NDN, Technical Report NDN-0017, 2014. http://named-data.net/techreports.html
Revision 2: February 14, 2014
Revision 3: August 25, 2014

# Consumer-Producer API for Named Data Networking

Ilya Moiseenko
UCLA
iliamo@cs.ucla.edu

Lixia Zhang
UCLA
lixia@cs.ucla.edu

## ABSTRACT

This paper presents Consumer-Producer integrated processing API that provides a generic programming interface to NDN communication protocols and architectural modules. A consumer context associates an NDN name prefix with consumer-specific data transfer parameters, controls Interest transmission and Data packet processing. A producer context associates an NDN name prefix with producer-specific data transfer parameters that control Data packet production and security, and Interests demultiplexing. Both API contexts are extensible to admit the functionality of future protocols and modules.

## 1. INTRODUCTION

Today's Internet architecture stands on IP — a universal network layer designed to create a communication network where packets are delivered to specific destinations. One of the primary design goals of IP was direct process-to-process communication. This was a premise for introducing the concept of the port, which binds a running process to a communication channel [1]. IP address, port, and transport protocol are bound together by the socket, representing a container for the current state of data transfer between IP endpoints.

The Internet has evolved from a network that interconnects hosts to a network that interconnects information objects. This 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.

Named Data Networking replaces the host-based addressing scheme and uses names of information objects to move packets in the network [2], [3]. Instead of sending packets to a given IP address and port, in NDN end hosts request desired data by sending Interest packets carrying application-level names of information objects. The NDN network returns the requested Data packets following the path of the Interests. In fetching Data, NDN treats channel and storage equally. The source could be as close as the cache of the same-hop or next-hop node, and as far as the original process-producer of the information. At the same time this design strengthens information safety with the concept of content-based security. The data itself is secured with publicly verifiable signatures and encryption, instead of relying on secure communication channels (Section 2).

As a new architecture, NDN requires an API that is totally different from IP's socket API. Socket abstraction cannot be reused, because it is founded on the concepts of an end-to-end virtual channel and a port that do not exist in NDN architecture. How a brand new API for NDN should look like is an open question. We advance NDN architecture by going through the cycles of 1) design 2) trial through pilot applications 3) revise, aiming to get it right after a number of cycles.

Motivated by our experience with building NDN applications, this paper proposes:

1. The notion of the *consumer context*, which associates an NDN name prefix with various data fetching, transmission, and content verification parameters, and integrates processing of Interest and Data packets on the consumer side (Sections 3 and 4).
2. The notion of the *producer context*, which associates an NDN name prefix with various packet framing, caching, content-based security, and namespace registration parameters, and integrates processing of Interest and Data packets on the producer side (Sections 3 and 4).
3. A Consumer-Producer API declaration for a range of common programming languages (Sections 4 and 5).
4. A description of API functionality and inner organization (Section 4).

## 2. NAMED DATA NETWORKING

An NDN network works with two distinct types of packets: *Interest* and *Data* (Figure 1). Both types of packets carry a *name*, which uniquely identifies an information object packaged in a single Data packet. Names in NDN are supplied by applications, and contain distinct components. Large information objects that cannot be carried in a single Data packet are segmented into multiple packets; the segment number is carried as a component at the end of the name: */com/foo/data/object/1*. Applications are expected to be designed with the appropriate naming scheme for the kind of communication they require.

To retrieve data, a consumer requests it by sending an Interest packet containing the name of the desired content. An NDN node uses this name and its Forwarding Information

Base (FIB) to forward the Interest towards likely sources. A Data packet whose name matches the name in the Interest is returned to the consumer by following the reverse path of the Interest using the information present in Pending Interest Table (PIT). PIT entries keep a record of recently forwarded Interest packets until the Data packets are returned. Each PIT entry contains an Interest name, the incoming interface(s) where the Interest has arrived, and the outgoing interface(s) to which the Interest has been forwarded.

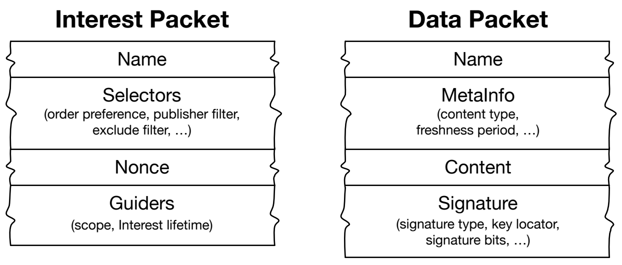
Interest Packet
Name
Selectors
(order preference, publisher filter, exclude filter, ...)
Nonce
Guiders
(scope, Interest lifetime)
Data Packet
Name
MetaInfo
(content type, freshness period, ...)
Content
Signature
(signature type, key locator, signature bits, ...)

