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.

What's more, in NDN consumers 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 both video project, the video publisher behaves as the producer and generates video and audio frames separately. The users behave as the consumer sending Interests asking for video to play back. The Consumer / Producer API simplified the work of data production and consumption in both side. For example, we find that some video frames are too large to be encapsulated by a single Data packet, and the producer side of the application would have to perform content segmentation in order to split the content into multiple Data packets. The Producer API will do the segmentation inside one video frame automatically. At the same time, a video frame cannot be retrieved by a single Interest packet, and the Consumer API will pipeline Interest packets and solve other tasks related to the retrieval of the application

frame as long as we set up the right Data Retrieval Protocol*(SDR/UDR/RDR)*. In the case of MPEG-DASH, all these low-level details are handled by the HTTP / TCP protocol machinery. We will talk about the implementation details in Section 6.

2.2 Gstreamer

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

For NDNLive, the raw video pictures captured by camera would be transferred to *Encoder_v* component and will be encoded into *H264* format. Then the encoded video is transmitted into *Parser_v* to get parsed into frames (*B, P or I frame*). For audio, the microphone will capture the audio then push the raw audio into *Encoder_a*. The encoder component will encode the raw audio into *AAC* format. The encoded audio stream will be transferred to *Parser_a* to get parsed then passed to Consumer / Producer API to be produced.

The main difference of NDNTube from NDNLive is the video source, which is the video file. Gstreamer should firstly read video file then pass it to the *Demuxer* component to separate video and audio stream. Because the video file is already encoded, so there is no *Encoder* component here. The separate encoded video or audio is pushed into the *Parser* to generate frames.

Because we need to extract the frames from the video source, so now we only support *H264* video encoded format and *AAC* audio encoded format for NDNLive and *MP4* file format for NDNTube.

2.3 Repo-ng

For NDNLive, the captured video and audio is always streaming and the producer just keeps producing the latest frames and doesn't care about the data it produced several minutes ago. The consumer will also only request for the current frame which is just produced. So as long as the producer is attached to the NDN network and then can respond to the Interests. The consumer can get the data back and play them back immediately.

But for NDNTube, once the video file is uploaded to the producer part, it is permanent. And the same video file could be requested several times. So the video file should be produced just once then be stored somewhere else and exposed to the NDN Network and waiting for retrieved. Otherwise, every time different users request for the same piece of data will cause the regeneration and production in the producer side if the cache doesn't contain such piece of data.

Repo-ng [8] is introduced to provide the permanent storage for the video data. Repo-ng (repo-new generation) is an implementation of NDN persistent in-network storage conforming to Repo protocol [9]. In the last report, we won't distinguish Repo-ng and Repo. Repo insertion is natively supported by the Consumer / Producer API. You can just set the *LOCAL_REPO* option as TRUE, all the data producer generated will be inserted into local repo, or set the option *REMOTE_REPO* with appropriate *Repo_Prefix*, the data will be inserted into the remote repo which matches the prefix.

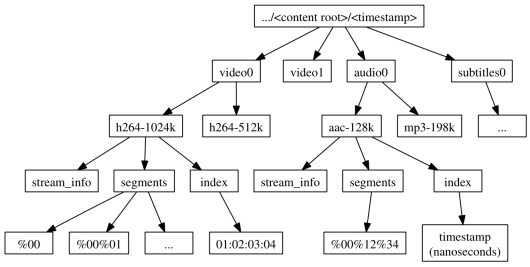

Figure 1: Prior NDNVideo Naming Space

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 concept are also the same, too.

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

In our project, the video and audio is chopped into frames. One frame may contain several segments. The segmentation process is handled by Consumer / Producer API. The application only focuses on the frame level and leave 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 leverage 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 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 influence the video fluency. Table 1 shows other differences such as dependencies, Gstreamer version and coding language.

4. DESIGN GOALS

The aim of developing NDNLive and NDNTube is to rewrite

	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

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 no matter the NDNLive or NDNTube, it can just send interest requesting the video content as long as it obtain the naming information of producer. 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. Let Repo take care of the 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 part should first verify if it belongs to the original 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 detail in Section 6.

