NDN-RTC Technical Report

P. Gusev, J. Burke

ABSTRACT

NDN-RTC is a fully implemented real-time videoconferencing library based on the WebRTC library and VP8 codec, which has been developed to understand the challenges of low-latency communication over information-centric networks like Named Data Networking (NDN). Through it an NDN-inspired pull-based paradigm has been explored, evaluated and presented in this paper. Also, an initial GUI application for real-time conferencing was built on this library and is now used by members of the NDN project team distributed over sixteen campuses.

1. INTRODUCTION

Named Data Networking (NDN) that is part of the "information-centric networking" field shifts from host-centered towards data-centered communication paradigm. In NDN, every elementary chunk of data has a hierarchical name (often human-readable) which is used by routers to satisfy data requests called "interests". This data is also cached by a router and used to answer similar requests in the future. Initrinsic caching ability of NDN can be leveraged by content distribution applications and halp to offload data publishers significantly in multi-peer scenarios.

Low-latency audio/video conferencing applications often require establishing direct peer-to-peer communication channels for the best user experience and face implementation challenges and inefficiencies related to connection-based approach currently used in most IP conferencing solutions. For example, consider the inefficiency of delivering duplicated media streams for other nine people participating in the same ten-party conference.

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

The rest of the paper is organized as follows. Section 2 covers background and prior work. Section 3 lists main goals of this project. Section 4 describes architecture of the library, designed namespace, data structures and algorithms used. Section 5 discusses implementation details. Section 6 evaluates main outcomes of this project. Section 7 addresses future work and existing issues. Finally, Section ?? concludes the paper.

2. BACKGROUND AND PRIOR WORK

TBD

3. GOALS

TBD

Goals:

- Low-latency (150-300ms) audio/video communication
- Multi-party conferencing
- Passive consumer (no negotiation and/or coordination)
- Content verification

4. ARCHITECTURE

There are two main roles defined in NDN-RTC: producer and consumer. In presence of NDN network the paradigm of real-time communication shifts from the push-based (when producer writes data to the socket and consumer reads it as fast as possible) to the pull-based (producer publishes data on the network with his own pace, while consumer has to request the data he needs and manage incoming data segments). In this case, the producerâĂŹs main task is to acquire video and audio data from media inputs, encode it, pack into network packets and store them in the cache for incoming Interests âĂŞ in this way, complexity shifts to the consumer, and scaling is supported by the network.

Consumer implements more sophisticated algorithms to achieve following goals:

- ensure fetching the latest data available in the network;
- choose appropriate media stream bandwiths provided by a producer by monitoring network conditions;

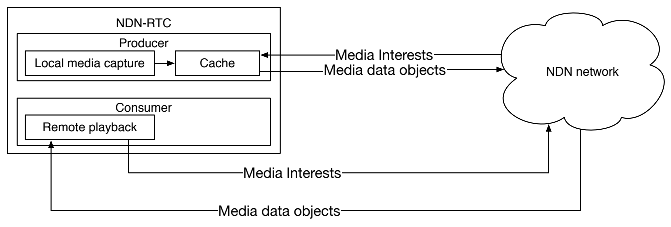


Figure 1: RTC over NDN

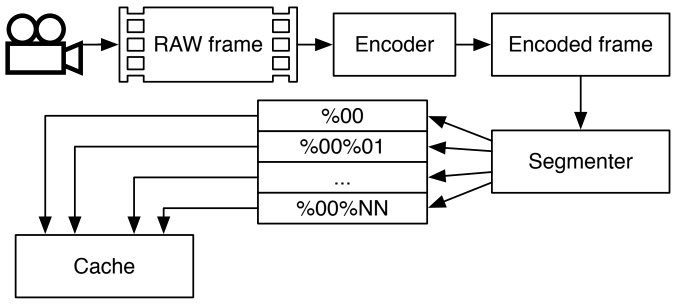


Figure 2: NDN-RTC producer

- playback fetched media in correct order;
- handle network latency and packet drops.

Figure 1 presents top-level overview of how NDN-RTC works. Local media capture and Cache belong to the producer part of the NDN-RTC. Media is stored in the cache which provides access to the data for all incoming interests. Remote playback represents consumer: it issues interests, prepares received media (assembles video frames from segments and re-orders them) and plays it back.