**Figure 1: NDN packet types.**

NDN nodes can cache any passing Data packet in a special memory buffer called Content Store (CS). Later, a cached Data packet can be used to satisfy an Interest. Consumers do not care whether a Data packet was served from the cache or the original process-producer, because they can validate the data by the signature.

An NDN network does not secure a communication channel; instead the process-producer of the information appends a cryptographic signature to bind the name to the content, and encrypts the payload of its outgoing Data packets. Later, consumers can verify the signature of each received Data packet wherever it has come from. This security model makes possible to decouple the trust in data from the place and time the data was obtained.

NDN network is similar to IP in the fact that it does not provide any guarantees regarding reliable packet delivery or packet ordering. It is the responsibility of applications and communication protocols built on top of NDN to provide services with additional transmission guarantees whenever it is needed.

## 3. API DESIGN GUIDELINES

The IP socket abstraction is a container for data transfer parameters holding the current state of transmission in a virtual channel. With some minor peculiarities, both server and client applications use sockets in the same way.

In this section, we explain why NDN needs a different abstraction, what functionality is required, and describe the basic idea behind the new programming model.

### 3.1 Functionality

**Name construction** is the essential part of NDN Interest/Data exchanges. In some cases, name manipulation may become labour-intensive. Consider IP transport layer protocols such as RTP that use timestamps to synchronize participants and sequence numbers to detect packet losses and reordering [4]. In NDN, timestamps and sequence numbers are expressed on the network layer as name components of Interest and Data packets. A programmer-friendly API will partially automate name construction and manipulation according to the internal logic of NDN communication protocols.

**Retransmission** includes both the actual protocol machinery (timers, window management) and buffering of the packets. Socket API provides a programming interface for an ordered reliable stream (TCP). Our experience with the development of NDN applications demonstrates that NDN API needs to have an interface for both an ordered reliable stream and a reliable datagram services.

**Reassembly** is an important aspect of TCP/IP socket API, because it provides a way for application designers to directly access the contents of the stream without dealing with packet headers. Reassembly is a mandatory service for NDN API, but we also recognize a need for having a programming interface for out-of-order packet processing similar to the raw IP sockets.

**Introspection** — set of instrumentation and monitoring techniques. TCP/IP socket produces numeric error codes that often have rather generic meaning which prevents application designer from understanding what exactly is going on inside the TCP connection. In NDN, there are multiple ways of fetching Data objects via Interest selectors, versioning, etc. and, therefore, a more fine-grained monitoring techniques are needed.

**Segmentation** is offered in the socket API via the *write()* primitive. This primitive converts a user-space buffer with the content to a necessary number of TCP segments. A *data packaging* service similar to TCP/IP segmentation will be highly beneficial for designers of NDN applications as well.

**Attachment to the network** is offered in the socket API via *bind()* primitive. Once socket is *bound*, an operating system is able to demultiplex packets for it. In NDN, packets are demultiplexed based on hierarchical NDN names. In case producer application wants its Data packets to be globally routed, it will most likely have to acquire a routable prefix from an NDN point of presence (PoP) node. The task of the API is to offer an interface to work with prefix discovery/registration protocol.

### 3.2 Core concepts

In NDN, name engineering is a crucial stage of application development, because the network fetches the data exclusively by its name. This also means that during the data fetching process, various data transfer parameters are the properties of the namespace — unlike the point-to-point IP network, where data transfer parameters are the properties of the channel between IP endpoints.

NDN applications that consume data are naturally different from NDN applications that produce data. As a result, they have a different set of data transfer parameters. Often NDN consumers care about fetching multiple segments of data, reliable data delivery, verifying the data, processing data, and even flow/congestion control. Figure 2 lists data transfer parameters that are essential to applications consuming data.

