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, NDN-RTC 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 IP 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. Signed, named 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. As a full discussion of NDN is outside the scope of this paper, please see the publications on the project website[1], including [16, 17, 8].

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

Efficient content distribution has long been a driver application for NDN research, as well as for the broader field of Information Centric Networking (ICN). Prior work in this area, including our own, is covered briefly in Section 3. However, *low-latency* applications such as "real-time conferencing" present particular design and implementation issues 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) and while trying to keep application-level latency low, is a significant challenge.

The NDN-RTC library was created to explore this arena experimentally, and it was designed, implemented, and evaluated to explore NDN's potential for scalable low-latency audio/video conferencing and "real-time" traffic 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 have also been working towards the goal of using NDN-RTC in NDN project-related videoconferences and meetings. To be useful for project communications, we needed reasonable CPU and bandwidth efficiency, echo cancellation, and modern video coding. As a result, NDN-RTC is a C++ library 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 details main project goals. Section 3 covers background and prior work. Section 4 describes the architecture of the library, designed namespace, data structures and algorithms. 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 was 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 the existing NDN testbed. We are also attempting 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. For example, existing solutions scale poorly to high numbers of producers and consumers without dedicated aggregation units.

The high-level motivation for NDN-RTC is to investigate if "real-time" content publishing can be achieved as with most other NDN data dissemination applications: The publisher 1) acquiring and transforming data, 2) naming, packetizing and signing it, and 3) passing it to an internal or external component that responds to Interests received from the "black box" of the NDN network (matched against available signed, named data chunks). And the consumer issues Interests using appropriate names and selectors to the "black box", at the rate necessary to achieve its objectives–as informed by the performance of the network in response to its requests–and reassembles and renders them. Once the namespace is defined, the publishing problem, so far, would seem 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 [14]. 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, specific design goals have been developed:

- **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 remain as future work.

3. BACKGROUND AND PRIOR WORK

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

A non-real time video-streaming software solution, NDNVideo was successfully tested and deployed over NDN, proving 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. The data chunks were named sequentially according to NDN naming conventions [15]. Additionally, the application namespace provided a time-based index which mapped individual data chunks to the media stream timeline and allowed the consumer to seek easily inside the stream. Even though the project worked well for live and stored audio/video streaming, it did not meet requirements for low-latency communication and could not be used as a conferencing solution.

Another recent attempt for audio/video streaming was made in [13]. 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 [11] which, unlike NDNVideo, operate on an 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 a constant rate by executing *consume()* call from Consumer/Producer API.

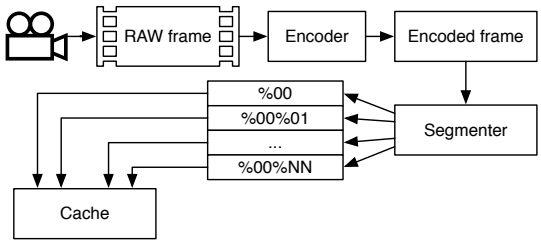
Other streaming-related works, such as [10] and [12], explored the advantages of using ICN networks for MPEG Dynamic Adaptive Streaming over HTTP. Although not related to low-latency streaming, the idea was to leverage a network's caching ability for serving chunks of video files over to multiple consumers.

An audio conferencing application was developed in [18]. 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 difficult to use for real world scenarios. 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, the decision was made for NDN-RTC to be built on top of the 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.

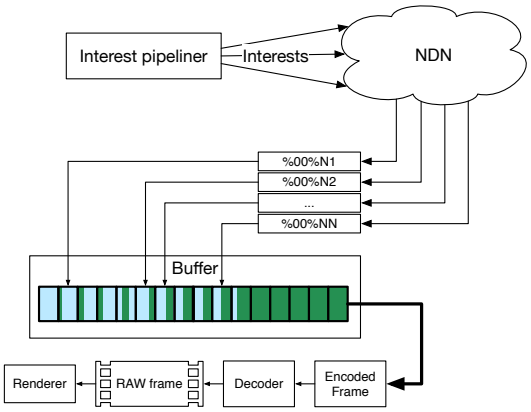
Another work [7] describing design and implementation of Voice-over-CCN worth mentioning. The tool provided similar level of quality compared to VoIP solutions with much greater scalability potential and simpler more flexible architecture.

