# NDN-RTC: Real-time videoconferencing over Named Data Networking

Peter Gusev
UCLA REMAP
peter@remap.ucla.edu

Jeff Burke
UCLA REMAP
jburke@remap.ucla.edu

## ABSTRACT

NDN-RTC is a real-time videoconferencing library that employs Named Data Networking (NDN), a proposed Future Internet Architecture. It was designed to provide an end-user experience similar to Skype or Google Hangouts, while taking advantage of the NDN architecture's name-based forwarding, data signatures, caching, and request aggregation. It demonstrates low-latency HD video communication over NDN, without direct producer-consumer coordination, which enables scaling to many consumers through the capacity of the network rather than the capacity of the producer. Internally, it employs widely used open source components, including the WebRTC library, VP9 codec, and OpenFEC for forward error correction. This paper presents the design, implementation in C++, and testing of NDN-RTC on the NDN testbed using a demonstration GUI conferencing application, NdnCon.

## 1. INTRODUCTION

Named Data Networking (NDN) is a proposed future Internet Architecture that shifts from the current host-centered paradigm of the IP Internet to data-centered communication. In NDN, every chunk of data has a hierarchical name, which can include human-readable components, and a cryptographic signature binding name, data, and the key of the publisher. Consumers of data issue "Interests" for these Data packets by name. Named, signed packets matching the Interest can be returned by any node on the network, including routers. NDN's intrinsic caching can be leveraged by content distribution applications and significantly help to reduce the load on data publishers in multi-consumer scenarios [6]. Additionally, duplicate Interests for the same content can be aggregated in routers, further reducing the load on those publishers and the network. A full discussion of NDN is outside the scope of this paper; see the publications on the project website[1], including [15, 16, 7].

[1]http://named-data.net



Efficient content distribution has long been a driver application for NDN research and the broader field of Information Centric Networking (ICN) as well. We briefly discuss prior work in this area, including our own, in Section 3. However, *low-latency* application like "real-time conferencing" presents some particular design and implementation challenges that have not been widely explored in publicly available prototypes or the NDN and ICN literature. For example, obtaining the "latest data" from a network with pervasive caching, without relying on direct consumer-producer communication (which impacts scaling potential), while trying to keep application-level latency low, is challenging.

The NDN-RTC library was created to explore this arena experimentally. It was designed, implemented, and evaluated to explore NDN's potential for scalable low-latency audio/video conferencing and "real-time" traffic more generally. This paper presents the current design and initial evaluation. NDN-RTC provides basic functionality for publishing audio/video streams, and for fetching these streams with low latency. This can be leveraged by desktop or web applications, such as the NdnCon sample application, for establishing multi-party conferences. The NDN network's caching and interest aggregation are leveraged without architectural modification, with the NDN-RTC library ensuring low-latency communication.

We are working towards the goal of using NDN-RTC in NDN project videoconferences and meetings. To be practically useful in project communications, we also needed reasonable CPU and bandwidth efficiency, echo cancellation, and modern video coding: as a result, NDN-RTC is a C++ library which is built on top of the widely used WebRTC library, incorporating its existing audio pipeline (including echo cancellation) and its video codec (VP9).

The remainder of this paper is organized as follows: Section 2 lists main project goals. Section 3 covers background and prior work. Section 4 describes architecture of the library, designed namespace, data structures and algorithms used. Section 5 discusses implementation details. Section 6 evaluates main outcomes. Finally, Section 7 provides a conclusion and explains future work.

## 2. DESIGN OBJECTIVES

The NDN project team uses application-driven research to explore NDN's affordances for modern applications and to refine the architecture itself. Though it is based on what we learned from the NDNVideo project [6], NDN-RTC is a clean slate design with new goals. The initial objectives of the NDN-RTC project are to explore *low-latency* audio/video

communication over NDN, and to provide a working multi-party conferencing application that can be used by NDN project team members across existing NDN testbeds. At the same time, we attempt to preserve network-supported scalability by avoiding direct consumer-producer communication (e.g., Interests that require new Data to be generated by the producer for each request).

Over IP, low-latency audio/video conferencing applications typically establish direct peer-to-peer communication channels, for the best user experience. However, they face implementation challenges and inefficiencies related to the connection-based approach currently used in most IP conferencing solutions. Existing solutions scale poorly to high numbers of producers and consumers without dedicated aggregation units, for example.

Our high-level motivation for NDN-RTC is to see if "real-time" content publishing can be achieved like most other NDN data dissemination applications: The publisher 1) acquires and transforms data, 2) names, packetizes, and signs it, 3) passes it to an internal or external component that responds to Interests received from the "black box" of the NDN network that are matched against available named, signed, data chunks. The consumer issues Interests using appropriate names and selectors to the "black box", at the rate necessary to achieve its objectives–as informed by performance of the network in response to its requests–and reassembles and renders them. Once the namespace is defined, the publishing problem, so far, seems straightforward. Complexity is at the consumer, which must determine what names to issue at what rate, to get the best quality of experience for the application. In the real-time conferencing arena, this means low latency access to the freshest data that the "black box" of the NDN network can deliver. Bidirectional communication is achieved by having each node participate as a publisher and consumer; conference setup and multi-party chat could be handled by applying techniques such as those developed in ChronoChat [13]; and multiple bitrates for consumers to use in adaptation are handled by simply publishing in multiple namespaces corresponding to multiple bitrates.[2]

Based on this high-level motivation, we developed several specific design goals:

- **Low-latency audio/video communication.** Library should be capable of maintaining low-latency (150-300ms) communication for audio and video, similar to driver applications such as Skype, WebEX and Google Hangouts.
- **Multi-party conferencing.** Publishing and fetching several media streams simultaneously should not require significant computational resources from the user and should maintain the same latency as in one-to-one conferencing.
- **Passive consumer & cacheability.** There should be no explicit negotiation or any coordination between active conference members as this may limit scalability and flexibility of use. Data should be cacheable for multiple consumers capable of decrypting it.
- **Data verification.** Library should provide content verification using existing NDN signature capabilities.
- **Encryption-based access control.** (Not implemented in this version.)

