YaNFD: Yet another Named Data Networking Forwarding Daemon

Eric Newberry
UCLA
Los Angeles, California, USA
enewberry@cs.ucla.edu

Xinyu Ma
UCLA
Los Angeles, California, USA
xinyu.ma@cs.ucla.edu

Lixia Zhang
UCLA
Los Angeles, California, USA
lixia@cs.ucla.edu

ABSTRACT

We have developed YaNFD as a new software packet forwarder for NDN. YaNFD achieves compatibility with existing NDN applications and forwarders, as well as high throughput. YaNFD features multi-threaded forwarding, a smaller and more streamlined codebase compared to existing implementations, and can be managed using existing NDN forwarder management utilities and protocols. In this paper, we discuss our implementation, including how it differs from previous forwarders based upon lessons learned during their development and use over the span of multiple years. Additionally, we present the lessons learned from our experience developing a new forwarder for NDN from the ground up.

CCS CONCEPTS

• **Networks** → **Routers**; *Network performance analysis.*

KEYWORDS

Named data networking, Information-centric networking, Forwarder design

ACM Reference Format:
Eric Newberry, Xinyu Ma, and Lixia Zhang. 2021. YaNFD: Yet another Named Data Networking Forwarding Daemon. In *8th ACM Conference on Information-Centric Networking (ICN '21), September 22–24, 2021, Paris, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3460417.3482969

1 INTRODUCTION

The Named Data Networking [32] (NDN) network architecture was developed in response to the gradual shift of Internet traffic away from host-to-host communication (e.g., file transfers and email) to content retrieval (e.g., streaming video). NDN replaces the stateless, push-based model of IP with stateful, pull-based requests for named, secured pieces of data. Because NDN pushes semantically-meaningful application layer identifiers down into the network layer, host identifiers are not used to request content, eliminating their use in the core forwarding processes.

Since NDN's stateful forwarding mechanisms requires a significantly different data plane architecture than IP's stateless forwarding, new packet forwarder implementations are required. As NDN is still in development, these forwarders are not implemented in hardware (on routers) or the kernel (on end hosts), as they are for IP – instead, they operate as user-space software.

For many years, the primary forwarder used for both experimentation and development in NDN has been the NDN Forwarding Daemon (NFD) [24]. NFD was among the first forwarders designed specifically for NDN and its development produced many innovative solutions to address the problems introduced by NDN's stateful forwarding plane. However, NFD is limited in some scenarios by its complex single-threaded forwarding implementation and heavyweight storage and compilation requirements. As such, new forwarders have been developed for other scenarios with different requirements, such as IoT or high-speed network cores. Despite this, until recently, there have been no attempts to provide alternatives to NFD for the general-purpose computing edge, such as personal computers and home routers. One exception is a recent effort reported in [5], which extends NFD to provide multithread support, and has demonstrated significant improvements over NFD.

Given the opportunity to create a more efficient NDN forwarder for computing edges, we designed a new forwarder, known as YaNFD, from the ground up [1]. YaNFD inherits no code from NFD and is written in Go, a modern high-level programming language. Designing YaNFD from scratch gave us the opportunity to evaluate which features from NFD were necessary for NDN forwarding and which we could safely discard or replace. As such, we have streamlined the NDN forwarding pipeline and have avoided many of the pitfalls that come from adapting an existing codebase to a specification [20] that is still under active development. Our evaluations demonstrate that YaNFD is able to achieve better performance than NFD, while having significantly smaller codebase and storage footprints. Additionally, we have provided further proof that multithreaded forwarding for NDN is not only faster, but quite feasible to design and straightforward to implement.

2 BACKGROUND

In this section, we provide an overview of the Named Data Networking (NDN) architecture (Section 2.1) and existing software packet forwarders for NDN, including lessons learned from these forwarders (Section 2.2).

2.1 Named Data Networking

Named Data Networking (NDN) is a network architecture that radically differs from the IP network architecture in use on the Internet today [16, 32]. NDN is an information-centric networking (ICN) architecture, meaning that it pushes the semantically-meaningful

ICN '21, September 22–24, 2021, Paris, France
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8460-5/21/09.
https://doi.org/10.1145/3460417.3482969

[1]The source code for YaNFD can be found at https://github.com/named-data/YaNFD

content identifiers used in the application layer down into the network layer. This allows forwarding to be performed directly on content "names", instead of the likely unrelated host identifiers.

NDN's design provides a number of benefits and optimizations [32] that would require application-layer workarounds or complex schemes in architectures based on host identifiers (such as IP). These include implicit multipath forwarding, implicit multicasting, implicit multihoming, loop detection, in-network caching, and semantically-meaningful security policies that can secure data both in transit and in storage using the same mechanisms [33].

Many of these features are enabled through NDN's use of a stateful forwarding plane [31], in contrast to the stateless forwarding plane used in IP. This stateful forwarding plane is composed of several data structures with distinct purposes:

- The *Pending Interest Table (PIT)* tracks outstanding requests for content (known as "Interests"), which are satisfied by returning pieces of content (sent in "Data" packets). Data packets can match Interests whose names are either exact matches or prefixes of their names. Exact matching is the default behavior [20], but prefix matching can be enabled for specific Interests by adding a flag field. The PIT also serves to aggregate requests for the same content name and allows loops to be detected through the inclusion of a randomly-generated "nonce" field in every Interest packet.
- The *Forwarding Information Base (FIB)* provides longest-prefix matching over NDN's hierarchical content names during the forwarding process. Its matching behavior is comparable to the forwarding table in IP devices.
- The *Routing Information Base (RIB)* stores and aggregates routes to content that have learned from a variety of sources, including routing protocols, manually-set routes, and application advertisements. Its contents are periodically "flattened" into the FIB to allow for fast lookups during the forwarding process, instead of needing to parse through the larger and more complex RIB.
- The *Content Store (CS)* caches recently-forwarded Data packets so that they can be used to satisfy future Interests for the same or similar content (using either exact or longest-prefix matching, like in the PIT).
- The *Dead Nonce List (DNL)* provides long-term loop detection for Interest packets. The names and nonces of PIT entries that have "expired" are moved to the DNL to reduce the storage overhead of the PIT. Similarly to the PIT, it compares an Interest's name and randomly-generated "nonce" field against stored records, dropping them if there is a match. However, in most NDN forwarders, this matching is performed using probabilistic hashing to further reduce storage and lookup overhead.
- The *Strategy Table* indicates which "forwarding strategy" will make forwarding decisions for a given prefix (using longest-prefix matching). Forwarding strategies make forwarding decisions on a per-packet basis. They operate as modules integrated into the forwarder to allow applications flexibility in how their Interests will be forwarded. Strategies determine which nexthop(s) Interests will be sent on, and can even choose to drop an Interest and not forward it further. The choice of strategy can be set for any arbitrary name prefix, allowing the optimal forwarding strategy to be set for each application (or portion of application) in the network, instead of a one-size-fits-all approach, as is taken for packets in IP. In current forwarders, the strategy setting must be changed individually on every forwarder in the network.

Forwarded packets are sent from and received on "faces", which in NDN can be either a local connection to a producer or consumer application on the same host, or a connection over a physical interface or an overlay tunnel to a remote forwarder. This allows all connections to be treated similarly by the forwarder, greatly simplifying the forwarding logic.

2.2 Existing Forwarders

The NDN project has been under active development and use for over a decade and, during this time, multiple software packet forwarders have been developed to support both research and software development. We evaluate each of these forwarders in turn, discussing their strengths and weaknesses, along with the lessons we have gleaned from the development and use of each forwarder.

2.2.1 CCNx. In the first few years of the NDN project, the existing CCNx forwarder [8] for ICN content was utilized for research and development work. However, due to the changing requirements of the NDN project and lessons learned as the architecture saw more use, it was decided to implement a new forwarder, known as NFD (as discussed later on). CCNx made several innovations that were later incorporated into NFD, such as combining the PIT, FIB, and CS into a single data structure, known as a "name tree". However, CCNx is now incompatible with the NDN specification due to the divergence of the NDN protocol and forwarding specifications from those used by CCNx.

