NDN, Technical Report NDN-0007, 2012. http://named-data.net/techreports.html
Revision 1: September 2012

NDNVideo: Random-access Live and Pre-recorded Streaming using NDN

Derek Kulinski
UCLA Computer Science Department
E-mail: kulinski@cs.ucla.edu

Jeff Burke
UCLA REMAP
E-mail: jburke@remap.ucla.edu

I. Introduction

The experimental future Internet architecture, Named Data Networking (NDN) [1], offers significant promise for content distribution applications. NDN treats data, instead of hosts, as the first-class entity on the network. Instead of addressing nodes on the network with IP addresses (numbers), NDN enables applications to name data (often in a human-readable way) and routes based on those names. Among other benefits, this approach, combined with per-packet content signatures, can reduce network traffic by enabling routers to cache data packets. If two consumers request the same data, the router can forward the same packet to both of them instead of requiring the data publisher to generate a separate packet.

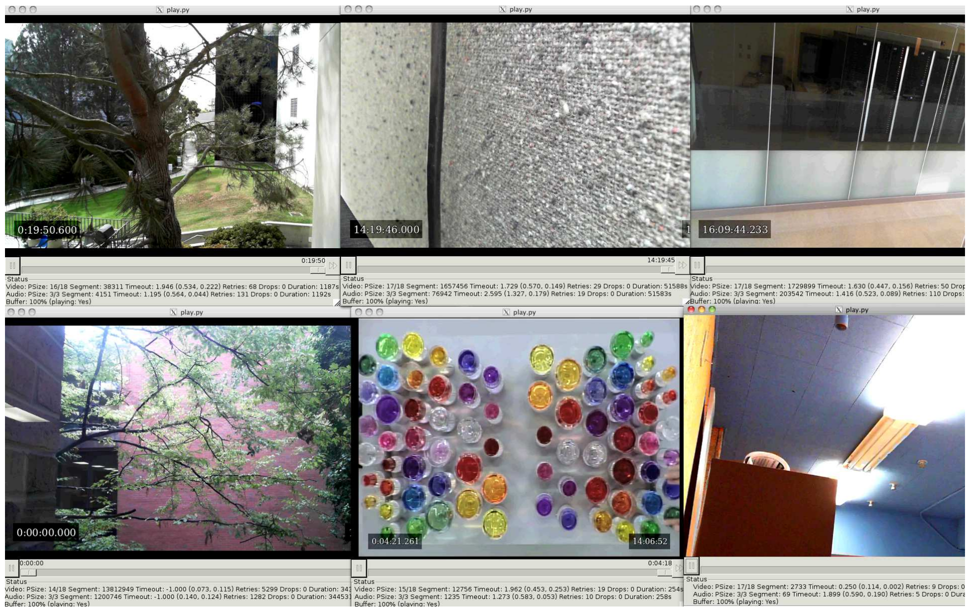
Video distribution services, such as YouTube, could benefit significantly from NDN. The site provides multiple videos, but using the current IP Internet, when multiple computers request the same video, YouTube needs to send duplicate packets to transmit the same video to each. (Even modern content distribution networks do not fully mitigate this requirement.) NDN would enable the network itself to cache frequently requested data, reducing load on YouTube servers and increasing performance for users. Interestingly, it could provide the same support to low-capability sources of video, such as mobile phones, enabling direct publishing to many consumers from even these devices. Additionally, by publishing video data using semantically meaningful names (e.g., timecode frame indexes), random access and other related features can be easily implemented.

To develop NDN's potential for scalable random-access video further, this project designed and implemented a complete software solution for video and audio streaming over NDN, called NDNVideo. NDNVideo leverages the network's features to provide highly scalable, random-access video from live and pre-recorded sources with a straightforward consumer and publisher implementation. It uses the GStreamer open-source media framework and PyCCN, Python bindings for the CCNx software router.

