# Consumer / Producer Communication with Application Level Framing in Named Data Networking

Ilya Moiseenko
UCLA
iliamo@cs.ucla.edu

Lijing Wang
Tsinghua University
wanglj11@mails.tsinghua.edu.cn

Lixia Zhang
UCLA
lixia@cs.ucla.edu

## ABSTRACT

Named Data Networking (NDN) is a general purpose network layer protocol which offers a set of rich functionality: in-network storage, multi-path forwarding, multicast delivery, and data-centric security. Above the network layer, system libraries simplify application developers' tasks by providing an easy to use yet powerful API to utilize the functions enabled by NDN. This paper presents the design of a Consumer / Producer programming interface, together with several mechanisms, that supports application level framing via NDN's data retrieval protocols to make NDN application programming easier and faster.

## Categories and Subject Descriptors

C.2 [**COMPUTER-COMMUNICATION NETWORKS**]: Network Architecture and Design; Network Protocols; Distributed Systems

## Keywords

NDN; API; Transport; Data retrieval;

## 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 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 end processes [1, 2].

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. This fundamental change in its usage suggests that the Internet's universal network layer would be much more organic to use a data dissemination protocol that can natively work with information objects instead of communication endpoints.

As a newly proposed architecture to meet this new usage, Named Data Networking (NDN) replaces IP's host-based addressing scheme by names of information objects in moving packets through the network [3, 4, 5]. 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. NDN secures data directly with a publicly verifiable signature, and with encryption as needed (Section 2).

As explained in [6], 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.

In this paper we present the design of a new API and its associated protocol suite that can play socket-equivalent roles in an NDN network. Our contributions can be summarized as follows:

- A Consumer / Producer programming model, which is specifically tailored for data dissemination in NDN networks.
- Associated data retrieval and content segmentation protocols.
- A number of supporting mechanisms such as a manifest and negative acknowledgement, utilized by the protocols below the API to facilitate the operation of applications.

We have implemented the Consumer / Producer API and the protocols, and validated them by using the API to develop several pilot applications. The focus of our evaluation is to have running, correctly behaving, and real-world applications, and to quantitatively measure the computational overhead experienced by a single producer publishing ADUs for multiple consumers.

The rest of the paper is organized as follows. Section II gives a brief overview of NDN architecture. Section III contains a detailed explanation of the programming model and programming abstractions. Section IV and V provide a description of important concepts and design of content fetching protocols. Section VI presents the

*ICN'15,* September 30–October 2, 2015, San Francisco, CA, USA.

ACM 978-1-4503-3855-4/15/09 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810156.2810160 .

evaluation of the design through real-world applications. Related work can be found in Section VII, and conclusion in Section VIII.

## 2. BACKGROUND

NDN works in a fundamentally different way than IP. To help the reader easily grasp its core concepts, in this section we first describe a toy application example, then use this example to explain the NDN basics.

### 2.1 A Simple-Video Application

We use a *Simple-Video* application as an example to help illustrate some basic concepts behind applications in general and possible issues that must be addressed by an application developer. Simple-Video produces two separate streams of data, video and audio, to give the consumers a choice of either watching the video with audio, listen to the audio only, or watch the video in silence. Both video and audio streams consist of a stream of data frames, and the application should allow a consumer to retrieve individual frames independently. The consumer application can, for example, stop fetching audio frames when user hits the "Mute" button, or skip some video frames after a pause in order to catch up the actual live video. In general, a video frame is likely too large to be carried in a single network packet, thus the video producer application also needs to segment one frame into multiple packets.

One of the existing technologies, MPEG-DASH [7], 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 variety of ways applications can produce and fetch data at the level of application frames, there are repetitive and labour intensive tasks related to the segmentation of the application data frames and the retrieval of segments to reassemble an application frame. In the case of MPEG-DASH, all these low-level details are handled by the HTTP / TCP protocol machinery.

### 2.2 Named Data Networking

An NDN network has two types of packets: *Interest* and *Data*. 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. A Data name in NDN is supplied by the given application and has multiple components in general. It is used to retrieve the packet across network; it also contains application specific information to facilitate packet processing. As an illustrative example, Simple-Video uses the following naming schema. The data name begins with routable components *"/com/youtube/"* which guides all Interest packets carrying this name prefix toward the data producer. The next component is the name identifier of the media resource. The component after that separates video and audio frames into separate namespaces (i.e. name subtrees). Both video and audio frames are named sequentially. Each video frame may consist of multiple segments, also named sequentially, while each audio frame is made of a single segment.