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 generating and data retrieval pattern, their architecture and namespace are described separately below.

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, 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.

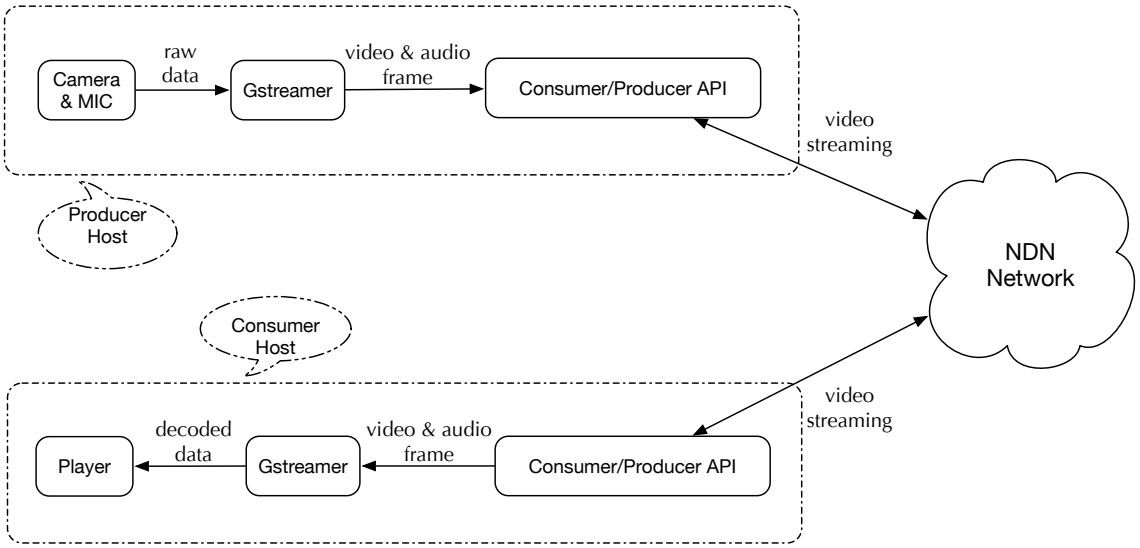

Figure 2: NDNLive Architecture

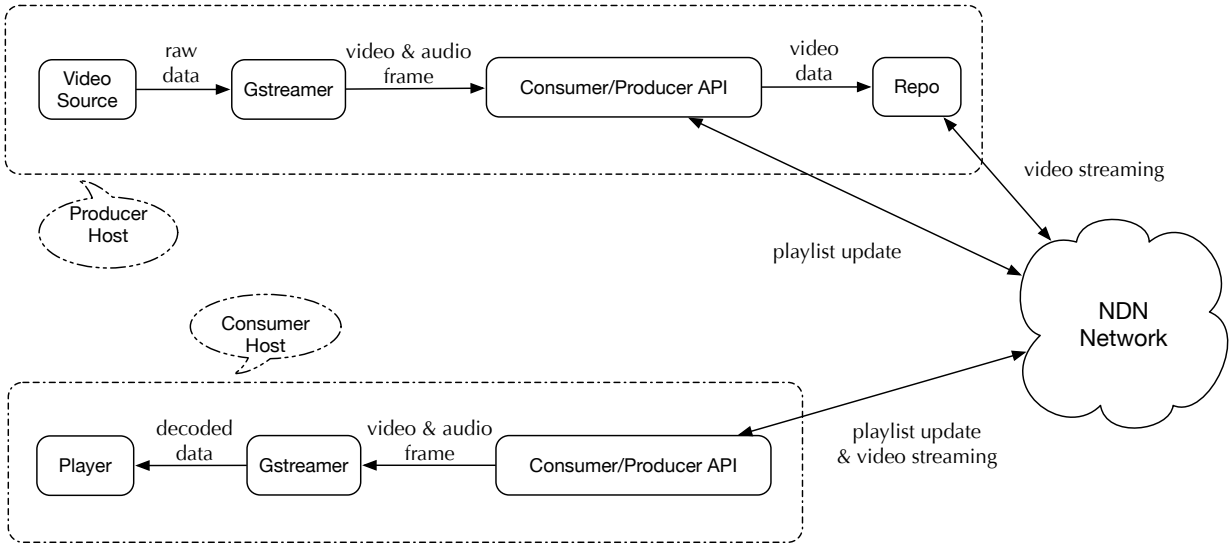

Figure 3: NDNTube Architecture

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.
- **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"