II. Background and prior work

The two video streaming applications for NDN available at the time the project was developed are both plug-ins for existing media frameworks: VLC[2] and GStreamer[3]. The VLC plugin works by requesting data previously stored in a CCNx repo[1] and playing it back as a file. The

[1]The CCN repository, or "repo", is a standard tool within the CCNx package[4]. It is designed for persistent content storage.



```
Status
Video: PSize: 16/18 Segment: 38311 Timeout: 1.946 (0.534, 0.222) Retries: 68 Drops: 0 Duration: 1187s
Audio: PSize: 3/3 Segment: 4151 Timeout: 1.195 (0.564, 0.044) Retries: 131 Drops: 0 Duration: 1192s
Buffer: 100% (playing: Yes)
```

```
Status
Video: PSize: 17/18 Segment: 1657456 Timeout: 1.729 (0.570, 0.149) Retries: 29 Drops: 0 Duration: 51588s
Audio: PSize: 3/3 Segment: 76942 Timeout: 2.595 (1.327, 0.179) Retries: 19 Drops: 0 Duration: 51583s
Buffer: 100% (playing: Yes)
```

```
Status
Video: PSize: 17/18 Segment: 1729899 Timeout: 1.630 (0.447, 0.156) Retries: 50 Drop
Audio: PSize: 3/3 Segment: 203542 Timeout: 1.416 (0.523, 0.089) Retries: 110 Drops:
Buffer: 100% (playing: Yes)
```

```
Status
Video: PSize: 14/18 Segment: 13812949 Timeout: -1.000 (0.073, 0.115) Retries: 5299 Drops: 0 Duration: 343
Audio: PSize: 3/3 Segment: 1200746 Timeout: -1.000 (0.140, 0.124) Retries: 1282 Drops: 0 Duration: 344531
Buffer: 100% (playing: Yes)
```

```
Status
Video: PSize: 15/18 Segment: 12756 Timeout: 1.962 (0.453, 0.253) Retries: 19 Drops: 0 Duration: 254s
Audio: PSize: 3/3 Segment: 1235 Timeout: 1.273 (0.583, 0.053) Retries: 10 Drops: 0 Duration: 258s
Buffer: 100% (playing: Yes)
```

```
Status
  Video: PSize: 17/18 Segment: 2733 Timeout: 0.250 (0.114, 0.002) Retries: 9 Drops: 0
  Audio: PSize: 3/3 Segment: 69 Timeout: 1.899 (0.590, 0.190) Retries: 5 Drops: 0 Dura
  Buffer: 100% (playing: Yes)
```

Figure 1. Screenshots from NDNVideo-based webcams on the NDN testbed.

GStreamer plugin, CCNxGST, is slightly more sophisticated; it has its a publisher and consumer, as well as its own protocol to transmit the data. CCNxGST uses GStreamer's power of configurability, and is able to transmit any type of data over the network. Like the VLC plugin, the GST plugin treats the video as a binary stream of data. It provides a callback that can be used by consumers to know which is the latest segment. Therefore, with the right container, it can stream live video. Unfortunately, since the code does not understand what kind of data it transmits/receives, it does not fully use the potential of the NDN network. Its approach is similar to streaming video by downloading a file over the HTTP protocol, as opposed to using protocols that leverage the content type, such as RTP.

III. Design Goals

The initial goals of the NDNVideo project were to design and implement live and pre-recorded video streaming with random access (even on live sources) and no session semantics or negotiation, while providing long-term storage of content and synchronized playback across multiple consumers. The driver applications include popular web video streaming (e.g., by Youtube, Vimeo, and others) and future applications in digital signage, multimedia live events, and professional media production.

A. Live and pre-recorded audio/video streaming to multiple consumers

The basic capability required was to stream live or pre-recorded audio and video to multiple consumers with quality consistent with current internet video expectations. (For example, the initial tests of the project stream standard definition 720x480 video using 1024kbps H.264 encoding, and we plan to scale up to 1080p high definition over time.)

Figure 2. Television Studio #1 at UCLA, broadcasting over NDNVideo to the GENI demonstration described in the implementation section.

B. Random access based on actual location in the video (by video frame)

The approach should enable simple, low-latency random access into streams using a timecode-based namespace (i.e., the hour, minute, second, and frame of the stream). The work aims to enable a variety of interactive video features for even simple consumers via this feature.

