Real World Implementation of Function Chaining in Named Data Networking

by

Hiroki Yoshii
5117F098-7

Advisor: Hidenori Nakazato
Research on Distributed Computing

A Thesis
Submitted to the Department of Computer Science and Communications Engineering, the Graduate School of Fundamental Science and Engineering of Waseda University in Partial Fulfillment of the Requirements for the Degree of Master of Engineering

February 1, 2018

Contents

Chapter 1

Introduction

1.1 Background

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 self-driving.

In response to this, edge computing has been proposed. Edge computing is the idea that by placing computing infrastructures closer to the edge device, data can be processed quicker and more efficiently compared to sending it out to a cloud server [3]. However, the problem with this idea is that it is heavily reliant on the location and processing power of a single infrastructure.

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 [4]. 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). By combining SFC and NDN we can make a more efficient IoT network.

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

1.2 Structure of this Paper

This research will be structured as follows.

- In Chapter 1, the introduction to this research will be detailed.
- In Chapter 2 and 3, the fundamental concepts used in this will research will be discussed.
- In Chapter 4, related work will be explained.
- In Chapter 5, the existing software of NDN will be introduced. In Chapter 6, the architecture for Push and Pull type NDN will be discussed.
- In Chapter 7, the architecture for NDN Function Chaining will be proposed.
- In Chapter 8, the implementation of NDN Function Chaining will be described.
- In Chapter 9, the test results of the above implementation will be displayed.
- In Chapter 10, the conclusion and future work will be stated.

Chapter 2

ICN and NDN

2.1 The Problem with the current Internet Architecture

The current Internet architecture uses IP addresses in order to route packets, which makes it a location-based protocol. It was initially developed as a communication network with packets containing information on the destination end point. At the time of development, people only needed to communicate, so this architecture was sufficient. However, as digital media, social networking, and smart phone applications became more popular, the Internet has turned into a content distribution network. Using a communication network for content distribution is inefficient and unsuitable.

As a solution to this problem, Information-Centric Networking (ICN) was proposed. With ICN, content can be retrieved through data name rather than data location. There are several implementations of ICN, but Named Data Networking (NDN) is considered to be the most popular and promising one.

2.2 ICN

ICN is a proposed Internet structure that has been engineered to efficiently deliver content. ICN is a content-centric network that is capable of retrieving data purely through data names. Content-centric networking allows data to be transported without being tied down to location, application, storage, or transport method. This results in networks becoming more efficient, flexible, and scalable[8].

There are many implementations of ICN, such as CCN/NDN, DONA, PURSUIT, and NetInf. In this research, we will be using NDN.

2.3 NDN

NDN is based on a network, which was proposed by Van Jacobson in 2006 called Content-Centric Network (CCN). The current Internet architecture is considered to be an hourglass shape with the network layer (IP) in the center of it. The middle curve separates the top and bottom layers resembling that both can be developed independently from one another. NDN does not alter this hourglass shape, but rather replaces the center portion from IP to content. This is represented in Fig. 2.1.

NDN implements ICN by using 2 types of packets: Interest and Data. Additionally, routers have 3 main components: Forwarding Information Base (FIB), Pending Interest Table (PIT), and Content Store (CS).

2.3.1 Interest and Data Packets

Interest packets are used to request content by Name. Data packets are used to package the actual content. More specifically, Interest packets are composed of Name, Selectors, Nonce, Guiders, Link, and Selected Delegation fields[3]. Data packets are composed of Name, Meta Info, Content, and Signature fields[4]. These are shown in Fig. 2.2. The most important fields of the

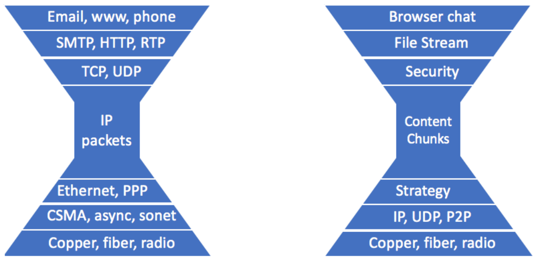


Figure 2.1: Hourglass Architecture

Interest packet are Name and Nonce. All other fields are optional. Nonce is short for Number used once. It is a random unique number given to each Interest packet for differentiating packets with the same Name.

Consumers will request content by specifying a Name using the Interest packet. The Interest packet will be routed through the network using the Name to a Producer. When the Interest packet arrives at the Producer, it will return the Data packet corresponding with the requested content Name.

Interest Packet
Content Name
Nonce
Interest Lifetime
Forwarding Hint

Data Packet
Name
MetaInfo
Content
Signature

Figure 2.2: Interest and Data Packet Structure

2.3.2 Forwarding Information Base (FIB)

The FIB is a routing table that maps Name prefix with next hop interfaces. When an Interest packet arrives at a router, the Interest Name is matched with the FIB records. If a match is found, the Interest packet is sent out to the corresponding next hop interfaces.

2.3.3 Pending Interest Table

The PIT is responsible for recording pending (unsatisfied) Interest packets and their incoming and outgoing interfaces. Data packets will use this information to return to the Consumer. Once a Data packet that matches a PIT entry is found, that entry is removed from the PIT.

2.3.4 Content Store (CS)

The CS allows routers to cache Data packets. This means popular content can be retrieved will very low latency, due to the Interest packet not needing to go to the Producer every time.

2.3.5 Architecture of NDN

Given the keywords from above, the following example Fig. 2.3 will demonstrate how the NDN architecture works.

1. The Consumer will send out an Interest packet. In Fig. 2.3 the Interest Name will be */prefix/data*.

2. When the Interest packet arrives at a NDN router, the router will check if a matching Name exists in the CS. If a match is found, the corresponding Data packet is returned. If a match is not found, the Interest will be forwarded according to the FIB. In Fig. 2.3 the FIB entry corresponding to */prefix/data* is */prefix*, so the Interest Packet will be forwarded to R2.

3. Additionally, the Interest packet and its incoming and outgoing interfaces will be recorded in the PIT. In Fig. 2.3, for Interest packet */prefix/data*, the incoming interface will be the Consumer, and the outgoing interface will be R2.

4. Steps 2 and 3 will be repeated at each NDN router.

5. When the Interest packet reaches the Producer, it will return the corresponding Data packet.

6. When the Data packet reaches a NDN router, a PIT look-up will be done. If a matching Interest Name is found, it will use the Incoming interface information to find its next destination. In Fig. 2.3 at R2, the Data packet matches the PIT entry with the Name */prefix/data* that corresponds to Incoming interface R1. Therefore, the Data packet will be sent out to R1.

7. This PIT entry is now considered satisfied, so it will be erased from the PIT.

8. Steps 6 and 7 are repeated at each NDN router until the Data packet reaches the Consumer.

In this way, the Consumer is able to retrieve its desired Data packet.

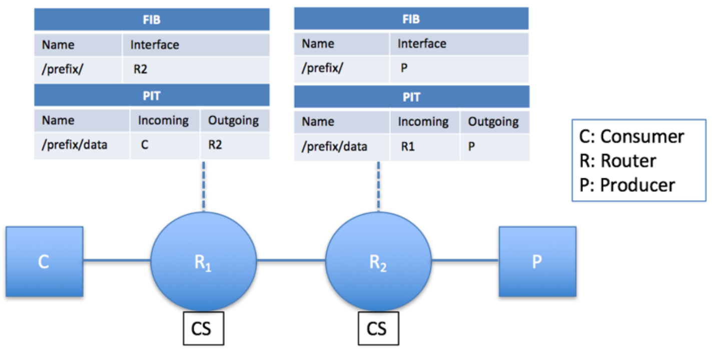


Figure 2.3: NDN Example

Chapter 3

Service Function Chaining

Service Function Chaining (SFC) is a combination of Software-Defined Networking (SDN) and Network Function Virtualization (NFV). SDN and NFV will be discussed below. Afterwards, SFC will be discussed.

3.1 SDN

The idea behind SDN is to eliminate infrastructure maintenance processes and guarantee easy management for cloud computing[8]. It achieves this by routing network traffic with software. SDN separates a network into a data and control plane. The control plane decides how each packet will be forwarded, which is usually decided by a centralized intelligent controller. The data plane is responsible for actually transporting packets according to the controller. Although virtualization technology has made computation and storage more flexible and available, it is hindered by the static nature of traditional networks[6]. SDN can potentially solve this problem due to its flexible nature.

3.2 NFV

NFV is the idea of decoupling network functions such as NAT, firewall, DNS, etc. from its proprietary hardware and running each as software on virtualized infrastructures. [5]. This gives network providers the benefits of agility, flexibility, and reduced costs [5].

3.3 SFC

SFC utilizes the capabilities of SDN and NFV to virtually chain network services together. More specifically, NFV functions are placed within the network. By utilizing SDN capabilities, packets can be programmatically routed to these NFV functions. An example is represented in Fig. 3.1.

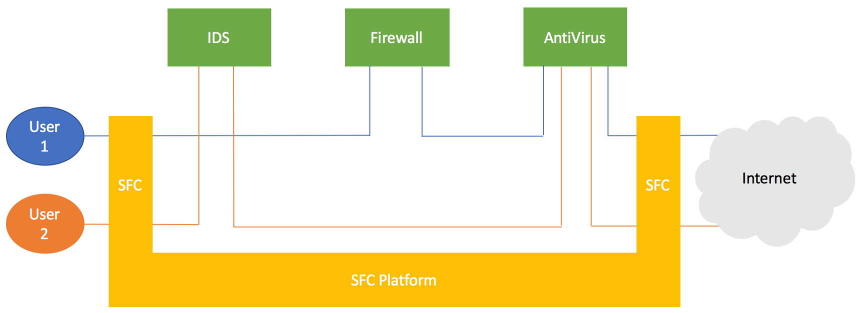


Figure 3.1: SFC Example

In this example there are 3 network functions: Intrusion Detection System (IDS), Firewall, and Anti-virus. User 1 and User 2 will use the SFC platform to connect to the Internet. User 1 wants to use the Firewall and Anti-virus functions, and User 2 wants to use the IDS and Anti-virus functions. The SFC platform will use its SDN capabilities to route User 1's packets to the Firewall and Anti-virus network functions. Similarly, the SFC platform will route User 2's packets to the IDS and Anti-virus network functions. In this way, a virtual chain of function services is created.

Chapter 4

Related Work

The idea of combining SFC with NDN has been proposed several times within the NDN research community. However, SFC in NDN is still in its early stages. The motivation of processing data within the network is similar across all proposals, but the architecture and execution differs.

4.1 Named Function Networking (NFN)

Sifalakis et al. [12] proposed a solution called Named Function Networking (NFN). In comparison to NDN, NFN uses Interests to request functions, parameters through the use of λ-expressions. In an NFN environment, NFN routers are deployed in addition to normal NDN router. NFN routers have computing capabilities, and are responsible for resolving NFN specific expressions and executing functions. A simple example of NFN architecture is shown in Fig. 4.1.

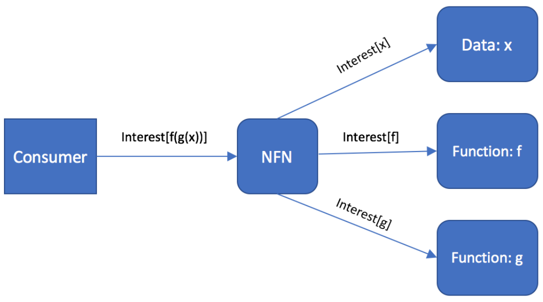


Figure 4.1: NFN Example

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 [10]. Data will be retrieved in the following way.

1. Interest */ f(/g(/x))* will be routed towards a 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[10]:

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. The Interest can be forwarded using */data* or */func* by using the expressions **3.** or **4.** respectively. If a result cannot be found on a certain path, a different path can be used 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.

4.2 ICN Function Chaining (ICN-FC)

Liu et al. [9] proposed a solution called ICN Function Chaining (ICN-FC). ICN-FC uses a simpler approach than NFN for chaining functions. It uses the $\leftarrow$ symbol to connect functions within the Interest name. A simple example is shown in Fig. 4.2.

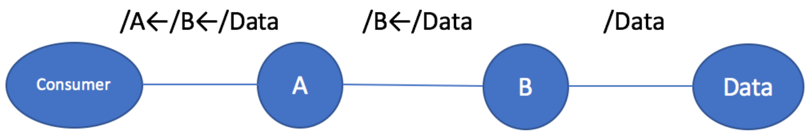


Figure 4.2: ICN-FC Example

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*$\leftarrow$*/B*$\leftarrow$*/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 is simplistic and effective, it can be improved on from a architectural stand point. It has a dynamic naming scheme, which can make tracing packets difficult for troubleshooting. Also, when multiple functions are chained, the Interest name can get increasingly long making it complex, which can compromise the simplicity of management. Additionally, it requires 2 PIT entries to be recorded for each Interest packet (before and after being processed by a function). This is not problematic from a practical stand point, but can congest the PIT, and make PIT search more complex. We have taken the basic idea of ICN-FC, and made architectural changes to expand upon it. This will be detailed in Chapter 7.

Chapter 5

NDN Tools

In this chapter, the tools necessary for using NDN will be detailed. These tools have been mainly developed by the NDN team at UCLA. These tools will be modified to support NDN-FC later in this research.

5.1 ndn-cxx

"ndn-cxx is a C++ library, implementing Named Data Networking (NDN) primitives that can be used to implement various NDN applications [1]." This includes code for Interest packet format, Data packet format, encoding/decoding of packets for network transferring, etc.

Of the many things ndn-cxx implements, one of the most important is a *face*. A face is a network interface where the NDN router will send and receive packets, and it represents a connection with a remote or local node. Each face has a unique ID.

5.2 NDN Forwarding Daemon (NFD)

"NFD is a network forwarder that implements and evolves together with the Named Data Networking (NDN) protocol [2]." It is the core component of NDN, and is responsible for routing NDN packets. It implements the FIB, PIT, and CS. It also uses a forwarding pipeline to route packets. Users are able to customize how the packets will be routed through the use of forwarding strategies. It is also responsible for communication with applications. It does use IP addresses as the underlying communication method, but the actual networking is done by NFD. The following will detail each component of NFD. The information in this chapter is provided in the NFD Developer Guide [7].

5.2.1 FIB Structure

The FIB is responsible for forwarding Interest packets to their corresponding data sources. In NFD, this is achieved by using the structure shown in Fig. 5.1. The FIB is consisted of FIB entries. Each entry is identified with an Interest name prefix, and each entry has information on NextHops. NextHop is comprised of face and cost. Cost is the weight of that connection, and lower values are usually desired for faster connections. The FIB will use longest prefix match to find an entry that matches the incoming Interest packet name most.

