Implementing Named Data Networking on FABRIC

Ashwin Nair
School of Computing and Augmented Intelligence
Arizona State University
Tempe, US
ajnair1@asu.edu

Jason Womack
School of Computing and Augmented Intelligence
Arizona State University
Tempe, US
rjwomack@asu.edu

Toby Sinkinson
School of Computing and Augmented Intelligence
Arizona State University
Tempe, US
gsinkins@asu.edu

Yingqiang Yuan
School of Computing and Augmented Intelligence
Arizona State University
Tempe, US
yyuan76@asu.edu

***Abstract*—Achieving high-speed Named Data Networking (NDN) forwarding on commodity hardware has now been shown to be possible. Meanwhile, the roll out of FABRIC, a national research infrastructure equipped with large amounts of compute and storage and interconnected by high speed optical links, makes it possible to easily recreate, and reproduce high speed NDN forwarding on commodity hardware to incrementally improve upon and continue innovation of the data structures and algorithms that make it possible. This project creates a notebook to facilitate this further development.**

***Index Terms*—Named data networking, DPDK, Hugepages, FABRIC, High-speed forwarding, Kernel bypass, Flow bifurcation, Reproducibility**

I. Introduction and Motivation

The Named Data Networking (NDN) project addresses the architectural mis-match [7] of today's Internet architecture and its usage. NDN replaces IP packets with data packets as the fundamental unit of communication, yet it maintains the same elegant and powerful hour glass shaped architecture. Keeping this same shape is in part why NDN can be overlaid over traditional IP architecture. This means that while NDN is able to innovate to meet the changing needs of modern internet use, such as the substantial increase in multimedia content delivery, the increased need for better security and privacy, the needs of IoT sensor networks [8], or moving large amounts of genomic data [9], it can innovate without requiring widespread hardware or structural upgrades. NDN development can be incremental and in parallel to other innovative strategies.

One of the challenges for NDN arises from its variable-length hierarchical naming of data strategy, which similar to DNS lookups [10], creates a bottleneck for NDN forwarding throughput. A team of researchers at NIST developed an NDN Data Plane Development Kit [1] that addresses this bottleneck and shows that is possible to achieve 100 Gbps forwarding rates on X86 commodity hardware by bypassing kernel space and eliminating system calls and interrupt handling. They made their development kit open source and our project [12] seeks to instantiate it on FABRIC for further development to allow other researchers to extend the project on a large scale testbed with sufficient compute and storage.

II. Related Work

A. Background Readings

1) NDN: The original idea for NDN was proposed by Zhang *et al* as an alternative to traditional host-centric IP networks [2]. NDNs have since been a topic for active research, especially for data-centric applications, such as multimedia content delivery. The fact that NDN does not always need to contact a host (a data center in the case of multimedia content delivery) means that intermediate forwarders (stateful devices used in NDN to forward packets) can effictively act as CDNs which would drastically reduce service times. NDN can also be used for ad hoc networks where the challenges in connectivity from one host to another can create high latency [3].

Packets in NDN are either interest packets requested by a receiver, or data packets signed and produced by a producer. All communication is driven by the receiver and to manage the traffic, NDN requires three data structures, a Forwarding Information Base (FIB), and Pending Interest Table (PIT), and a Content Store (CS). The FIB is used by the router to keep track of the interfaces on which an interest packet arrived. In turn, the FIB is used to forward the data packets according to the list of interfaces that requested interest in the data. While waiting for data packets to arrive, the interest and the first interface that requested the data are kept in the PIT. When multiple interests arrive, only the first one is sent upstream to reduce traffic. Once data arrives, it is both forwarded to all receivers that requested it and stored locally in the CS, allowing faster retrieval for future requests. Once cached, the interest is removed from the PIT, since it is no longer pending.

2) FABRIC: FABRIC is a testbed spanning multiple sites in the USA and can be extended to include 4 additional nodes in Asia and Europe. The testbed provides high programmability as well as large amounts of compute and storage [6]. The large amounts of storage were integral to our implementation of NDN. Additionally, FABRIC provides one of the network interfaces known to work for the project, specifically the NVIDIA Mellanox ConnectX SmartNIC which has optimized 10/100 Gbps Ethernet adapters and allows the splitting of user and kernel space traffic.

B. Contextual Work

There have been a number implementations of NDN on dedicated NDN test beds and more recently on commodity hardware [1] [3]. One such implementation, NDN-DPDK had the goal of achieving high throughput by utilizing the DPDK framework. This is the implementation we migrated onto FABRIC and includes the traffic generator module, the forwarder module, and the file server module from the NDN-DPDK codebase.