C. Ability to synchronize playback of multiple consumers

For future digital signage, multi-camera, and interactive applications, the goal is to enable consumer-side synchronization of streams using the timecode-based namespace.

D. Passive consumers (no session semantics or negotiation)

The protocol should require no session semantics, which enables completely "passive" consumers of content and scalability without impact on the original source of the video.

E. Archival access to live streams

The protocol should provide on-the-fly archival of live streams. This makes them indistinguishable with pre-recorded streams for the purposes of most consumer applications. By using the CCN repository, storage is limited by repository capabilities only.

F. Content verification and provenance

Additionally, NDN's per-packet cryptographic signatures on ContentObjects provide a starting point for providing content verification and provenance in video applications.

$$\underbrace{\texttt{/ndn/ucla.edu/stream}}_{\text{routing prefix}}\ \underbrace{\texttt{/\%FD\%04\%F65ub\%0E}}_{\text{version number}}\ \underbrace{\texttt{/video0}}_{\text{video no.}}\ \underbrace{\texttt{/h264-1024k}}_{\text{codec}}\ \underbrace{\texttt{/segments/\%00}}_{\text{requested data}}$$

Figure 3. Example of data packet name.

IV. Architecture

The NDNVideo protocol has two types of participants: publisher and consumer. The streaming relationship is one-to-many (i.e. a single publisher publishes data that is received by many consumers). Unlike IP, NDN is pull-based, enabling the video publisher to be much simpler than its equivalent on TCP/IP. NDNVideo was designed to require no direct interaction between consumer and producer. The publisher simply prepares the packets, signs them, and stores them in the CCN repository for later retrieval by the consumers. The consumer no longer needs to inform the publisher about its Quality of Service because it is in full control of how much data and at what rate the data is being received. If, for any reason, the consumer needs to upgrade or downgrade bit rate, it can do so seamlessly by requesting different data and seeking to the same point.

A. Name Hierarchy

NDNVideo uses a simple packet format for encoded data rather than transmitting a typical container (avi, asf, qt, mov, and matroska are all typical video containers). For example, while publishing a pre-recorded Quicktime file, it extracts the streamed data from the Quicktime container and publishes every audio and video stream separately under a different part of the namespace. These streams, which could be in any format supported by GStreamer, can be requested and played back individually or in combination. Unlike previously-mentioned applications[3], [2], the protocol segments and provides access to the stream using semantically meaningful names, e.g. frames for video and samples for audio. Such structuring of data provides many benefits, one of which enables the publisher to uniquely name every frame and, in turn, allows the consumer to easily seek to a specific place in the stream. To allow more efficient playback after seeking, the data is also provided in a namespace using consecutive segment names.

An example name in the protocol is shown in figure 3. The name components are:

- **routing prefix** - a name component that is used to route Interests in the NDN network
- **version number** - version number based on current time. It is used to avoid name clashes if, for some reason, streaming is restarted or, in case of stored video, the publisher wants to replace it with a newer version
- **video no.** - video source number (e.g. first camera)
- **codec** - which codec was used to compress the stream (as an example, we use H.264 @ 1024 kbit/s)
- **requested data** - the index or segments of data requested (e.g., first frame of the video)

For a given stream, the following parameters are available as a collection of ContentObjects:

- `<stream>/key` - key used for signing the ContentObjects
- `<stream>/stream_info` - stream metadata (in case of video resolution: frame rate, etc; in case of audio: sampling frequency, number of channels)
- `<stream>/segments` - incrementally numbered data segments. This is where the actual stream data is stored. The segments are numbered incrementally, enabling the consumer to perform Interest pipelining based on predictable segment names.