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

Encoded frames vary in size, but average bitrate stays the same. For example, these are the average sizes of frames for 1000 kbps stream:

- **Key frames** $\approx$ 30KB;
- **Delta frames** $\approx$ 3-7KB.

Therefore, depending on the underlying internet protocol used (IP in the existing NDN testbed), producer may need to segment encoded frames into a smaller chunks and provide clear naming conventions for consumer to fetch them (see Figure 2).

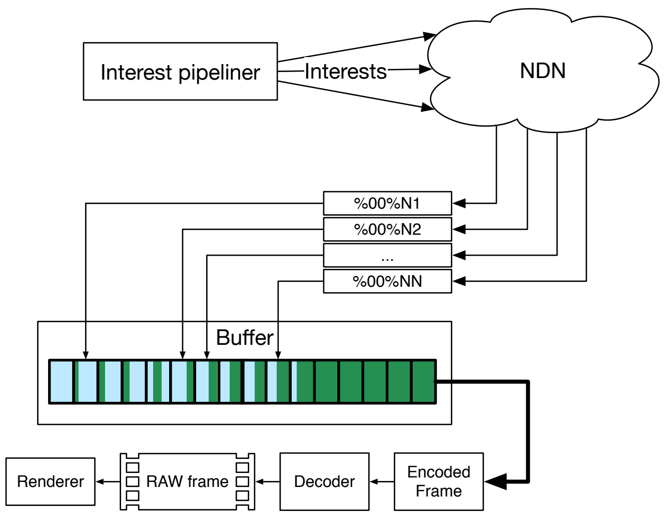


Figure 3: NDN-RTC consumer

Consumer.
Consumer takes into account that media packets are presented by separate segments in the network. Therefore, consumer implements mechanisms of interest pipelining and frame buffering (see Figure 3). Interest pipeliner issues interests for individual segments and is controlled by some higher-level logic which achieves one out of four consumer's goals - ensures that only the latest data is fetched. Packets reordering, drops and latency fluctuations absorbtion are attained by the frame buffer which introduces buffering delay and re-arranges received frames in the playback order.

4.1 Namespace

As there is no direct consumer-produder communication, it is producer's job to provide as much supportive information as possible so that consumer is able to use this information in order to achieve her goals. There are several kinds of data producer can publish and these kinds should be reflected in the namespace:

- Media data
  - Video frames
  - Audio samples
- Forward error correction data
- Producer's streams meta info

Besides that, the namespace should also reflect data specialization hierarchy - from general concepts in the root to more specialized entities towards the leaves.

NDN-RTC producer uses a concept of *media stream* for describing published media. Media stream represents a flow of raw media packets (video frames or audio samples) coming from an input device. Streams usually derive names from their input devices. It is quite natural for a producer to publish several media streams simultaneously, if she has

