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 the "thin waist" of the Internet 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 "Interest" packets for these "Data" packets by name. Signed, named Data 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 [7]. 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 [20, 21, 9].

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" videoconferencing present particular design and implementation issues that have been less 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. It was designed, implemented, and evaluated to explore NDN's potential for scalable low-latency audio/video conferencing and "real-time" traffic more generally. The project is ongoing; 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 providing 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 performance. As a result, NDN-RTC is built in C++ for performance, on top of the NDN Common Client Library [16]. It leverages the widely-used WebRTC library, incorporating its existing audio pipeline (including echo cancellation) and 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

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

to refine the architecture itself. Though it is based on what was learned from our NDNVideo project [7], NDN-RTC is a clean slate design with new goals. The initial objectives have been 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 sender-driven peer-to-peer communication channels for media. However, existing solutions scale poorly to high numbers of producers and consumers without dedicated aggregation units. In these scenarios, they face implementation challenges and inefficiencies related to the connection-based approach currently used in most IP conferencing solutions.

The motivating concept of NDN-RTC as a research experiment is to use it to investigate if "real-time" content publishing can be achieved in a manner consistent with most other NDN data dissemination applications, described as follows.

Bidirectional communication is achieved by having each node participate as a publisher and consumer. The publisher 1) acquires and transforms media data, 2) names, packetizes and signs it, and 3) then passes the packets to an internal or external component that responds to Interests received from the "black box" of the NDN network with signed, named data chunks. The consumer issues Interests with appropriate names and selectors to that "black box" of the NDN network, at the rate necessary to achieve its objectives and be a good citizen of the network[2]–as informed by the performance of the network it observes in response to its requests. It reassembles and renders them.

Once the namespace is defined, the publishing problem in this scenario (and in practice, at least so far) is relatively straightforward. Complexity is at the the consumer, which must determine what names to issue at what rate, to get the best quality of experience for the application. For real-time conferencing, this means low-latency access to the freshest data that the "black box" of the NDN network can deliver to a given consumer.

Conference setup and multi-party chat could be handled by applying techniques such as those developed in ChronoChat [18]. Multiple bitrates for consumers to use in adaptation are initially handled by publishing in multiple namespaces corresponding to multiple bitrates.[3]

Based on this high-level concept, specific design goals were developed:

- **Low-latency audio/video communication.** The library should be capable of maintaining low-latency (approx. 250-750ms) communication for audio and video, similar to consumer videoconferencing applications such as Microsoft Skype, Cisco WebEx and Google Hangouts.
- **Adaptive bitrate.** The library and namespace should support multiple bitrates (from which consumers will select) and lay the ground work for research on adaptive schemes in the future.
- **Multi-party conferencing.** Publishing and fetching several media streams simultaneously should be straightforward.
- **Passive consumer & cacheability.** There should be no explicit negotiation or coordination between publisher and consumer for the media transmission itself, to allow future explorations of highly scalable consumer to producer ratios.
- **Data verification.** The library should provide content verification using existing NDN signature capabilities.
- **Encryption-based access control.** While not implemented in the current version, nothing should preclude the addition of encryption-based access control in the future, including the use of broadcast and group encryption schemes.

Due to the fact that NDN-RTC is first of its kind "RTC-over-NDN" implementation attempt, it relies on a number of theoretical assumptions about the network, cache and low-latency data delivery. These assumptions are presented throughout the paper and reflect authors' understanding of data delivery in ICN. Nevertheless, initial testing results lay out the ground for research works in such ICN areas as congestion control, cache exhaustion, real-time data distribution, etc. We hope, that NDN-RTC will play essential role in these researches as an exploratory tool.

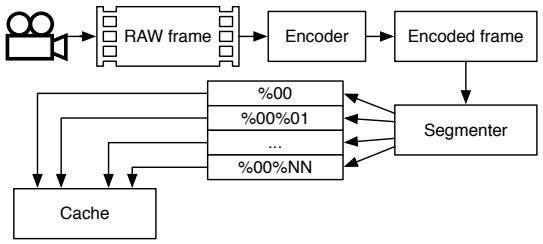
3. BACKGROUND AND PRIOR WORK

Most video streaming work on ICN has focused on streaming playout with buffers larger than what can be used for videoconferencing. For example, NDNVideo was successfully tested and deployed over NDN [7], scaling to approximately one thousand consumers from a single, straightforward publisher over plain-vanilla NDN. [6] The project focused on developing random-access pre-recorded and live video for location-based and mixed reality applications. Audio and video content, sliced into data chunks, was published into an NDN repository for online and offline access. The data chunks were named sequentially according to NDN naming conventions [19]. 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 pre-recorded media streaming, it did not meet requirements for low-latency communication, and its architecture was not immediately extensible for use as a conferencing solution.