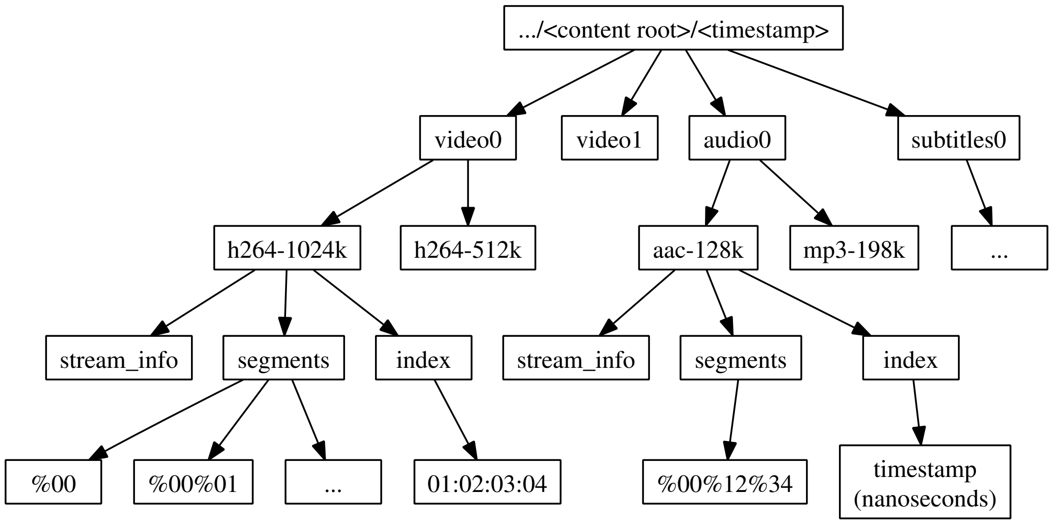


Figure 4. NDNVideo namespace.

- `<stream>/index` - time to segment number mapping. This name is used to allow consumers to perform seeking inside the stream data, permitting queries to convert time into segment number. Currently, a nanosecond resolution timestamp is used for both audio and video. This provides consistency across each type of media, and can be converted to frame-based timecode by the requesting consumer using the frame rate information available under `stream_info`. An index entry is generated per keyframe for video, or at a regular interval (currently two seconds) for audio. This namespace is not used during the continuous playback.

The NDNVideo namespace design is shown in figure 4.

B. Content Objects

The NDNVideo publisher generates signed ContentObjects from an input video stream, which can be either live or pre-recorded, and places them immediately into the local CCN repository. Network requests go only to that repository or caches in the network, never directly to the publisher. The data generated by the source are processed in units of frames (in case of video) and samples (in case of audio), which are called buffers. The buffers, in addition to data, have timestamp (in nanoseconds) and duration. The timestamp tells the player at what point in time the buffer is supposed to be played, while the duration tells how long[2].

1) Initial approach (video): The design goal was to provide random-access via video timecode, in which video is indexed by frames (e.g., HH:MM:SS:FF). Initially, every frame was placed into a separate ContentObject. The index namespace in this case contained keyframes (I-frames[3]) only.

[2]This allows for transmission of streams with varied frame rates. Such formats are becoming more popular, because they save bandwidth by lowering frame rates when not much action occurs in the video.

[3]H.264 video has several types of frames: keyframes (I-frames) and delta frames (P- and B-frames). I-frames contain all information necessary to render a picture, and they are the biggest. P-frames are much smaller, because they only contain the difference from the previous I- or P-frame. B-frames are even smaller, since they are like P-frames but also bi-directional. B-frames require that preceding and following frames are decoded first. An example pattern (referred as Group of Pictures - GOP) for encoded video could be: I, B, B, P, B, B, P, B, B, P, B, B, I.

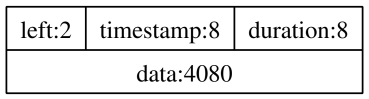


Figure 5. ContentObject (data packet) payload - initial approach. Values after the colons denote number of bytes.

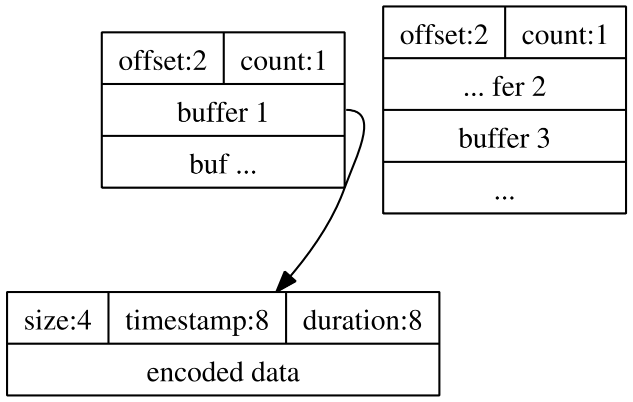


Figure 6. ContentObject payload - final approach.