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 its own pace, while the consumer has to request



(a) Producer



(b) Consumer

Figure 1: NDN-RTC producer and consumer operation.

data needed and to 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, and scaling is supported by the network.

In the case of video streaming, the producer uses video encoding in order to reduce the sizes 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. 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 the 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 transport's performance for delivering objects of this size, the producer may need to segment encoded frames into smaller chunks and provide clear naming conventions. Based on our current observations of performance and the prevalence of UDP as a transport for he NDN testbed, NDN-RTC currently packetizes media into segments that are less than the typical 1500 byte MTU.

4.2 Namespace

The NDN-RTC namespace defines names for media (segmented video frames and bundled audio samples), error correction data, and metadata, as shown in Figure **??**. As there is no direct consumer-producer communication, the namespace is designed to efficiently support the type of fetching operations performed by the consumer, as described below.

4.2.1 Media

The NDN-RTC producer describes published media for a given source as a collection of *media streams*. A media stream represents a flow of media data, such as video frames or audio samples, coming from a source–currently, an input device on the producer. (For now, names for streams are derived from their corresponding device information.) A typical publisher will publish several media streams simultaneously–e.g., at least camera and microphone, but also a separate stream for screenshots. The data from stream is encoded at one or more bitrates configured on the producer, so in the name hierarchy, each stream has children corresponding to different encoder instances called *media threads*. Media threads allow the producer to, for example, provide the same media stream in several quality levels, such as low, medium and high quality, so that the consumer can choose the media thread suitable for its requirements and current network conditions.

NDN-RTC packetizes encoder output directly and adds some basic metadata to each packet. Encoded media segments published under the hierarchical names described above, with video frames further separated into two domains per frame type, `delta` and `key`, each numbered sequentially and independently. The next level in the tree separates data by type, either media or parity. Parity data for forward error correction, if the producer opts to publish it, can be used by a consumer to recover frames that miss one or more segments. The deepest level of the namespace defines individual data segments. These segments are numbered sequentially, and their names conform to NDN naming conventions [15].

Audio streams, are handled differently, as their samples are smaller than the maximum payload size and there is no equivalent to the key/delta frame distinction in the audio codecs in use. All audio packets are published under the `delta` namespace. Multiple audio samples are bundled into one data packet, until the size of one data segment is reached, and published only after that.

4.2.2 Metadata

In addition to data carried in names, NDN-RTC uses both stream- and packet-level metadata. Consumers need to know the producer's specific namespace structure in order to fetch data successfully. To save consumers from traversing the producer's namespace, the producer publishes meta-information about current streams under the `session info` at the rate of 1Hz. Thus, consumers can retrieve actual information about the producer's publishing state. Additionally, data names carry further metadata as part of each packet, which can be used 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 position for current frame; this is different from the *frame#* which is a sequence number for the frame in its domain (i.e. `key` or `delta`);