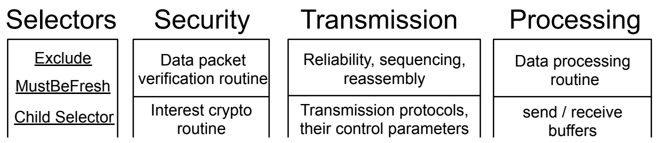

**Figure 2: Consumer-specific data transfer parameters**

We would like to introduce a generic operation (1), which binds a name prefix to consumer-specific data transfer parameters and operation (2), which starts the actual data transfer activity. In other words, operation (1) creates an association between a name prefix and transfer parameters that influence the actual data transfer.

$$associate(name\,prefix,\ transfer\,parameters) \quad (1)$$

$$consume(name\,prefix) \quad (2)$$

NDN producers care about packaging content in the Data packets (segmentation), securing Data packets, controlling the caching rate, and processing Interest packets (demultiplexing). Figure 3 lists data transfer parameters that are essential to applications producing data.

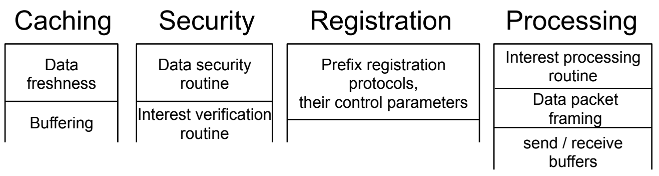

**Figure 3: Producer-specific data transfer parameters**

The same generic operation (1) can be used to bind a name prefix to producer-specific data transfer parameters. Operation (3) is similar to the *write()* operation in Socket API in a way that it converts a buffer with actual content to the Data packets of appropriate size — performs segmentation.

$$produce(name\,prefix,\ payload) \quad (3)$$

## 3.3 Design paradigm

Any network communication protocol consists of two parts: *transfer control* and *data manipulation* [5]. 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. These functions 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 TCP/IP protocol stack, *transfer control* and *data manipulation* functions are often executed independently in separate layers. A vivid example of the traditional layered design is the HTTP-SSL-TCP-IP stack. HTTP messages are passed to the session layer, where they are encrypted and placed in the transport layer, where a 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 design and implementation of subsystems whose scope is restricted to a small subset of the suite's 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 [5].

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 Application Data Units (ADUs), it is 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.

To summarize, a generic API for integrated processing will hide the complexity of communication protocols, while providing a way to customize data manipulation actions in critical areas such as security, event monitoring and error handling. Generic API provides a uniform design "language" that can be used by programmers to conveniently work with NDN network, and convey the meaning of NDN application in a clear and compact way.

## 4. API

In this section we give technical definitions of the concepts of *consumer context* and *producer context*, and provide necessary API primitives to work with them.

## 4.1 Consumer context

A process that wants to fetch Data packets performs three basic steps: creates a consumer context with desired parameters, starts data transfer using the consumer context, and adjusts data transfer parameters if necessary (Table 1).

### *4.1.1 consumer()*

*consumer()* creates a context that controls how Interests are expressed and how returning Data packets are processed. A consumer context is initialized with the following parameters: a) name prefix, b) desired reliability of packet delivery, c) indication of a possible multi-segment data fetching.

Consumer context contains many other parameters. Some parameters are already included in the context with the default values; other parameters can be freely added at any time. In both cases, to add or modify context parameters,