5.2.2 CS Structure

The CS is responsible for caching Data packets. It will cache Data packets according to Data name. If an Interest packet with the exact name is received, the CS will return the cached Data packet instead of forwarding the Interest packet to the data source. If the CS is full, it will use a caching algorithm selected or defined by the user to optimize cache hit rates.

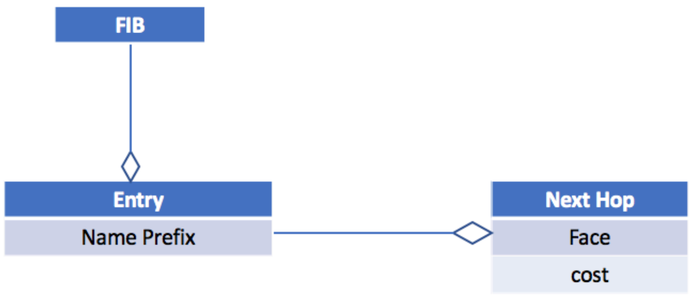


Figure 5.1: FIB Structure [7]

5.2.3 PIT Structure

The PIT is responsible for keeping track of pending Interest packets, and sending Data packets back to the consumer. In NFD, this is achieved by using the structure shown in Fig. 5.2. The PIT is consisted of PIT entries. Each entry is identified with an Interest. Each recorded Interest will have an in-record and an out-record. The in-record tracks which face the Interest packet came from, and the out-record tracks which face the Interest packet went out to. When a Data packet arrives, the PIT will use exact match on the Data name. If a matching PIT entry (Interest name) is found, it will use the in-record face to send out the Data packet. Using this method, the Data packet is able to go back to the consumer. The PIT also records Nonces, to check for Interest packet loops.

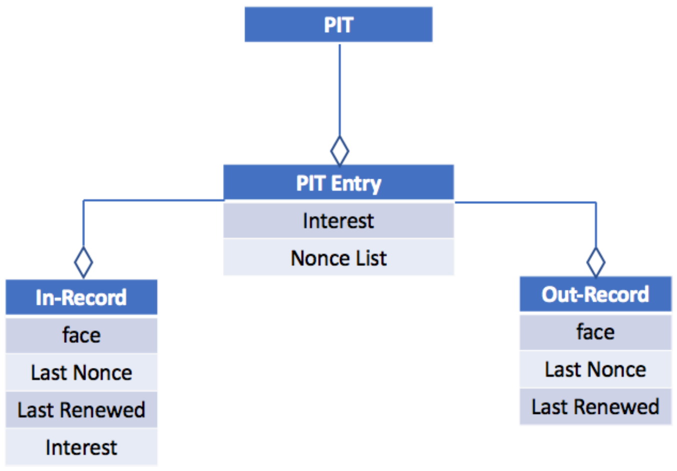


Figure 5.2: PIT Structure [7]

5.2.4 Interest Forwarding Pipeline

The Interest forwarding pipeline is responsible for forwarding Interest packets to the data source. NFD will process Interest packets as shown in Fig. 5.3.

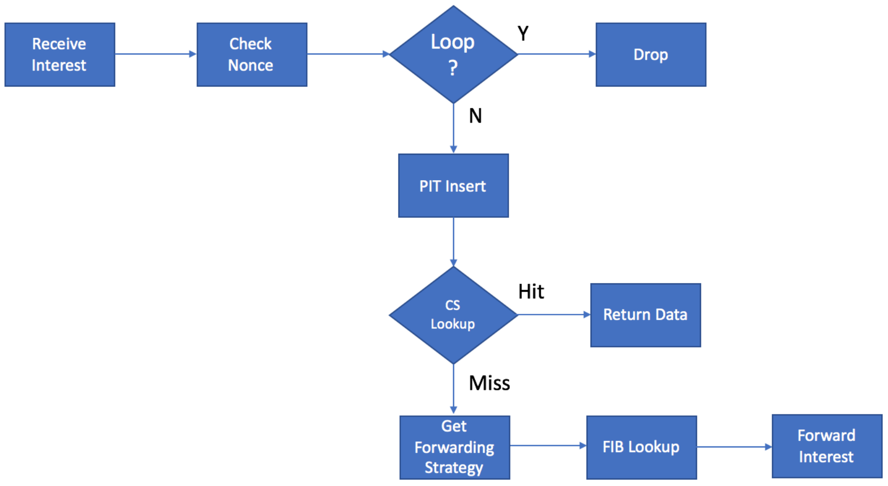


Figure 5.3: Interest Forwarding Pipeline

When an Interest packet is received, the nonce will be checked for any loops. If it is a loop, it will dropped. If it is not a loop, the Interest will be inserted into the PIT. Next, a CS look-up will be done. If there is a cache hit, the corresponding Data packet will be returned. if there is a cache miss, the a FIB look-up will be done. The forwarding strategy for the Interest packet will be retrieved, and it will be forwarded accordingly.

5.2.5 Data Processing Pipeline

The Data processing pipeline is responsible for sending Data packets back towards the consumer. NFD will process Data packets as shown in Fig. 5.4.

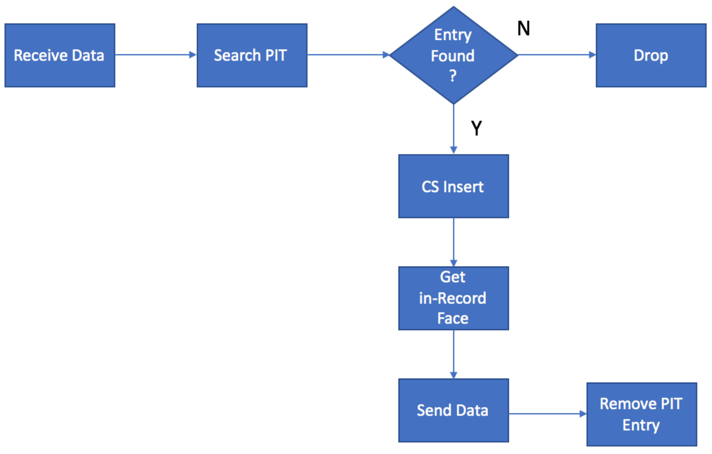


Figure 5.4: Data Processing Pipeline

When a Data packet is received, a PIT search will be done. If a matching entry is not found, it will be dropped. It can also be cached if the user specifies. If a matching entry is found, the corresponding in-record is retrieved, and the Data packet will be sent out to the face recorded in the in-record. Finally, the PIT entry will be removed.

5.2.6 Forwarding Strategy

Through the use of forwarding strategies, users are able to configure what actions are done during the forwarding process. These actions are configurable through the use of 4 triggers: After

Receive Interest, Before Satisfy Interest, Before Expire Interest, and After Receive Nack.

After Receive Interest is triggered after a CS look-up miss in the Interest Forwarding Pipeline from Fig. 5.3. Users will usually do a FIB look-up first to get next hops. Then users will decide how an Interest packet will be sent according to the next hops. There are also several pre-defined strategies such as best-route and multicast. Best-route will send Interest packet out to the next hop of the lowest cost. Multicast will send the Interest packet out to all next hops. By default, best-route is selected.

Before Satisfy Interest is triggered after a PIT entry is satisfied. This is usually used to get data measurements. By default, it does nothing.

Before Expire Interest is triggered when a PIT entry is expired. By default, it does nothing.

After Receive Nack is triggered after a Nack is received. By default, it does nothing.

5.2.7 Communication with Applications

For NDN to be useful, packets will eventually have to reach applications. NFD communicates with applications through the use of inter-process communication (IPC). NFD is able to inter-process communicate through the use of Unix domain sockets. By using these sockets, NFD is able to constantly listen for any local connection requests coming from an application. These sockets require a name, and the default name for NFD is set to *unix:///var/run/nfd.sock*. When nodes create a face, it will try to connect to *unix:///var/run/nfd.sock* unless stated otherwise. NFD will pick up the signal, and establish a connection. When the connection is established, NFD will create a face for it. By sending packets to this face, NFD is able to send packets to applications. Applications can also send packets from their face to NFD's face to send packets back to NFD. This system is shown in Fig. 5.5.

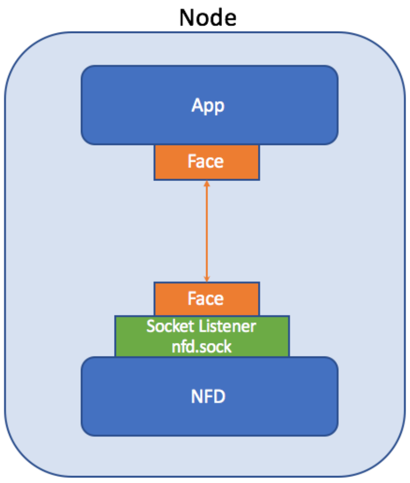


Figure 5.5: NFD Communication with Application

5.3 Consumer Producer API

Consumer Producer API "presents a new network programming interface to NDN communication protocols and architectural modules [11]." 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. The following will detail how the Consumer-Producer API works.

5.3.1 Naming Scheme

In Consumer Producer API, packets are named as */prefix/content_name/segment_number*. For example */test/pic.png/00* would mean segment number *0* of content name *pic.png* of the prefix *test.*

5.3.2 Retrieval Methods

There are 3 methods of retrieval: Simple Data Retrieval (SDR), Unreliable Data Retrieval (UDR), Reliable Data Retrieval (RDR). SDR simply sends one Interest packet for one Data packet. It does not guarantee delivery or retrieval. This retrieval method does not natively support segmentation and reassembly. UDR is more advanced than SDR in the way that it accounts for segmentation. However, it does not guarantee retrieval. RDR is similar to UDR, except it will keep sending Interest packets until all necessary Data packets are received.

5.3.3 Consumer API

The consumer API provides an interface to easily deploy consumers. The main components are shown in Fig. 5.6.

consumer(name prefix, retrieval method)

consume(name suffix)
setContextOption(handle, option name, value)
getContextOption(handle, option name)

Figure 5.6: Consumer Context [11]

To instantiate a consumer, a user must provide a Name prefix for the Interest packet it should send, and the retrieval method. Additionally, the user must set context options. Context options are used to configure Interest packet settings, and declare what happens after certain checkpoints are passed. The *consume* function is used to send the Interest packet with the chosen suffix attached. An example is shown in Fig. 5.7.

```
processPayload(Consumer& con, int8_t* content, size_t contentSize){...}

Consumer con(Name("/test"), RDR);
con.setContextOption(INTEREST_LIFETIME, 5000);
con.setContextOption(CONTENT_RETRIEVED,
    (ConsumerContentCallback)bind(
    &CallbackContainer::processPayload, &container, _1, _2, _3));
con.consume("pic.png");
```

Figure 5.7: Consumer API Example

In this example, the consumer's name is */test*, and it will use retrieval method *Reliable Data Retrieval (RDR)*. With the first context option, the Interest lifetime is set to 5000 milliseconds. With the second context option, the callback function named *processPayload* will execute when the checkpoint *content retrieved* is passed, which refers to when a content is received. Finally, *consume* will send out an Interest packet from this consumer with the name of */test/pic.png*. The *consume* function will automatically add segement numbers.

In this way, users are able to simply deploy consumers.

5.3.4 Producer API

The producer API provides an interface to easily deploy producers. It works similarly to the consumer API. The main components are shown in Fig. 5.8.

To instantiate a producer, a user must provide a Name prefix for the content it will provide. Additionally, the user must set context options. Context options are used to configure producer settings, and declare what happens after certain checkpoints are passed. The *produce* function is used to send the content with the chosen suffix attached. The content must be serialized before being put into *produce*. If the content exceeds the maximum size of a single packet, the producer will segment the content into the appropriate amount of Data packets. The *attach*() is used to register the producer to the network. An example is shown in Fig. 5.9.

producer(name prefix)
produce(name suffix, content, content size) **attach**() **setContextOption**(handle, option name, value) **getContextOption**(handle, option name)

Figure 5.8: Producer Context [11]

```
onInterest(Producer& pro, Interest interest){....}

Producer pro(Name("/test"));
pro.setContextOption(INTEREST_ENTER_CNTX,
     (ProducerInterestCallback)bind(
     &CallbackContainer::onInterest, &container, _1, _2));
pro.attach();
uint8_t* content = new uint8_t[100000];
pro.produce(Name("pic.png"), content, 100000);
```

Figure 5.9: Producer API Example

In this example, the producer's name is */test*. With this context option, the callback function *onInterest* will be executed when the checkpoint *interest enter context* is passed, which refers to when an Interest packet is received. The function *attach*() will register this producer to the network, making it reachable on the network. In this example, the content is an empty array of 8-bit unsigned integer with the size of 100000. The 8-bit unsigned integer is used, since it is equivalent to a byte. Finally, *produce* is used to send the content */test/pic.png*. It should be noted that the *produce* function will also need to know the serialized content and its size. The *produce* function will automatically add segment numbers.

In this way, users are able to simply deploy producers.

5.3.5 Consumer Producer API Communication Architecture

A key feature of the Consumer Producer API is that it abstracts the segmentation and reassembly of packets. Most content, such as videos and pictures, will not fit into one Data packet, which makes segmentation almost necessary in most cases. The following will explain how the Consumer Producer API is able to achieve segmentation in NDN.

Segmentation Challenges in NDN

In NDN, consumers send Interest packets out for the content they need. The problem with this architecture when segmentation is necessary is that the consumer has no idea how many segments the content will be. The consumer needs to send Interest packets out for each segment, but it does not know how many it needs to send. Although theoretically it is possible to continuously send out Interest packets, this is not efficient nor optimal. Consumer Producer API overcomes this challenge through the use of Final Block IDs.

Final Block ID

The Final Block ID is a value kept within the Meta Info of a Data packet. It records the last segment number of a content. Its default value is set to 0 (segment numbers start at 0), since every content is at least one packet. When the consumer receives a Data packet, it is able to know how many Interest packets it needs to send. The following section will explain how the Final Block ID is specifically used.

Communication

1. The consumer will send out a single Interest packet.

2. When the producer receives the Interest packet, it will segment the requested content, and calculate the last segment number from the size of the file.
3. The producer will record the last segment number in the first Data packet.
4. It will send out the first Data packet to the consumer.
5. When the consumer receives this Data packet, it will extract the Final Block ID. Now the consumer knows exactly how many Interest packets it needs to send out.
6. The received Data packet is placed in a buffer, and the remaining necessary Interest packets are sent out.
7. When the producer receives the Interest packets, it will return the remaining Data packets.
8. When the consumer receives the Data packets, it will place the Data packets in the buffer, and reassemble the content.

An example of this process is shown in Fig. 5.10.

