Title
Consumer / Producer communication with application level framing in Named Data Networking

Author
Moiseenko, Ilya

Publication Date
2015

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

Consumer / Producer communication with application level framing in Named Data Networking

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Ilya Moiseenko

2015

ABSTRACT OF THE DISSERTATION

Consumer / Producer communication with application level framing in Named Data Networking

by

Ilya Moiseenko

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2015

Professor Lixia Zhang, Chair

Over time the Internet has evolved from a network that interconnects hosts to a network that interconnects information objects broadly defined; these objects range from movie files, Facebook content, twitter messages, to sensor data and authenticated device actuation commands.

As a newly proposed architecture to meet this new usage, Named Data Networking (NDN) replaces the host-based addressing scheme by names of information objects in moving packets through the network. In an NDN network consumers send *Interest packets* carrying application-level names to request information objects, and the network returns the requested *Data packets* reversing the path of the Interests. As a new way of doing networking, NDN introduces new design patterns for applications. To make the content available in the network, one needs to consider multiple design choices, which range from name structure and security model to more basic issues such as data segmentation. To fetch content, one also faces new considerations such as the presence of caching in the network and the question of data validation, in addition to conventional issues of loss recovery and error corrections. What kind of application interface should be provided to ease the application development? And what protocols would be needed to

support the interface? Clearly socket abstraction and associated protocols cannot be reused, because the model of a virtual channel between two communicating processes supported by socket does not exist in the NDN architecture.

This dissertation addresses the above challenges with the design of a new Consumer / Producer API and its associated protocol suite that can play socket-equivalent roles in an NDN network. We demonstrate how the API can be used to invoke the most fundamental communication patterns in an NDN network and also show how stand-alone and web-based applications can be designed on top of the API. While the architecture of NDN is still a subject to the changes, we believe that the Consumer / Producer API framework can be successfully extended to accommodate new protocols, communication patterns and applications.

The dissertation of Ilya Moiseenko is approved.

Tarek Abdelzaher

Mario Gerla

Songwu Lu

Lixia Zhang, Committee Chair

University of California, Los Angeles

2015

Table of Contents

List of Figures

Acknowledgments

I would like to gratefully and sincerely thank my academic advisor Prof. Lixia Zhang for providing invaluable support throughout my Ph.D. program. I also want to thank Jeff Burke, David Oran, Mark Stapp, and my colleagues from UCLA's Internet Research Laboratory Alexander Afanasyev, Yingdi Yu, Wentao Shang and others for their support and deep technical discussions.

Chapter 6 is a version of [MSO14]. David Oran (PI) contributed "Compressed Name Component" pattern (Section 6.2.2) and Mark Stapp contributed "Application Data Field" pattern (Section 6.2.4).

I have a special thanks to Lijing Wang who had developed NDNtube and NDNlive video applications, Valerie Runge who had developed NDNRadio application, Mickey Sweatt who had developed a NDN Bittorent application and other people who were the first to use the API in their projects. Consumer / Producer model is an experimental research, and having other people build the software with it helped to verify and validate its design and discover all sorts of problems in the software library implementation.

Vita

2009	B.Tech. (Computer Science), Bauman Moscow State Technical University, Moscow, Russia.
2011	M.Tech. (Computer Science), Bauman Moscow State Technical University, Moscow, Russia.
2013	M.S. (Computer Science), UCLA, Los Angeles, California.
2011–2015	Graduate Research Assistant, Computer Science Department, UCLA.
2012–2015	Teaching Assistant, Computer Science Department, UCLA.
2012	Research Intern, Palo Alto Research Center (PARC), Palo Alto, California.
2013	Research Intern, Qualcomm Research Center, Bridgewater, New Jersey.
2014	Research Intern, Cisco Systems, Cambridge, Massachusetts.
2015	Research Software Engineer, Cisco Systems, Cambridge, Massachusetts.

Publications

A. Afanasyev, I. Moiseenko, and L. Zhang, "ndnSIM: NDN simulator for NS-3," *Tech.Report NDN-0005*, 2012

C. Yi, A. Afanasyev, I. Moiseenko, L. Wang, B. Zhang, and L. Zhang, "A Case for Stateful Forwarding Plane," *Computer Communications*, vol. 36, no. 7, 2013.

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

A. Attam, I. Moiseenko, "NDNBlue: NDN over Bluetooth," Technical Report NDN-0015, NDN, 2013.

A. Afanasyev, J. Shi, B. Zhang, L. Zhang, I. Moiseenko, Y. Yu, W. Shang, et al. "NFD developer's guide," Technical Report NDN-0021, NDN, 2014.

I. Moiseenko, M. Stapp, and D. Oran, "Communication patterns for web interaction in Named Data Networking," in *Proc. of ICN Sigcomm 2014,* 2014.

I. Moiseenko, "Fetching content in Named Data Networking with embedded manifests," Technical Report NDN-0025, NDN, 2014.

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

L. Wang, I.Moiseenko and L.Zhang, "NDNlive and NDNtube: Live and Prerecorded Video Streaming over NDN," Report NDN-0031, NDN, 2015.

I. Moiseenko, L. Wang, and L. Zhang, "Consumer / Producer communication with application level framing in Named Data Networking," in *Proc. of ICN Sigcomm 2015,* 2015.

CHAPTER 1

Introduction

Today's Internet architecture stands on IP — a universal network layer designed to create a point-to-point communication network where packets are delivered to specific destinations, enabling direct process-to-process communication. This was a premise for introducing the concept of the socket, which binds a running process to a communication channel, and represents a container for the current state of data transfer between two processes [JM71, JFL86].

Over time Internet has evolved from a network that interconnects hosts to a network that interconnects information objects broadly defined; these objects range from movie files to Facebook content to sensor data and to authenticated device actuation commands. This fundamental change in its usage suggests that the Internet's universal network layer will be much more organic as a distribution network natively working with information objects instead of communication endpoints.

As a newly proposed architecture to meet this new usage, Named Data Networking (NDN) replaces the host-based addressing scheme by names of information objects to move packets in the network [JST09, L 10, ZAB14]. In an NDN network consumers send Interest packets carrying application-level names to request information objects, and the network returns the requested Data packets following the path of the Interests. NDN strengthens information safety with the concept of content-based security. The data itself is secured with publicly verifiable signature and encryption (Section 2.2).

1.1 Challenges of developing NDN applications

As explained in [CT90], network applications work with Application Data Units (ADU) — units of data represented in a most suitable form for each given use-case. For example, a video playback application typically handles data in the unit of video frames; a multi-user game's ADUs are objects representing users' current status; and for an intelligent home application, ADUs may represent sensor readings. NDN enables applications to communicate using ADUs.

As a new way of doing networking, NDN introduces new design patterns for applications. To make the content available through the network, one needs to consider multiple design choices, which range from name structure and security model to more basic issues such as data segmentation. To fetch content, one also faces new considerations such as the presence of caching in the network and the question of data validation, in addition to conventional issues of data loss recovery and error corrections. What kind of application interface should be provided to ease the application development? And what protocols would be needed to support the interface? Clearly socket abstraction and associated protocols cannot be reused, because the model of a virtual channel between two communicating processes supported by a socket does not exist in the NDN architecture.

1.2 Design goals

In this work we aim to create a comprehensive design of a communication model — API and necessary underlying components that:

- gives application developers adequate freedom when handling ADUs, at the same time minimizes the complexity associated with the production and retrieval of ADUs of any size;
- has built-in content retrieval and segmentation protocols and can host newly

developed protocols;

- can be applied to a broad range of NDN applications;
- significantly simplifies application development and does not require application developers to be experts in NDN technology.

In TCP/IP networking, similar tasks are managed by the socket API. A socket is a container for data transfer parameters holding the current state of transmission in a virtual channel between two processes running on IP hosts. Because a socket creates a duplex pipe for data to flow in both directions, server and client applications use sockets in more or less the same way with a few minor differences (e.g. *listen()* and *accept()* calls). Socket has no use without being attached to the channel (e.g. *bind()* or *connect()*). To support "time asynchrony" or delay tolerance between communicating parties, application developers often resort to higher level abstractions (e.g. ZeroMQ [M]) suitable for queuing and passing messages.

NDN is a pull-based data dissemination protocol, therefore applications that consume data behave differently from applications that produce data. Consequently these applications need different sets of data transfer parameters. Producer applications, in general, care about ADU segmentation, securing and caching / storing Data packets, and incoming Interest demultiplexing. Consumer applications, on the other hand, care about fetching all Data packets of each ADU, fetching reliability, verification of received data, as well as flow and congestion control by controlling their Interest generation rates.

These observations prompt us to design two programming abstractions: one for consumer applications, and another one for producer applications.

To have data delivered over the Internet, large ADUs must be segmented, because the packet size is limited by network MTU. There are two major differences in how TCP/IP and NDN handle data segmentation. First, because TCP treats

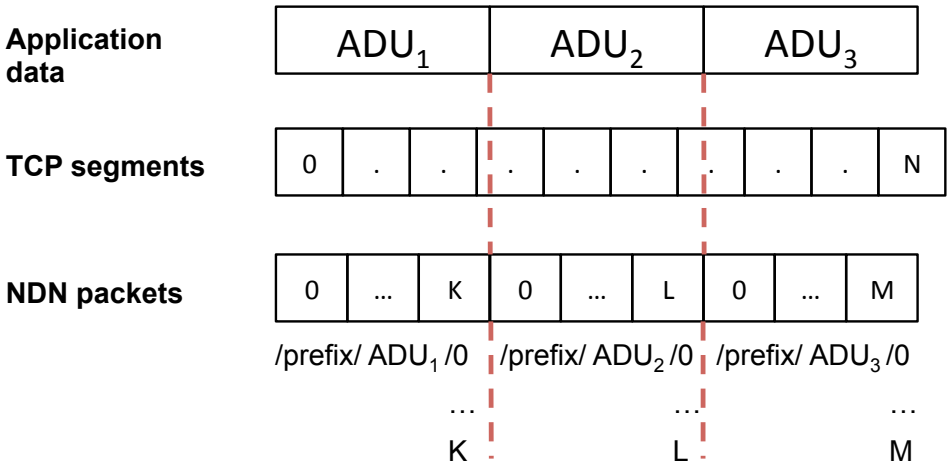

Figure 1.1: TCP/IP segmentation does not preserve boundaries of application frames (ADUs). NDN segmentation exposes these boundaries through naming.

all application data as byte streams, TCP segmentation ignores ADU boundaries, thus ADUs can only be identified after the segment reassembly (Figure 1.1). NDN data packets carry the names of individual ADUs or ADU segments, therefore these packets match to application's data units directly.[1]

The second, and related, difference is the degree of insight and control that application can have during data transfer. In the simple example shown in Figure 1.1, if TCP/IP is used to send several ADUs back to back across the network and one of the segments is lost in transit, all the subsequent ADUs, even if they arrive at the destination, will be blocked from getting delivered to the application. This is a well known head-of-line (HOL) blocking problem. On the other hand, if NDN is used and faces the same segment loss problem, all successfully received ADUs can be immediately delivered to the applications without waiting for the recovery of the missing segment.

[1]The terms "ADU segment", "data segment", and "Data packet" are used interchangeably in this dissertation.

1.3 Consumer / Producer API design overview

At the present time, NDN applications are developed on top of the basic NDN network abstractions: Interest and Data packets. All major functionality such as ADU segmentation, queueing, ADU reassembly, Interest retransmission, loss and error recovery, etc. is a direct responsibility of the application. As a result, application development process takes a lot of time and effort, and often requires deep expertise with Named Data Networking technology.

Since sockets are the de facto standard API for network programming, Consumer / Producer API was designed to allow developing NDN applications in a similar to socket API manner contrary to the current practice of using some arbitrary set of NDN concepts loosely related to each other. The biggest conceptual difference between the socket API and Consumer / Producer API is that our API is used for transferring ADUs in and out of the NDN network, whereas sockets are used for transferring byte-streams or datagrams between IP endpoints.

Consumer and producer contexts have a similar life-cycle to sockets that consists of four stages: (1) creation and deletion, (2) configuration, (3) attaching to the network, and (4) data transfer (Figure 1.2).

Function	TCP/IP socket API	NDN Consumer/Producer API
Creation and deletion	socket(), close()	consumer(), producer(), delete()
Configuration	setsockopt(), getsockopt()	setcontextopt(), getcontextopt()
Attaching to network	bind(), connect()	attach() *
Data transfer	send(), receive()	consume(), produce(), nack()

Figure 1.2: Consumer and producer contexts have a similar life-cycle to the socket API.

Sockets are not capable of performing any operations if not attached to the

network via *bind()* and *connect()*. This is different from producer context, which can publish data to in-memory or persistent storage (e.g. in Repo) without being attached to the network. This functionality is highly important for a large number of applications, because Consumer / Producer communication in NDN is inherently time decoupled. Time asynchrony can extend to relatively extreme levels when data is published into a permanent storage far ahead (e.g. hours, weeks, months) of any possible data fetching.

From the viewpoint of software engineering, Consumer / Producer framework adheres to the Inversion of Control (IoC) design pattern for the purpose of increasing the modularity of the software library and making it extensible for ongoing research work. In particular, the modularity is necessary for experimentation with trust models, and other security related functionality. Inversion of Control is a design in which custom-written portions of a computer program receive the flow of control from a generic, reusable library. An IoC software architecture inverts the control as compared to traditional procedural programming: in traditional programming, the custom code that expresses the purpose of the program calls into reusable libraries to take care of generic tasks, but with inversion of control, it is the reusable code that calls into the custom, or task-specific, code (e.g. event-handlers, callbacks, user-defined routines).

1.4 Contributions of this work

Contributions of this dissertation can be summarized as follows:

- Design and implementation of Consumer / Producer contexts — programming abstractions specifically tailored for data dissemination in NDN networks (Chapter 3).
- Design and implementation of a number of supporting concepts: manifest

and application level negative acknowledgement, utilized by the protocols below the API to facilitate the operation of applications. (Chapter 3).

- Design and implementation of the initial Consumer / Producer protocol suite — data retrieval and segmentation protocols (Chapter 4).
- Description of communication patterns available via the Consumer / Producer model and evaluation of the Consumer / Producer model through a number of pilot applications built on top of the API (Chapter 5).
- Analysis of the bidirectional Consumer / Producer communication, which is necessary for web-based applications (Chapter 6).

CHAPTER 2

Background

This chapter briefly introduces Named Data Networking (NDN) architecture along with Application Level Framing (ALF) and Integrated Layer Processing (ILP) principles, which serve as a foundation for this thesis. NDN works in a fundamentally different way than IP. To help the reader better understand its basic concepts, in this chapter we first describe a toy application example, then use this example to explain the NDN basics and Consumer / Producer communication throughout the thesis.

2.1 A Simple-Video Application

A Simple-Video application is used as an example to help illustrate the concepts behind any rational application and possible problems that must be solved by an application developer, and the concepts in named data networking. Simple-Video produces two separate streams of data, video and audio, to give the consumers a choice of either watching the video with audio or listen to the audio only (Figure 2.1). Both video and audio streams consist of data frames, and the application should allow consumers to retrieve any frames independently. The consumer application can, for example, stop fetching audio frames when user hits the 'Mute' button, or can skip some video frames after a pause in order to catch up the actual 'live' video. In most cases, a video frame is likely too large to be encapsulated in a single network packet, and the video producer application would have to perform content segmentation in order to split one frame into multiple packets.

An existing technology, MPEG-DASH [Sto11] produces the content, with either a mixed audio/video stream or two separate streams, into a sequence of small file segments of equal time duration. File segments are later served over HTTP from the origin media server or intermediate HTTP caching servers. While there is a great variety in the way how application produces and fetches data at the level of application frames, there are repetitive and labour intensive tasks related to the segmentation of the application frames and retrieval of the segments that make an application frame. In the case of MPEG-DASH, all these low-level details are handled by the HTTP / TCP protocol machinery.

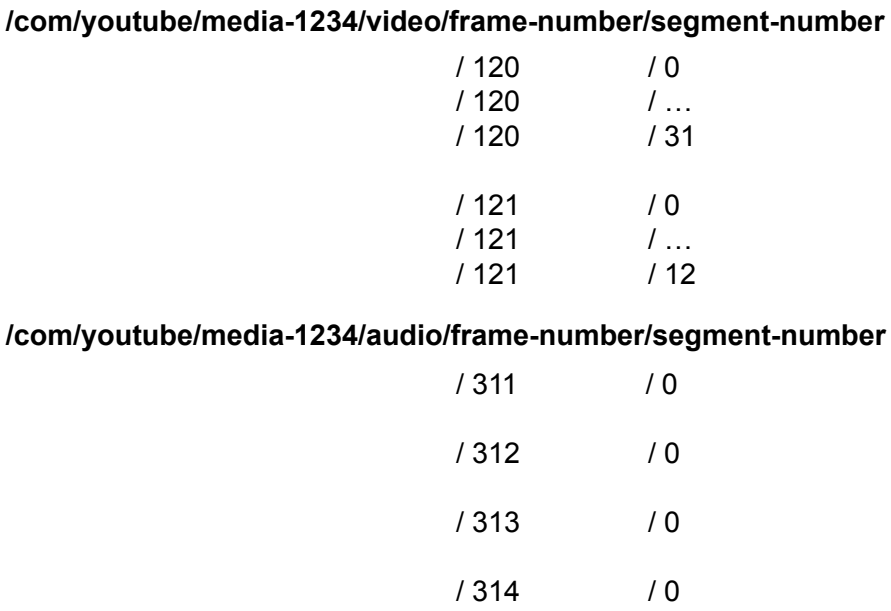

Figure 2.1: A naming scheme for an exemplary video playback application.

2.2 Named Data Networking (NDN) architecture

An NDN network works with two distinct types of packets: *Interest* and *Data* (Figure 2.2). Consumers send Interest packets, i.e. expressing Interests in receiving specific pieces of data. Producers produce Data packets to satisfy received

Interests. Both types of packets carry a data *name*, which uniquely identifies a piece of information object carried in a *single* Data packet. Data names in NDN are supplied by applications. Generally speaking, an NDN name is made of multiple components and is used to deliver the packet across network; it also contains application specific information to facilitate packet processing. As an example, Figure 2.1 shows the naming scheme used by Simple-Video in an NDN network. The name begins with routable components *"/com/youtube/"* which guides the Interest carrying this name toward the data producer. The next component is the name identifier of the media resource. The next name component separates video and audio frames into separate namespaces (e.g. name subtrees). Both video and audio frames are named sequentially. Each video frame consists of multiple segments, also named sequentially, while each audio frame is made of a single segment.

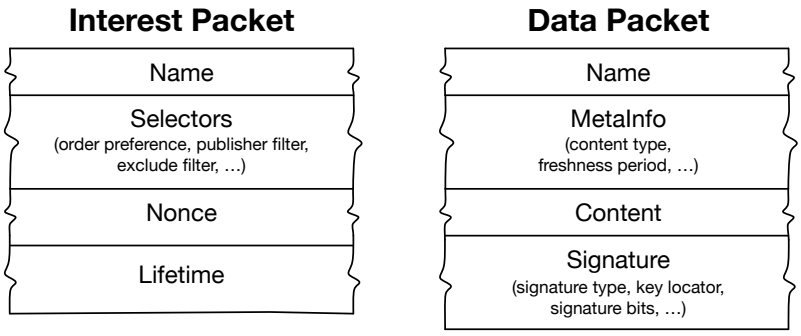

Figure 2.2: NDN packet types.

To retrieve a Data packet, a consumer application requests it by sending an Interest packet containing the name of the desired content. An NDN router looks up this name against its Forwarding Information Base (FIB) to forward the Interest towards likely locations of the data, and keeps a record of all forwarded Interest packets in a Pending Interest Table (PIT) until the corresponding Data packet is returned, or until the record times out. Each PIT entry contains an Interest name, the incoming interface(s) where the Interest arrived from (to support

multicast delivery of the returning Data packet), and the outgoing interface(s) to which the Interest has been forwarded (an Interest can be forwarded along multiple paths towards the data). When an Interest meets a Data packet whose name matches the name in the Interest, which can happen either at a router cache or at the data producer, the Data packet is returned to the consumer application by reversing the path of the Interest using the information present in routers's PITs. The routers along the way can also cache the Data packets.

An NDN network secures data by having the producer of the information append a cryptographic signature to bind the name to the content (and encrypt the payload of its outgoing Data packets if needed). Consumers can verify the signature of each received Data packet no matter from where it is retrieved. This security model decouples the trust in data from the place and time the data is obtained, enabling NDN nodes to cache any passing Data packets. Packet routers must have buffer space to support statistical multiplexing. An NDN router uses this buffer space for both statistical multiplexing and data caching, thus calls this buffer space Content Store (CS). A cached Data packet can be used to satisfy an Interest with a matching name. Note that an NDN network consider all data immutable and the name to data binding is unique, any updated data leads to a new version number in the name.

Similar to IP, an NDN network provides datagram delivery. Data consumers who desire reliable data fetching need to enable reliable fetching services as needed. Data consumers can also regulate data flow through pacing Interest packets transmissions.

Data production can be independent from data consumption. This inherent asynchrony creates quite delicate consumer-producer coordination issues, such as the availability of data, the specifics of data, etc. Interest *Selectors* is one of the available mechanisms for coordination between *multiple* consumers and *multiple* producers. Any Interest may carry optional Selectors that set additional condi-

tions for content retrieval from the cache or producer application. For example, when a consumer fetched a Data packet P that is not the desired one, it can try again by adding an Exclude selector to the Interest, where the Exclude selector may contain either name components of P or P's digest.

2.3 NDN inside a Node

NDN Forwarding Daemon (NFD) plays the role of multiplexer between applications and network interfaces inside a node [ASZ14]. NFD does not distinguish between applications and network interfaces, and views them as *Faces*. As mentioned above, NFD has a content store which performs opportunistic caching of passing by Data packets. NFD can also have a face to a local repository (e.g. Repo [CSC14]), which provides managed storage of Data packets.

To make data available, producer-process registers its name prefix with NFD to be able to receive interests for its data. The local NFD adds the prefix to its FIB and also forwards the prefix to next router. Consumer-process does not need registration — it simply sends Interest packets to the local NFD which then forwards the Interest toward the producer, either locally or remotely.

2.4 Repo

`Repo` is one of the core components for most NDN-based communications, providing long-term storage of NDN Data. After a producer application puts Data packets into Repo, the packets are used to satisfy Interests in the same way as packets residing in the cache of NFD.

The most fundamental difference between Repo and NFD cache is the fact that Repo is managed by the applications through a Repo API, whereas NFD cache has an application-agnostic data indexing and eviction techniques.

Repo API supports insertion and deletion of single and multiple Data packets at once. The general syntax of Repo API is:

"/<repo-prefix>/<command>/<CommandParameter>/<SignedInterestComponents>"

- <repo-prefix> is a prefix of a specific Repo node;
- <command> refers to the name of the operation (e.g. insert, delete);
- <CommandParameter> specifies names of the Data packets to be added or removed and other selection criteria;
- <SignedInterestComponents> contain information about the signature with replay protection.

Since it is not possible to push data in NDN network, the basic idea behind the data insertion API is to fetch Data from the application using an Interest-Interest-Data-Data exchange. The exchange is initiated by an application that issues a signed Interest towards a Repo node. Repo replies to the successfully verified insertion command with a single or multiple Interest packets that retrieve the Data packet(s) specified in the initial insertion command. After Data packet(s) are successfully inserted, the Repo closes the transaction by replying with a Data packet containing the completion status of an initial insertion command.

Deletion is much simpler as it only involves a regular Interest-Data exchange. It is possible to remove either a single specific Data packet by its name or multiple Data packets under a certain name prefix.

2.5 Application Level Framing (ALF)

The seminal paper [CT90], published 25 years ago, clearly articulated the value of applying the concept of application level framing to network protocol development by directly using application data unit (ADU). The second related principle is

the Integrated Layer Processing at the end hosts, which is enabled by the ADU concept.

Internet protocol suite consists of multiple layers, where socket API represents an interface between all application layers (application, presentation, session) and the transport layer. In other words, socket API performs conversion of data organized in application units (ADUs) into data organized into transmission units, which often creates problems related to retransmission, congestion/flow control, multiplexing, error correction, etc.

What defines a suitable size for an ADU? The fundamental characteristic of D. Clark's and D.Tennenhouse's definition is that each ADU can be processed out of order with respect to other ADUs. This rule permits the ADU boundaries to take the place of the packet boundaries for purposes of manipulation functions such as end-to-end error detecting codes or moving into application address space.

At the same time, the ADU now becomes the unit of error recovery. Since the ADU is defined to be the smallest unit which the application (or presentation conversion function) can deal with out of order, it follows that if even part of an ADU is lost in transmission, the application will, in general, be unable to deal with it. Since our application layer takes on the responsibility of recovering lost data, it will almost certainly need to assume the whole ADU is lost, even if parts exist. Unless the presentation layer can translate the identity of the lost data into terms the application understands, the application cannot understand which of its elements have actually been lost.