Similar to IP, an NDN network provides datagram delivery. Data consumers who desire reliable data fetching need to use reliable fetching services; they may also need to regulate data flow through pacing Interest packets transmissions.

Data production can be either on demand, or independent from data consumption. The inherent asynchrony between producers and consumers creates delicate coordination issues in between, such as the availability of data, the specifics of data, *etc. Interest Selectors* is one of the available mechanisms to facilitate data fetching by *multiple* consumers, potentially from *multiple* producers. Any Interest may carry optional Selectors that specify additional conditions (besides name matching) for content retrieval. For example, when a consumer sends an Interest with a name prefix, and receives a Data packet $P$ that is not the desired one, it can try again by re-sending the Interest with an Exclude selector which may contain either the exact name of $P$ or $P$'s digest (e.g. hash of the packet).

### 2.3 NDN inside a Node

The NDN Forwarding Daemon (NFD) is a multiplexer between applications and network interfaces inside a node [8]; NFD makes no distinction between applications and network interfaces, and views them all as *Faces*. NFD has a content store for opportunistic caching of passing-by Data packets. NFD may also have a face to a local repository (e.g. Repo-NG [9]), which provides managed storage of Data packets.

To make data available, the 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 (e.g. forwarding table) and also forwards the prefix to the next router. Consumer-process does not perform any registration — it simply sends Interest packets to the local NFD which then forwards the Interests toward the producer, either locally or remotely.

## 3. PROGRAMMING MODEL

In this section, we define *consumer context* and *producer context* abstractions, together with the rationale behind our design.

### 3.1 Socket is inappropriate

We aim to design a programming abstraction that would give application developers adequate freedom when handling ADUs, at the same time minimize the complexity associated with the production and retrieval of ADUs of any size.

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 [10]) suitable for queuing and passing messages.

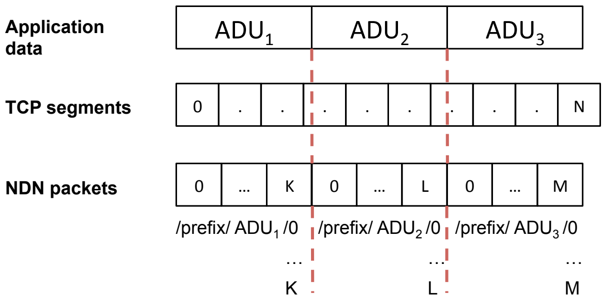

**Figure 1: TCP/IP segmentation does not preserve boundaries of application frames (ADUs). NDN segmentation exposes these boundaries through naming.**

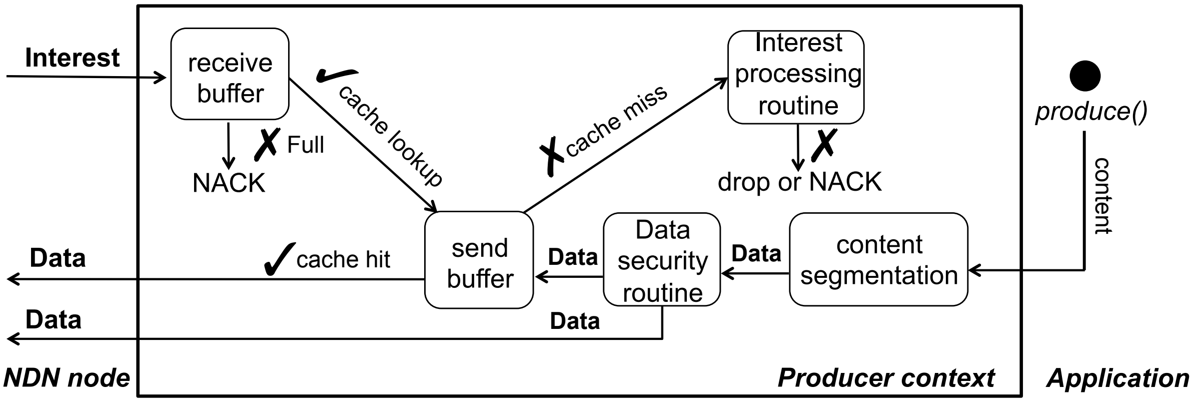

**Figure 2: Producer context can publish data with or without network connectivity.**

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.