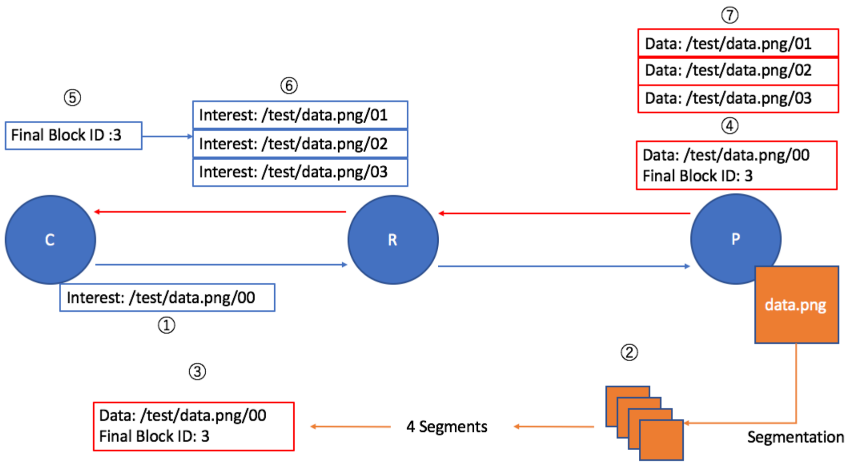


Figure 5.10: Consumer Producer API Segmentation

In this example, the consumer sends out an Interest with the name of */test/data.png/00*. When the producer receives it, it segments *data.png*. It calculates the segment number to be 4, so the Final Block ID will be 3. The Data packet */test/data.png/00* is sent with its Final Block ID being 3. When the consumer receives it, it extracts the Final Block ID. Now it knows that the final segment number is 3, so it sends out the remaining Interest packets with the segment number *01*, *02*, and *03* appended. When the producer receives these Interest packets, it returns the corresponding Data packets. Finally, the consumer is able to receive all Data packets it requested for.

Chapter 6

Push and Pull Type Communication in NDN

6.1 Push/Pull Type Communication in IoT Environments

For any type of communication, it is usually possible to categorize it into 2 types: push and pull. Pull is where the receiver requests for information from a sender beforehand. Push is where information is sent to a receiver by the sender regardless of requests. For NDN-FC, our main focus has been on IoT environments, and push/pull type communication can be applied here also. For example, in an IoT environment, there are many times when real-time data is necessary such as an alert to trigger certain actions. This would require a push type communication. There will be other times where users would like to decide when to get data. This would require a pull type communication. This is summarized in Fig. 6.1.

Push	Pull
Information is sent regardless of request	Request is sent beforehand by receiver
Real-Time Data Alerts Triggered By Events	Control Data Flow Users Decide When to get Data
eg. Security Camera $\rightarrow$ Alert anomaly	eg. Get temperature of a specific room

Figure 6.1: Push Pull Examples in IoT

For the reasons stated above, in order to have a fully effective IoT network, it is necessary to have an architecture that supports both push and pull types.

6.2 Push Type Communication in NDN

As seen in Section 2.3, NDN is natively a pull type architecture, and does not support push type communication. However, in order to make an effective IoT network, a push type architecture for NDN is necessary. The following will show how we decided to deploy push type communication into NDN.

6.3 Basic Architecture of Push Type NDN

Using Fig. 6.2 as an example, the architecture of Push Type NDN will be explained.

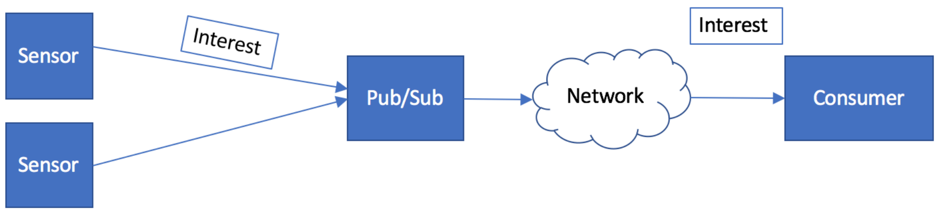


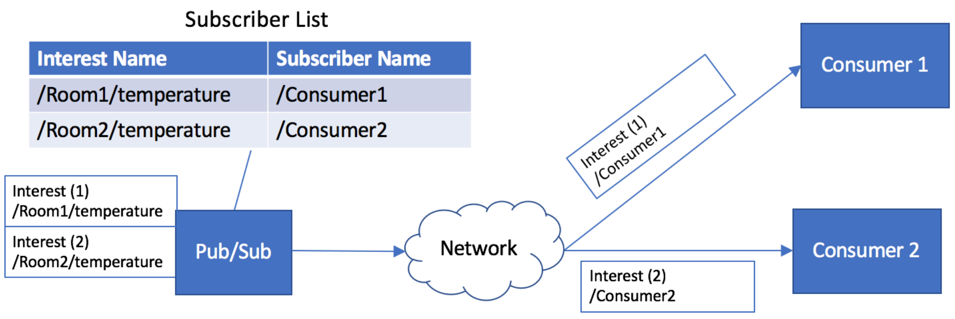
Figure 6.2: Push NDN Example

Firstly, content will be put inside the Interest packet, and the Data packet will be used as an ACK packet. Therefore, a Payload field will be added to the Interest packet. It will also have a Subscriber Name field, which will be explained later. The Interest packet format is shown in Fig. 6.3.

Interest Packet
Content Name
Subscriber Name
Nonce
Interest Lifetime
Forwarding Hint
Payload

Figure 6.3: Push Type NDN Interest Packet Format

Producers will send out these Interest to a Publisher/Subscriber node. There will always be a Publisher/Subscriber node within a network. The reason for this is that producers have no way of knowing who needs the content. There could be multiple routers connected to the producer, and it will not know which router to send to. Especially in an IoT environment, producers are usually restricted devices such as sensors, so there is no way of computing its next destination. However, with a Publisher/Subscriber node, the sensor only needs to know where that node is. The Publisher/Subscriber node will have a subscriber list, which will be used to decide where to forward the Interest packet. The subscriber list will be a mapping of Interest name and subscriber. In the Subscriber Name field of the Interest packet, the chosen subscriber will be recorded. Using this Subscriber Name field, the Interest packet will be forwarded. An example of the subscriber list is shown in Fig. 6.4.


Subscriber List

Interest Name	Subscriber Name
/Room1/temperature	/Consumer1
/Room2/temperature	/Consumer2

Figure 6.4: Subscriber List Example

Chapter 7

NDN Function Chaining (NDN-FC) Architecture

In this chapter, the architecture of NDN-FC for pull and push type communication will be explained.

7.1 Interest Packet Format

In NDN-FC, we have added a Function Name field to the Interest packet format from Fig. 6.3. In this field, the chain of functions will be listed. This will effectively replace the Subscriber Name field, since it achieves the same job. The result is shown in Fig. 7.1. The Data packet will be kept the same.

Interest Packet
Content Name
Function Name
Nonce
Interest Lifetime
Forwarding Hint
Payload

Figure 7.1: NDN-FC Interest Packet Format

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

7.2 Pull Type NDN-FC Basic Architecture

For basic pull type 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. 7.2.

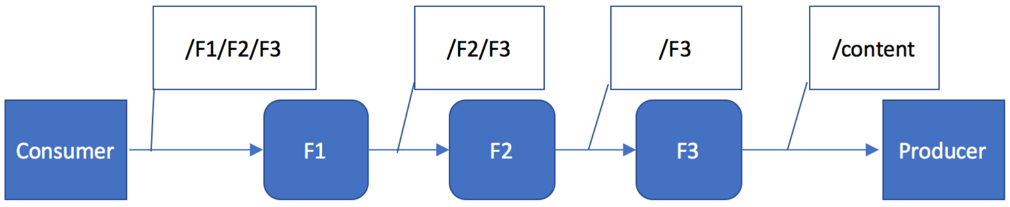


Figure 7.2: NDN-FC Example

In this example, the Consumer will send out an Interest packet with the Content Name field being */content* and the Function Name field being */F1/F2/F3*. When it passes through function node *F1*, */F1* will be removed from the Function Name field. Likewise, the Function Field will be */F3* after passing through function node *F2*. After it passes through function node *F3*, it will be forwarded according to Content Name */content*. Due to the NDN protocol, the Data packet will move hop-by-hop in the reverse order *F3* → *F2* → *F1*.

7.3 Push Type NDN-FC Basic Architecture

In push type NDN-FC, there will be a Publisher/Subscriber node for reasons mentioned in Section 6.2. Especially in function chaining, the producer has no way of knowing which subscriber needs which functions, so the Publisher/Subscriber node is crucial. The Publisher/Subscriber node will be considered a function, and producers will assign Publisher/Subscriber node in the Function Name field. Publisher/Subscriber node will use the subscriber list to determine necessary functions and subscribers. The subscriber list will be a mapping of Interest Name, Functions, and Subscribers. An example is shown in Fig. 7.3 and Fig. 7.4.

Interest Name	Subscriber Name	Functions
/content	Consumer	/F1/F2
/Room/temperature	Consumer2	/F1/F2/F3

Figure 7.3: NDN-FC Subscriber List

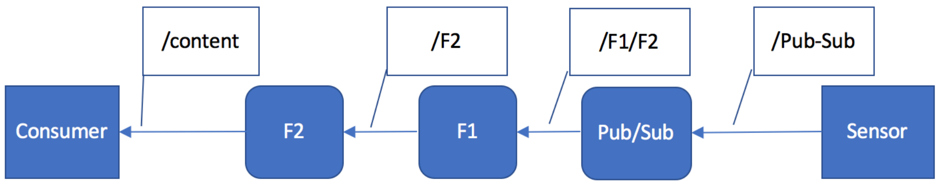


Figure 7.4: NDN-FC Publisher/Subscriber Example

In this example, the sensor sends out an Interest packet with Content Name */content* and Function Name */Pub-Sub*. When the Publisher/Subscriber node receives the Interest packet, it will search its subscriber list. In this case, the first entry will match, so the Function Name field will be changed to */F1/F2*. Then, the Interest packet will be sent out and be forwarded accordingly.

Chapter 8

Implementing NDN Function Chaining (NDN-FC)

In this chapter, the steps for implementing NDN-FC will be explained. Although we have proposed an architecture for push type communication, we have not been able to implement it as of yet. However, we believe most key features implemented in this chapter can be applied to the push type architecture also.

In our implementation, consumers, producers, and functions will be considered as applications, and NFD will be used as the router. A simple network implementation image is shown in Fig. 8.1.

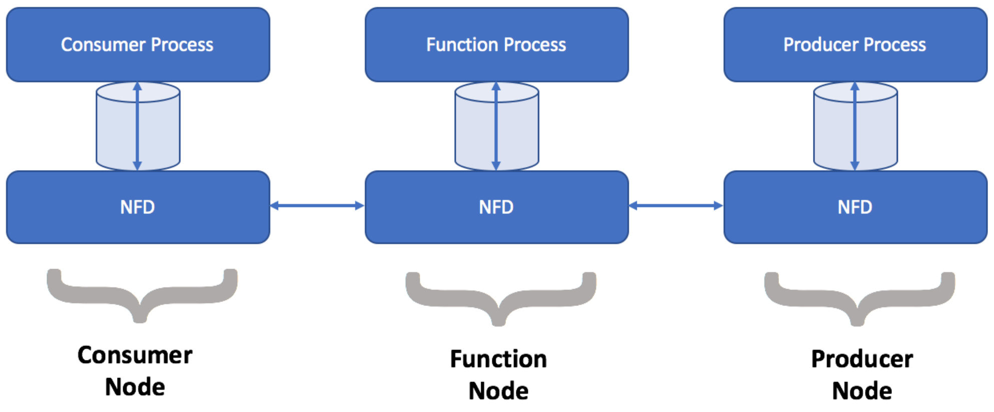


Figure 8.1: NDN-FC Implementation Image

8.1 Implementing NDN-FC

There are 4 major steps for implementing NDN-FC, which are listed below.

- Extending ndn-cxx
- Extending NFD
- Implementing Functions
- Extending Consumer Producer API

8.1.1 ndn-cxx Extension

In order to support NDN-FC, the Interest packet format must be modified. Pull type only requires the Function Name field to be added. Fig. 8.2 represents how the Interest packet format will look like.

Interest Packet
Content Name
Function Name
Nonce
Interest Lifetime
Forwarding Hint

Figure 8.2: Pull Type NDN-FC Interest Packet Format

The basic structure of the Content Name field will be used for the Function Name field. The *getFunction*, *setFunction*, and *hasFunction* functions have been added to the Interest class. The *getFunction* returns the Function Name field. The *setFunction* sets the Function Name field. The *hasFunction* checks to see if there are any functions in the Function Name field. The *removeHeadFunction* function has also been added to the Interest class in order to remove the leading function in the Function Name field. Fig. 8.3 represents the added functionalities.

```
Interest {
    ...
    Function m_function;    //added Function Field
    ...
    Function getFunction();     //gets Function Field
    void setFunction(Function function); //sets Function Field
    bool hasFunction();     //checks for Function Field
    void removeHeadFunction();  //removes the head function
    ...
}
```

Figure 8.3: Interest Class

8.1.2 NFD Extension

In order to support NDN-FC, 3 main things must be added: the function forwarding strategy, nonce refresh, and PIT in-record sequence numbers.

Function Forwarding Strategy

Using the strategy API from Section 5.2.6, we will create a new forwarding strategy for function chaining. The basic flow of the strategy is shown in Fig. 8.4.

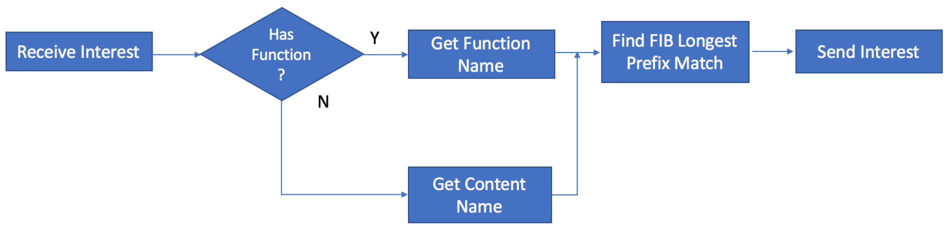


Figure 8.4: NDN-FC Forwarding Strategy

In this strategy, when *afterReceiveInterest* is triggered, the Function Name field will be checked using *hasFunction* from Section 8.1.1. If it has a Function Name, it will be retrieved. If it does not, the Content Name will be retrieve. Since the Content Name and Function Name fields are similar in structure, the FIB does not need to differentiate between the two to do a FIB look-up. When a matching entry is found, the Interest will be sent out to the corresponding face. All other triggers will be kept at their default values.

Nonce Refresh and PIT In-record Sequence Numbers

The difference between functions and consumers/producers is that it sends and receives both Interests and Data packets. In 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 PIT 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 a Interest packet is shown in Fig. 8.5.

