NDNLive and NDNTube: Live and Pre-recorded Video Streaming over NDN

Lijing Wang
Tsinghua University
wanglj11@mails.tsinghua.edu.cn

Ilya Moiseenko
UCLA
iliamo@cs.ucla.edu

Lixia Zhang
UCLA
lixia@cs.ucla.edu

1. INTRODUCTION

We have seen great changes of Internet communication pattern in recent years. The Named Data Networking (NDN) was proposed as a new Internet architecture that aims to overcome the weaknesses of the current host-based communication architecture in order to naturally accommodate emerging patterns of communication [1, 2, 3]. By naming data instead of their locations, NDN transforms data into a first class entity, which offers significant promise for content distribution applications, such as video playback application. NDN reduces network traffic by enabling routers to cache data packets. If multiple users request the same video file, the router can forward the same packet to them instead of requiring the video publisher to generate a separate packet. On the contrary, in current TCP/IP implementation, when clients request the same video, the publisher needs to send duplicate packets to transfer the exactly same video.

NDN consumer applications send Interest packets carrying application level names to request information objects, and the network returns the requested Data packets following the path of the Interests. The naming strategy greatly enables the flexibility of application designing. Because applications work with Application Data Units (ADU) — units of information represented in a most suitable form for each given use-case [4]. For example, a multi-user game's ADUs are objects representing current user's status; for an intelligent home application, ADUs represent current sensor readings; and for a video playback application, data is typically handled in the unit of video frames. The naming space just matches the ADU naturally.

However, we found that ADUs are not well considered in traditional video playback application running over TCP/IP. For example, MPEG-DASH technique [5] works by breaking multiplexed or unmultiplexed content into a sequence of small file segments of equal time duration. File segments are later served over HTTP from the origin media servers or intermediate HTTP caching servers. And such segmentation does not preserve boundaries of video frames (ADUs). But in our project every video or audio frame has a unique name, NDN segmentation exposes these boundaries through naming. These frames can be fetched independently according to user's different needs. For example, Consumer application can skip some video frames when packet losses occur in order to keep playing the actual 'live' video.

In this technical report, we will propose two NDN-based video project: NDNLive and NDNTube. They are live and pre-recorded video streaming project over NDN, which follows the ADU designing pattern. The following sections are organized as below. We will introduce Consumer / Producer API [6] and Gstreamer [7], which are the libraries we use for NDN Interests-Data exchanging and media processing in Section 2. The prior work will be compared in Section 3. Then we talk about the architecture and implementation of each project in Sections 5 and 6. Some experimental results will be shown in Section 7. At last, we will conclude our projects in Section 8.

2. BACKGROUND

2.1 Consumer / Producer API

Consumer-Producer API [6] provides a generic programming interface to NDN communication protocols and architectural modules. A consumer context associates an NDN name prefix with various data fetching, transmission, and content verification parameters, and integrates processing of Interest and Data packets on the consumer side. A producer context associates an NDN name prefix with various packet framing, caching, content-based security, and namespace registration parameters, and integrates processing of Interest and Data packets on the producer side.

In NDNlive and NDNtube, the video publisher consists of multiple producers generating video and audio frames separately. The corresponding video players consist of multiple consumers sending Interests for the video and audio frames. Consumer / Producer API simplifies the application logic at both sides: media production and media consumption. Since video frames are too large to be encapsulated by a single Data packet, the media production pipeline has to include a content segmentation step in order to split the content into multiple Data packets. Producer API provides this segmentation functionality. At the same time, since a video frame cannot be retrieved by a single Interest packet, UDR and RDR protocols behind the Consumer API automatically pipeline Interest packets and solve other tasks related to the

retrieval of the application frame. We will talk about the implementation details in Section 6.

2.2 Gstreamer

We use Gstreamer [7] to handle the media processing part.

In NDNLive, raw video images captured by the camera are transferred to the *Encoder_v* component and are encoded into *H264* format. Then the encoded video is passed to the *Parser_v* to be parsed into frames (*B, P or I frame*). The microphone captures the raw audio, which is passed to the *Encoder_a*. The encoder component encodes the raw audio into *AAC* format. The encoded audio stream is transferred to the *Parser_a* to be parsed into audio frames, which are passed to the Producer API for any possible segmentation. Video and audio data is retrieved frame by frame that are passed to the video *Decoder_v* and audio *Decoder_a* for the decoding into the format which the video *Player_v* or audio *Player_a* can play.

In NDNTube, the source for video and audio streams is an mp4 file containing *H264* video and *AAC* audio. Gstreamer opens the file and passes it to the *Demuxer* component to separate video and audio streams. Since the video file is already encoded, the media processing pipeline does not have an *Encoder* component in it. The encoded video or audio streams are separately pushed into the *Parser* to generate video and audio frames.

