forward Error Correction (FEC) in Reliable Multicast," RFC 3453 (Informational), RFC Editor, Fremont, CA, USA, pp. 1–18, Dec. 2002. [Online]. Available: https://www.rfc-editor.org/rfc/rfc3453.txt

[57] J. Postel, "Internet Protocol," RFC 791 (Internet Standard), RFC Editor, Fremont, CA, USA, pp. 1–51, Sep. 1981, updated by RFCs 1349, 2474, 6864. [Online]. Available: https://www.rfc-editor.org/rfc/rfc791.txt

[58] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," RFC 2460 (Draft Standard), RFC Editor, Fremont, CA, USA, pp. 1–39, Dec. 1998, updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946, 7045, 7112. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2460.txt

[59] A. Afanasyev, J. Shi, L. Wang, B. Zhang, and L. Zhang, "Packet Fragmentation in NDN: Why NDN Uses Hop-By-Hop Fragmentation (NDN Memo)," NDN, NDN Memo, Technical Report NDN-0032, May 2015.

[60] D. Fisher, "A Look Behind the Future Internet Architectures Efforts," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 45–49, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2656877.2656884

[61] A. Venkataramani, J. F. Kurose, D. Raychaudhuri, K. Nagaraja, M. Mao, and S. Banerjee, "MobilityFirst: A Mobility-centric and Trustworthy Internet Architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 74–80, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2656877.2656888

[62] T. Anderson, K. Birman, R. Broberg, M. Caesar, D. Comer, C. Cotton, M. J. Freedman, A. Haeberlen, Z. G. Ives, A. Krishnamurthy, W. Lehr, B. T. Loo, D. Mazières, A. Nicolosi, J. M. Smith, I. Stoica, R. van Renesse, M. Walfish, H. Weatherspoon, and C. S. Yoo, "A Brief Overview of the NEBULA Future Internet Architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 81–86, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2656877.2656889

[63] D. Naylor, M. K. Mukerjee, P. Agyapong, R. Grandl, R. Kang, M. Machado, S. Brown, C. Doucette, H.-C. Hsiao, D. Han, T. H.-J. Kim, H. Lim, C. Ovon, D. Zhou, S. B. Lee, Y.-H. Lin, C. Stuart, D. Barrett, A. Akella, D. Andersen, J. Byers, L. Dabbish, M. Kaminsky, S. Kiesler, J. Peha, A. Perrig, S. Seshan, M. Sirbu, and P. Steenkiste, "XIA: Architecting a More Trustworthy and Evolvable Internet," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 50–57, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2656877.2656885

[64] T. Wolf, J. Griffioen, K. L. Calvert, R. Dutta, G. N. Rouskas, I. Baldin, and A. Nagurney, "ChoiceNet: Toward an Economy Plane for the Internet," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 58–65, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2656877.2656886

[65] G. Xylomenos, C. N. Ververidis, V. A. Siris, N. Fotiou, C. Tsilopoulos, X. Vasilakos, K. V. Katsaros, and G. C. Polyzos, "A Survey of Information-Centric Networking Research," *IEEE Communications Surveys Tutorials*, vol. 16, no. 2, pp. 1024–1049, Second 2014.

[66] N. Fotiou, P. Nikander, D. Trossen, and G. C. Polyzos, "Developing Information Networking Further: From PSIRP to PURSUIT." in *BROADNETS*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, I. Tomkos, C. Bouras, G. Ellinas, P. Demestichas, and P. Sinha, Eds., vol. 66. Springer, 2010, pp. 1–13. [Online]. Available: http://dblp.uni-trier.de/db/conf/broadnets/broadnets2010.html#FotiouNTP10

[67] C. Dannewitz, D. Kutscher, B. Ohlman, S. Farrell, B. Ahlgren, and H. Karl, "Network of Information (NetInf) An information-centric networking architecture," *Computer Communications*, vol. 36, no. 7, pp. 721 – 735, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0140366413000364

[68] G. Garca, A. Beben, F. J. Ramn, A. Maeso, I. Psaras, G. Pavlou, N. Wang, J. liwiski, S. Spirou, S. Soursos, and E. Hadjioannou, "COMET: Content mediator architecture for content-aware networks," in *2011 Future Network Mobile Summit*, June 2011, pp. 1–8.

[69] N. B. Melazzi, *CONVERGENCE: Extending the Media Concept to Include Representations of Real World Objects*. New York, NY: Springer New York, 2010, pp. 129–140. [Online]. Available: http://dx.doi.org/10.1007/978-1-4419-1674-7_13

[70] D. Kutscher, S. Eum, K. Pentikousis, I. Psaras, D. Corujo, D. Saucez, T. Schmidt, and M. Waehlisch, "Information-Centric Networking (ICN) Research Challenges," RFC 7927 (Informational), RFC Editor, Fremont, CA, USA, pp. 1–38, Jul. 2016. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7927.txt

[71] Yi, Cheng, "Adaptive Forwarding in Named Data Networking," Ph.D. dissertation, The University of Arizona.