Caching, directing, and forwarding all depend upon being able to quickly lookup the data requested or data produced using the name of the data. Because variable length names introduce large latency during the forwarding process, NDN-DPDK developed a method to take advantage of parallelism from multi-core CPU's and share NDN tables as much as possible. To achieve this, NDN-DPDK builds upon open source libraries from DPDK [11], a fast packet processing development kit, that allows data structures to be pre-allocated in 1 GB hugepages. Hugepages are used to reduce the number of pages, which in turn reduces page address translation time because there are fewer page table entries stored in cache. This in turn, reduces the likelihood of cache misses. Using DPDK also eliminates the need to call *malloc()* on the packet processing hot path. Additionally, each CPU, memory DIMM and PCIe are manged through a NUMA socket.

NDN-DPDK adds several data structures to ensure that NDN routes correctly with multiple threads; A Name Dispatch Table is added to make sure that interests are sent to a corresponding thread and a PIT token was added to ensure that neighboring routers agree on a hash function to assign interests to threads before routers communicate. This also ensures that common name prefixes are sent to the same thread.

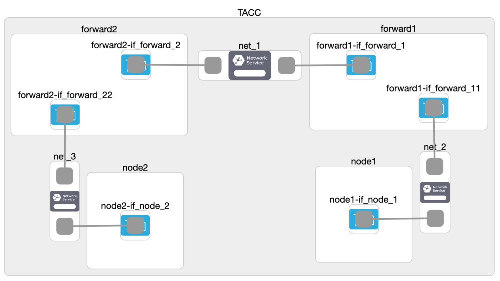


Fig. 1: FABRIC topology on site TACC to demonstrate the basic NDN setup being used

III. Experimental Set-Up

We broke up the installation and activation of NDN-DPDK into four phases, each having a distinct goal and result:

A. Phase I: Install the NDN-DPDK Package

To be able to run NDN on FABRIC, every node in the NDN network must have the base NDN-DPDK package installed. The necessary steps to do this are provided in the notebook installation and are outlined below.

1) Install MLNX OFED NVIDIA drivers to support ConnectX-6 network interfaces on FABRIC.
2) Install NDN-DPDK dependencies using the provided script `ndndpdk-depends.sh`
3) Clone the NDN-DPDK and DPDK GitHub.
4) Install the NDN-DPDK package onto the nodes using the script `install.sh`

B. Phase II: Activate the Forwarder

To create forwarding nodes, the NDN-DPDK package must be configured to handle forwarding. In order to do this, a large portion of the machine's RAM must be allocated to hugepages for the cache. The default configuration we are using on FABRIC is 64 1GB hugepages, which requires a slightly higher amount of RAM to be reserved (roughly 80GB is sufficient). The steps to configure this and the full activation are provided below.

1) Configure the hugepages using `dpdk-hugepages.py` from the DPDK package.
2) Activate the forwarder using `ndndpdk-ctrl activate-forwarder`.
3) Set up the network interfaces using `create-ether-port` on their MAC address.
4) Bind the interfaces using `create-eth-face`.

Once these steps are finished, the setup can be tested by using *NDNping*, a tool to ping based on namespace. With two nodes activated as a forwarder, one can start an *NDNping* on a sample namespace (ex. /example/P) and the other must insert an FIB entry using `insert-fib` with the example namespace and *NDNping* as a client.

C. Phase III: Activate the Traffic Generator

Traffic generators are activated in a similar way using the traffic generator activation configuration. Due to the variability in use cases for an NDN traffic generator, it must be activated with a parameter set using `trafficgen.schema.json` in the NDN-DPDK package. Different traffic patterns can be set, as well as the role of consumer or producer. Some of the patterns include randomized interest packets, probability based interest packets, and packet generation rate. Once the JSON file is set, the traffic generator can be activated using the commands below.

1) Activate the traffic generator using the parameters set passed into `activate-trafficgen`.
2) Create and bind the network interfaces the same as Phase II.
3) Start the traffic generator using `start-trafficgen` with the MAC address of the interface.

After this is finished, the installation can be verified in the same way as Phase II: using *NDNping* with another forwarder.

D. Phase IV: Activate the File Server

The file server uses a forwarder in order to send the data at high speeds, and requires both forwarder activation and file server activation running on the same node. In order to do this, a flag has to be set in order to differentiate the two processes on the node. The steps to do so are defined below.

1) Perform phase II steps on the node to activate the forwarder on the default port (3030).
2) Using the flag *–gqlserver http:127.0.0.1:3031* run `activate-fileserver`.