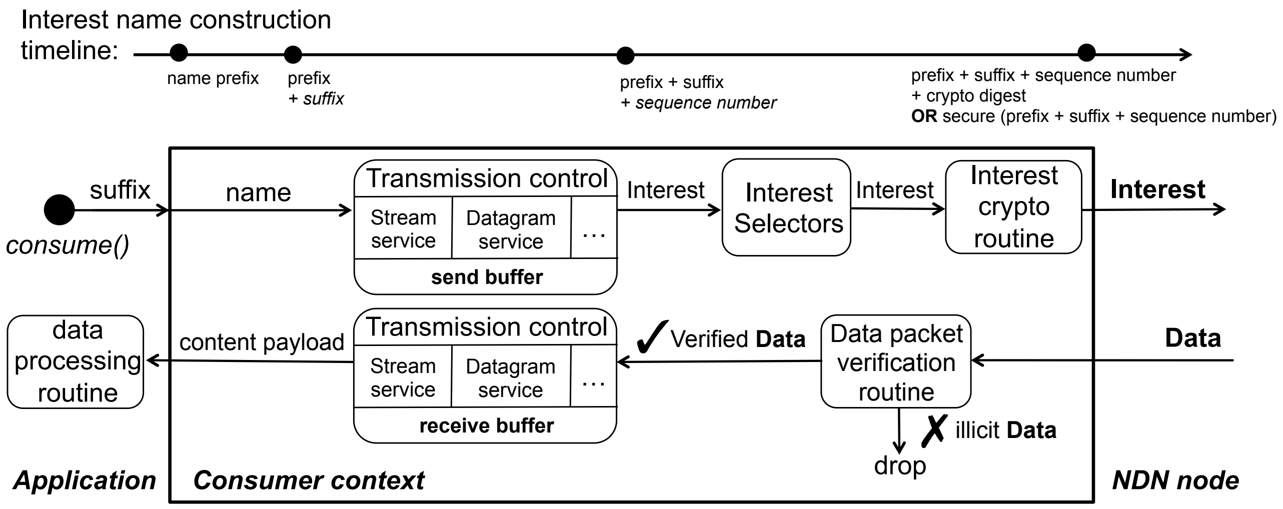

**Figure 4: Integrated processing of Interest and Data packets in the consumer context.**

| | |
|---|---|
| Initialization | **consumer** (name prefix, type, sequencing) ➔ handle |
| Primitives | **consume** (handle, name suffix)<br>**stop** (handle)<br>**close** (handle)<br>**setcontextopt** (handle, option name, value)<br>**getcontextopt** (handle, option name) |

**Table 1: API primitives for consuming data: consumer() creates a context, consume() starts data transfer, stop() terminates data transfer, close() destroys context.**

an application designer uses the *setcontextopt()* primitive.

**name prefix** — prefix of a meaningful application-specific name that is expected to bring Data packet(s) back. Name prefix can contain any components except the sequence number component as it is appended automatically by the API.

**type** — specifies type of protocol machinery to be used for content retrieval. If *UNRELIABLE* option is used, consumer context will not attempt to recover potentially lost Data segments with retransmission of Interest packets. With *RELIABLE* option, consumer context will perform Interest retransmission until the exhaustion of allowed number of retransmissions for any individual Interest packet.

**sequencing** — specifies whether consumer context must append a segment number to the name of each outgoing Interest packet, and keep incrementing it in Interests fetching next Data segments. Default values include: a) *DATAGRAM*, b) *SEQUENCE*. If *SEQUENCE* is selected, consumer context will keep expressing new Interest packets with incremented sequence numbers until the last segment of data is received. How and in what order these Interests are expressed is controlled by the transmission control protocols (i.e. Interest pipelining) which are built in the consumer context (Figure 4). To change what protocol is used or to modify its controlling parameters, application designer executes *setcontextopt()* primitive.

### *4.1.2 consume()*

*consume()* starts data transfer for a specified name with the behavior defined by the consumer context. For example, if the context was created with the *datagram* option, then calling *consume()* will result in expressing exactly one Interest packet retransmitted the number of times specified in the *reliability* option. If the *SEQUENCE* option was used, then calling *consume()* will result in consumer context expressing automatically sequenced Interest packets until the last Data packet is received. In such case, Data packets are ordered and reassembled automatically in the receive buffer, unless the *Raw* mode is activated.

*consume()* accepts the following parameters.

**handle** — unique identifier of the consumer context.

**name suffix** — additional name components, which are appended to the name prefix specified in the consumer context at initialization stage. The name suffix must not contain sequence number, because it is appended later by the transmission protocol.

Name suffixes can be used to fetch different information objects using the same transmission parameters specified in the consumer context. Another motivation for logical separation of name prefix and suffix is the necessity to give application designers the tool to perform on-the-fly changes in the data transfer process without the need to recreate a whole new consumer context for that purpose. For example, when the desired Data packets cannot be fetched and Interest packets time out, an application designer can decide to restart data fetching for the information object using another version number, or timestamp, or simply to fetch different information object with a different name suffix.

### *4.1.3 setcontextopt()*

*setcontextopt()* primitive is used to assign or modify parameters of the consumer context:

- Interest selectors
- Transmission control protocol: flow & congestion con-

trol (i.e. Interest pipelining).

