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 uses Named Data Networking (NDN), a 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 data naming, signing, caching, and request aggregation. It demonstrates low-latency HD video communication on NDN, without direct producer-consumer coordination, enabling scaling to many consumers. Internally, it employs widely used open source components, including the WebRTC library, VP9 codec, and OpenFEC (forward error correction). This paper presents the design, implementation in C++, and testing of NDN-RTC on the NDN testbed using a graphic conferencing application.

1. INTRODUCTION

Named Data Networking (NDN) is a Future Internet Architecture that shifts from the current host-centered paradigm towards data-centered communication. [**?**] In NDN, every chunk of data has a hierarchical name (often human-readable) which is used by routers to satisfy data requests called "interests". This data is also cached by routers and used to answer similar future requests. NDN's intrinsic caching ability can be leveraged by content distribution applications and significantly help to offload data publishers in multi-peer scenarios.

Low-latency audio/video conferencing applications often require establishing direct peer-to-peer communication channels, for the best user experience. They face implementation challenges and inefficiencies related to the connection-based approach currently used in most IP conferencing solutions. This lack of effectiveness is evident to anyone who has attempted to deliver duplicated media streams to nine people participating in a ten-party conference, for example.

NDN-RTC has been designed and implemented to further develop and explore NDN's potential for scalable low-latency audio/video conferencing. NDN-RTC provides basic functionality for publishing audio/video streams, and for fetching any streams being published which can be leveraged by desktop or web applications for establishing multi-party conferences. While NDN provides scalability advantages, NDN-RTC ensures low-latency communication. It assures that data being fetched are the most recent data available on the network. NDN-RTC is a C++ library which is built on top of WebRTC library, incorporating existing audio pipeline (including echo cancellation) and existing video codec (VP9).

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

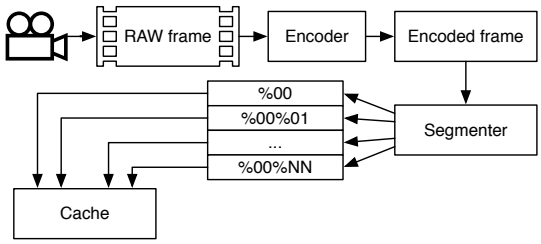
2. BACKGROUND AND PRIOR WORK

To the authors' knowledge, NDN-RTC is one of the few applications with perceptual real-time requirements being tested on an Information Centric Networking platform over a multi-hop testbed.

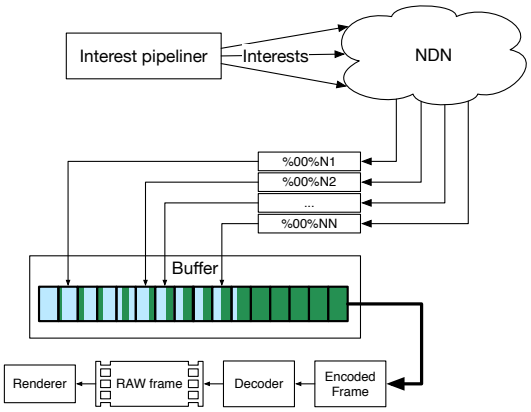
A non-real time video-streaming software solution, NDNVideo [**?**], was successfully tested and deployed on NDN, and proved high scalability. The project focused on developing random-access and live video for location-based and mixed reality applications. Another conferencing application (audio-only, however) was developed in [**?**]. It leveraged use of Mumble VoIP software, but used NDN as a transport. Initial effort for conference and user discovery was made in the recent 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.

3. DESIGN OBJECTIVES

The NDN project team uses application-driven research to explore NDN's affordances for modern applications and to refine the architecture itself. Initial goals of the NDN-RTC project are to explore low-latency audio/video communication over NDN, and to provide a multi-party conferencing application which can be used by NDN project team members across existing NDN testbeds. Additionally, this should



(a) Producer



(b) Consumer

Figure 1: NDN-RTC producer and consumer operation.

be used as a traffic generator application for future research of NDN routing algorithms, congestion control, and general testing of the network architecture ideas.

- **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.)

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 producer writes data to the socket, and consumer reads it as fast as possible) to pull-based (producer publishes data on the network at own pace, while consumer has to request data needed and manage incoming data segments).