Once the file server is activated, it can be verified using the *ndncat* tool. This is a tool that builds, sends, and receives files based on segment size and namespace. Using this, two file server activated nodes can send a segmented file across them and then verify the SHA256 sum to see if it transferred correctly.

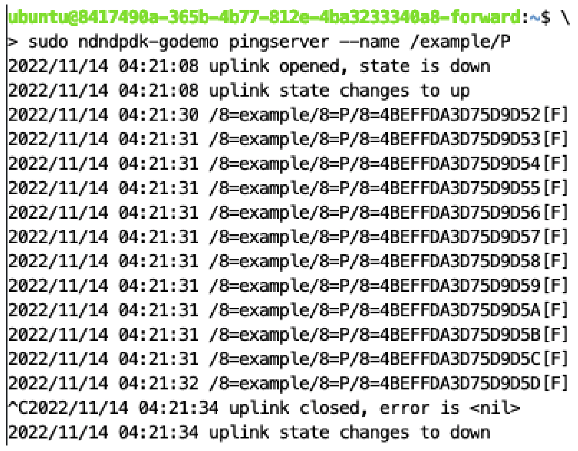
IV. Results

All phases have been successfully installed onto FABRIC nodes and tested using their respective methods. The installation of the phases has also been saved and automated using a Jupyter Notebook for the project on FABRIC.

Phase I of the installation is automated in the notebook, and the installation is validated at the end of the execution. Using the notebook, all resources reserved in the slice will execute phase I and install the base package. While performing the base installation, we ran into a few different challenges. FABRIC uses NVIDIA ConnectX-6 network interfaces which require drivers to be installed prior to any other DPDK installation. We didn't know of the necessity for these drivers, which held our installation up by multiple days. We also wanted to match NIST's demonstrated topology using 64 1GB hugepages, but we found that because we only reserved 64GB of RAM, we could not reserve the full amount of hugepages and this also took time to find out.

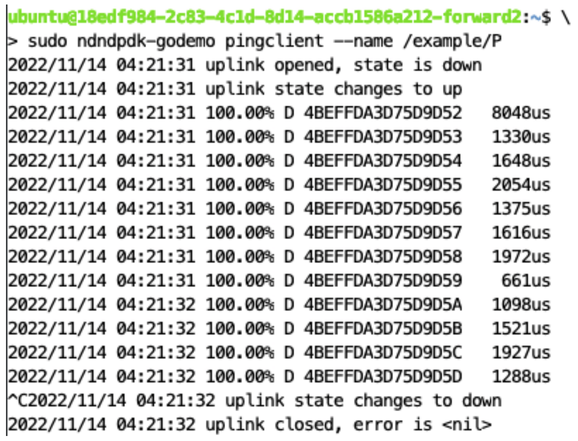
We were able to install the second phase successfully onto FABRIC nodes and we made the installation automatic through our Jupyter Notebook. Using the notebook, Phase II is activated onto all nodes in the FABRIC slice that start in *forward*. FABRIC nodes must be reserved with this naming scheme in mind. Once the forwarders are activated, the network interfaces can be tested using *NDNping*. Figures 2 and 3 show a successful *NDNping* from one forwarder acting as a server to another forwarder acting as a client using the topology in figure 1. In order to ping, the client forwarder must contain a FIB entry of the namespace and ID of the interface connecting the two forwarders using `insert-fib`.

Activating the traffic generator in Phase III was not perfectly implemented, but we were able to make the traffic generator work. The `watch-trafficgen` feature of the traffic generator, which allows the user to see activate traffic flowing through the generator, never successfully activated due to an error in the GraphQL mutations necessary in the NDN-DPDK code. We were unable to pass in the proper mutations and this kept throwing an error stating the mutation *sqeSubmit* was



```
ubuntu@8417490a-365b-4b77-812e-4ba3233340a8-forward:~$ \
> sudo ndndpdk-godemo pingserver --name /example/P
2022/11/14 04:21:08 uplink opened, state is down
2022/11/14 04:21:08 uplink state changes to up
2022/11/14 04:21:30 /8=example/8=P/8=4BEFFDA3D75D9D52[F]
2022/11/14 04:21:31 /8=example/8=P/8=4BEFFDA3D75D9D53[F]
2022/11/14 04:21:31 /8=example/8=P/8=4BEFFDA3D75D9D54[F]
2022/11/14 04:21:31 /8=example/8=P/8=4BEFFDA3D75D9D55[F]
2022/11/14 04:21:31 /8=example/8=P/8=4BEFFDA3D75D9D56[F]
2022/11/14 04:21:31 /8=example/8=P/8=4BEFFDA3D75D9D57[F]
2022/11/14 04:21:31 /8=example/8=P/8=4BEFFDA3D75D9D58[F]
2022/11/14 04:21:31 /8=example/8=P/8=4BEFFDA3D75D9D59[F]
2022/11/14 04:21:31 /8=example/8=P/8=4BEFFDA3D75D9D5A[F]
2022/11/14 04:21:31 /8=example/8=P/8=4BEFFDA3D75D9D5B[F]
2022/11/14 04:21:31 /8=example/8=P/8=4BEFFDA3D75D9D5C[F]
2022/11/14 04:21:32 /8=example/8=P/8=4BEFFDA3D75D9D5D[F]
^C2022/11/14 04:21:34 uplink closed, error is <nil>
2022/11/14 04:21:34 uplink state changes to down
```