2.3 Repo-ng

NDNLive streams the captured video and audio non-stop. Therefore, the media publisher just keeps producing new frames and does not care about the data it produced several minutes ago. The consumer is also interested only in recent video and audio frames. As long as the producer is attached to the NDN network, it will serve the incoming Interests.

NDNTube publishes the video only once — all Data packets corresponding to audio and video frames are permanent and never change after the initial publication. Since the same video could be requested multiple times by different users, it is reasonable to store the produced Data packets in a 'database' which is exposed to the requests from the network. Otherwise, every time a different user requests the same video, the corresponding video and audio Data packets would have to be republished (and signed) in case NDN cache has not been able to satisfy these Interests.

Repo-ng [8] is used as a permanent storage for the video and audio content. Repo-ng (repo-new generation) is an implementation of NDN persistent in-network storage, which exposes a Repo protocol [9] allowing write access to applications. Repo insertion is natively supported by the Producer API with *LOCAL_REPO* option (if repo is running on the local host) or *REMOTE_REPO_PREFIX* option to point to the right remote repo by its name prefix.

3. PRIOR WORK

An similar work called NDNVideo was described in this technical report [10]. Their aims are also to provide live and pre-recorded video streaming over NDN. They use Gstreamer to process media and Repo as the permanent storage. Producer and consumer concepts are the same, too.

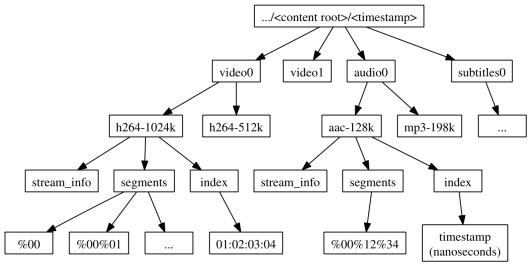

Figure 1: Prior NDNVideo Naming Space

But the way how we handle framing is quite different. In the prior NDNVideo project, the video or audio stream is chopped into fixed-size segments . A mapping between time and segment number is introduced to keep the video and audio synced (Figure 1). Seeking fuction is also supported by the time-segment mapping mechanism.

In our project, the video and audio stream is chopped into frames. One frame may contain several segments. The segmentation process is handled by Consumer / Producer API as we described above in Section 2.1. The application only focuses on the frame level and leaves other task to Consumer / Producer API. We think this application level framing is more like the true NDN way, which we mentioned in Section 2. Every frame has a unique name and is produced and consumed in one time. Only one frame missing won't affect other frames, thus leverages the whole impact to the playing back.

On the contrary, the fixed-size segmentation breaks the integrity of frames (ADU boundary). Only when all the packages are received correctly, the playing back progress can be guaranteed. So we think the prior NDNVideo is more like a TCP/IP way.

The application level framing also provides the flexibility to the video consumer. For example, in NDNLive, if the previous frame can't be retrieved on time or not integrated, the consumer can just skip this bad frame to keep the video streaming. We can see from the evaluation that it won't affect the video fluency. Table 1 shows other differences such as dependencies, Gstreamer version and coding language.

	NDNLive & NDNTube	NDNVideo
Dependencies	ndn-cxx / NFD Consumer / Producer API	CCNx / CCNR pyccn
Gstreamer	1.x	0.1
Framing	video & audio frames	fixed segments
Language	c++	python

Table 1: Comparison with NDNVideo

4. DESIGN GOALS

The aim of developing NDNLive and NDNTube is to rewrite the NDNVideo project by using Consumer / Producer API and therefore can be compacted with ndn-cxx libary [11] and NFD [12]. As a typical use case, these two projects hope to give a careful examination of the design and implementation of Consumer / Producer API. At the same time, NDNLive and NDNTube can satisfy all the design goals of the previous NDNVideo [10]:

- *"Live and pre-recorded video&audio streaming to multiple users"*

  NDNLive is to provide the live video&audio streaming to multiple users and guarantee the fluency of the streaming. NDNTube is to provide a Youtube-like service, which produces the pre-recorded video for multiple users to choose and playback. The quality of video should be guaranteed.

- *"Random access based on actual location in the video"*

  We use frame as the basic operation unit. Most time the relationship between time and frame number can be easily discovered. For example, the video and audio rate are fixed for one given pre-recorded video. Then we can compute the related frame number according to the time information and frame rate. Because we won't store the live stream, so we only support random access for NDNTube.

- *"Ability to synchronize playback of multiple consumers"*

  The synchronization is guaranteed by Gstreamer. Every frame we extracted is in form of *GstBuffer* [13] (Data-passing buffer type of Gstreamer), which contains timestamp information. Then video and audio can be synchronized when playing back according to the timestamp information. Although we don't use Gstreamer, the relationship between time and frame is naturally maintained by video or audio encoded format. The synchronization could be solved relying on these relations.