[2]Adaptive solutions based on scalable coding present some additional challenges that are left as future work.

# 3. BACKGROUND AND PRIOR WORK

To the authors' knowledge, NDN-RTC is one of the few applications which meets real-time requirements and has been tested over existing NDN testbed.

A non-real time video-streaming software solution, NDNVideo, was successfully tested and deployed over NDN, and proved its high scalability [6]. The project focused on developing random-access and live video for location-based and mixed reality applications. Audio and video content sliced into data chunks was published into a CCN repository for online and offline access. Data chunks are named sequentually according to NDN naming conventions [14]. In addition, application namespace provides time-based index which maps individual data chunks to the media stream timeline and allows consumer to seek easily inside the stream. Even though the project worked well for live and stored audio/video streaming, it didn't meet requirements for low-latency communication and couldn't be used as a conferencing solution.

Another recent attempt for audio/video streaming was made in [12]. The technical report describes implementation details of two applications, NDNlive and NDNtube, which are designed to work with the latest NFD (NDN Forwarder Daemon) and use Repo-ng [1] for offline media storage. These applications rely heavily on Consumer/Producer API [10] which, unlike NDNVideo, operate on ADU (Application Data Units) level - individual frames from audio and video streams are segmented, named and published over NDN for remote access. These ADUs are then retrieved by consumers at constant rate by executing *consume()* call from Consumer/Producer API.

Other streaming-related works like [9] and [11] explored the advantages of using ICN networks for MPEG Dynamic Adaptive Streaming over HTTP. However not related to low-latency streaming, the idea is to leverage network's caching ability for serving chunks of video files over to multiple consumers.

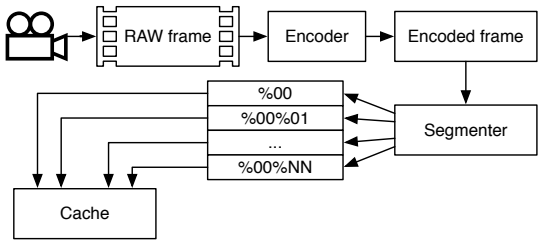
An audio conferencing application was developed in [17]. It leveraged use of Mumble VoIP software and used NDN as a transport. However, echo cancellation was not implemented in this tool which made it quite hard to use in real life. Initial effort for conference and user discovery was made in this work as well, suggesting that building on an existing, resilient platform is the best way to generate a usable application. Therefore, NDN-RTC was chosen to be built on top of WebRTC library, in order to utilize its' audio-processing capabilities and video codecs, and potentially give an opportunity for easier integration with supported web-browsers.

# 4. APPLICATION ARCHITECTURE

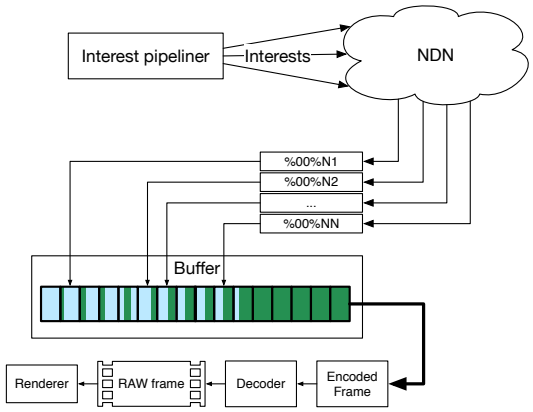
There are two main roles defined in NDN-RTC: producer and consumer. With NDN, the paradigm of real-time communication shifts from push-based (when the producer writes data to the socket, and the consumer reads it as fast as possible) to pull-based (the producer publishes data on the network at own pace, while the consumer has to request data needed and manage incoming data segments).

## 4.1 Producer

The producer's main tasks are to acquire video and audio data from media inputs, encode them, pack them into network packets, and store them in the cache for incoming Interests. In this way, complexity shifts to the consumer,




(a) Producer




(b) Consumer

**Figure 1: NDN-RTC producer and consumer operation.**

and scaling is supported by the network.

In the case of video streaming, the producer uses video encoding in order to reduce size of the frames. There are two types of encoded frames: *Key* and *Delta*. Key frames contain most of the video information, and do not depend on any previous frames to be decoded. Whereas Delta frames are dependent on the previous frames (received after the last Key frame), and cannot be decoded without significant visual artifacts if any of the Key frames are missing.

Encoded frames vary in size, but average bitrate stays the same. For example, the average sizes of frames for 1000 kbps stream using VP8/VP9: Key frames are ≈ 30KB, and delta frames are ≈ 3-7KB. Therefore, depending on the underlying Internet Protocol used (IP in the existing NDN testbed), the producer may need to segment encoded frames into smaller chunks and provide clear naming conventions for the consumer to fetch them.

## 4.2 Namespace

As there is no direct consumer-producer communication, it is the producer's job to provide as much supportive information as possible, so that the consumer is able to use this information. These kinds should be reflected in the namespace: media data (segmented video frames and bundled audio samples), error correction data, and metadata.

### 4.2.1 Media

The NDN-RTC producer uses a *media stream* concept for describing published media. Media stream represents a flow of raw media packets (video frames or audio samples) coming from an input device. Streams usually derive names from their input devices. It is quite natural for a producer to publish several media streams simultaneously, if there is more than one device from which to publish media (for instance, video from camera, audio from microphone, and another video stream for computer screenshots). As all raw data should be encoded, the next level in the name hierarchy represents different encoder instances called *media threads*. Thus, media threads allow the producer to provide the same media stream in several copies (for instance in low, medium and high quality, so that the consumer can choose media thread more suitable for current network conditions).

NDN-RTC does not use any specific media container format for delivering media to consumers. Instead, encoded media packets are segmented if needed and published under distinctive hierarchical names. Video frames are separated into two domains as per frame type, *delta* and *key*, and numbered sequentially and independently. Sequence numbers for both delta and key frames start from 0. Next level specializes data type, either media data or parity. Parity data, if the producer opts to publish it, can be used by a consumer to recover frames that miss one or more segments. The topmost level of the namespace defines individual data segments. These segments are also numbered sequentially, and segment numbers conform to NDN naming conventions [14].