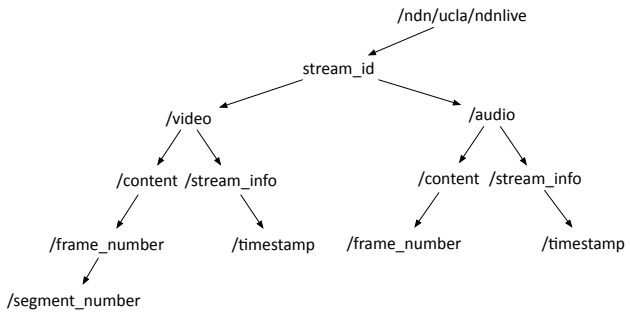

Figure 4: NDNLive Namespace

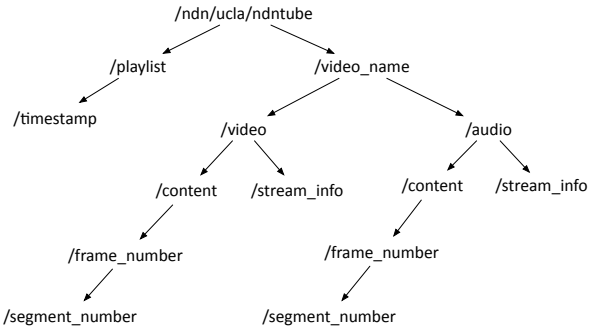

Figure 5: NDNTube Namespace

   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

Next-NDNVideo is developed using Consumer / Producer API over NDN. 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