This suggests that ADU lengths should be reasonably bounded, so that when data is lost the application need do no more work than necessary. Indeed, since the loss of even one bit will trigger the loss of a whole ADU, excessively large ADUs might prevent useful progress at all, since the probability of any ADU having at least one uncorrected error would approach one.

However, there will be a minimum unit, based on the details of the application data, below which the application cannot break the data and still deal with parts lost or out of order. If this minimum natural size of the ADU is too large to provide a practical unit of error free transmission, then it will be necessary to define an artificial set of subunits into which an ADU is broken for error recovery. Although this partitioning may be provided by an independent protocol module, the overall responsibility for retransmission must rest with the application.

Integrated Layer Processing is the second important concept that was introduced in that paper. According to the paper, any communication protocol consists of two major parts: transfer control and data manipulation. The transfer control part processes the information in packet headers, and maintains the state needed for the protocol operation. Interest retransmission, packet ordering, congestion/flow control are the examples of transfer control functionality. The data manipulation part processes packet data. Examples of data manipulation functions are content segmentation, signing and encryption of Data packets, verification of Interest and Data packets. They all have in common that they process large amounts of data, which often involves data transformation and copying between memory buffers.

In a traditional implementation of a protocol stack, transfer control and data manipulation functions are often executed independently on separate layers. A vivid example of the traditional layered design is HTTP-TLS-TCP-IP bundle. HTTP messages are passed to the session layer, where they are encrypted and placed in the transport layer, where buffer with encrypted content is segmented in TCP segments.

Layered architecture provides isolation between distinct layers. A major architectural benefit of isolation is that it facilitates the implementation of subsystems whose scope is restricted to a small subset of the suites layers. However, it also causes sequential processing of each unit of information by each layer, which often

imposes precedence or ordering constraints that limit the opportunities for many useful optimizations.

The idea of integrated layer processing is to combine the data manipulation and transfer control functions of several traditional protocol layers into one processing loop. Because NDN operates with ADUs, it is now more feasible to process packets in one integrated processing loop. ADU-based protocols are more suitable for applying integrated layer processing than traditional protocols (i.e. IP), because ADU packets are independent from each other and therefore can be processed out of order.

A generic API for such integrated processing loop could potentially hide the complexity of communication protocols, while providing a way to customize data manipulation actions in critical areas such as confidentiality and privacy, event monitoring and error handling.

CHAPTER 3

Consumer / Producer communication abstractions

This chapter defines *consumer context* and *producer context* programming abstractions and describes the rationale behind their design. It also contains a discussion of the asynchrony between the consumers and producers that represents the most fundamental problem of communication in NDN, and describes the mechanisms capable of solving some of the issues related to the consumer / producer asynchrony.

3.1 Design goals for the producer abstraction

Given that NDN producers and consumers do not directly communicate, one basic question for producers is where to put the generated data. At this point the following three application patterns were identified to be so important that producer abstraction must naturally support them.

1. Realtime ADU publishing (and consumption), which can be used by a large number of applications including video conferencing, games, etc. Publishers may need to "wait for pull" and keep the ADUs in memory temporarily to handle a possible mismatch between production and consumption timing.

2. ADU publishing to stable storage, to support potentially large asynchronies between ADU publishing and consumption in terms of time, as well as in

terms of data popularity ("publish once - consume multiple times"). This publication pattern can be beneficial for static content services, such as video and web-content backend applications.

3. ADU publishing to remote stable storage, to support mobile publishers and IoT publishers. This publication pattern allows smartphones and sensors to get around their resource limitations by moving the content to stable locations.

3.2 Producer context

A **producer context** is used to publish data under a common prefix (Figure 3.2). It is initialized by calling **producer()** primitive with a given name prefix parameter (Table 3.1). Unlike a server side socket in TCP/IP, a producer context is ready to publish data even without being attached to the network and in the absence of any incoming Interests. Except the case of on-demand publishing, our consumer / producer model has no requirement for publishers and consumers being 'connected' at the same time, therefore data publication can take place any time, including when the producer is disconnected. In our Simple-Video example, the publisher publishes data at its own pace, ahead of fetching by any consumers.

Initialization	**producer** (name prefix) ➔ context
Primitives	**attach** (context) **produce** (context, name suffix, content) **nack** (context, negative acknowledgement) **delete** (context) **setcontextopt** (context, option name, value) **getcontextopt** (context, option name)

Table 3.1: API primitives for producing data: producer() creates a context, attach() connects it to the network, produce() outputs Data packets.

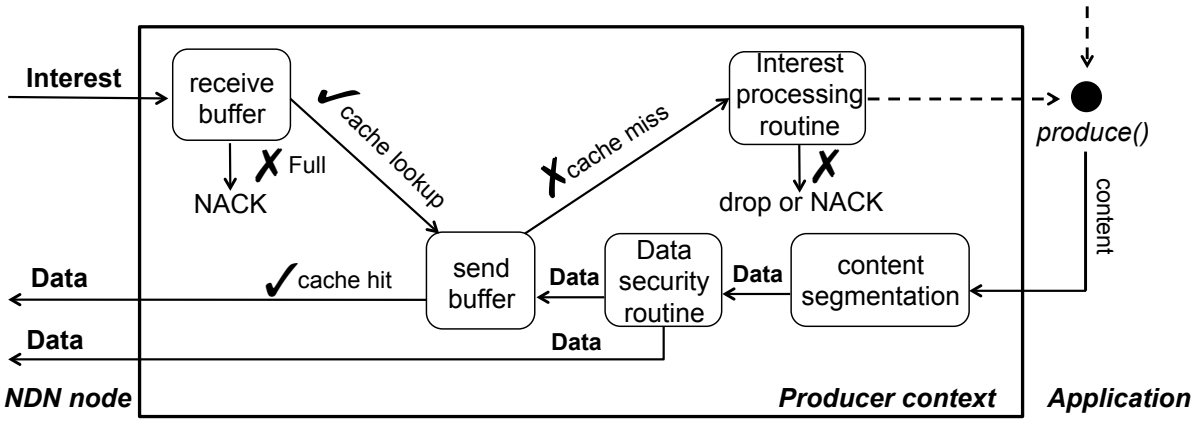

Figure 3.1: Producer context can publish data with or without network connectivity.

An application process calls **produce()** operation to start data publication, passing the name suffix and application frame (ADU) content. In the Simple-Video example, the name suffix is a frame number. In general cases, the name suffix parameter allows application developers to reuse the same producer context to publish data in any name subtree. In the Simple-Video application example, one context is used for publishing all video frames, and another context is used for publishing all audio frames. The locations of the producer contexts in the name tree are illustrated in Figure 3.2.

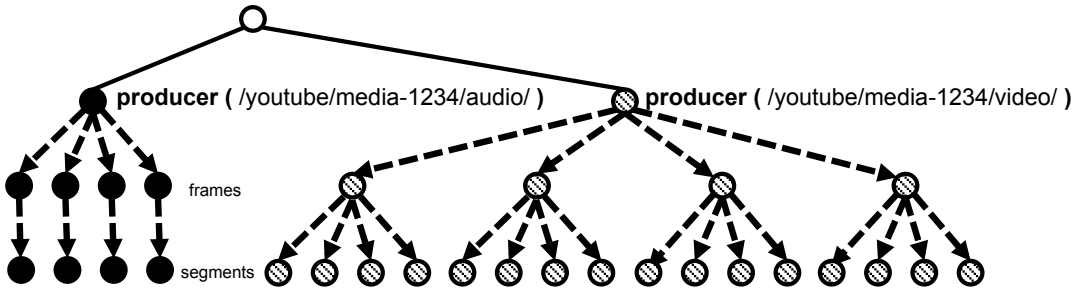

Figure 3.2: Producer context is initialized with a name prefix common for all information objects that it generates.

The **produce()** operation finishes when 1) the application frame (ADU) is

segmented into an appropriate number of Data packets, 2) the segment number is appended to each packet name, 3) each packet is secured (e.g. signed), and 4) pushed in the send buffer and out of the context (Figure 3.1). By default, the segments are temporarily stored in the send buffer — in-memory storage of Data packets, while some producer applications may want to write the resulting Data packets in a permanent storage, such as NDNFS or Repo-NG [SWD14, CSC14].

The context's send buffer is different from the socket's send buffer in two ways. First, the socket's send buffer is used to retransmit unacknowledged segments, whereas the producer context's send buffer is used as a temporary cache of Data packets that is being looked up by incoming Interest packet. In other words, send buffer softens the time asynchrony between data production and fetching. Second, in a socket, packets are evicted after being acknowledged, whereas in a producer context, Data packets are evicted based on memory availability, e.g. when the application calls **produce()** with an already full buffer under FIFO eviction policy.

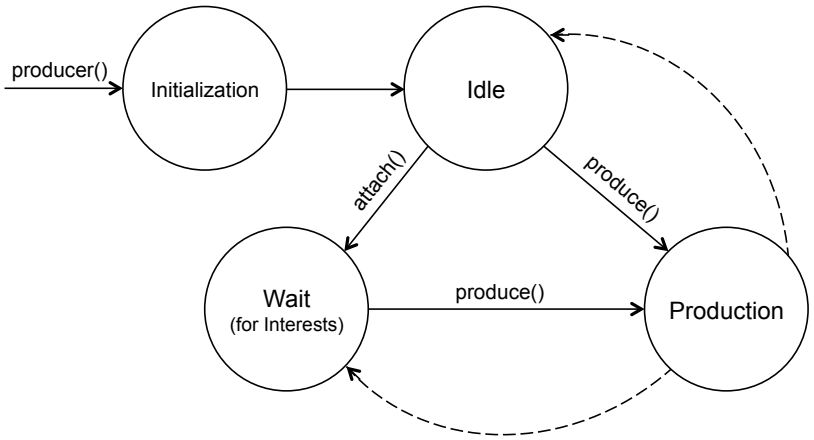

Figure 3.3: Producer context can publish data regardless of being attached to the network.

In order to receive Interests for its data, the producer context must be attached to the local NFD by calling **attach()** operation. The arriving Interests get into a receive buffer and wait there for their turn to be matched with Data packets in

the send buffer. If an Interest matches a Data packet by the name and Interest selectors successfully, the Interests is satisfied from the send buffer. If a matching Data packet is not found, an application can be informed about the Interest.

In some conditions, the rate of incoming Interest packets may be too high for a particular producer context to process as quickly as they arrive. In other conditions, the requested data cannot be generated within the Interest's lifetime span. Instead of letting the consumers timeout blindly, application can use **nack()** operation to satisfy the Interests with a negative acknowledgement (Section 3.6.1), so that the consumer(s) can handle the situation in a most informed way.

3.2.1 Context options

Primitives **getcontextopt()** and **setcontextopt()** are used to manipulate parameters of the producer context, such as:

- Size of the send buffer — internal cache of Data packets. The send buffer size has an impact on the performance of the content publishing application (Section 5.1.1).
- Size of the receive buffer for arriving Interests. Large receive buffer size may increase queuing delay at the producer side, and a small size may cause more frequent congestion events in the Producer API that results in automatic transmission of network level NACKs with congestion signal.
- Data freshness period, which affects duration of time the content stays in cache of intermediate NDN nodes.
- ADU segmentation parameters, such as Data packets size in the range of supported by NFD (e.g. [0KB .. 8KB]), signature type [RSA-SHA1, RSA-SHA256, RSA-SHA512, HMAC-SHA1, HMAC-SHA256] and KeyLocator length (in bytes).

- Forwarding strategy: BEST ROUTE, BROADCAST (Section 5.1.5)
- Permanent storage mode: local repo (Section 5.1.3) or remote repo prefix (Section 5.1.4)
- Fast signing mode using the embedded manifests (Section 5.1.2)
- Event handlers (e.g. user-provided routines, callback functions): Interest arrival, cache hit/miss events, Data signing and encryption, Data transmission, etc.

3.3 Design goals for the consumer abstraction

In identifying the design goals for the consumer abstraction, an initial assumption is made that, generally speaking, individual applications would like to organize ADU fetching according to their own priorities. Therefore the design goals are described in terms of what kinds of support that applications may desire in handling the relations between ADUs. Given we are still experimenting with this new consumer / producer API, the current sets of design goals, as stated below, may be further revised over time as we gain deeper understanding of applications' needs. The same can be said for the goals of the producer abstraction.

At this time the new consumer abstraction model should support the following application patterns.

1. Sequential fetching of ADUs, with allowance of missing any ADU in the stream if necessary. This can be used to support real time media streaming applications.
2. Parallel fetching of ADUs to speed up content transfer. This can benefit applications like web download and torrent.

3. Fetching of individual, dynamically generated ADUs, as needed by web and IoT applications.

3.4 Consumer context

A **consumer context** abstraction is a container that associates a name prefix with consumer-specific transfer parameters. Consumer context controls Interest transmission and processing of fetched Data packets. It is initialized by calling **consumer()** primitive with two parameters: 1) a name prefix, 2) a data retrieval protocol.

Note that, in general cases, the name prefix is not a complete name of the ADU. Since a given NDN namespace forms a name tree, an application developer can reuse a single consumer context repeatedly to fetch multiple ADUs under the same name prefix. In the Simple-Video application example, one can use one context to fetch all video frames, and another context to fetch all audio frames. The locations of the consumer contexts in the name tree are illustrated in Figure 3.5.

Initialization	**consumer** (name prefix, retrieval protocol) ➔ context
Primitives	**consume** (context, name suffix) **stop** (context) **delete** (context) **setcontextopt** (context, option name, value) **getcontextopt** (context, option name)

Table 3.2: API primitives for consuming data: consumer() creates a context, consume() starts data transfer, stop() terminates data transfer, delete() destroys context.

The data retrieval starts when an application calls **consume()** operation, which takes the name suffix as an input parameter. In the case of Simple-Video

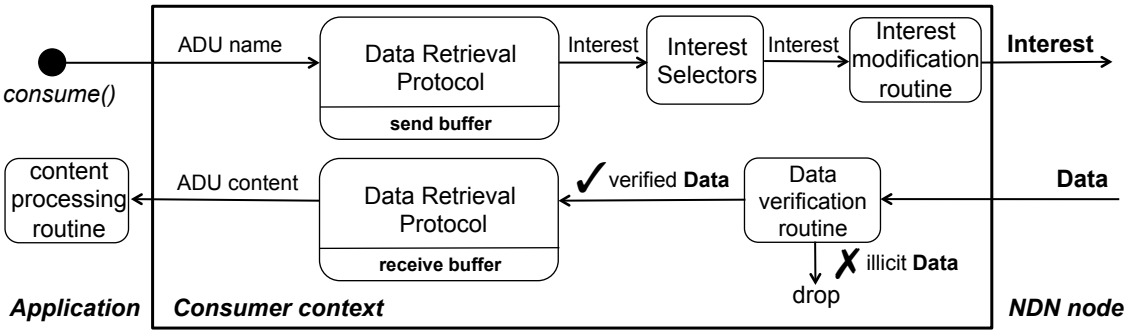

Figure 3.4: Event-based processing of Interest and Data packets in the consumer context.

application, the name suffix is a frame number. Name suffix parameter allows application developer to reuse the same context for fetching multiple ADUs (Figure 3.5). Inside the context, the data retrieval protocol (Chapter 4) generates Interests and processes incoming Data packets with other related events (Figure 3.4).

The data retrieval stops under one of the three conditions: 1) last Data packet of the ADU has been successfully fetched, validated and reassembled (if needed); 2) irrecoverable fetching error has occurred; or 3) **stop()** operation has been called.

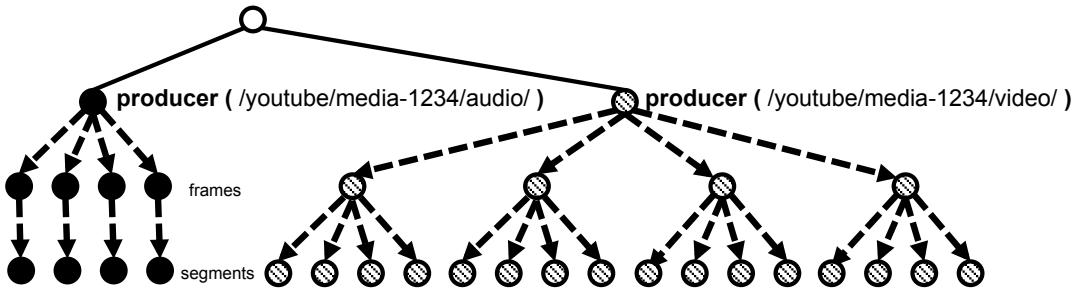

Figure 3.5: Consumer context is initialized with a name prefix defining the range of information objects that can be retrieved from the network.

3.4.1 Context options

Primitives **getcontextopt()** and **setcontextopt()** are used to manipulate parameters of the consumer context, and parameters of its underlying data retrieval protocols, such as:

- Interest selectors: exclude, child, freshness, min/max name components, etc.
- Max sliding Interest window size can enforce a hard limit on the number of Interest sent per RTT.
- Fast start threshold impacts the number of ADU segments that can be fetched after 2 round-trips. The default value is 16 Interests.
- Maximum number of Interest retransmissions for the lost packets. Default value is 16 retransmissions.
- Maximum length of an exclusion filter. Default value is 5 exclusions (e.g. packet digests).
- Interest lifetime, which by default is calculated based on RTT estimation, can be changed to a constant user-defined value.
- Forwarding strategy: BEST ROUTE, BROADCAST (Section 5.1.5)
- Event handlers (e.g. user-provided routines, callback functions): Interest modification, Interest transmission, Data arrival, Data verification, ADU reassembly, etc.

3.5 The problem of asynchrony between a consumer and a producer

The fundamental problem of the communication between consumers and producers in NDN and one of the major differences with the push IP-based synchronous

style of communication is an inherent asynchrony between consumer and producers. Consumers and producers not only are fetching and publishing application frames at a different rates, but also at a different moments of time. Production can happen well in advance of any fetching (e.g. hours, days, months). In other cases, data can be generated dynamically with some delay after the Interest arrival. Another significant problem is the asynchrony in the amount of shared knowledge about application frames between consumers and producers. Often the consumer is 'step behind' in knowing what the latest version available, the exact data name and alternatively named data packets, and other retrieval options. Sometimes the consumer has to perform an expensive content discovery phase in order to bootstrap the data retrieval of valuable content.

3.6 Negative acknowledgement

Negative acknowledgements is one of the consumer / producer coordination mechanisms. This section talks about application level and network level negative acknowledgments in more detail.

3.6.1 Application level NACK

In NDN, consumer applications pull desired Data packets from the network by expressing Interests. If an Interest does not find matching Data along the way, it arrives at the producer context, which either finds the matching Data packet from the send buffer, or otherwise informs the application to produce the requested data. The latter case happens when some specific data is being requested and produced for the first time.

Since NDN is a pull-based network protocol, it shares some common polling related challenges with HTTP [MSO14]. An HTTP client can "short poll" the HTTP server (i.e. sending regular requests) in an attempt to receive the most up-

to-date data. The HTTP server responds with empty reply in case the requested data is not ready, and the poll request will be repeated again after the client timeout. To avoid HTTP clients generating requests too frequently, which can lead to unacceptable burdens on the server and the network, HTTP long polling is commonly used. Long polling is a technique of keeping HTTP requests pending or "hanging" at the server until the requested data is ready to be sent back to the client.

Long polling works well for HTTP, because the underlying TCP connection ensures that HTTP request is reliably delivered to the server, and that the HTTP client is still waiting for the data. Since NDN network layer does not, on its own, ensure reliable transmission of an Interest all the way to the producer, and, more importantly, outstanding NDN Interests consume router resources (by occupying PIT entries), the long polling technique is not a feasible solution. In order to efficiently handle the polling of dynamically generated data in NDN, two conditions must be satisfied: 1) consumer application must be certain that its Interest packet has successfully reached the producer, and 2) producer application can regulate the polling frequency according to its current conditions.

A negative acknowledgement (NACK) can satisfy these two conditions. Application level NACK is defined as a sub-type of NDN Data packet, which is generated when the requested data is unavailable. A NACK carries an error code, a retry timer value, and other optional application-defined fields filled by the producer application. It informs the consumer that 1) the Interest for its requested data has been received by the producer, and 2) the error code contains information to advice the consumer for best next action. Currently, two error codes are defined as follows:

1. RETRY-AFTER — prompts the data retrieval protocol to schedule Interest retransmission based on the timeout value in the negative acknowledgement. This mechanism is somewhat similar to Retry-After HTTP and SIP header

field [FGM99a, RSC02]. NACK with Retry-After field does not change the Interest pipeline size.

2. NO-DATA — prompts the data retrieval protocol at the consumer side to terminate its operation.

Since NACK packet must be signed like all other Data packet, additional measures [AMM13] must be taken to prevent malicious consumers from launching a Denial-of-Service attack by forcing the producer application to generate and sign excessive amounts of NACK packets.

Since NACKs are NDN Data packets, they can be cached at intermediate NDN routers, so that the same NACK packet can be used to satisfy the Interest packets from multiple consumers requesting the same piece of data.

3.6.1.1 Estimation of NACK lifetime

A cached NACK becomes stale when its lifetime (e.g. the FreshnessPeriod field), whose value is set by the producer context, expires. As a rule of thumb, the lifetime of a NACK packet must not be longer than the retry timeout value contained in it, otherwise the consumers attempting retry after the timeout will receive the same cached NACK again, and consequently will wait for another timeout period. One must also keep in mind that a Data packet can stay at each router hop for the FreshnessPeriod before it becomes stale, and that there can be multiple router hops between a producer and its consumers. Therefore, it is proposed to set the FreshnessPeriod of a NACK to be within a small fraction (10%) of the application specified retry timer.

3.6.1.2 Estimation of Retry-After timeout

Retry-After timeout is calculated by the application, because its time scale is typically much larger than the time scale of network transmission (e.g. RTT). The application is in a good position to make accurate estimation of how much time is necessary to prepare an ADU, which potentially involves such operations as: writing or retrieving data from non-NDN database (e.g. SQL, NoSQL, etc.), expensive computations, or communication with some other applications over NDN or non-NDN networks. Network stack (e.g. producer context) has a very limited insight in the operations of the application located above the stack, and therefore it cannot estimate Retry-After timeout accurately.

Sometimes even the application has difficulties with calculating a Retry-After timeout, especially when ADU publishing involves some interaction with external components, such as databases, remote servers, etc. In such cases, two strategies do exist:

1. Overestimation of Retry-After timeout. This approach guarantees ADU existence by the time when consumer performs retransmission, but may lead to a user-perceived or real unresponsiveness of the application. However, this method can be used for several classes of applications that rely on heavily asynchronous communication model.

2. Underestimation of Retry-After timeout. This approach increases responsiveness of the application, but may lead to the situation where a consumer performs retransmission when the ADU is still not ready. In general, it is a responsibility of the application to decide whether to ignore such Interest retransmissions or issue a new application NACK with a new Retry-After timeout, which accurately estimates current conditions.

Ignoring retransmitted Interests is still relatively safe, because 1) a consumer already knows from the first NACK that the producer has received an initial Interest(s) for the ADU and is preparing a response, and 2) a consumer will perform a number of Interest retransmissions and do an exponential backoff every time. However, there is a risk that the data retrieval protocol at the consumer side will simply give up on retransmitting Interest(s) in the case if Retry-After timer was heavily underestimated.

Reissuing of a new application NACK is a much safer technique, because it allows to 'spend' consumer-side retransmissions less actively. It is a responsibility of the application to provide an updated Retry-After timeout value, and it is a responsibility of the Producer API to ensure the absence of naming conflicts among old and new application NACK[1] using a versioning model.

3.6.1.3 Versioning model

The purpose of NACK versioning model is to prevent re-usage of the previously assigned names. It is important to preserve name consistency not only during the lifecycle of the application or producer context, but also during the larger timespans, such as between producer context deletion/instantiation, application and operating system restarts. Three versioning models had been considered:

1. **Sequential**

   Sequential versioning model is a model ensuring that each new version of the content is given a unique monotonically increasing non-negative number. The advantage of this model is that the consumer can easily discover the

[1] All data is immutable in NDN. New content cannot be named the same as any existing content.

most recent available version by either issuing Interest with a rightmost-child selector or by speculative fetching of content (e.g. probing). The disadvantage of this model is a requirement to keep state (e.g. last used version number) in order to assign next version number. One can argue that a producer application can avoid the requirement of preserving the state by using the same 'latest version discovery mechanism' as a consumer application. Such a technique would add a considerable latency to the NACK publication process and is not fully reliable, which could lead to undesirable name collisions.

2. **Random**

   Random versioning model is a model ensuring that each new version of the content is given a unique random number. The advantage of this model consists in the fact that producer application does not have to keep any state about prior versions of the content. The disadvantage is that consumer application has no way of knowing the order of versions and, therefore, cannot decide what is the latest version available.