2.2.2 NFD. The Named Data Networking Forwarding Daemon (or NFD) [24], was designed as a general-purpose software packet forwarder for NDN, with its implementation commencing in 2014. NFD is the de facto "official" forwarder for the NDN project and has been extensively maintained and refactored over the years as the NDN architecture and specifications have changed. It is worth noting that, although NFD is the reference implementation of the NDN protocol, it is not the specification of the protocol, which is a separate document [20]. Therefore, just because NFD behaves in a certain way, this does not require that every other forwarder behave in the same way to be compatible with the NDN protocol. This difference can also be thought of as follows: the protocol specification states "what" an NDN-compatible *must* and *may* do, while the NFD implementation provides one way "how" such behavior *can* be realized.

NFD was designed with flexibility in mind and is the current forwarder of choice for general purpose edge environments (e.g., personal computers), and is used on the official public NDN project testbed [21]. Additionally, NFD's design has been documented extensively [3].

One of NFD's most significant limitations is that it only uses a single thread for its entire forwarding pipeline, including receiving packets, processing them, and then sending them out again – this can limit its forwarding throughput. Moreover, most management operations and performed in the main forwarding thread (with the notable exception of RIB management). Additionally, NFD was designed with modularity in mind and thus features many generic interfaces that have rarely been expanded beyond one or two concrete implementations. For example, NFD's face system is quite elaborate, allowing in theory for both the link-layer services and underlying transport mechanisms to be swapped in and out with ease. However, as of this writing, only one concrete implementation of a link service has been added to NFD (the NDNLPv2 link service [23]), and, although its design has been improved over the years, other components in the forwarder have been implemented to expect an NDNLPv2-compatible link service, going against the ideal of true modularity in the NFD face system. These design goals also make the NFD codebase quite complex and daunting to new developers, potentially dissuading them from improving NFD or utilizing it in scenarios where implementation changes are necessary.

While issues like the ones mentioned above are pervasive in the NFD codebase, many good lessons can be gleaned from NFD's innovations and development processes. The general structure of the NFD codebase isolates unrelated component groups quite well, such as splitting the forwarding layer from the face layer. Moreover, there are instances where modularity is done right and is frequently utilized to provide new features, such as the forwarding strategy system – this extension point has seen much use of the years, particularly in the research literature [1, 6, 17, 18]. Additionally, NFD integrates multiple data structures [31] (namely, the FIB, PIT, Strategy table, and Measurements table) into a single data structure, like CCNx's name tree, to reduce storage overhead [3].

2.2.3 *ndn-lite.*

Despite NFD's versatility, it can be quite "heavyweight" in terms of resource requirements and compilation overhead. This is not ideal for mobile and constrained edge devices where there are significant system limitations in terms of power and memory. These limitations have led to the development of the ndn-lite [19] forwarder. Given the constraints of mobile and related devices, ndn-lite makes several design decisions aimed at simplifying the forwarder design and codebase. The most notable difference between ndn-lite and other forwarders is that ndn-lite runs in the same thread as the local application utilizing it, instead of as a separate process communicating with applications over socket-based faces. While this means that a single forwarder can only support a single application, it also allows the overhead of maintaining and using a network connection to the forwarder to/from said application to be eliminated.

Another simplification made by ndn-lite is the removal of negative acknowledgements (or "Nacks") from the forwarding pipeline. Nacks are sent when an upstream forwarder cannot forward an Interest further for some reason, such as there being no matching route, a link being too congested, or the Interest being a duplicate of another recent Interest. Nacks increase the complexity of the NDN forwarding pipeline by adding a third type of packet that must be handled separately. In fact, it has been suggested that Nacks, being a later addition to the NDN forwarding pipeline, impact forwarding in unexpected ways due to packet ordering [15]. Moreover, there are security issues associated with using network-layer Nacks [7]. Instead, timeouts can be used as an indication that an Interest needs to be retransmitted from the consumer application.

From ndn-lite's simplified design, we can learn that removing excessive features and overly complex design aspects (such as false modularity and excessive network-layer pipelines) can reduce the size and complexity of the forwarder codebase, leading to a clearer design and an increase ease of modification if future circumstances necessitate such a change. Moreover, by reducing the scope of the scenarios a forwarder is designed to run in, one can make better assumptions that allow one to optimize forwarder performance and reduce forwarding overhead.

2.2.4 *NDN-DPDK.*

While NFD was designed with research and development in mind, as discussed above, it can suffer from performance issues at scale. Therefore, as NDN moves toward deployment at larger scales, it will be unable to keep up with the demands of core networks. The NDN-DPDK [28] forwarder was designed for high throughput networks and is capable of forwarding NDN traffic at rates of up to 100 Gbps, all the while running on general-purpose, albeit high-performance, computing hardware. NDN-DPDK is able to significantly outperform other forwarders by using the Data Plane Development Kit (DPDK) [10] system to process received packets. Using DPDK allows the forwarder to bypass the operating system's networking stack and perform high-speed network packet processing directly in user space.

To enable high performance forwarding, NDN-DPDK has made a number of design improvements, including the use of multiple forwarding threads, separate threads for input and output on each physical interface, and more efficient lookup algorithms.

When using multiple threads, one of the most important considerations is which forwarding thread each incoming packet will be "dispatched" to. NDN-DPDK dispatches Interests based upon the lower-order bits of the hash of the first k components (by default 2) of their name – a scheme that appears to be quite similar to [29]. Since the forwarding state of an Interest is stored only in the thread that processes it, Data packets that satisfy Interests must be dispatched to the same thread(s) that the Interest(s) they satisfy were earlier dispatched to. Given that Interests can in many cases be satisfied by Data packets with names "longer" (in terms of number of components) than they are, NDN-DPDK cannot simply perform a similar prefix-based dispatching for Data packets as it does for Interest packets, while still meeting its performance goals. Instead, it must rely upon "PIT tokens" – pieces of information attached to outgoing Interests and returned with their respective satisfying Data packets – to indicate which thread should process a Data packet. If neighboring forwarders do not attach PIT tokens to returning Data packets, NDN-DPDK will be unable to process them and such Data packets will be dropped, requiring support for this feature in adjacent forwarders.

However, while NDN-DPDK is able to forward packets at high speed, its system requirements make it infeasible for use at the network edge. Namely, NDN-DPDK cannot be run without DPDK and, at this time, DPDK only supports a subset of NIC devices at full forwarding speeds [11, 25] – these specific NICs may not be present on edge systems. Moreover, to achieve high levels of performance, NDN-DPDK runs the DPDK Poll-Mode Driver (PMD) in an infinite loop with no sleeping, leading to around 100% CPU load on at least one core all the time. This is not feasible in edge environments, for obvious reasons of increased power consumption and noise, as well as reducing the performance of other apps and potentially harming the user experience.

2.2.5 MW-NFD. Multi-Worker NFD (MW-NFD) [5] is an NDN forwarder developed from a fork of the NFD codebase. MW-NFD has modified NFD's design to utilize multiple threads for forwarding, as compared to NFD's single-threaded forwarding architecture. MW-NFD creates a set of "worker" threads to perform NDN's stateful packet forwarding, as well as an input thread for each face. However, it gives responsibility for transmission on faces to worker threads, instead of having a separate output thread for each face, as is done in NDN-DPDK. MW-NFD uses a similar short name prefix dispatching system to NDN-DPDK (based on the first k name components, and also using $k = 2$ components by default) and uses PIT tokens to dispatch returning Data packets. However, unlike NDN-DPDK, is prefers, but not require, PIT tokens and will fall back to prefix dispatching for Data packets if a received Data packet does not have an attached PIT token [12].