- *"Passive consumers (no session semantics or negotiation)"*

  Every time the consumer want to play back the video, it can just send Interest requesting the video content as long as it obtains the naming information of producers. There is no session semantics or negotiation between producer and consumer. For example, NDNTube can work well even without frame producer attached to the NDN Network.

- *"Archival access to live streams"*

  The live stream can also be written into the Repo. And Repo will take over the duty of Interest satisfaction. Then the archival access to live stream is possible.

- *"Content verification and provenance"*

  In NDN, every package will be signed by the original producer, the consumer should first verify the data to detect if it belongs to the producer. If it failed, the package will be just regarded. The Consumer / Producer API does some optimization to speed up the signing progress, we will talk about the details in Section 6.1.3.

5. DESIGN

NDNLive and NDNTube are all based on Consumer / Producer API over Named Data Networking. They contains two kinds of roles - producer and consumer. According to the content production and data retrieval pattern, their architecture and namespace will be described below separately.

5.1 Architecture

NDNLive is *Live Streaming*, which the producer captures video from camera and audio from microphone, then passes them to Gstreamer to get raw data encoded and extract the video and audio frames. At last the frames are published to NDN Network by Consumer / Producer API. The consumer can send interest asking for the video stream at any time, it will get the latest video and audio frames, then pass them to Gstreamer to get decoded and at the end the player can play them back (Figure 2).

NDNTube is *Pre-recorded Streaming*, which is more like Youtube. The video source is the pre-recorded video file. As we described in Section2.3, the video and audio frames associated with this video file will be written into Repo in advance. And Repo will take over the duty of responding to the Interests request frames. Then there is no need for the frame producer to attach to the NDN Network. Another difference from NDNLive, in this case the consumer must first know what video files the producer has. So the consumer should send interest asking for the latest playing list and then chose one to play. So the producer only needs to keep publishing the latest playing list containing all the names of video files to the NDN Network (Figure 3).

5.2 Namespace

NDNLive and NDNTube produce video and audio stream separately. Every single frame need a unique name. And before consuming the video and audio content, it should first use the stream information to set up the playing pipeline. There are many components in common between them.

NDNLive Naming.

The following is an example name of NDNLive.

> "/ndn/ucla/ndnlive/publisher-1/video/content /8/%00%00"

- **Routing Prefix:** "/ndn/ucla/ndnlive" is the prefix.