## 3.2 Design goals

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 all application data as byte streams, TCP segmentation ignores ADU boundaries, thus ADUs can only be identified after the segment reassembly (Figure 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, 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.

### *3.2.1 Goals for the consumer abstraction*

In identifying the design goals for the consumer abstraction, we make an initial assumption that, generally speaking, individual applications would like to organize ADU fetching according to their own priorities. Therefore we describe the design goals 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 we believe that 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.

### *3.2.2 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. We have identified the following three application patterns to be supported at this point.

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.

## 3.3 Producer context

A **producer context** is used to publish data under a common prefix (Figure 4). It is initialized by calling **producer()** primitive with a given name prefix parameter. 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

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

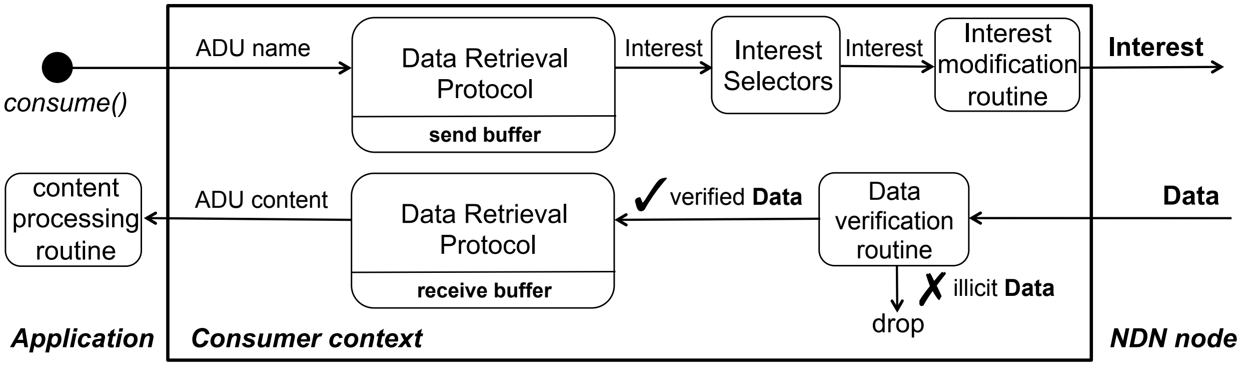

**Figure 3: Event-based processing of Interest and Data packets in the consumer context.**

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.

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 4.

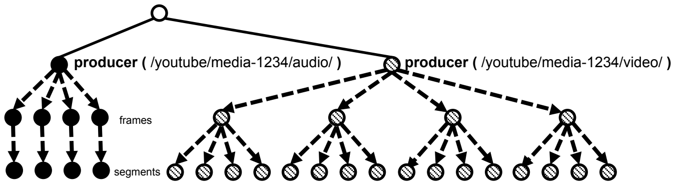
producer ( /youtube/media-1234/audio/ )
producer ( /youtube/media-1234/video/ )
frames
segments

**Figure 4: 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 2). 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 [9, 11].

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.

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 4.1), so that the consumer(s) can handle the situation in a most informed way.

## 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 5.

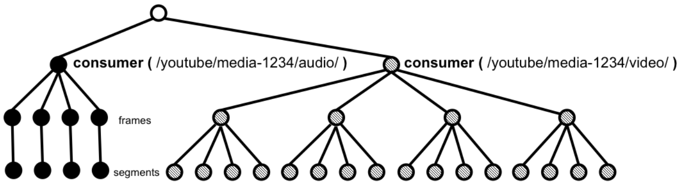
consumer ( /youtube/media-1234/audio/ )
consumer ( /youtube/media-1234/video/ )
frames
segments

**Figure 5: Consumer context is initialized with a name prefix defining the range of information objects that can be retrieved from the network.**

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 application, the name suffix is a frame number. Name suffix parameter allows application developer to reuse the same context for fetching multiple ADUs (Figure 5). Inside the context, the data retrieval protocol (Section 5) generates Interests and processes incoming Data packets with other related events (Figure 3).