In the case of audio streams, there are some differences. First, there are no key frames; therefore all audio packets are published under the *delta* namespace. Second, audio samples are significantly smaller than video frames and do not require to be segmented. In practice, it appears that multiple audio samples can be bundled into one data segment. Instead of segmenting audio packets, they are bundled together until the size of one data segment is reached, and published only after that.

Consumers need to know the producer's streams structure in order to fetch data successfully. In order to save consumers from traversing the producer's namespace, which can be time-consuming and unreliable, the producer publishes meta information about current streams under *session info* component. Thus, consumers can retrieve up-to-date information about the producer's state.

### 4.2.2 Metadata

Besides naming data objects, data names can be appended by some additional media-level meta information, which can be utlilized by consumers regardless of which frame segment was received first. Four components are added at the end of every data segment name:

*seg_name*/**num_seg/playback#/paired_seq#/num_parity**

*num_seg* - total number of segments for this frame;

*playback#* - absolute playback number for current frame; this is different from the *frame#* which is a sequential number for the frame in its domain (i.e. Key or Delta);

*paired_seq#* - sequential number of the corresponding frame from other domain (i.e., for delta frames, it is sequential number of the corresponding key frame which is required for decoding);

*num_parity* - number of parity segments for particular frame.

The pull-based nature of NDN and our experimental application's deliberate avoidance of explicit consumer-producer synchronization (allowing publisher-independent scaling) have shown the importance of providing sufficient meta information on the producer side. Such information (which could

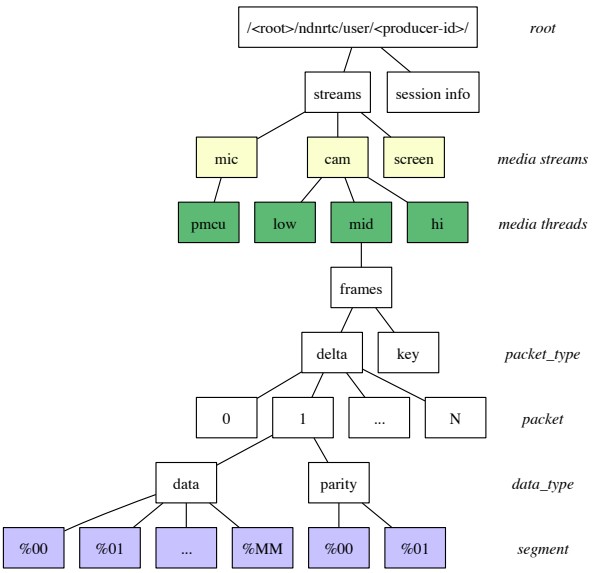
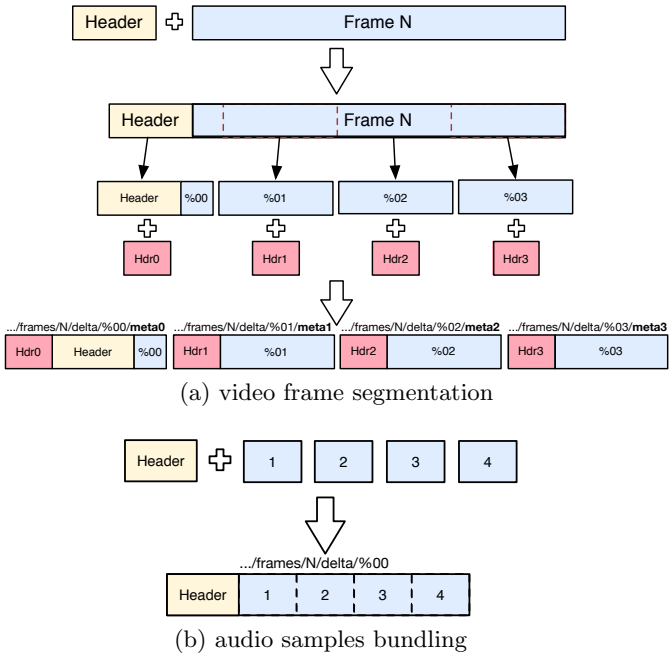



**Figure 2: NDN-RTC namespace**




(a) video frame segmentation

(b) audio samples bundling

**Figure 3: Segmentation and bundling**

include Interests' nonce values, Interests' arrival timestamps and data generation delays), if added to the returned data segment, may help consumers evaluate relevant network performance, detect congestion and assess whether incoming data is likely to be stale (delayed beyond the path delay). Furthermore, keeping historical data on the consumer side may help Interest pipelining in the future. For instance, providing the average number of segments per frame type helps consumers estimate the number of required initial Interests to fetch upcoming frames. This helps keep frame fetching cycles minimal.

## 4.3 Data objects

Producer generates signed data objects from input media streams and places them in cache instantly. Incoming Interests retrieve data from cache, if it is present, or forwarded further to the producer, if the requested data has yet to be produced. In such cases, the producer maintains a Pending Interests Table (PIT), which is checked every time a new data object is generated. If an Interest for a newly generated data object exists in the PIT, it gets answered, and PIT entry is erased.

### 4.3.1 Media stream

Packet payload consists of two types of data – media stream data and metadata. Video stream data contains actual bytes received from library's video encoder which represent encoded frame. For the audio, NDN-RTC uses RTP and RTCP packets coming from WebRTC audio processing pipeline which are then fed into similar pipeline on the consumer side for proper rendering and corrections (like echo-cancellation, gain control, etc.).

### 4.3.2 Metadata

Besides actual stream data, data objects contain some amount of meta information which is prepended during frames segmenting (see Figure 3(a)). There are two header types: *Frame header* and *Segment header*. Frame header is prepended to segment #0 of each individual video frame, and contains media-specific information (such as size of a frame), timestamp, current rate and unix timestamp, which can be used for calculating actual delay between NTP-synchronized producer and consumers (see Figure 4). Segment header is prepended to every segment of a frame. It carries network-level information which can be used by consumers to estimate current network conditions and origin of the data objects received:

*Interest arrival timestamp*: Timestamp of the interest arrival. Monitoring this value and interest expression timestamps over time may give consumers a clue about how it takes for Interests to reach the producer. This value is only valid when nonce value belongs to one of a given consumer's Interests.