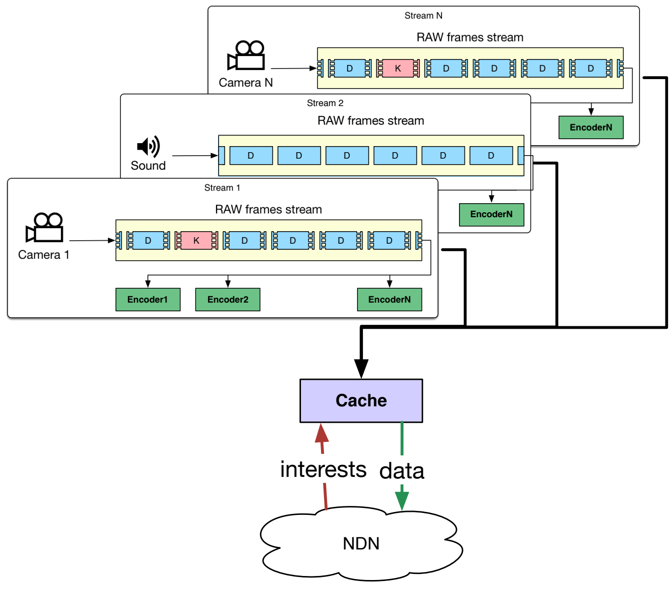


Figure 4: NDN-RTC media streams hierarchy

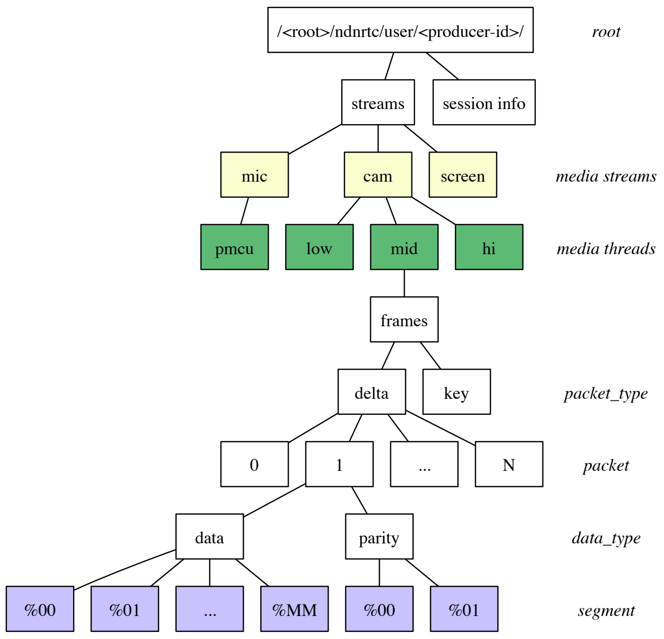


Figure 5: NDN-RTC namespace

more than one device to publish media from (for instance, publish video from camera, audio from microphone and another video stream for sharing computer screen). As all raw data should be encoded, the next level in the name hierarchy represents different encoder instances called *media threads*. Thus, media threads allow producer to provide same media stream in several copies, for instance in low, medium and high quality for video, so that consumer can have a chance to choose media thread more suitable for the current network conditions.

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

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

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

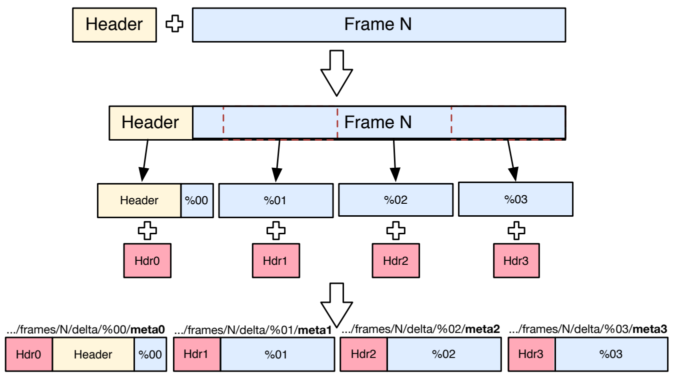
Besides naming data objects, data names can be appended by some additional media-level meta information which can be utlilized by consumers regardless of which frame segment was received first. Three components are added at the end of every data segment name:

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

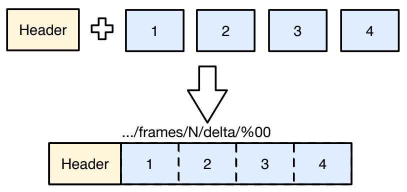
- *playback#* - absolute playback number for current frame; this is different from the *frame#* which is a sequential number for the frame in its' domain (i.e. Key or Delta);
- *paired_seq#* - sequential number of the corresponding frame from other domain (i.e. for delta frames, it is sequential number of the corresponding key frame which is required for decoding);
- *num_parity* - number of parity segments for this particular frame.

4.2 Data objects

Producer generates signed data objects from input media streams and places them in cache instantly. Incoming interests retrieve data from cache, if it is present, or forwarded further to the producer, if the requested data has not been produced yet. In such cases, producer maintains a pending interests table (PIT), which is checked every time new data object is generated. If


Header
Frame N
Header
Frame N
Header
%00
%01
%02
%03
Hdr0
Hdr1
Hdr2
Hdr3
.../frames/N/delta/%00/meta0
.../frames/N/delta/%01/meta1
.../frames/N/delta/%02/meta2
.../frames/N/delta/%03/meta3
Hdr0
Header
%00
Hdr1
%01
Hdr2
%02
Hdr3
%03

(a) video frame segmentation



(b) audio samples bundling

Figure 6: Segmentation and bundling

an interest for newly generated data object exists in the PIT, it gets answered and PIT entry is erased.