- Transmission control parameters: rate limit, retransmissions, etc.
- Size of the receive buffer holding the payload of a single Data packet (*datagram* context) or multiple reassembled Data packets (*SEQUENCE* context)
- Raw mode — deactivates packet ordering and reassembly, and exposes packets in a wire format.
- Callback function that modifies the name of Interest packet in order to obfuscate some of its name components or add cryptographic digest. Digest, for instance, can be used by the producer to authenticate the consumer.
- Callback function that performs verification of incoming Data packets (Figure 4). It accepts a single Data packet and returns *True* if verification succeeds and *False* if it fails. If verification fails, the Data packet is dropped. If verification succeeds, the payload of the Data packet is placed in the receive buffer of the consumer context and will be available for reading.
- Callback function that processes the payload contained in Data packets. The routine accesses the receive buffer of the consumer context, which contains the payload of a single (*datagram* context) or multi-segment Data packets that were reassembled in a *SEQUENCE* context. Application designers can modify the length of the receive buffer using a *setcontextopt()* primitive.
- Callback functions that monitor events such as Interest timeouts, Interest retransmissions, Data packet arrival, Data verification successes/failures, etc. These instrumentation functions cannot modify packets and instead serve the purpose of gathering feedback on the data fetching process that can be used by the application designer to adjust data transfer parameters (i.e. selectors, rate limit, etc.), or restart data transfer with a different name suffix (i.e. version, timestamp, etc.).

*setcontextopt()* accepts the following parameters:
**handle** — unique identifier of the consumer context.
**option name** — name of the parameter to be modified.
**value** — new parameter value, which depending on the type of parameter, can be an integer, enumerator value, string, or a function pointer.

#### 4.1.4 *stop()*

*stop()* terminates data transfer in the consumer context. Context stops expressing any new Interests and processing incoming Data packets. Buffers are freed. The consumer context is not destroyed and application designer can reuse it by calling the *consume()* primitive.

#### 4.1.5 *close()*

*close()* primitive destroys consumer context. All callback functions are disconnected.

| | |
|---|---|
| Initialization | **producer** (name prefix) ➔ handle |
| Primitives | **produce** (handle, name suffix, payload)<br>**setup** (handle)<br>**close** (handle)<br>**setcontextopt** (handle, option name, value)<br>**getcontextopt** (handle, option name) |

**Table 2: API primitives for producing data: producer() creates a context, setup() prepares context for demultiplexing Interest packets, produce() outputs Data packets.**

## 4.2 Producer context

A process that wants to provide its data for transfer performs four basic steps: creates a producer context with desired parameters, setups Interest demultiplexing, creates content payload and passes it to the context, and destroys the context when no longer needed (Table 2).

#### 4.2.1 *producer()*

*producer()* creates a context that controls how data is produced and secured, and how Interests are demultiplexed. Producer context is initialized with a single parameter — application specific name prefix of the data. Internally, producer context contains many other parameters. Some parameters are already included in the context with the default values, other parameters can be freely added at any time. In both cases, to add or modify context parameters, application designer uses the *setcontextopt()* primitive.

**name prefix** — namespace where the data will be produced. This namespace (prefix) is used by the localhost NDN node to demultiplex incoming Interests and pass them in the producer context.

#### 4.2.2 *setup()*

*setup()* primitive activates producer context for receiving Interest packets and producing Data packets. Activation includes multiple steps. The obligatory step is FIB entry creation in the localhost NDN node in order to successfully demultiplex Interest packets arriving on the host. An additional step is needed if process-producer wants to be reachable from the global Internet. In order to achieve it, two requirements must be met: a) an adjacent NDN node(s) must forward Interest packets towards the host where producer application is running, b) the names of produced Data packets must begin with routable prefix. These needs are satisfied by the prefix registration and prefix discovery protocols [6]. The application designer executes *setcontextopt()* primitive to select prefix registration protocol along with its parameters (i.e. ACL credentials). Third step initializes send/receive buffers and connects Data packet security callback, Interest verification callback, Interest processing callback and other callback functions (Figure 5).