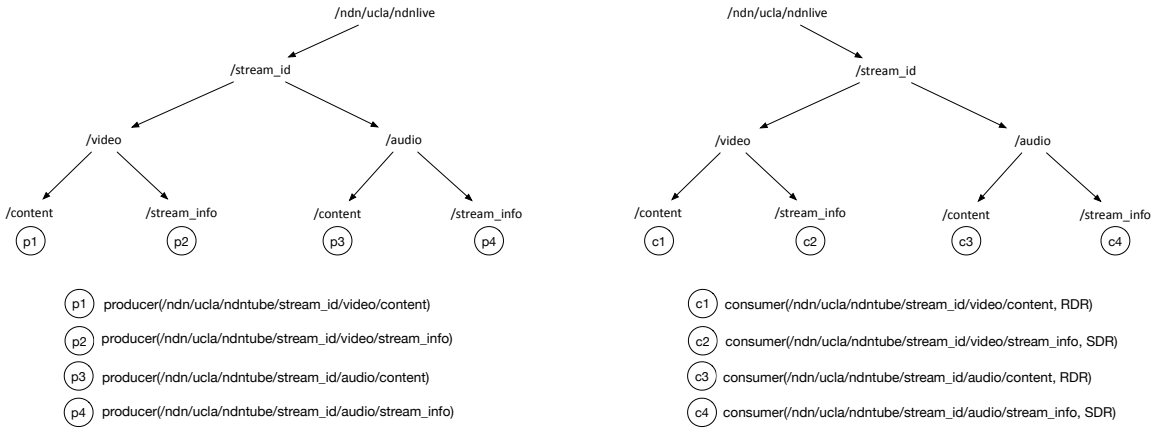

Figure 6: NDNLive Producer and Consumer Structure

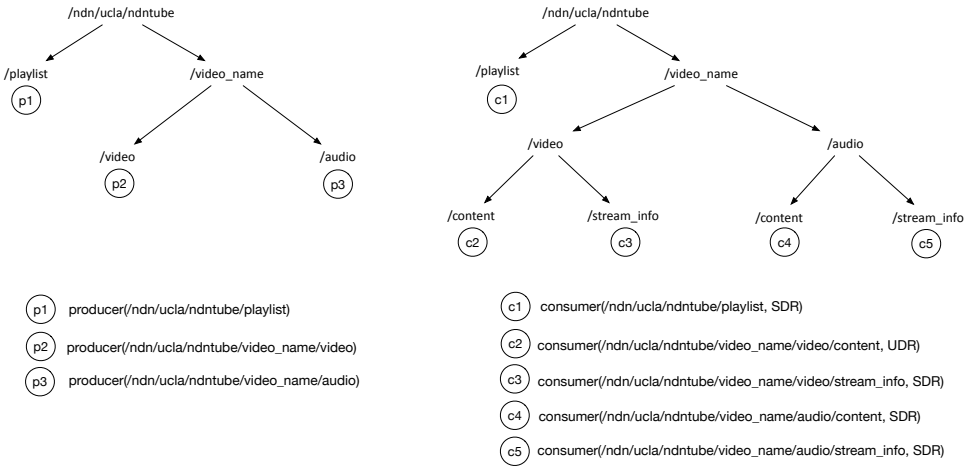

Figure 7: NDNTube Producer and Consumer Structure

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 Live Streaming and Prerecorded respectively.

6.1 Live Streaming

The architecture of Live Streaming is shown as Figure 2. It has four producers: video stream information producer, video content producer, audio stream information producer and audio content producer. Before the consumer asks for the true video data, it must fetch the live stream information to set up the Gstreamer playing pipeline. The stream information includes frame rate, width, height, stream format and so on. For Live streaming, to inform consumer of the starting frame number, the producer also need to produce the latest frame number if requested. The video and audio are produced and consumed separately, so they must be in separate thread. In both (producer and consumer) side, the Gstreamer should keep running in the background, so Gstreamer needs another thread.

[To conclude, in each side we have three threads running at the same time. Components inside one black dotted rectangle belong to the same thread.] ——- have problems! now the consumer has video and audio running in the same thread, and boost scheduler will schdule the consume process every video or audio interval according to the video or audio frame rate.

The following part will describe some details about media processing, the data retrieval protocols we use and the pseudo code.

6.1.1 Media Processing

Producer.

The producer will capture video from camera and audio from microphone, then produce them frame by frame.

The raw video pictures captured by camera would be transferred to *Encoder_v* component and will be encoded into *H264* format. Then the encoded video is transmitted into *Parser_v* to be parsed into frames (*B, P or I frame*). The last component is the *Producer_v*, which is in charge of producing video frame by frame.

For audio, the microphone will capture the audio then push the raw audio into *Encoder_a*. The encoder component will encode the raw audio into *AAC* format. The encoded audio stream will be transferred to *Parser_a* to get parsed then passed to *Producer_a* component. Finally, this audio producer will produce the audio frame by frame.

Consumer.

The consumer also processes video and audio separately.

For video, the *Consumer_v* component keeps sending interests to fetch the video data. And it will get data retrieved frame by frame. Then the video frame will be thrown to the *Decoder_v* to get decoded into the format which the *Player_v* can play it back. For audio, the *Consumer_a* component fetches the audio data by sending audio interests. The retrieved audio frame will be transferred to *Decoder_a* to get decoded. The decoded audio will be sent to *Player_a*. At the end, the *Player_v* and *Player_a* should play video and audio together.

Except for the last component, the others are all introduced above. The last component of Stream_info is the information type.

- **Info Type:**
  - **pipeline:** means that it asks for essential information to set up the playing back pipeline.
  - **current_id:** means that it asks for the current frame number of this video. This is used only by NDNLive.
  - **final_id:** means that it asks for the final frame number of this video. This is used only by NDNTube.

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.

6.1.2 Data Retrieval Protocol

Streaminfo Retrieval.

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.

Frames Retrieval.

Considering about the situation of live streaming, 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. Then the consumer part should take care of the segments reassemble and ordering stuff.

6.1.3 Pseudocode

Algorithm 1 Live video producer

$h_v \leftarrow$ **producer**(/ndn/ucla/livevideo/video/)
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/livevideo/audio/)
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

There are two situations we should consider carefully. 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. It 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 existing (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.

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.