Fig. 2: *NDNping* on FABRIC node on the site TACC running NDN-DPDK pinging the namespace /example/P as a server. The ping output shows that chunks of /example/P are being forwarded (F) to another node. The last index increments per chunk sent.

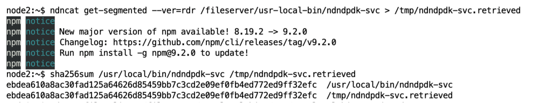


```
ubuntu@18edf984-2c83-4c1d-8d14-accb1586a212-forward2:~$ \
> sudo ndndpdk-godemo pingclient --name /example/P
2022/11/14 04:21:31 uplink opened, state is down
2022/11/14 04:21:31 uplink state changes to up
2022/11/14 04:21:31 100.00% D 4BEFFDA3D75D9D52   8048us
2022/11/14 04:21:31 100.00% D 4BEFFDA3D75D9D53   1330us
2022/11/14 04:21:31 100.00% D 4BEFFDA3D75D9D54   1648us
2022/11/14 04:21:31 100.00% D 4BEFFDA3D75D9D55   2054us
2022/11/14 04:21:31 100.00% D 4BEFFDA3D75D9D56   1375us
2022/11/14 04:21:31 100.00% D 4BEFFDA3D75D9D57   1616us
2022/11/14 04:21:31 100.00% D 4BEFFDA3D75D9D58   1972us
2022/11/14 04:21:31 100.00% D 4BEFFDA3D75D9D59    661us
2022/11/14 04:21:32 100.00% D 4BEFFDA3D75D9D5A   1098us
2022/11/14 04:21:32 100.00% D 4BEFFDA3D75D9D5B   1521us
2022/11/14 04:21:32 100.00% D 4BEFFDA3D75D9D5C   1927us
2022/11/14 04:21:32 100.00% D 4BEFFDA3D75D9D5D   1288us
^C2022/11/14 04:21:32 uplink state changes to down
2022/11/14 04:21:32 uplink closed, error is <nil>
```

Fig. 3: *NDNping* on FABRIC node on the site TACC running NDN-DPDK pinging the namespace /example/P as a client. This node has the FIB entry for /example/P bound to the network interface connected to the server node. The output shows it's receiving chunks with 0 loss (100% data transmission) from the server

missing, and code changes did not solve this. We were able to verify the traffic generator worked through other means though. Every network interface contains statistics of the data running through it, and using this we could get the count of transmitted/received packets that made it through the interface. With two traffic generators activated in the topology in figure 1, we could see that the producer activated traffic generator was sending packets while the consumer was receiving the same amount of packets.

The file server was able to be fully installed on FABRIC but not fully automated in the Jupyter Notebook due to the multiple processes involved in it. Using the topology in figure 1, a node with the file server activated as a server and another



```
node2:~$ ndncat get-segmented --ver=rdr /fileserver/usr-local-bin/ndndpdk-svc > /tmp/ndndpdk-svc.retrieved
npm notice
npm notice New major version of npm available! 8.19.2 -> 9.2.0
npm notice Changelog: https://github.com/npm/cli/releases/tag/v9.2.0
npm notice Run npm install -g npm@9.2.0 to update!
npm notice
node2:~$ sha256sum /usr/local/bin/ndndpdk-svc /tmp/ndndpdk-svc.retrieved
ebdea610a8ac30fad125a64626d85459bb7c3cd2e09ef0fb4ed772ed9ff32efc  /usr/local/bin/ndndpdk-svc
ebdea610a8ac30fad125a64626d85459bb7c3cd2e09ef0fb4ed772ed9ff32efc  /tmp/ndndpdk-svc.retrieved
```

Fig. 4: *ndncat* showing a client node activated as a file server on the site TACC getting a file from a file server in the server role. The file is verified using the SHA256 sum between the two files to verify the transfer

with it activated as a client were able to send files across the network. This was verified using *ndncat* as shown in figure 4. This shows a client activated file server that is getting a segmented version of the NDN-DPDK package from another node acting as the server. The SHA256 sums were verified, showing that the activation and usage was successful.