Figure **??** presents a top-level overview of how NDN-RTC works. Local media capture and cache belong to the producer. Media is stored in the cache which provides access to the data for all incoming interests. Remote playback represents the consumer: issues interests, prepares received media (assembles video frames from segments and re-orders them) and plays it back.

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, and scaling is supported by the network.

In the case of video streaming, 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 $\approx$ 30KB, and delta frames are $\approx$ 3-7KB. Therefore, depending on the underlying Internet Protocol used (IP in the existing NDN testbed), producer may need to segment encoded frames into smaller chunks and provide clear naming conventions for consumer to fetch them.

4.2 Namespace

As there is no direct consumer-producer communication, it is producer's job to provide as much supportive information as possible, so that 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

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 producer to provide the same media stream in several copies (for instance in low, medium and high quality, so that 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 producer opts to publish it, can be used by con-

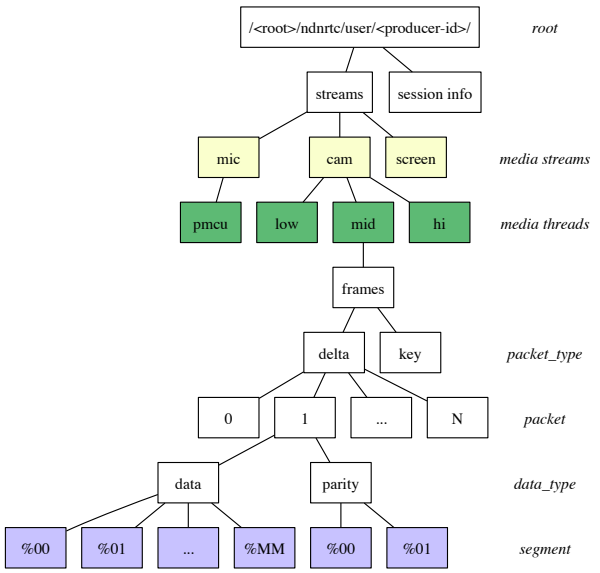
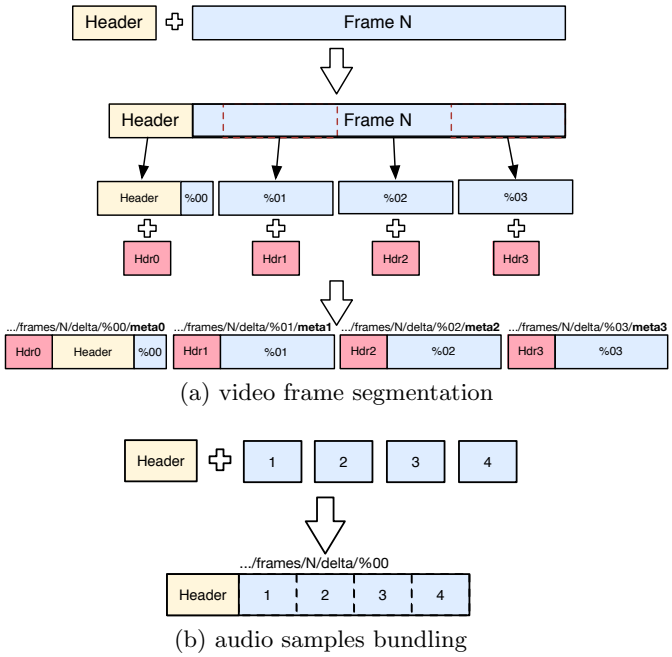


Figure 2: NDN-RTC namespace



(a) video frame segmentation

(b) audio samples bundling

Figure 3: Segmentation and bundling

sumer 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 [**?**].