*Generation delay*: Time interval in milliseconds between Interest arrival and segment publishing. A consumer can use this value in order to control the number of outstanding Interests. This value is only valid when nonce value belongs to one of the consumer's Interests.

*Interest nonce*: Nonce value of the interest which requested this particular segment. There are three meaningful cases: 1) *value belongs to the Interest issued previously* - consumer received non-cached data requested by Interest issued previously; 2) *value is non-zero, but does not belong to any of the previously issued Interests* - consumer received data requested by some other consumer; data may be cached; 3) *value is zero* - consumer received data which was cached on the producer side and never requested by anyone before.

Audio samples are not prepended by any segment header, however the whole audio bundle is prepended by the same frame header (see Figure 3(b)).

## 4.4 Consumer

The NDN-RTC architecture is receiver-driven, in which the consumer aims to achieve the following goals:

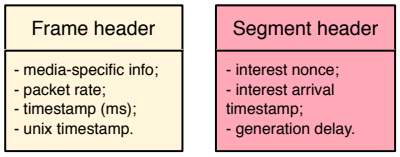



**Figure 4: Frame and segment headers**

- fetch the latest data available in the network;
- choose the most appropriate media stream bandwidth from those provided by the producer (by monitoring network conditions);
- fetch (and, if necessary reassemble) media in the correct order for playback;
- mitigate, as far as possible, the impact of network latency and packet drops on the viewer's quality of experience.

### 4.4.1 Interest pipelining and Data buffering

Media streams are packetized by the publisher, currently into segments that are less than the typical 1500 byte MTU of the current Internet. The consumer implements Interest pipelining and Data buffering (see Figure 1(b)) to fetch and reassemble video and audio data. An asynchronous Interest pipeline issues Interests for individual segments. Packet re-ordering and drops and latency fluctuations absorption are handled by the frame buffer, which re-arranges received frames in the playback order.

### 4.4.2 Frame fetching

The consumer obtains the number of segments per frame from metadata in the received segment; however, to minimize latency, it should issue a pipeline of Interests simultaneously. Therefore, at first, the consumer uses an estimate of the number of segments it must fetch for a given frame, issuing $M$ Interests (see Figure 5). If Interests arrive too early, they will be held in the producer's PIT and stay there until the frame is captured and packetized. We call the delay between interest arrival and availability of the media data the **generation delay**, $d_{gen}$. Once the encoded frame is segmented into $N$ segments and published, Interests $0-M$ are answered and the Data returns to the requestor(s). Upon receiving the first data segment, the consumer knows the exact number of segments for the current frame, and issues $N - M$ more Interests for the missing segments, if any. These segments will be satisfied by data with no generation delay, as the frame has been published already by the producer. The time interval between receiving the very first segment and when the frame is fully assembled is represented by $d_{asm}$ and called **assembly time**. Note that for frames that are smaller than the estimate, some Interests may go unanswered; this is currently a tradeoff made to try to keep latency for the frame as a whole low. These Interests have low lifetimes, of about 300ms.

Of course, additional round trips for requesting missing data segments increase overall frame assembly time and the possibility that the frame will be incomplete by the time it should be played back. This problem can be mitigated if the consumer is able to make more accurate estimates of the number of initial Interests. The latest versions of the library uses different namespaces for bitrates, and within each bit frame, for key frames and delta frames, making it straightforward for the consumer to keep interests outstanding for the next frame in each and to estimate the number of interests needed per frame, as the average number of segments varies greatly for different frame types. Additionally, it tracks the average number of segments per frame type, adapting its estimates over time.

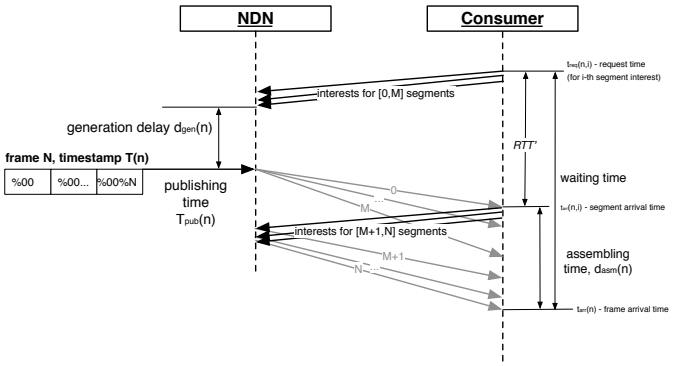



**Figure 5: Fetching frame**

The similar process used for fetching audio, though for now, audio bundles are represented by just one segment.

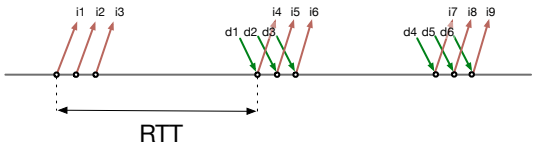
### 4.4.3 Buffering

As in sender-driven delivery, the consumer uses a "jitter buffer" in order to manage out-of-order data arrivals and variations in network delay, and as a place to assemble segments into frames. However, the role of such a "jitter buffer" has some NDN-specific aspects. In sender-based video delivery, buffer slots can be allocated only when data arrives. A pull-based paradigm requires the consumer to request data by name explicitly, however. Therefore, after expressing an Interest, the consumers "knows" that new data is coming, and a frame slot should be reserved in the buffer. Practically, this means that there will always be some number of reserved empty slots in the buffer. Thus, the NDN-RTC jitter buffer's size is expressed in terms of two values measured in milliseconds: its *playback size* is the playback duration in milliseconds of all complete ordered frames by the moment of retrieveing next frame from the buffer; its *estimated size* is *playback size* + *number of reserved slots* $\times$ $1/$*producer rate*.