3. **Time-based**

   Time-based versioning model is a model ensuring that each new version of the content is given a unique non-monotonically increasing non-negative number. This model is used in the Producer API, because it allows consumers to know the order of versions and does not require producer context to keep any state between context deletion/instantiation, application and operating system restarts. The source of the time values is a localhost operating system time service encoded as UTC Unix Epoch time (Figure 3.6)

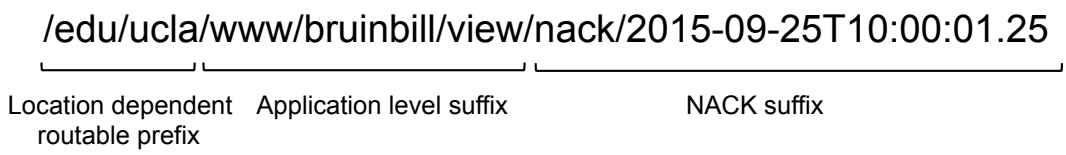

Figure 3.6: Producer context automatically appends a time-based version name component to every NACK.

3.6.2 Network level NACK

Network level NACK (Interest NACK) was not present in the original design of NDN. It was introduced later in the joint work of UCLA and University of Arizona [YAM13]. In the original design of NDN, when a router forwards an Interest it has to wait until the timer expires to learn that the interface does not work. This waiting process slows down the detection of and recovery from network errors. In addition, the unsatisfied Interest is left in the network until its lifetime expires, which will affect the correct processing of other Interests requesting the same Data.

The network level NACK (Interest NACK) was introduced to resolve these type of problems. When an NDN node cannot satisfy or forward an Interest (e.g., there is no interface available for the requested name), it sends an Interest NACK back to the downstream node. The downstream node is promptly notified of the problem by the interest NACK, and thus can take recovery actions right away. Interest NACK also cleans up the dangling states in the network. In the absence of packet loss or hijacks, every pending Interest should be consumed by either a returned Data packet or a NACK. Upon receiving a NACK, a node may return it further downstream if it has exhausted all forwarding options.

An Interest NACK carries the same name and nonce as the corresponding Interest, plus an error code explaining the reason why the NACK is generated so that actions can be taken accordingly by the downstream node. The following list

provides a few common reasons. New code may be added in the future as we gain more experience with the system.

- *Duplicate*: The Interest is a duplicate of a recently forwarded Interest. This happens when the upstream node detects a looped Interest.
- *Congestion*: The one-to-one flow balance between Interest and Data packets gives NDN an effective way to prevent congestion inside networks. By pacing Interests sent to the upstream direction (towards producer) of a link, one can prevent congestion (caused by Data) on the downstream direction of the link.
- *No Prefix*: If an upstream node has no path to forward the Interest, it is beneficial to infrom downstream node to stop sending future Interests under the same name prefix. This may happen when N does not want to provide service for the named data, e.g., due to its own policy.

In the absence of packet losses, every pending Interest is consumed by either a returned Data packet or a NACK. Returning NACKs brings two benefits to the system: it cleans up the pending Interest state much faster than waiting for timeout, and it allows the downstream nodes to learn the specific cause of a NACK, so that they can take informed recovery actions. Note that an Interest NACK is different from an ICMP message; the former goes to the previous hop while the latter is sent to the source host, hence their effects are entirely different.

3.6.2.1 Network level NACK usage in Consumer/Producer API

Due to the absence of receiver window advertisement mechanism, a producer context can become congested if Interest arrival rate is higher than the Interest satisfaction rate over a period of time. Figure 3.1 helps to see that incoming Interests are queued in the receive buffer awaiting processing, which fills up in the congestion scenario.

What actions must be taken when a producer context is in a congested or near-congested state? Since an application only works with the Interests that have been dequeued and not matched to Data in the send buffer, it is the ultimate responsibility of the producer context to drop excessive Interests and reply with congestion NACKs.

This thesis does not offer a recommendation regarding which particular queuing discipline is the best option for the producer context. Tail-drop, classic Random Early Detection (RED) [FJ93], Weighted RED, Blue [FKS99], PIE [PNP13], or any other AQM techniques could be considered as a single-queue disciplines.

A multi-queue design of the receive buffer seems to be a promising approach to the problem. Since NDN names are hierarchical, it appears to be feasible to distribute incoming Interest packets on either a per-ADU basis, or per-prefix basis. For example, queuing Interests on a per-ADU basis would allow to avoid dropping Interests for the ADUs that are already being processed, which can potentially result in a faster flow completion. Per-ADU queueing discipline can also take input in a form of application NACKs generated by the application directly that could trigger a removal of all Interests in that particular per-ADU queue.

In the Consumer API, network level NACK benefits data retrieval protocols that do not have to accept losses as signals of congestion, but rely on explicit congestion NACK signals instead. In other words, the processing of congestion NACK is somewhat similar to the processing of ECN (Explicit Congestion Notification) in ECN-enabled TCP/IP protocol suite.

3.7 Manifest

A well-built NDN application fully utilizes "many-to-many with caching in-between" communication paradigm. To keep consumers best informed of the production progress of the data that they are interested in fetching, a producer application

may package together necessary meta-information to distribute to consumers.

Manifest, proposed in [BDN12], is one of the means to facilitate the operation of consumer applications by distributing a catalogue.

The catalogue may contain either ordinary NDN names, or special names which associate the hash (e.g. digest) of a Data packet with its name. The primary benefit of using catalogues to carry data names with associated packet digests is the elimination of cryptographic signing operations for those Data packets. Instead of signing, a publisher computes a simple hash of every newly produced Data packet, populates the manifest with names carrying digests, and signs only the manifest. Consumer applications can verify Data packets by fetching the manifest and comparing their digest with the digest listed in the catalogue. As proposed in [BDN12], manifests carrying the catalogue of names need to be fetched before data fetching, which introduces an additional round-trip latency.

This thesis proposes to embed a manifest as an ADU in the same sequence with Data packets to eliminate the undesirable latency from fetching manifest [I 14]. A producer context can perform this operation when an ADU is segmented by the **produce()** API primitive. The basic idea is to establish a convention of naming the manifest as the first segment of the Data packets to be published, so that consumers simply fetch the manifest together with data via Interest pipelining. In case where an ADU's size is too large so that the names of all its segments cannot fit into a single manifest packet, multiple manifest packets can be periodically interleaved with data packets as shown in Figure 3.7.

Manifest embedding enables the consumer application an opportunity to fetch manifests together with Data packets within the same sliding Interest window.[2] By letting the KeyLocator field in each Data packet point to the corresponding embedded manifest, a consumer application is able to verify each received Data

[2]Sliding Interest window includes already sent not-yet-satisfied Interests, as well as the Interests scheduled for transmission at the moment of time.

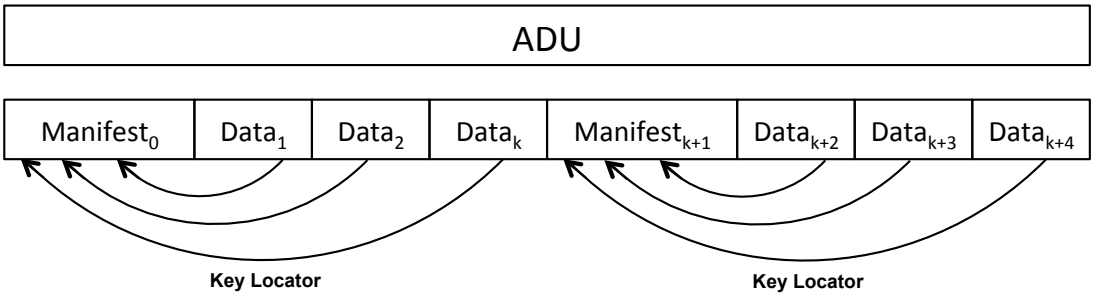

Figure 3.7: Manifests are embedded in the sequence of data packets when application data is being segmented.

packets immediately without waiting for the rest of the Data packets.

A manifest is realized as a sub-type of NDN Data packet. In addition to the catalogue of names, manifest can also carry miscellaneous meta-information in a form of key-value pairs, such as:

- **Current data production rate.** Live streaming applications can benefit from knowing the current rate of Data packet production (packaging) and using this knowledge to pace Interest packets.
- **Other available versions.** Applications working with multi-version content can discover available versions of ADUs without iterative discovery using Interest selectors which can be time consuming.
- **First and Last ADU sibling**. In most cases, the producer of the ADU knows the total number of ADUs that constitute some larger information object (e.g. a video stream). Our Simple-Video application uses the last ADU name to understand where the video ends (e.g. frame #2500).

This is a preliminary description of a manifest abstraction, which is being actively researched by the community these days. It is important to note that (1) the manifest described above is created by the publisher of the original content,

and (2) manifest embedding is not the only viable way of publishing manifests by the original content producer.

There are multiple ways of organizing data and manifests, which are based on the idea of manifests pointing to other manifests. For example, one can easily imagine a way to construct a tree of manifests. Such tree can either match the relationships of the underlying user content objects or create an alternative hierarchy (i.e. index). A unidirectional tree would allow a 'manifest traversal robot' at the consumer side to go down from the root manifest towards the leaf manifests, which could be just enough for many search algorithms. A bidirectional tree requires child manifests to point to the parent manifests, but in turn allows to traverse the manifest tree starting from any manifest (not only the root). Tree is not the only useful data structure for manifests, for example a graph structure of manifests might also have good properties for some applications.

Manifest can be published not only by the original content producers, but also by the third-party applications or system / network services. Third-party applications may include various content aggregators that re-mix and embed the original content from various original publishers. System / network services may include various lookup and directory services that assist application with discovering devices, content and services.

In many of these cases, a 'manifest traversal robot' at the consumer side is likely to prefer to know the type of content the manifest entries refer to. To achieve that goal the definition of a manifest would need to be extended to support not only the standard NDN names, but also NDN names with attached metadata (e.g. PublisherKeyID), as well as so called NDN Link Objects that bind two names (e.g. location independent & location dependent) together using a cryptographic signature [AYW15].

CHAPTER 4

Data Retrieval Protocols

Based on our experience with building NDN applications, the initial set of data retrieval protocols includes: Simple Data Retrieval (SDR), Unreliable Data Retrieval (UDR), and Reliable Data Retrieval (RDR).

4.1 Simple Data Retrieval

Any communication in an NDN network involves Interest / Data exchanges, and Simple Data Retrieval protocol (SDR) is the simplest form of fetching Data from NDN networks: send one Interest to retrieveone Data packet. SDR provides no guarantee of Interest or Data delivery. If SDR cannot verify an incoming Data packet, the packet is dropped.

SDR can be used by the applications that:

- do not know the name of the application frame (ADU) and, therefore, need to discover it using the name prefix and Interest selectors, which could be set via the **setcontextoption()** API primitive;
- know the name of ADUs and have small ADUs that fit in one Data packet;
- want to directly control Interest transmission and error corrections.

4.2 Unreliable Data Retrieval

When an Application Data Unit (ADU) is too large to fit in a single Data packet, the **produce()** API primitive automatically segments this ADU into an appropriate number of Data packets. In this case, the consumer first needs to send a sequence of Interest packets to fetch all the data packets of the same ADU, then it needs to reassemble these Data packets into the ADU, which often implies dealing with packet losses and error corrections, as well as packet ordering.

UDR is designed to meet the needs of applications that have relaxed requirements for the reliability and ordering of the Data packets, and are unwilling to pay the price in the latency of loss recovery, or in the performance overhead associated with other means of reliable delivery. UDR fetches all Data packets that belong to a single ADU in an unreliable and unordered way, with a simple flow control and best-effort Interest retransmission as explained below.

UDR makes use of the FinalBlockID, one of the optional fields carried in an NDN data packet, by having the producer set the FinalBlockID to the number of segments in an ADU. UDR fetches the ADU of a given name by starting with the segment number zero, and learns about the total number of segments to be fetched as soon as any Data packet is received. Next, the protocol enters the fast start phase and sends as many Interests as *MIN (FinalBlockID, Fast start threshold)*.[1] If the value of FinalBlockID is greater than the fast start threshold value, UDR completes fast start phase and begins to multiplicatively increase sliding Interest window size in a way similar to the TCP slow start phase. If any Interest times out during the multiplicative increase phase, the sliding window size is reduced by half. To get the basic intuition behind this flow control scheme, consider a common use case where the ADU consists of a small number (≤ 15) of Data packets: UDR can fetch such small ADUs in two RTTs and avoid bursty

[1]The default fast start threshold is 16 Interest packets, which could be modified via setcontextoption() API call.

transmission for much larger ADUs (e.g. hundreds of Data packets).

UDR's best-effort Interest retransmission works in the following way: at any given time, if three out-of-order Data packets arrive at the consumer, UDR immediately retransmits the Interest for the missing Data packet(s).[2] UDR can perform multiple fast retransmissions per sliding Interest window by keeping an accurate track of missing and contiguous segment numbers.

UDR does not perform any persistent error correction; it does not run retransmission timers, nor retransmits Interests upon receiving NACKs, which are passed up to the application. UDR drops Data packets that fail data verification. UDR delivers each received Data packets to applications as soon as possible without enforcing ordering, thus applications handle received packets directly and are responsible for the ADU reassembly. This also offers an opportunity for the applications to perform specially tailored error and loss recovery.

In summary, UDR functionality includes best effort fetching of single- and multi-segment application data frames (ADUs), and best effort fast retransmission for potentially lost segments. "Deadline-oriented" consumer applications (e.g. live streaming) can benefit from using UDR's machinery and extending it with the custom functionalities appropriate at the application level.

4.3 Reliable Data Retrieval

When an Interest packet fails to bring back the corresponding Data packet, it can be due to one of the multiple reasons:

1. the Interest is lost in transit before it reaches the data, which may reside in cache, or need to be produced;

[2]The current implementation of NFD will forward a retransmitted Interest even if the original Interest has not expired, if the retransmission arrives from the same face and is at least 100ms after the original Interest.

2. the Interest reaches the producer-application but the application does not respond due to various reasons;
3. the returning Data packet is lost;
4. the returning Data packet fails the signature validation (e.g. content is poisoned, etc.).

Reliable Data Retrieval protocol (RDR) uses Interest retransmission timers to handle packet losses (cases 1 & 3 in the above), and uses application negative acknowledgements to handle case 2. The Interest is retransmitted if the expressed Interest packet is not satisfied when it times out, or if the negative acknowledgment carrying Retry-After field is retrieved instead of the actual data.

Data verification error can be caused by packet tampering, content poisoning by a non-credible publisher, expired certificate of a credible publisher, or other cases depending on the selected trust model. Several in-network mechanisms of mitigating content poisoning attacks have been proposed by [CGT13], and [GTU14] describing the content ranking algorithms based on the users' feedback.

While the Data verification operation is performed separately by the security part of the library, Data retrieval protocol makes an attempt to recover from this type of error. To recover from the Data verification failure, RDR performs retransmission of the Interest packet with exclude selector set to exclude any possible Data packet having the same name and the digest (e.g. hash, checksum) of the packet that has failed verification. Because exclude selector tells NDN router to retrieve an alternative Data packet, which in general case requires an extra work to be performed by the router, large excludes (e.g. containing a lot of excluded name components or digests) can affect the performance of NDN router. RDR limits its exclude selector to five digests, which means that the protocol attempts up to five retransmissions in order to recover from the Data verification

failure.

RDR provides reliable and ordered delivery of the ADU to the consumer application. Unlike TCP, RDR does not attempt to establish a connection between the consumer and producer applications. In RDR, the retrieval of every ADU begins with sending an Interest packet for segment number zero, and is finished when the last segment is successfully retrieved. Similar to UDR, the producer sets the FinalBlockID field in each Data packet to the last segment number. RDR's flow control has the same fast start and multiplicative increase phases as UDR does.

Figure 4.1 illustrates an example where the consumer application uses RDR to retrieve dynamically generated data and handle verification errors. The first Interest is satisfied by poisoned content from the router cache, which is returned back to the consumer context. The RDR checks the content with the user-specified verification routine, and retransmits the Interest(s) with the exclude selector carrying the digest of the poisoned content. Since the routers respect the exclude selectors, this second Interest reaches the producer context, which needs some time (e.g. several seconds) to prepare the content, and therefore replies with the Retry-After NACK. This Retry-After NACK packet has /nack in its name suffix, therefore it has no impact on the poisoned content in the cache. When the consumer RDR receives the Retry-After NACK, it schedules the Interest retransmission accordingly, which later successfully retrieves the content from the producer context. Now the router has two Data packets with identical names but different digests, they can be either stored side by side or replace one another depending on the router Content Store policies.

Producer applications can mitigate excessively high rate of Interest arrivals by responding with negative acknowledgements carrying either Retry-After or No-Data fields, depending on the data being asked. RDR's flow control utilizes these NACKs as discussed in Section 3.6.1. The traditional mechanism of TCP window size advertisement for flow control purpose is not applicable in NDN, given the

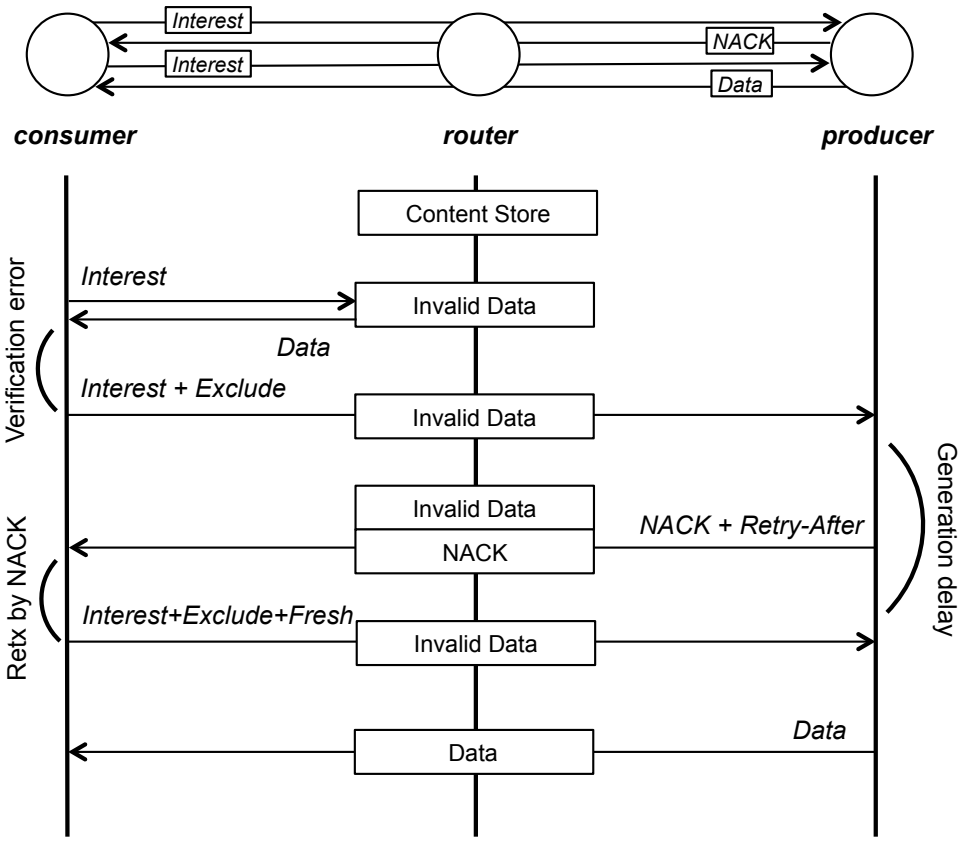

Figure 4.1: RDR recovers from the Data verification error and handles dynamic data generation delay.

absence of a "connection" between consumers and producers.

Congestion is controlled at the NDN forwarding plane by utilizing network level NACK mechanism as discussed in Section 3.6.2. The NFD running on the local node is expected to handle network level NACK, perform congestion control, and enforce fairness among multiple applications running on the same node.

If three out-of-order Data packets arrive at the consumer, RDR performs opportunistic fast retransmission of the Interest for the missing Data packet, in the same way as UDR.

In the presence of the manifests embedded in the sequence of Data packets (Section 3.7), RDR performs verification of Data packets with help of catalogues of names in the corresponding manifest segments. If any Data packet fails its

verification with the catalogue, RDR retransmits the Interest packet with the implicit digest. Since the correct digest is already known from the manifest, there is no need to use exclude selector in this case.

If a sequence of Data packets does not contain embedded manifests with catalogues of names, RDR verifies each packet's signature independently, and performs error correction using the exclude selector as described earlier.

In summary, RDR functionality includes:

- reliable fetching of a single- or multi-segment application frame (ADU) that may be either pre-generated ahead of time by the producer application and potentially cached by NDN routers, or dynamically generated upon an Interest arrival;
- low overhead consumption of dynamically generated application frame (ADU) through the use of NACK packets published by the producer application; and
- persistent recovery from the Data verification failures.

CHAPTER 5

Use cases and communication patterns

This section describes the most fundamental NDN communication patterns available through the Consumer / Producer API framework and introduces the design of the few pilot applications built on top of the API.

5.1 Common communication patterns

NDN specific communication properties such as built-in transient and permanent storage, pull-based data retrieval, multi-path forwarding and multicast content dissemination provide the foundation for some distinct communication patterns. The following sections describe the ways how applications can exploit these patterns using the Consumer / Producer primitives.

5.1.1 Realtime publishing & consumption

A pattern with realtime ADU publishing and consumption is used by a large number of applications, such as video conferencing, games, etc. (Figure 5.1). This pattern can be characterized with two different aspects:

- if the producer is not able to pre-generate the content before it is requested, the consumer(s) are waiting for the data, since the on-demand content generation takes additional time due to Interest processing, content segmentation and signing, etc.

- if the producer is able to pre-generate the content before it is requested by the consumer(s), the producer is waiting for pull, which requires keeping the ADUs in memory for some period of time to handle a possible mismatch between production and consumption rates and timing.

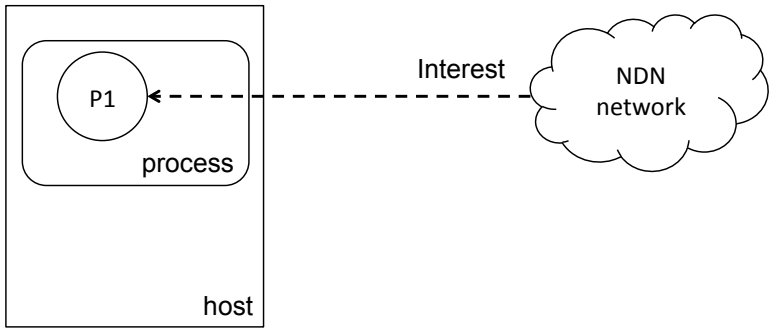

Figure 5.1: Realtime publishing & consumption states: a) Wait for pull b) Wait for data

Unlike the socket's send buffer, which stores the segments that belong to a particular connection, producer's send buffer stores Data packets that share some common name prefix. In other words, different ADUs are temporarily stored in the same send buffer. This also means that Interests for different ADUs might potentially go to the same producer context (e.g. the same send buffer). These two factors create an interesting dynamic between the ADUs that are currently stored in the send buffer and the ADUs that have been evicted from the buffer and, therefore, have to be republished (e.g. signed) again.

The publisher-process is exposed to different computational costs depending on the size of the send buffer. The publishing cost is lower if the send buffer is large, allowing more ADUs to be kept in memory, and is higher if the send buffer is small, which causes more "ADU republishing events".

The experiment modeled the behavior of:

1. a basic web server publishing 20 ADUs (50 KB each) such as personalized html pages and other dynamic content. Each ADU consists of 30 Data

packets. The send buffer sizes that had been tested are [0, 25, 50, 100, 200, 400, 800] of Data packets.

2. multiple basic web consumers requesting 20 personalized web resources (ADUs) in random order. The retrieval of the random ADU can potentially cause the "ADU republishing event" depending on the contents and the size of the producer's send buffer.

The third factor that affects space-computation tradeoff is the Interest pipeline size at the consumer when it tries to fetch any particular ADU. Interests sent within a larger window fetch more Data packets at once, leading to fewer "ADU republishing events".

Figure 5.2 demonstrates that without a send buffer, multiple consumers, each pipelining many Interests, are able to overload the producer due to signing and segmentation overhead. A larger buffer effectively amortizes these same costs across multiple consumers.