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 producer's streams structure in order to fetch data successfully. In order to save consumer from traversing actual producer's name tree, which can be time-consuming and unreliable, 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. Three components are added at the end of every data segment name:

.../**segment#/playback#/paired_seq#/num_parity**

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 producer side. Such information (which could 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 consumer side may help Interest pipelining in the future. For instance, providing the average number of segments per frame type helps consumer guess 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 forward further to the producer, if the requested data has yet to be produced. In such cases, 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

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 types of headers: *Frame header* and *Segment header*. Frame header is prepended to segment #0 of each individual video frame and

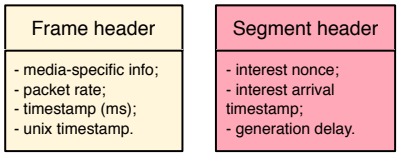


Figure 4: Frame and segment headers

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 consumer for making certain assumptions about 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 consumer a clue about how long does it take for interests to reach producer. This value is only valid when nonce value belongs to one of the consumer's interests.

Generation delay. Time interval in milliseconds between interest arrival and segment publishing. Consumer can use this value to her advantage 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 doesn't 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 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 Consumer implements more sophisticated algorithms to achieve following goals:

- ensure fetching the latest data available in the network;
- choose appropriate media stream bandwiths provided by a producer by monitoring network conditions;
- playback fetched media in correct order;
- handle network latency and packet drops.

Consumer takes into account that media packets are presented by separate segments in the network. Therefore, consumer implements mechanisms of interest pipelining and frame buffering (see Figure 1(b)). Interest pipeliner issues interests for individual segments and is controlled by some higher-level logic which achieves one out of four consumer's goals - ensures that only the latest data is fetched. Packets re-ordering, drops and latency fluctuations absorbtion are attained by the frame buffer which introduces buffering delay and re-arranges received frames in the playback order.

4.4.1 Frame fetching

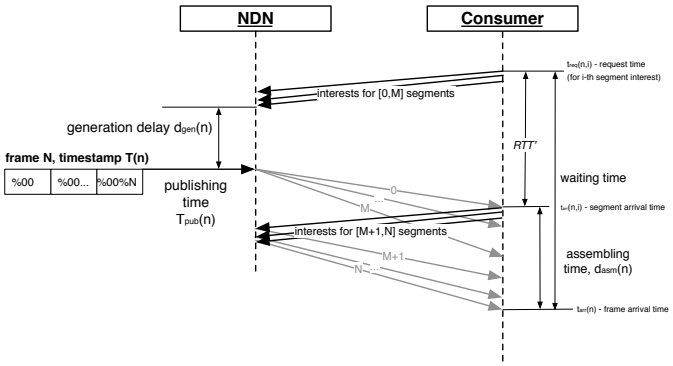


Figure 5: Fetching frame

Consumer does not know the total number of segments beforehand, unless the very first segment is fetched. In this case, consumer can retrieve metadata from the received segment and acquire total number of segments for the current frame. Therefore, at first attempt, consumer tries to make a "best guess" at the number of segments needed to fetch, by issuing M Interests (see Figure 5). If Interests arrive too early, they will be added to the producer's PIT and stay there for some amount of time d_{gen} called `generation delay`. Once encoded frame is segmented into N segments, they are published, and if $N \geq M$, Interests $0 - M$ are answered with the data which travels back to consumer. Upon receiving first data segment, consumer determines the total 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 producer). The time interval between receiving very first segment and when the frame is fully assembled is represented by d_{asm} and called `assembling time`.

Needless to say, additional round-trips for requesting missing data segments increase overall frame assembling time and chances that the frame will be incomplete by the time it should be played back. This problem could be avoided if consumer is able to make better guesses of the number of initial Interests. Therefore, the following considerations were introduced in later versions of the library:

- consumer should know what type of frame to be fetched (as average number of segments varies greatly for different frame types);
- consumer should track average number of segments per frame type.

The first consideration was implemented by introducing separate namespaces for key and delta frames. The second consideration helps consumer better guess the number of initial interests.

4.4.2 Buffering

As in traditional streaming applications, consumer uses frame buffering in order to tackle out-of-order data arrivals and network delay deviations. Consumer-side jitter buffer is also used as a place for assembling frames by segments. However, the definition of jitter buffer is extended for NDN. In a traditional push-based approach, buffer can not contain empty frame slots; they are allocated/reserved only when data arrives. A pull-based paradigm requires consumer to

request data explicitly. Therefore, after expressing Interest, consumer "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, jitter buffer's size has two measurements:

playback size - playback duration of all complete ordered frames by the moment;

estimated size = *playback size* + *number of reserved slots* × 1/*producer rate*.