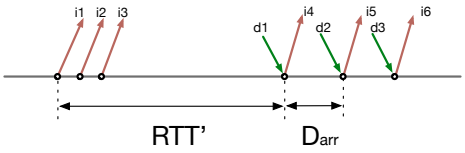
The difference between estimated buffer size and playback size corresponds to the effective RTT, which we call $RTT'$. (This cannot be smaller than the actual network RTT value.) Minimizing $RTT'$ indicates that consumer receives the most recent data for the leeast amount of outstanding Interests. Monitoring this value over times provides the consumer information on possible "sync" status with the producer. For example, the consumer may use it during the fetching process, as will be discussed in the next section.

### 4.4.4 Interest expression control

A key challenge in a consumer-driven model for videoconferencing in a caching network is to ensure the consumer gets the latest data, without resorting to direct producer-consumer communication that would limit scaling. To get fresh data, the consumer cannot rely on using such flags as *AnswerOriginKind* and *RightMostChild*. The high rate of streaming data relative to network latency means there is no guarantee that the data satisfying those flags received by a consumer will be the most recent one. Instead, it is neces-




(a) Bursty arrival of cached data, which reflects interests expression pattern and indicates that the data is not the latest.




(b) Periodic arrival of fresh data, reflects publishing pattern and sample rate.

**Figure 6: Getting the latest data: arrival patterns for the cached and most recent data**

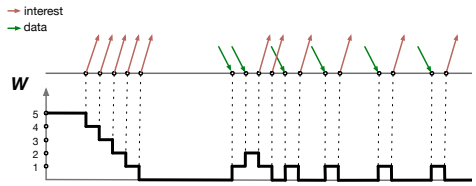
sary to use other indicators to ensure that the consumer is requesting the most up-to-date stream data possible given its network connectivity.

Our initial solution is to leverage the known segment publishing rate, which is available in stream-level metadata, and note that, under normal operation, old, cached samples are likely to be retrieved more quickly than new data. [3] We define the interarrival delay ($D_{arr}$) as the time between receipt of successive samples at a given consumer. Delays in the most recent samples follow the publishers' generation pattern, but cached data will follow the Interest expression temporal pattern. Therefore, by monitoring inter-arrival delays of consecutive media samples and comparing them to the timing of Interest expression, consumers can estimate data freshness (see Figure 6).

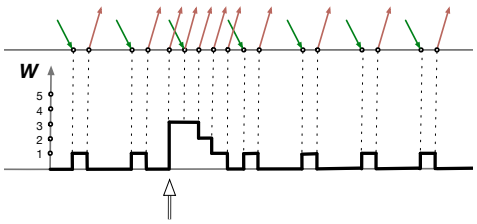
During bootstrapping, the consumer "chases" the producer and aims to exhaust network cache of historical (non-real time) segments. By increasing the number of outstanding Interests, the consumer "pulls cached data" out of the network, unless the freshest data begin to arrive. In order to control Interest expression, a concept of $W$ (roughly an "Interest window") is introduced (see Figure 7). It expresses Interests only when $W > 0$. At every moment, $W$ indicates how many outstanding Interests can be sent. Before the bootstrapping phase, the consumer initializes $W$ with a value which reflects the consumer's estimate of how many Interests are needed in order to exhaust network cache and reach the most recent data.

$W$ provides a simple mechanism to speed up or slow down Interests expression. Any increase in $W$ value makes the consumer issue more Interests (Figure 7(b)), whereas any decrease in $W$ holds the consumer back from sending any new Interests (Figure 7(c)). Larger values of $W$ make the consumer reach a synchronized state with the producer more quickly. However, a larger value means a larger number of outstanding Interests and larger $RTT'$ because of longer

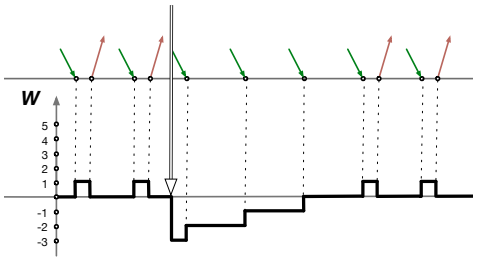
[3]If the consumer is the *only* consumer of the stream, its interests will go directly to the publisher, which also yields the correct behavior. A more complex challenge, for further study, is when segments are inconsistently cached in different ways along the path(s) that Interests take.




(a) $W$ concept




(b) interests bursting ($W + 3$)




(c) interests withholding ($W - 3$)

**Figure 7: Managing interest expression**

generation delays $d_{gen}$ for each media sample. By adjusting the value of $W$ and observing inter-arrival delays $D_{arr}$, the consumer can find minimal $RTT'$ value while still getting non-cached data, thus achieving the best synchronization state with the producer.

For more complex scenarios of video streaming, the consumer controls expression of "batches" of Interests rather than individual Interests, because video frames are composed of several segments. In this case, $W$ is adjusted on a per-frame basis, rather than per-segment. In all other respects, the same logic (as above) can be applied.

The bootstrapping phase starts with issuing an Interest with the enabled *RightMostChild* selector, in Delta namespace for audio and Key namespace for video. The reason this process differs for video streams is that the consumer is not interested in fetching Delta frames without having corresponding Key frames for decoding. Once an initial data segment of a sample with number $S_{seed}$ has been received, the consumer initializes $W$ with initial value $N$ and asks for the next sample data $S_{seed}+1$ in the appropriate namespace. Upon receiving the first segments of sample $S_{seed} + 1$, the consumer initiates the fetching process (described above) for all namespaces (Delta and Key, if available). This bootstrapping phase stops when the consumer finds the minimal value of $W$ which still allows receiving the most recent data, i.e., consumer reaches synchronized state with the producer and switches to a normal fetching phase where no adjustments for $W$ are needed.

*Application-level PIT.* In most cases, consumers aim to

express Interests for the data not yet produced, to be immediately satisfied when data is produced. The current NDN-CPP library provides a producer-side Memory Content Cache implementation into which data is published. However, this is only useful when data has been published and put in the cache before an Interest for this data has arrived. For the missing data, the Interest is forwarded to the producer application which stores it in the internal Pending Interests Table (PIT) unless requested data is ready. This functionality seems quite common for low-latency applications, and has now been incorporated into the NDN-CPP library implementation.