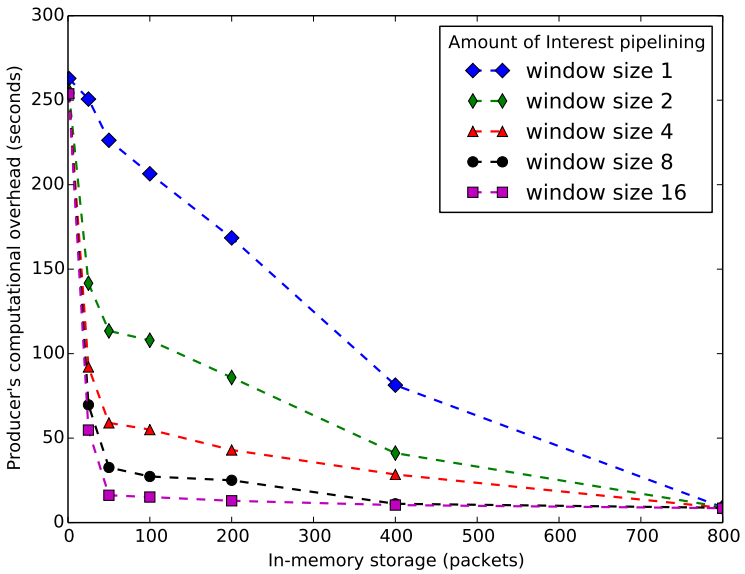

Figure 5.2: Benefit of having producer's send buffer for serving multi-packet ADUs to multiple consumers that pipeline Interests.

5.1.2 Fast signing & verification

Producer API has an additional mechanism for improving the speed of ADU publishing — manifest embedding in the sequence of Data packets (Figure 5.3). This technique relies on the fact that many applications reassemble ADU from multiple Data packets, and therefore, do not have a strong need for independent verification of each individual Data packet.

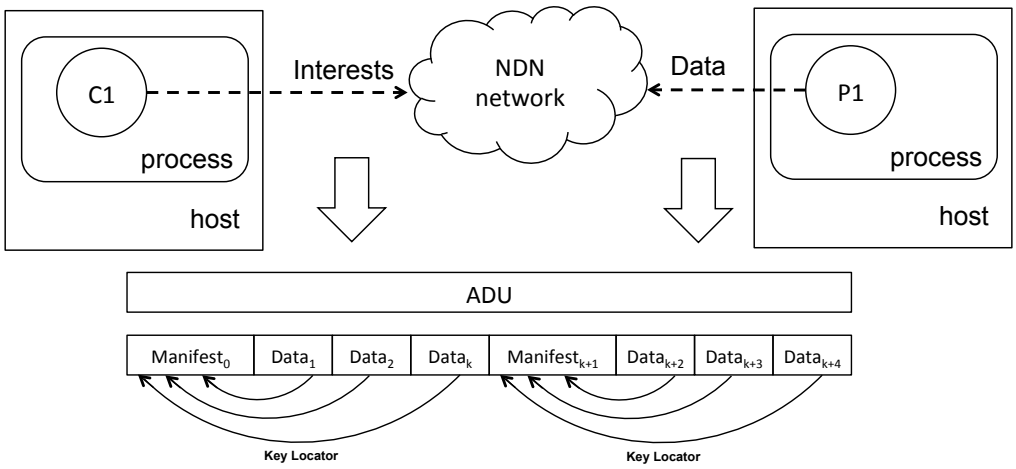

Figure 5.3: Producer amortizes the signing cost by embedding manifests in the ADU.

Algorithm 1 Publishing with fast signing

1: $h \leftarrow$ **producer**("/edu/ucla/video-1234")

2: **setcontextopt**(h, **FAST_SIGNING**, *true*)

3: **produce**(h,"/frame1", video frame payload)

The same experiment with a single Web server and multiple Web consumers had been performed to understand the performance implications of embedding manifests in the sequence of Data packets. This technique demonstrated up to 32 times increase of the speed of ADU publishing as illustrated by the Figure 5.4. Both experiments were conducted on the Mac OS X platform with trusted plat-

form module (TPM)[1] used to produce RSA signatures with SHA256 digest.

Consumers running RDR protocol do not need to take any special action to enable fast verification since the protocol discovers this pattern automatically. A special option must be set in the producer API to activate fast signing (Algorithm 1).

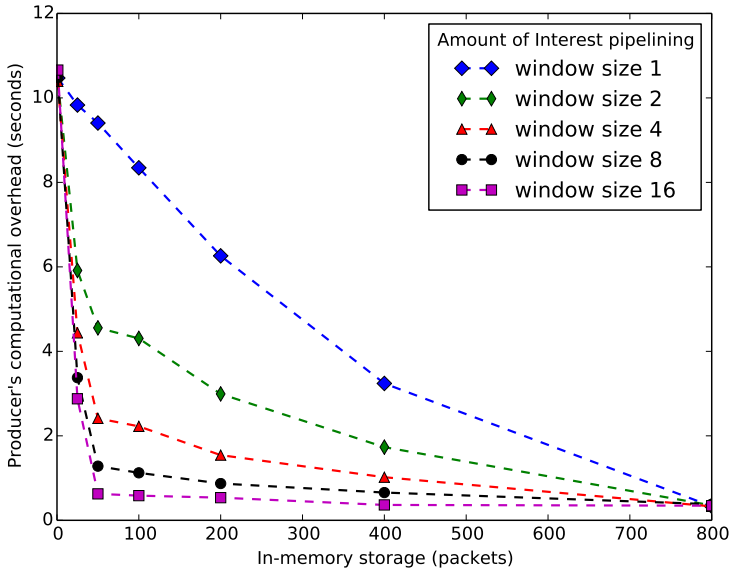

Figure 5.4: Benefit of embedding manifests in the multi-packet ADUs fetched by multiple consumers that pipeline Interests.

5.1.3 Largely asynchronous publishing & consumption

A pattern with largely asynchronous ADU publishing and consumption usually requires the permanent storage of Data packets, because the lifetime of the content can be much longer then the uptime of the process that publishes the content. The pattern "Publish once - consume multiple time" is beneficial for static content servers, such as video and web-content backend applications.

[1]Mac OS X Keychain

When the lifetime of the data is much longer than the lifetime of the publisher-process, the data can be published into a Repo (Figure 5.5). To publish data into the local Repo, the producer context does not have to be attached to the NFD as shown in Algorithm 2.

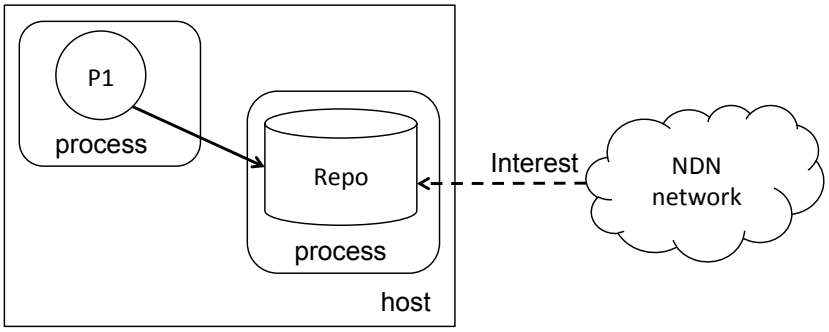

Figure 5.5: Largely asynchronous publishing & consumption through local permanent storage (Repo)

Algorithm 2 Publishing into the local repository (Repo)

1: $h \leftarrow$ **producer**("/com/youtube/video-1234")
2: **setcontextopt**(h, **LOCAL_REPO**, *true*)
3: **produce**(h,"/frame1", video frame payload)
4: **delete**(h)

5.1.4 Mobile asynchronous publishing

In general, the publisher's mobility has no impact on forwarding Data packets back to the consumers, because NDN's stateful forwarding plane keeps the state of data requests at each network hop. However, when the producer is moving, routing updates must be made in order to keep the content reachable for the consumers.

One way to avoid costly routing updates is to publish the content to some location-static permanent storage, such as a remote Repo, and route Interest

packets to this Repo instead of the mobile publisher (Figure 5.6). The "publish and forget" communication pattern is also beneficial for the low-capability publishers, such as IoT publishers, since they cannot afford to keep the content due to memory and energy constraints.

Publishing to the remote Repo can be easily enabled with the following code snippet (Algorithm 3).

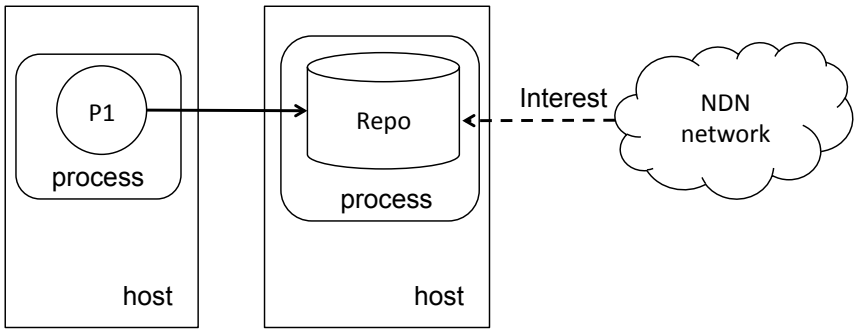

Figure 5.6: Remote permanent storage

Algorithm 3 Publishing to the remote repository

1: $h \leftarrow$ **producer**("/health-monitor-app/")
2: **setcontextopt**(h, **REMOTE_REPO_PREFIX**, */edu/ucla/hospital/ilya*)
3: **attach**(h)
4: **produce**(h,"/health-report/06-10-2015", payload)
5: **delete**(h)

5.1.5 Localhost broadcasting

Some categories of applications may require an event (e.g. Interest) delivery to all interested parties inside the node. In such cases, an IP-network application would send a UDP datagram to the broadcast address (e.g. 255.255.255.0) to the pre-agreed port number. An NDN-network application can achieve the same goals by setting the broadcast forwarding strategy for a specific name prefix, because NDN Forwarding Daemon (NFD) serves as a multiplexer between applications and

network interfaces inside a node. The broadcast strategy forwards every Interest to all upstreams, indicated by the supplied FIB entry (Figure 5.7).

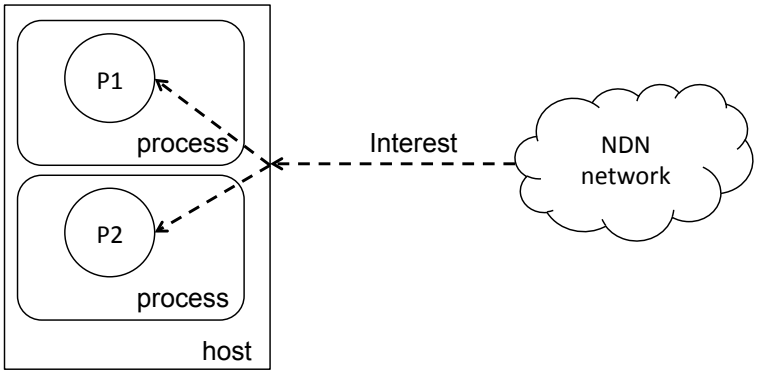

Figure 5.7: Broadcasting

The broadcast strategy can be set from both producer and consumer contexts with the following code (Algorithm 4 and 5)

Algorithm 4 Setting a broadcast strategy from the producer

1: $h \leftarrow$ **producer**("/localhost/service/eventsink")
2: **setcontextopt**(h, **FORWARDING_STRATEGY**, *BROADCAST*)
3: **attach**(h)

Algorithm 5 Setting a broadcast strategy from the consumer

1: $h \leftarrow$ **consumer**("/localhost/service/eventsink", SDR)
2: **setcontextopt**(h, **FORWARDING_STRATEGY**, *BROADCAST*)
3: **consume**(h, "/restart")

Broadcast forwarding strategy is not the only strategy available in NFD, and all other forwarding strategies (e.g. BEST_ROUTE, etc.) can be activated in the same way as shown above. The forwarding strategy in NFD is a decision maker, deciding whether, when, and where to forward the Interests [ASZ14]. The main motivation for having multiple strategies is that our experience with NDN application showed that there a single fixed strategy cannot fit the needs for all

applications. For example, some applications may require to multicast Interests to all available Faces to retrieve any matching copy of the Data as soon as possible, while the other may want to retrieve Data only from locations pointed by the routing system.

To provide the maximum flexibility, NFD allows per-namespace selection of the specific strategy [ASZ14]. This per-namespace strategy choice is recorded in StrategyChoice table, which is consulted in the forwarding pipelines when decision about Interest forwarding needs to be made. In addition to the Interest forwarding decision points, strategy can also receive notifications when the forwarded Interests are getting satisfied or timed out. Therefore, strategy presents a closed loop subsystem in NFD to control Interest forwarding. Conceptually, a strategy can be considered a program, which is written for an abstract machine and determines how to forward Interests. All current NFD strategies are written in C++ and are built-in into the NFD binary. However, future releases of NFD may allow custom strategies to be loaded at runtime and/or written in a scripting language against the strategy API abstract machine.

5.1.6 Sequential fetching

With NDN Consumer / Producer model, a TCP-like stream semantics can be achieved by fetching ADUs sequentially (Figure 5.8) using the single consumer context by calling multiple *consume()* operations with different name suffixes as illustrated in the Algorithm 6.

Note that *consume()* operation takes at least one round trip time to complete the retrieval of a single packet ADU and more round trips for a multi-packet ADU. Depending on the network round trip time and ADU size, it might be impossible to achieve high goodput with a single consumer context running in a single thread. In such cases, it is recommended to the application developer to

switch to the parallel ADU fetching pattern.

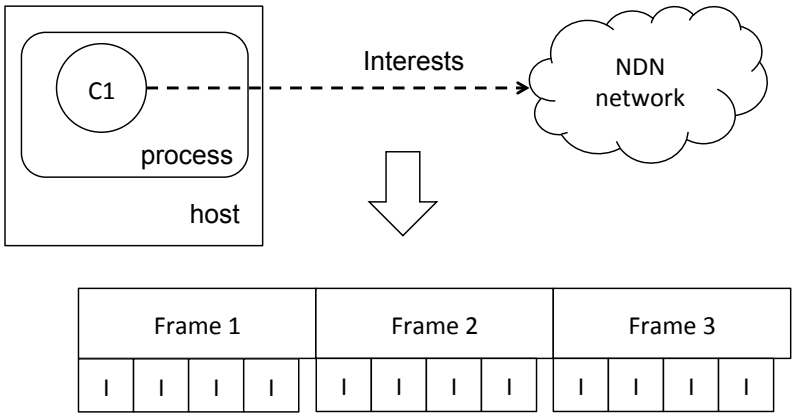

Figure 5.8: Sequential fetching of ADUs

Algorithm 6 Sequential fetching of ADUs with a single consumer

1: $h \leftarrow$ **consumer**("/cam01/video/", RDR)
2: **while** *NOT End-Of-Stream* **do**
3: $\quad Name\ suffix \leftarrow$ ADU number
4: $\quad$ **consume**(h, $Name\ suffix$)
5: **end while**

5.1.7 Parallel fetching

Parallel fetching of ADUs is used to speed up content transfer. This pattern is beneficial to web, torrent, video and many other low latency and high throughput applications. Since ADUs are independent from each other, they can be transferred and processed at the same time, which makes the parallel ADU fetching pattern one of the key communication patterns for NDN applications (Figure 5.9).

In its current implementation, Consumer / Producer API offers two ways of parallel ADU fetching:

- in a single thread with multiple consumer contexs using asynchronous function call *asyncConsume()*

- in multiple threads with multiple consumer contexts using the standard *consume()* function (Algorithm 7).

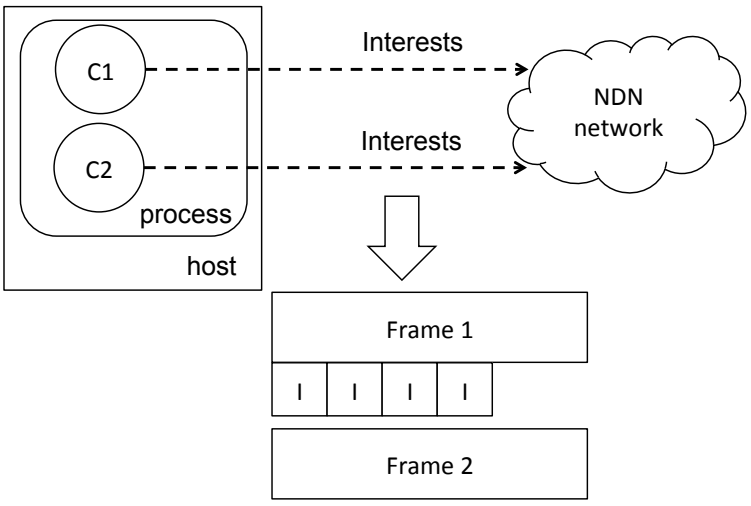

Figure 5.9: Parallel fetching with multiple consumer contexts

Algorithm 7 Parallel fetching of ADUs with two consumers

1: $h_1 \leftarrow$ **consumer**("/cam01/video/", RDR)
2: $h_2 \leftarrow$ **consumer**("/cam01/video/", RDR)

Thread 1

3: **while** *NOT End-Of-Stream* **do**
4: $Name\ suffix \leftarrow$ odd ADU number
5: **consume**(h_1, $Name\ suffix$)
6: **end while**

Thread 2

7: **while** *NOT End-Of-Stream* **do**
8: $Name\ suffix \leftarrow$ even ADU number
9: **consume**(h_2, $Name\ suffix$)
10: **end while**

5.2 Applications

A few pilot applications were built by the developers (e.g. undergraduate, graduate and visiting students at UCLA) that do not have deep expertise in computer networking in general and in Named Data Networking in particular. This section reports on their experiences using Consumer / Producer API framework in real environments.

5.2.1 NDNlive

NDNlive is capable of streaming live video captured by the camera and handling network problems by dropping individual video or audio frames. Figure 5.10 illustrates the architecture of NDNlive, which consists of two major components:

- *Publisher*

  NDNlive is a *live streaming* application; the publisher captures video from the camera and audio from the microphone and passes it to the Gstreamer to encode the raw media data and extract individual video and audio frames. The video and audio frames are published to NDN network with via multiple producer contexts.

- *Player*

  The video player uses the RDR protocol (UDR is a viable option as well) of Consumer API protocol suite to generate Interest packets for specific video and audio frames, which are later passed to the Gstreamer for decoding purposes. The player application is responsible for timing the consumption of individual frames, i.e. pacing of *consume()* calls.

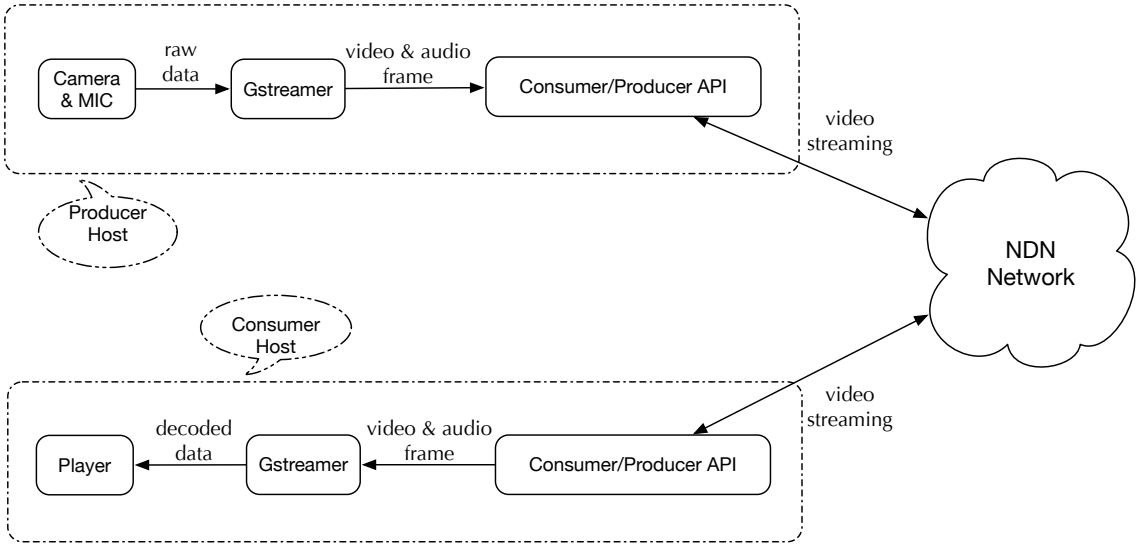

Figure 5.10: NDNlive architecture

The following name is a typical name of the Data packet that is a part of a video frame.

"/ndn/ucla/NDNlive/stream-1/video/content/8/%00%00"

- **Routable Prefix:** "/ndn/ucla/NDNlive" is the routable prefix used by NFD forwarders to direct Interest packets towards the NDNlive publisher.
- **Stream_ID:** "/stream-1" is a stream identifier used to distinguish among live streams. Note that stream ID could be a part of the routable prefix.
- **Video or Audio:** "/video" is a markup component to distinguish between video and audio streams.
- **Content or Stream_Info:** All frames go under "/content" prefix, and all stream information goes under "/stream_info" prefix.
- **Frame number:** "/8" is frame number, which is used to identify each individual video and audio frame.

- **Segment number:** "%00%00" is the segment number required to identify each individual Data packet, because most video frames are too large to fit in a single Data packet, and have to be broken into multiple Data packets.

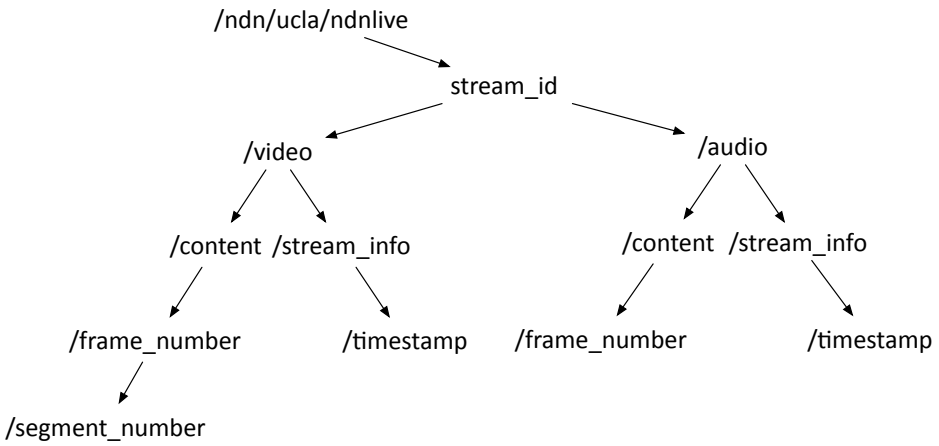

Figure 5.11: NDNlive namespace

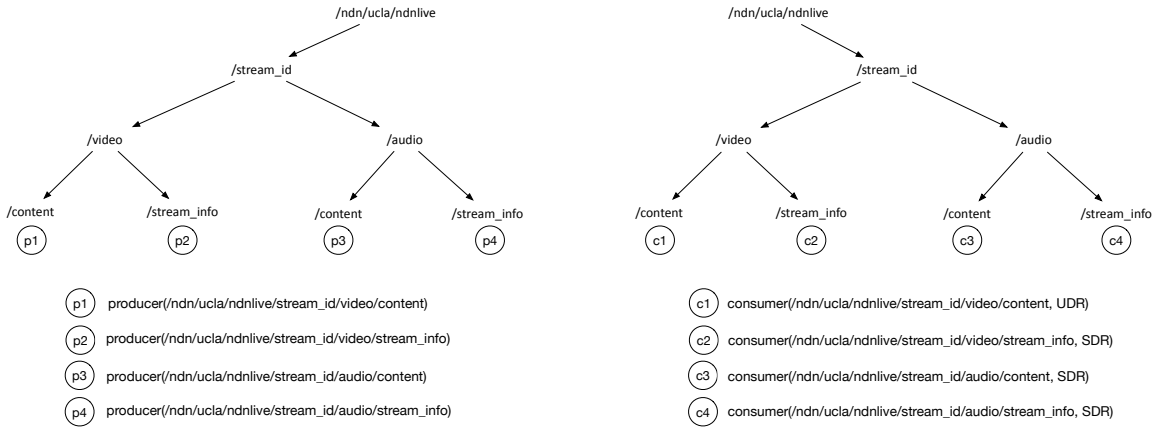

Figure 5.12: Locations of producers and consumers in the NDNlive namespace

5.2.1.1 Publisher

Publisher's application has four producers: video content producer p_1, video stream information producer p_2, audio content producer p_3 and audio stream

information producer p_4. Figure 5.12 shows the locations of the producers in the NDNlive namespace.

Two content_producers (p_1 and p_3) continuously publish video and audio frames by incrementally increasing the corresponding frame numbers (Figure 5.11).

Two stream_info producers (p_2 and p_4) continuously publish up-to-date information about the live streaming media: current frame number, frame rate, video width and height, encoding format (Figure 5.11).

Negative Acknowledgement

In some situations, the live stream publisher is not able to satisfy Interests with actual data (e.g. video and audio frames).

1. The consumer may sometimes miscalculate the pacing of video and audio frames and request the frame that does not exist at the moment (e.g. ahead of the production). The publisher can inform the consumer about this situation using *nack()* function of the Consumer / Producer API.

   As illustrated by the Algorithm 8, producer calls *nack()* function with *PRODUCER_DELAY* header containing the anticipated time value after which the data may become available.