Besides actual stream data, data objects contain some amount of meta information which is prepended during frames segmenting (see Figure 6(a)). There are two types of headers: *Frame header* and *Segment header*. Frame header is prepended to segment #0 of each individual video frame and contains media-specific information (such as size of a frame), timestamp, current rate and unix timestamp, which can be used for calculating actual delay between NTP-synchronized producer and consumers (see Figure 7). Segment header is prepended to every segment of a frame. It carries network-level information which can be used by consumer for making certain assumptions about current network conditions and origin of the data objects received:

- Interest nonce

  Nonce value of the interest which requested this particular segment. There are three meaningful cases:

  - *value belongs to the interest issued previously* - consumer received non-cached data requested by interest issued previously;
  - *value is non-zero, but doesn't belong to any of the previously issued interests* - consumer received data requested by some other consumer; data may be cached;
  - *value is zero* - consumer received data which was cached on producer side and never requested by anyone before.

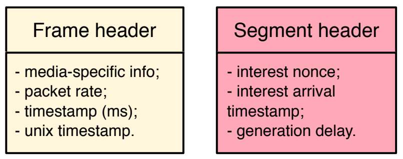


Figure 7: Frame and segment headers

- Interest arrival timestamp

  Timestamp of the interest arrival. Monitoring this value and interest expression timestamps over time may give consumer a clue about how long does it take for interests to reach producer. This value is only valid when nonce value belongs to one of the consumer's interests.

- Generation delay

  Time interval in milliseconds between interest arrival and segment publishing. Consumer can use this value to her advantage in order to control the number of outstanding interests. This value is only valid when nonce value belongs to one of the consumer's interests.

Audio samples are not prepended by any segment header, however the whole audio bundle is prepended by the same frame header (see Figure 6(b)).

4.3 Consumer protocol

4.3.1 Frame fetching

Consumer doesn't know the total number of segments beforehand, unless the very first segment is fetched - in this case, consumer can retrieve metadata from the received segment and get total number of segments for the current frame. Therefore, at first attempt, consumer tries to make a "best guess" in the number of segments she needs to fetch by issuing M interests (see Figure 8). If interests arrive too early, they will be added to the producer's PIT and stay there for some amount of time d_{gen} called `generation delay`. Once encoded frame is segmented into N segments, they are published and if $N \geq M$, interests $0 - M$ are answered with the data which travels back to consumer. Upon receiving first data segment, consumer determines the total number of segments for the current frame and issues $N - M$ more interests for the missing segments if any. These segments will be satisfied by data with no generation delay (as the frame has been published already by producer). The time interval between receiving very first segment and until the frame is fully assembled is represented by d_{asm} and called `assembling time`.

It is needlesss to say, that additional round-trips for requesting missing data segments increase overall frame assembling time and chances that the frame will be incomplete by the time it should be played back. This

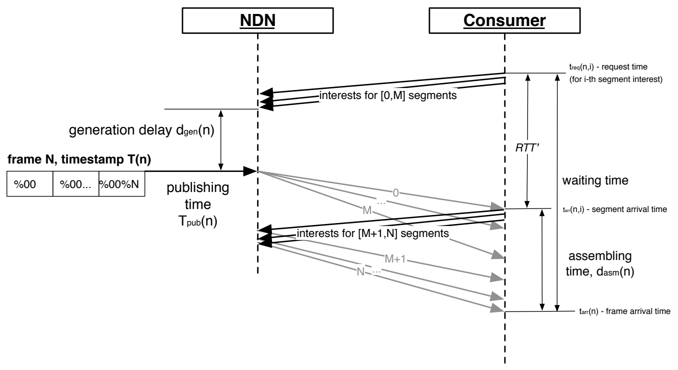

NDN
Consumer
interests for [0,M] segments
generation delay $d_{gen}(n)$
frame N, timestamp T(n)
%00
%00...
%00%N
publishing time $T_{pub}(n)$
RTT'
waiting time
interests for [M+1,N] segments
assembling time, $d_{asm}(n)$

Figure 8: Fetching frame

problem could have been avoided if consumer could make a better guesses in the number of initial interests. Therefore, the following considerations were introduced in later versions of the library:

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

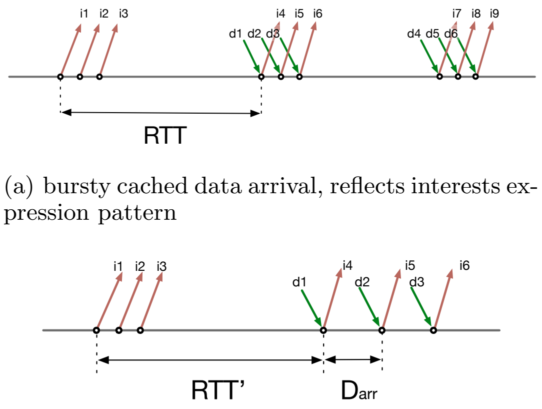
The first consideration was implemented by introducing separate namespaces for key and delta frames. The second consideration helps consumer do better at guessing the number of initial interests.