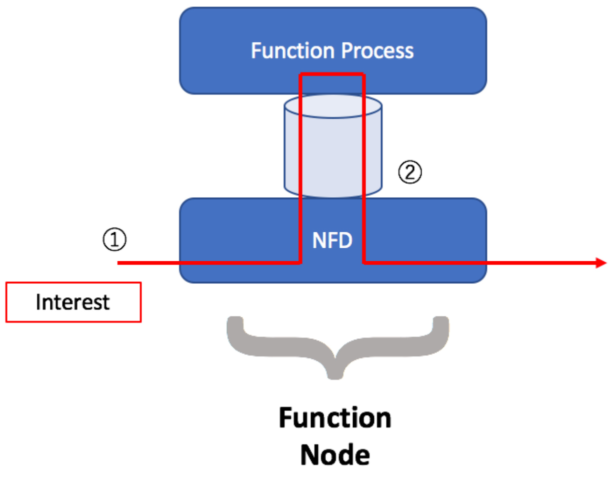


Figure 8.5: Interest Packet Path

This presents the problem of this Interest packet being detected as a loop. To overcome this problem, a nonce refresh will be executed after the loop detection. The nonce refresh will give the Interest packet a new nonce, which will make NFD think that it is a different Interest packet. In this way, we can effectively bypass the loop detection without completely removing it.

However, there is still another problem present. The problem is that the PIT records two in-records(① and ② of Fig. 8.5) linked to a single PIT entry. This is problematic because when the Data packet corresponding to the Interest packet is returned, it will be sent out to both faces at once. This is shown in Fig. 8.6.

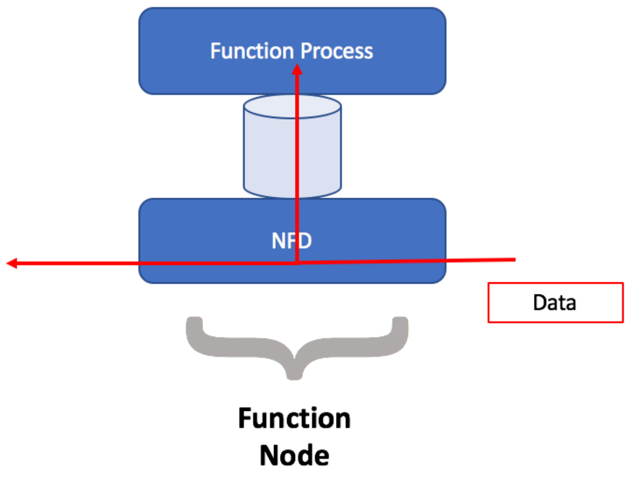


Figure 8.6: Data Packet Path

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 will be 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. This is shown in Fig. 8.7.

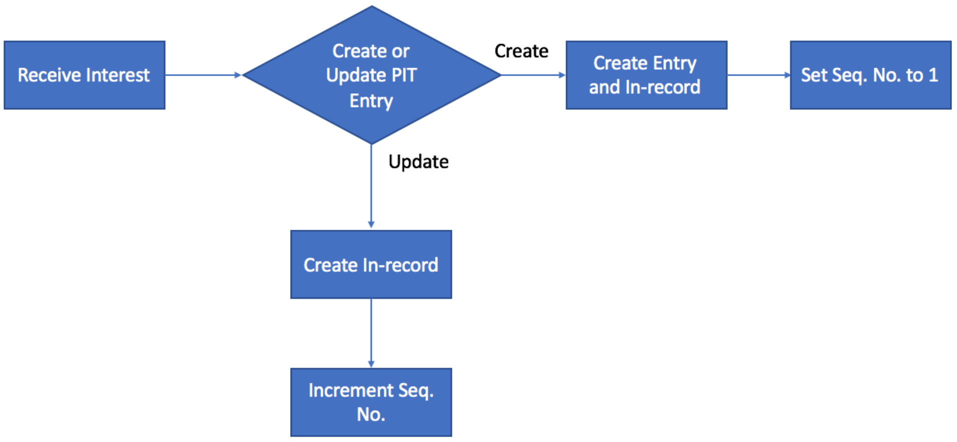


Figure 8.7: Sequence Number Interest Packet Process

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. This is shown in Fig. 8.8.

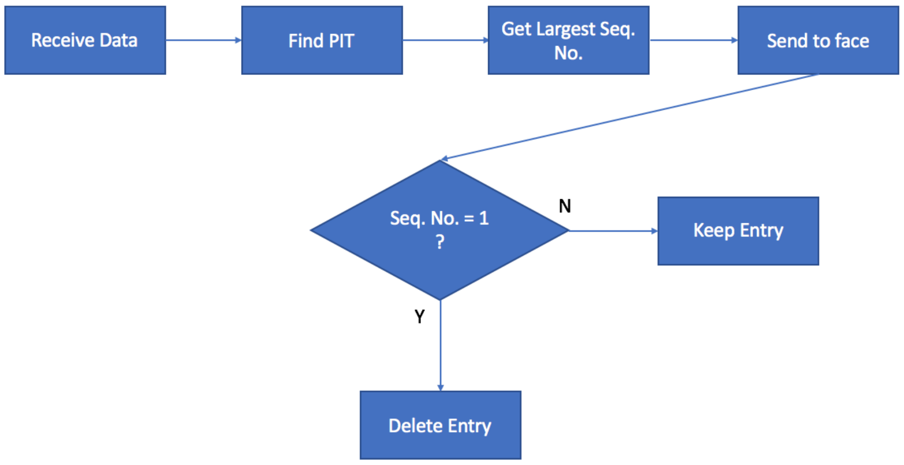
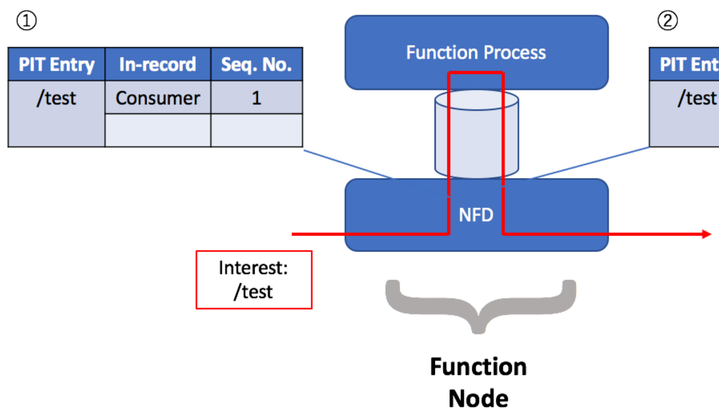


Figure 8.8: Sequence Number Data Packet Process

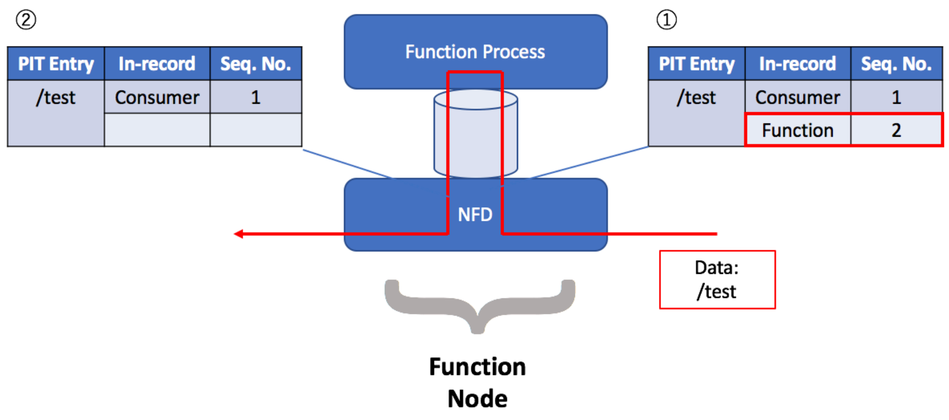
Fig. 8.9 shows a simple example of how sequence numbers work when an Interest packet is received, and Fig. 8.10 shows a simple example of how sequence numbers work when a Data packet is received.



PIT Entry	In-record	Seq. No.
/test	Consumer	1

PIT Entry	In-record	Seq. No.
/test	Consumer	1
	Function	2

Figure 8.9: Sequence Number Interest Packet Example



PIT Entry	In-record	Seq. No.
/test	Consumer	1

PIT Entry	In-record	Seq. No.
/test	Consumer	1
	Function	2

Figure 8.10: Sequence Number Data Packet Example

8.2 Function Implementation

The idea behind implementing a function is very simple. When an Interest packet is received, it will forward it back to NFD. Although this may seem pointless, this step is required for creating a PIT in-record for the function, which will later be used by the Data packet. When a Data packet is received, it will check the Final Block ID to check 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 is done executing, the outcome will be segmented again, and sent towards the consumer. To put simply, the function will reassemble, execute, and segment.

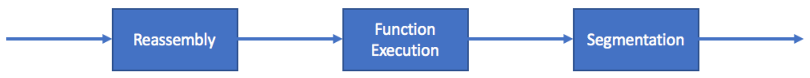


Figure 8.11: Function Implementation

8.3 Consumer Producer API Extension

As stated in Section 5.3.5, 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 only sends 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. 8.12.

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.

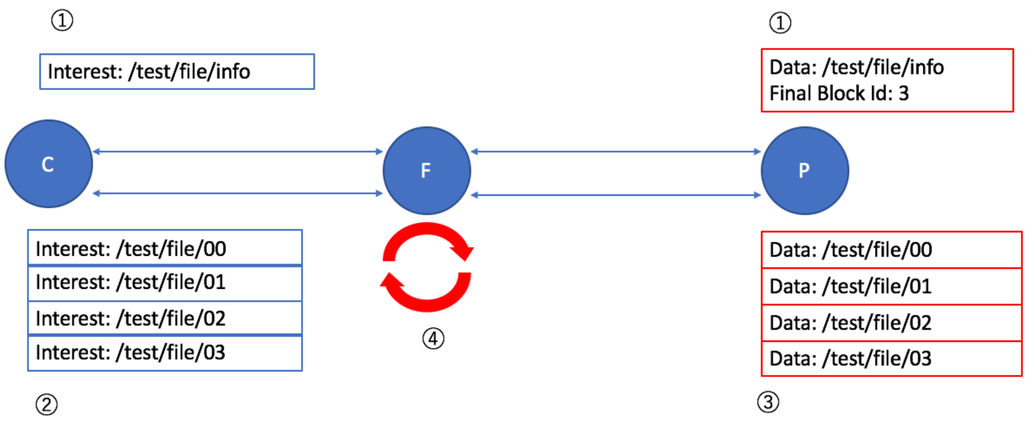


Figure 8.12: NDN-FC Consumer Producer API Example

However, this is flawed in the sense that it does not account for content size change. 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

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. 8.13.

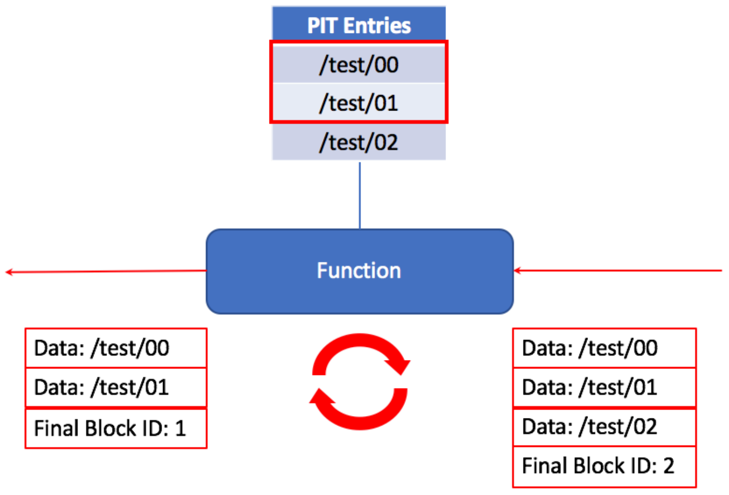


Figure 8.13: Less Than Original Content Example

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. The remaining PIT entry will eventually expire and be deleted.

Greater than the Original Segment Count

When the new segment count is greater than the original segment count, several functionalities must be added. This is because Data packets with Content Names that do not exist in the PIT are created. To overcome this problem, additional Interest packets must be sent to that function. This is achievable through the following way. An example is shown in Fig. 8.14.

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. Therefore, they are able to calculate the original Final Block ID. When the Data packets sent in the previous step arrive, the consumer/function will extract the Final Block ID, and compare it with their original segment count.
4. The consumer/function will send out the necessary Interest packets.
5. The PIT will have the new Interest packet inserted.
6. The function returns the remain Data packets.

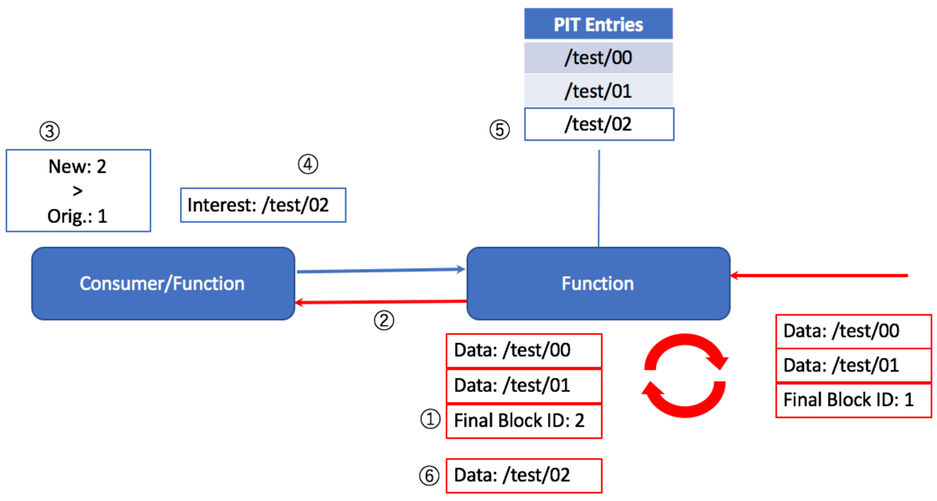


Figure 8.14: Greater Than Original Content Example

In this example, 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 this, it will extract the Final Block ID, and it will compare it with the original Final Block ID. In this case, the new Final Block ID is 2, and the original is 1. Therefore 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 pull type NDN-FC.

Chapter 9

Implementation Testing

In this chapter, the testing methods of NDN-FC will be detailed.

9.1 Scenario

In order to test if segmentation was successful, 3 segmentation patterns were used. They are equal to the original segment count, less than the original segment count, and greater than the original segment count. This will be tested in the scenario shown in Fig. 9.1.