After making their modifications to the NFD codebase, the authors of MW-NFD were able to demonstrate forwarding speeds of up to 13 times of those they were able to achieve with similar traffic in NFD. At the same time, their forwarder maintains compatibility with NFD and existing NDN applications that communicate with NFD, allowing it to run in the same NDN network as NFD nodes and provide the same communication interfaces for NDN applications to use.

MW-NFD makes several design choices to allow multi-threaded forwarding, including sharding data structures across threads – namely, they shard the PIT, CS, FIB, and Measurements tables for each worker thread. This necessitated that they develop mechanisms to properly separate data into separate data structures across threads, such as separating FIB entries by prefix. However, while MW-NFD works around the major performance bottleneck of NFD, namely the use of only a single thread for forwarding, it carries over many of the other limitations of the NFD codebase, such as false modularity and a complex codebase. Additionally, MW-NFD uses near 100% CPU utilization polling in its input threads. This can have a significant performance and power consumption impact on edge devices, even during idle times.

3 DESIGN

The design of YaNFD is heavily inspired by the designs of multiple previous NDN forwarders, as discussed in Section 2.2. Above, we sought to extract "lessons learned" from our experiences implementing and using NDN forwarders over the years. We use these, as well as the improvements and optimizations to the NDN protocol over the same time frame, to guide the design of our new NDN forwarder. In this section, we discuss the forwarding piplines in use by NDN, and then proceed to lay out the design of our new forwarder, as well as the provide justifications for why made different design decisions compared to existing NDN forwarders.

3.1 NDN Forwarding Pipelines

The forwarding pipelines of YaNFD approximately follow the designs of NFD's forwarding pipelines, as described in detail in NFD's "Developer's Guide" [3]. However, we modified these pipelines to accommodate the use of several thread-local, as opposed to global, data structures. Additionally, we have modified the pipelines to pass packets between the different components of the forwarding pipeline using queues (utilizing Go's "channels" feature), as opposed to direct function calls. We also use channels to handle timeouts and similar events, as opposed to callbacks in NFD.

NDN's forwarding pipelines are unusual compared to other networking data paths, particularly those in IP, because they are stateful, with Data packets specifically "satisfying" specific Interests that were earlier forwarded by the host. Additionally, NDN's forwarding pipelines are different than these existing pipelines because forwarding behavior differs in the network layer for different types of packets (namely, the previously-mentioned Interests and Data packets), as opposed to having a uniform forwarding pipeline for all data plane packets. We now provide a high-level overview of the NDN forwarding pipeline as implemented in YaNFD, derived from [31] and [3].

3.1.1 Interest Pipeline. When an Interest is received our forwarder, it first determines whether the Interest's "hop limit" field (similar to IP's TTL field) has reached zero – if so, it will be dropped; otherwise, the field will be decremented by one. Next, it checks whether the Interest was received on a non-local face and, if so, whether is a local management command – if such a "scope violation" is discovered, the Interest will be dropped. After this, the Interest's name and nonce are first compared against the Dead Nonce List (DNL) – the Interest will be dropped if a match is found in the DNL, as this indicates that it is looping [2]. Next, if a Forwarding Hint [4] field is present in the Interest, our forwarder checks whether the name contained in it matches the producer region configured in the forwarder – if so, the Forwarding Hint field will be removed from the Interest; otherwise, this field is left intact.

After performing these checks, our forwarder adds the Interest to its Pending Interest Table (PIT). If an matching entry is found to already exist in the PIT, the forwarder checks whether an incoming face record *for a different face* already exists in the PIT entry with the same nonce as the Interest; if so, the Interest will be dropped as duplicate – drops do not occur if the same nonce is found for the same incoming face, as this is treated as a retransmission instead of a loop. Next, the "expiration timer" for the PIT entry will be canceled, preventing the PIT entry from expiring.

If there was no existing incoming face record for the incoming face in the PIT entry, the Interest's name will be compared against the entries in the forwarder's Content Store (CS) to determine if there is a cached Data packet that can be used to satisfy the Interest without needing to forward it further. If so, this cached Data packet will be returned on the face the Interest arrived on – the Interest will be considered satisfied and the PIT entry's expiration timer will be restarted. Otherwise, an incoming face record for the incoming face will be added to or updated in the PIT entry and the Interest will continue in the pipeline.

Finally, the forwarder determines which face the Interest will be sent out on. If the Interest contains a field that indicates which face it should be sent out on (known as the "NextHopFaceId"), the forwarder will use that face. Otherwise, the Interest name will be compared against the Forwarding Information Base (FIB) to find the

[2]For performance, both NFD and YaNFD implement the DNL as a probabilistic data structure that matches through hashing.

appropriate nexthop(s), which are then passed to the forwarding strategy layer, which forwards the Interest based upon its strategy-specific policies [3]. If no matching prefix can be found in the FIB, the Interest will be dropped [4].

There are many opportunities to process Interests in parallel, given that temporally local Interests for different names update different entries in the PIT. Meanwhile, outside of the PIT, all data structure interactions in the primary Interest forwarding pipeline are read-only, providing excellent opportunities for parallelization, such as via shared global data structures using readers-writers locking [9].

3.1.2 Data Pipeline. Data packets follow the reverse forwarding path(s) of the Interest(s) they satisfy. After performing similar local/non-local scope checks as for Interests, Data packets are compared to the PIT, finding all matching entries (if none are found, the Data packet will be discarded, although a setting can allow such "unsolicited" Data packets to be cached for future Interests). If more than one matching PIT entry is found, the Data packet will be forwarded to all downstream faces listed in each matching PIT entry. If only a single matching PIT entry is found, the forwarding strategy controlling the most specific namespace matching the Interest's name will be allowed to control the forwarding of the Data packet, since there is no risk of a conflict between forwarding strategies.

Data packets are cached in the CS to allow them to satisfy future Interests for the same or similar content. PIT entries matched by a Data packet will be marked as satisfied, but will not be immediately deleted – instead, they will be removed when its expiration timer expires to allow Interest names and nonces to be migrated to the DNL [3].

Unlike the Interest pipeline, the Data pipeline involves significantly more write accesses to global Data structures, including both the CS and PIT. Therefore, the problem of providing parallel Data pipelines is more complex and must be handled via mechanisms such as sharded data structures. We have addressed these barriers to parallelization in the design of YaNFD, as discussed in the following sections.

3.2 Design Overview

A high-level overview of YaNFD's design is presented in Figure 1. Our design features a user-configurable number of forwarding threads (by default, eight), two threads for each face: one for receiving (input) and one for sending (output), and a separate management thread. We inherited the link service/transport modular split from NFD (as shown in the figure), as this design allows each type of transport (e.g., UDP, Unix stream, or Ethernet) to be treated identically by the forwarder. In this design, the link service provides common features as part of the link protocol – currently, we implement NDNLPv2, which provides optional fragmentation/reassembly, hop-by-hop reliability (not yet implemented at the time of writing), congestion marking, and other features [23]. Meanwhile, the transport is responsible for interface-specific operations, such as reading to and writing from a socket or Ethernet packet capture handle.

The link service passes received packets on to the appropriate forwarding thread by hashing complete names for Interests and through PIT tokens for Data packets [5]. PIT tokens generated by YaNFD contain both the thread ID and the identifier of the particular PIT entry the Data packet satisfies. Packets are queued to be processed in the order they are received by the appropriate forwarding thread, which will then perform its forwarding computations, including passing Interests to the appropriate forwarding strategy before pushing the packet on to the appropriate outgoing face thread queue(s), which will then process and send the packets using their link service and transport-specific mechanisms.

