Real-World Implementation of Function Chaining in Named Data Networking for IoT Environments

Yohei Kumamoto
Waseda University
yoheimmtw@akane.waseda.jp

Hiroki Yoshii
Microsoft Japan Co., Ltd.
ramenhy@akane.waseda.jp

Hidenori Nakazato
Waseda University
nakazato@waseda.jp

***Abstract*—In this paper, we discuss how to implement function chaining in Named Data Networking (NDN), an incarnation of information centric networking technology, for real-world IoT environments. We explain our new architecture, called NDN-FC, for function chaining over NDN, and how to extend existing NDN software to support function chaining. The key features discussed in this paper are Interest and Data packet structure, forwarding methods, and segmentation and reassembly methods of a content. Even in IoT environments, it is possible that most content, such as image and video, does not fit into a single Data packet. Segmentation and reassembly of a content is therefore crucial. The feasibility of our proposed protocol for segmentation and reassembly is displayed through a prototype implementation. In order to support lightweight operation of functions, the implementation is extended to use Docker container technology to run functions. The performance of Docker implementation and virtual machine implementation are compared.**

***Index Terms*—NDN, function chaining, IoT, segmentation, reassembly**

I. Introduction

In recent years, the number of IoT devices have been increasing rapidly. Consequently, there has also been a growth in IoT applications and services, many of which require low latency such as factory automation and autonomous driving.

In response to this, edge computing has been proposed. Edge computing is the idea that by placing computing infrastructures closer to the edge devices, data can be processed quicker and more efficiently compared to sending it out to a cloud server [1]. However, the problem with this idea is that it is heavily reliant on the location and processing power of a single edge.

As a solution to this problem, we will apply the idea of Service Function Chaining (SFC). With SFC, users can control traffic through software to route packets to the desired network services, which creates a virtual chain of network services [2]. We use this idea and place computing resources throughout the network, and run functions on them to process the data. By chaining these functions, data can be processed in a sequential manner to obtain the desired output. We can also strategically and dynamically place these functions to prevent and relieve network congestion and load. In order to achieve this, we will adopt a new communication protocol called Named Data Networking (NDN). NDN routes data by Content Name as opposed to the traditional IP, which routes according to location. This will allow a more intuitive and simple management of routing. This will be called NDN Function Chaining (NDN-FC). NDN-FC provides us the flexibility to choose an appropriate function instance out of many available instances with the same function name. By combining SFC and NDN we can make a more efficient IoT network.

In this paper we will discuss how to extend NDN to support Function Chaining and in-network processing.

The research leading to these results has been supported by the EU-JAPAN initiative by the EC Horizon 2020 Work Programme (2018-2020) Grant Agreement No.814918 and Ministry of Internal Affairs and Communications "Federating IoT and cloud infrastructures to provide scalable and interoperable Smart Cities applications, by introducing novel IoT virtualization technologies (Fed4IoT)."

II. Related Work

A. Named Function Networking (NFN)

C. Tschudin and M. Sifalakis [3] proposed a solution called Named Function Networking (NFN). In comparison to NDN, NFN uses Interest packets to request functions and parameters through the use of λ-expressions. In an NFN environment, NFN routers are deployed in addition to normal NDN routers. NFN routers have computing capabilities, and are responsible for resolving NFN specific expressions and executing functions. A simple architecture of NFN is shown in Fig. 1.

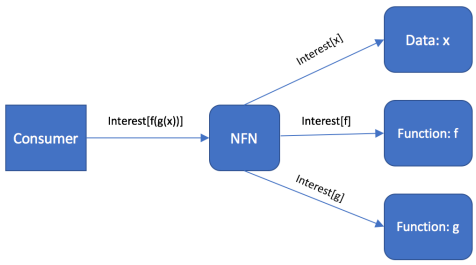

Fig. 1. NFN Example

In this example, the consumer sends out an Interest packet with the name */f (/g (/x))*. */f* and */g* represent functions, and */x* represents the parameter data. Functions exist as byte code, and are passed around in binary form [4]. Data will be retrieved in the following way.

1) Interest */f(/g(/x))* will be routed towards an NFN router.
2) When it arrives at the NFN router, the name will be parsed into each component: */f*, */g*, and */x*.

3) At this point, the NFN router will create Interest packets for each component and forward them to their respective producers.
4) When Data packets for each Interest are received, the NFN router will execute */f (/g (/x))*.
5) Finally, the result will be sent back to the consumer.