[72] P. Baran, "On Distributed Communications Networks," *Communications Systems, IEEE Transactions on*, vol. 12, March 1964.

[73] M. Meisel, V. Pappas, and L. Zhang, "Listen First, Broadcast Later: Topology-Agnostic Forwarding under High Dynamics," in *ITA 2010*.

[74] C. Partridge, R. Walsh, M. Gillen, G. Lauer, J. Lowry, W. T. Strayer, D. Kong, D. Levin, J. Loyall, and M. Paulitsch, "A Secure Content Network in Space," ser. CHANTS '12, 2012.

[75] A. W. Kazi, "CCN Forwarding Strategies," Ph.D. dissertation, Stony Brook University, 2015.

[76] O. Ascigil, V. Sourlas, I. Psaras, and G. Pavlou, "Opportunistic off-path content discovery in information-centric networks," in *LANMAN 2016*.

[77] E. J. Rosensweig and J. Kurose, "Breadcrumbs: Efficient, Best-Effort Content Location in Cache Networks," in *IEEE INFOCOM 2009*.

[78] J. Cao, D. Pei, X. Zhang, B. Zhang, and Y. Zhao, "Fetching Popular Data from the Nearest Replica in NDN," in *2016 25th International Conference on Computer Communication and Networks (ICCCN)*, Aug 2016, pp. 1–9.

[79] W. So, A. Narayanan, and D. Oran, "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*, ser. ANCS '13, 2013.

[80] Y. Wang, K. He, H. Dai, W. Meng, J. Jiang, B. Liu, and Y. Chen, "Scalable Name Lookup in NDN Using Effective Name Component Encoding," in *2012 IEEE 32nd International Conference on Distributed Computing Systems*, 2012.

[81] M. Varvello, D. Perino, and J. Esteban, "Caesar: A Content Router for High Speed Forwarding," in *Proceedings of the Second Edition of the ICN Workshop on Information-centric Networking*, ser. ICN '12, 2012.

[82] H. Yuan and P. Crowley, "Scalable Pending Interest Table design: From principles to practice," in *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, 2014.

[83] A. Broder, M. Mitzenmacher, and A. B. I. M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," in *Internet Mathematics*, 2002, pp. 636–646.

[84] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest Prefix Matching Using Bloom Filters," in *SIGCOMM 2003*.

[85] W. So, A. Narayanan, D. Oran, and Y. Wang, "Toward Fast NDN Software Forwarding Lookup Engine Based on Hash Tables," in *ANCS 2012*.

[86] Y. Wang, T. Pan, Z. Mi, H. Dai, X. Guo, T. Zhang, B. Liu, and Q. Dong, "NameFilter: Achieving fast name lookup with low memory cost via applying two-stage Bloom filters," in *INFOCOM 2013*.

[87] W. Quan, C. Xu, J. Guan, H. Zhang, and L. A. Grieco, "Scalable Name Lookup with Adaptive Prefix Bloom Filter for Named Data Networking," *Communications Letters, IEEE*, vol. 18, no. 1, 2014.

[88] Z. Li, K. Liu, Y. Zhao, and Y. Ma, "MaPIT: An Enhanced Pending Interest Table for NDN With Mapping Bloom Filter," *Communications Letters, IEEE*, vol. 18, no. 11, pp. 1915–1918, 2014.

[89] T. Benson, A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *IMC 2010*.

[90] S. D. Gribble, "UC Berkeley Home IP HTTP Traces," http://www.acm.org/sigcomm/ITA/, 1997.

[91] C. Cunha, A. Bestavros, and M. Crovella, "Characteristics of WWW Client-based Traces," Tech. Rep., 1995.

[92] D. Ellard and M. Seltzer, "New NFS Tracing Tools and Techniques for System Analysis," in *LISA 2003*.

[93] B. Callaghan, B. Pawlowski, and P. Staubach, "NFS Version 3 Protocol Specification," RFC 1813 (Informational), RFC Editor, Fremont, CA, USA, pp. 1–126, Jun. 1995. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1813.txt

[94] Y. Bernet, P. Ford, R. Yavatkar, F. Baker, L. Zhang, M. Speer, R. Braden, B. Davie, J. Wroclawski, and E. Felstaine, "A Framework for Integrated Services Operation over Diffserv Networks," RFC 2998 (Informational), RFC Editor, Fremont, CA, USA, pp. 1–31, Nov. 2000. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2998.txt