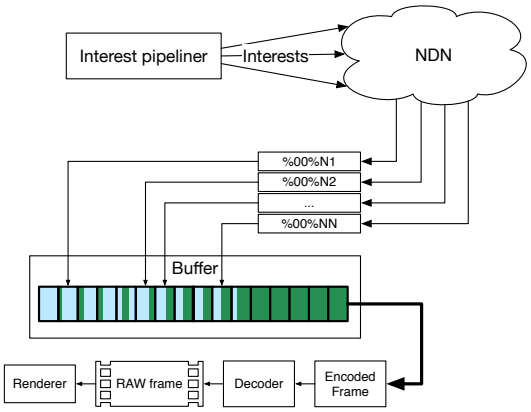
This work has been extended by others in [17], which describes the details of two applications, NDNlive and NDNtube, that are designed to work with the latest NFD (NDN Forwarder Daemon) and use repo-ng [1] for offline media storage. These applications employ a new API for application developers, the Consumer/Producer API [12], which works with higher-level "Application Data Units" rather than Interests/Data packets. At the moment of publication of current work NDNlive was still under development, however the whole project does not address real-time audio/video conferencing. NDN-RTC on the other hand, was developed from the ground up with low-latency restriction in mind which influenced every aspect of design as will be discussed further.

[2]While work on congestion control and defining proper behaviour of such applications on the network is underway, it is future work with respect to this paper.

[3]Adaptive solutions based on scalable coding present some additional challenges that remain as future work.



(a) Producer



(b) Consumer

Figure 1: NDN-RTC producer and consumer operation.

Other streaming video work includes [11] and [14], which explore the advantages of using ICN networks for MPEG Dynamic Adaptive Streaming over HTTP. Although not directly related to low-latency streaming, these works also leverage ICN networks' caching ability for serving chunks of video files efficiently to multiple consumers.

The Voice-over-CCN project [8] was an early exploration of real-time communication over ICN, providing a similar level of quality compared to VoIP solutions with much greater scalability potential and simpler, more flexible architecture. Early in the NDN project, an audio conferencing application was developed in [22]. It leveraged use of Mumble VoIP software and successfully used NDN as a transport. Initial effort for conference and user discovery was made in this work as well, which was continued in ChronoChat (mentioned above). However, echo cancellation did not work well, which made it difficult to use in real world scenarios. Therefore, we elected to build NDN-RTC on top of the WebRTC library, despite the additional, significant implementation complexity, in order to use its audio-processing capabilities and video codecs, and potentially give an opportunity for easier integration with supported web browsers. No significant modifications are needed to WebRTC as used in NDN-RTC, which means that so far we have been able to incorporate new versions of that library as needed.

4. APPLICATION ARCHITECTURE

There are two application roles in NDN-RTC: producer and consumer. In bidirectional communication, applications play both roles, but a variety of other multi-party and one-to-many scenarios can be achieved with the same abstraction. The paradigm of real-time communication shifts from the sender-driven approach of the IP network, where the producer writes data to a destination host and the consumer reads it as it arrives, to a receiver-driven approach, in which the producer publishes data to network-connected storage at its own pace, while the consumer requests data as needed and manages the relationship between outgoing Interests, incoming data segments, and buffer fill.

4.1 Producer

The producer's main tasks are to acquire video and audio data from media inputs, encode them, *name them*, marshal the named data into network packets, sign the packets, and store them in an application-level cache[4] that will asynchronously respond to incoming Interests. In this way, flow control responsibility shifts to the consumer, and scaling can be supported by network caches downstream rather than publishing infrastructure at the application level.

4.2 Namespace

A primary design question is how the data should be named so it can be retrieved by the consumer with desired properties. The NDN-RTC namespace defines names for media (segmented video frames and bundled audio samples), error correction data, and metadata, as shown in Figure 2. 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 introduced in Section 4 and detailed in Section 4.4.

4.2.1 Media

The NDN-RTC producer abstracts the 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., a camera and a microphone, but also a separate stream for screenshots. The data from a 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 the WebRTC video encoder output directly. Encoded media segments published under the hierarchical names described above, with video frames further separated into two namespaces per frame type, `delta` and `key`, each numbered sequentially and independently. For reasons described in Section 6, the namespace distinguishes in this way between two types of encoded video frames, key and delta frames. Key frames 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. As needed, frame data

[4]Currently, this cache is provided to the application by the NDN-CCL library.