Additionally, through the use of λ-expressions, it is possible to express a single function chain in multiple ways. For example, *func(data)* can be expressed in the following ways [4]:

1) **func data**
2) **(λzy.z y) func data**
3) **(λy.func y) data**
4) **(λz.z data) func**

The benefit of this is that the same result can be obtained through multiple different paths. For example, expressions 3. and 4. can be represented as */data/(λy.func y)* and */func/(λz.z data)* respectively. This means */data* or */func* can be used for forwarding by using the above expressions to obtain the same content. If a result cannot be found on the path */data*, the NFN router can use */func* to find or compute the result.

Although NFN is a versatile and resilient architecture, it requires the user to understand λ-calculus to make full use of its capabilities. This makes Interest names complex, which can negate NDN's benefit of simple management. This complexity can also make troubleshooting difficult.

B. ICN Function Chaining (ICN-FC)

L. Liu et al. [5] proposed a solution called ICN Function Chaining (ICN-FC). ICN-FC uses a simpler approach than NFN for chaining functions. It uses the ← symbol to connect functions within the Interest name. A simple example is shown in Fig. 2.

Fig. 2. ICN-FC Example

In this example, *A* and *B* represent functions, and *Data* represents the producer where the required data is located. In order to chain functions, the Interest name will be set to */A←/B←/Data*. Each time the Interest packet passes through a function, that function will be removed from the namespace. By removing the function from the namespace, it is possible to dynamically change the route of the Interest to a different function.

Although ICN-FC proposes the concept of one mechanism to perform function chaining in NDN environment, some detail is missing. INC-FC creates 2 PIT entries for each forwarding of an Interest packet with function chaining specification and one of the entries is for the processing of a function. The face specified by the entry cannot not be derived from face where the Interest packet is arrived. How to specify the entry is missing from the proposal. Also, ICN-FC is evaluated assuming video processing. Although the segmentation / reassembly between the source of a video and the first function to process the video is discussed, how to transfer a large video file between functions is also missing. This paper fills the gap between the ICN-FC proposal and real implementation.

III. NDN Function Chaining (NDN-FC) Architecture

A. Interest Packet Format

In NDN-FC, we have added a Function Name field to the Interest packet format from the original packet format. In this field, the chain of functions will be listed. The result is shown in Fig. 3. The Data packet will be kept the same as the original packet format.

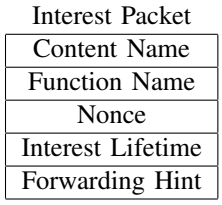

Fig. 3. NDN-FC Interest Packet Format

In contrast to ICN-FC, in NDN-FC, the Content name will be kept static. Instead, the Function Name field will be responsible for forwarding Interest packets to functions. The reason behind keeping Content names static is to allow easier packet tracing for troubleshooting purposes, and prevent the Content Name from becoming overly complex.

B. NDN-FC Basic Architecture

For basic NDN-FC forwarding, we will use a similar method to ICN-FC. Function flow will be set in the Function Name field. Functions will be separated by / symbol. After the Interest packet passes through a function, that function will be removed from that field. It is possible to register function names in the FIB, which will forward Interest packets. If there are no functions specified, the Interest packet will be forwarded by the Content Name. Therefore, the Content Name will effectively represent the parameter for the first function. It should be noted that the functions will be executed in the reverse order of the Function Name field, since the Data packet goes in the reverse order of the Interest packet. An example of NDN-FC is shown in Fig. 4.

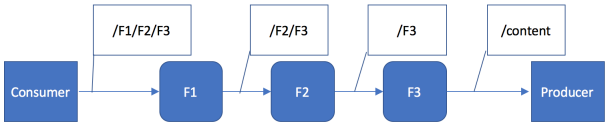

Fig. 4. NDN-FC Example

IV. Implementing NDN-FC

A. ndn-cxx Extension

ndn-cxx is a C++ library, implementing NDN primitives that can be used to implement various NDN applications [6].

In order to support NDN-FC, the Interest packet format must be modified. This only requires the Function Name field to be added. Fig. 3 from Section III will represent what the packet format will look like. The basic structure of the Content Name field will be used for the Function Name field.

B. NFD Extension

NDN Forwarding Daemon (NFD) is the core component of NDN, and is responsible for routing NDN packets. We extended NFD to support edge computing and SFC.