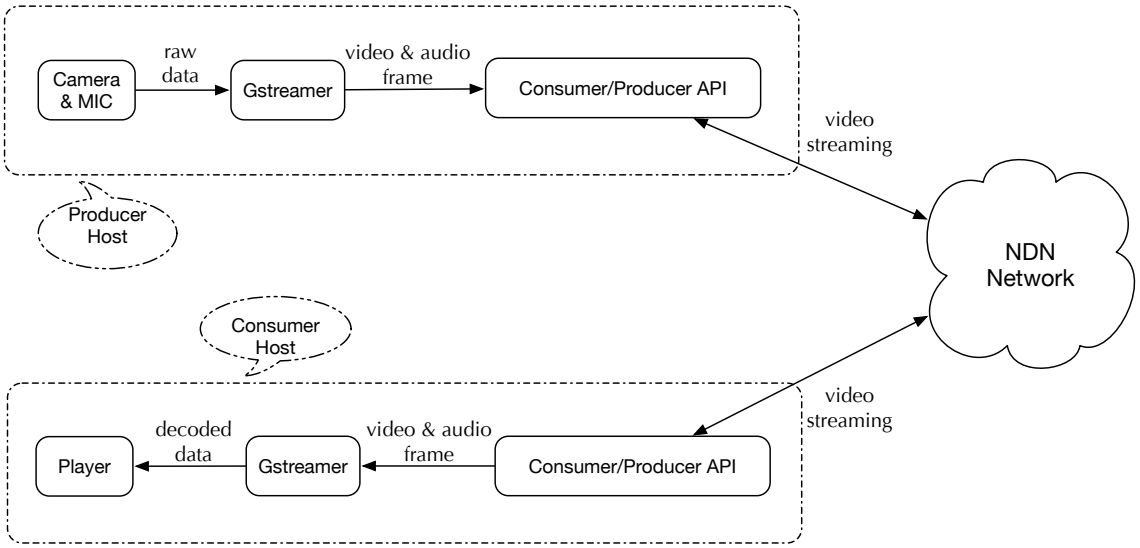

Figure 2: NDNLive Architecture

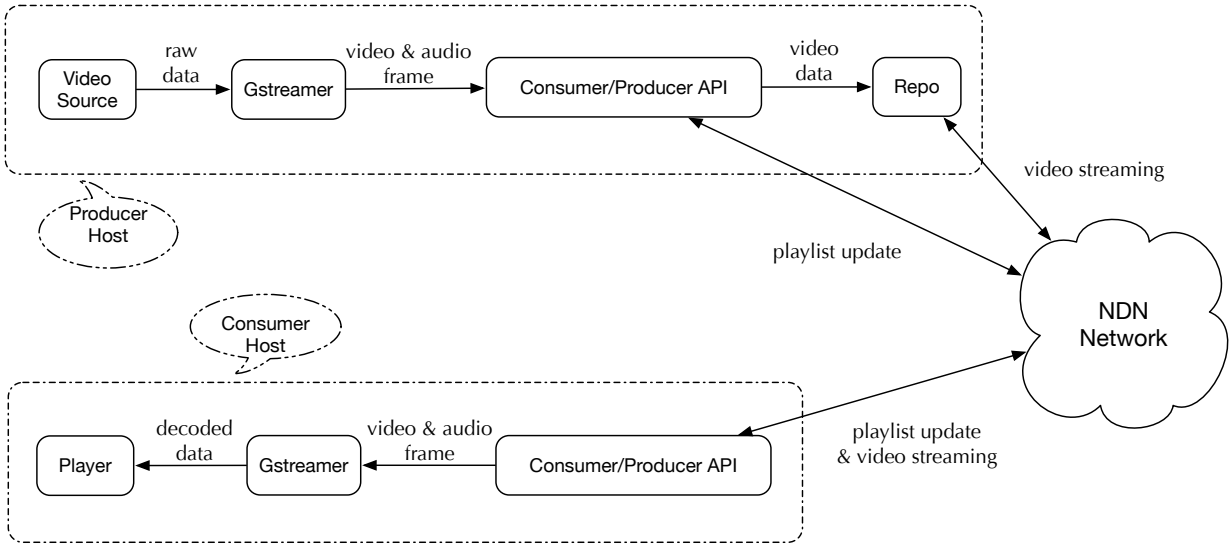

Figure 3: NDNTube Architecture

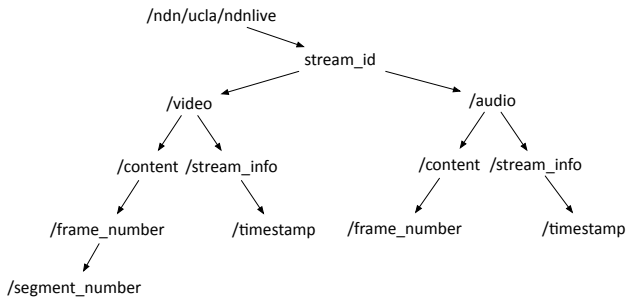

Figure 4: NDNLive Namespace

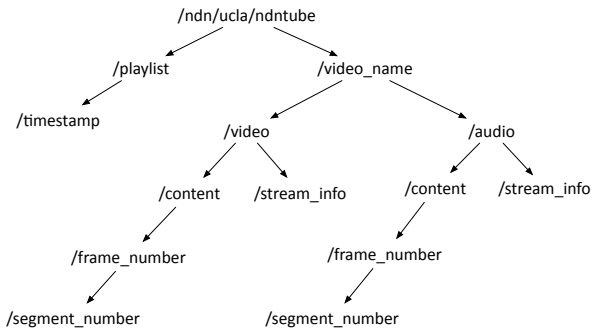

Figure 5: NDNTube Namespace

- **Stream_Id:** "/stream-1" is a representation for one specific live stream, because there could be several producers under the same prefix to publish different live stream.
- **Video or Audio:** "/video" is a mark to distinguish video and audio.
- **Content or Stream_Info:** "/content" represents the frames and "/stream_info" represents the stream information.
- **Frame Number:** "/8" is frame number, which Stream-info does not have this component.
- **Segment Number:** "%00%00" is the segment number. Because most video frames would contain more than one segment, this component is essential. As we mentioned before, the Consumer / Producer API will do the segmentation processing, so the segment number will be appended by the API automatically. But audio frame in NDNLive is always smaller than one segment. There is no segment number for audio frame, and stream_info does not have this component, neither.

Then we can conclude that the above name stands for a piece of data which is the segment 0 inside the 8th video frame of stream-1 under the prefix of /ndn/ucla/ndnlive.

The relative stream information name is shown as below:

> "ndn/ucla/ndnlive/video-1234/video/stream_info /1428725107049"

Because the stream_info would contain the current frame number of video and audio. And the consumer always want to retrieve the latest stream_info to set up the pipeline and also the starting requested frame number. So we append a timestamp component at the end of stream_info to help the consumer retrieve the latest one.

The whole name space of NDNLive should look like Figure 4.