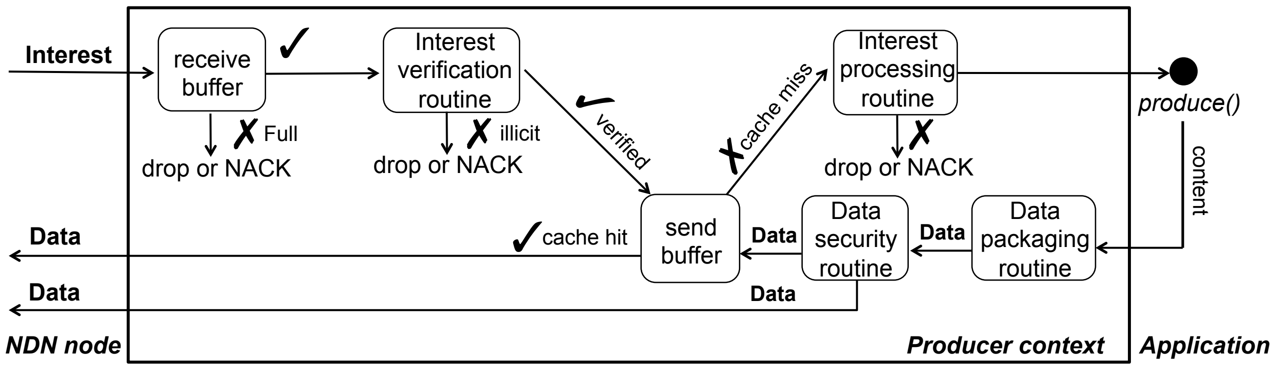

**Figure 5: Integrated processing of Interest and Data packets in the producer context.**

### *4.2.3 setcontextopt()*

*setcontextopt()* primitive is used to assign or modify parameters of the producer context:

- Packet framing — Data packet size, signature type
- Size of the receive buffer holding the demultiplexed Interest packets waiting for processing. When this buffer is full, program has enough work tasks in its queue and other demultiplexed Interest packets might be dropped (Figure 5)
- Size of the send buffer holding Data packets recently produced within this producer context. Send buffer is a simple cache with FIFO replacement policy, which resides in the producer context. Send buffer softens the difference between data production and fetching rates. This is an important optimization for handling Interest retransmission without re-producing and re-signing the same content again.
- Callback function that runs application specific verification of Interest packets after they are demultiplexed by the localhost NDN node and placed in the receive buffer, but before they reach the send buffer or the Interest processing routine. Interest verification is useful when it is necessary to authenticate Interest sender or in other ways to authorize data production. Whenever verification fails, Interest is dropped and not processed any further (Figure 5)
- Callback functions that monitor events such as Interest drops, cache (send buffer) hits/misses, Interest verification failures, etc.
- Data freshness period, which influences the rate of data caching by the network NDN nodes
- Prefix registration protocol used to setup proper forwarding of the Interests from the adjacent NDN node towards the host running the producer application.
- Prefix registration protocol properties and parameters (i.e. user credentials).

*setcontextopt()* accepts the following parameters:

**handle** — unique identifier of the producer context.

**option name** — name of the parameter to be modified.

**value** — new parameter value, which depending on the type of parameter, can be an integer, enumerator value, string or a function pointer.

### *4.2.4 produce()*

*produce()* primitive is used to pass content payload in the producer context in order to convert it in a single or multiple NDN Data packets. If application designer has specified a security function callback for the producer context, NDN Data packet(s) are signed and encrypted by this function, then placed in the send buffer of the producer context and immediately passed to the localhost NDN node. *produce()* primitive is usually called from the Interest processing routine, but can also be executed from any other location in the program.

*produce()* accepts the following parameters:

**handle** — unique identifier of the producer context.

**name suffix** — additional name components, which are appended on the per-information-object basis, such as version number, timestamp, etc.

**payload** — input buffer containing content payload. If provided buffer does not fit in a single Data packet of specified size, producer context performs segmentation of the content payload into multiple sequentially named Data packets and places all of them in the send buffer. The send buffer handles the situation when *produce()* makes more segments than currently requested by the consumer. If the next sequential Interest packets are still in transit, they are served from the send buffer, instead of being re-produced and re-signed again. Send buffer is a very important producer-side optimization for the multi-segment and reliable data fetching. Data packets are replaced in the send buffer according to FIFO policy. If the size of the supplied content payload buffer exceeds the size of the send buffer, *produce()* returns an error.

Normally, the input content payload buffer contains unsecured content, because the user-specified security function callback will secure the complete Data packet(s) to give Data packet standard content-based security properties (Figure 5).

### *4.2.5 close()*

*close()* primitive destroys producer context. Buffers are cleaned and function callbacks are disconnected. FIB entries in a localhost NDN node are removed. Prefix registration protocol performs cleanup when it is necessary.