2. Consumers join the live stream after the publisher. Since the publisher of the live stream stores only a limited number of the most recently produced audio and video frames, some consumers might request the frames that has already expired everywhere in the network. In this case, the publisher calls the *nack()* operation with textitNO-DATA header, which informs consumers that a particular video or audio frame is no longer available.

The operation of NDNlive publisher is shown in Algorithm 8.

Algorithm 8 NDNlive producer

```
1: h_v ← producer(/ndn/ucla/NDNlive/stream-1/video/
2:  content)
3: setcontextopt(h_v, cache_miss, ProcessInterest)
4: attach(h_v)
5: while TRUE do
6:     Name suffix_v ← video frame number
7:     content_v ← video frame captured from camera
8:     produce(h_v, Name suffix_v, content_v)
9: end while
10: h_a ← producer(/ndn/ucla/NDNlive/stream-1/audio/
11:  content)
12: setcontextopt(h_a, cache_miss, ProcessInterest)
13: attach(h_a)
14: while TRUE do
15:     Name suffix_a ← audio frame number
16:     content_a ← audio frame captured from mirophone
17:     produce(h_a, Name suffix_a, content_a)
18: end while
19: function ProcessInterest(Producer h, Interest i)
20:     if NOT Ready then
21:         appNack ← AppNack(i, RETRY-AFTER)
22:         setdelay(appNack, estimated_time)
23:         nack(h, appNack)
24:     end if
25:     if Out of Date then
26:         appNack ← AppNack(i, NO-DATA)
27:         nack(h, appNack)
28:     end if
29: end function
```

Algorithm 9 NDNlive consumer

$h_v \leftarrow$ **consumer**(/ndn/ucla/NDNlive//stream-1/video/
content, *UDR*)
setcontextopt(h_v, **new_segment**, *ReassambleVideo*)
while *reaching Video_Interval* **do**
    $Name\ suffix_v \leftarrow$ video frame number
    consume(h_v, $Name\ suffix_v$)
    $framenumber++$
end while
function ReassembleVideo(Data **packet**)
    content $\leftarrow$ reassemble **segment**
    if *Final_Segment* **then**
        video $\leftarrow$ decode **content**
        Play *video*
    end if
end function
$h_a \leftarrow$ **consumer**(/ndn/ucla/NDNlive/stream-1/audio/
content, *SDR*)
setcontextopt(h_a, **new_content**, *ProcessAudio*)
while *reaching Audio_Interval* **do**
    $Name\ suffix_a \leftarrow$ audio frame number
    consume(h_a, $Name\ suffix_a$)
    $framenumber++$
end while
function ReassembleAudio(Data **content**)
    audio $\leftarrow$ decode **content**
    Play *audio*
end function

5.2.1.2 Player

NDNlive consumer must fetch the live stream information to set up the Gstreamer playback pipeline before it can request any of the audio or video frames. The application has four consumers (Figure 5.12): video content consumer c_1, video stream information consumer c_2, audio content consumer c_3 and audio stream information consumer c_4.

Data retrieval

Consumer / Producer API protocol suite offers three data retrieval protocols: SDR, UDR, RDR. This section describes how NDNlive player uses SDR and UDR protocols. The operation of NDNlive player is shown in the Algorithm 9.

1. *Content Retrieval*

   In the case of the live media streaming, the player must continue retrieving video and audio frames at all times in order to keep up with the data production rate. All packets of each frame must be retrieved as fast as possible and the fetching process should not block other frames because of packet losses.

   NDNlive video content consumer uses **UDR** (*Unreliable Data Retrieval*) protocol for video frame retrieval. Since **UDR** pipelines Interests transmission and does not provide ordering, some Data packets may arrive out of order. NDNlive player takes care of Data packet reassembly and drops the whole frame is any of its packets are lost.

   NDNlive audio content consumer uses **SDR** (*Simple Data Retrieval*) for audio frame retrieval. SDR does not pipeline Interest packets, which satisfies our requirements, since the audio frame is small enough to fit in just one Data packet.

2. *Stream Information Retrieval*

   Stream information is periodically updated by the video publisher, which essentially means creation of a new Data packet with a unique name (e.g. new timestamp name component). The consumer that is trying to join the live stream does not know the unique name of the latest stream information object, and therefore cannot use UDR or RDR protocols that require such knowledge. A simple solution of this problem is to use **SDR** (*Simple Data Retrieval*) protocol with *Right_Most_Child* option set as TRUE. The protocol generates a single Interest packet with *RightmostChildSelector* which is capable of fetching the latest stream info object.

Frame-to-frame interval

The player should control the speed of frame consumption, while the pipelining of Interests inside each frame is handled by the consumer context. If the application consumes different frames too aggressively and the data is not yet produced by the publisher application, the playback may collapse. If the application consumes frames too slow, the playback may fall behind the video generation. NDNlive uses constant frame rate encoding, therefore for a video, which is encoded by 30 frames per second, the interval between frames is 33 millisecond. In other words, if the next frame cannot be retrieved within 33 ms, the playback will stop.

Our strategy to scheduling frame consumption is to adjust it to the real-time RTT. The amount of frames which are requested at the same time is equal to the frame pipeline window. Any finished frame will allow fetching of the following one frame. For example, if the current frame pipeline window includes frame 13 to frame 22, the completion of frame 12 fetching immediately triggers the fetching of the frame 23.

Frame_pipeline_window is calculated according to the observed *Frame_RTT*:

$$Pipeline\_Window = Frame\_RTT/(1000/FrameRate)$$

$Frame\_RTT = Accumulated\_Time / Frame\_Numbers$

The *Frame_RTT* is equals to the average retrieval time of a single frame. The intuition behind this formula is very straightforward. If the speed of frame retrieval must be matched to the speed of frame generation, the number of frames requested at the same time must be no less than the frame pipeline window.

Synchronization of video and audio

Since NDNlive is streaming video and audio separately, it is a vital problem to keep these streams synced. When video and audio frames are captured, they are timestamped by the Gstreamer. The time information is recorded in *GstBuffer* data structure containing the media data, and transferred along with every video or audio frame. When the consumer fetches the video or audio frames separately, the video and audio frames are pushed into the same *GstQueue*. Gstreamer extracts the timestamps present in the video and audio frames, and displays the content in synchronized mode.

5.2.1.3 Influence on the API

The live video streaming project demonstrated the need for the fast signing / verification communication patterns, because on-the-fly signing of a high number of Data packets needed to accommodate the video frames once again proved to be a performance bottleneck like it did in NDNvideo project [D 12], which had to downgrade to SHA256 digest computation suitable for content integrity protection only.

Application developer experienced some ambiguity with selecting the data retrieval protocol for the video player. In one hand, UDR protocol satisfies latency requirements of the live streaming applications, but forces application to reassemble application frame segments. In another hand, RDR reassembles ADU segments, but might take more time due to the possible retransmissions and error

corrections. However, it is possible to fine tune RDR to be as fast as UDR by setting the maximum number of retransmissions to zero via the **setcontextoption()** API primitive. In such case, the video frames are automatically dropped each time when a single Data packet is not delivered at the first try.

5.2.2 NDNtube

Figure 5.13 illustrates the architecture of NDNtube, which can be summarized as follows:

- *Publisher*

  NDNtube is a *pre-recorded media streaming* application, therefore the publisher works with existing video files stored on the disk. The publisher reads the file from the disk, extracts video and audio frames from it and publishes these frames to the Repo. After that, the Repo takes over the duty of responding to the Interests requesting the frames.

- *Player*

  Comparing to the NDNlive, NDNtube player has an additional functionality for displaying the list of the currently available video resources (e.g. playlist). In order to support this feature, NDNtube publisher keeps publishing the updated playlist every time a new video is added to the collection.

NDNtube's namespace is mostly similar to NDNlive, with the following four differences:

1. *Playlist namespace branch*

   The user of NDNtube can select any video from the list of available ones (e.g. playlist). The typical name of the playlist is shown below.

   "/ndn/ucla/NDNtube/playlist/1428725107042"

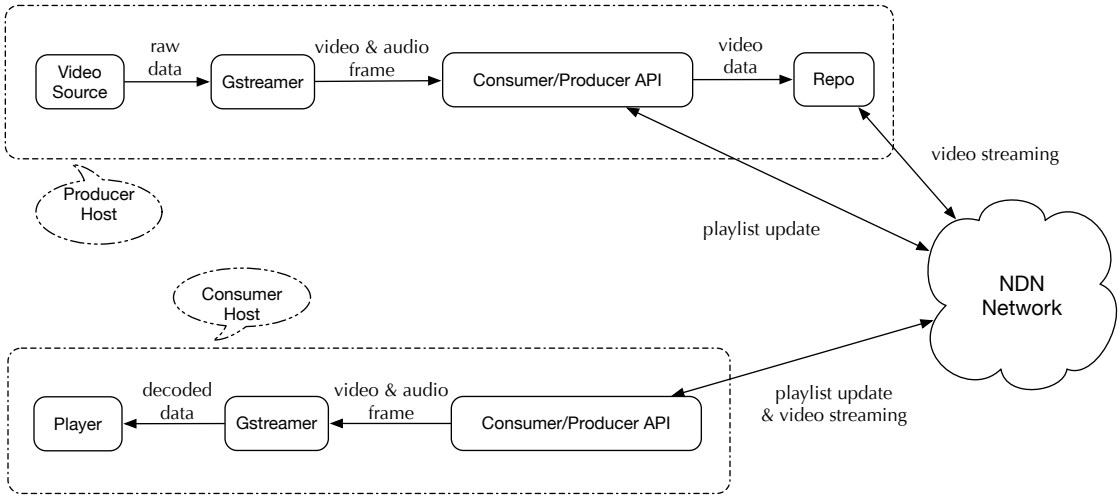

Figure 5.13: NDNtube architecture

The playlist is identified by the timestamp name component, because it is updated every time the new file is added or the old file is removed from the collection of media resources. The consumer is interested in the latests version of it (e.g. rightmost).

2. *Video name*

   The name of the video must match one of the video names provided by the playlist. Semantically, the video name component serves the similar purpose as the stream ID component in the NDNlive application.

3. *Permanent stream information*

   In NDNtube, the information object carrying auxiliary video encoding information (e.g. final frame number, width, height, etc.) is published only once and is not updated after that, unlike the stream information in NDNlive's live streams. As a result, the name of the information object does not contain a timestamp component.

4. *Multi-packet audio frames*

   Since some mp4 video files that are added to the collection of media resources

contain a high quality audio stream, the audio frames have to be broken into Data packets that have unique segment name component (Figure 5.14).

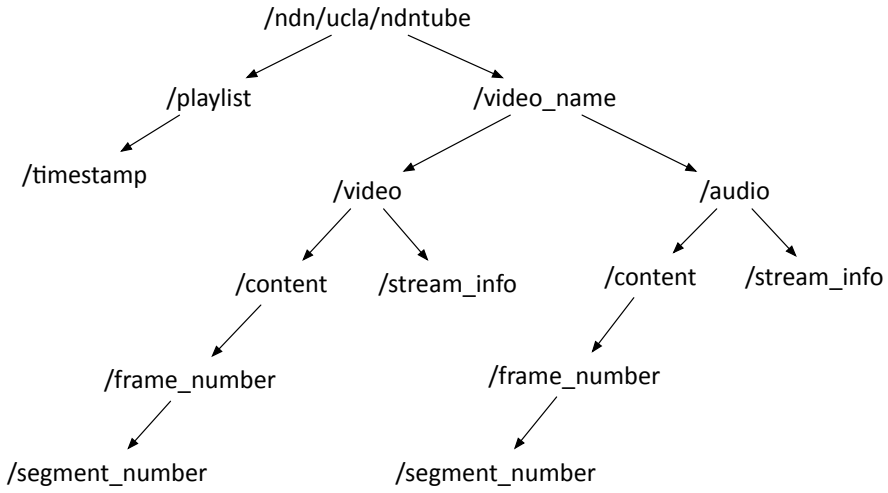

Figure 5.14: NDNtube namespace

Although the namespace of NDNtube might look very similar to the namespace of NDNlive, the patterns of the data production and retrieval are quite different.

5.2.2.1 Publisher

Publisher's application has three producers: dynamic playlist producer, video content producer and audio content producer. Figure 5.15 shows the locations of the producers in the NDNtube namespace.

Playlist producer P_1 is responsible for generating the latest playlist every time a video file is added or removed from the collection of media resources. Producer P_1 runs for the duration of the publisher application's lifetime.

Video content producer P_2 is responsible for publishing the video frames and the stream information object for a each particular media resource. Since producer P_2 is configured with *LOCAL_REPO* option, all packets are written to the Repo running on the same local host. After all video frames as well as stream

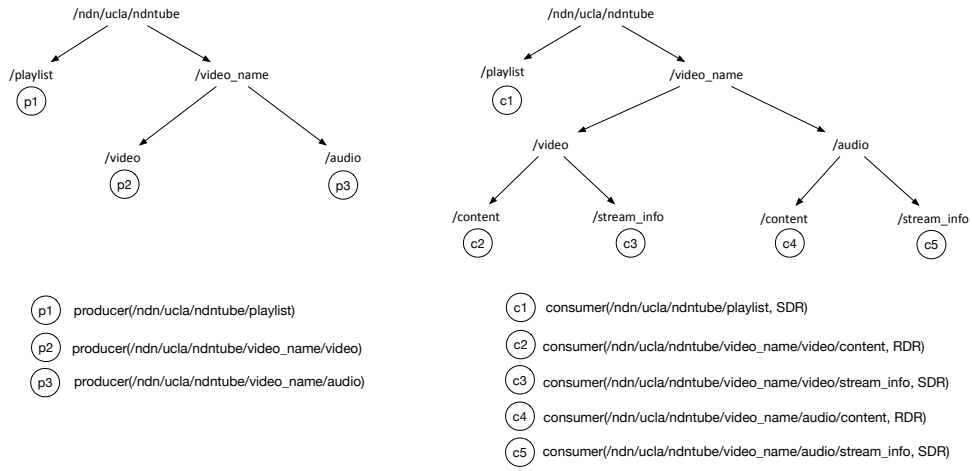

Figure 5.15: Locations of producers and consumers in the NDNtube namespace

information objects are successfully inserted in the repo, producer P_2 terminates its execution, and the publisher-process continues to run.

Audio content producer P_3 is responsible for publishing the audio frames and the stream information objects for each particular media resource. Since producer P_3 is configured with *LOCAL_REPO* option, all packets are written to the repo running on the same local host. After all audio frames as well as stream information objects are successfully inserted in the repo, producer P_3 terminates its execution (Algorithm 10).

5.2.2.2 Player

The application has five consumers: playlist consumer C_1, video content consumer C_2, video stream information consumer C_3, audio content consumer C_4 and audio stream information consumer C_5. Figure 5.15 shows the locations of the consumers in the NDNtube namespace.

Data retrieval

This section describes how NDNtube player uses SDR and RDR protocols. The operation of NDNtube player is shown in the Algorithm 11.

Algorithm 10 NDNtube publisher

1: $h_v \leftarrow$ **producer**(/ndn/ucla/NDNtube/video-1234/video)
2: **setcontextopt**(h_v, **local_repo**, *TRUE*)
3: **while** *NOT Final_Frame* **do**
4: $Name\ suffix_v \leftarrow$ video frame number
5: $content_v \leftarrow$ video frame
6: **produce**(h_v, $Name\ suffix_v$, $content_v$)
7: **end while**
8: $h_a \leftarrow$ **producer**(/ndn/ucla/NDNtube/video-1234/audio)
9: **setcontextopt**(h_a, **local_repo**, *TRUE*)
10: **while** *NOT Final_Frame* **do**
11: $Name\ suffix_a \leftarrow$ audio frame number
12: $content_a \leftarrow$ audio frame
13: **produce**(h_a, $Name\ suffix_a$, $content_a$)
14: **end while**

1. *Content Retrieval*

   All video and audio frames as well as stream information objects are retrieved by **RDR** (*Reliable Data Retrieval*) protocol, which provides ordered and reliable fetching of Data packets. NDNtube video player does not consume a live streaming media, and consequently can afford much larger buffering delays in order to preserve the original quality of the video and audio resources. By default, NDNtube buffers for at least two seconds of video and audio frames of real playback time before it begins (or resumes) its playback. Buffering allows to soften the delays of frame retrieval due to possible Interest retransmissions done by the RDR protocol.

   An expected but nevertheless interesting effect of frame-by-frame reliable delivery shows itself in rare cases when a particular video or audio frame

cannot be retrieved within a reasonable amount of time (e.g. Interest retransmissions) and application faces the choice whether it wants to skip the frame or try to consume() it again. Since our goal was to prototype a Youtube-like user experience, in this situation, NDNtube consumer will try to retrieve the same frame again.

2. *Playlist Retrieval*

   Playlist is periodically updated by the video publisher, which essentially means creation of a new Data packet with a unique name (e.g. new timestamp name component). The consumer that is trying to obtain the names of available media resources does not know the unique name of the latest playlist, and therefore cannot use UDR or RDR protocols which require such knowledge. A simple solution of this problem is to use **SDR** (*Simple Data Retrieval*) protocol with *Right_Most_Child* option set as TRUE. The protocol generates a single Interest packet with *RightmostChildSelector* which is capable of fetching the latest playlist.

Frame-to-frame interval

Since all the content and stream information already exists in the Repo for a long time, consumer can be quite aggressive with fetching video and audio frames. By default, NDNtube player does not wait any time between the retrieved frames and, in fact, fetches 25 video frames in parallel via the **consume()** operation. The number of simultaneously fetched frames does not have to be the same throughout the playback and can change depending on network conditions (e.g. congestion signal, increased RTT).

The NDNtube consumer's operation is shown in Algorithm 11.

Algorithm 11 NDNtube consumer

1: $h_v \leftarrow$ **consumer**(/ndn/ucla/NDNtube/video-1234/
2: video, *RDR*)
3: **setcontextopt**(h_v, **new_content**, *ProcessVideo*)
4: **while** *NOT Final_Frame* **do**
5: $Name\ suffix_v \leftarrow$ video frame number
6: **consume**(h_v, $Name\ suffix_v$)
7: $framenumber++$
8: **end while**
9: **function** ProcessVideo(byte[] **content**)
10: *video* $\leftarrow$ decode **content**
11: Play *video*
12: **end function**
13: $h_a \leftarrow$ **consumer**(ndn/ucla/NDNtube/video-1234/
14: audio, *RDR*)
15: **setcontextopt**(h_a, **new_content**, *ProcessAudio*)
16: **while** *NOT Final_Frame* **do**
17: $Name\ suffix_a \leftarrow$ audio frame number
18: **consume**(h_a, $Name\ suffix_a$)
19: $framenumber++$
20: **end while**
21: **function** ProcessAudio(byte[] **content**)
22: *audio* $\leftarrow$ decode **content**
23: Play *audio*
24: **end function**

5.2.2.3 Influence on the API

The video streaming project taught us many lessons. First, the application struggled with achieving high throughput of the video frames, which was limited to about 20 frames per second (less than 0.5 Mbits). At that moment of time, RDR protocol used a slow start with AIMD flow control scheme, therefore most **consume()** operations were taking multiple round-trips to complete the data retrieval of a single video frame.

We experimented with keeping the current sliding Interest window size between the **consume()** calls in order to do slow start only once per each video stream and to keep the window size large enough to fetch all video frame segments during the same round-trip. This design caused an extremely high rate of unsatisfied Interest packets 90% due to the fact that the video stream is composed of key and delta frames. Key frames tend to be large (20 Data packets), whereas delta frames usually fit in one Data packet. The highly unpredictable video frame size distribution invalidates the concept of re-using the size of the sliding Interest window for consecutive **consume()** calls.

As a result, the video player was rewritten to use the parallel consumption communication pattern with multiple threads. It was noted that API could support parallel consumption in a single thread. Such functionality had been implemented later.

The sheer size of the video content made it clear that the video publisher does not want to do the signing every time the client requests the content and it would be much more scalable to perform this expensive operation only once. This also means that the signed Data packets must be stored outside of the running application process. These considerations prompted us to add the integration with Repo storage to the producer context API.

5.2.3 NDNradio

NDNRadio is a stand-alone application that streams radio over NDN instead of HTTP/TCP/IP protocol stack. The design of NDNRadio allows almost full independence between the player and the station, with very little configuration information needing to be exchanged between the two.

NDNRadio station sources the audio from Radio Reddit using the REST API; it is completely stateless, and contains no information about the NDNRadio players currently or previously listening to it. An NDNRadio station can run non-stop, with all NDNRadio players listening to that station automatically updating themselves at the appropriate time.

The NDNRadio player finds all available stations at startup in a decentralized manner, and then displays those stations to the end user in a user-friendly, easy to use GUI. When the user selects the station to play, NDNRadio player acquires the station information: a) what name must be currently used to get the initial audio frame, b) when to request a new start time from the station, and c) when to restart the frame number.

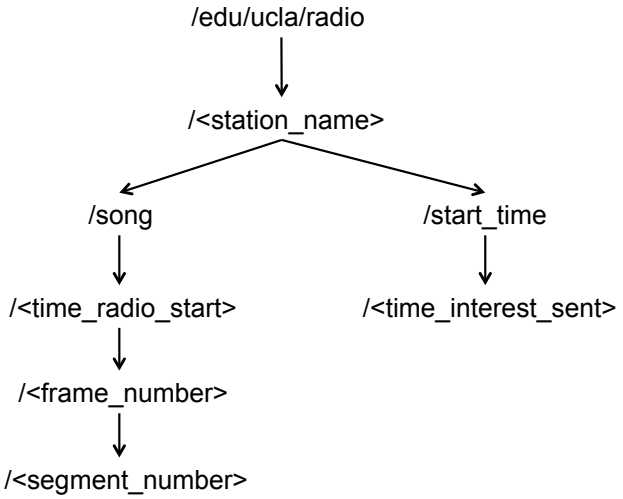

Figure 5.16: NDNRadio namespace design.

5.2.3.1 Station

A single NDNRadio station consists of two NDN producers (Figure 5.17) and one external connection to a Reddit audio stream (e.g. rock, jazz, blues, pop, etc.). Multiple radio stations must be run in separate processes or threads.

Producer P_1 is responsible for 1) replying to the station discovery requests with a packet carrying the station name and dummy payload, and 2) replying to the requests for the station configuration information.[2]

Producer P_2 runs in a separate thread from the producer P_1. It publishes audio pieces obtained from the external Reddit audio streaming resource in a non-stop fashion.

5.2.3.2 Player

The NDNRadio player is a multithreaded application with the audio player (g-streamer) and NDN consumer working in separate threads. Separation of data transfer and audio playback functionality in separate threads allows faster transfer of audio frames. The NDNRadio player also has a buffering time of 3 seconds.

NDNRadio player can play only one stream at a time and is composed of three consumer contexts (Figure 5.17):

- Consumer C_1 runs before the NDNRadio player creates its GUI. It identifies all available stations using the name discovery mechanism through the iterative exclusion accessible with SDR protocol of Consumer / Producer API framework. Iterative exclusion works based on the assumption that different radio stations share the same routable prefix.[3]

[2]Please, note that these two operations can be merged into one.

[3]In the current implementation, radio stations are running in separate processes on the single machine. The broadcast forwarding strategy was set from the Producer API to ensure Interest forwarding to all producers.

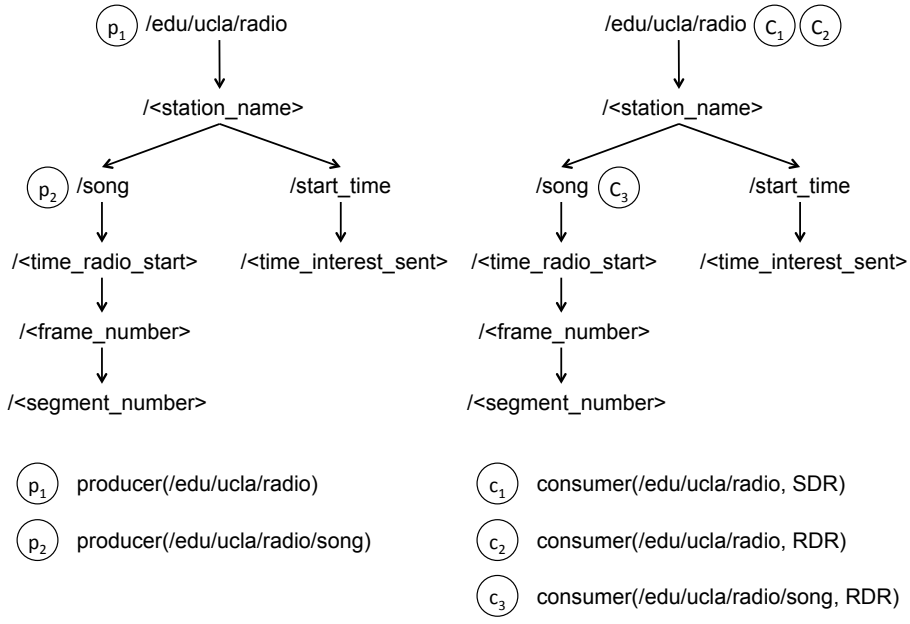

Figure 5.17: Locations of producers and consumers in the NDNRadio namespace.