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

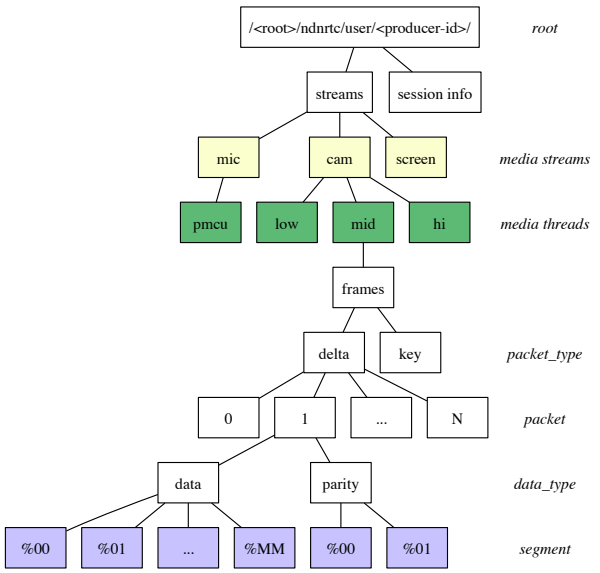
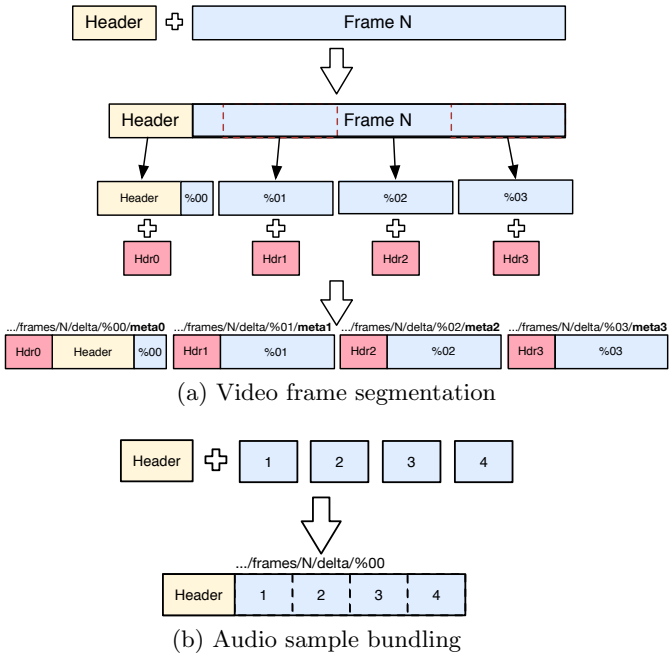


Figure 2: NDN-RTC namespace



(a) Video frame segmentation

(b) Audio sample bundling

Figure 3: Segmentation and bundling

`num_parity` - number of parity segments for the frame.

4.3 Data objects

4.3.1 Media stream

Each media packet payload consists of two types of data – media stream data and metadata. Video stream data contains actual bytes received from the library's video encoder, which represent the encoded frame. For the audio, NDN-RTC captures and encapsulates RTP and RTCP packets coming from the WebRTC audio processing pipeline, in order to obtain echo cancellation, gain control and other features, which are then fed into a similar pipeline on the consumer side for proper rendering and corrections.[3]

4.3.2 Metadata

The receiver-driven architecture of NDN-RTC and our experimental application's deliberate avoidance of explicit consumer-producer synchronization (to explore network scaling support) have shown the importance of providing sufficient meta information on the producer side. Some of this is done in the namespace, as described above. Other information is provided in the data objects themselves. Such separation is currently experimental and provides some benefits like, for example, to quickly retrieve total number of segments in the frame without the need of content verification and parsing.

Packet-level metadata is applied to video as a header prepended to each segment (see Figure 3(a)). Each audio bundle (packet) is prepended by the same frame header (see Figure 3(b)). There are two header types: *frame* and *segment*. Frame headers are prepended to segment #0 of each individual video frame and contain media-specific information (such as frame size), timestamp, current rate and Unix timestamp[4] (see Figure 4).

Additionally, as an aid to experimentation and for optimizing two-way conversations, the header also carries the producer's observations of Interest arrival that can be used by consumers to hint fetching and playback choices. The following make use of the Interest nonce value (and thus may not be as useful in larger multi-party calls):

Interest nonce: Nonce of the Interest which *first* requested this particular segment. There are three meaningful cases: 1) *Value belongs to the Interest issued previously* - consumer received non-cached data requested by previously issued Interest; 2) *Value is non-zero, but it 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 were cached on the producer side and never requested by anyone previously.

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

However, this data will not make any sense for the consumers who did not issue Interest with the nonce value stored in this data packet.

4.4 Consumer

[3]This is an artifact of the current implementation, and as features of these IP-based protocols are not used, would be eliminated in the future.