3.2.1 Interest Dispatching Logic. Notably, our design differs from both NDN-DPDK and MW-NFD in that it uses the hash of the *entire* Interest name to dispatch packets, instead of just the hash of a fixed-length prefix. This is because we posit that only using the first k name components may not effective at spreading work equally among multiple forwarding threads when much of the traffic is for the same prefix, particularly given how short the default value of k is for both forwarders. While core routers, the target deployment environment of NDN-DPDK, will likely forward a diverse set of traffic at any given time, at the edge, it is quite likely that requests for content under the same prefix will be sent out in a short period of time (for example, requests under `/net/named-data` within the Named Data Networking organization). Such behaviors will be particularly notable when performing bulk data transfers where one large object is split up into many segments, all of which would likely be forwarded via the same forwarding thread when using NDN-DPDK's dispatching scheme.

Moreover, since one can predict which forwarding thread will handle a given short prefix by knowing the hashing algorithm (or reverse engineering it from PIT tokens sent upstream), it is easier for attackers to selectively overload a given forwarding thread by sending Interests under a given prefix. With full name hashing, one would need to check that each generated Interest name hashes to the given thread that one wishes to overload, since this will vary even for very similarly named Interests. Meanwhile, with prefix-based hashing, one could just send Interests under a given prefix as quickly as possible, requiring less computing power than performing the same attack on a full name hashing forwarder. Therefore, we take a different approach in YaNFD and use the entire name for Interest dispatching.

3.2.2 Global and Thread-specific Data Structures. Given that each forwarding thread in YaNFD needs to be able to be operate without blocking other forwarding threads as much as possible, we sought to eliminate as much mutable shared state between forwarding threads as possible. In this regard, while the FIB and Strategy tables change relatively infrequently and are only used for read-only lookups by the forwarding threads, the PIT, Content Store, and Dead Nonce List are modified frequently. Meanwhile, the Measurements Table can

[3]The Interest will be dropped before being sent to a non-local face if its hop limit was *decremented to zero* to avoid sending packets that the next forwarder will simply drop.
[4]In some other NDN forwarders, the lack of a route will result in a negative acknowledgement (Nack) being sent downstream. However, we made the decision to not support Nacks in YaNFD, as we discuss later on.

[5]For Data packets received from local producers, it is not feasible to expect a PIT token to be attached to returning Data packets, since this is not part of the standard NDN interface with applications. Therefore, in this case, we simply dispatch to multiple forwarding threads based upon the hash of all prefixes of the Data name, dropping the packet if there are no matching PIT entries.

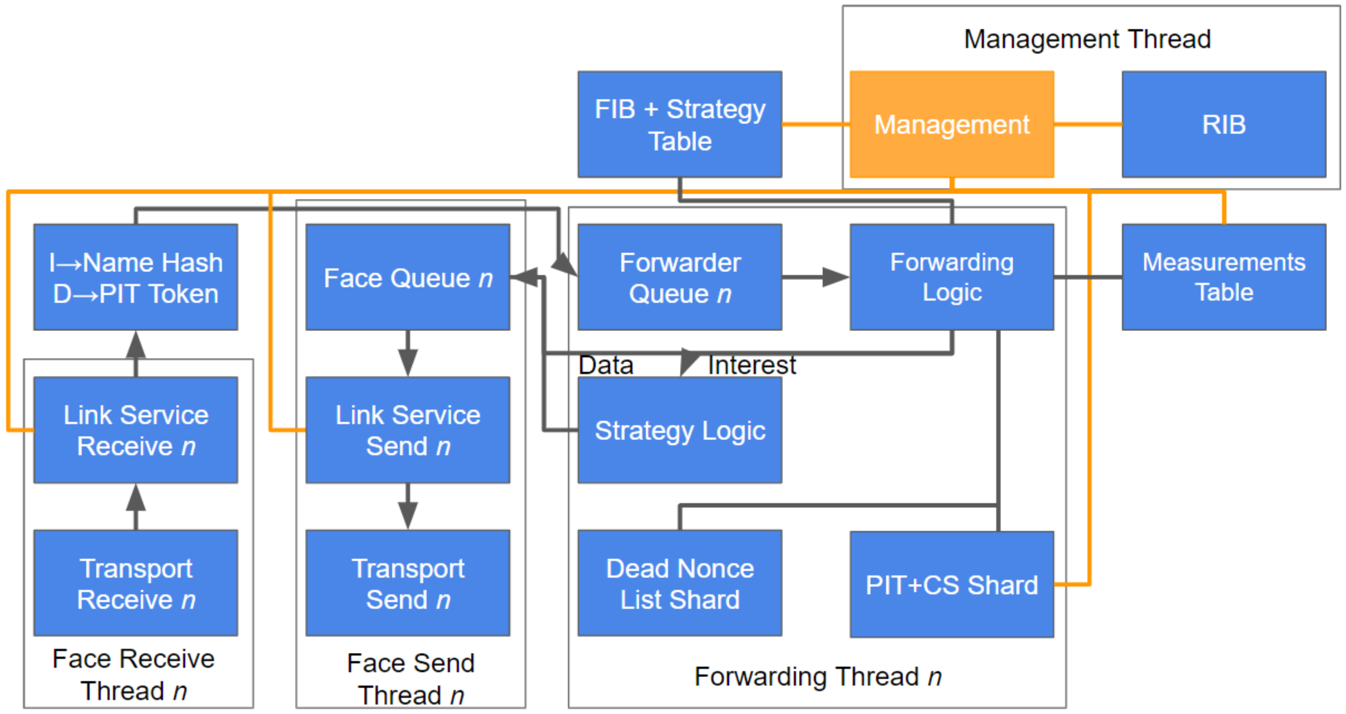


Figure 1: Design overview of YaNFD

be made lock-free by using atomic updates to the values contained within. Therefore, we made the FIB and Strategy tables (combined into a single tree-based data structure to reduce storage overhead), as well as the Measurements table, global. Meanwhile, we gave each forwarding thread a separate PIT, Content Store, and Dead Nonce List (with the first two combined into a single tree-based data structure to reduce storage overhead) [6].

When an update needs to be made to the global FIB-Strategy Table, our separate management thread will lock it from being read by forwarding threads using a readers-writers lock [9]. During updates to FIB entries, routes obtained from various sources (e.g., statically-set by an administrator or originating from a routing protocol or application announcement) will be "flattened" and pushed down from the RIB into the FIB.

Additionally, forwarding strategies need to be sharded among the forwarding threads to avoid locking data structures in the strategy layer. If forwarding strategies in different threads need to interact, they can use the global Measurements Table, which allows lock-free reads and writes.

Although the IP forwarding plane is significantly simpler than the NDN forwarding plane, many of the issues we discuss above with developing a multi-threaded forwarding plane for NDN can be compared to past efforts to develop multi-threaded IP forwarders. In particular, both forwarding planes involve shared data structures that must be atomically updated, such as the Pending Interest Table, FIB, Content Store, and Strategy table in NDN and the FIB in IP.

[6]The Dead Nonce List was not combined with the PIT-CS because it is a probabilistic, hash-based data structure, as opposed to an exact/prefix-matching, tree-based data structure

4 METHODOLOGY

We conducted evaluations to evaluate both the correctness and performance of YaNFD. Our evaluations were conducted on an Ubuntu 20.04 server. This server features an AMD EPYC 7702P processor [2], with 64 physical cores (128 threads) running at 2.0 GHz, and 256 GB of RAM. We now discuss the applications we used, along with the design of our experiments.

4.1 Applications

To put the forwarders we evaluated under realistic strain, our evaluations considered their performance when transferring large files. As NDN uses pull-based communication, content is sent by a "producer" application (in "Data" packets) in response to requests (in "Interest" packets) made by a "consumer" application. Therefore, our evaluations were conducted using the NDN file transfer application pair, `ndnputchunks` (producer) and `ndncatchunks` (consumer), which are part of the ndn-tools package [22]. Our evaluations used version 0.7.1 of the ndn-tools package.