# 5. IMPLEMENTATION

NDN-RTC is implemented as a library written in C++, which is available at `https://github.com/remap/ndnrtc`. It provides publisher interface for publishing arbitrary number of media streams (audio or video), and consumer interface with a callback for rendering decoded video frames in a host application. OS X platform is currently supported; Linux build instructions will be added soon. The library distribution also comes with a simple console application which demonstrates the use of NDN-RTC library.

NDN-RTC exploits some functionality from several third-party libraries it is linked against with: NDN-CPP [8] is used for NDN connectivity. The WebRTC framework [3] is utilized in two ways: 1) incorporation of the existing video codec; 2) full incorporation of the existing WebRTC audio pipeline, including echo cancellation; 3) OpenFec [2] library is utilized for forwad error correction support.

Apart from the library, first desktop NDN videoconferencing application NdnCon [4] was implemented atop NDN-RTC. It provides convenient UI for publishing and fetching media streams, text chat and organizing multi-party audio/video conferences.

# 6. EVALUATION

During the course of this project, there were several architecture design iterations that introduced successive improvements in the overall quality of the algorithms. (These are described further in this section.) Each iteration tackled problems that were revealed during tests (mostly in practice rather than simulated).

## 6.1 Video streaming performance

Video streaming performance in previous versions of NDN-RTC largely suffered from video "hiccups", even when being tested on trivial one-hop topologies. The cause of this problem turned out to be an inefficient frame fetching process.

As described in previous sections, Key frames are usually much heavier than Delta frames and require more data segments for delivery. In early library versions, this differentiation was not reflected in the producer's namespace, which made consumer to pipeline equal number of initial interests ($M$ on Figure 5) for both frame types. This resulted in larger assembling times ($d_{asm}$) for the Key frames and additional round-trips of missing interests. Increased assembling time quite often caused skipping incomplete Key frames as they were not assembled by the time they should be played out. Eventually, all the consequent Delta frames were skipped as well which degraded overall video streaming experience by introducing video "hiccup" effect.

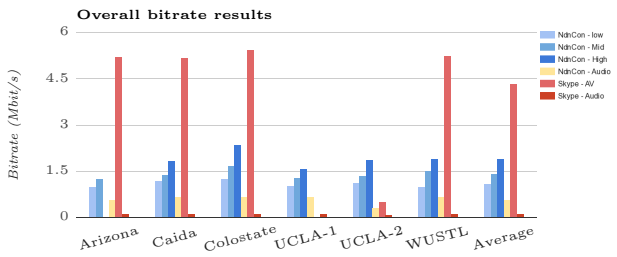



**Figure 8: 2-peer conference tests compared to Skype**

Having separate namespace for Key frames allowed consumer to maintain separate interest pipelines per frame type and collect historical data on the average number of interests required to fully retrieve one frame of each type in one round trip.

## 6.2 Quality of experience

A series of tests were taken in order assess efficiency and quality of service compared to Skype calls. Each test was comprised of six runs of 2-person 5-minute conference talks using NdnCon (GUI conferencing application build atop NDN-RTC):

- 3 runs of audio+video with low, medium and high video bandwidths settings (0.5, 0.7 and 1.5 Mbit/s accordingly);
- 1 run of audio-only conference;
- 1 run of Skype audio+video conference;
- 1 run of Skype audio-only conference.

Tests were taken across existing NDN testbed between REMAP hub (aleph.ndn.ucla.edu) and six other hubs. Tests covered both one-hop and multi-hop (with several intermediate hubs) topologies.

Figure 8 shows overall bitrate usage results. Firstly, whereas Skype has fully utilized link capacity between peers and delivered higher bitrate videos, NdnCon did not adjust to the current network conditions which makes feature of adaptive rate control highly desirable.

Actual average bitrates are turned out to be slightly higher than pre-configured video streams – 0.7, 0.9 and 1.8 Mbit/s for low, medium and high bandwiths – which can be explained by NDN packet overhead which is about 280-330 bytes large and account for ≈30% of segment size (1000 bytes). Having such large overhead makes transferring audio samples in separate segments highly inefficient. Thus, further improvent to the algorithms was to bundle consecutive audio samples unless they fill the size of a segment. With 90 Kbit/s audio, approximately 5 audio samples can be added to 1000-bytes data segment. This improvement eventually reduced audio bandwith effectively (and interests number on consumer side) and made it comparable to Skype audio bandwidths.

The overall user experience for these 2-person conferences was subjectively assessed as being higher than average - 3 points on a scale from 0 to 5 (Skype calls were taken as a 5-point user experiences).

## 6.3 Consumer-Producer synchronization

In previous library versions, the consumer "chased" the producer's time-series data by exhausting cached data and

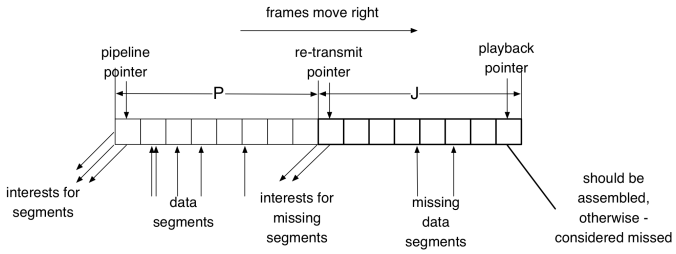
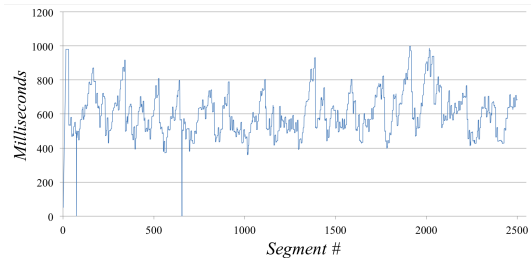
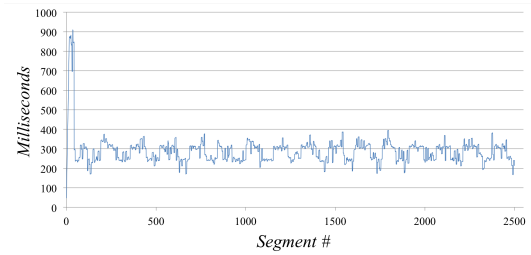