[4]Producer timestamp is *not* required but can be used for calculating actual delay between NTP-synchronized producers and consumers)

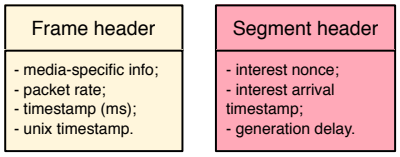


Figure 4: Frame and segment headers

The NDN-RTC architecture is receiver-driven. The consumer aims to achieve 2) choose the most appropriate media stream bandwidth from those provided by the producer (by monitoring network conditions); 3) fetch (and, if necessary, reassemble) media in the correct order for playback; 4) mitigate, as far as possible, the impact of network latency and packet drops on the viewer's quality of experience. As described above, ne of our research goals is to explore performance of the consumer in meeting these requirements without requiring direct communication with the producer.

4.4.1 Interest pipelining and Data buffering

The consumer implements Interest pipelining and data buffering (see Figure 1(b)) to fetch and reassemble video and audio data while allowing for out-of-order arrival and interest reexpression. An asynchronous Interest pipeline issues Interests for individual segments. A frame buffer handles re-ordering of packets, and informs the pipeline of its status to prompt interest reexpression.

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.

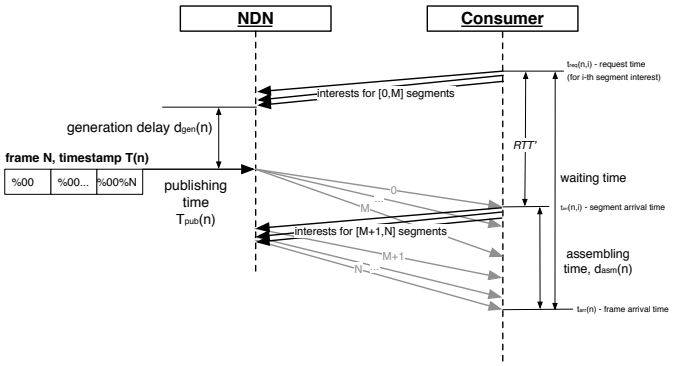


Figure 5: Fetching frame

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

4.4.3 Buffering

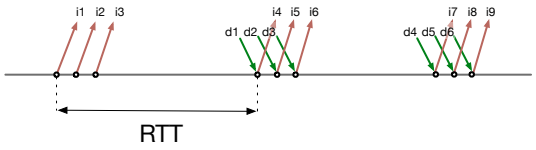
As in sender-driven delivery, the consumer uses a "jitter buffer" in order to manage out-of-order data arrivals and variations in network delay, and as a place to assemble segments into frames (see Figure 9). However, the role of such a "jitter buffer" has some NDN-specific aspects. In sender-based video 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 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, 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* which reflects an estimated size of the buffer in case if all reserved slots have data (this corresponds to the theoretical unreal case when interests are answered instantly).

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). Monitoring this value over times provides the consumer information on possible "sync" status with the producer as will be described further.

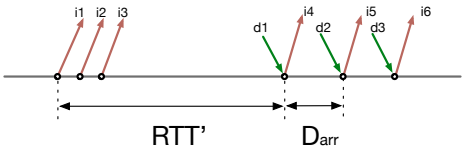
Another role played by the "jitter buffer" is the retransmission control. As presented on Figure 9, at some point inside the buffer (J milliseconds from the buffer end) there is a checkpoint, where every frame is being checked for completeness. In such cases, when incomplete frame can not be recovered using available parity data, Interests for the missing segments are re-issued.

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



(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

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

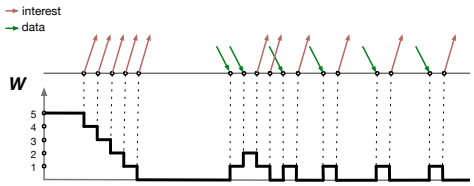
Our initial solution is to leverage the known segment publishing rate, which is available in stream-level metadata, and note that, under normal operation, old, cached samples are likely to be retrieved more quickly than new data. [5] We define the interarrival delay (D_{arr}) as the time between receipt of successive samples by 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).

The consumer operates in two modes. The *bootstrapping mode* is active when consumer initiates data fetching and tries to exhaust cached data by changing the number of outstanding Interests. After the consumer has exhausted the cache, it switches into the *playback mode* and expresses Interests at the constant rate.