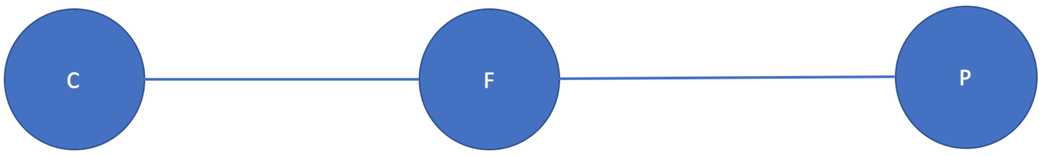


Figure 9.1: Implementation Testing Scenario

The producer will have a piece of content with the Content Name of */test/producer/test.png*. The function will be named */A*, and the consumer will send Interest packets out for the content located at the producer. The producer will also have the Final Block ID of *test.png* available at */test/producer/info*.

We will assume that the function will process this image file *test.png* into an equally sized file, a smaller sized file, and a larger sized file. To simulate this function execution, we have placed files on the function node that is equal to, less than, and greater than the original file. In this experiment, we have prepared two files with the sizes of 15.4kB and 56.9kB. We will use these files to create the 3 segmentation possibilities listed above. The test cases are shown in Fig. 9.2.

Equal to	Original: 15.4kB → Post-process: 15.4kB
Less than	Original: 56.9kB → Post-process: 15.4kB
Greater than	Original: 15.4kB → Post-process: 56.9kB

Figure 9.2: Test Cases

9.2 Setup

In this setup, we have created 3 virtual machines in VirtualBox where each machine resembles a consumer, function, and producer. NFD will be run on each machine, and applications will be run with Consumer Producer API. The result will look like Fig. 8.1.

To setup the network, it is necessary to link the nodes(machines) together. This is done through the use of the *nfdc* command provided in NFD. Fig. 9.3 shows the commands used to link these nodes.

```
nfdc face create remote udp://IP-address
nfdc route add prefix /routing-prefix nexthop face-id
```

Figure 9.3: nfdc Command Usage

The first command will create an outgoing face to the node specified with the IP address. It will output a face ID that was given to it by NFD. This will create a connection with the node. The second command will create a FIB entry for the provided prefix and face ID.

For our test cases, the consumer will be setup as follows. Let's assume the function's IP address is 10.42.0.187.

```
nfdc face create remote udp://10.42.0.187
```

Let's assume NFD gave this face an ID of 280.

```
nfdc route add prefix /test/producer nexthop 280
nfdc route add prefix /A nexthop 280
```

The first route will be used by Interest packet */test/producer/info*. The second route will be used by Interest packets for */test/producer/test.png*.

Similarly, the function will be setup as follows. Let's assume the producer's IP address is 10.42.0.188.

```
nfdc face create remote udp://10.42.0.188
```

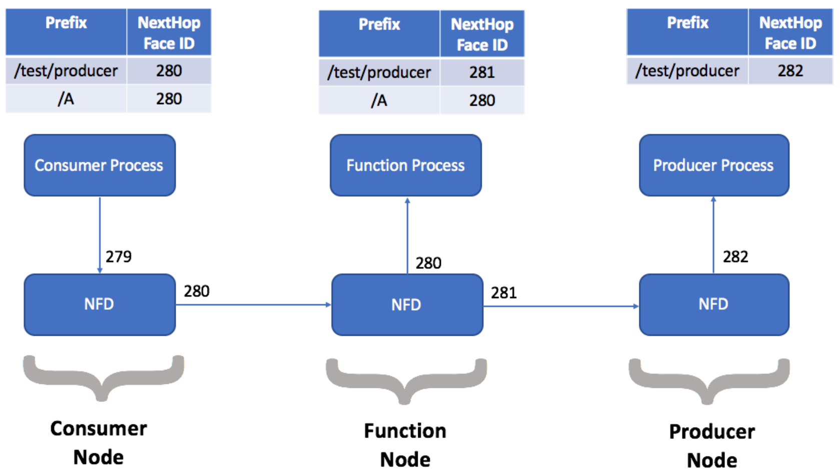
Let's assume NFD gave this face an ID of 281. It should be noted that this is a completely different instance of NFD from the one running on the consumer, so it is possible for NFD to assign 280 to this face also.

```
nfdc route add prefix /test/producer nexthop 281
```

This will allow Interest packets */test/producer/info* and */test/producer/test.png* to reach the producer.

When function and producer processes(applications) are executed, they will create a face, which will be caught by NFD via the socket explained in Section 5.2.7. Therefore, routing for those will be automatically added to the FIB.

When everything is completed, the network and FIBs will look like Fig. 9.4.



Prefix	NextHop Face ID
/test/producer	280
/A	280

Prefix	NextHop Face ID
/test/producer	281
/A	280

Prefix	NextHop Face ID
/test/producer	282

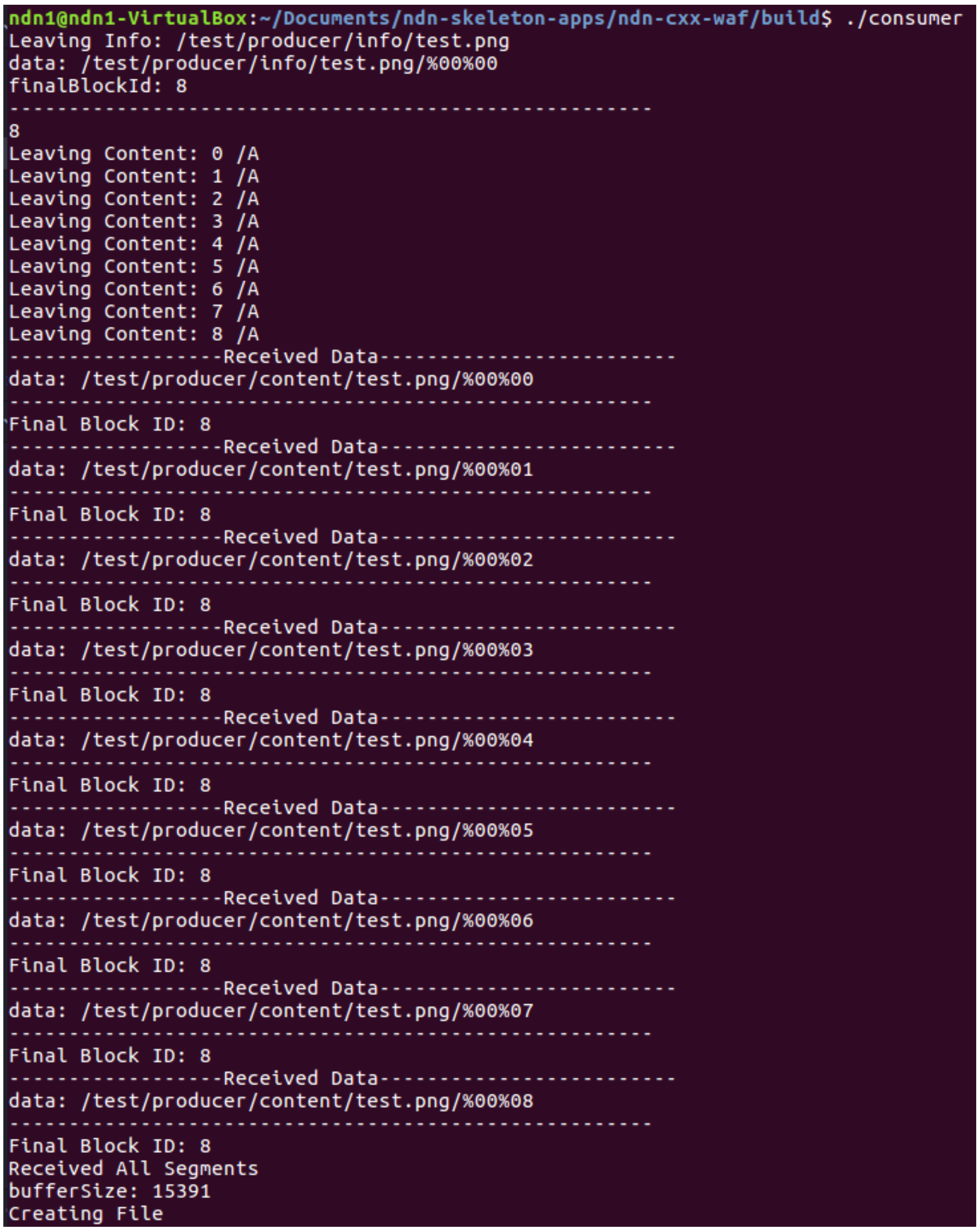
Figure 9.4: Completed Networking

9.3 Results

The following will show the results of the above scenario. Due to the fact that the output of the terminal is very long, only the important parts of the output will be shown.

9.3.1 Equal to the Original Segment Count

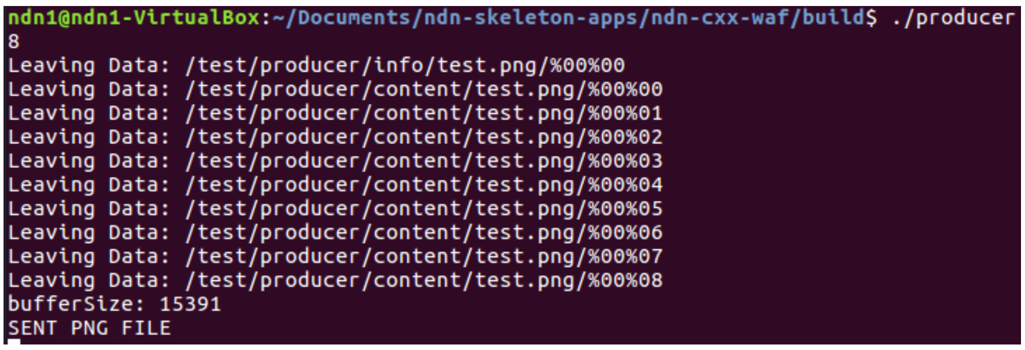
Fig. 9.5 shows that the consumer first sends out an Interest packet to get the Final Block ID, which in this case is 8. Next, the consumer sends out 9 Interest packets all with the Function Name field set to */A*.



```
ndn1@ndn1-VirtualBox:~/Documents/ndn-skeleton-apps/ndn-cxx-waf/build$ ./consumer
Leaving Info: /test/producer/info/test.png
data: /test/producer/info/test.png/%00%00
finalBlockId: 8
-----------------------------------------------------
8
Leaving Content: 0 /A
Leaving Content: 1 /A
Leaving Content: 2 /A
Leaving Content: 3 /A
Leaving Content: 4 /A
Leaving Content: 5 /A
Leaving Content: 6 /A
Leaving Content: 7 /A
Leaving Content: 8 /A
-----------------Received Data------------------------
data: /test/producer/content/test.png/%00%00
-----------------------------------------------------
Final Block ID: 8
-----------------Received Data------------------------
data: /test/producer/content/test.png/%00%01
-----------------------------------------------------
Final Block ID: 8
-----------------Received Data------------------------
data: /test/producer/content/test.png/%00%02
-----------------------------------------------------
Final Block ID: 8
-----------------Received Data------------------------
data: /test/producer/content/test.png/%00%03
-----------------------------------------------------
Final Block ID: 8
-----------------Received Data------------------------
data: /test/producer/content/test.png/%00%04
-----------------------------------------------------
Final Block ID: 8
-----------------Received Data------------------------
data: /test/producer/content/test.png/%00%05
-----------------------------------------------------
Final Block ID: 8
-----------------Received Data------------------------
data: /test/producer/content/test.png/%00%06
-----------------------------------------------------
Final Block ID: 8
-----------------Received Data------------------------
data: /test/producer/content/test.png/%00%07
-----------------------------------------------------
Final Block ID: 8
-----------------Received Data------------------------
data: /test/producer/content/test.png/%00%08
-----------------------------------------------------
Final Block ID: 8
Received All Segments
bufferSize: 15391
Creating File
```

Figure 9.5: Equal to Consumer

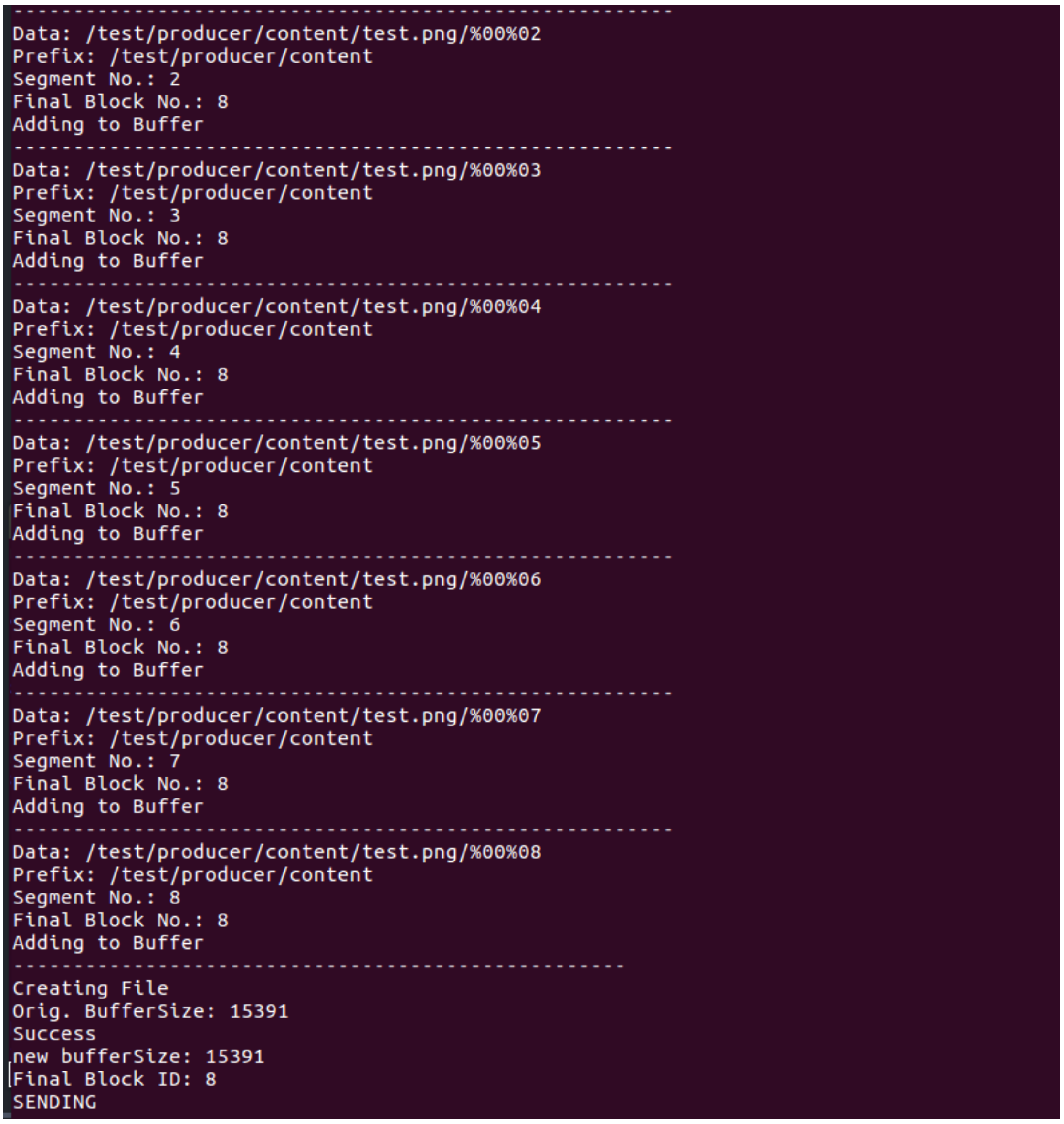
Fig. 9.6 shows that the producer received these Interest packets, and sent out the Data packets. The buffer size shows that the image was 15.4kB, which is equal to 9 segments(Final Block ID: 8).



```
ndn1@ndn1-VirtualBox:~/Documents/ndn-skeleton-apps/ndn-cxx-waf/build$ ./producer
8
Leaving Data: /test/producer/info/test.png/%00%00
Leaving Data: /test/producer/content/test.png/%00%00
Leaving Data: /test/producer/content/test.png/%00%01
Leaving Data: /test/producer/content/test.png/%00%02
Leaving Data: /test/producer/content/test.png/%00%03
Leaving Data: /test/producer/content/test.png/%00%04
Leaving Data: /test/producer/content/test.png/%00%05
Leaving Data: /test/producer/content/test.png/%00%06
Leaving Data: /test/producer/content/test.png/%00%07
Leaving Data: /test/producer/content/test.png/%00%08
bufferSize: 15391
SENT PNG FILE
```

Figure 9.6: Equal to Producer

Fig. 9.7 shows that the function */A* received the Data packets from the producer. It reassembles the content, and processes the image. The post-process image is also 15.4kB. Finally, if we refer back to Fig. 9.5, it can be seen that the consumer received the content.