The ndnputchunks producer divides the file into multiple segments, creating Data packets named like `/<prefix>/<version>/<segment>`, with "segment" identifying the order of the particular segment within the file being transferred. In our performance evaluations, our "prefix" consisted of a single unique name component for each file two bytes in length, and we specified a constant four byte "version" for all of our evaluations to ensure consistent hashing across trials. Each segment created by ndncatchunks (at the time of writing) is 4400 bytes in length, except the last segment, which contains any remaining data. Given that the Data packet header fields when using a short prefix like the one we are using for our evaluations are only up to a few tens of bytes in length, we can presume the size of the average Data packet in our evaluations to be on the order of 4400 bytes. Meanwhile, Interest packets sent in our evaluations will be significantly shorter, since they only contain the requested name and a few other header fields on the order of tens of bytes. Given that one Interest will retrieve one Data packet, the theoretically perfect traffic mixture between Interest and Data packets will be 1:1. However, given the likelihood of congestion or other events that cause the loss of Interest or Data packets, this ratio will rarely be exactly 1:1.

The ndncatchunks consumer retrieves a complete copy of the latest version of file. If no specific version is specified to the consumer, the latest version of the file will first be discovered using a "discovery Interest" [22]. ndncatchunks supports a variety of congestion control algorithms to adapt its receive window size at runtime. By default, it uses CUBIC congestion control with support for explicit congestion marking by intermediate nodes [27]. However, we found the use of congestion control is result in significant and unexpected variations in performance due to the unpredictability of loss and congestion events. Therefore, we disabled congestion control and instead used a fixed window size of 100 segments. This means that that each consumer would only ever have up to 100 outstanding Interests at a given time. Therefore, after filling up the window, an Interest for a previously unrequested segment could only be sent after a segment in the current window was received.

4.2 Experimental Design

We evaluated our forwarder for both correctness (i.e., meeting the protocol specification and being able to interoperate with NFD) and performance (i.e., how quickly it is able to forward traffic). For each type of experiment, we used a different topology and application setup, which we will now discuss in turn.

4.2.1 Correctness. Our correctness evaluations used two virtualized hosts running Ubuntu 20.04 on VirtualBox 6.1.16. These hosts were connected using a VirtualBox internal network. The producer host ran a forwarder and the "ndnputchunks" producer application from ndn-tools [22]. Meanwhile, the consumer host ran a forwarder and the complementary "ndncatchunks" consumer application. Using these applications, we transferred a 100 MB file from the producer to the consumer. This simple topology is shown in Figure 2a.

4.2.2 Performance. Our performance evaluations utilized three producers and three consumers, each connected directly to the forwarder over Unix stream socket faces. A diagram of our experiment is shown in Figure 2b. Each producer application served a large file under a different prefix, with each consumer requesting a different file. We repeated our evaluations for different file sizes. We conducted the same evaluation scenario for YaNFD, NFD, and MW-NFD, repeating 100 trials for each forwarder and each file size, repeating any trials where a transfer did not complete successfully due to the use of a fixed window.

For the two multi-threaded forwarders in our evaluations (YaNFD and MW-NFD), we set the number of forwarding threads to eight, since this is just over twice the number of flows, while still being a power of two. We chose a power of two because network administrators would likely set the number of threads to a power of two, given that the number of cores in modern CPUs tend to be a power of two.

Table 1: Values of Interest packet fields during correctness transfers.

CanBePrefix	Yes (first Interest), No (further Interests)
MustBeFresh	Yes (first Interest), No (further Interests)
Nonce	Random 4 byte value
InterestLifetime	N/A (implicitly 4000 ms [20])

Table 2: Values of Data packet fields during correctness transfers.

ContentType	N/A (implicitly "BLOB" [20])
FreshnessPeriod	10 ms (first Data), 10000 ms (further)
FinalBlockId	N/A (first Data), Final segment ID (further)
SignatureType	DigestSha256 [20]
SignatureValue	32 byte SHA-256 hash of Content

The length of the prefix of each Data packet was set to a single component, bringing each Data packet's full name length to three components. As MW-NFD uses a prefix of each Interest's name for forwarding, we changed its default settings to have the prefix be the maximum length of each Interest packet in our scenario – otherwise, all Interests and Data for the same transferred file would be processed by the same worker thread, which would unduly impact performance by effectively eliminating any multi-threading potential for a given flow.

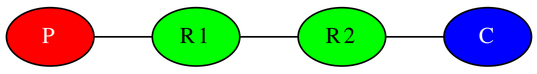
The Content Stores were disabled for all trials, as we found an issue where RTTs increased greatly with time for both NFD and YaNFD with them enabled. As in-network caching in NDN is simply an optimization to avoid needing to forward an Interest all the way to the producer if a content has recently been requested through a point in the network closer to the consumer, disabling the use of the CS does not impact forwarding correctness.

5 EVALUATION

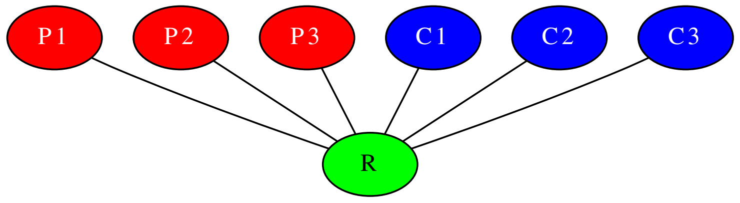
We conducted multiple evaluations of YaNFD to determine the correctness of its implementation (i.e., whether it was able to forward NDN traffic properly), whether it was able to interoperate with existing NDN forwarders, and to determine how well it performed compared to existing NDN forwarders for edge environments, both in terms of forwarding rates and overhead.

5.1 Correctness and Interoperability

An important consideration when developing a network application that intends to communicate with existing network applications is protocol compatibility. While the NFD forwarder is not itself the NDN protocol specification, it is the *reference implementation* of the NDN protocol specification. Therefore, because of this, and because NFD is used in most existing NDN networks, we evaluated our forwarder's correctness by evaluating its ability to interoperate with (i.e., correctly exchange Interest and Data packets) with NFD.



(a) Correctness evaluation scenario. Red indicates producer applications, blue indicates consumer applications, and green indicates forwarder software.



(b) Performance evaluation scenario. Red indicates producer applications, blue indicates consumer applications, and green indicates the forwarder software that was changed between YaNFD, NFD, and MW-NFD.

Figure 2: Topologies used for our evaluations.

During the development of YaNFD, we evaluated it against the NDN testbed [21] to ensure that it is both able to perform basic NDN forwarding operations and correctly interoperate with existing NDN forwarders – in particular, NFD, which is the only forwarder currently running on the testbed backbone. We found that we were able to successfully send NDN pings (from ndn-tools [22]) to various nodes on the testbed, some operating over multiple hops from where we connected to the testbed. As part of this, we pinged multiple testbed nodes throughout the testbed (connecting through a North American node), including testbed nodes in Asia, Europe, and South America.

Although the above evaluation confirms that YaNFD is, at a high level, compatible with existing NDN forwarders, it does not evaluate YaNFD's correctness is regards to more detailed specifics of the NDN protocol specification. These include the various optional fields present in Interest packets, such as MustBeFresh and CanBePrefix, as well as longest-prefix matching for names. To evaluate the correctness of our implementation with more complex aspects of the NDN protocol spec, we utilized the ndnchunks consumer-producer application pair from ndn-tools [22].

We conducted two experiments to transfer a 100 MB file between two directly-connected nodes virtualized using Ubuntu 20.04 running on VirtualBox. In each experiment, one node ran NFD and the other ran YaNFD, as discussed in Section 4.2.1. The producer and consumer applications were swapped between the YaNFD and NFD nodes in each experiment in order to provide coverage of both roles. In both, experiments ndnchunks was able to successfully transfer the file, which indicates that our forwarder is able to correctly interoperate with existing NDN networks running NFD, allowing it to be deployed in combination with NFD.

We have listed the fields present in the Interest and Data packets during these transfers (excluding the "Name" and "Content" fields) in Tables 1 and 2, respectively.