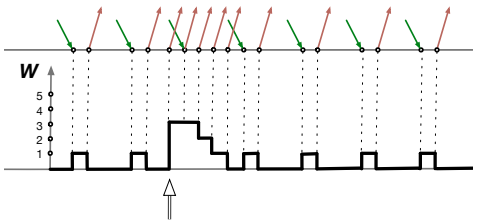
During bootstrapping, the consumer "chases" the producer and aims to exhaust network cache of historical (non-real time) segments. By increasing the number of outstanding Interests, the consumer "pulls cached data" out of the network, unless the freshest data begin to arrive.

In order to control Interest expression, a concept of "Interest demand" (further is referenced by variable W) is introduced (see Figure 7). The consumer expresses new Interests

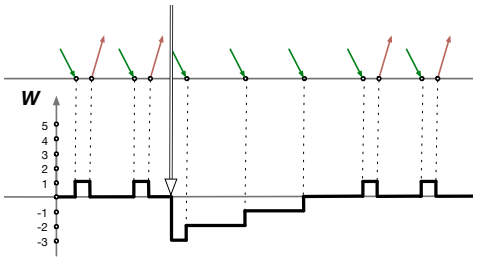
[5]If the consumer is the *only* consumer of the stream, its Interests will go directly to the publisher, which also yields the correct behavior. A more complex challenge, for further study, is when segments are inconsistently cached in different ways along the path(s) that Interests take.



(a) "Interest demand" concept, W



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



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

Figure 7: Managing Interest expression

only when the demand is positive, i.e. $W > 0$. At every moment, W indicates how many outstanding Interests should 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.

However inspired by TCP congestion window, Interest demand concept bears its own meaning for ICN networks. W is controlled by the consumer and directly influences how fast consumer can echaust the cache. Every time a new Interest is expressed, W is decremented, and when new data arrives, W is incremented, thus allowing the consumer to issue more Interests. Even though, the consumer may opt to change Interest demand in some cases as discussed below, Interest expression is driven by the incoming data.

In current design, there are two indicators which reason any changes to the W - effective RTT (RTT') and inter-arrival delay (D_{arr}). Whenever D_{arr} stops to fluctuate outrageously and begin to deviate around constant value, that means that consumer receives the freshest data available. However, it does not necessarily mean that the consumer issues Interests efficiently. Figure 11(a) shows that even though the consumer has exhausted cache rather quickly, RTT' is three times larger than the actual RTT for the network (100ms), which means that the majority of the issued Interests remain pending whlie waiting for the requested data to be produced.

The consumer makes several iterative attempts to adjust the W while bootstrapping which can be described as fol-

lows:

1. The consumer initializes default Interest demand DW for some value, sets $W = DW$ and initiates Interests expression.
2. If the consumer did not receive freshest data during allocated time (1000ms), increase Interest demand: $W = W + 0.5DW; DW = DW + 0.5DW$.
3. Whenever consumer receives freshest data (cache exhausted), decrease Interest demand: $W = W - 0.5DW; DW = DW - 0.5DW$ and wait for either of the two possible outcomes:

   - RTT' decreases and the consumer still receives the freshest data – go to step 3;
   - the consumer starts to get cached data (D_{arr} fluctuates greatly) – restore previous value for DW, increase W accordingly and stop any further W adjustments as the consumer has reached perfect sycnhornized state with the producer.

During playback, W provides a simple mechanism to speed up or slow down Interest 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 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.

The consumer continues to observe RTT' and D_{arr} in the playback mode. Whenever D_{arr} indicates that no fresh data is being received, the consumer increases Interest demand and starts the adjusting process over again to find minimal RTT' for the new conditions. Such approach helps the consumers to adjust in cases when data may suddenly start to arrive from different network hub which introduces new network RTT.

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 DW 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 and the consumer switches to the playback mode.

5. IMPLEMENTATION

NDN-RTC is implemented as a library written in C++, which is available at `https://github.com/remap/ndnrtc`. It provides a publisher interface for publishing an arbitrary number of media streams (audio or video) and a consumer interface with a callback for rendering decoded video frames in a host application. The 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 the NDN-RTC library.