Algorithm 2 Live video consumer

$h_v \leftarrow$ **consumer**(/ndn/ucla/livevideo/video/, *UDR*)
setcontextopt(h_v, **new_segment**, *ReassambleVideo*)
while *TRUE* **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/livevideo/audio/, *UDR*)
setcontextopt(h_a, **new_segment**, *ReassambleAudio*)
while *NOT EOS* **do**
    $Name\ suffix_a \leftarrow$ audio frame number
    consume(h_a, $Name\ suffix_a$)
    $framenumber++$
end while

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

The Pre-recorded Streaming architecture is slightly different from Live Streaming . Firstly, **Repo-ng** is added in the producer side to provide the constant storage for video and audio data. Secondly, the playing list producer and consumer are introduced. This producer is to produce the playing list, which records the video names in one specific directory of producer. Because the video files could be added or deleted anytime, every time the producer detected the change of the directory, it will produce a new playing list with the same name except for appending a new timestamp.

The other two producers are almost the same with Live Streaming One producer produces the stream information of the recorded video. The other takes charge of producing the frames, one slight difference is that we produce audio frames after video frames. This is to avoid the writing congestion of repo. The difference is that the stream info and frame data will be inserted into **Repo-ng**, but not attached to the NDN Network. Because different from live streaming, the pre-

recorded video is permanent. Once the data was produced and the same data could be requested many times. However, for the live streaming, it just keeps producing and doesn't care about the data it produced several minutes ago. So the video file should be only produced once then written into Repo-ng and should not be produced again. The Repo-ng will respond to the interests. This can also relieve the producing pressure of producer.

The consumer part should send interests to request the playing list in advance. Only after the consumer has the knowledge of all the files which producer has, it can ask for one specific video to play back. Same as live streaming, the pre-recorded video consumer also needs to retrieve the *Streaminfo* of the video file to set up the playing pipeline of Gstreamer. Then the consumer keeps sending video and audio content retrieval interest with the frame number increased one by one. One difference from live streaming is that, the pre-recorded consumer must know the ending frame of the video file. So except for the basic information of pipeline, it should contain the final video and audio frame id. Then the consumer can know when to stop sending interests. On the contrary, the live streaming consumer just keep increasing the frame number until user close it. Because the video is alive and will be producing all the time. But the live streaming needs to obtain the current frame number to follow the step with the producer. The pre-recorded video consumer does not have this part.

[Another problem] —- Should I mention the blocking style which we mimic youtube?

6.1.4 Media Processing

The main difference from live streaming is the video source. The pre-recorded video source is the video file. Gstreamer should first read video file then pass it to the *Demuxer* component to separate video and audio stream. Because the video file is already encoded, so there is no *Encoder* component here. The separate encoded video or audio is pushed into the *Parser* to generate frames. The frames will be produced by Consumer / Producer API and inserted into *Repo-ng*. The API will handle the segmentation automatically. The media processing of consumer part is the same with live streaming.

There also exists the synchronization problem between video and audio. As we describe above 6.1.1, the Gstreamer will handle the synchronization part as long as we give the video and audio frame correct timestamps. In live streaming, it is the capturing component who stamps the frames. In pre-recorded streaming, 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*.

6.1.5 Data Retrieval

Same with *Streaminfo* tetrieval of live streaming, 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 frames retrieval, we should use **RDR** (*Reliable Data Retrieval*). Because we can't stand any segment missing for the recorded video, we always want the good quality of video and audio. If the video segments are not received on time 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 buffer is full.

6.1.6 Pseudocode

[Say something here...]

Algorithm 3 Pre-recorded video publisher

$h_v \leftarrow$ **producer**(/ndn/ucla/recordvideo/video-1234/video/)
setcontextopt(h_v, **repo_prefix**, */ndn/ucla/repo*)
while *NOT EOF* **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/recordvideo/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

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 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.

Algorithm 4 Pre-recorded video consumer

$h_v \leftarrow$ **consumer**(/ndn/ucla/recordvideo/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/recordvideo/video-1234/audio/, *RDR*)
setcontextopt(h_a, **new_content**, *ProcessAudio*)

while *NOT EOS* **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

[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