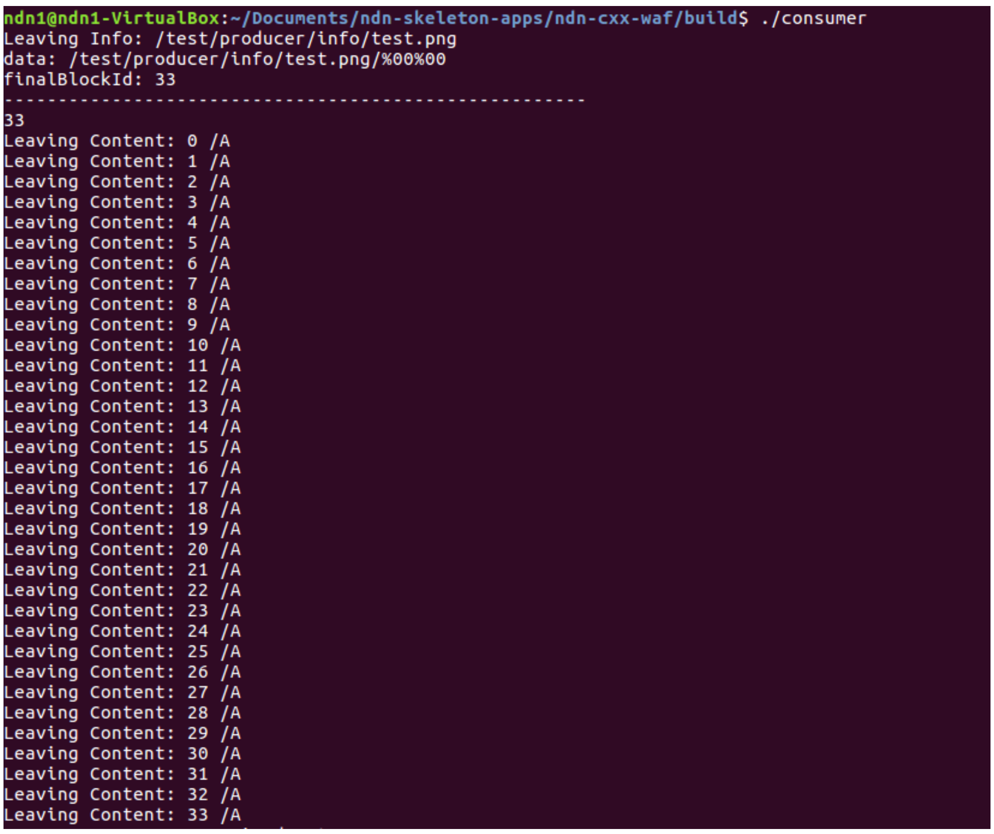


```
--------------------------------------------------------
Data: /test/producer/content/test.png/%00%02
Prefix: /test/producer/content
Segment No.: 2
Final Block No.: 8
Adding to Buffer
--------------------------------------------------------
Data: /test/producer/content/test.png/%00%03
Prefix: /test/producer/content
Segment No.: 3
Final Block No.: 8
Adding to Buffer
--------------------------------------------------------
Data: /test/producer/content/test.png/%00%04
Prefix: /test/producer/content
Segment No.: 4
Final Block No.: 8
Adding to Buffer
--------------------------------------------------------
Data: /test/producer/content/test.png/%00%05
Prefix: /test/producer/content
Segment No.: 5
Final Block No.: 8
Adding to Buffer
--------------------------------------------------------
Data: /test/producer/content/test.png/%00%06
Prefix: /test/producer/content
Segment No.: 6
Final Block No.: 8
Adding to Buffer
--------------------------------------------------------
Data: /test/producer/content/test.png/%00%07
Prefix: /test/producer/content
Segment No.: 7
Final Block No.: 8
Adding to Buffer
--------------------------------------------------------
Data: /test/producer/content/test.png/%00%08
Prefix: /test/producer/content
Segment No.: 8
Final Block No.: 8
Adding to Buffer
----------------------------------------------------
Creating File
Orig. BufferSize: 15391
Success
new bufferSize: 15391
Final Block ID: 8
SENDING
```

Figure 9.7: Equal to Function

9.3.2 Less than the Original Segment Count

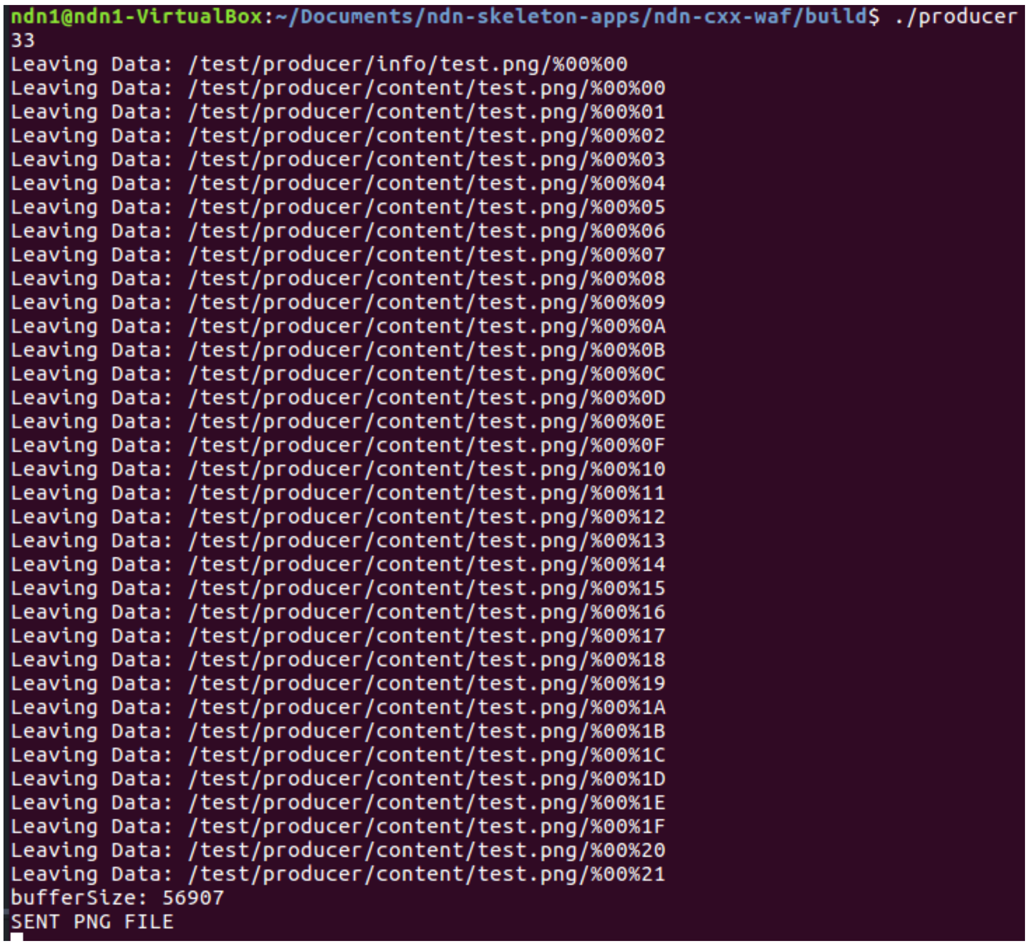
Fig. 9.8 shows that the consumer first sends out an Interest packet to get the Final Block ID, which in this case is 33. Next, the consumer sends out 34 Interest packets all with the Function Name field set to */A*.



```
ndn1@ndn1-VirtualBox:~/Documents/ndn-skeleton-apps/ndn-cxx-waf/build$ ./consumer
Leaving Info: /test/producer/info/test.png
data: /test/producer/info/test.png/%00%00
finalBlockId: 33
------------------------------------------------------
33
Leaving Content: 0 /A
Leaving Content: 1 /A
Leaving Content: 2 /A
Leaving Content: 3 /A
Leaving Content: 4 /A
Leaving Content: 5 /A
Leaving Content: 6 /A
Leaving Content: 7 /A
Leaving Content: 8 /A
Leaving Content: 9 /A
Leaving Content: 10 /A
Leaving Content: 11 /A
Leaving Content: 12 /A
Leaving Content: 13 /A
Leaving Content: 14 /A
Leaving Content: 15 /A
Leaving Content: 16 /A
Leaving Content: 17 /A
Leaving Content: 18 /A
Leaving Content: 19 /A
Leaving Content: 20 /A
Leaving Content: 21 /A
Leaving Content: 22 /A
Leaving Content: 23 /A
Leaving Content: 24 /A
Leaving Content: 25 /A
Leaving Content: 26 /A
Leaving Content: 27 /A
Leaving Content: 28 /A
Leaving Content: 29 /A
Leaving Content: 30 /A
Leaving Content: 31 /A
Leaving Content: 32 /A
Leaving Content: 33 /A
```

Figure 9.8: Less than Consumer Send

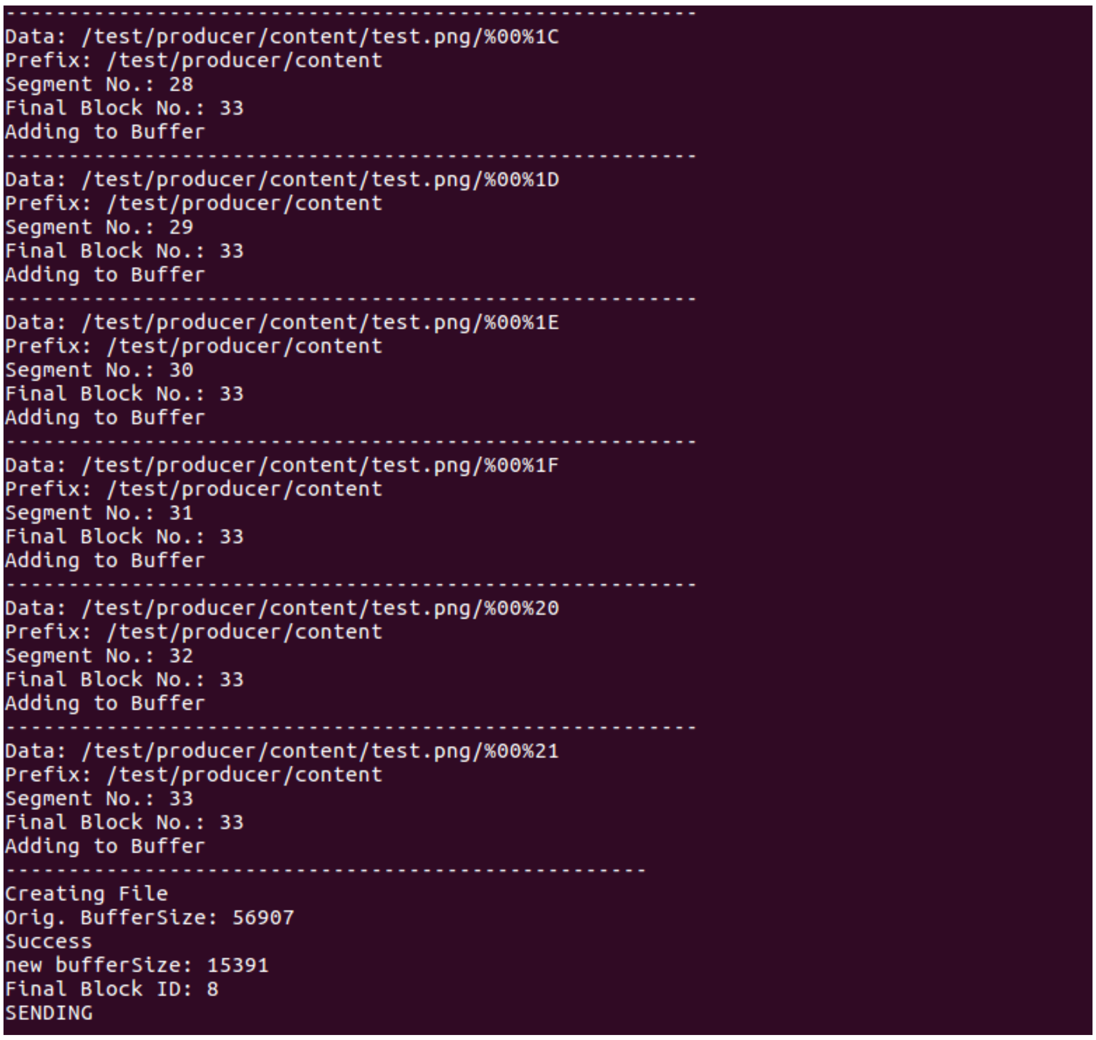
Fig. 9.9 shows that the producer received these Interest packets, and sent out the Data packets. The buffer size shows that the image was 56.9kB, which is equal to 34 segments(Final Block ID: 33).



```
ndn1@ndn1-VirtualBox:~/Documents/ndn-skeleton-apps/ndn-cxx-waf/build$ ./producer
33
Leaving Data: /test/producer/info/test.png/%00%00
Leaving Data: /test/producer/content/test.png/%00%00
Leaving Data: /test/producer/content/test.png/%00%01
Leaving Data: /test/producer/content/test.png/%00%02
Leaving Data: /test/producer/content/test.png/%00%03
Leaving Data: /test/producer/content/test.png/%00%04
Leaving Data: /test/producer/content/test.png/%00%05
Leaving Data: /test/producer/content/test.png/%00%06
Leaving Data: /test/producer/content/test.png/%00%07
Leaving Data: /test/producer/content/test.png/%00%08
Leaving Data: /test/producer/content/test.png/%00%09
Leaving Data: /test/producer/content/test.png/%00%0A
Leaving Data: /test/producer/content/test.png/%00%0B
Leaving Data: /test/producer/content/test.png/%00%0C
Leaving Data: /test/producer/content/test.png/%00%0D
Leaving Data: /test/producer/content/test.png/%00%0E
Leaving Data: /test/producer/content/test.png/%00%0F
Leaving Data: /test/producer/content/test.png/%00%10
Leaving Data: /test/producer/content/test.png/%00%11
Leaving Data: /test/producer/content/test.png/%00%12
Leaving Data: /test/producer/content/test.png/%00%13
Leaving Data: /test/producer/content/test.png/%00%14
Leaving Data: /test/producer/content/test.png/%00%15
Leaving Data: /test/producer/content/test.png/%00%16
Leaving Data: /test/producer/content/test.png/%00%17
Leaving Data: /test/producer/content/test.png/%00%18
Leaving Data: /test/producer/content/test.png/%00%19
Leaving Data: /test/producer/content/test.png/%00%1A
Leaving Data: /test/producer/content/test.png/%00%1B
Leaving Data: /test/producer/content/test.png/%00%1C
Leaving Data: /test/producer/content/test.png/%00%1D
Leaving Data: /test/producer/content/test.png/%00%1E
Leaving Data: /test/producer/content/test.png/%00%1F
Leaving Data: /test/producer/content/test.png/%00%20
Leaving Data: /test/producer/content/test.png/%00%21
bufferSize: 56907
SENT PNG FILE
```