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.

# 4. SUPPORTING MECHANISMS

To support efficient consumer / producer communication described in the previous sections, we introduce two new mechanisms: negative acknowledgements and manifests. This section talks about these mechanisms in more detail.

## 4.1 Negative acknowledgement

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 [12]. 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. We define a NACK 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 [13, 14]. 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 [15] 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. 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 we propose to set the FreshnessPeriod of a NACK to be within a small fraction (10%) of the application specified retry timer.

## 4.2 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 [16], 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 [16], manifests carrying the catalogue of names need to be fetched before data fetching, which introduces an additional round-trip latency.

We propose to embed a manifest as an ADU in the same sequence with Data packets to eliminate the undesirable latency from fetching manifest [17]. 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 6.

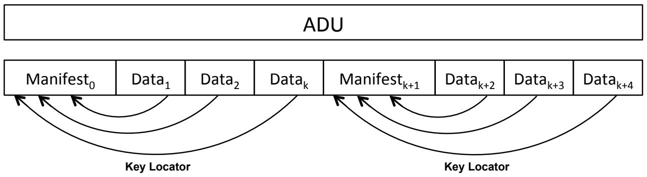

**Figure 6: Manifests are embedded in the sequence of data packets when application data is being segmented.**

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 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 SimpleVideo application uses the last ADU name to understand where the video ends (e.g. frame #2500).

# 5. DATA RETRIEVAL PROTOCOLS

Based on our experience from developing NDN applications, we have designed an initial set of data retrieval protocols: Simple Data Retrieval (SDR), Unreliable Data Retrieval (UDR), and Reliable Data Retrieval (RDR).

## 5.1 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.

## 5.2 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)*.[3] 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 windows 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 ($\leq$ 15) of Data packets: UDR can fetch such small ADUs in two RTTs and avoid bursty transmission for much larger ADUs (e.g. hundreds of Data packets).

UDR's best-effort Internet 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).[4] 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 deletes Data packets that fail data verifications. 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.

## 5.3 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 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.

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

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

[4] 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.

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 [18], and [19] 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 7 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 4.1. The traditional mechanism of TCP window size advertisement for flow control purpose is not applicable in NDN, given the absence of a "connection" between consumers and producers.

Congestion is controlled at the NDN forwarding plane by utilizing Interest NACK mechanism [20]. Note that Interest NACK is different from the Data NACK proposed in this paper — it is a network layer packet and its main purpose is to assist NDN routers in performing quick and informed recovery from network problems, such as prefix hijacks, link failures, and network congestion. The NFD running in the local node is expected to handle Interest NACK, perform congestion control, and enforce fairness among multiple consumer applications running on the same node. It may also be beneficial to further propagate Interest NACKs up to the data retrieval protocols, an issue that we plan to investigate.

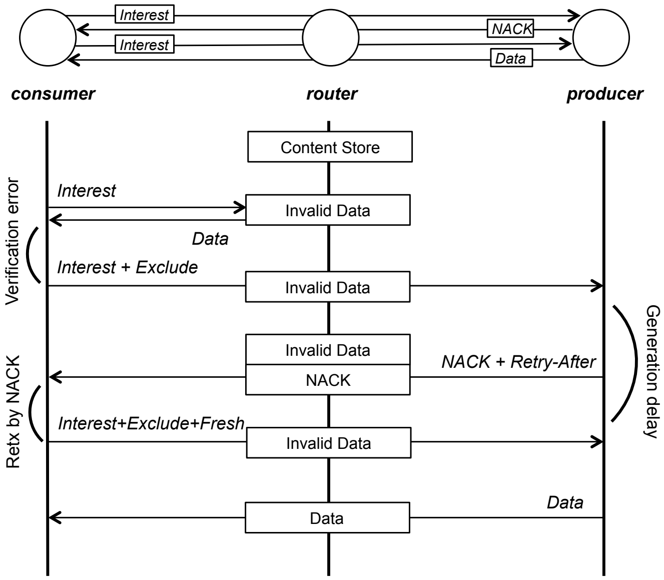

**Figure 7: RDR recovers from the Data verification error and handles dynamic data generation delay.**

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 4.2), 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.