4.3.2 Bufferization

As in traditional streaming applications, consumer uses frame bufferization in order to tackle out-of-order data arrivals and network delay deviations. Consumer-side jitter buffer is also used as a place for assembling frames by segments. However, the definition of jitter buffer is extended for NDN networks. In traditional push-based approach, buffer can not contain empty frame slots - they are allocated/reserved only when data arrives. Pull-based paradigm requires consumer to request data explicitly. Therefore, after expressing interest consumer "knows" that new data is coming and a frame slot should be reserved in the buffer. Practically, this means, that there will always be some number of empty reserved slots in the buffer. Thus, jitter buffer's size have two measurements:

- *playback size* - playback duration of all complete ordered frames by the moment;
- *estimated size* = *playback size* + *number of reserved slots* × 1/*producer rate*

The difference between estimated size and playback size (RTT') can't be smaller than the current average RTT value. In fact, keeping this value at a minimum indicates that consumer receives the most recent



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

(b) stable fresh data arrival, reflects publishing pattern

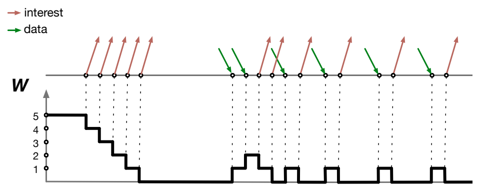
Figure 9: Arrival patterns for the cached and most recent data

data with the minimal amount of outstanding interests. Monitoring this value over time may help consumer to get a clue on her "sync" status with the producer. For example, consumer may use it during fetching process as will be discussed in the next section.

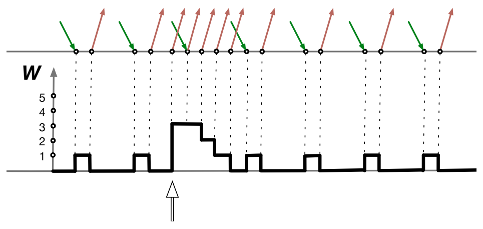
4.3.3 Fetching process

The key challenge in a consumer-driven model for videoconferencing is to *ensure the consumer gets the latest data in a caching network*, without resorting to direct producer-consumer communication that would limit scaling. To get fresh data, which can be cached but should not be the newest available for the consumer's path, the consumer cannot rely only on using such flags as *AnswerOriginKind* and *RightMostChild*. The high frequency nature of streaming data makes no guarantees that the data satisfying those flags received by a consumer will be the most recent one. Instead, it is necessary to use other indicators to ensure that the network supplies up-to-date stream data. The basic solution is to leverage the known segment publishing rate and assume, under normal operation, that a series of old, cached samples, can be retrieved more quickly than new data. The inter-arrival delays (D_{arr}) of the most recent samples follow the publishers' generation pattern but cached data will follow interest expression temporal pattern. Therefore, by monitoring inter-arrival delays of consequtive media samples, consumers can make educated assumptions about data freshness (see Figure 9).

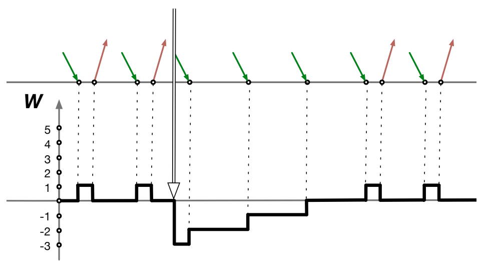
During bootstrapping, the consumer "chases" the producer and aims to exhasut network cache of historical (non-real time) segments. By incerasing the number of outstanding interests, consumer "pulls cached data" out of the network unless the freshest data start to arrive. In order to control interest expresssion, a