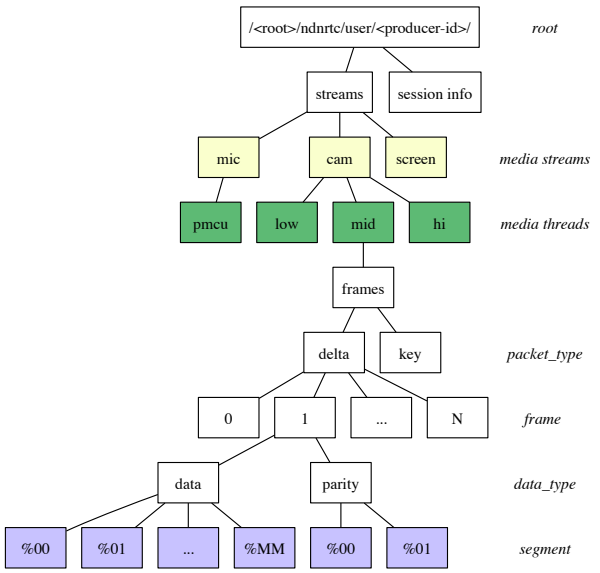


Figure 2: NDN-RTC namespace

is segmented with continuous sequence numbers per frame time.[5]

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. In the case of both parity and regular frames, the deepest level of the namespace defines individual data segments. These segments are numbered sequentially, and their names conform to NDN naming conventions [19].

Audio streams are handled more simply, as their samples are much 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

NDN-RTC uses both stream-level 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 current 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_pos/paired_seq#/num_parity`

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

`playback_pos` - 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);

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

Metadata in the name, rather than in the packet, is expected to be useful for application components or services that may not need to understand or with to decode the packet payload.

4.3 Data objects

The producer generates signed data objects from input media streams and places them in an in-memory, application-level cache. These objects contain stream data and metadata.

4.3.1 Media stream

Each media packet payload consists of two types of data – a chunk from the media stream and metadata that describes it. Video stream data contains raw bytes received from the WebRTC library's video encoder, which represent the encoded frame. For 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.[6]

4.3.2 Metadata

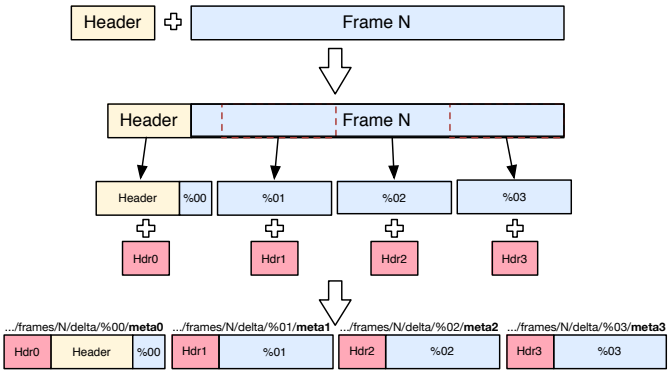
The receiver-driven architecture of NDN-RTC and our experimental approach's deliberate avoidance of explicit consumer-producer synchronization (to explore network scaling support) have suggested the importance of providing sufficient meta information from the producer. The most critical and potentially most useful for a variety of applications and services is provided in the namespace, as described above. Other information is provided in the data objects themselves. Such separation is currently experimental and provides some benefits, e.g., quickly being able to retrieve the total number of segments in the frame without the need for content decoding or knowledge of the exact payload format.

Packet-level metadata is applied to video as a header prepended to segment #0 (see Figure 3(a)). Each audio bundle (packet) is prepended by the same frame header, as seen in Figure 3(b). There are two header types: *frame* and *segment*. The frame header contains media-specific information (such as frame size), timestamp, current rate and Unix timestamp[7] (see Figure 4).

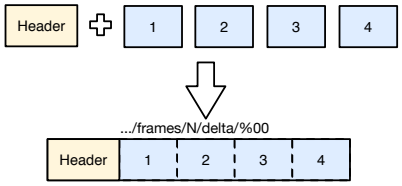
[5]For example, the average sizes of frames for 1000 kbps stream using VP8/VP9: key frames are ≈ 30KB, and delta frames are ≈ 3-7KB. Therefore, depending on the underlying transport's performance for delivering objects of this size, the producer may need to segment encoded frames into smaller chunks and name them in a way that makes reassembly straightforward. Based on our current observations of performance and the prevalence of UDP as a transport for the NDN testbed, NDN-RTC currently packetizes media into segments that are less than the typical 1500 byte MTU.

[6]This is an artifact of the current implementation to benefit from the full audio pipeline of WebRTC; features of these protocols are not used, would be eliminated in the future.

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



(a) Video frame segmentation.



(b) Audio sample bundling.