The difference between estimated size and playback size (RTT') can not be smaller than the current average RTT value. In fact, keeping this value at a minimum indicates that consumer receives the most recent data with the minimal amount of outstanding Interests. Monitoring this value over time may help consumer get a better clue on the "sync" status with the producer. For example, consumer may use it during the fetching process, as will be discussed in the next section.

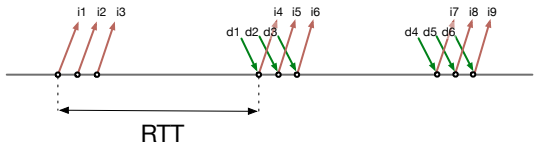
4.4.3 Interest expression control

The key challenge in a consumer-driven model for videoconferencing is to *ensure the consumer gets the latest data in a caching network*, without resorting to direct producer-consumer communication that would limit scaling. To get fresh data, which can be cached but should not be the newest available for the consumer's path, the consumer cannot rely only on using such flags as *AnswerOriginKind* and *RightMostChild*. The high frequency nature of streaming data makes no guarantees that the data satisfying those flags received by a consumer will be the most recent one. Instead, it is necessary to use other indicators to ensure that the network supplies up-to-date stream data. The basic solution is to leverage the known segment publishing rate and assume, under normal operation, that a series of old, cached samples, can be retrieved more quickly than new data. The inter-arrival delays (D_{arr}) of the most recent samples follow the publishers' generation pattern but cached data will follow interest expression temporal pattern. Therefore, by monitoring inter-arrival delays of consequtive media samples, consumers can make educated assumptions about data freshness (see Figure 6).

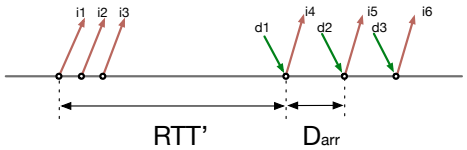
During bootstrapping, the consumer "chases" the producer and aims to exhasut network cache of historical (non-real time) segments. By incerasing the number of outstanding interests, consumer "pulls cached data" out of the network unless the freshest data start to arrive. In order to control interest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

W provides a simple mechanism which can be used to speed up or slow down interests expression. Any increase in W value makes consumer to issue more interests (Figure 7(b)), whereas decrease in W holds consumer back from sending any new interests (Figure 7(c)). Larger values of W make consumer faster reach synchronized state with producer. However, larger value means larger number of outstanding interests and larger RTT' because of longer generation delays d_{gen} for each media sample. By adjusting the value of W and observing inter-arrival delays D_{arr} consumer can find minimal RTT' value while still getting non-cached data, thus achieving best synchronization state with producer.

For more complex scenario of video streaming, consumer controls expression of "bulks" of interests instead of individual interests, because video frames are composed of several

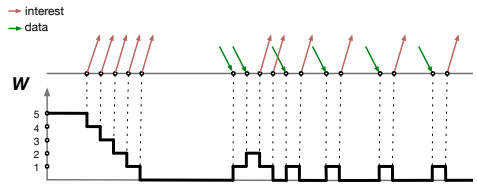


(a) bursty cached data arrival, reflects interests expression pattern

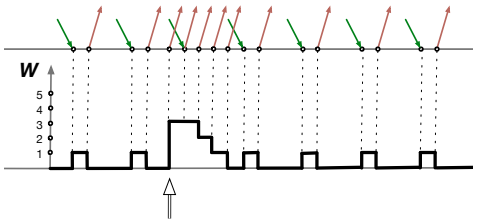


(b) stable fresh data arrival, reflects publishing pattern

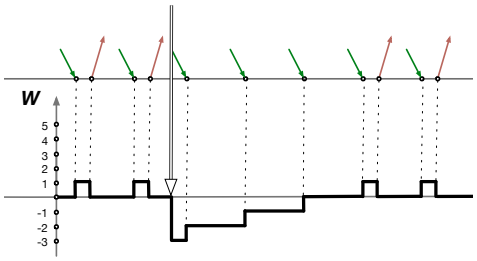
Figure 6: Getting the latest data: arrival patterns for the cached and most recent data



(a) W concept



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



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

Figure 7: Managing interest expression

segments. In this case, W is adjusted on per-frame basis, rather than per-segment. In all other respects, the same above logic can be applied.