5.2 Performance and Overhead

5.2.1 Forwarding Performance. To evaluate the raw throughput available through an NDN forwarder, one can have one or more pairs of producers and consumers connected directly through a single forwarder. This excludes external factors such as network delay and the overhead of simulated or emulated topologies. We conducted file transfer tests using ndnchunks [22] for three NDN forwarders for edge environments: YaNFD, NFD [24], and MW-NFD [12].

We evaluated using three consumers requesting a different content from one of three producers on the same host. Our evaluations were conducted in terms of "goodput" (throughput from the perspective of the application layer) for three file sizes: 100 MB, 1 GB, and 5 GB. We averaged these values across the three consumers and then averaged this value over all trials for the same forwarder and the same file size. The results of our evaluations are shown in Figure 3.

We found that YaNFD was able to perform better than NFD for the larger two of the three file sizes, while it was just slower than NFD with the smallest of the three file sizes. Meanwhile, MW-NFD significantly outperforms the other forwarders in all three scenarios. We believe the combination of multi-threading and busy waiting allows MW-NFD to achieve much greater forwarding rates than NFD and YaNFD. Moreover, its lack of an output queue for faces (and instead having the forwarding/worker thread directly send on the socket) allows it to avoid the overhead of another queue, as exists between the forwarding thread and the face output thread in YaNFD.

Interestingly, YaNFD's performance appears to only increase with file size, almost doubling from 100 MB to 5 GB. This indicates that YaNFD has not yet reached its full forwarding capacity in these evaluations, unlike NFD and MW-NFD.

At the same time, we believe that YaNFD is unable to achieve the same forwarding rates as MW-NFD due to a lack of busy waiting (as discussed above) and the overhead of garbage collection in the Go programming language. We profiled the performance of YaNFD and found that it spent an average of 45% of its total runtime (over our five profiled runs) running the garbage collection mark worker [14] when transferring three 1 GB files. We believe that the lack of a garbage collector in C++ provides a significant performance advantage to MW-NFD and NFD, since they do not have this overhead.

Additionally, we ran YaNFD again with only a single forwarding thread and found that it performed significantly slower than YaNFD with eight forwarding threads. This indicates that YaNFD's multiple forwarding threads greatly improve its performance.

5.2.2 Overhead. It is important to consider the overhead of running an NDN forwarder on edge systems, such as desktop and notebook computers, since these systems run other applications. If using an NDN forwarder significantly impacted the battery life or responsiveness of a computer through excessive CPU and memory

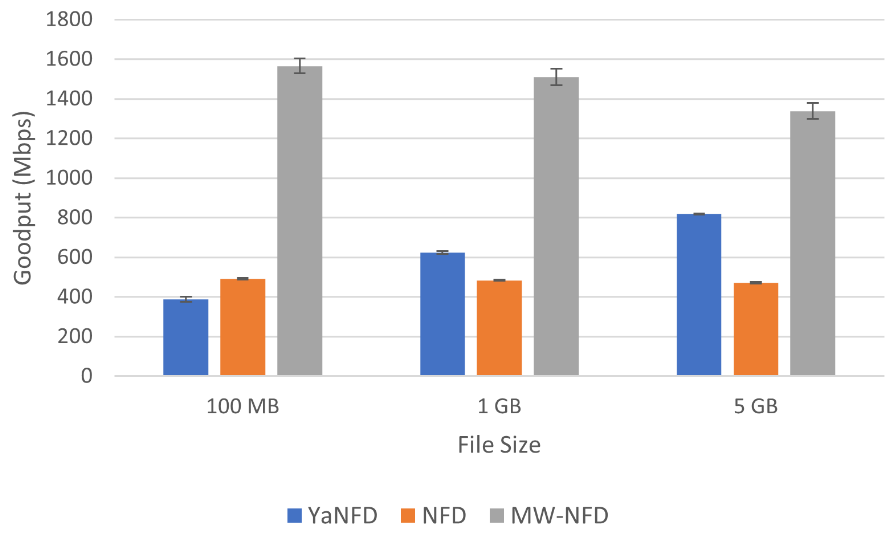


Figure 3: "Goodput" (application-layer throughput) of file transfers with ndnchunks when both the producer and consumer were running on the same host. A fixed-sized congestion window ($W = 100$ packets) was utilized. Error bars indicate the calculated 95% confidence interval using a normal distribution.)

usage, it could hamper the deployment of NDN in edge environments [7].

Therefore, we evaluated the runtime overhead of forwarding during the above scenario. Our evaluations were conducted in terms of both CPU utilization and memory utilization. We used psrecord [26] to track these metrics and generate a plot for each trial. The overheads of each forwarding during a trial of the scenario above are presented in Figure 4.

As can be seen in the figures, the CPU and memory utilizations of YaNFD and MW-NFD are significantly greater than NFD during file transfers. Notably, the CPU utilizations of YaNFD and MW-NFD are over 100% due to the use of multi-threading. MW-NFD's relatively consistent high CPU utilization is explained by its use of 100% CPU polling on the each on the 8 CPU cores it was assigned to. Meanwhile, YaNFD experiences significantly greater CPU utilization for the duration of the transfer – this is most likely caused by Go's multi-threading system, which spreads a large number of "goroutines" across a large number of worker threads. Since the machine we conducted these evaluations on has 64 physical cores (with 128 threads), the numbers seen in the figure indicate a large number of simultaneous workers on a large number of cores. We conducted another evaluation of YaNFD's overhead over a longer period of time while transferring a single file, with our results shown in Figure 4d. We found that YaNFD's CPU utilization dropped to effectively zero after the file had been successfully transferred, with brief, albeit minor spikes on a regular basis thereafter. Moreover, memory utilization decreased as it was eventually garbage collected. Therefore, given this and the results of our profiling of YaNFD, we believe that much of the overhead and performance reduction of YaNFD is from Go's garbage collection services. Meanwhile, this overhead is not present in either NFD and MW-NFD, as they are both written in C++, which does not have garbage collection.

[7]It is worth noting that if the forwarder were instead run only on a local router instead of on edge devices, the overhead factors may be different or less significant.

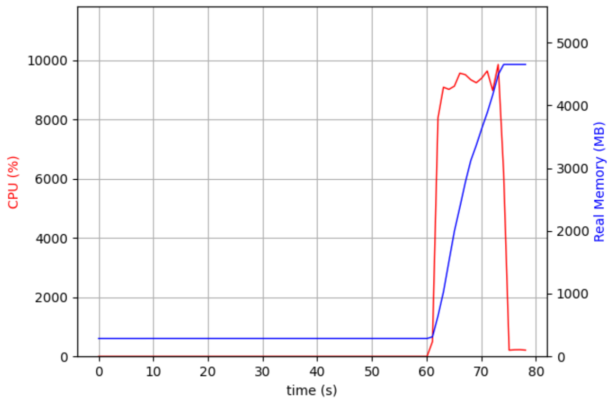
One significant takeaway from these graphs is that YaNFD shows very little idle CPU utilization. On edge systems that are shared with other applications, or even on routers shared with IP traffic, it is important to consider the interactions and impact on other applications. Therefore, our results show that YaNFD does not overutilize resources during idle times, potentially harming the performance of other applications, while still having significant room for improvement during forwarding.

At the same time, the storage footprint of YaNFD is significantly lower compared to both NFD and MW-NFD, with YaNFD's executable being 6.7 MB at the time of writing and the sizes of NFD and MW-NFD's executables in their latest releases being 124 MB and 144 MB, respectively, plus 144 MB for the ndn-cxx library upon which they both depend.