NDNTube Naming.

The namespace of NDNTube is very similar to NDNLive. There are four differences.

1. *Playlist added*

   NDNTube will have a playlist component, which NDNLive does not. The name example is shown as below:

   > "/ndn/ucla/ndntube/playlist/1428725107042"

   Because the playlist can be changed at anytime as long as a new file is added or deleted. The consumer always want to retrieve the latest one. We append a timestamp component at the end to distinguish the obsolete and latest playlist.

2. *Video_Name instead of Stream_Id*

   We should set a component to specific one video file instead of a live stream_id.

3. *No Timestamp under Stream_Info*

   In NDNTube the related stream information of one video file is always the same, so there is no need to add the timestamp component.

4. *Audio also needs Segment_Number*

   The audio frames may also contain more than one segments, because it's not under our control, the quality of MP4 file will influence the size of audio frames.

The whole name space of NDNTube is shown as Figure 5.

6. IMPLEMENTATION

NDNLive and NDNTue are both developed using Consumer / Producer API. This API is an modification version of ndn-cxx library and requires NFD running to forward interests. To compact with Consumer / Producer API and NFD, the project is also written in C++. We use Gstreamer 1.4.3 (other branch not tested) to process media. The supporting platform is UNIX-Like such as Mac OS and Linux. We will explain the implementation details about NDNLive and NDNTube respectively.

6.1 NDNLive

As we describe above (Figure 2), the whole implementation is divided into producer host and consumer host. We

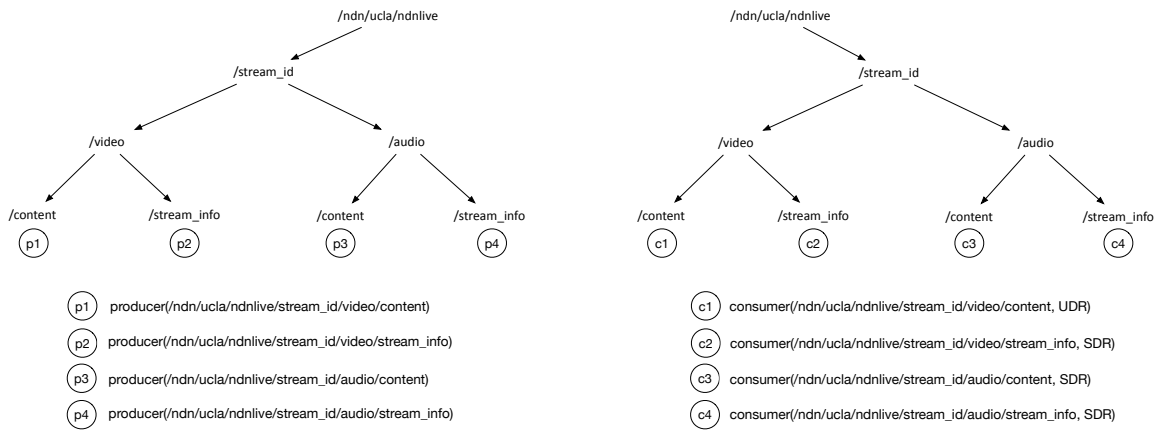

Figure 6: NDNLive Producer and Consumer Structure

will introduce the implementation details of each side, then describe some other vital parts we should pay attention to, such as *Signing and Verification, Synchronization*.

6.1.1 Producer

Four producers are presented in producer host (Figure 6): video content producer, video stream information producer, audio content producer and audio stream information producer.

The content_producer keeps producing frames with frame number increasing incrementally (Figure 4)and publish them to the NDN Network.

The stream_info producer aims to provide the information about the live streaming such as frame rate, width, height, stream format. What's more, to help the consumer to catch up with the latest frame, the current frame number should also be included. To distinguish the obsolete stream_info, timestamp will be appended at the end of name (Figure 4).

Negative Acknowledgement.

There are two situations we should consider carefully.

1. The first one is that, because once the consumer started consuming frames, it will have no idea the about the current frame number which producer is producing. The consumer may sometimes request for a frame number ahead of the producing. It is the producer's duty to inform the consumer about such knowledge. We introduce **NACK**(*Negative Acknowledgment*) to handle such situation.

   For example, in Algorithm 1 , when the interest asks for a piece of data not existed (out of date or not be produced yet), this will trigger the *cache_miss* callback function (*Process_Interest*). In that function, if the data was not produced (*not_ready*), the producer will set up an *APPLICATION_NACK* with *PRODUCER_DELAY* option for this interest together with the estimated delay time.

2. At the same time, although before the consumer starts to consume frames, it will ask for the current number, such information may also go out of date because of the network delay. These out-of-date frames will never be produced again, because the streaming is live. When faced with such situation, the producer will simply send a **NACK** with *NO-DATA* option.