- Consumer C_2 is activated when the user hits the play button on the GUI. It is responsible for retrieving the radio station configuration information via the RDR protocol. The information is used for constructing the request for audio frames of the specific audio stream by the consumer C_3.

- Consumer C_3 sequentially fetches the audio frames of the audio stream using the knowledge about the audio stream naming obtained by the consumer C_2. Consumer C_3 has the longest lifetime out of these three consumers.

5.2.3.3 Influence on the API

Streaming radio over NDN confirmed our expectations about the need to support the basic "single Interest - single Data" communication pattern via the API (e.g. SDR protocol). The radio player used SDR protocol with iterative exclusion to gain the knowledge about available radio stations.

Another communication pattern that is heavily used in NDNradio is a sequential fetching of the song fragments. Early implementations of the API library had

some problems with fast exiting / resuming to the new **consume()** operation that were fixed as soon as the project demonstrated this inefficiency.

5.2.4 Bittorrent over NDN

Bittorent is a peer-to-peer system with two types of users: seeders and leechers. Seeders act as producers of all content they are willing to serve to the network, and leechers act as consumers. All interaction with the network is done at the chunk granularity, where each chunk is a fixed-size portion of a file. The adopted Bittorent protocol leverages torrent files from the original protocol to encapsulate the metadata required to participate in sharing. Also in the spirit of torrent, when a leecher successfully downloads a chunk, it immediately acts as a seeder for this chunk.

The original format of torrent files, which traditionally includes unnecessary information (announceList, announceURL), is preserved in order to have the same baseline for comparing different implementations of the transport / network layers. Another benefit, is the opportunity for the user to load any content and torrent files from an existing TCP/IP based Bittorent implementation and port them to the NDN Bittorent application without any additional effort.

NDN Bittorent names start from torrent/ name component followed by the name of the specific torrent <torrent name>/, which is taken directly from the torrent file. The next component of the application-level name is a unique name for each chunk, as this is the granularity at which the torrent application is implemented. The application developers decided to use a unique index for each chunk, making the next portion of the name <chunk id>. So the complete name for a given chunk is: /torrent/<torrent name>/<chunk id>

5.2.4.1 Seeder & Leecher

Seeder is the component that is responsible for servicing chunk requests from the peers. It utilizes the Producer API to handle network layer communications. In other words, Bittorent's seeder is responsible only for handling events associated with incoming Interest packets and constructing outgoing torrent chunks. Until the TorrentClient gives the seeder chunks to upload, the seeder will send a NACK for any Interests for data it does not have.

The seeder maintains the list of uploadable chunks, so that it can handle the CacheMiss events from the Producer API whenever an Interest arrives requesting data that was not cached in the network or in the in-memory cache of the producer context. In other words, this is called whenever the seeder must actually upload data to the network.

Leecher is responsible for correctly acquiring the necessary chunks required to complete the files in the torrent. The leecher is implemented using the Consumer API, which manages them majority of networking operations. A failure to download a chunk is handled by trying to download the chunk again via the **consume()** API primitive. The leecher is also responsible for communicating with the torrent client when a requested chunk has been downloaded and verified.

5.2.4.2 Influence on the API

When the leecher successfully downloads the chunk, it saves the chunk to the disk. Similarly to the Bittorent application, the seeder process picks up the new files and publishes these chunks via the **produce()** primitive. While it is possible to recreate the original structure of the chunk (e.g. put the right files in it) by looking at the bittorent file, it is not clear how sign this new "reproduced" chunk, because the leecher doesn't have the private key of the original publisher of the content. Since the original Bittorent protocol offers only an integrity protection

of the chunks via checksums, NDN Bittorent can easily achieve parity with it by doing a simple checksum over the content, but some concerns had been raised wether there should be a mechanism of preserving the original signatures of the Data packets even after the ADU is reassembled.

This problem motivated us to add the extension to the **produce()** function, which accepts a single Data packet instead of the application frame. Using this extension, for example, an NDN Bittorrent application can pass the unmodified Data packets from the leecher to the seeder without loosing the original signatures.

Since each Bittorent client usually downloads multiple chunks simultaneously, we received a request from the application developer to enable parallel consumption within a single thread. The implementation of API has been extended with **asyncConsume()** operation allowing to schedule multiple fetching operations within the same thread.

CHAPTER 6

Bidirectional Consumer / Producer communication

The Web today is a universal platform for many kinds of services, from familiar content browsing and media streaming to purpose-built applications hosted in browsers and in stand-alone agents. The backbone of the web is the HTTP protocol [BFF96] [FGM99b], which is based on a request/response model running on top of a point-to-point connection to a server. A client sends a request in the form of a message containing a URI [BFM05], request meta-information, and possible body content. The server responds with a message containing entity meta-information, and possible entity-body content.

In this chapter, we examine diverse approaches to matching the needs of this important category of modern applications to the capabilities of the NDN protocol architecture. The analysis shows that some of the desirable communication patterns (Sections 6.2.4, 6.2.2 and 6.2.5.2) are not fully supported at the current stage of the NDN architecture design. While the NDN architecture is still flexible and being shaped via experimentation, and these communication patterns might become available in the future, this chapter contains an example of "possible-today" design of HTTP/RESTful style applications built on top of Consumer / Producer API framework (Section 6.4).

6.1 Characteristics of HTTP/RESTful communication

In addition to carrying content data, the HTTP protocol defines a wide range of meta-data for both requests and responses. The meta-data are carried in HTTP headers, part of the messaging protocol. The meta-data sent in client requests may be information about the client application itself, including information about acceptable content languages and data encodings.

A large fraction of these Web applications use a transactional paradigm known as Representational State Transfer (REST) [FT02]. REST improves scalability by distributing application state from servers to clients. A pure RESTful request is self-contained — it carries all the information necessary for a service to process the request. Without the client-side context, a RESTful service may be inefficient or impaired, or may not be able to function at all. The familiar HTTP cookie is a simple form of distributed context, where a service uses the HTTP protocol to convey tokens, often opaque, to its clients. The tokens are typically unique to each client; this allows the service to associate multiple requests from a given client together. The cookie may carry client-side state directly, or may be used as a reference to state held at the server.

The usability of these distributed applications depends on the latency between user action and the rendering of a result. Because the Web is composed of multiple highly distributed services, this issue of latency (round-trips within the transport or the application protocol) has considerable importance in the system design. Modern browsers and other applications have evolved to be ever more efficient in the way they use round-trips, by caching content locally, and by reusing DNS information and TCP connections. Some modern browsers even speculatively initiate DNS queries and TCP connection activity in order to improve perceived responsiveness [Gri].

As we view the current state of HTTP/RESTful communication, then, we see

these key points:

- Clients have data to send in their requests, in the form of HTTP header meta-data and other application-specific RESTful state.
- Many client requests are intimately bound to the client context data associated with them; the context and the request are carried together in the HTTP communication protocol messages. The client-specific data tends to make each request unique even when clients are accessing common resources or services.
- Latency and number of network round-trips are key factors in efficiency and perceived responsiveness.

Given our understanding of the existing NDN protocols, RESTful interactions encounter a number of challenges:

- All of the client-side context and meta-data associated with a request must be encoded in the Interest name field: no other field is present in the base NDN architecture.
- NDN Content objects are immutable, and the object names are bound to fixed data. Services are not able to return different results based on client-specific processing unless the clients use unique names in their requests.
- NDN's stateful forwarding supports clients (consumers) who do not have to have a globally-routable address (name). Web services that require bidirectional data flow cannot get their own requests to their clients unless the clients have routable names.
- Many web sites and RESTful applications depend on being able to identify (or at least count) the specific clients making requests. NDN mechanisms

like Interest aggregation and pervasive caching prevent producers from seeing some Interest packets.

In the next section, we will examine a range of approaches to supporting RESTful and Web applications on NDN networks. Each approach addresses the challenges we have outlined above in a different way, adapting the basic NDN protocols in more or less significant ways.

6.2 Theoretically possible communication patterns

In this section, we explore communication patterns suitable for running transactional and interactive REST- or Web-like applications over NDN. We focus on the key issue as we see it: how can servers obtain the client meta-data and context information that is associated with client requests? We start with an approach that relies solely on the existing design of NDN, but several important drawbacks compel us to try out alternative patterns which introduce various degrees of changes in NDN. The two types of NDN packets divide the discussion into two corresponding categories: approaches using clients' Interest packets alone, and approaches where the server is obliged to retrieve information from client-sourced Data packets.

We discuss benefits and disadvantages of each communication pattern as well as the effect each has on the network, considering several specific factors as we examine each approach:

- Interest name size: impact on routers
- Data name size: impact on Data packet efficiency
- Round-trips
- Client data segmentation and reliable delivery

- Potential security vulnerabilities: reflection, amplification, flooding, spoofing/poisoning

6.2.1 Name Component

In the basic NDN protocol design, an Interest packet carries little more than a Name field. In some previous work, Interest names have been used to pass commands to an NDN router [ccn14], to pass authenticated requests to a lighting controller [BHM12], and to convey the current state of a system to support distributed dataset synchronization [ZA13].

Today's interactive Web applications need to pass meta-information and application-specific data with their requests, so we begin by examining the consequences of using the Interest's Name field to convey that information. An Interest packet is routed through the network via the name it carries. Application meta-information and client-side data required for a particular type of request could be carried by appending it using one or more trailing name components. This pattern is illustrated in Figure 6.1.

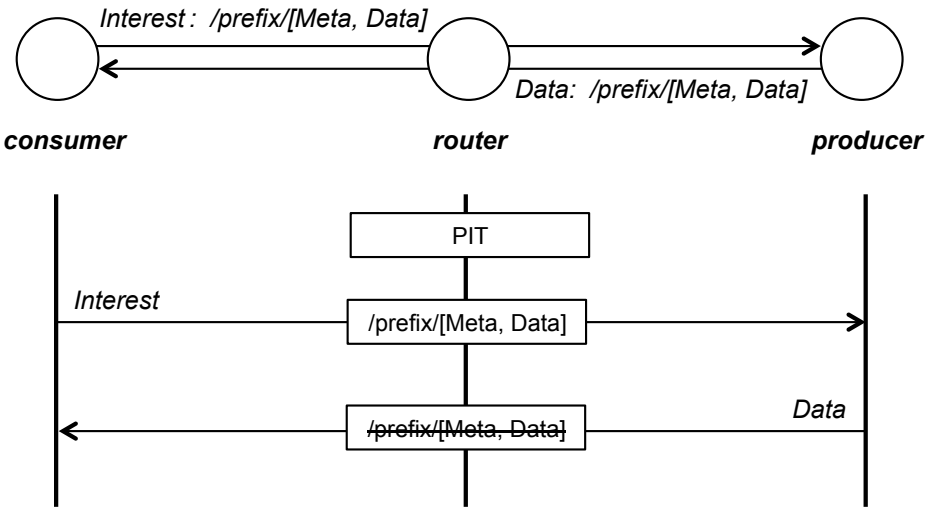

Figure 6.1: Interest name carries client-side information.

This communication pattern works with the current NDN architecture, and naively seems fairly natural. The client-side data is bound to each Interest packet directly, satisfying the server's expectation that the client-side context will be present along with each client request.

However, there are a number of significant drawbacks to this simple approach. The first concern is related to stateful packet forwarding in NDN. Contemporary HTTP requests that perform browsing often convey hundreds of bytes (or even kilobytes) of supplemental information in HTTP headers [NJA13][Ram]. If this meta-information and application-specific data is placed in the Interest name, there may be a significant additional overhead on intermediary NDN routers. Each router will have to process these large names, increasing the computational load, and the accumulated name state held in their PIT data structures will consume substantially more memory.

A second concern is decreased network throughput and increased nodal processing delays. The entire name must be echoed in each Data packet. Inside the NDN router, longer names may lead to more operations on name components, slowing down packet processing. The name-to-payload ratio can turn out to be far from optimal. Regardless of the eventual fragmentation scheme NDN proposes, large names will reduce available packet space, reducing space for the actual content. This leads to decreased goodput, and potentially more fragmentation and reassembly operations per Data packet.

A third concern is the possibility of cases where a single Interest name is not able to carry all required application data. While there is no clear consensus within the NDN community on the maximum allowed size of the name, there is a clear possibility that meta-information and application data (e.g. HTTP POST or a large cookie) may be larger than the maximum name length can accommodate. Within the constraints of the current NDN protocol, meta-information and application data would have to be subdivided into multiple Interests' names, trans-

mitted as multiple Interest packets and reassembled by the producer. Figure 6.2 illustrates how NDN might accommodate transmitting arbitrary sized client-side data to the producer, and retrieving an arbitrary sized response from it.

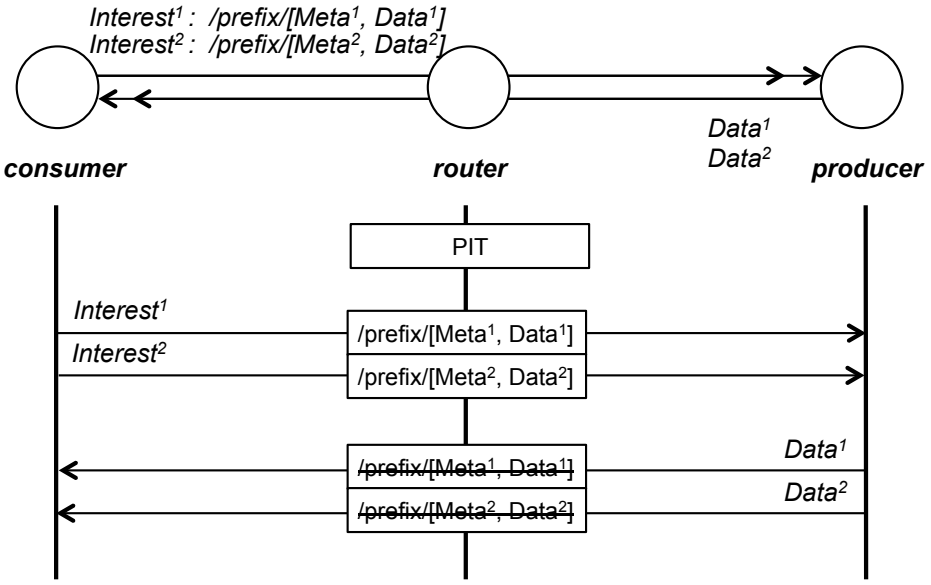

Figure 6.2: Client data carried in multiple Interests.

According to this pattern, the consumer sends no fewer Interests than needed to both accommodate client-side data in Interests and fetch all segments of the producer's reply. This pattern appears to take only a single round-trip to transmit the whole request and receive the whole reply. But once multiple related packet transmissions are introduced, we now need to consider some sort of reliable delivery of consumer-supplied information. That is, the client must re-transmit its Interests in the absence of any timely response or acknowledgement that they have been delivered. This complexity leads us to examine some alternative protocol approaches.

The conventional design of NDN interactions is that the producer acknowledges arrival of Interest packets in its Data packets, but the situation may be more nuanced. The completion time for Web and application requests requiring dynamic on-demand content can vary widely. As a result, it is not clear how the client

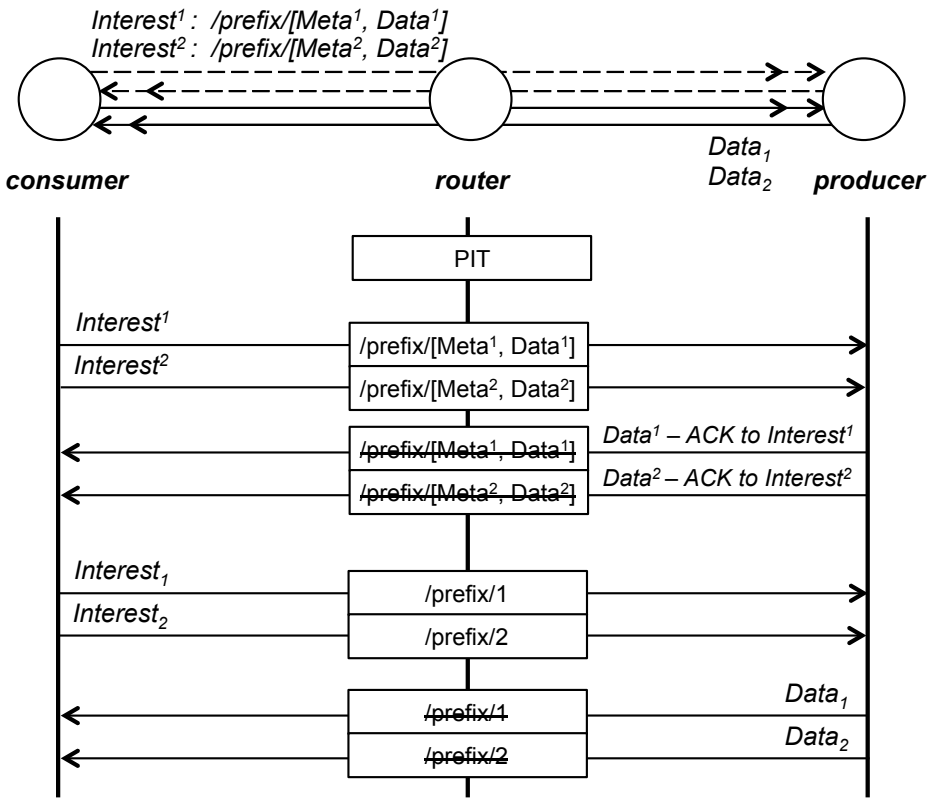

Figure 6.3: Two-phase Interest exchange.

should estimate waiting time between Interest retransmissions. One extreme is to use an Interest retransmission timer at the scale of network RTT. But this may result in many unnecessary retransmissions of Interests if the server processing time is significantly greater than RTT. The other extreme is to use a timer scaled to the tolerable application response delay. This in turn results in poor responsiveness in cases when network retransmission is indeed necessary. NDN protocol mechanics do not inherently distinguish network-level and application-level responsiveness, despite the substantially differing time scales.

One solution might be for the producer to use Data packets to acknowledge delivery of Interest packets containing meta-information and application data. The producer application would acknowledge each of these Interest packets prior to the execution of the actual content request, resulting in a two-phase operation.

First, an initial set of Interest packets conveys the client-side data; Data packets from the producer acknowledge receipt of this information. Then a second round of Interest packets retrieves the actual producer-side content. This pattern is illustrated in Figure 6.3.

Note that this approach employs parallel Interest transmission to reduce overall latency. Separating the delivery of client data from the Interests used to retrieve producer content eliminates the need for all Interests in the exchange to use the same name, reducing the Interest size penalty. However, a significant problem with this approach is that the first round of 'acknowledgement' Data packets must be signed with the producer's private key in order to be considered valid. Signing a Data packet is computationally costly. If malicious clients flood Interests like these, this could lead to a denial of service (DoS) attack on the producer. In addition, the producer requires some means of associating the client-side data in the initial round of Interests with the subsequent Interests for the producer's content, resulting in more interaction state at the server.

6.2.2 Compressed name component

Including significant client-side data in Interest names raises concerns about memory scalability for the PITs of intermediary NDN routers, and decreased throughput due to the need to echo the entire name in each Data packet. These concerns can be partially addressed by compressing the client-side data into a constant size compact representation, and using this representation in the router PIT and in Data messages.

To achieve compression, a specialized Name component could be introduced to hold client meta-information and application data. An NDN router recognizing this specialized name component could then compute a hash of the component. This operation would effectively reduce the amount of state held in the PIT,

compressing variable meta-information and application data into a constant size hash value. In order to forward Data packets back to the consumer, the producer application would replace the specialized name component with the corresponding hash value. As a result, Data packet names would continue to match the names in the PITs of intermediary routers, while occupying less space. This technique is illustrated in Figure 6.4.

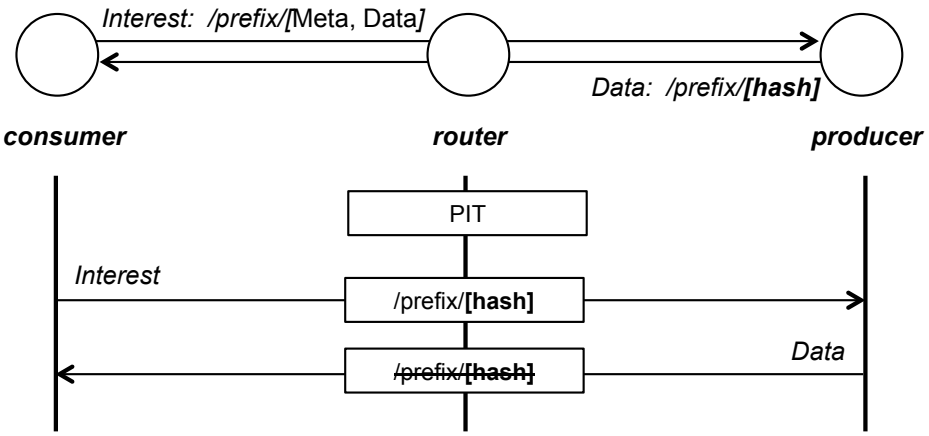

Figure 6.4: Consumer-supplied name-component is compressed to a hash.

6.2.3 Common Issues with Interest Names

Even with name component compression, all protocol approaches where meta-information and application data are pushed in Interest packet name components still have a number of common problems.

Exposure of meta-information and application data impairs confidentiality. If meta-information similar to HTTP cookies and HTTP headers such as Referer and User-agent are passed unencrypted in an Interest name component, the user can be easily tracked and deanonymized by a third-party observer. If security-sensitive data is held in these meta-information structures, the compromise could be even more substantial.

Signature generation must be performed on-the-fly for all Data packets that acknowledge the arrival of Interest packets with names carrying meta-information and application data. The per-client information creates names that are unpredictable, so the producer application must build and sign the corresponding Data packets dynamically. This introduces a potential vulnerability to a resource-exhaustion attack. NDN signature generation with public key cryptography is computationally expensive — significantly more expensive than, for example, SYN cookie generation.

Interest packet flooding in NDN networks can be a vector for Distributed Denial of Service (DDoS) attacks[GTU13]. It has been shown that many Interest flooding attacks can be mitigated by exploiting stateful forwarding in NDN routers, such as by observing the rate with which Interests successfully retrieve Data packets on a per-prefix per-interface basis[AMM13]. If meta-information and application data is pushed in Interests and if producer applications acknowledge every Interest with a Data packet, the per-prefix per-interface statistics may be distorted. An artificially high Interest satisfaction rate might jeopardize detection and mitigation of Interest flooding attacks.

6.2.4 Application Data field

We have examined some approaches to carrying client-side information in Interest names; now we'll explore sending request meta-data in the Interest packet, but outside the Interest name. In this approach, an Interest carries the additional application data in an *ApplicationData* field. This field would contain opaque data, and thereby not influence the operation of NDN routers or their processing in any way. The Interest name only requests the named content, and does not carry any client- or application-specific information. Figure 6.5 illustrates this approach.

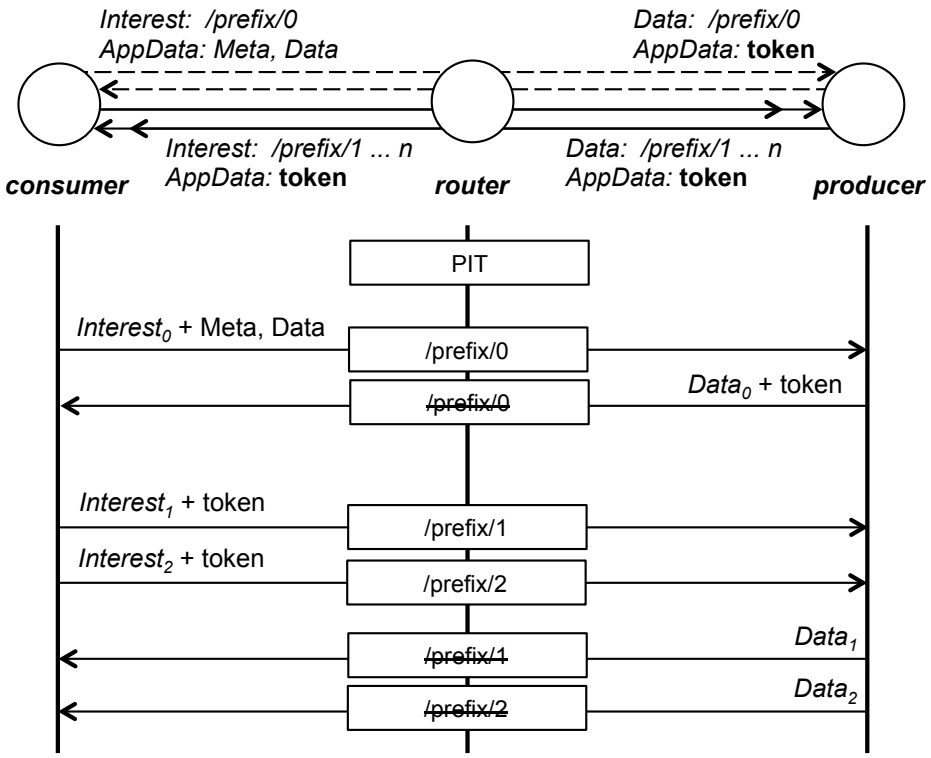

Figure 6.5: Interest carrying ApplicationData field.