1) Function Forwarding Strategy: Using the strategy API of NFD, we created a new forwarding strategy for function chaining. The basic flow of the strategy is shown in Fig. 5.

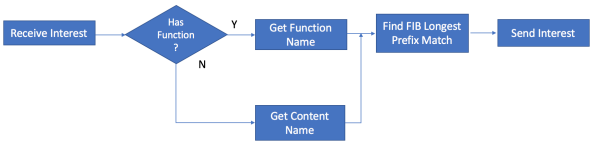

Fig. 5. NDN-FC Forwarding Strategy

For NDN-FC we will use the *afterReceiveInterest* trigger, which fires after NFD receives an Interest packet. In this strategy, when *afterReceiveInterest* is triggered, the Function Name field will be checked using *hasFunction* described in Section IV-A. When an Interest packet is received by NFD, i.e., router, NFD checks the Function Name filed in the Interest packet. If the packet has a Function Name, the Function Name will be retrieved. If the Function Name field is empty, the Content Name will be retrieved. Since the Content Name and Function Name fields are similar in structure, the Forwarding Information Base (FIB), which holds routing information, does not need to differentiate between the two to do a FIB look-up for packet forwarding. When a matching entry is found, the Interest will be sent out to the corresponding interface, or *face* in NDN term.

2) PIT In-record Sequence Numbers: The difference between functions and consumers/producers is that it both sends and receives Interest and Data packets. On the contrary, consumers only send Interest packets and receive Data packets. Producers only receive Interest packets and send Data packets. From a technical stand point, a function is a combination of a consumer and a producer. However, this brings up a problem with the Pending Interest Table (PIT), which holds the reverse path of Interest packets to their original sender, i.e., consumer, while using our proposed architecture. When an Interest packet goes through a function node, it enters the NFD router of that node twice. The path of an Interest packet is shown in Fig. 6.

The problem is that the PIT records two in-records(① and ② in Fig. 6) linked to a single PIT entry, i.e., a single Content Name. This is problematic because when the Data packet

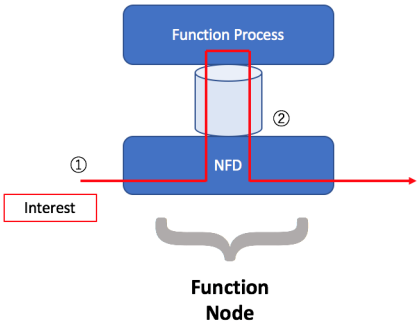

Fig. 6. Interest Packet Path

corresponding to the Interest packet is returned, it will be sent out to both faces at once. This is shown in Fig. 7.

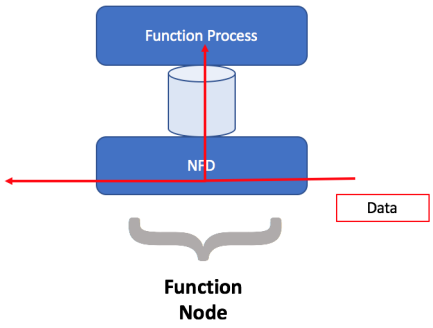

Fig. 7. Data Packet Path

This causes the non-processed Data packet to reach the consumer first. Furthermore, the PIT entry will be satisfied and deleted, so the processed Data packet (from the function process) will not have a PIT entry to refer to; therefore making it incapable of reaching the consumer. To overcome this problem, sequence numbers for in-records is applied. Each time an in-record for a PIT entry is created, it will be given a sequence number, which starts at 1 and increments by 1 every time the Interest packet with the corresponding Content Name is forwarded.

After a Data packet is received and a PIT entry is found, NFD will search for the in-record with the largest sequence number, and send it out to that face. This will ensure that the Data packet will go in the reverse path of the Interest packet. If the largest sequence number is anything larger than 1, it will not delete that PIT entry. In other words, the PIT entry will only be deleted after the last remaining in-record is satisfied. This will prevent non-processed Data packets from going to the consumer, and it will prevent the PIT entry from being removed until the processed Data packet is returned. Fig. 8 shows a simple example of how sequence numbers work when an Interest packet is received, and Fig. 9 shows a simple example of how sequence numbers work when a Data packet is received.