6.1.2 Consumer

Before the consumer asks for the true video data, it must fetch the live stream information to set up the Gstreamer playing pipeline. There are four consumers: video content consumer, video stream information consumer, audio content consumer and audio stream information consumer.

Data Retrieval Protocol.

There are three Data Retrieval Protocols in Consumer / Producer API : **SDR, UDR, RDR**. We will illustrate which protocol we used for each consumer.

1. *Content Retrieval*

   Considering about the live streaming situation, the consumer part needs to keep the video and audio retrieving progress running all the time. The aim is to fetch all the segments inside one frame as soon as possible. The fetching process should NOT be blocked because of one segment missing. So we use **UDR** (*Unreliable Data Retrieval*) for frames retrieval of living streaming. Because the **UDR** will pipeline the Interests sending, and the segments may received out of order, then the consumer part should take care of the segments reassemble and ordering stuff. If one segment of frame is not retrieved on time, then the whole frame will be skipped. But for audio, the size of audio frame is small enough to fill in one segment, so we would use **SDR** (*Simple Data Retrieval*) for audio retrieval.

2. *Stream Information Retrieval*

Algorithm 1 NDNLive producer

$h_v \leftarrow$ **producer**(/ndn/ucla/ndnlive/stream-1/video/ content)
setcontextopt(h_v, **cache_miss**, *ProcessInterest*)
attach(h_v)
while *TRUE* **do**
    $Name\ suffix_v \leftarrow$ video frame number
    $content_v \leftarrow$ video frame captured from Camera
    produce(h_v, $Name\ suffix_v$, $content_v$)
end while

$h_a \leftarrow$ **producer**(/ndn/ucla/ndnlive/stream-1/audio/ content)
setcontextopt(h_a, **cache_miss**, *ProcessInterest*)
attach(h_a)
while *TRUE* **do**
    $Name\ suffix_a \leftarrow$ audio frame number
    $content_a \leftarrow$ audio frame captured from mirophone
    produce(h_a, $Name\ suffix_a$, $content_a$)
end while

function PROCESSINTEREST(Producer **h**, Interest **i**)
    if *NOT Ready* **then**
        $appNack \leftarrow$ **AppNack**(i, **RETRY-AFTER**)
        setdelay($appNack$, $estimated\_time$)
        nack(h, $appNack$)
    end if
    if *Out of Date* **then**
        $appNack \leftarrow$ **AppNack**(i, **NO-DATA**)
        nack(h, $appNack$)
    end if
end function

Algorithm 2 NDNLive consumer

$h_v \leftarrow$ **consumer**(/ndn/ucla/ndnlive//stream-1/video/ content, *UDR*)
setcontextopt(h_v, **new_segment**, *ReassambleVideo*)
while *reach Consume_Interval_Video* **do**
    $Name\ suffix_v \leftarrow$ video frame number
    consume(h_v, $Name\ suffix_v$)
    $framenumber++$
end while

function REASSAMBLEVIDEO(Data **segment**)
    $content \leftarrow$ reassamble **segment**
    if *Final_Segment* **then**
        $video \leftarrow$ decode **content**
        Play $video$
    end if
end function

$h_a \leftarrow$ **consumer**(/ndn/ucla/ndnlive/stream-1/audio/ content, *SDR*)
setcontextopt(h_a, **new_content**, *ProcessAudio*)
while *reach Consume_Interval_Audio* **do**
    $Name\ suffix_a \leftarrow$ audio frame number
    consume(h_a, $Name\ suffix_a$)
    $framenumber++$
end while

function REASSAMBLEAUDIO(Data **content**)
    $audio \leftarrow$ decode **content**
    Play $audio$
end function

Because the stream information contains only one segment and will be fetched only one time (at the beginning of the playing back). We use **SDR** (*Simple Data Retrieval*) to fetch the stream info for video and audio. Except for the basic stream information, the consumer also needs to obtain the current frame number the producer just produced. So that the frame consumer can start from this frame number and increase it one by one. To retrieve the latest stream information, *Right_Most_Child* option should be set as TRUE (Algorithm 2).

Consume Interval.

In consumer part, we should control the Interest sending speed. If we send them too aggressively, the data in producer side may not get ready. If we send them too slowly, the playing back may not match the video generating speed. Our solution is to send Interests according to the video and audio frame rate. For example, the video frame rate is 30 frame/second, then the *consume_interval* should be $1000/30 \approx$ 33.3 millisecond. The consume function should be called every *consume_interval*. The boost scheduler will schedule the consume process every video or audio interval according to the video or audio *consume_interval*.

6.1.3 Some other vital parts

Signing and Verification.