The client includes an AppData field in its "base" Interest packet - the Interest for segment zero of a possibly segmented Data object. For Web-like interactions, the AppData field would carry meta-information about the client application, including stored cookies (i.e. what is found today in HTTP headers). In a standalone RESTful application, the field would carry client-side application context data.

The AppData field is opaque to routers. No special name components are present, and no special name processing takes place at routers. If a client Interest packets name a cacheable object, intermediate routers can perform normal CS processing and return the cached data. If an application requires server-side processing, client Interests must use unique-ified names so that Interests from different clients avoid aggregation.

The client does not have to send the entire AppData in each Interest during

a multi-segment exchange. In an ongoing exchange of packets to retrieve larger, segmented Data objects, the server may need to associate the correct client context with each individual Interest in order to respond properly. To accomplish this, the server could generate a token — presumably shorter than the entire client context data — and return it to the client with the first Data packet. Subsequent Interests then would include this token in the AppData, allowing the server to properly associate the client meta-data with each individual Interest packet. If a series of exchanges required dynamic, frequently updated client context, obviously that context would have to be transferred between client and server as it changed.

Employing such a token mechanism requires that each Interest contain either the client context, or a corresponding server-generated token. This may affect a client's choice of initial Interest window size. If the initial Interest window size is just one, data fetching efficiency during the first round trip is reduced. If the initial Interest window is greater than one, the client has not been offered a server-side token, so it must transmit redundant application data in each Interest in this window. The choice of initial window size for Interests using a scheme like this may have delicate tradeoffs.

The communication pattern with the Application Data field has the following benefits:

- The Interest name does not need any special processing. There is no need for complex name matching at the PIT or CS: exact-match for names is available.
- The application context information travels directly with the Interests; the client context, name, and returning data remain bound together.
- The application data can be transferred just once, with the initial Interest. Subsequent Interests can refer to the context if a server-generated token is returned in Data packets.

- No additional round-trips are needed.

Any scheme that "pushes" client data in Interest packets increases Interest packet size, possibly substantially. The NDN property of flow balance assumes that Interest packets will generally be small compared to the corresponding Data packets. Pushing 'unsolicited' data might compromise that property. To address this concern we might consider a limit on the size of Interest packets. A 4KB limit, for example, would be adequate for most current Web-like interactions [NJA13]. However, this is still quite large — possibly large enough to make bandwidth accounting for Interests more important. A RESTful application that required a larger client payload would need to send multiple Interests, or use a different mechanism.

6.2.5 Data Locator field

The alternative to pushing client-side data with Interest packets is a communication pattern where the producer application pulls data it needs from the client. An essential piece of such protocols is a so called Interest-Interest exchange[BGN12]. In this exchange, an initial Interest packet is expressed by the consumer application as usual. This initial Interest prompts the producer application to express one or more Interest packets in return. These requests from the producer retrieve client-specific information from the client; the producer then uses that information to satisfy the client's original Interests.

The Interest-Interest information could be placed in the initial client Interest name, but this approach would suffer from some of the same constraints as the examples in the previous sections — extremely long NDN names have drawbacks. Enclosing one name in another, for example, will not allow both names to approach their maximum lengths, which is inconvenient for application designers.

In our view, a better alternative would be to introduce an optional *DataLo-*

cator field in the Interest packet. The presence of the DataLocator would serve as an indication for the producer that some supplemental information — meta-information, consumer-supplied data, etc. — is available to be fetched from the client before processing the initial request. The DataLocator would therefore contain a name the producer could use to express Interests that reach the client application. We discuss some variations of this mechanism below.

6.2.5.1 Routable name

This pattern requires the consumer application to provide a routable name at which it can be reached. The client must be prepared to package necessary meta information and application data in properly-formatted and signed Data packet(s). The consumer application might acquire a routable prefix from the point of presence (PoP) of the Internet Service Provider (ISP) that it is currently connected to, or through some other means.

The consumer application sends an Interest packet containing the name for the producer to use in a DataLocator field. When the producer application receives the Interest, it transmits an Interest packet using the name specified in the DataLocator field to fetch meta-information and/or application data associated with the client's request. This communication pattern is illustrated in Figure 6.6.

The immediate advantage of this protocol is eliminating "pushed" data from client's Interests, which do not need to convey more than a single name. This restores the NDN flow balance property. A second important benefit is that the producer application is now in control of the data retrieval process. The producer is subject to standard NDN flow control and congestion control mechanisms as it retrieves Data from the client.

A third benefit is that some client-side data can benefit from NDN's natural on-path caching. Web cookies that represent the state of the server, kept on the

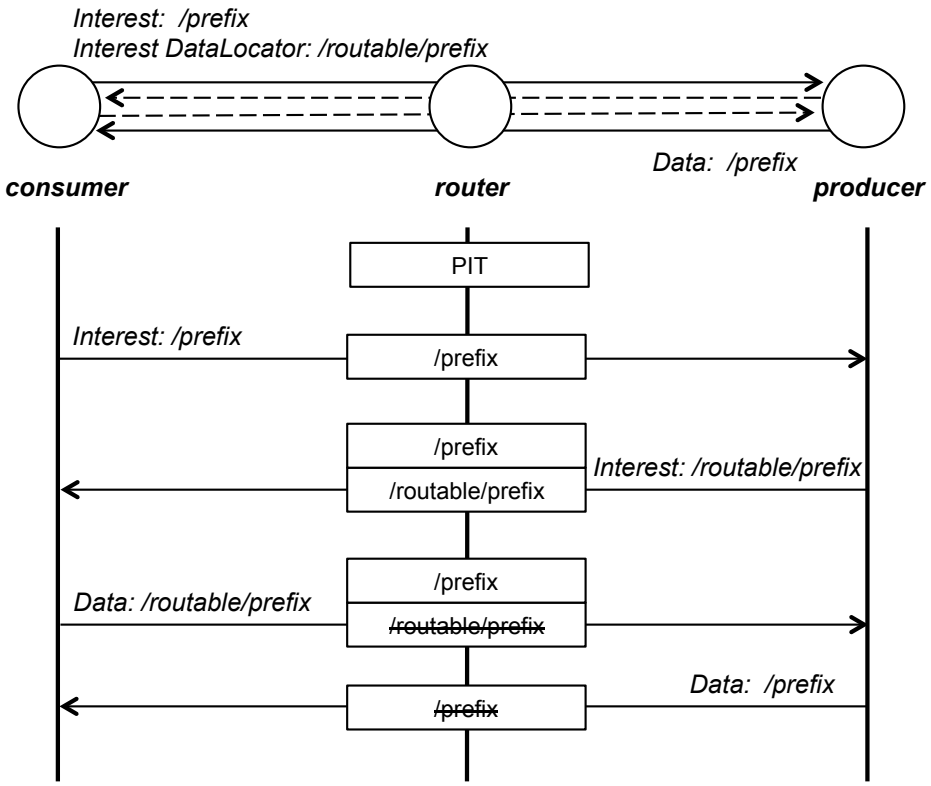

Figure 6.6: Interest-Interest exchange with routable name.

client, may be stable for extended periods of time. Client data associated with related idempotent requests (e.g. HTTP GETs) can be cached in the intermediary routers that are located closer to the producer. Both the client and server therefore benefit from the NDN mechanisms that localize traffic and reduce latency.

However, the use of routable names for the server to fetch client data has several drawbacks. First, the client must acquire and convey a routable name prefix. A mobile consumer will either have to acquire a new prefix every time its connectivity changes, or use some sort of indirection service to map a stable name alias to its current routable prefix. This adds complexity, and introduces the possibility of traffic interruptions.

Second, the DataLocator mechanism's use of a routable name could be used to launch a reflection attack involving the producer. If an attacker specifies the name

of a target third party, the producer will be induced to direct Interests to that third party. The reflection attack might be mitigated if the DataLocator is inspected when Interests enter the client's Internet Service Provider (ISP) network. The ISP ingress router could perform a check similar to an ingress filter in Reverse Path Forwarding (RPF) [BS04], accepting and forwarding Interest packets carrying DataLocators that will route to the source face. The router would drop any Interests with DataLocator names that would route elsewhere.

6.2.5.2 Non-routable transient name

The problems caused by the use of routable prefixes in the DataLocator field prompt us to explore the possibility of using non-routable prefixes for client-side data. This approach uses the per-packet router PIT state to construct an ephemeral path for Interests going back from the producer to the client in a manner somewhat like Kite [Y 14]. As shown in (Figure 6.7), this introduces several changes in the forwarding mechanism of an NDN router:

1. The client constructs a unique name, preferably using a distinguished (by convention) non-routable prefix, and includes it in a DataLocator field.

2. When an Interest containing a DataLocator field arrives at a router, the DataLocator name is saved in the PIT along with the name in the Interest packet itself.

3. The producer responds with an Interest using the non-routable name taken from the DataLocator. As the producer's Interest moves through the network, each NDN router performs an exact match on the producer's Interest name using the extended PIT entries created as it forwarded the client's original Interest. If the router finds a match, it creates a new PIT entry for the non-routable name with the egress interface matching the ingress interface of the original Interest. The FIB is not consulted: the producer's

Interest is forwarded on the inverse path of the consumer's original Interest packet using the PIT alone.

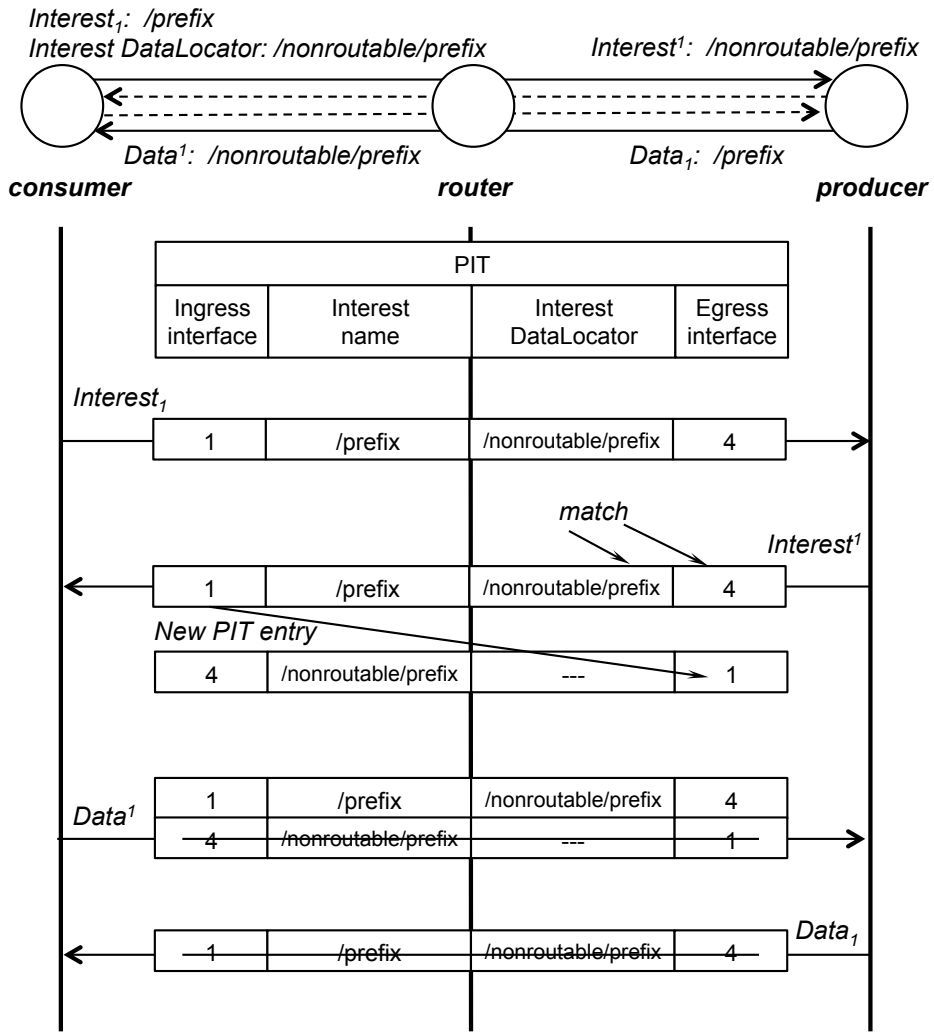

Figure 6.7: Interest-Interest exchange with non-routable name.

The DataLocator name is not independently routable. If the server (or anyone else) tries to access this information object outside the context of the enclosing Interest/Data exchange, the operation will fail. Further, since the names used cannot be forwarded outside the reverse path, reflection attacks are eliminated.

The fact that these non-routable Interests bypass the normal FIB does not prevent them from being satisfied by a Content Store. If a router's CS cache has

a matching entry, this entry can be returned to the producer. However, the non-routable name can take any form, including self-certifying and other flat names, and therefore reverse forwarding cannot depend on longest prefix lookup.

When a mobile consumer changes its connectivity, the path for reverse Interest packets can be quickly rebuilt by client-side retransmission of its Interest packet, which will create necessary PIT state again.

If client meta information or application data is too large to fit in one Data packet, the consumer application segments it into multiple Data packets just as would be done for any large Data object. The producer application issues multiple interests to retrieve the entire information object. In order to accommodate this, the algorithm matching DataLocator names in the PIT ignores any segment number name component. Pipelining would allow a producer to fetch arbitrary size client data with minimal round trips.

6.3 Comparison of communication patterns

In this section, we develop a simple analytic model and use it to characterize each communication pattern. We model bidirectional traffic between an HTTP/Web-like client (consumer) and an HTTP/Web-like server (producer); network traffic is an obvious, key metric applicable to all of the communication patterns we have considered.

The model applies NDN *segmentation* as data objects grow large. Segmentation is the operation where a content producer splits a large data object into smaller pieces, naming and signing each separately. NDN flow balance, the one-to-one correspondence between Interest and Data packets, assumes that Data packets have constant size for simplifying hop-by-hop flow- and congestion control. A large name field reduces the available space for the content payload in each fixed-size Data packet. A given content object requires more or fewer segments

(packets) depending on the payload space made available in each pattern.

The model utilizes a base Interest name (prefix) that is 50 bytes long, with 512 bytes of client-side data. We do not argue that this is accurate or representative of actual traffic; rather that it is not unrealistic given the current web traffic patterns [NJA13].

We use this simplified arithmetic equation to compute the number of segments (NoS):

$$Number\ of\ segments\ (NoS) = \frac{Producer\ content\ size}{Space\ for\ content}$$

The choice of the communication pattern affects the amount of space available for content in Data packets. In the name component pattern, all of the client data is appended to the base name prefix. In the compressed name component pattern, Data packet names have a large hash value (e.g. SHA-512) appended to the base name prefix. In the Interest acknowledgement pattern, client data is not echoed in the name of Data segments. In the application data pattern, producer generates and echoes back a token (e.g. SHA-256), carried in its Data segments. In the DataLocator pattern, client data is not echoed in the name of Data segments.

$$Space = Data\ size - \begin{cases} Prefix - Client\ data & (Name\ component) \\ Prefix - Hash & (Compressed\ name) \\ Prefix & (Interest\ ack.) \\ Prefix - Token & (Application\ data) \\ Prefix & (Data\ Locator) \end{cases}$$

Name component pattern carries client data in the Interest name. Each segment is fetched with an Interest carrying the relatively large name.

$$Interest\ traffic = NoS * (Prefix + Client\ data)$$

Pattern / **Criteria**	Name component	Compressed Name component	Interest acknowledgement	Application Data field	Routable DataLocator	Non-routable DataLocator
Impact on router memory	**Large**	Normal	**Large**	Normal	Normal	2 x Normal
Payload/Name ratio in Data packets	**Low**	Normal	**Low**	Normal	Normal	Normal
Round trips	1 round	1 round	**2 rounds**	1 round	**2 rounds**	**2 rounds**
Support of large client data	**No**	**No**	Multiple Interest packets	**No**	Multiple Data packets	Multiple Data packets
Retransmission of client data	**Slow, timescale of application RTT**	**Slow, timescale of application RTT**	Fast, timescale of network RTT	Fast, timescale of network RTT	Fast, timescale of network RTT	Fast, timescale of network RTT
Disruption scenarios	PIT inflation DoS on router's fragmentation & reassembly	DoS on router's hashing	DoS on producer's signing DDoS with Interest flooding		Client mobility Reflection attack	

Table 6.1: Comparison chart of communication patterns.

Large names force the producer to send more segments, increasing the amount of bidirectional traffic:

$$2way\,traffic = Interest\,traffic + NoS * Data\,size$$

Compressed name component pattern carries client data in each Interest name, but echoes back only the hash of the client data in each Data packet of the producer's response.

$$Interest\,traffic = NoS * (Prefix + Client\,data)$$

Total amount of bidirectional traffic:

$$2way\,traffic = Interest\,traffic + NoS * Data\,size$$

Interest acknowledgement pattern uses an initial series of Interests containing client data, acknowledged with signed Data packets. The producer's actual content is then fetched using the normal length Interest packets.

First round — acknowledged delivery of client data. A Data packet with Interest acknowledgement has a negligible payload, therefore:

$$1st\ round\ traffic = (Prefix + Client\ Data) * 2$$

Second round — fetching segmented content from the producer.

$$2nd\ round\ traffic = NoS * prefix + NoS * Data\ size$$

Total amount of bidirectional traffic:

$$2way\ traffic = 1st\ round\ traffic + 2nd\ round\ traffic$$

Application Data pattern carries client data in a single Interest, in a special Interest packet field. The producer generates and echoes back a token, carried in its Data segments. Subsequent Interests, if any, use the producer's token and do not have to convey the client data explicitly.

$$Interest\ traffic = (Prefix + Client\ data) + NoS * (Prefix + Token)$$

Total amount of bidirectional traffic:

$$2way\ traffic = Interest\ traffic + NoS * Data\ size$$

DataLocator pattern carries an additional name in a special field of each Interest packet. For the purposes of this arithmetic traffic model, the *routable* and *non-routable* locator names are identical. We use 50-byte name lengths for both the content name and the DataLocator name.

The client's Interest packets carry the content name and a DataLocator name. The producer responds with its own Interest(s) using the DataLocator name.

$$Interest\text{-}Interest\ traffic = 2 * Prefix * NoS + Prefix$$

The client provides its client data in Data object(s) using the DataLocator name, then retrieves the actual content from the producer.

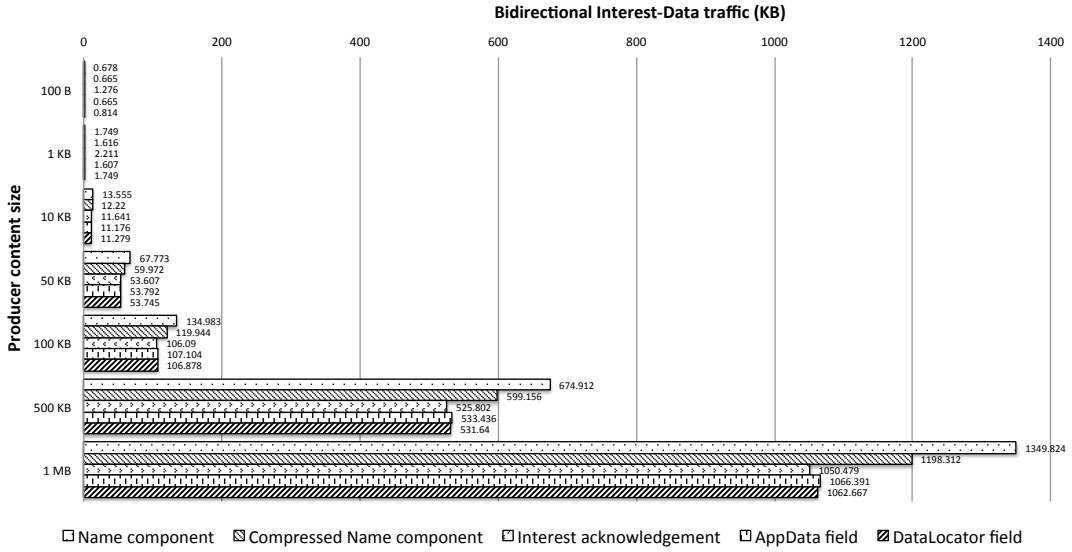

Figure 6.8: Bidirectional traffic model using 512 bytes of client data.

$$Data\text{-}Data\,traffic = Data\,size + NoS * Data\,size$$

Total amount of bidirectional traffic.

$$2way\,traffic = Interest\text{-}Interest\,traffic + Data\text{-}Data\,traffic$$

Figure 6.8 summarizes the results of applying this arithmetic model. The canonical NDN Interest formulation proves to be noticibly less efficient than any other. Name field size has an obvious impact for any but the smallest contents; even the use of a compressed name component has a considerable though less-dramatic impact. The network bandwidth used by the other three protocol patterns is roughly equivalent. This particular metric does not distinguish among these other approaches particularly, though we highlight some key comparison points in Table 6.1 and discuss it in the next section.

6.4 HTTP/RESTful interaction using the Consumer / Producer API

At the present moment, NDN architecture does not support Compressed Name component, Application Data Field and Non-routable Data Locator communication patterns. Bidirectional Consumer / Producer interaction can only be achieved either with Name component or Routable Data Locator patterns. This section describes how Consumer / Producer API framework can be used for this purpose.

6.4.1 Name Component pattern

With the Name Component pattern, all necessary application state goes into the name suffix, whereas the name prefix is used to forward the Interest towards the web service. In this example, we prototype the work of the login page of the email web-service. To get access to the personal inbox, the user provides his/her login and password information which is potentially packaged with some other related information such as language and encoding settings, browser type, etc. in the so called "application state". The Algorithm 12 demonstrates how the application state can be passed from the web client to the web service using the Name Component communication pattern and the retrieval of the web service's reply using the RDR protocol.

6.4.2 Routable Data Locator pattern

In the previous sections, the Routable Data Locator pattern speculated on the possible extension of the Interest packet format, which could have a separate field to carry the routable name (Data locator). However, the existing packet format does not allow any additional name field. A web client application can work around this problem by placing the routable data locator in the suffix of the

Algorithm 12 Passing client state in the name suffix

1: $h \leftarrow$ **consumer**("/com/google/mail/sign-in", RDR)

2: *name component* $\leftarrow$ login, password, language, encoding, etc.

3: **setcontextopt**(h, **ON_CONTENT**, *OnServerReply*)

4: **consume**(h, *name component*)

5: **function** OnServerReply(Producer **h**, Interest **i**)

6: Process the reply from the service and redirect to the next page

7: **end function**

Interest name. A web service can rely on the application specific name design to retrieve the meaningful Data locator from the name suffix of the arriving Interest.

In this example, we prototype the operation of the login page of the Gmail service. To get access to the personal inbox, the user provides his/her login and password information which is potentially packaged with some other related information such as language and encoding settings, browser type, etc. in the so called "application state". The Algorithm 13 demonstrates how the application state can be published by the client and how the client can notify the web service about the application state. Lines 1 — 4 are responsible for publishing the web client's application state under the routable name prefix, obtained through the Internet Service Provider (e.g. Verizon). The name of the application state also includes automatically generated name component(s) which ensure its uniqueness across the web system for a short period of time. Lines 5 – 8 demonstrate that consumer context can be used to notify the web service "/com/google/mail/sign-in") about the application state ("/6758-6855-3857") and retrieve the server's response, which is processed in line 9 – 10.

The Algorithm 14 demonstrates how the web service processes the notification from the web client, fetches its application state and responds with a personalized content. A notification in this design is an Interest carrying "/com/google/mail/sign-

Algorithm 13 Web client publishes the application state and notifies the web server

1: $h_p \leftarrow$ **producer**("/verizon/residential/02145/")
2: **attach**(h_p)
3: *application state* $\leftarrow$ login, password, language, encoding, etc.
4: **produce**(h_p, "/6758-6855-3857", *application state*)

5: $h_c \leftarrow$ **consumer**("/com/google/mail/sign-in", RDR)
6: *name component* $\leftarrow$ "/verizon/residential/02145/6758-6855-3857"
7: **setcontextopt**(h_c, **ON_CONTENT**, *OnServerReply*)
8: **consume**(h_c, *name component*)

9: **function** OnServerReply(Consumer **h**, byte[] **content**)
10: Process the web service's reply and redirect to the next page
11: **end function**

in/<DataLocatorName>" name.