**Figure 9: Bufferization in earlier library versions**




(a) $d_{gen} \approx 600ms$ resulted in 50% retranmissions

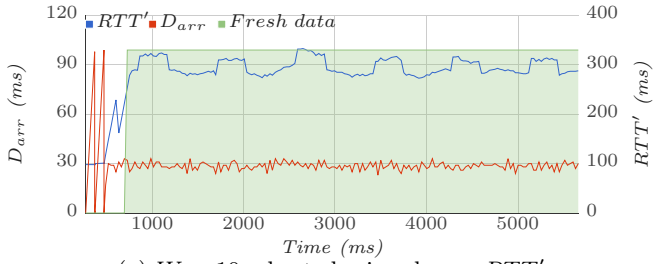



(b) $d_{gen} \approx 310ms$ - no redundant retransmissions

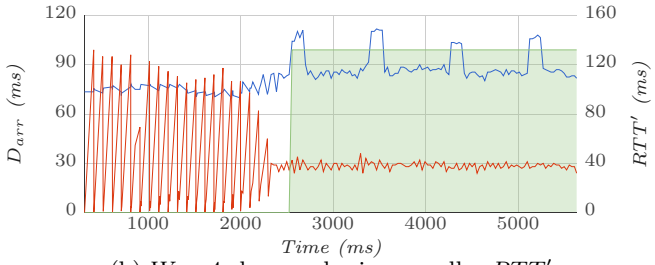
**Figure 10: Two separate runs of earlier library version on similar topology (one-hop): random results for data generation delay $d_{gen}$ due to poor consumer-producer synchornization**

issuing large number of outstanding interests. However, there were no mechanism for consumer to figure out how early those interests are issued and whether interest expression should be postponed in order to eliminate timeouts. For two similar test runs (one-hop topology) the number of timed out interests and retransmissions could vary greatly (either ≈1% or ≈50%). The latter case happened due to incorrect consumer's synchronization with the producer - interests were issued too early so that they were timed out before any data has been produced. This problem could be solved by increased interests' lifetime. However, for previous library versions, bufferization mechanism (see Figure 9) dictated interests' lifetime - in fact, in order to maintain re-transmission checkpoint, all interests entering the buffer should have lifetime equals half of the current buffer size. This approach resulted in unavoidable interests time outs in cases when consumer issued interests far too early before the actual data was produced.

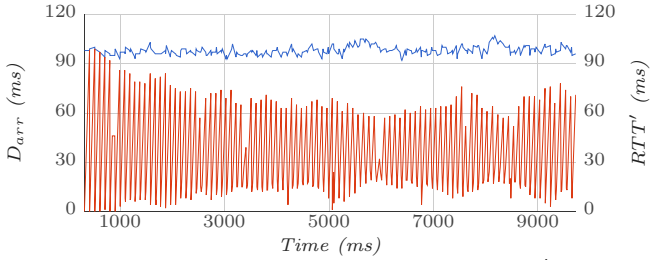
For the current version of the library, re-transmission checkpoint is placed at $RTT$ milliseconds from the end of the




(a) $W = 10$: short chasing, larger $RTT'$




(b) $W = 4$: longer chasing, smaller $RTT'$



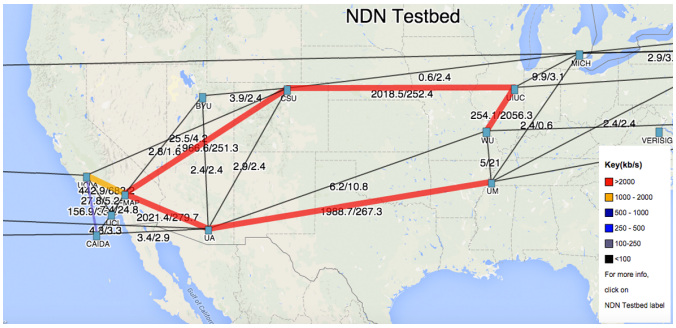

(c) $W = 3$: consumer can't exhasut cache, $RTT' = RTT$

**Figure 11: Larger $W$ decreases "chasing" phase, but increases $RTT'$ for the same network configuration ($RTT \approx 100ms$)**

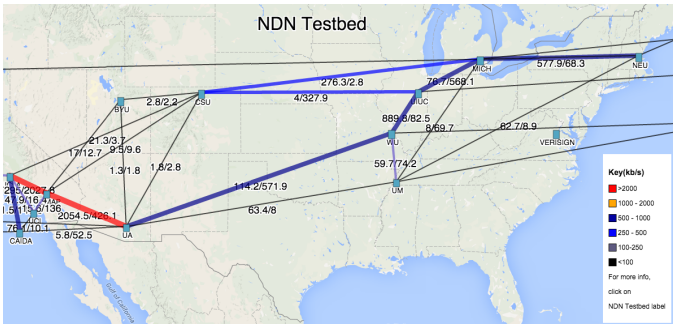
buffer ($J = RTT$ on the Figure 9), this, together with updated NFD retransmission strategy [5], alllowed larger interests lifetimes.

Moreover, the problem described above could not have happened if consumer would know that it issues interests too early. Chasing algorithm in older library versions was exhausting network cache too aggressively - interests were issued constantly unless they fill up the buffer. After that, frames were retrieved at producer's rate and more interests issued unless consumer starts to receive the most recent data. This approach lacks from any knowledge about issued interests and data generation delay and $RTT'$ are not taken into account.