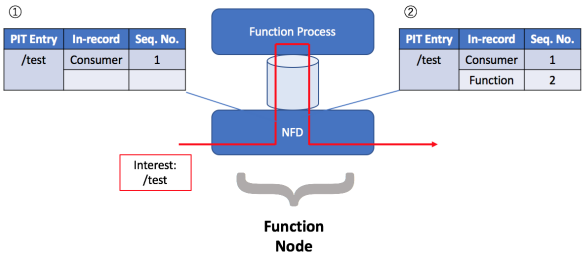

Fig. 8. Sequence Number Interest Packet Example

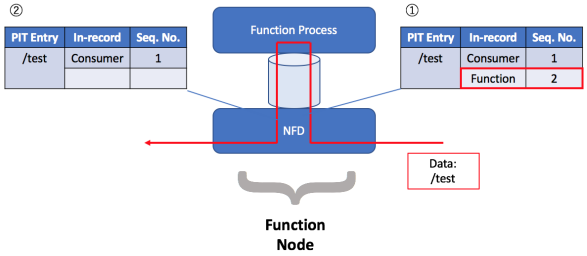

Fig. 9. Sequence Number Data Packet Example

V. Function Implementation

When a function process receives an Interest packet, it will forward the interest packet back to NFD. This step is required for creating a PIT in-record for the function, which will later be used by the Data packet. When the function process receives a Data packet, it will check the Final Block ID to see how many Data packets to expect. When all Data packets are received, it will reassemble the content. After the reassembly is done, it will execute the function using the content as the parameter. When the function execution is complete, the outcome will be segmented again, and sent towards the consumer. In short, the function will reassemble, execute, and segment.

Fig. 10. Function Implementation

VI. Consumer Producer API Extension

Consumer Producer API is a module to provide "a new network programming interface to NDN communication protocols" [7]. If NFD is the network layer, Consumer Producer API would be the application layer. The main benefits of the Consumer-Producer API are that it creates an easy interface for deploying consumers and producers, it handles data segmentation/reassembly, and it comes with multiple retrieval methods.

Most content will not fit into a single Data packet. Therefore, segmentation and reassemble is inevitable for making a effective network. We will extend the Consumer Producer API to support NDN-FC.

In the original architecture of Consumer Producer API, the consumer initially sends only 1 Interest packet, and will send the remaining Interest packets after receiving the Final Block ID. However, for NDN-FC, we will need to reassemble the content at the Function Node. This means that sending only 1 Interest packet will not work, since the function needs all Data packets to start execution. Therefore we will change the architecture in the following way. A simple example is shown in Fig. 11.

1) The Final Block ID will be requested prior to the communication. For example, the consumer could send out an Interest packet named */test/file/info* to obtain the Final Block ID of content *test/file*.
2) Using the Final Block ID, the consumer will send all Interest packets.
3) When the producer receives all the Interest packets, it will send out all of its Data packets.
4) The function will process the content, and return the result to the consumer.

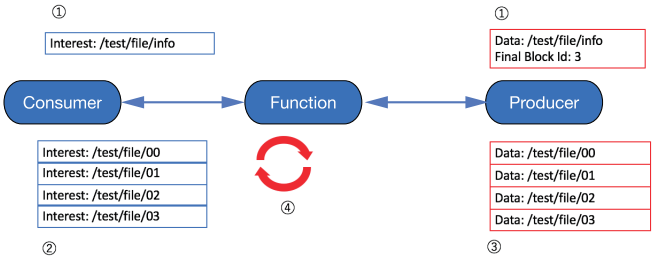

Fig. 11. NDN-FC Consumer Producer API Example

However, this is flawed in the sense that it does not account for content size change at functions. After a function is executed, it is likely that the content's file size will change. In other words, the segment count is susceptible to change. Therefore, the Interest packet count will have to accommodate for the new segment count. There are 2 possible ways that segment count can change, and they are listed below.

- Less than the original segment count
- Greater than the original segment count

1) Less than the Original Segment Count: When the new segment count is less than the original segment count, no changes are necessary. This is because all necessary PIT entries for sending Data packets are available. An example is shown in Fig. 12.

In this example, the original content has three segments. After the function execution, it becomes two segments. However, the PIT has the required entries, */test/00 and /test/01*, for sending to be successful. When the consumer receives */test/00*, it will extract the Final Block ID, which in this case is 1, so it knows that the new content is two segments instead of three. The remaining PIT entry for */test/02* will eventually expire and be deleted.