# 6. EVALUATION

We took an application-driven approach to guide the design and development of the Consumer / Producer API. This section reports on our experience from implementing and using the Consumer / Producer API in real environments, and some preliminary analysis of the space-computation tradeoff at the producer side.

## 6.1 Implementation

The Consumer / Producer API is implemented as a user-space library written in C++, which runs on Mac OS X, Linux, and BSD. The library is a branch of ndn-cxx 0.3 library [21], and is currently available at *https://github.com/iliamo/Consumer-Producer-API*.

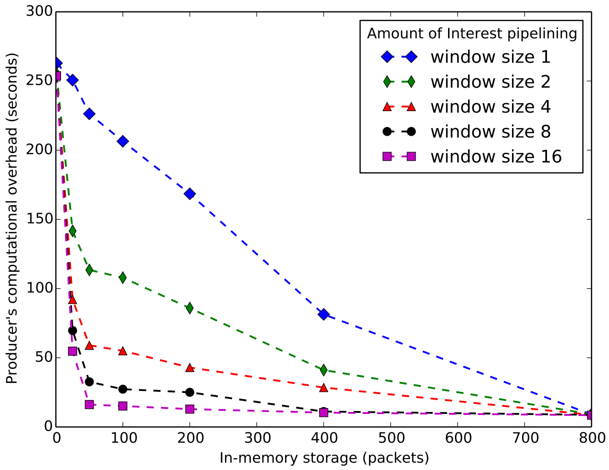

**Figure 8: Benefit of having producer's send buffer for serving multi-packet ADUs to multiple consumers that pipeline Interests.**

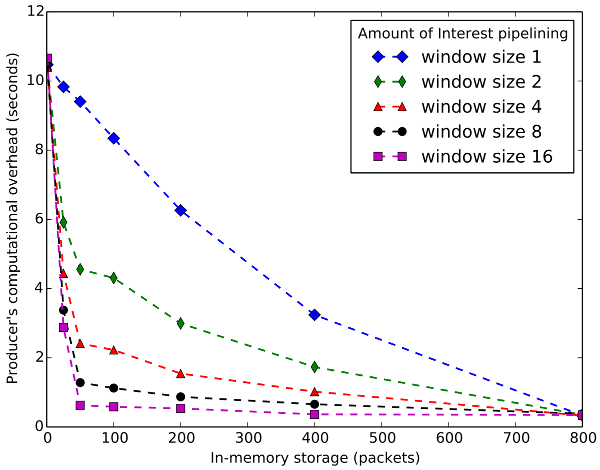

**Figure 9: Benefit of embedding manifests in the multi-packet ADUs fetched by multiple consumers that pipeline Interests.**

The pilot applications we developed over this new API include audio and video distributions, and sensing data collection. Our experience shows that the biggest benefit of this new API is a significant reduction in application development efforts as well as the complexity of the resulting implementation, as compared to developing applications directly on top of NDN's raw Interest / Data exchange API.

The first application we tried is NDNtube which distributes static video contents. NDNtube is an example of applications with significant time asynchrony between data publishing and consumption [22]. Video and audio frames are published once to permanent storage, and served to the consumers from storage using the RDR protocol. A similar application, NDNvideo [23], was used in a few large scale demonstrations earlier and showed that, in contrast to applications based on HTTP/TCP stack, an NDN network performs scalable content distributions without any special configuration [24].

The next application, NDNlive, is similar to NDNtube but supports live TV broadcasting instead of streaming static contents. Yet another application is NDNradio, which prototypes the iTunes radio application. It is worth mentioning that NDNradio was developed by a student without much programming experience, when she first started the project with NDN's raw Interest / Data API, she could not make much progress, and switching to the new API enabled her to successfully finished the implementation. Both NDNlive and NDNradio are examples of realtime publishing / consumption applications. Since real time data lose their value shortly after publication, these Data packets are served from short term in-memory storage — the send buffer of the producer.

Our latest application prototype developed on top of the new API is home sensor data collections. It periodically wakes up from sleep to collect new measurement samples and can also produce data on demand. If the requested data is not available yet, it uses *nack()* with *Retry-After* code to inform the consumer about the future availability of the data (e.g. in 10 seconds) and then enters the sleep mode during this time period.