Figure 9.9: Less than Producer

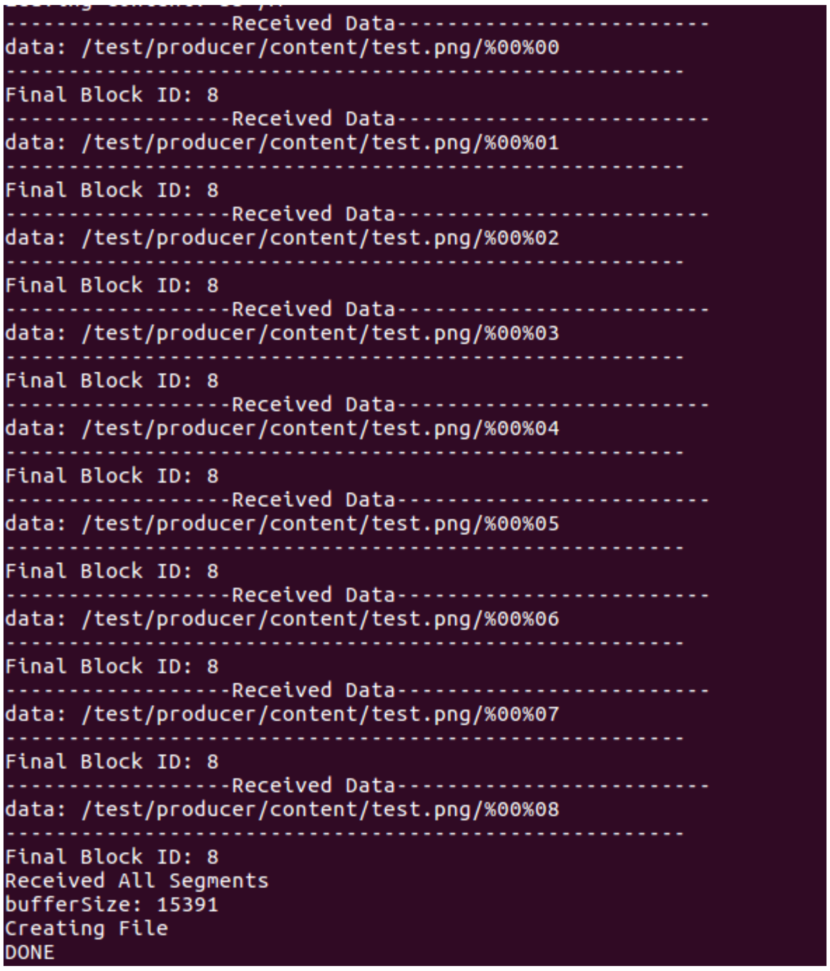
Fig. 9.7 shows that the function */A* received the Data packets from the producer. It reassembles the content (buffer size 56.9kB), and processes the image. The post-process image is 15.4kB, which has a Final Block ID of 8. This is sent back to the consumer.



```
-------------------------------------------------------
Data: /test/producer/content/test.png/%00%1C
Prefix: /test/producer/content
Segment No.: 28
Final Block No.: 33
Adding to Buffer
-------------------------------------------------------
Data: /test/producer/content/test.png/%00%1D
Prefix: /test/producer/content
Segment No.: 29
Final Block No.: 33
Adding to Buffer
-------------------------------------------------------
Data: /test/producer/content/test.png/%00%1E
Prefix: /test/producer/content
Segment No.: 30
Final Block No.: 33
Adding to Buffer
-------------------------------------------------------
Data: /test/producer/content/test.png/%00%1F
Prefix: /test/producer/content
Segment No.: 31
Final Block No.: 33
Adding to Buffer
-------------------------------------------------------
Data: /test/producer/content/test.png/%00%20
Prefix: /test/producer/content
Segment No.: 32
Final Block No.: 33
Adding to Buffer
-------------------------------------------------------
Data: /test/producer/content/test.png/%00%21
Prefix: /test/producer/content
Segment No.: 33
Final Block No.: 33
Adding to Buffer
---------------------------------------------------
Creating File
Orig. BufferSize: 56907
Success
new bufferSize: 15391
Final Block ID: 8
SENDING
```

Figure 9.10: Less than Function

Finally, Fig 9.11 shows that the consumer received all 9 segments that came from function $/A$.

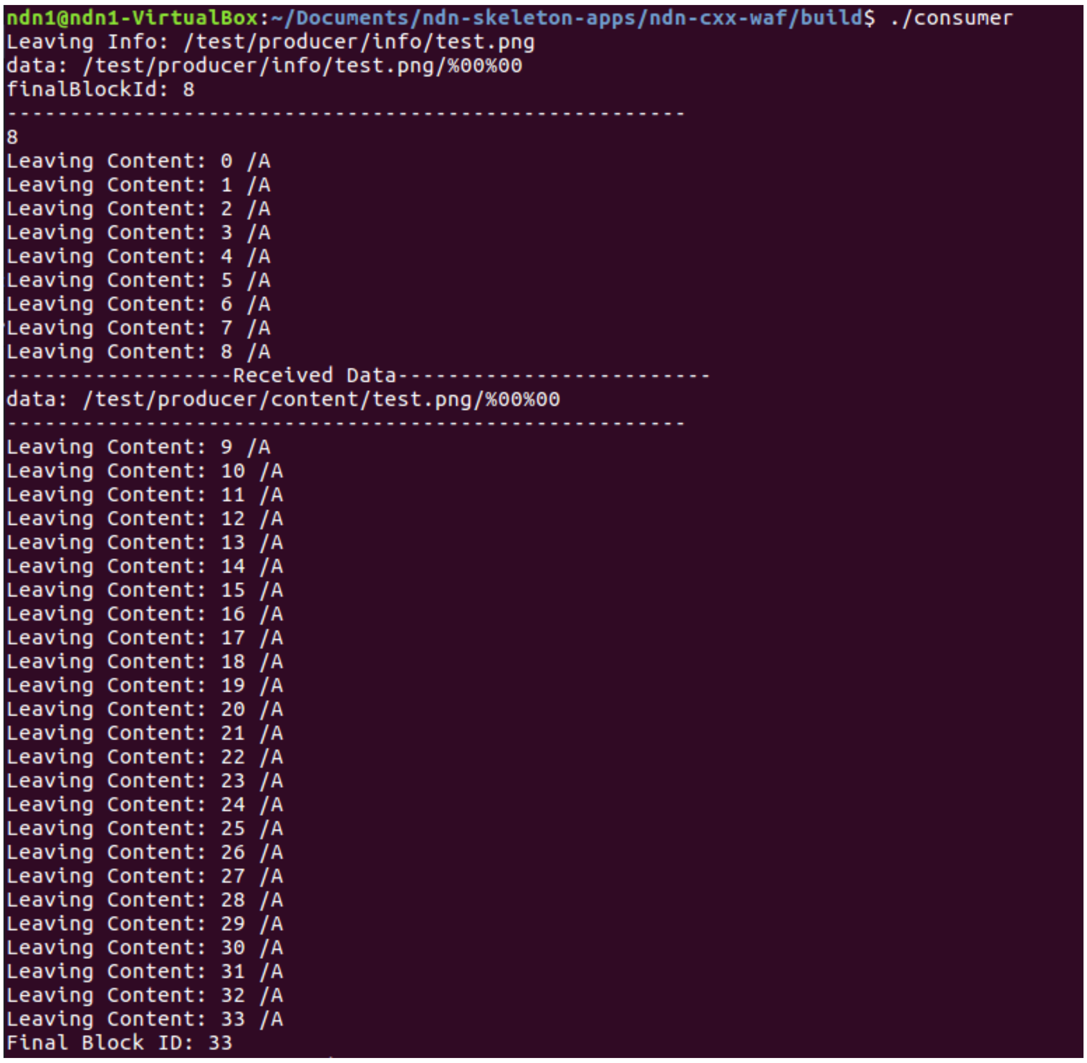


```
------------------Received Data-------------------------
data: /test/producer/content/test.png/%00%00
------------------------------------------------------
Final Block ID: 8
------------------Received Data-------------------------
data: /test/producer/content/test.png/%00%01
------------------------------------------------------
Final Block ID: 8
------------------Received Data-------------------------
data: /test/producer/content/test.png/%00%02
------------------------------------------------------
Final Block ID: 8
------------------Received Data-------------------------
data: /test/producer/content/test.png/%00%03
------------------------------------------------------
Final Block ID: 8
------------------Received Data-------------------------
data: /test/producer/content/test.png/%00%04
------------------------------------------------------
Final Block ID: 8
------------------Received Data-------------------------
data: /test/producer/content/test.png/%00%05
------------------------------------------------------
Final Block ID: 8
------------------Received Data-------------------------
data: /test/producer/content/test.png/%00%06
------------------------------------------------------
Final Block ID: 8
------------------Received Data-------------------------
data: /test/producer/content/test.png/%00%07
------------------------------------------------------
Final Block ID: 8
------------------Received Data-------------------------
data: /test/producer/content/test.png/%00%08
------------------------------------------------------
Final Block ID: 8
Received All Segments
bufferSize: 15391
Creating File
DONE
```

Figure 9.11: Less than Consumer Receive

9.3.3 Greater than the Original Segment Count

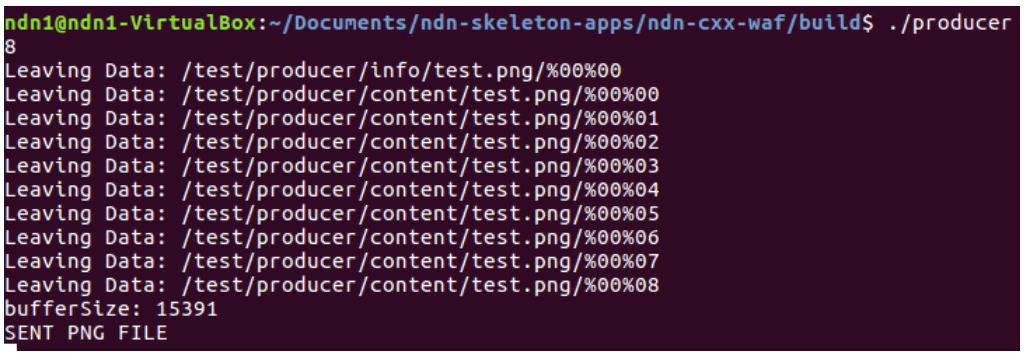
Similar to the above, the consumer will first send out an Interest packet to get the Final Block ID, which in this case is 8. Next, the consumer sends out 9 Interest packets all with the Function Name field set to */A*.



```
ndn1@ndn1-VirtualBox:~/Documents/ndn-skeleton-apps/ndn-cxx-waf/build$ ./consumer
Leaving Info: /test/producer/info/test.png
data: /test/producer/info/test.png/%00%00
finalBlockId: 8
-----------------------------------------------------
8
Leaving Content: 0 /A
Leaving Content: 1 /A
Leaving Content: 2 /A
Leaving Content: 3 /A
Leaving Content: 4 /A
Leaving Content: 5 /A
Leaving Content: 6 /A
Leaving Content: 7 /A
Leaving Content: 8 /A
-----------------Received Data------------------------
data: /test/producer/content/test.png/%00%00
-----------------------------------------------------
Leaving Content: 9 /A
Leaving Content: 10 /A
Leaving Content: 11 /A
Leaving Content: 12 /A
Leaving Content: 13 /A
Leaving Content: 14 /A
Leaving Content: 15 /A
Leaving Content: 16 /A
Leaving Content: 17 /A
Leaving Content: 18 /A
Leaving Content: 19 /A
Leaving Content: 20 /A
Leaving Content: 21 /A
Leaving Content: 22 /A
Leaving Content: 23 /A
Leaving Content: 24 /A
Leaving Content: 25 /A
Leaving Content: 26 /A
Leaving Content: 27 /A
Leaving Content: 28 /A
Leaving Content: 29 /A
Leaving Content: 30 /A
Leaving Content: 31 /A
Leaving Content: 32 /A
Leaving Content: 33 /A
Final Block ID: 33
```