Bootstrapping phase starts with issuing interest with enabled *RightMostChild* selector in delta namespace for audio and key namespace for video. The reason, why this process differs for video streams is that consumer is not interested in fetching delta frames without having corresponding key frame for decoding. Once initial data segment of sample with number S_{seed} has been received, consumer initializes W with some initial value N and asks for the next sample data $S_{seed}+1$ in the appropriate namespace. Upon receiving first segments of sample $S_{seed}+1$, consumer initiates fetching process described above for all namespaces (delta and key, if available). Bootstrapping phase stops when consumer finds minimal value of W which still allows receiving the most recent data - i.e. consumer reaches synchronized state with 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, so that they may 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, it is only useful when data has been published and put in the cache before interest for this data has arrived. For the missing data, the interest is forwarded to producer application which has to store it in internal pending interests table (PIT) unless requested data is ready. This functionality seems quite common for low-latency applications 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 supported currently, though 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 [**?**] is used for NDN connectivity. The WebRTC framework [**?**] 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 [**?**] library is utilized for forwad error correction support.

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

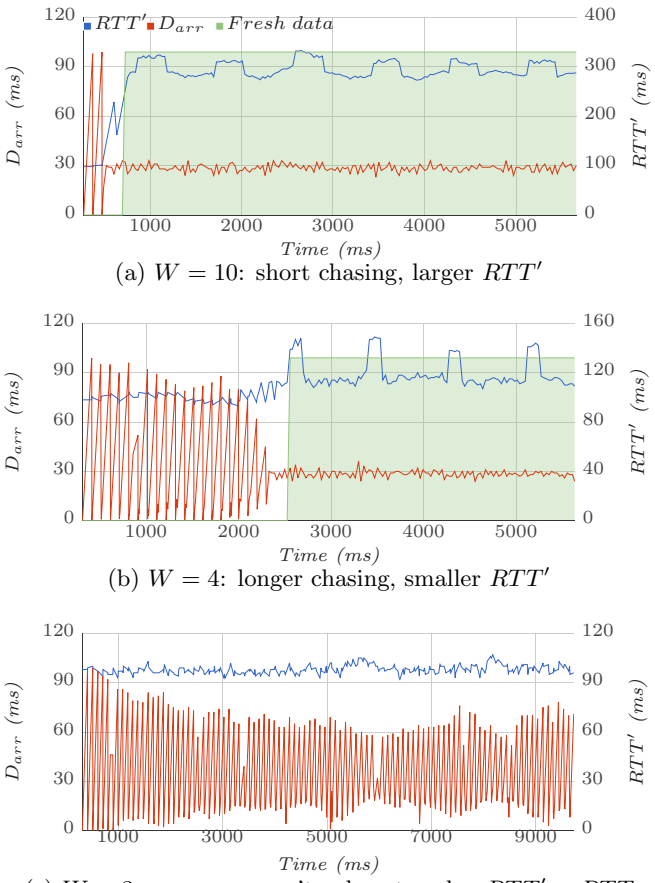
6. EVALUATION

Most tests done in practice rather than simulation.

6.1 Test setup

Include testbed topologies used, collaboration with WUSTL

6.2 Quality of experience



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

How was the end-user quality of experience by the time the paper was written What kinds of things did you add to support it.

6.3 Scaling

How did things seem to scale?

6.4 Performance of buffering and interest expression control

How did the interest expression control work, and get adjusted

How did buffering work, and get adjusted?

(Figure 8 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.

7. CONCLUSION AND FUTURE WORK

Discuss quality of experience.

Future work:

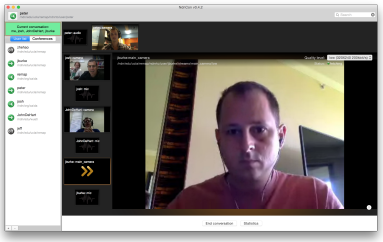


Figure 9: NdnCon screenshot [@@REPLACE]

Adaptive rate control. In the current library design, 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 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.

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.