NDN-RTC exploits some functionality from several third-party libraries with which it is linked. NDN-CPP [9] is used for NDN connectivity. The WebRTC framework [3] is used in two ways: 1) incorporation of the existing video codec; 2) full incorporation of the existing WebRTC audio pipeline, including echo cancellation. OpenFEC [2] is used for forward error correction support.

Some features were incorporated into the library based on our experience in this application. 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.

In addition to the library, the first desktop NDN videconferencing application NdnCon [4] was implemented on top of NDN-RTC. It provides a convenient UI for publishing and fetching media streams, text chat, and organizing multi-party audio/video conferences. It was used, along with a command-line interface for the evaluation below.

6. EVALUATION & ITERATIVE REFINEMENT

During the course of NDN-RTC's initial development, there were several application design iterations that each introduced improvements in the overall quality of experience for the end user, as well as application efficiency in terms of bandwidth and computation. Each iteration tackled problems that were revealed during tests (mostly in practice rather than simulated). These motivated namespace, application packet format and other revisions, which are reflected in the design (detailed above) and described further in this section.

6.1 Video streaming performance

A series of tests were conducted in order to assess efficiency and quality of service compared to Skype video calls. Each test was comprised of six runs of two-person, five-minute conference talks using NdnCon (the graphical conferencing application built on top of NDN-RTC):

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

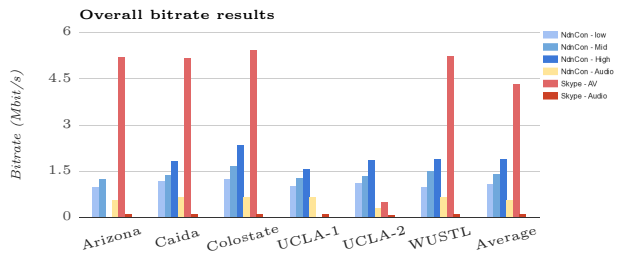


Figure 8: 2-peer conference tests compared to Skype

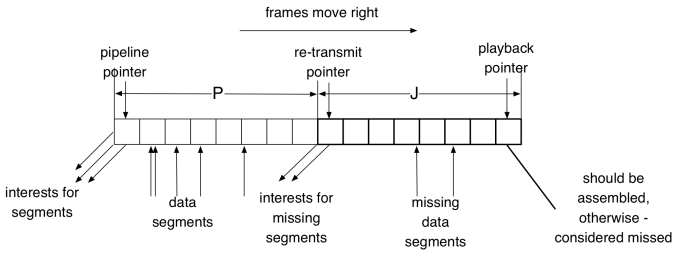


Figure 9: Frame buffer

- one run of Skype audio-only conference.

Tests were conducted across the existing NDN testbed, between the UCLA REMAP hub and six other hubs. These tests covered both one-hop and multi-hop (with several intermediate hubs) topologies.

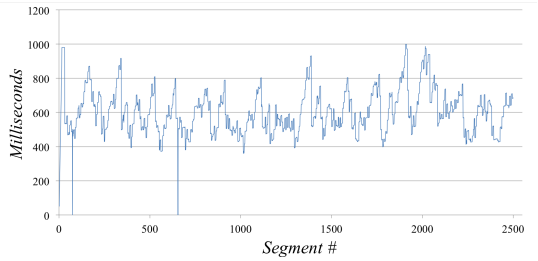
Figure 8 shows overall bitrate usage results. Whereas Skype has fully utilized link capacity between peers, and delivered higher bitrate videos, NdnCon did not adjust to the current network conditions which make adaptive rate control features highly desirable.

Actual average bitrates 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 bandwidths respectively– which can be explained by NDN packet overhead which is approximately 280-330 bytes and accounts for ≈30% of the segment size (1000 bytes). Having such large overhead makes transferring audio samples in separate segments highly inefficient. Thus, further improvements to the algorithms were made by bundling consecutive audio samples unless they filled the size of a segment. With 90 Kbit/s audio, approximately five audio samples can be added to a 1000-bytes data segment. Eventually, this improvement effectively reduced audio bandwidth(and the number of Interests on the consumer side), making it comparable to Skype audio bandwidths.