(a) W concept

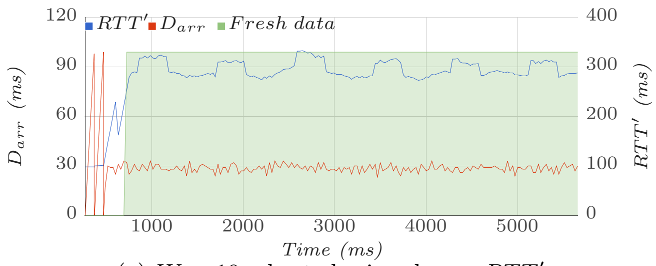


(b) interests bursting ($W + 3$)

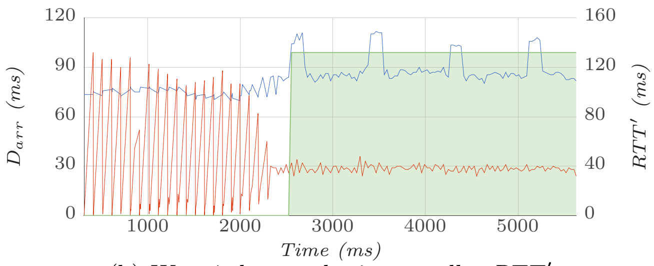


(c) interests withholding ($W - 3$)

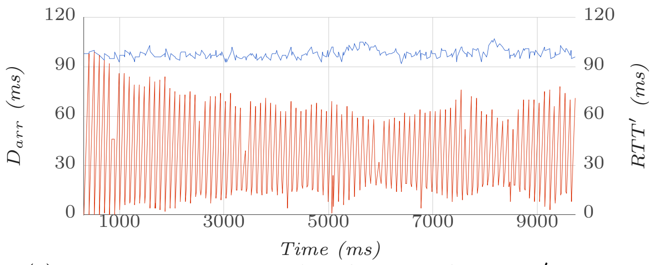
Figure 10: Interest expression control



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



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



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

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

concept of W (roughly an "interest window") is introduced (see Figure 10). Consumer expresses interests only when $W > 0$. At every moment, W indicates how many outstanding interests can be sent out. Before the bootstrapping phase, consumer initializes W with some value which reflects consumer's idea on how many interests are needed in order to exhaust network cache and reach the most recent data (Figure 11 shows how bigger value of W helps to exhaust cache faster). The number of outstanding interests is controlled by a consumer and directly influences how fast consumer can "chase" the producer. Every time a new interest is expressed, W gets decremented and when new data arrives, W is incremented, thus allowing consumer to issue more interests.

W provides a simple mechanism which can be used to speed up or slow down interests expression. Any increase in W value makes consumer to issue more interests (Figure 10(b)), whereas decrease in W holds consumer back from sending any new interests (Figure 10(c)). Larger values of W make consumer faster reach synchronized state with producer. However, larger value means larger number of outstanding interests and larger RTT' because of longer generation delays d_{gen} for each media sample. By adjusting the value of W and observing inter-arrival delays D_{arr} consumer can find minimal RTT' value while still getting non-cached data, thus achieving best synchronization state with producer.

For more complex scenario of video streaming, consumer controls expression of "bulks" of interests instead of individual interests, because video frames are composed of several segments. In this case, W is adjusted on per-frame basis, rather than per-segment. In all other respects, the same above logic can be applied.

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

5. IMPLEMENTATION

TBD

6. EVALUATION

6.1 Role of metadata and historical data

The pull-based nature of NDN and our experimental application's deliberate avoidance of explicit consumer-producer synchronization (allowing publisher-independent scaling) have shown the importance of providing sufficient meta information on producer side. Such information–which could include interests' nonce values, interests' arrival timestamps and data generation delays–if added to the returned data segment may help consumers evaluate relevant network performance, detect congestion and assess whether incoming data is likely to be stale (delayed beyond the path delay). Further, keeping historical data on consumer side may help perform better interest pipelining in the future. For instance, providing the average number of segments per frame type helps consumer make better guesses on the number of required initial interests to fetch upcoming frames, thus keeping frame fetching cycles minimal.

6.2 Application-level PIT

In most cases, consumers aim to express interests for the data not yet produced, so that they may be immediately satisfied when data is produced. The current NDN-CPP library provides a producer-side Memory Content Cache implementation into which data is published. However, it is only useful when data has been published and put in the cache before interest for this data has arrived. For the missing data, the interest is forwarded to producer application which has to store it in internal pending interests table (PIT) unless requested data is ready. This functionality seems quite common for low-latency applications has now been incorporated into the NDN-CPP library implementation.

7. ISSUES AND FUTURE WORK

8. CONCLUSION