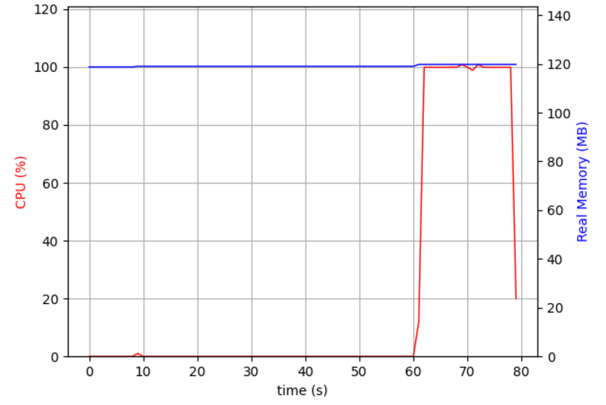
6 DISCUSSION

Our evaluations have shown that multi-threaded forwarding can significantly improve application-layer throughput in NDN networks at the edge. This has been shown by the perform of not just YaNFD, but also MW-NFD. However, one notable advantage that YaNFD has over MW-NFD is that it does not use busy waiting, which significantly lowers CPU overhead during idle periods, which are commonplace on edge devices.

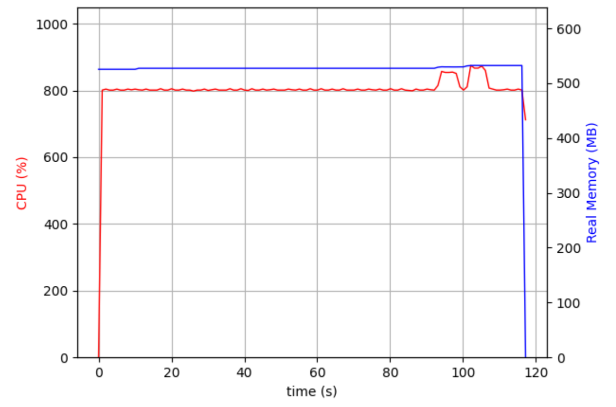
Additionally, our experiences developing YaNFD have given us important insights into the NDN forwarder development process. As we took "lessons learned" from the development processes and resulting software packages of previous NDN forwarders, we in turn present the lessons we have learned from our experience developing YaNFD. We believe these also provide interesting insights into developing high-throughput services in other fields.



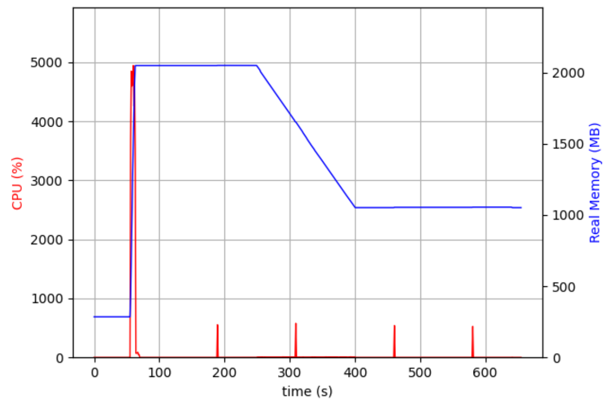
(a) YaNFD



(b) NFD



(c) MW-NFD



(d) YaNFD long-term overhead

Figure 4: A sample of memory and CPU overhead of each forwarder when transferring a 1 GB file using ndnchunks using a fixed window ($W = 100$ packets). Note that file transfers in a-c do not start until approximately 60 seconds in to allow the producer to load the file into memory.

First, we have discovered that it is important to consider the impact of a language’s garbage collection features when implementing high speed forwarding (or processing) services that cannot easily reuse packet (or other) data structures. During our evaluations, we found that garbage collection-related operations took up much of the runtime of YaNFD during simple file transfers. Meanwhile, the forwarders we compared against were both written in C++, a non-garbage collecting language, and did not have this overhead.

Additionally, we posit that one must consider the overhead of continuous polling when developing forwarders for edge environments. In these scenarios, hosts will likely be performing a large number of non-networking tasks that will be competing for the same limited memory and computation resources. Our forwarder is able to achieve better forwarding performance than the single-threaded NFD forwarder, while using almost none of the CPU during idle periods (as opposed to the multi-threaded MW-NFD, which performs 100% CPU polling even during idle periods). This demonstrates that edge environments do not require these overheads to perform well during NDN traffic forwarding.

6.1 Future Work

In addition to resolving the performance and overhead issues currently present in YaNFD, we plan to add new features to improve the codebase. We will now discuss a sampling of future work that we believe to be important. First, we hope to make forwarding strategies loadable at runtime, avoiding the need to recompile the forwarder every time a forwarding strategy needs to be added or modified. We have already made attempts to accomplish this in YaNFD, including using WASM [30] and Go plugins [13]. However, none of these approaches were suitable for YaNFD at this time: WASM strategy interfaces and logic require a significant amount of time to implement, taking away resources from implementing other, more critical aspects of the forwarder. Meanwhile, we encountered cyclic dependency issues between the strategy Go plugins and forwarder codebase, as strategies need to both be called from and make calls into the various data structures in the forwarder.

Additionally, at the moment, YaNFD is incompatible with the NLSR software router [17]. This is because YaNFD does not implement the face event management service due to time constraints during implementation. While YaNFD is able to fully function with manually-added routes, this is less than ideal in a real, dynamic network and therefore adding NLSR compatibility is a high priority.

Moreover, when connected directly to a producer application comes from prefix-based dispatching, YaNFD needs every Data packet received from that application to be dispatched to every thread that *every single one of its name prefixes* hashes to. This can have a large performance impact. The most straightforward solution to this issue would be to require support for PIT tokens in NDN applications. This would be a simple change and would simply require a bit more state be added to applications, while significantly improving the performance of YaNFD.

7 CONCLUSION

We have developed YaNFD as a multi-threaded alternative to existing software packet forwarders for the NDN architecture. As shown in our evaluations, YaNFD is able to achieve significantly better performance than the single-threaded NFD forwarder, but underperforms compared to other multi-threaded NDN forwarders like MW-NFD. At the same time, our results provide further evidence that multi-thread NDN forwarders for edge environments can greatly increase the throughput of NDN forwarding in typical edge applications. Therefore, we see little reason to not deploy multi-threaded NDN forwarders in devices capable of running more than one thread at a time.

We believe that YaNFD's current performance deficiencies compared to other multi-threaded NDN edge forwarders are primarily caused by the overhead of garbage collection in Go. Additionally, we believe the use of Go contributes to YaNFD's greater memory and CPU utilization. However, we found that the idle CPU utilization of YaNFD was significantly lower than existing multi-threaded NDN forwarders, as we avoided 100% CPU polling.

Additionally, our experience has shown us that successful NDN forwarders do not require a large development team or a complex codebase – our forwarder was developed over a few months largely by a single developer and features, at the time of writing, around 13 thousand lines of production code and compiles to an executable less than 7 MB in size. At the same time, NFD, the primary NDN forwarder in use for research and development at the time of this writing, features (excluding tests) close to 82k lines of code in the forwarder itself and 74k lines of code in the supporting library, with a combined executable and shared library size of approximately 268 MB. We hope that our experience and codebase can encourage the use of NDN in exciting new applications, which may not be able to bundle any of the existing NDN forwarders into their packages due to storage constraints.

ACKNOWLEDGMENTS

The authors would like to acknowledge the input of several long-term collaborators during the design and development of YaNFD, including, but not limited to: Alexander Afanasyev of Florida International University, Beichuan Zhang of the University of Arizona, Davide Pesavento and Junxiao Shi of the National Institute of Standards and Technology, Lan Wang of the University of Memphis, and Philipp Moll of the University of California, Los Angeles. These individuals were instrumental in helping us grasp the design decisions behind the NFD forwarder, as well as providing insightful feedback during the implementation and development of YaNFD.

Moreover, the authors would like to thank Sung Hyuk Byun of the Electronics and Telecommunications Research Institute for providing insight into the design of MW-NFD, as well as advice on experimental design. Beyond this, we would also like to acknowledge the insightful comments of the anonymous reviewers and our shepherd, Craig Partridge of Colorado State University.

This work was partially supported by the National Science Foundation under award CNS-1719403.