## 6.2 Space-Computation Tradeoff Analysis

Compared to the well tuned TCP/IP implementations, we expect the performance of the Consumer / Producer API and its protocols to be largely comparable, but likely lower due to several factors: the larger NDN header overhead, the untuned implementation of the NFD and the library, as well as the cost from packet signing and verification. However in those applications where multiple users request the same data, the producer API can potentially perform significantly better than the server side socket, because every ADU (e.g. its Data packets) can serve multiple consumers. More specifically, the producer API has two mechanisms that help speed up ADU publishing: an in-memory cache of Data packets (send buffer) and an optional technique of manifest embedding in the sequence of Data packets.

Unlike a TCP socket's send buffer, which stores the segments that belong to a single connection, a producer's send buffer stores Data packets that share some common name prefix. However given the producer's send buffer has a finite size, newly produced data packet may push out the packets that were produced earlier. If those evicted packets are requested again, the producer has to generate them again. Therefore the publishing cost would be lower if the send buffer is large in size, allowing more ADUs to be kept in the memory, and would be higher if the send buffer is small, which causes more "ADU republishing events".

In a controlled experiment, we modeled the behavior of:

- multiple basic web consumers requesting 20 different personalized web resources (ADUs) in a 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; and
- a basic web server serving the above consumers with personalized html pages and other dynamic content. Each ADU consists of 30 Data packets.

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 larger Interest window, fetch more Data packets at once, leading to fewer "ADU republishing events". Figure 8 demonstrates that in the absence of the send buffer, multiple consumers that pipeline Interest packets lead to significant load (e.g. signing and segmentation) at the producer, while the larger buffer effectively amortizes publishing cost across multiple consumers.

In the experiment the send buffer size is set to [0, 25, 50, 100, 200, 400, 800] of Data packets, and the results are shown in Figure 8. It shows that without the send buffer, multiple consumers, each sending many Interests, are able to overload the producer due to the signing and segmentation overhead. A larger buffer can effectively amortize these same costs across multiple consumers.

We ran the same experiment 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 9. Both experiments were conducted on the Mac OS X platform with trusted platform module (Mac OS X Keychain) used to produce RSA signatures with SHA256 digest.

# 7. RELATED WORK

Over the years multiple efforts have attempted to adapt application level framing at the transport layer (Structured Streams [25], HTTP 2.0 [26]), or above the transport layer (Publish/Subscribe [27]). However all these efforts are built over the existing TCP/IP protocol stack. A more recent effort (Named Data Socket [28]) proposed to provide some ALF support over an NDN network through a modified socket system. A few publish / subscribe systems have been proposed for NDN overlays, such as COPSS [29] and DDS-over-CDN-over-NDN [30]. 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 & HTTP 2.0

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 [25]. 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.

HTTP 2.0 proposal addresses similar issues by optimizing the mapping of HTTP's semantics to an underlying stream [26]. 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.

However both SST and HTTP 2.0 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 [27]. 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.

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 [29] 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 [30] 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 [28]. 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. The assumption is that an ADU corresponds to a content segment and defines the granularity for which the application can support out-of-order packets and recovery from packet losses. Therefore, the publisher-process is in charge of defining the proper ADU size based on application constraints. NaNet socket provides a datagram ADU (single-segment) and reliable byte-stream content retrieval mechanisms.

# 8. CONCLUSION

The seminal paper [6], 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). [31] further demonstrated that communicating by ADUs is particularly valuable in building many-to-many distributed applications. However because the work done in [31] 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. 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.

## 9. REFERENCES

[1] J.M. Winett, "RFC 147 - The Definition of a Socket," Tech. Rep., 1971.

[2] W. Joy, R. Fabry, S. Leffler, M. McKusick, and M. Karels, "Berkeley Software Architecture Manual 4.3 BSD Edition," *UNIX Programmer Supplementary Documents*, vol. 1, pp. 4–3, 1986.

[3] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking Named Content," in *Proc. of CoNEXT*, 2009.

[4] L. Zhang et al., "Named Data Networking (NDN) Project," Tech. Rep. NDN-0001, October 2010.