Figure 3: Segmentation and bundling

Additionally, as an aid for experimentation and for optimizing two-way conversations, the segment header also carries the producer's observations of Interest arrival that can be used by consumers to hint fetching and playback choices. The latter makes use of the Interest nonce value, and thus may not be as useful in larger multi-party calls. For this reason, we consider these primarily for debugging and experimental purposes at this time:

Interest nonce: Nonce of the Interest which *first* requested this particular segment. Example interpretations include the following: 1) Value belongs to an 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.

Interest arrival timestamp: Timestamp of the Interest arrival. Monitoring publisher arrival timestamps may give consumers information 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 again only valid when the nonce value belongs to one of the consumer's Interests.

4.4 Consumer

While the data is published in the above scheme, in NDN-RTC's receiver-driven architecture, the consumer then aims to 1) choose the most appropriate media stream bandwidth from those provided by the producer (by monitoring network conditions); 2) fetch (and, if necessary, reassemble) media in the correct order for playback; 3) mitigate, as far as possible, the impact of network latency and packet drops on the viewer's quality of experience.

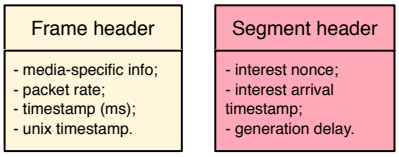


Figure 4: Frame and (experimental) segment headers.

4.4.1 *Interest pipelining and Data buffering*

The consumer implements Interest pipelining and data buffering, as shown in 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. Independently, 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, as illustrated in 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}. Conceptually, we wish to keep this interval low, to avoid accumulating outstanding Interests with short lifetimes, but we also do not wish to have Interests arrive after data is published, as this increases latency from the end-user's perspective. 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 from the metadata 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 low for the frame as a whole. 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 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

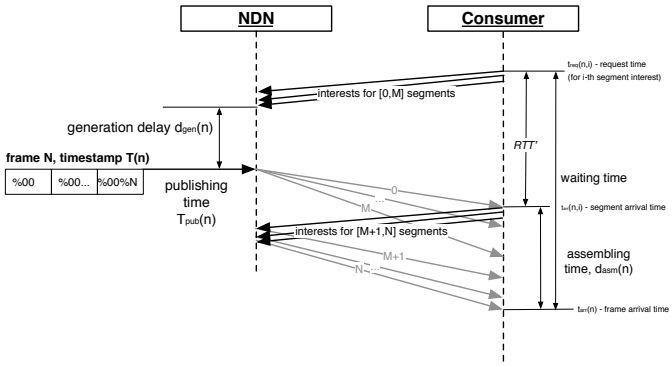
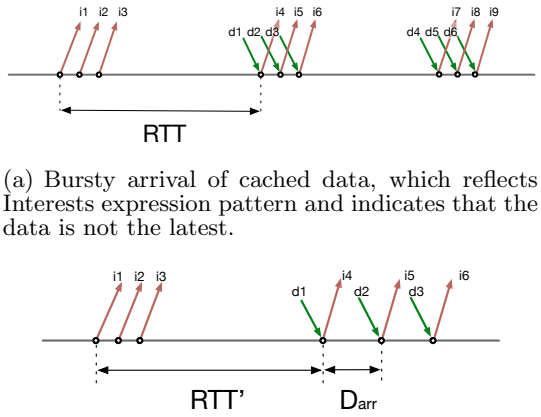


Figure 5: Fetching frame



(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

the two frame types. Additionally, the consumer tracks the average number of segments per frame type, adapting its estimates over time. A similar process is used for fetching audio, though for now, audio bundles are represented by just one segment.

The presence of valuable interest-level metadata in each data chunk requires consumer to operate on segment level and makes it complicated to abstract fragmentation to a lower layer (i.e. Consumer-Producer API [12]).

4.4.3 Buffering

As in sender-driven delivery, the consumer uses a "jitter buffer" 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 buffer has some NDN-specific aspects. In sender-driven video delivery, buffer slots can be allocated per segment using sequential numbering. A pull-based paradigm requires the consumer to request data by name explicitly, however, and organize it by frame as well as segment. Therefore, after expressing an Interest, the consumer "knows" that new data is coming, and a named frame slot can be reserved in the buffer, though the number of segments is not known. 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.

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 below.

Another role played by the "jitter buffer" is retransmission control. As shown in 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. At the checkpoint, 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

This section explores current mechanisms for Interest expression and re-expression in more detail. A key challenge in a consumer-driven model for videoconferencing in a caching network appears to be how to ensure the consumer gets the latest data, without (per our design goal) resorting to direct producer-consumer communication.

To get fresh data, the consumer cannot rely on using such flags in the protocol as *AnswerOriginKind* and *RightMostChild*. The frame period for streaming video is of the same order of magnitude of network round trip time, suggesting 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 and receiving the most up-to-date stream data possible given its (potentially evolving) network connectivity.