Every NDN package should be signed with the producer's private key, only the verified frame can be retrieved successfully. But signing and verification are very time consuming. Consumer / Producer uses *Manifest* [6] to improve the signing and verification performance.

Instead of signing every segment in one frame, the producer only needs signing and verifying the Manifest. This option can be easily turned on or off by set *EMBEDED_ MANIFEST* as TRUE or FALSE.

Synchronization between video and audio.

Since we process video and audio separately, it is a vital problem to keep them synced. Gstreamer can handle the synchronization for us in this way:

When video and audio are captured, they are timestamped

by the Gstreamer. The time information will be recorded in *GstBuffer* data structure which Gstreamer used to contain media data. This time information will also be transferred along with video or audio frame. Then when the consumer fetches the video or audio frame separately, the video and audio frames will be pushed into the same *GstQueue*. Gstreamer will extract the timestamps hiding in the video and audio frames, then play them back together according to the timestamps.

6.2 NDNTube

Although NDNTube is very similar to NDNLive, data production and retrieval pattern are quite different from NDNLive. We will describe them in detail in this section.

Algorithm 3 NDNTube producer

$h_v \leftarrow$ **producer**(/ndn/ucla/ndntube/video-1234/video)
setcontextopt(h_v, **local_repo**, *TRUE*)
while *NOT Final Frame* **do**
  $Name\ suffix_v \leftarrow$ video frame number
  $content_v \leftarrow$ video frame
  produce(h_v, $Name\ suffix_v$, $content_v$)
end while

$h_a \leftarrow$ **producer**(/ndn/ucla/ndntube/video-1234/audio)
setcontextopt(h_a, **repo_prefix**, */ndn/ucla/repo*)
while *NOT EOF* **do**
  $Name\ suffix_a \leftarrow$ audio frame number
  $content_a \leftarrow$ audio frame
  produce(h_a, $Name\ suffix_a$, $content_a$)
end while

6.2.1 Producer

There are three producers: playlist producer, video producer and audio producer. Playlist producer is responsible for generating the latest playlist every time it detected video file added or deleted.

Different from NDNLive, we combine content and stream_info producer into one video or audio producer. Because for NDNLive, producers need to respond to the Interests coming from consumer directly. And content producer and stream_info producer have different callbacks when the Interest enters the contexts or a cache_miss is triggered, so we should separate them. But for NDNTube, all the stream_info and content will be inserted into repo, and repo will take care of the response to consumers. There is no need to separate them. And once the producer finished producing the content and stream information, it can be offline, doesn't need to attach to the NDN Network.

The producer side's Pseudocode is shown as Algorithm 3.

6.2.2 Consumer

Algorithm 4 NDNTube consumer

$h_v \leftarrow$ **consumer**(/ndn/ucla/ndntube/video-1234/video, *RDR*)
setcontextopt(h_v, **new_content**, *ProcessVideo*)
while *NOT EOS* **do**
  $Name\ suffix_v \leftarrow$ video frame number
  consume(h_v, $Name\ suffix_v$)
  $framenumber++$
end while

function PROCESSVIDEO(byte[] **content**)
  $video \leftarrow$ decode **content**
  Play $video$
end function

$h_a \leftarrow$ **consumer**(ndn/ucla/ndntube/video-1234/audio, *RDR*)
setcontextopt(h_a, **new_content**, *ProcessAudio*)
while *NOT Final Frame* **do**
  $Name\ suffix_a \leftarrow$ audio frame number
  consume(h_a, $Name\ suffix_a$)
  $framenumber++$
end while

function PROCESSAUDIO(byte[] **content**)
  $audio \leftarrow$ decode **content**
  Play $audio$
end function

There are five consumers: playlist consumer, video content consumer, video stream information consumer, audio content consumer and audio stream info consumer.

Data Retrieval Protocol.

Same with *Streaminfo* tetrieval of NDNLive, we use **SDR** to retrieve stream information and the playing list. We want to fetch the latest version of playing list, so we should set *Right_Most_Child* option as TRUE as well.

However, for the content retrieval, we should use **RDR** (*Reliable Data Retrieval*). Because we can't stand any segment missing for the pre-recorded video, and we always want the good quality of video and audio. If the video segments are not received on time, the Interest requesting for that segment will be retransmitted. This retransmission is done by Consumer / Producer API.