[5] 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.

[6] D. D. Clark and D. L. Tennenhouse, "Architectural Considerations for a New Generation of Protocols," *SIGCOMM Comput. Commun. Rev.*, vol. 20, no. 4, pp. 200–208, Aug. 1990.

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

[8] A. Afanasyev, J. Shi, B. Zhang, L. Zhang, I. Moiseenko, Y. Yu, W. Shang, Y. Huang, J. P. Abraham, S. DiBenedetto *et al.*, "NFD developer's guide," NDN Tech. Report NDN-0021, 2014.

[9] S. Chen, W. Shi, J. Cao, A. Afanasyev, and L. Zhang, "NDN Repo: An NDN Persistent Storage Model," 2014.

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

[11] W. Shang, Z. Wen, Q. Ding, A. Afanasyev, and L. Zhang, "NDNFS: An NDN-friendly File System," 2014.

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

[13] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol–HTTP/1.1," 1999.

[14] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, E. Schooler *et al.*, "SIP: session initiation protocol," RFC 3261, Internet Engineering Task Force, Tech. Rep., 2002.

[15] A. Afanasyev, P. Mahadevan, I. Moiseenko, E. Uzun, and L. Zhang, "Interest flooding attack and countermeasures in named data networking," in *IFIP Networking Conference, 2013*. IEEE, 2013, pp. 1–9.

[16] M. Baugher, B. Davie, A. Narayanan, and D. Oran, "Self-verifying names for read-only named data." in *INFOCOM Workshops*, vol. 12, 2012, pp. 274–279.

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

[18] M. Conti, P. Gasti, and M. Teoli, "A lightweight mechanism for detection of cache pollution attacks in named data networking," *Computer Networks*, vol. 57, no. 16, pp. 3178–3191, 2013.

[19] C. Ghali, G. Tsudik, and E. 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.

[20] C. Yi, A. Afanasyev, I. Moiseenko, L. Wang, B. Zhang, and L. Zhang, "A Case for Stateful Forwarding Plane," *Comput. Commun.*, vol. 36, no. 7, pp. 779–791, Apr. 2013.

[21] http://named-data.net/doc/ndn-cxx/0.3.0/.

[22] L. Wang, I. Moiseenko, and L. Zhang, "NDNlive and NDNtube: Live and Prerecorded Video Streaming over NDN," NDN, Tech. Rep. 31, May 2015.

[23] D. Kulinski and J. Burke, "NDN Video: Live and Prerecorded Streaming over NDN," UCLA, Tech. Rep., 2012.

[24] P. Crowley, J. DeHart, and H. Yuan, "Performance in Performance in Named Data Networking," 2013 FIA PI Meeting, November 2013. [Online]. Available: http://named-data.net/wp-content/uploads/2014/05/FIA-2013-NDN-Perf-11-15-2013.pdf

[25] B. Ford, "Structured streams: a new transport abstraction," in *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4. ACM, 2007, pp. 361–372.

[26] https://tools.ietf.org/html/draft-ietf-httpbis-http2.

[27] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.

[28] M. Gallo, L. Gu, D. Perino, and M. Varvello, "NaNET: socket API and protocol stack for process-to-content network communication," in *Proceedings of the 1st international conference on Information-centric networking*. ACM, 2014, pp. 185–186.

[29] J. Chen, M. Arumaithurai, L. Jiao, X. Fu, and K. Ramakrishnan, "Copss: An efficient content oriented publish/subscribe system," in *Architectures for Networking and Communications Systems (ANCS), 2011 Seventh ACM/IEEE Symposium on*. IEEE, 2011, pp. 99–110.

[30] C. Partridge, R. Walsh, M. Gillen, G. Lauer, J. Lowry, W. T. Strayer, D. Kong, D. Levin, J. Loyall, and M. Paulitsch, "A secure content network in space," in *Proceedings of the Seventh ACM International Workshop on Challenged Networks*, ser. CHANTS '12. New York, NY, USA: ACM, 2012, pp. 43–50.

[31] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang, "A reliable multicast framework for light-weight sessions and application level framing," *IEEE/ACM Transactions on Networking (TON)*, vol. 5, no. 6, pp. 784–803, 1997.