REFERENCES

[1] Ahmed Aboud, Haifa Touati, and Brahim Hnich. 2019. Efficient forwarding strategy in a NDN-based internet of things. *Cluster Computing* 22 (September 2019), 805–818. https://doi.org/10.1007/s10586-018-2859-7
[2] Advanced Micro Devices. [n.d.]. 2nd Gen AMD EPYC(tm) 7702P. https://www.amd.com/en/products/cpu/amd-epyc-7702p
[3] Alexander Afanasyev, Junxiao Shi, Beichuan Zhang, Lixia Zhang, Ilya Moiseenko, Yingdi Yu, Wentao Shang, Yanbiao Li, Spyridon Mastorakis, Yi Huang, Jerald Paul Abraham, Eric Newberry, Teng Liang, Klaus Schneider, Steve DiBenedetto, Chengyu Fan, Susmit Shannigrahi, Christos Papadopoulos, Davide Pesavento, Giulio Grassi, Giovanni Pau, Hang Zhang, Tian Song, Haowei Yuan, Hila Ben Abraham, Patrick Crowley, Syed Obaid Amin, Vince Lehman, Muktadir Chowdhury, Ashlesh Gawande, Lan Wang, and Nicholas Gordon. 2018. *NFD Developer's Guide*. Technical Report NDN-0021. Named Data Networking. https://named-data.net/wp-content/uploads/2018/07/ndn-0021-10-nfd-developer-guide.pdf Revision 10.
[4] Afanasyev Afanasyev, Cheng Yi, Lan Wang, Beichuan Zhang, and Lixia Zhang. 2015. SNAMP: Secure namespace mapping to scale NDN forwarding. In *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 281–286. https://doi.org/10.1109/INFCOMW.2015.7179398
[5] Sung Hyuk Byun, Jongseok Lee, Dong Myung Sul, and Namseok Ko. 2020. Multi-Worker NFD: An NFD-Compatible High-Speed NDN Forwarder. In *Proceedings of the 7th ACM Conference on Information-Centric Networking* (Virtual Event, Canada) *(ICN '20)*. Association for Computing Machinery, New York, NY, USA, 166–168. https://doi.org/10.1145/3405656.3420233
[6] Muktadir Chowdhury, Junaid Ahmed Khan, and Lan Wang. 2019. Smart Forwarding in NDN VANET. In *Proceedings of the 6th ACM Conference on Information-Centric Networking* (Macao, China) *(ICN '19)*. Association for Computing Machinery, New York, NY, USA, 153–154. https://doi.org/10.1145/3357150.3357408
[7] Alberto Compagno, Mauro Conti, Cesar Ghali, and Gene Tsudik. 2015. To NACK or Not to NACK? Negative Acknowledgments in Information-Centric Networking. In *2015 24th International Conference on Computer Communication and Networks (ICCCN)*. 1–10. https://doi.org/10.1109/ICCCN.2015.7288477
[8] Content-Centric Networking. [n.d.]. CCNx. https://github.com/ProjectCCNx/ccnx
[9] P. J. Courtois, F. Heymans, and D. L. Parnas. 1971. Concurrent Control with "Readers" and "Writers". *Commun. ACM* 14, 10 (Oct. 1971), 667–668. https://doi.org/10.1145/362759.362813
[10] DPDK Project. [n.d.]. Data Plane Development Kit. https://www.dpdk.org/
[11] DPDK Project. [n.d.]. DPDK NICS. https://core.dpdk.org/supported/nics/
[12] Electronics and Telecommunications Research Institute. [n.d.]. MW-NFD(Multi-Worker NFD): An NFD-compatible High-speed NDN Forwarder. https://github.com/etri/mw-nfd
[13] Go. [n.d.]. plugin. https://golang.org/pkg/plugin/
[14] Go. [n.d.]. runtime. https://golang.org/pkg/runtime
[15] Van Jacobson. 2019. Watching NDN's Waist: How Simplicity Creates Innovation and Opportunity. https://www.youtube.com/watch?v=69p78tfm29o Conference Presentation.

[16] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. 2009. Networking Named Content. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies* (Rome, Italy) *(CoNEXT '09)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1658939.1658941

[17] Vince Lehman, Ashlesh Gawande, Beichuan Zhang, Lixia Zhang, Rodrigo Aldecoa, Dmitri Krioukov, and Lan Wang. 2016. An experimental investigation of hyperbolic routing with a smart forwarding plane in NDN. In *2016 IEEE/ACM 24th International Symposium on Quality of Service (IWQoS)*. 1–10. https://doi.org/10.1109/IWQoS.2016.7590394

[18] Kai Lei, Jie Yuan, and Jiawei Wang. 2015. MDPF: An NDN Probabilistic Forwarding Strategy Based on Maximizing Deviation Method. In *2015 IEEE Global Communications Conference (GLOBECOM)*. 1–7. https://doi.org/10.1109/GLOCOM.2015.7417024

[19] Named Data Networking. [n.d.]. ndn-lite. https://ndn-lite.named-data.net/

[20] Named Data Networking. [n.d.]. NDN Packet Specification. https://named-data.net/doc/NDN-packet-spec/current/

[21] Named Data Networking. [n.d.]. NDN Testbed. https://named-data.net/ndn-testbed/

[22] Named Data Networking. [n.d.]. ndn-tools. https://github.com/named-data/ndn-tools

[23] Named Data Networking. [n.d.]. NDNLPv2. https://redmine.named-data.net/projects/nfd/wiki/NDNLPv2

[24] Named Data Networking. [n.d.]. NFD: Named Data Networking Forwarding Daemon. https://named-data.net/doc/NFD/current

[25] National Institute of Standards and Technology. [n.d.]. Hardware Known to Work with NDN-DPDK. https://github.com/usnistgov/ndn-dpdk/blob/main/docs/hardware.md

[26] Thomas Robitaille. [n.d.]. psrecord. https://github.com/astrofrog/psrecord

[27] Klaus Schneider, Cheng Yi, Beichuan Zhang, and Lixia Zhang. 2016. A Practical Congestion Control Scheme for Named Data Networking. In *Proceedings of the 3rd ACM Conference on Information-Centric Networking* (Kyoto, Japan) *(ACM-ICN '16)*. Association for Computing Machinery, New York, NY, USA, 21–30. https://doi.org/10.1145/2984356.2984369

[28] Junxiao Shi, Davide Pesavento, and Lotfi Benmohamed. 2020. NDN-DPDK: NDN Forwarding at 100 Gbps on Commodity Hardware. In *Proceedings of the 7th ACM Conference on Information-Centric Networking* (Virtual Event, Canada) *(ICN '20)*. Association for Computing Machinery, New York, NY, USA, 30–40. https://doi.org/10.1145/3405656.3418715

[29] Won So, Ashok Narayanan, and David Oran. 2013. Named data networking on a router: Fast and DoS-resistant forwarding with hash tables. In *Architectures for Networking and Communications Systems*. 215–225. https://doi.org/10.1109/ANCS.2013.6665203

[30] WebAssembly. [n.d.]. WebAssembly. https://webassembly.org/

[31] Cheng Yi, Alexander Afanasyev, Ilya Moiseenko, Lan Wang, Beichuan Zhang, and Lixia Zhang. 2013. A case for stateful forwarding plane. *Computer Communications* 36, 7 (2013), 779–791. https://doi.org/10.1016/j.comcom.2013.01.005

[32] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, kc claffy, Patrick Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. 2014. Named Data Networking. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 66–73. https://doi.org/10.1145/2656877.2656887

[33] Zhiyi Zhang, Yingdi Yu, Haitao Zhang, Eric Newberry, Spyridon Mastorakis, Yanbiao Li, Alexander Afanasyev, and Lixia Zhang. 2018. An Overview of Security Support in Named Data Networking. *IEEE Communications Magazine* 56, 11 (2018), 62–68. https://doi.org/10.1109/MCOM.2018.1701147