V. Summary, Conclusions, and Future Work

We worked to implement NDN on the FABRIC testbed. Our proposed design was to use the NIST NDN-DPDK package [4] and install it on a FABRIC slice using the different NDN components (forwarder, consumer, producer) in order to send namespace based traffic across the FABRIC network at high speeds. Using FABRIC Jupyter Notebooks, this NDN design is repeatable on other FABRIC sites, though some modifications are necessary on sites which restrict to IPv6 traffic only.

We were able to successfully implement our design on the testbed, with multiple FABRIC nodes being able to send traffic to each other using namespace based routing. We aim to use our implementation to prove that NDN can move data across the network between two nodes more efficiently than over the standard IP network.

Our immediate next goal is to experiment with NDN across the FABRIC network. Specifically, we want to compare the performance of NDN with a standard IP network with varying traffic types (such as transfer of large files or video straming applications) and network configurations. To do this we will need to add a measurement process using the scratch area in hugepages by storing collected measurements on the FIB entry itself.

Using the FABRIC NDN Jupyter Notebook opened up possibilities for future work in the NDN research space with FABRIC. It is possible to scale the notebook to other sites and and configure the slice to fit specific research needs. The open source nature of the NIST NDN-DPDK project [4] also allows for modifications to be done to the NDN protocol itself opening up many other possibilities. An example of this is implementing a predictive algorithm which automatically supplies data packets ahead based on the interest packets that came before it.

References

[1] Shi, Junxiao and Pesavento, Davide and Benmohamed, Lotfi, "NDN-DPDK: NDN forwarding at 100 Gbps on commodity hardware" in Proceedings of the 7th ACM Conference on Information-Centric Networking, 2020, pp. 30–40

[2] Zhang, Lixia and Afanasyev, Alexander and Burke, Jeffrey and Jacobson, Van and Claffy, KC and Crowley, Patrick and Papadopoulos, Christos and Wang, Lan and Zhang, Beichuan, "Named data networking" in ACM SIGCOMM Computer Communication Review, vol.44, pp. 66–73, 2014

[3] Divya Saxena, Vaskar Raychoudhury, Neeraj Suri, Christian Becker, Jiannong Cao, "Named Data Networking: A survey, Computer Science Review, Volume 19, 2016, Pages 15-55, ISSN 1574-0137, https://doi.org/10.1016/j.cosrev.2016.01.001. (https://www.sciencedirect.com/science/article/pii/S1574013715300599)

[4] NDN-DPDK: High-Speed Named Data Networking Forwarder [https://github.com/usnistgov/ndn-dpdk]

[5] NDN Traffic Generator [https://github.com/named-data/ndn-traffic-generator]

[6] I. Baldin et al., "FABRIC: A National-Scale Programmable Experimental Network Infrastructure," in IEEE Internet Computing, vol. 23, no. 6, pp. 38-47, 1 Nov.-Dec. 2019, doi: 10.1109/MIC.2019.2958545.

[7] Vishwa Pratap Singh and R.L. Ujjwal, "A walkthrough of name data networking: Architecture, functionalities, operations and open issues" in Sustainable Computing: Informatics and Systems, 2020, pp. 100419

[8] Emmanuel Baccelli, Christian Mehlis, Oliver Hahm, Thomas C. Schmidt, and Matthias Wählisch. 2014. Information centric networking in the IoT: experiments with NDN in the wild. In Proceedings of the 1st ACM Conference on Information-Centric Networking (ACM-ICN '14). Association for Computing Machinery, New York, NY, USA, 77–86. https://doi.org/10.1145/2660129.2660144

[9] Susmit Shannigrahi, Chengyu Fan, Christos Papadopoulos, and Alex Feltus. 2018. NDN-SCI for managing large scale genomics data. In Proceedings of the 5th ACM Conference on Information-Centric Networking (ICN '18). Association for Computing Machinery, New York, NY, USA, 204–205. https://doi.org/10.1145/3267955.3269022

[10] Ke-Jou Hsu, James Choncholas, Ketan Bhardwaj, and Ada Gavrilovska. 2020. DNS Does Not Suffice for MEC-CDN. In Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets '20). Association for Computing Machinery, New York, NY, USA, 212–218. https://doi.org/10.1145/3422604.3425931

[11] DPDK: Data Plane Development Kit [https://github.com/DPDK/dpdk]

[12] FABRIC-NDN: Implementing NDN on the FABRIC Testbed[https://github.com/initialguess/fabric-ndn]