Since some packets (mainly keyframes) can be bigger than a single packet, frames were split by the publisher code into multiple segments using standard CCNx conventions. The packet format is shown in figure 5. The **timestamp** and **duration** are the values given to us by the encoder. The **left** field is used when a frame is bigger than a single packet, and it tells how many more packets are left to obtain a full frame. The above packet worked very well while streaming video, but it is problematic for audio, leading to the refinement described below.

2) Final approach (adding audio): Digital audio in the target applications use sampling rates ranging from 22 kHz to 96 kHz and beyond (in professional media). If every sample were stored under a separate name as is done in video, every packet header would be bigger than its content, and there would be an impractical number of packets per second. The packet format was redesigned to address this, and is shown in figure 6. (Note that “active names” could be used to enable this abstraction, but avoiding any direct communication between consumers and the publishing process is the goal.)

NDNVideo handles large (video) and small (audio) buffers by packing the data in two layers. The inner layer contains all information necessary for the playback of the buffer (at this time, it is timestamp, duration, and data length), and it represents a single buffer. Then, multiple buffers are put inside one packet by trying to fill it completely. If a buffer cannot be fit into one packet, it is split into multiple packets to maximize utilization. The outer layer of the packet contains two additional fields: **offset** and **count**. The **count** tells us how many buffers begin in a given packet, and the **offset** is used to tell where the first buffer starts. The **offset** is used only on packet loss, and it lets the consumer quickly resume processing from the next available buffer. If **count** is 0, the packet is a continuation of the previous buffer and does not contain any new buffers; if the previous packet was lost, this one should also be discarded. (In this case, the **offset** has no meaning, and it is set to 0 in our implementation.)

Name: .../index	
AnswerOriginKind: NONE	
ChildSelector: RIGHTMOST	
Exclude:	00:00:05:01 ANY

Figure 7. Interest issued to seek to timecode 00:00:05:00. **Name** - the namespace to search for frame content object. **AnswerOriginKind** - ignore localhost (use network). **ChildSelector** - return the last element in the namespace after performing the exclusion. **Exclude** - do not return anything greater than 00:00:05:00.

C. Communication

1) Streaming playback: Given the pull-based nature of NDN, in NDNVideo, the consumer is fully in charge of data that it is receiving, which are pulled from the `segments` name through use of segment numbers. Such an approach makes the data names predictable, enabling the consumer to pipeline requests for the data. This is necessary to provide playback with satisfying quality, especially when latency between the publisher and the consumer is high. If a single buffer is contained in multiple packets, the packet header information is used to put the buffer back together. In case of packet loss, the consumer can either request the same ContentObject again or issue an Interest for the next segment. If the segment is considered lost, the **offset** field in figure 6 is used to determine the point at which the next buffer starts. The code does not need to wait for the buffers that start at the beginning of packets (e.g. keyframes).

2) Random-access seeking: Seeking in NDNVideo is done by the consumer issuing an Interest in the `index` namespace corresponding to the desired timecode, with Interest parameters designed to return the nearest keyframe ContentObject, which in turn maps time to segment number using a simple ASCII text payload. (The number of segments per frame can vary, so the consumer cannot know which segment corresponds to a specific time.) The names are dependent on the stream type. Currently, the publisher uses timecode for the video, which is a video editing standard. For example, timecode provided by a user interface is in the HH:MM:SS:FF (where HH, MM, SS are hour, minute, second, and FF is a frame number) format, which is converted to nanoseconds and expressed using CCNx segment notation, then used to issue Interests.

The publisher indexes only keyframes, since the closest preceding keyframe is needed to properly decode a specific frame of the video. To obtain this, the consumer uses NDN's Interest exclusions. The figure 7 shows an example of an Interest that seeks to timecode 00:00:05:00. In this case, the consumer would not know the timecode of the closest preceding keyframe (for example 00:00:04:20). So, it issues an Interest using the parent name `.../index` with Interest parameters set to get the correct keyframe.