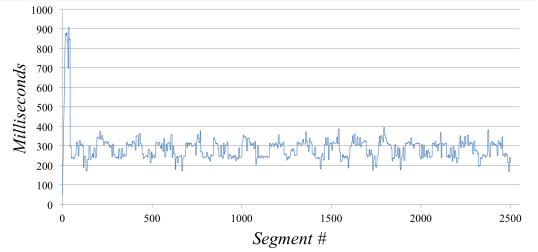
The results of such testing influenced iterative updates to the design, two of which are described below.

Separation of key and delta frame namespaces.

Video streaming performance in early versions of NDN-RTC 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 much larger than delta frames and require more data segments for delivery. In early NDN-RTC versions, this differentiation was not reflected in the producer's namespace and consumer has to issue equal number of initial Interests (M on Figure 5) regardless of the frame type. This resulted in additional round trips of missing Interests and, consequently, larger assembling times (d_{asm}) for key frames. Increased assembling time quite often caused skipping incomplete key frames, as they were not assembled by the time they should have been played out. Eventually, all the subsequent delta frames were skipped as well, which degraded the overall video streaming experience by introducing the video "hiccup" effect. Having a separate namespace for key frames enables consumers to maintain separate Interest pipelines per frame type and collect historical data on the average number of Interests required to retrieve one frame of each type in one round trip.



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



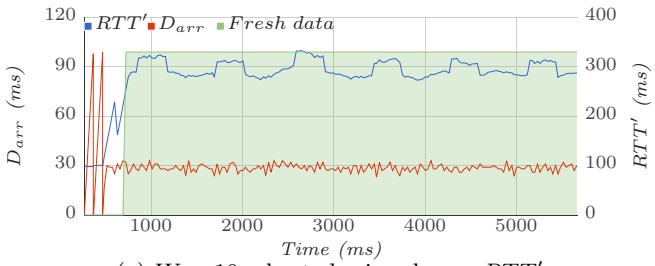
(b) $d_{gen} \approx 310ms$ - no redundant retransmissions

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

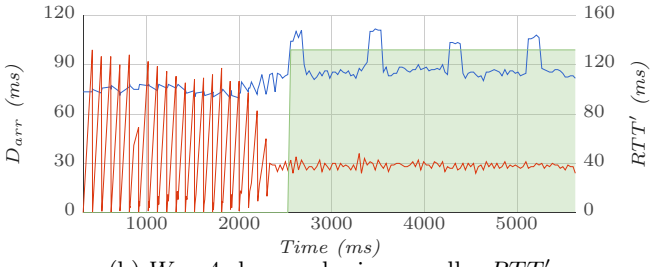
6.2 Consumer-Producer synchronization

Bootstrap behavior.

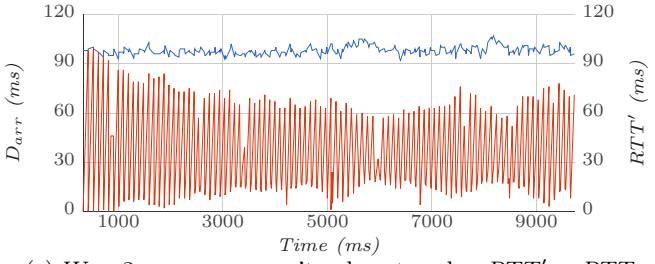
In previous library versions, the consumer "chased" the producer's time-series data by exhausting cached data and issuing a large number of outstanding Interests. However, there was no mechanism for the consumer to figure out how early those Interests are issued and whether Interest expression should be postponed in order to eliminate time outs. For two similar test runs (one-hop topology), the number of timed out Interests and re-transmissions varied greatly (either ≈1% or ≈50%). One was due to an incorrect consumer's synchronization with the producer; Interests were issued too early, so they timed out before any data had been produced. This problem can be solved by increasing the Interests' lifetime. However, for previous library versions, the mechanism for buffering (see Figure 9) dictated the Interests' lifetime. In fact, in order to maintain the re-transmission checkpoint,