## 5. USING NDN API CONTEXTS

In this section we describe how existing NDN applications can be implemented using Consumer-Producer API.

### 5.1 File synchronization

We start our explanation of the API with one of the simple NDN applications that requires a reliable stream service, but does not have an elaborate security model. NDN FileSync is a distributed peer-to-peer application that implements file synchronization in a shared directory [7].

Sample data packet name: */broadcast/apps/filesync/class217 /Reports/Report.pdf/<timestamp>*. Application's Interest packets contain a name of the file, which needs to be downloaded from any other peer. When this Interest is received, application parses its name in order to locate the file on the disk, and then packages the file in Data packets.

**Pseudocode 1** Sharing a file

```
 1: h ← producer("/broadcast/apps/filesync")
 2: setcontextopt(h, packet_size, 16KB)
 3: setcontextopt(h, interest_callback, ProcessInterest)
 4: setup(h)
 5: function ProcessInterest(Interest i)
 6:     Name suffix ← read i.name to understand what file is needed
 7:     content ← read file from disk
 8:     Name suffix ← append current time stamp
 9:     produce(h, Name suffix, content)
10: end function
```

**Pseudocode 2** Downloading a file

```
1: h ← consumer("/broadcast/apps/filesync", RELIABLE, SEQUENCE)
2: setcontextopt(h, receive_buffer_size, 20MB)
3: setcontextopt(h, content_callback, ProcessContent)
4: consume(h, "/class217/Reports/Report.pdf")
5: function ProcessContent(byte[] content)
6:     file ← read content
7:     Save file on disk
8: end function
```

### 5.2 Live video streaming

The second example application that we would like to discuss is the live video streaming application that requires an unreliable stream service. NDNVideo also has an interesting data production technique; instead of processing incoming Interest packets, it places Data packets, containing recently captured video frames in the user-space buffer or a localhost Content Store, and lets incoming Interest fetch the Data from these buffers [8].

Sample data packet name: */edu/ucla/stream/<timestamp> /video0/h264-1024k/segments/00%*. Components */edu/ucla* are the routable name, followed by the application-specific name components representing video frames in a specific video encoding format.

**Pseudocode 3** Producing video

```
1: h ← producer("/edu/ucla/stream")
2: setcontextopt(h, packet_size, 8KB)
3: setcontextopt(h, send_buffer_size, 100MB)
4: setup(h)
5: while True do
6:     Name suffix ← name and quality of the video
7:     content ← encode captured video
8:     produce(h, Name suffix, content)
9: end while
```

**Pseudocode 4** Consuming video

```
1: h ← consumer("/edu/ucla/stream", UNRELIABLE, SEQUENCE)
2: setcontextopt(h, receive_buffer_size, 1MB)
3: setcontextopt(h, send_rate, 50ms)
4: setcontextopt(h, content_callback, ProcessContent)
5: consume(h, "%FD%04%F65ub/video0/h264-1024k")
6: function ProcessContent(byte[] content)
7:     video ← decode content
8:     Display video
9: end function
```

### 5.3 Building Automation System

Lighting control system is an example component of a Building Automation Systems (BAS). The basic idea of the application is that Interest packets are issued by the controller and represent some action of the user: turning on/off the light, changing the color of the light, etc. Lightning panel receives an Interest and performs a requested activity. Because the user must be sure that requested action was performed successfully or failed, the application requires a reliable datagram service. Lighting control over NDN secures BAS with identifying entities on the network via an asymmetric key pair and authenticating Interest packets for control, using either RSA signatures or HMACs [9].

An example of data packet name: */ndn/ucla/boelter/3551 /lights/fixture/41/rgb-8bit-hex/<color>/<state>/<signature>*. Where <color> carries the name of the color of the light to avoid bidirectional Interest-Data exchange, and <signature> carries signature bits to authenticate request.