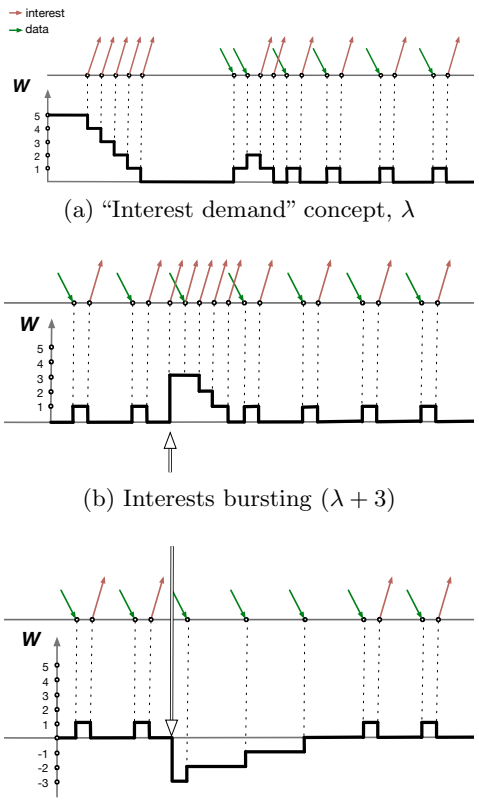
Our current solution is to leverage the known segment publishing rate, which is available in stream-level metadata, and note that, under normal operation, old, cached samples are likely to be retrieved more quickly than new data. [8] We define the **interarrival delay** (d_{arr}) as the time between receipt of successive samples by a given consumer.

The consumer expects that delays in the most recent samples follow the publishers' generation pattern, but cached data will follow the Interest expression's temporal pattern. Therefore, by monitoring inter-arrival delays of consecutive media samples and comparing them to the timing of its own Interest expression and the expected generation pattern, consumers can estimate data freshness (see Figure 6).

NDN-RTC interest expression is managed in two modes. The *bootstrapping mode* is active when a consumer first 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*, described further below.

During bootstrapping, the consumer "chases" the producer

[8] 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, λ

(b) Interests bursting ($\lambda + 3$)

(c) Interests withholding ($\lambda - 3$)

Figure 7: Managing Interest expression

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, the NDN-RTC consumer tracks a quantity called "Interest demand", λ, which can be interpreted as how many outstanding Interests should be sent at the current time (see Figure 7). The consumer expresses new Interests when the demand is positive, i.e. $\lambda > 0$. For example, before the bootstrapping phase, the consumer initializes λ 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. In playback, every time a new Interest is expressed, λ is decremented, and when new data arrives, λ is incremented, thus enabling the consumer to issue more Interests.[9]

Bootstrapping. In current design, there are two experimentally determined indicators that are used by the consumer to adjust λ: effective RTT (RTT') and inter-arrival delay d_{arr}). As described above, at bootstrapping (and re-acquisition), the consumer interprets d_{arr} stabilization around a relatively constant period, that means that consumer receives the freshest data available from the network. However, it does not necessarily mean that the consumer issues Interests efficiently. Figure 10(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 while waiting for the requested data to be produced.

The consumer makes several iterative attempts to adjust λ during bootstrapping, which can be described as follows:

1. The consumer initializes Interest demand with λ_d, and initiates Interests expression.
2. If the consumer did not receive freshest data during allocated time[10], increase Interest demand: $\lambda = \lambda + 0.5\lambda_d; \lambda_d = \lambda_d + 0.5\lambda_d$.
3. Whenever consumer receives freshest data (cache exhausted), decrease Interest demand: $\lambda = \lambda - 0.5\lambda_d; \lambda_d = \lambda_d - 0.5\lambda_d$ and wait for either of the two possible outcomes: a) RTT' decreases and the consumer still receives the freshest data – go to step 3; b) the consumer starts to get cached data (d_{arr} fluctuates greatly) – restore previous value for λ_d, increase λ accordingly and stop any further adjustments as the consumer has achieved sufficient synchronization with the producer.

One should note that λ is an actual counter of how many Interests can be issued more and λ_d represents total number of outstanding Interests, therefore $\lambda_d - \lambda$ shows how many outstanding Interests consumer has issued at any given point in time. The way λ and λ_d are adjusted was determined empirically and may be a good topic for further research.

Note that bootstrapping begins 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 λ with initial value λ_d 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). The bootstrapping phase stops when the consumer finds the minimal value of λ which still allows receiving the most recent data and the consumer switches to the playback mode.