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



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



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

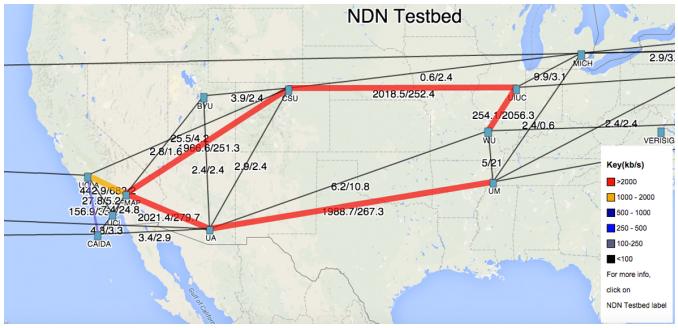
Figure 11: Larger W decreases "chasing" phase, but increases RTT' for the same network configuration ($RTT \approx 100ms$)

all Interests entering the buffer had a lifetime equal to half of the current buffer size. This approach resulted in unavoidable Interest time outs, in the cases when the consumer issued Interests far too early (before the actual data was produced).

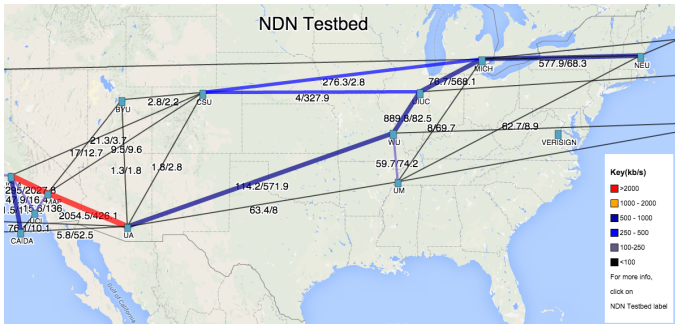
For the current version of the library, the re-transmission checkpoint is placed at RTT milliseconds from the end of the buffer ($J = RTT$ on the Figure 9). This, together with an updated NFD re-transmission strategy [5], allows for larger Interests' lifetimes.

Moreover, the problem described above can not occur if the consumer knows that it is issuing Interests too early. The chasing algorithm in older library versions was exhausting the network cache too aggressively; Interests were issued constantly until they filled up the buffer. After that, frames were retrieved at the producer's rate, and more Interests were issued unless the consumer started to receive the most recent data. This approach suffers from a lack of knowledge about issued Interests and data generation delay, and RTT' are not taken into account.

With the introduction of the W concept, the consumer has full control of the Interest expression. Figure 11 shows how a larger value of W helps to exhaust the cache more quickly. The number of outstanding Interests is controlled



(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

by a consumer and directly influences how fast consumer can "chase" the producer. Thus, the consumer can control the "agressiveness" of cache exhaustion directly.

6.3 Multi-party use

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

Another attempt to test multi-party conferencing ability included four peers, each publishing three video streams and one audio stream and fetching one video and one audio stream from each of the other participants. Participants were distributed across four NDN testbed hubs - UCLA, REMAP, CAIDA and WashU (Figure 12(b)). These results were also generally satisfying. However, for one user (connected to the WashU hub), audio cutoffs occurred more often than for the other participants. The root causes of this have yet to be revealed. Many-to-many test scenarios are complex, and will be included in future work.

7. CONCLUSION AND FUTURE WORK

Discuss quality of experience.

Future work:

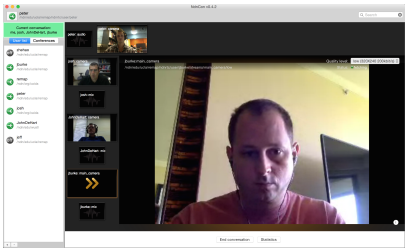


Figure 13: NdnCon screenshot [@@REPLACE]

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 the consumer to select the most appropriate stream for current network conditions. However, the selection is made manually and depends on the 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, the consumer will have more freedom for adapting media streams to the current network. This opportunity will be explored and added in 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 N. Briggs. Voccn: voice-over content-centric networks. *Proceedings of the âĂę*, 2009.

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

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

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

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

[12] 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.

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

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

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

[16] 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.

[17] 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.

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