2) Greater than the Original Segment Count: To support the case where the new segment count is greater than the

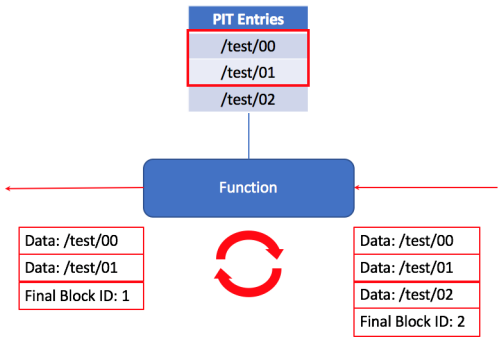

Fig. 12. Example of Content Size Shrink

original segment count, several functionalities must be added. This is because Data packets with Content Names that do not exist in the PIT must be created. To do so, additional Interest packets must be sent to that function. This is achievable through the following way. An example is shown in Fig. 13.

1) After the function is executed, the new Final Block ID is recorded in the newly created Data segments.
2) The same amount of segments as the original segment count are sent out.
3) Consumer and functions keep track of the number of Interest packets sent and know the original Final Block ID. When the consumer/function receives the Data packets with the updated Final Block ID, it compares it with the original one.
4) The consumer/function will send out the additional Interest packets.
5) The PIT will have the new Interest packet inserted.
6) The function returns the remaining Data packets.

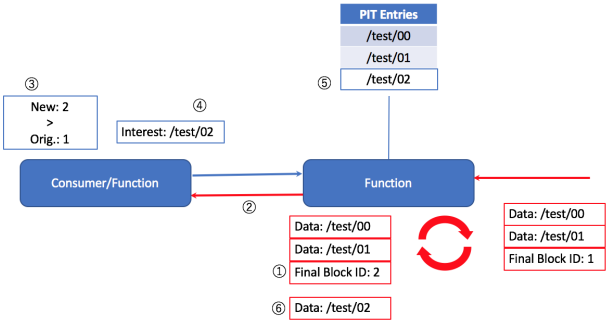

Fig. 13. Example of Content Size Expansion

In the example in Fig. 13, the original Final Block ID is 1. However, after the function execution, the Final Block ID becomes 2. Now the function will send Data segments */test/00 and /test/01* only, since they are the only ones available in the PIT at the moment. When the consumer/function receives these Data packets, it will extract the Final Block ID, and compare it with the original Final Block ID. In this case, the new Final Block ID is 2, and the original is 1. It needs to send an Interest packet for */test/02*. When it reaches the function, the PIT entry for it will be added. Now the Data segment */test/02* has a corresponding PIT entry, so it will be sent. The consumer/function will now have all segments for the processed content.

In this way, we are able to fully implement NDN-FC with segmentation/reassembly.

VII. Test and Results

A. Scenario

To confirm the correctness of the proposed segmentation handling, "greater than the original segment count" case was tested. The test configuration is shown in Fig. 14.

Fig. 14. Implementation Testing Scenario

Each of consumer, function, and producer, was run on its own virtual machine or docker container. Table I shows the versions of the software used in this experiment. The producer has a piece of content with the Content Name of */test/producer/test.png*. The function is named as */A*, and the consumer sends Interest packets out for the content located at the producer. The producer provides the Final Block ID of *test.png* by requesting */test/producer/info*.

The function processes this image file *test.png* into a larger sized file. To simulate this function execution, we prepared two files with the sizes of 15.4kB and 56.9kB. The function processes the 15.4kB file into a 56.9kB file.

TABLE I
EXPERIMENTAL ENVIRONMENT

	version
Viutualbox	6.0.12
Ubuntu	16.04
Docker	19.03.1

B. Results

Here we show the output of our test when using the docker version in Figs. 15 ∼ 18. It should be noted that due to the length of the output, only the important parts are shown.

The consumer first sends out an Interest packet to get the Final Block ID, which in this case is 8 (Fig. 15 ①). Next, the consumer sends out 9 Interest packets all with the Function Name field set to */A* (②).

Fig. 16 shows that the producer received these Interest packets, and sent out the Data packets. The buffer size shows that the image was 15.4kB, which is equal to 9 segments (Final Block ID: 8).