Playback. During playback, λ provides a manageable mechanism to speed up or slow down Interest expression, coupling the asynchronous interest expression mechanism with the status of the playback buffer. An increase in λ value makes the consumer issue more Interests (Figure 7(b)), whereas any decrease in λ holds the consumer back from sending any new Interests (Figure 7(c)). Larger values of λ 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 λ and observing inter-arrival delays d_{arr}, the consumer can find minimal RTT' value while still getting non-cached data, adapting towards a loose synchronization 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

[9]While inspired by the TCP congestion window, the Interest demand, as currently employed in NDN-RTC, may play a different role in ICN networks, which we are exploring experimentally in this application.

[10]In the currently implementation, 1000ms.

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.

Interest batches. Practically, for video, the consumer controls expression of "batches" of Interests rather than individual Interests, because video frames are composed of several segments. λ is adjusted on a per-frame basis, rather than per-segment.

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 API for publishing an arbitrary number of media streams (audio or video) and a consumer API with a callback for rendering decoded video frames in a host application. NDN-RTC does not provide conference call setup functionality. This task was intentionally left out to be solved be a host app.

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 builds on functionality from several third-party libraries with which it is linked. NDN-CPP [10] 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.

To demonstrate and evaluate the library, a desktop NDN videoconferencing application, NdnCon, [4] was implemented on top of NDN-RTC. It provides a convenient user interface 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 design iterations that each introduced improvements in the overall quality of experience for the end user, as well as in application efficiency in terms of bandwidth and computation. Each iteration tackled problems that were revealed during tests. These motivated namespace, application packet format and other revisions, which are reflected in the design detailed above and described further in this section.

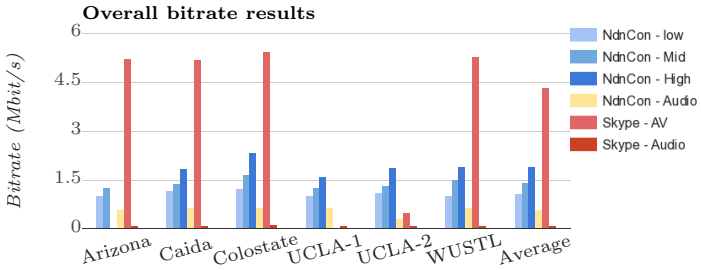


Figure 8: 2-peer conference tests compared to Skype

6.1 Video streaming performance

A series of tests were conducted in order to assess bandwidth usage and perceived quality 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: a) three runs of audio+video with low, medium and high video bandwidths settings (0.5, 0.7 and 1.5 Mbit/s accordingly); b) one run of audio-only conference; c) one run of Skype audio+video conference; d) 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) paths.

Figure 8 shows overall bitrate usage results. Whereas Skype has adapted to use 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 as future work. Early in these tests, audio sample bundling was quickly introduced in NDN-RTC to reduce 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. In early NDN-RTC versions, the difference in size, and thus segments, of key frames and delta frames was not reflected in the producer's namespace and consumers were forced 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 that resulted in missed playout deadlines. 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.

6.2 Consumer-Producer synchronization

Bootstrap behavior. In initial library versions based on the approach taken in NDNVideo, the consumer "chased"

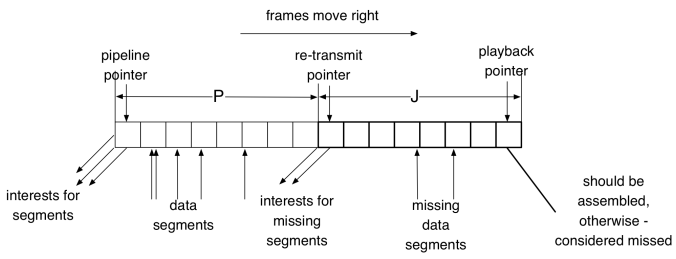


Figure 9: Frame buffer

the producer's time-series data by exhausting cached data via issuing a large number of outstanding Interests. However, there was no mechanism for the consumer to figure out when to issue those Interests and whether Interest expression should be postponed in order to reduce timeouts. 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 was addressed by increasing the Interests' lifetime.

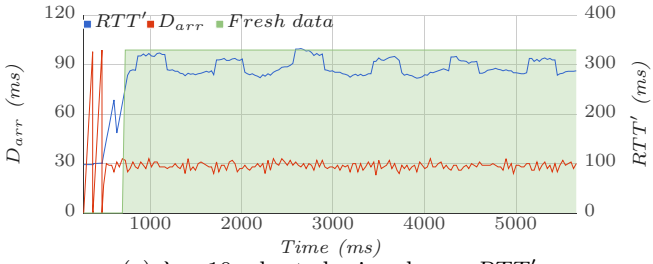
However, for previous library versions, the mechanism for buffering (see Figure 9) dictated the Interests' lifetime. In fact, in order to maintain the re-transmission checkpoint, all Interests entering the buffer had a lifetime equal to half of the current buffer size. This approach resulted in unavoidable Interest timeouts, in the cases when the consumer issued Interests far too early, before the actual data was produced. This was further complicated by forwarding strategies in NFD that did not handle consumer-initiated retransmission over short periods.