**Pseudocode 5** NDN gateway for lighting panel

```
1: h ← producer("/ndn/ucla/boelter/3551/lights/fixture/41/rgb-8bit-hex")
2: setcontextopt(h, verification_callback, VerifyInterest)
3: setcontextopt(h, interest_callback, ProcessInterest)
4: setcontextopt(h, security_callback, SecureData)
5: setup(h)
```

**Pseudocode 6** NDN gateway for lighting panel (continued)

```
 6: function VERIFYINTEREST(Interest i)
 7:     $Signature$ ← read signature bits from i.name
 8:     if $Signature$ is valid then
 9:         return True
10:     else
11:         $Name\ suffix$ ← read <color>, <state>, <signature> from i.name
12:         produce(h, $Name\ postfix$, VerificationNACK)
13:         return False
14:     end if
15: end function
16: function PROCESSINTEREST(Interest i)
17:     $Color\ name$ ← read the color from i.name
18:     Set the color of the light to $Color\ name$
19:     $content$ ← status of the operation
20:     produce($h$, $Color\ name$, $content$)
21: end function
22: function SECUREDATA(Data d)
23:     $Encrypted\ data$ ← encrypt(d)
24:     $Secured\ data$ ← sign($Encrypted\ data$)
25:     return $Secured\ data$
26: end function
```

**Pseudocode 7** Light controller

```
 1: $h$ ← consumer("/ndn/ucla/boelter/3551/lights", RELIABLE, DATAGRAM)
 2: setcontextopt($h$, crypto_callback, SignInterest)
 3: setcontextopt($h$, verification_callback, VerifyData)
 4: setcontextopt($h$, content_callback, ProcessContent)
 5: consume($h$, "/fixture/41/rgb-8bit-hex/FAF87F")
 6: function SIGNINTEREST(Interest i)
 7:     $Signed\ Interest$ ← sign(i)
 8:     return $Signed\ Interest$
 9: end function
10: function VERIFYDATA(Data d)
11:     if verify(d.signature, d.keyLocator) is True then
12:         return True
13:     else
14:         return False
15:     end if
16: end function
17: function PROCESSCONTENT(byte[] content)
18:     $Status$ ← read content
19:     Display $Status$ to user
20: end function
```

## 6. CONCLUSION

NDN fetches the data exclusively by its name, which means that during the data fetching process, various data transfer parameters are the properties of the namespace — unlike the IP network, where data transfer parameters are the properties of the communication channel between IP endpoints.

In this paper, we introduced Consumer-Producer integrated processing API, which represents a generic application programming interface to NDN communication protocols and architectural modules. The API with integrated processing of Interest and Data packets provides the following functionality to application designers:

1. Retrieval of single-segment and multi-segment content.
2. Reliable transmission services.
3. Reassembly and packet ordering services.
4. Packaging of the content in Data packets of constant size and buffering recently produced Data segments in the in-application cache.
5. Plugging-in user defined content-based security actions such as Interest security routine, Data security routine, Interest and Data verification routines.
6. Monitoring events related to Interest and Data processing in order to provide detailed feedback to the application.

This paper introduces two new programming abstractions: consumer context and producer context. Contexts keep all necessary state of ongoing data transmission happening under a specific name prefix. Both producer and consumer contexts are designed to be used with multiple name suffixes, but as it was mentioned earlier, different name suffixes share the common transmission characteristics associated with the prefix.

Both contexts can include additional parameters that were not described in this paper, but will be necessary for an additional functionality and future communication protocols. One of the possible extensions is automatic data versioning. Because every segment of data is immutable in NDN, most applications have to deal with multi-versioned content. Context abstraction can be used to automate name construction and manipulation related to the production and consumption of multi-versioned data.

## 7. ACKNOWLEDGEMENTS

We are thankful to Jeff Burke, Mark Stapp and David Oran for providing criticism of earlier drafts of this paper. The ideas developed in this paper are motivated by the experience in developing NDN applications by the members of Internet Research Lab (IRL), Center for Research in Engineering, Media and Performance (REMAP), and students of UCLA Computer Science department. This work is a contribution to NSF project CNS-1040868 FIA: Collaborative Research: Named Data Networking (NDN).

## 8. REFERENCES

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

[2] 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.

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

[4] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, “RFC 3550 - RTP: A Transport Protocol for Real-Time Applications,” 2003.

[5] 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.

[6] [Online]. Available: https://www.ccnx.org/releases/latest/doc/technical/Registration.html

[7] J. Lindblom, M. Huang, J. Burke, L. Zhang, “FileSync/NDN: Peer-to-Peer File Sync over Named Data Networking,” UCLA, Tech. Rep., 2013.

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

[9] J. Burke, A. Horn, A. Marianantoni, “Authenticated Lighting Control Using Named Data Networking,” UCLA, Tech. Rep., 2012.