Figure 9.12: Greater than Consumer Send

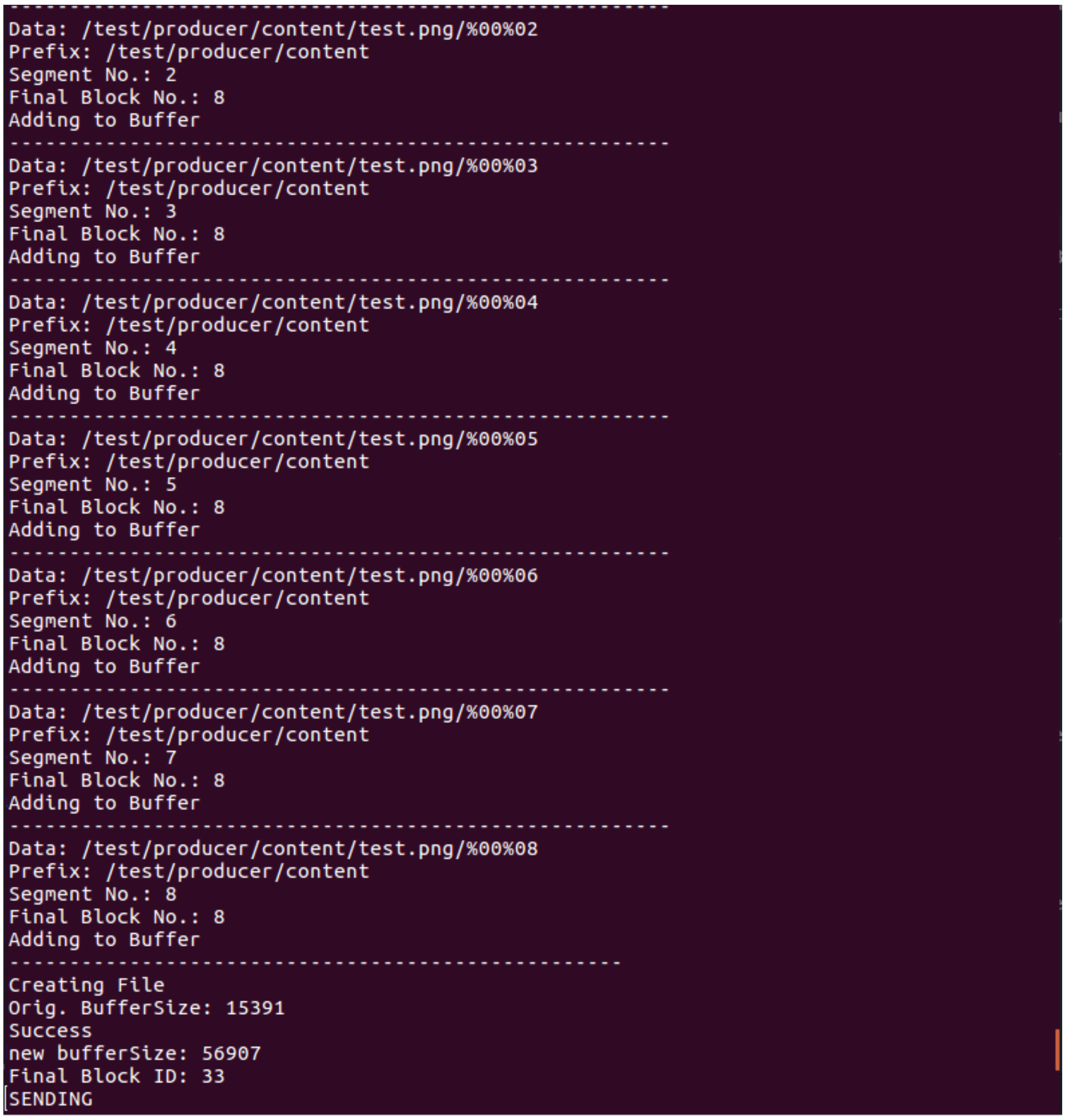
Fig. 9.13 shows that the producer received these Interest packets, and sent out the Data packets. The buffer size shows that the image was 15.4kB, which is equal to 9 segments(Final Block ID: 8).



```
ndn1@ndn1-VirtualBox:~/Documents/ndn-skeleton-apps/ndn-cxx-waf/build$ ./producer
8
Leaving Data: /test/producer/info/test.png/%00%00
Leaving Data: /test/producer/content/test.png/%00%00
Leaving Data: /test/producer/content/test.png/%00%01
Leaving Data: /test/producer/content/test.png/%00%02
Leaving Data: /test/producer/content/test.png/%00%03
Leaving Data: /test/producer/content/test.png/%00%04
Leaving Data: /test/producer/content/test.png/%00%05
Leaving Data: /test/producer/content/test.png/%00%06
Leaving Data: /test/producer/content/test.png/%00%07
Leaving Data: /test/producer/content/test.png/%00%08
bufferSize: 15391
SENT PNG FILE
```

Figure 9.13: Greater than Producer

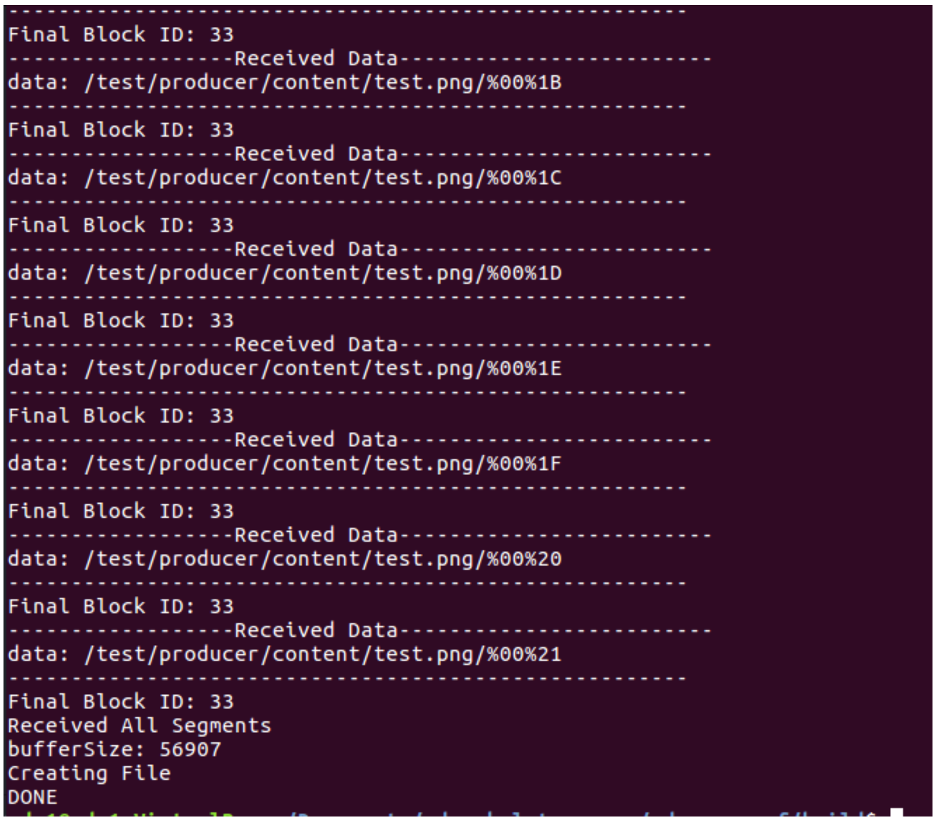
Fig. 9.14 shows that the function */A* received the Data packets from the producer. It reassembles the content (buffer size 15.4kB), and processes the image. The post-process image is 56.9kB, which has a Final Block ID of 33. The first 9 Interest packets are sent back to the consumer (the first post-processed Data packet received by the consumer can be seen in Fig. 9.12), so the consumer knows the new Final Block ID of 33. Now that the consumer knows the new Final Block ID, it will send out Interest packets with segment number 9 through 33.



```
-------------------------------------------------------
Data: /test/producer/content/test.png/%00%02
Prefix: /test/producer/content
Segment No.: 2
Final Block No.: 8
Adding to Buffer
-------------------------------------------------------
Data: /test/producer/content/test.png/%00%03
Prefix: /test/producer/content
Segment No.: 3
Final Block No.: 8
Adding to Buffer
-------------------------------------------------------
Data: /test/producer/content/test.png/%00%04
Prefix: /test/producer/content
Segment No.: 4
Final Block No.: 8
Adding to Buffer
-------------------------------------------------------
Data: /test/producer/content/test.png/%00%05
Prefix: /test/producer/content
Segment No.: 5
Final Block No.: 8
Adding to Buffer
-------------------------------------------------------
Data: /test/producer/content/test.png/%00%06
Prefix: /test/producer/content
Segment No.: 6
Final Block No.: 8
Adding to Buffer
-------------------------------------------------------
Data: /test/producer/content/test.png/%00%07
Prefix: /test/producer/content
Segment No.: 7
Final Block No.: 8
Adding to Buffer
-------------------------------------------------------
Data: /test/producer/content/test.png/%00%08
Prefix: /test/producer/content
Segment No.: 8
Final Block No.: 8
Adding to Buffer
---------------------------------------------------
Creating File
Orig. BufferSize: 15391
Success
new bufferSize: 56907
Final Block ID: 33
SENDING
```

Figure 9.14: Greater than Function

Finally, Fig 9.15 shows that the consumer received all 34 segments that came from function */A*.



```
------------------------------------------------------
Final Block ID: 33
------------------Received Data------------------------
data: /test/producer/content/test.png/%00%1B
------------------------------------------------------
Final Block ID: 33
------------------Received Data------------------------
data: /test/producer/content/test.png/%00%1C
------------------------------------------------------
Final Block ID: 33
------------------Received Data------------------------
data: /test/producer/content/test.png/%00%1D
------------------------------------------------------
Final Block ID: 33
------------------Received Data------------------------
data: /test/producer/content/test.png/%00%1E
------------------------------------------------------
Final Block ID: 33
------------------Received Data------------------------
data: /test/producer/content/test.png/%00%1F
------------------------------------------------------
Final Block ID: 33
------------------Received Data------------------------
data: /test/producer/content/test.png/%00%20
------------------------------------------------------
Final Block ID: 33
------------------Received Data------------------------
data: /test/producer/content/test.png/%00%21
------------------------------------------------------
Final Block ID: 33
Received All Segments
bufferSize: 56907
Creating File
DONE
```

Figure 9.15: Greater than Consumer Receive

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

Chapter 10

Conclusion and Future Work

10.1 Conclusion

In this research, we have discussed the necessity of function chaining especially in IoT environments. We have also proposed an architecture for supporting push and pull type function chaining in NDN for IoT environments. While many papers use simulators(mainly ndnSIM) to show proof-of-concept, we have focused more on real-world usability by implementing and testing on real machines. Many papers present theoretic feasibility of their proposed function chaining architecture, but do not discuss segmentation. Since most content does not fit into a single Data packet, we believe segmentation an important aspect of real-world application. Therefore, we have focused on implementing segmentation into NDN-FC using existing NDN software.

10.2 Future Work

Our future work includes things such as implementing push type NDN-FC, and testing with more complex scenarios. We have attempted implementing push type NDN-FC, but we have had complications with NFD when adding a payload into the Interest packet. This will require further research of the ndn-cxx library and NFD. 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 packets according to network load and congestion.

Through this research, we hope we can encourage more research of function chaining in NDN.

10.3 Acknowledgments

The results of this research are based on the the Ministry of Internal Affairs and Communications' project "R & D of fundamental technologies for effective use of wired/wireless optimal control type radio waves corresponding to increased IoT equipment" for the research and development of the expansion of radio wave resources.

I deeply appreciate Professor Hidenori Nakazato who has given me much guidance on a daily basis. Also, I would like to thank everyone and seniors of the Nakazato laboratory whom I have been studying together with.

10.4 Code for NDN-FC

- ndn-cxx-FC: https://github.com/oookabutoooo/ndn-cxx-FCv2.git
- NFD-FC: https://github.com/oookabutoooo/NFD-FC.git
- Consumer Producer API: https://github.com/oookabutoooo/Consumer-Producer-API-FC.git
- Applications: https://github.com/oookabutoooo/ndn-skeleton-apps.git

Bibliography

[1] ndn-cxx: Ndn c library with experimental extensions 0.6.3-41-g322e76e7 documentation. Available at https://named-data.net/doc/ndn-cxx/current/README.html.

[2] Nfd - named data networking forwarding daemon 0.6.4-27-g9120cee documentation. Available at https://named-data.net/doc/NFD/current/overview.html.

[3] What is edge computing? Available at https://www.ge.com/digital/blog/what-edge-computing.

[4] What is network service chaining or service function chaining. Available at https://www.sdxcentral.com/sdn/network-virtualization/definitions/what-is-network-service-chaining/.

[5] What's network functions virtualization (nfv)? Available at https://www.sdxcentral.com/nfv/definitions/whats-network-functions-virtualization-nfv/.

[6] What's software-defined networking (sdn)? Available at https://www.sdxcentral.com/sdn/definitions/what-the-definition-of-software-defined-networking-sdn/.

[7] B. Zhang L. Zhang I. Moiseenko Y. Yu W. Shang Y. Li S. Mastorakis Y. Huang J. P. Abraham E. Newberry S. DiBenedetto C. Fan C. Papadopoulos D. Pesavento G. Grassi G. Pau H. Zhang T. Song H. Yuan H. B. Abraham P. Crowley S. O. Amin V. Lehman A. Afanasyev, J. Shi and L. Wang. Nfd developer ' s guide. In *NDN, Technical Report NDN-0021*, 2015.

[8] A. E. Fergougui K. Benzekki and A. E. Elalaoui. Software-defined networking (sdn): a survey. *Security and Communication Networks*, 9(18):5803–5833, 2016.

[9] M. Bahrami L. Xie A. Ito S. Mnatsakanyan G. Qu Z. Ye L. Liu, Y. Peng and H. Guo. Icn-fc: An information-centric networking based framework for efficient functional chaining. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–7, May 2017.

[10] C. Scherb M. Sifalakis, B. Kohler and C. Tschudin. An information centric network for computing the distribution of computations. In *Proceedings of the 1st ACM Conference on Information-Centric Networking*, ACM-ICN '14, pages 137–146, New York, NY, USA, 2014. ACM.

[11] Ilya Moiseenko and Lixia Zhang. Consumer-producer api for named data networking. In *NDN, Technical Report NDN-0017*, 2014.

[12] C. Tschudin and M. Sifalakis. Named functions and cached computations. In *2014 IEEE 11th Consumer Communications and Networking Conference (CCNC)*, pages 851–857, Jan 2014.

Publications

1. Hiroki Yoshii. Implementing and running functions on ndn function chaining. In *2018 Waseda-UKM-UMS-Hanyang IT Workshop*, 2018.
2. Hiroki Yoshii. IoT function chaining in ndn. In *2017 Waseda-UKM-UMS-Hanyang IT Workshop*, 2017.
3. Hiroki Yoshii and Hidenori Nakazato. Real-world implementation of function chaining in named data networking for iot environments,. In *IEEE ICC 2019 Workshop Information Centric Networks Solutions For Real World Applications (ICN-SRA)*, 2019（投稿中）.
4. 吉井宏希, 中里秀則. NDN におけるファンクション・チェイニングの実装. 技術研究報告 CS2018-39, 電子情報通信学会, 7 月 2018.
5. 吉井宏希, 白岩善昭, 中里秀則. NDN における IoT データのファンクション・チェイニング. 2017 年電子情報通信学会通信ソサイエティ大会講演論文集, pp. B–8–1, 9 月 2017.