With introduction of $W$ concept, consumer have full control of the interest expression. Figure 11 shows how bigger value of $W$ helps to exhaust cache faster. The number of outstanding interests is controlled by a consumer and directly influences how fast consumer can "chase" the producer. Every time a new interest is expressed, $W$ gets decremented and when new data arrives, $W$ is incremented, thus allowing consumer to issue more interests. $W$ concept allows "lazy" start for consumer - by specifying smaller $W$, consumer issues less interests. Further, consumer observes cache exhaustion by monitoring $D_{arr}$ and, if cache has not been




(a) NDN testbed utilization during biweekly NDN seminar




(b) NDN testbed utilization during 4-peer call between UCLA, REMAP, WashU and Caida hubs

**Figure 12: NDN testbed utilization during one-to-many and many-to-many scenarios**

exhausted during allocated time, consumer may increase the value of $W$ in order to express more interests. Similarly, consumer may opt to decrease $W$ in cases where original value resulted in too agressive behaviour.

## 6.4 Community use

Initial attempts of deploying NdnCon conferencing application for NDN Community were made in early 2015. NdnCon was used to stream NDN seminar over the existing NDN testbed. To do that, an audio/video bridge was set up using third-party tools like Soundflower and CamTwist Studio allowing to publish captured screen and audio feed from the seminar (taken in Cisco WebEX conferencing tool) over the NDN. Figure 12(a) shows NDN testbed utilization during 1-hour conference call. It is expected that media streams were consumed by 5 to 8 people. Overall quality was subjectively satisfying as reported by users.

Another run was made in an attempt to test multi-party conferencing ability. This test included 4 peers each publishing 3 video streams and 1 audio stream and fetching 1 video and 1 audio stream from each of the other participants. Participants were distributed accross four campuses - UCLA, REMAP, CAIDA and WashU (Figure 12(b)). These results were subjectively satisfying, however for one user (connected to WashU hub), audio cut-offs happened more often than for the other participants. The root causes of this have not been revealed yet. Many-to-many test scenarious are complex and will be taken for future work.

# 7. CONCLUSION AND FUTURE WORK

Discuss quality of experience.

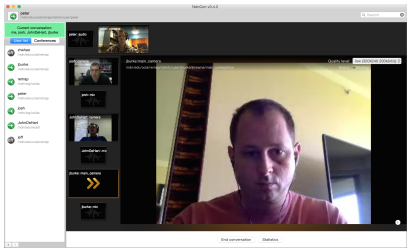


**Figure 13: NdnCon screenshot [@@REPLACE]**

Future work:

**Adaptive rate control.** In the current library design, the producer may deliberately choose to publish several copies of the same video stream with different encoding parameters, thus allowing consumer to select the most appropriate stream for current network conditions. However, the selection is made manually and depends on user's perceptual assessment of the retrieved media. Implementation of adaptive rate control would simplify this process, and allow the network to be utilized more efficiently based on current conditions.

**Scalable video coding.** An elegant way to offload the producer from publishing multiple copies of the same video stream in different bandwidths is to utilize Scalable Video Coding. By reflecting SVC layers in the namespace, consumer will have more freedom for adapting media streams to the current network. This opportunity should be explored and added in a future versions.

**Audio prioritization.** For quality of experience in typical audio/videoconferencing applications, audio should be prioritized over video. This can be done at the application level and we may provide such support in the future.

**Encryption-based access control** The current NDN-RTC design supports basic content signing and verification. However, further basic security features have yet to be implemented, e.g., media data encryption, consumer access control.

**Conference management** Work on ndncon. unknown but verified publishers trust;

# 8. ACKNOWLEDGEMENTS

This project was partially supported by the National Science Foundation (award CNS-1345318 and others) and a grant from Cisco. The authors thank Lixia Zhang, Van Jacobson, and David Oran, as well as Eiichi Muramoto, Takahiro Yoneda, and Ryota Ohnishi from Panasonic Research, for their input and feedback. John DeHart, Josh Polterock, Jeff Thompson, and others on the NDN team provided invaluable testing of NdnCon. The initial forward error correction approach in NDN-RTC was by Daisuke Ando.

# 9. REFERENCES

[1] Repo-ng. `https://github.com/named-data/repo-ng`.
[2] OpenFEC Library. `http://openfec.org`.
[3] WebRTC Project. `http://www.webrtc.org`.
[4] NdnCon GiHub Repository. `https://github.com/remap/ndncon`, September 2014.

[5] NFD version 0.2.0 release notes. http://named-data.net/doc/NFD/0.3.1/RELEASE_NOTES.html#nfd-version-0-2-0-changes-since-version-0-1-0, August 2014.

[6] J. B. Derek Kulinski. NDNVideo: random-access live and pre-recorded streaming using ndn. Technical report, UCLA, September 2012.

[7] V. Jacobson, D. Smetters, and J. Thornton. Networking named content. *... Emerging networking ...*, 2009.

[8] J. B. Jeff Thompson. NDN Common Client Libraries. *NDN, Technical Report NDN-0007*, September 2012.

[9] S. Lederer, C. Mueller, and B. Rainer. Adaptive streaming over content centric networks in mobile networks using multiple links. *âĂę (ICC)*, 2013.

[10] I. Moiseenko and L. Zhang. Consumer-producer api for named data networking. *Proceedings of the 1st international conference ...*, 2014.

[11] D. Posch, C. Kreuzberger, and B. Rainer. Client starvation: a shortcoming of client-driven adaptive streaming in named data networking. *Proceedings of the 1st âĂę*, 2014.

[12] L. Wang, I. Moiseenko, and L. Zhang. Ndnlive and ndntube: Live and prerecorded video streaming over ndn. Technical report, UCLA, 2015.

[13] Y. Yu. ChronoChat. https://github.com/named-data/ChronoChat.

[14] Y. Yu, A. Afanasyev, Z. Zhu, and L. Zhang. Ndn technical memo: Naming conventions. Technical report, UCLA, July 2014.

[15] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, K. Claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang. Named data networking. Technical report, 2014.

[16] L. Zhang, D. Estrin, J. Burke, V. Jacobson, J. Thorton, D. K. Smetters, B. Zhang, G. Tsudik, K. Claffy, D. Krioukov, D. Massey, C. Papadopoulos, T. Abdelzaher, L. Wang, P. Crowley, and E. Yeh. Named data networking tech report 001. Technical report, 2010.

[17] Z. Zhu, S. Wang, X. Yang, V. Jacobson, and L. Zhang. Act: audio conference tool over named data networking. pages 68–73, 2011.