For the current version of the library, the re-transmission checkpoint is placed at a time estimated to be the effective RTT 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. In fact, current implementation does not rely on Interests' lifetimes anymore and every Interest is set to have 2000 ms lifetime.

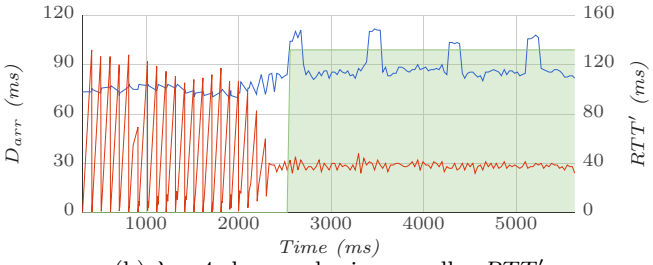
Moreover, the problem described above can not occur if the consumer knows that it is issuing Interests too early. The chasing algorithm in older library versions was exhausting the network cache too aggressively; Interests were issued constantly until they filled up the buffer. With the introduction of the λ concept, the consumer has more useful control of the Interest expression. Figure 10 shows how a larger value of λ helps to exhaust the cache more quickly. The number of outstanding Interests is controlled by a consumer and directly influences how fast consumer can "chase" the producer. Thus, the consumer can control the "agressiveness" of cache exhaustion.

6.3 Multi-party use

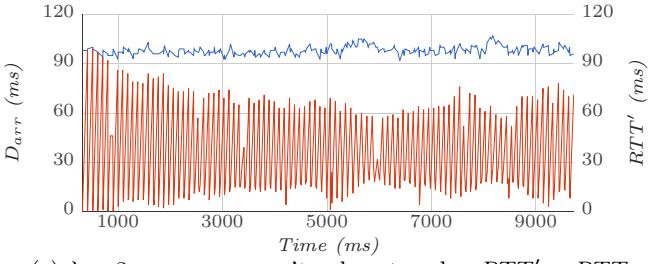
Initial attempts to deploy the NdnCon conferencing application were made in early 2015. NdnCon was used to stream an NDN seminar over the existing NDN testbed. An audio/video bridge was set up using third-party tools, such as Soundflower and CamTwist Studio, allowing cap-



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



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



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

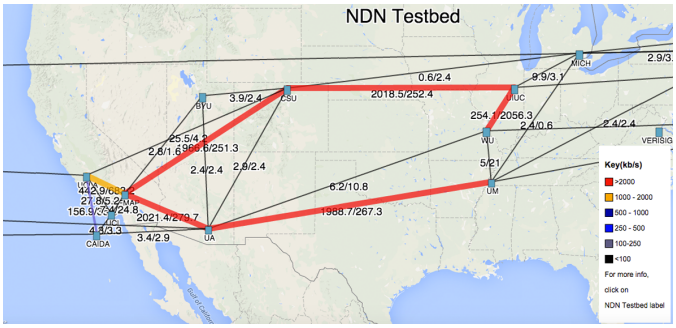
Figure 10: Larger λ decreases "chasing" phase, but increases RTT' for the same network configuration ($RTT \approx 100ms$)

tured screen and audio feeds from existing IP-based conferencing tools to be simulcast. Figure 11(a) shows an example of instantaneous 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.

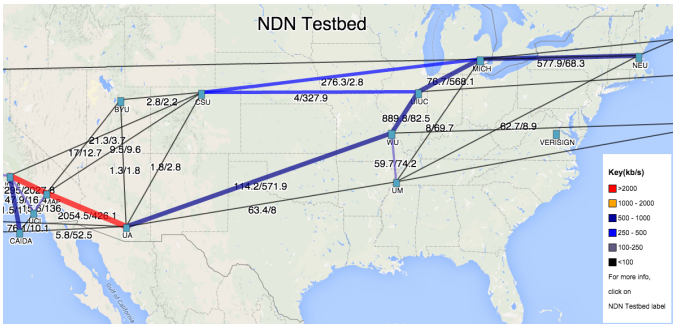
Other tests of multi-party conferencing ability included four peers, each publishing three video streams and one audio stream and fetching one video and one audio stream from each of the other participants. Participants were distributed across four NDN testbed hubs - UCLA, REMAP, CAIDA and WUSTL (Figure 11(b)). These results were also generally satisfying. However, for one user, audio cutoffs occurred more often than for the other participants. The root causes of this have yet to be found. Many-to-many test scenarios are complex, and will be included in future work.