If all the retransmission failed to get the data. The consumer will resend the Interest for that frame. Such retransmission is application level. It won't send the Interest asking for the next frame until it gets the requested frame or several times application level retransmission. At the same time, when lacking of frames the *Buffering* mechanism will be triggered. Only when Gstreamer accumulates enough video and audio frames (such as two seconds duration), it will continue to play back. Otherwise, it will be just paused until the

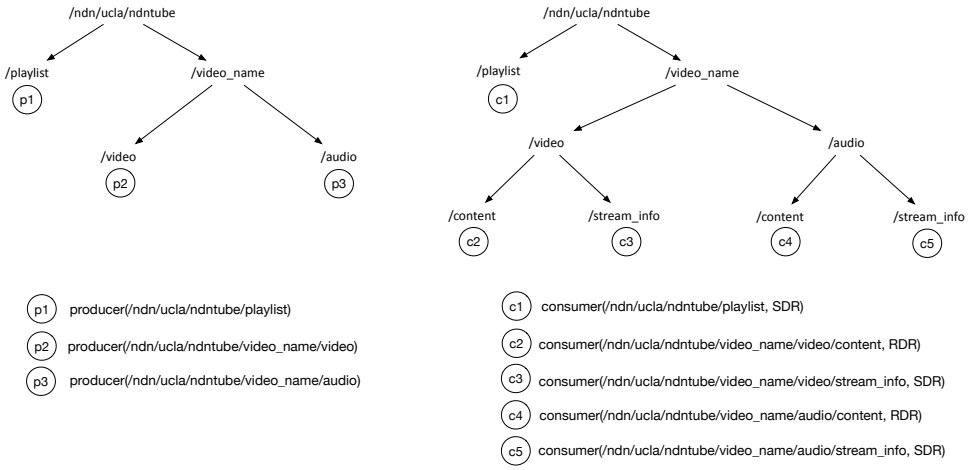

Figure 7: NDNTube Producer and Consumer Structure

buffer is full.

Other issues.

There also exists the synchronization problem between video and audio. As we describe above 6.1.3, the Gstreamer will handle the synchronization part as long as we give the video and audio frame correct timestamps. In NDNLive, it is the capturing component who stamps the frames. In NDNTube, it is the *Dumxer* who is responsible for time stamping. Once the media data flows through *Dumxer*, this component will separate the video stream and audio stream according to their file type such as *MP4* and adding the time information in each *GstBuffer*.

Because all the content and stream information are all already existed and written into Repo. Then Repo takes over the responsibility. There are not *NACK* in producer part. Also due to this reason, the consumer side should retrieve the data as soon as possible to keep the quality and fluency of video playing back. The default *Consume_Interval* is 0 in NDNTube.

The consumer side's Pseudocode is shown as Algorithm 4.

7. EVALUATION

When some frames missing, the performance does not be affected too much. The audio will not be affected at all. For video, when the missing video frame is the key frame, it will appear one second mosaic. But if it was other type frame, the picture is fluent enough.

[I really don't know what to talk about this part... All evaluation result seems not pretty good...] – How to measure the quantification?

8. CONCLUSION

9. REFERENCES

[1] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking Named Content," in *Proc. of CoNEXT*, 2009.

[2] L. Zhang et al., "Named Data Networking (NDN) Project," Tech. Rep. NDN-0001, October 2010.

[3] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, K. Claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, "Named Data Networking," *ACM SIGCOMM Computer Communication Review*, July 2014.

[4] D. D. Clark and D. L. Tennenhouse, "Architectural Considerations for a New Generation of Protocols," *SIGCOMM Comput. Commun. Rev.*, vol. 20, no. 4, pp. 200–208, Aug. 1990.

[5] T. Stockhammer, "Dynamic adaptive streaming over HTTP: standards and design principles," in *Proceedings of the second annual ACM conference on Multimedia systems*. ACM, 2011, pp. 133–144.

[6] L. Z. Ilya Moiseenko, "Consumer-producer api consumer-producer api for named data networking," Technical Report NDN-0017, NDN, Tech. Rep. NDN-0017, August 2014.

[7] [Online]. Available: http://gstreamer.freedesktop.org/

[8] S. Chen, W. Shi, J. Cao, A. Afanasyev, and L. Zhang, "NDN Repo: An NDN Persistent Storage Model," 2014. [Online]. Available: http://learn.tsinghua.edu.cn:8080/2011011088/WeiqiShi/content/NDNRepo.pdf

[9] [Online]. Available: http://redmine.named-data.net/projects/repo-ng/wiki/Repo_Protocol_Specification

[10] D. Kulinski and J. Burke, "NDN Video: Live and Prerecorded Streaming over NDN," UCLA, Tech. Rep., 2012.

[11] [Online]. Available: http://named-data.net/doc/ndn-cxx/current/

[12] A. Afanasyev, J. Shi, B. Zhang, L. Zhang, I. Moiseenko, Y. Yu, W. Shang, Y. Huang, J. P. Abraham, S. DiBenedetto *et al.*, "NFD developer's

guide," Technical Report NDN-0021, NDN, Tech. Rep., 2014.
[13] [Online]. Available: http://gstreamer.freedesktop.org/data/doc/gstreamer/head/gstreamer/html/GstBuffer.html