Lines 1 – 3 set the producer context, which is responsible for processing new notifications from the users trying to sign in to their accounts. Lines 4 – 9 show how to use the consumer context with RDR protocol to fetch the application state from the web client by knowing the Data Locator name. Lines 10 – 15 are executed after the client's application state is successfully fetched to publish the response to the initial notification ("/com/google/mail/sign-in/<DataLocatorName>").

Algorithm 14 Web service receives the notification, fetches the application state of the client and publishes a response message

1: $h_p \leftarrow$ **producer**("/com/google/mail/sign-in")
2: **setcontextopt**(h_p, **CACHE_MISS**, *OnNotification*)
3: **attach**(h_p)

4: **function** OnNotification(Producer **h**, Interest **i**)
5: *application state name* $\leftarrow$ extract from i.name
6: $h_c \leftarrow$ **consumer**(*application state name*, RDR)
7: **setcontextopt**(h_c, **ON_CONTENT**, *OnApplicationState*)
8: **consume**(h_c, "")
9: **end function**

10: **function** OnApplicationState(Consumer **h**, byte[] **content**)
11: Process login, password, language, encoding, etc. from the **content**
12: *service reply* $\leftarrow$ web service reply status, redirection page, etc.
13: *unique request name* $\leftarrow$ **getcontextopt**(h, PREFIX)
14: **produce**(h_p, *unique request name*, *service reply*)
15: **end function**

CHAPTER 7

Related work

Over the years multiple efforts have attempted to adapt application level framing at the transport layer (Structured Streams [For07], HTTP 2.0 [htt]), or above the transport layer (Publish/Subscribe [EFG03]). However all these efforts are built over the existing TCP/IP protocol stack. A more recent effort (Named Data Socket [GGP14]) proposed to provide some ALF support over an NDN network through a modified socket system. PARC has recently proposed a CCNx Portal API with RTA (Ready-To-Assemble) and datagram protocols available through it. A few publish / subscribe systems have been proposed for NDN overlays, such as COPSS [CAJ11] and DDS-over-CDN-over-NDN [PWG12].

In this section, we provide a brief description of the above research directions and highlight their differences with the Consumer / Producer API which is specifically designed to work over NDN.

7.1 Structured streams

It has been well recognized that TCP's byte stream model does not match all applications' needs, while UDP's best effort datagram model leaves too much work to applications. Structured Stream Transport (SST) enhances the traditional stream abstraction with a hierarchical hereditary structure, allowing applications to create lightweight child streams from any existing stream [For07]. Unlike TCP, these lightweight streams offer independent data transfer and flow control for each

stream, allowing different transactions to proceed in parallel without head-of-line blocking, but sharing one congestion control context. SST supports both reliable and best-effort delivery in a way that semantically unifies datagrams with streams and solves the classic "large datagram" problem.

Primitives	**create_substream** (stream) ➔ new_stream **listen_substream** (stream) **accept_substream** (stream) ➔ new_stream

Table 7.1: API for working with structured streams.

HTTP 2.0 proposal addresses similar issues by optimizing the mapping of HTTP's semantics to an underlying stream [htt]. Its key features include: 1) multiplexing of HTTP requests over a single connection, allowing concurrent HTTP requests/responses, and 2) prioritization of the requests, providing the ability to indicate which HTTP request is more important than others, and therefore avoid head-of-line blocking.

SCTP [Ste] and DCCP [FHK06] have been designed to preserve the initial structure of application data units and provide reliable / unreliable and ordered / unordered transmission services.

However all protocols mentioned above are confined to IP's point-to-point packet delivery, and the application data units are invisible at the network layer. Consequently their data priority only has the effect at the end-to-end level, their scalability (for web service) must rely on other means to address, and their requirement of the direct connectivity between client and server makes them infeasible in mobile and delay tolerant scenarios.

7.2 Publish / Subscribe

Publish / subscribe communication offers multi-point non-host-based addressing: topic-based, content-based, and type-based [EFG03]. Subscribers register their

interest in events by calling a *subscribe()* operation on the event service, without knowing the effective sources of these events. This subscription information remains stored in the event service and is not forwarded to publishers. The symmetric operation *unsubscribe()* terminates a subscription. Event-based nature of this interaction leads to time decoupling between subscribers and publishers. To generate an event, a publisher typically calls a *publish()* operation. The event service propagates the event to all relevant subscribers. Publishers also often have the ability to advertise the nature of their future events through an *advertise()* operation.

Primitives	**subscribe** (expression, timeout) ➔ s_handle **unsubscribe** (s_handle) **advertise** (expression, timeout) ➔ p_handle **unadvertise** (p_handle) **publish** (event)

Table 7.2: Common API primitives for publishing and subscribing to the events.

Publish / subscribe communication work with application data units, but is different from the consumer / producer communication in some important ways. First, the majority of publish / subscribe systems run on top of today's point-to-point transport layer (e.g. TCP, SCTP), which provides reliable delivery and segmentation. The rendezvous point (e.g. event service) between publishers and subscribers raise concerns about single point of failure and system scalability. Other concerns include the feasibility of supporting realtime, on-demand dynamic data production, due to the additional latency caused by the introduction of the event service.

Second, for the few publish / subscribe systems capable of running on top of Named Data Networking, their designs are not centered on the data directly. COPSS [CAJ11] introduces a push-based delivery mechanism using multicast in a content centric framework. At the content centric forwarding layer, COPSS

uses a multiple-sender, multiple-receiver multicast capability with the use of Rendezvous Points (RP). DDS-over-CDN-over-NDN [PWG12] offers a push-based delivery over simplified Content Delivery Network (sCDN). When DDS has created subscriber and publisher entities, sCDN is invoked to send a subscription message from the subscriber. This message is flooded through the network to look for an appropriate publisher. When a publisher is found, the requested content objects are forwarded to the subscriber by following the appropriate directed acyclic graph (DAG) in a hop-by-hop, reliable store-and-forward manner.

7.3 Named Networking Socket

Named Networking Socket is an implementation of the process-to-content (PCC) communication model [GGP14]. The design extends Unix implementation of the BSD socket with a novel Named Networking domain, which implies a layered architecture with distinctive network, transport and application layers. The API does not perform conversion of application data unit (ADU) to transmission units. As a result, the application cannot use ADUs that exceed the MTU. NaNet socket provides a datagram ADU (single-segment) and reliable byte-stream content retrieval mechanisms.

Initialization	**socket** (domain, type, protocol) ➔ handle
Primitives	**bind** (handle, address) **listen** (handle, backlog) **connect** (handle, address) **sendto** (handle, buffer, destination) **receivefrom** (handle, buffer, source) **setsocktopt** (handle, option name, value) **getsockopt** (handle, option name) **close** (handle)

Table 7.3: NaNET primitives for consuming and publishing data in NDN network.

7.4 CCNx Portal

CCNx Portal API is a programmatic mechanism to create, maintain and use multiple CCN Transport Stacks, each associated with a specific Portal instance (Table 7.4). At the present moment, the provided transports are: RTA (Ready-To-Assemble) and Datagram protocols. RTA uses the chunking protocol [ccn15] to split the user data into CCNx Content Objects.

Initialization	**createPortal**(protocol, attributes) ➔ handle
Primitives	**listen** (handle, prefix) **ignore**(handle, prefix) **send** (handle, message) **receive** (handle) **isError**(handle) **getError**(handle) **isEOF**(handle) **setAttributes**(handle, attributes) **getAttributes**(handle) **release**(handle)

Table 7.4: CCNx Portal API primitives.

The data transfer starts when the application creates an Interest and puts it in the **send()** operation (an example of the consumer in Algorithm 15). In the case, if the Interest name contains a segment number component, the data retrieval starts from that segment. In the case, if the Interest name does not contain a segment number component, the data retrieval starts from the segment zero. During the data retrieval, if the portal receives a new Interest with the same prefix and a different segment number, the sliding Interest window is re-adjusted to start from the specified segment. **send()** operation also accepts Data packets and other control messages such as InterestReturn message that is somewhat similar to the network level NACK in NDN.

It is possible to cancel the data retrieval in several ways:

1. Close the connection via the **close()** API primitive. This will cancel all in progress sessions and drop any undelivered Data packets.

2. Use **send()** API primitive to put a control message with the name prefix which data retrieval to be canceled.

Retrieved Data packets are passed up to the application in-order, and are dropped if they arrive outside of the sliding Interest window. Control messages and Interest packets are passed up to the application unmodified (an example of the producer in Algorithm 16).

Algorithm 15 HelloWorld consumer

```
1: h ← createPortal(RTA, Blocking)
2: name ← "/Hello/World"
3: interest ← createSimple(name)
4: message ← createFromInterest(interest)
5: send(h, message)
6: while isError(h) = false do
7:     response ← receive(h)
8:     contentObject ← getContentObject(response)
9:     display contentObject
10: end while
```

Portal API has built-in content verification capability. An application might use **setAttributes()** API primitive to add new items to the lists of trusted certificates, issuers and keys, which are later used for the verification of retrieved Data packets.

7.4.1 RTA protocol

In order to map TCP-style protocol machinery onto NDN semantics Interest packets are treated as TCP acknowledgements, because they give the sender a permis-

Algorithm 16 HelloWorld publisher

```
1: h ← createPortal(RTA, Blocking)
2: listenName ← "/Hello"
3: contentName ← "/Hello/World"
4: listen(h, listenName)
5: while true do
6:     request ← receive(h)
7:     interestName ← request.name
8:     if interestName = contentName then
9:         contentObject ← createContentObject(contentName, payload)
10:        message ← createFromContentObject(contentObject)
11:        send(h, message)
12:    end if
13: end while
```

sion to send data in the network. Therefore, the node that is expressing Interests it responsible for Interest pacing to avoid exceeding the network capacity. In RTA this is achieved by monitoring the delay of returning Data packets and backing off in the case if the delay grows quickly. If the delay is in the range of expected values, RTA increases the sliding Interest window size linearly. RTA has a slow start phase, during which it doubles Interest window size every other RTT until it reached the slow start threshold or the delay of data retrieval increases too much.

RTA uses the standard RFC6298 Retransmission Timeout (RTO) calculation methods per flow control session. RTA flow corresponds to the user object's prefix.

7.5 Extensible API

Some research projects propose even higher level API abstractions, which would allow application to express its communication semantics to the network stack

and get the desired style of communication in return. NetAPI is an interface that decouples applications from network mechanisms to foster innovation below the API [DFK07, AHZ09]. NetAPI hides implementation mechanisms from the application and captures application intent to let the network stack understand application requirements.

Primitives	**open** (scheme://resource, options) ➔ handle **get** (handle, options) ➔ message **put** (handle, message, options) **control** (handle, options) ➔ result **listen** (handle) **accept** (handle) **close** (handle)

Table 7.5: NetAPI primitives.

The user starts a connection through the **open()** call, which returns a connection handle. NetAPI takes a Uniform Resource Identifier (URI) of the form *scheme://scheme-specific-part*. The scheme selects one of a number of communication schemes, such as *web://*, *video://*, or *voice://*. The scheme defines what types of messages and options can be used in API operations and how they are interpreted.

The **put()** and **get()** operations send and receive messages over the connection. NetAPI messages are application-defined data units (ADUs), such as video frames in a video scheme. They consist of data plus a list of key-value properties. This lets the scheme implementation distinguish between messages types and understand the semantics of each message.

The **control()** function is used for scheme-specific control operations, such as seeking in a streaming media scheme. It takes an options argument, which is a list of key-value pairs interpreted by the scheme.

CHAPTER 8

Conclusions

The seminal paper [CT90], published 25 years ago, clearly articulated the value of applying the concept of application level framing to network protocol development by directly using application data unit (ADU). [FJL97] further demonstrated that communicating by ADUs is particularly valuable in building many-to-many distributed applications. However because the work done in [FJL97] was built upon the existing IP protocol stack where the network layer had no concept of data, the authors used IP multicast group, enhanced with various tweaks, to get packets to the interested nodes.

In today's Internet, network and transport layers are completely decoupled from application layers in namespace, because each layer has its own namespace (e.g. address and port versus application data names), and in timing, because socket simply gets a virtual channel ready, but the application decides when packets are actually sent. This insulation makes it easy to design each part on its own, however when multiple layers are put together, they often do not work most coherently. NDN's direct use of application names at network layer removed the insulation, which opens new potential for developing an overall cohesive system where applications can make the best use from the network transport.

NDN is able to support application level framing throughout the network, and Consumer / Producer API makes it easy for applications to publish and retrieve application frames from the network. The API was designed iteratively by evaluating new functionality through application development and going back

to the design phase. The Internet radio streaming application demonstrated the need for easy to use sequential frame fetching pattern, the static video streaming demonstrated the need for parallel fetching and publishing into the storage, the live video streaming needed a fast way of signing the video frames at the live rate. The analysis of the feasibility of running a web-style RESTful applications proved the need for the efficient bidirectional Consumer / Producer interaction, because the modern web-services depend on the user context pushed by the web-clients. The present work provides an insight into the design of NDN-based web clients and servers that use Consumer / Producer API to communicate.

Our experience with several pilot applications proved that Consumer / Producer API benefits application developers in terms of ease of development and functionality. The Consumer / Producer API is still at its early development stage, and we would like to invite others to experiment with it and help further improve its functionality.

We anticipate that the designed Consumer / Producer model will continue to evolve to further accommodate the needs of NDN applications. One of the most valuable additions to the model would be an ability to select a trust schema from the set of available trust models during the initialization of the consumer and producer contexts in a similar way as how the selection of the data retrieval protocol is done now. Since the trust schema defines the relationships between data names and keys, both consumer and producer applications would benefit from the automated partial name construction based on the selected schema.

Another large area for enhancement of the model is the integration with new data retrieval and segmentation protocols. The analysis of a few protocols such as InfoMax [Jon15] and NDN-RTC [Pet15] showed that it is possible to provide the wide range of protocol's functionality and configurability through the Consumer / Producer API by adding new context options to it, because the model itself does not put any strict limitations on the name construction operations performed by

the protocols. There is a high possibility that new protocols will take an advantage of publishing and traversing trees, lists and graphs of manifests.

Versioning models represent an important research topic. Since NDN data is immutable, applications must manage their namespaces in a very consistent manner in order to avoid name collisions, which otherwise would introduce a lot of ambiguity in the operation of the applications and the network itself. Unfortunately, managing content versioning is not always easy as we demonstrated in the discussion of an application NACK versioning model. Questions appear almost instantaneously when an application developer starts working on the code that involves construction of the names which include versions. What version format to use? What is the minimal incremental value? How long and where to store already used versions that cannot be reused again? Consumer / Producer model can potentially offer multiple built-in versioning models to the application developers who need to publish the content, and offer the data retrieval protocols that semantically understand versioning and fetch the right content.

References

[AHZ09] Ganesh Ananthanarayanan, Kurtis Heimerl, Matei Zaharia, Michael Demmer, Teemu Koponen, Arsalan Tavakoli, Scott Shenker, and Ion Stoica. "A New Communications API." 2009.

[AMM13] Alexander Afanasyev, Priya Mahadevan, Ilya Moiseenko, Ersin Uzun, and Lixia Zhang. "Interest flooding attack and countermeasures in Named Data Networking." In *IFIP Networking Conference, 2013*, pp. 1–9. IEEE, 2013.

[ASZ14] Alexander Afanasyev, Junxiao Shi, Beichuan Zhang, Lixia Zhang, Ilya Moiseenko, Yingdi Yu, Wentao Shang, Yi Huang, J Paul Abraham, Steve DiBenedetto, et al. "NFD developer's guide." Technical report, Technical Report NDN-0021, NDN, 2014.

[AYW15] Alexander Afanasyev, Cheng Yi, Lan Wang, Beichuan Zhang, and Lixia Zhang. "SNAMP: Secure Namespace Mapping to Scale NDN Forwarding." In *Proceedings of 18th IEEE Global Internet Symposium (GI 2015)*, 2015.

[BDN12] Mark Baugher, Bruce Davie, Ashok Narayanan, and Dave Oran. "Self-verifying names for read-only named data." In *INFOCOM Workshops*, volume 12, pp. 274–279, 2012.

[BFF96] Tim Berners-Lee, Roy Fielding, and H Frystyk. "RFC 1945: Hypertext Transfer Protocol–HTTP/1.0." *The Internet Society*, 1996.

[BFM05] Tim Berners-Lee, Roy Fielding, and Larry Masinter. "RFC 3986: Uniform resource identifier (URI): Generic syntax." *The Internet Society*, 2005.

[BGN12] Jeff Burke, Paolo Gasti, Naveen Nathan, and Gene Tsudik. "Securing Instrumented Environments over Content-Centric Networking: the Case of Lighting Control." *arXiv preprint arXiv:1208.1336*, 2012.

[BHM12] J Burke, A Horn, and A Marianantoni. "Authenticated lighting control using Named Data Networking." *UCLA, NDN Technical Report NDN-0011*, 2012.

[BS04] Fred Baker and Pekka Savola. "RFC 3704: Ingress filtering for multihomed networks." Technical report, 2004.

[CAJ11] Jiachen Chen, Mayutan Arumaithurai, Lei Jiao, Xiaoming Fu, and KK Ramakrishnan. "Copss: An efficient content oriented publish/subscribe system." In *Architectures for Networking and Com-*

munications Systems (ANCS), 2011 Seventh ACM/IEEE Symposium on, pp. 99–110. IEEE, 2011.

[ccn14] "CCNx Face Management and Registration Protocol.", 2014.

[ccn15] "http://tools.ietf.org/html/draft-mosko-icnrg-ccnxchunking-00.", January 2015.

[CGT13] Mauro Conti, Paolo Gasti, and Marco Teoli. "A lightweight mechanism for detection of cache pollution attacks in Named Data Networking." *Computer Networks*, **57**(16):3178–3191, 2013.

[CSC14] Shuo Chen, Weiqi Shi, Junwei Cao, Alexander Afanasyev, and Lixia Zhang. "NDN Repo: An NDN Persistent Storage Model." 2014.

[CT90] D. D. Clark and D. L. Tennenhouse. "Architectural Considerations for a New Generation of Protocols." *SIGCOMM Comput. Commun. Rev.*, **20**(4):200–208, August 1990.

[D 12] D. Kulinski and J. Burke. "NDN Video: Live and Prerecorded Streaming over NDN." Technical report, UCLA, 2012.

[DFK07] Michael J Demmer, Kevin R Fall, Teemu Koponen, and Scott Shenker. "Towards a Modern Communications API." In *HotNets*, 2007.

[EFG03] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. "The many faces of publish/subscribe." *ACM Computing Surveys (CSUR)*, **35**(2):114–131, 2003.

[FGM99a] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. "Hypertext transfer protocol–HTTP/1.1.", 1999.

[FGM99b] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. "RFC 2616: Hypertext Transfer Protocol–HTTP/1.1." *The Internet Society*, 1999.

[FHK06] Sally Floyd, Mark Handley, and Eddie Kohler. "Datagram congestion control protocol (DCCP)." 2006.

[FJ93] Sally Floyd and Van Jacobson. "Random early detection gateways for congestion avoidance." *Networking, IEEE/ACM Transactions on*, **1**(4):397–413, 1993.

[FJL97] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. "A reliable multicast framework for light-weight sessions and application level framing." *IEEE/ACM Transactions on Networking (TON)*, **5**(6):784–803, 1997.

[FKS99] Wu-chang Feng, Dilip Kandlur, Debanjan Saha, and Kang Shin. "BLUE: A new class of active queue management algorithms." *Ann Arbor*, **1001**:48105, 1999.

[For07] Bryan Ford. "Structured streams: a new transport abstraction." In *ACM SIGCOMM Computer Communication Review*, volume 37, pp. 361–372. ACM, 2007.

[FT02] Roy T Fielding and Richard N Taylor. "Principled design of the modern Web architecture." *ACM TOIT*, **2**(2):115–150, 2002.

[GGP14] Massimo Gallo, Lin Gu, Diego Perino, and Matteo Varvello. "NaNET: socket API and protocol stack for process-to-content network communication." In *Proceedings of the 1st international conference on Information-centric networking*, pp. 185–186. ACM, 2014.

[Gri] Ilya Grigorik. "High Performance Networking in Google Chrome.".

[GTU13] Paolo Gasti, Gene Tsudik, Ersin Uzun, and Lixia Zhang. "DoS and DDoS in Named Data Networking." In *ICCCN 2013*, pp. 1–7. IEEE, 2013.

[GTU14] Cesar Ghali, Gene Tsudik, and Ersin Uzun. "Needle in a haystack: Mitigating content poisoning in named-data networking." In *Proceedings of NDSS Workshop on Security of Emerging Networking Technologies (SENT)*, 2014.

[htt] "https://tools.ietf.org/html/draft-ietf-httpbis-http2-04.".

[I 14] I. Moiseenko. "Fetching content in Named Data Networking with embedded manifests." Technical Report NDN-0025, NDN Technical Report, September 2014.

[JFL86] William Joy, Robert Fabry, Samuel Leffler, M McKusick, and Michael Karels. "Berkeley Software Architecture Manual 4.3 BSD Edition." *UNIX Programmer Supplementary Documents*, **1**:4–3, 1986.

[JM71] J.M. Winett. "RFC 147 - The Definition of a Socket." Technical report, 1971.

[Jon15] Jongdeog Lee, Akash Kapoor, Md Tanvir Amin, Zeyuan Zhang, Radhika Goyal, Zhehao Wang, Tarek Abdelzaher. "InfoMax: An Information Maximizing Transport Layer Protocol for Named Data Networks." In *Proceedings of the International Conference on Computer Communications and Networks (ICCCN'15)*, 2015.

[JST09] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. "Networking Named Content." In *Proc. of CoNEXT*, 2009.

[L 10] L. Zhang et al. "Named Data Networking (NDN) Project." Technical Report NDN-0001, October 2010.

[M] M. Hurton, I. Barber, P. Hintjens. "ZeroMQ Message Transport Protocol." http://rfc.zeromq.org/spec:23.

[MSO14] Ilya Moiseenko, Mark Stapp, and David Oran. "Communication patterns for web interaction in named data networking." In *Proceedings of the 1st international conference on Information-centric networking*, pp. 87–96. ACM, 2014.

[NJA13] Ben Newton, Kevin Jeffay, and Jay Aikat. "The Continued Evolution of Web Traffic." In *MASCOTS*, pp. 80–89. IEEE Computer Society, 2013.

[Pet15] Peter Gusev and Jeff Burke. "NDN-RTC: Real-time videoconferencing over Named Data Networking." In *Proceedings of the 2st international conference on Information-centric networking*, 2015.

[PNP13] Rong Pan, Prem Natarajan, Chiara Piglione, Mythili Suryanarayana Prabhu, Venkatachalam Subramanian, Fred Baker, and Bill VerSteeg. "PIE: A lightweight control scheme to address the bufferbloat problem." In *High Performance Switching and Routing (HPSR), 2013 IEEE 14th International Conference on*, pp. 148–155. IEEE, 2013.

[PWG12] Craig Partridge, Robert Walsh, Matthew Gillen, Gregory Lauer, John Lowry, W. Timothy Strayer, Derrick Kong, David Levin, Joseph Loyall, and Michael Paulitsch. "A Secure Content Network in Space." In *Proceedings of the Seventh ACM International Workshop on Challenged Networks*, CHANTS '12, pp. 43–50, New York, NY, USA, 2012. ACM.

[Ram] Sreeram Ramachandran. "Web Metrics: Size and Number of Resources.".

[RSC02] Jonathan Rosenberg, Henning Schulzrinne, Gonzalo Camarillo, Alan Johnston, Jon Peterson, Robert Sparks, Mark Handley, Eve Schooler, et al. "SIP: session initiation protocol." Technical report, RFC 3261, Internet Engineering Task Force, 2002.

[Ste] RR Stewart. "RFC 4960: Stream Control Transmission Protocol, IETF, Sep. 2007.".

[Sto11] Thomas Stockhammer. "Dynamic adaptive streaming over HTTP: standards and design principles." In *Proceedings of the second annual ACM conference on Multimedia systems*, pp. 133–144. ACM, 2011.

[SWD14] Wentao Shang, Zhe Wen, Qiuhan Ding, Alexander Afanasyev, and Lixia Zhang. "NDNFS: An NDN-friendly File System." 2014.

[Y 14] Y. Zhang, H. Zhang, and L. Zhang. "Kite: A Mobility Support Scheme for NDN." Technical report, NDN Project, 2014.

[YAM13] Cheng Yi, Alexander Afanasyev, Ilya Moiseenko, Lan Wang, Beichuan Zhang, and Lixia Zhang. "A Case for Stateful Forwarding Plane." *Comput. Commun.*, **36**(7):779–791, April 2013.

[ZA13] Zhenkai Zhu and Alexander Afanasyev. "Let's ChronoSync: Decentralized Dataset State Synchronization in Named Data Networking." In *Proceedings of ICNP*, 2013.

[ZAB14] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, K. Claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang. "Named Data Networking." *ACM SIGCOMM Computer Communication Review*, July 2014.