7. CONCLUSION AND FUTURE WORK

This paper presents the design, implementation, and initial experimental evaluation of NDN-RTC, a low-latency videoconferencing library built on Named Data Networking. Our approach to this project has been experimentally driven so far, and has generated a functional low-latency streaming tool that we can now use a platform for exploring important



(a) NDN testbed utilization during biweekly NDN seminar using NdnCon for simulcast.



(b) NDN testbed utilization during 4-peer call between UCLA, REMAP, WUSTL and CAIDA hubs.

Figure 11: NDN testbed utilization during one-to-many and many-to-many scenarios.

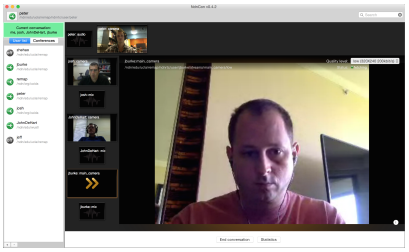


Figure 12: NdnCon screenshot.

design challenges in real-time media over NDN. This is a rich area, and some of the future work that we have identified includes:

Scalability tests. NDN-RTC has proven the viability of assumptions taken to implement low-latency media streaming over NDN. However, more well-established tests are needed to question the assumptions and explore the limits of scalability.

Adaptive rate control. In the current design, the producer can choose to publish the same video stream at different bitrates, thus requiring the consuming user to manually select the best stream but laying the groundwork for adaptive rate control as a near-term effort. Such work will need to determine if monitoring of d_{arr} and other approaches described above can address challenges of such adaptation over ICN, as suggested in papers such as [15].

Congestion control. Congestion control algorithms and their impact on the receiver-driven design of NDN-RTC are an open challenge that requires collaboration between application developers and architecture researchers.

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.

Scalable video coding. A more efficient way to relieve the producer from having to publish multiple copies of the same content at different bandwidths may be to use scalable video coding. By reflecting SVC layers in the namespace, the consumer will have more freedom for adapting media streams to the current network. Just as with audio, the SVC base layer may need to be prioritized.

Encryption-based access control. The current NDN-RTC design supports basic content signing and verification. However, a prominent requirement of many types of videoconferencing is confidentiality, which we expect can be supported through encryption-based access control. While it may appear that encryption could limit the gains offered by caching, recent work exploring that application of advanced cryptographic techniques such as attribute-based encryption to multimedia in ICN [13] suggest interesting new directions for balancing benefits of ICN with security requirements.

Inter-consumer sycnhronization. The absence of direct consumer-producer coordination shifted the complexity of "RTC-over-NDN" streaming to the consumer. Another related aspect of teleconferencing which was not covered by this work is to ensure media playback synchronism accross different consumers. Synchronization primitives that were designed in ChronoChat [18] may become a good starting point in addressing this issue.

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, for their input and feedback. John DeHart, Josh Polterock, Jeff Thompson, Zhehao Wang 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`, August 2014.
[6] P. Crowley. Named data networking: Presentation and demo. In *China-America Frontiers of Engineering Symposium*, Frontiers of Engineering, 2013.
[7] J. B. Derek Kulinski. NDNVideo: random-access live and pre-recorded streaming using ndn. Technical report, UCLA, September 2012.
[8] V. Jacobson, D. K. Smetters, N. H. Briggs, M. F. Plass, P. Stewart, J. D. Thornton, and R. L. Braynard. Voccn: voice-over content-centric networks. In *Proceedings of the 2009 workshop on Re-architecting the internet*, pages 1–6. ACM, 2009.

[9] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pages 1–12. ACM, 2009.

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

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

[12] I. Moiseenko and L. Zhang. Consumer-producer api for named data networking. In *Proceedings of the 1st international conference on Information-centric networking*, pages 177–178. ACM, 2014.

[13] J. P. Papanis, S. I. Papapanagiotou, A. S. Mousas, G. V. Lioudakis, D. I. Kaklamani, and I. S. Venieris. On the use of attribute-based encryption for multimedia content protection over information-centric networks. *Transactions on Emerging Telecommunications Technologies*, 25(4):422–435, 2014.

[14] 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.

[15] D. Posch, C. Kreuzberger, B. Rainer, and H. Hellwagner. Client starvation: a shortcoming of client-driven adaptive streaming in named data networking. In *Proceedings of the 1st international conference on Information-centric networking*, pages 183–184. ACM, 2014.

[16] J. Thompson and J. Burke. Ndn common client libraries. Technical Report NDN-0024, Revision 1, NDN, September 2014.

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

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

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

[20] 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.

[21] 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.

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