Fig. 17 shows that the function */A* received the Data packets from the producer. It reassembles the content (buffer size 15.4kB), and processes the image. The post-process image is 56.9kB, which has a Final Block ID of 33. Since the first 9

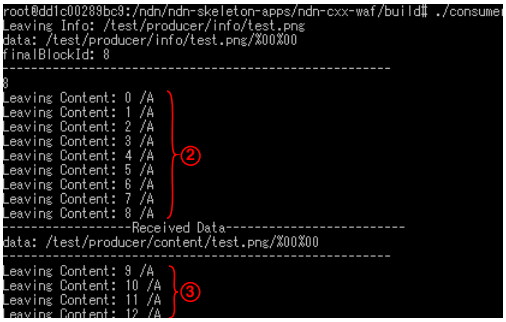

Fig. 15. Consumer Output (Interest Packet Transmission)

Fig. 16. Producer Output

Data packets were already received by the consumer as seen in Fig. 15, the consumer knows the new Final Block ID of 33. Now the consumer knows the new Final Block ID and sends out Interest packets with segment numbers 9 through 33 (Fig. 15 ③).

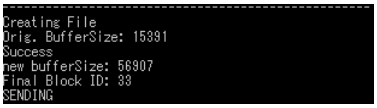

Fig. 17. Function Output

Finally, Fig. 18 shows that the consumer received all 34 segments that came from function */A*.

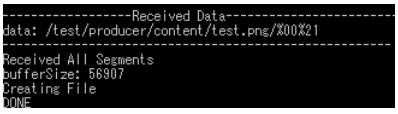

Fig. 18. Consumer Output (Data Packet Reception)

From these results, it can be seen that NDN-FC was successfully implemented with proper segmentation.

C. Comparison of virtual machine and Docker container

To compare the startup time of functions in NDN-FC in two environments: virtual machine and Docker container, we created virtual machine implementation in addition to Docker implementation. As the startup time of a function in Docker container, we measured the time to startup a Docker container of a function from its image using "Measure-Command" command in Windows PowerShell. The startup time of a virtual machine of a function is calculated from the startup log which can be displayed with "dmesg" command. Table II shows the results of five measurements with 3072MB of memory allocation. From Table II, we can be confirmed that the startup time of Docker container is shorter.

TABLE II
STARTUP TIME COMPARISON

number	docker container startup time(s)	VM startup time(s)
1	4.831765	62.47054
2	2.869693	47.32371
3	2.591142	50.25787
4	2.723558	46.92482
5	2.753842	45.98574

VIII. Conclusion and Future Work

In this paper, we have discussed the necessity of function chaining especially in IoT environments. We have also proposed an architecture for supporting function chaining in NDN for IoT environments. While many previous researches use simulations to show proof-of-concept, we have focused more on real-world usability by implementing and testing on real machines. We believe segmentation is an important aspect of real-world application and have focused on implementing segmentation into NDN-FC using existing NDN software.

Many functions may be deployed on the fly in the environment providing function chaining. The resources consumed by each function in such an environment should be minimum not to exhaust resource at a node. To support the minimum resource consumption with on-the-fly deployment of functions, container technology such as Docker is a viable alternative. The lightweight characteristics of the container technology can be exploited to support load balancing in function execution.

Our future work includes implementing push type NDN-FC, and testing with more complex scenarios. Our test results show that segmentation properly works with a basic network structure, but further work with scalability may be required. As a final product, we would like the network to intelligently place functions, and orchestrate functions according to network load and congestion.

References

[1] GE Digital, "What is edge computing?, https://www.ge.com/digital/blog/what-edge-computing.
[2] "What is network service chaining? Definition," https://www.sdxcentral.com/sdn/network-virtualization/definitions/what-is-network-service-chaining/.
[3] C. Tschudin and M. Sifalakis, "Named functions and cached computations," in 2014 IEEE 11th Consumer Communications and Networking Conference (CCNC), pp. 851-857, Jan. 2014.
[4] M. Sifalakis, B. Kohler, C. Scherb, andC. Tschudin, "An information centric network for computing the distribution of computations," in ICN 2014 - Proceedings of the 1st International Conference on Information-Centric Networking, pp. 137-146, Sept. 2014.
[5] L. Liu, Y. Peng, M. Bahrami, L. Xie, A. Ito, S. Mnatsakanyan, G. Qu, Z. Ye, H. Guo, "ICN-FC: An information-centric networking based framework for efficient functional chaining," in 2017 IEEE International Conference on Communications (ICC), pp. 1-7, May 2017.
[6] "ndn-cxx overview," https://named-data.net/doc/ndn-cxx/current/README.html
[7] I. Moiseenko, L. Zhang, "Consumer-producer API for named data networking," in NDN, Technical Report NDN-0017, 2014.