First, by setting **ChildSelector** to **RIGHTMOST**, the consumer instructs the network to return the last element in the `index` namespace. Exclusions are used to tell the network what the consumer does not want to receive. This is to be done by providing individual names as well

Name: .../index	
ChildSelector: RIGHTMOST	
Exclude:	ANY <previously received index>

Figure 8. Interest issued to determine pre-recorded stream duration.

as ranges. By placing 00:00:05:01 and **ANY**[4] in the exclusion list, it is expressed that there is no interest in anything with or after timecode 00:00:05:01. That, combined with rightmost child selector, asks the network to return the latest index entry, just before 00:00:05:01. Since 00:00:04:20 is the nearest keyframe, that is what will be received as the resulting ContentObject. Finally, the **NONE** value for **AnswerOriginKind** tells the local network daemon to not use local Content Store and to use the network to issue the request. (This is important as previous data cached in the local Content Store may cause incorrect values to be returned.)

To render the stream, because the keyframe is at 00:00:04:20, but the desired start is at 00:00:05:01, the consumer silently fetches and decodes the required frames, and starts playback at 00:00:05:00. (Note that the data in the index ContentObject contains the segment number from which to start fetching the data for streaming playback, so this playback will happen with data from the segment namespace.)

3) Determining length of the live stream: Again, NDNVideo was designed so that there is no interaction between the publisher and consumers. This approach provides the ability to seek throughout the video. However, a typical live streaming requirement of the consumer is to seek to the most recent data. Therefore, a mechanism is needed to determine the (ever-increasing) duration of the stream. The Interest parameters shown in figure 8 are used to determine stream duration. This Interest is issued periodically during playback, and it checks for the last entry in the `index` namespace. The exclusion is used to bypass NDN's Content Stores between the consumer and the publisher. Without that parameter, the consumer would get previously retrieved information cached by the network. In the worst case, this approach may take N queries (where N is number of connecting nodes accessed) before converging on the correct "latest position". (This worst case happens when every node in the path(s) to the publisher caches a different response to the same query.)

4) Determining Interest timeout: In the ideal case, the consumer will get responses to all the Interests it issues. Unfortunately, packets can be dropped due to network congestion or any other issues (one of which is mentioned later in this paper). In order to provide seamless playback, it is important to know how long to wait before assuming that an Interest is lost. Interests for the data can then be quickly reissued or assumed unavailable, and the consumer can move on to the next segment.

The NDNVideo consumer adjusts its Interest timeout based on previous RTT values. For every Interest sent, the consumer stores the current timestamp; once response arrives, it uses that information to calculate round-trip time. Since the RTT values can vary greatly, the consumer smooths them out by using a low pass filter $SRTT = (1 - \alpha) * SRTT + \alpha * RTT$, and uses the

[4]Interest exclusions specify a sorted list of child names that the network is not to return. The **ANY** wildcard entry is supported and works similarly to an asterisk in filesystem globbing. If there are entries next to **ANY** they set bounds for the exclusion. For example, 10-ANY excludes 10 and everything after it, ANY-15 excludes everything from beginning up to and including 15, and 10-ANY-15 excludes everything between 10 and 15.

TCP/IP formula for smoothed variance $RTTVAR = (1 - \beta) * RTTVAR + \beta * |SRTT - RTT|$. The recommended values from RFC 2988[5], which are $\alpha = \frac{1}{8}$ and $\beta = \frac{1}{4}$, seem to work well for us. The Interest lifetime is calculated using $T_i = SRTT + 3 * RTTVAR$. The $3 * RTTVAR$ (rather than the $4 * RTTVAR$ of RFC 2988) was picked based on observations of the implementation's performance.

D. Usage modes

Some special cases require additional mechanisms in the NDNVideo consumer.

1) Live streaming: In the NDN approach, the publisher is much simpler than the corresponding one in IP; however, some complexity is shifted to the stream consumer. For example, cooperation from the publisher to maintain QoS (as done in RTP) is no longer needed. In this case, the consumer knows well whether data packets are lost, and there is no need to inform the publisher of how fast it needs to send the data, since the rate can be controlled on its own.

However, sometimes the push architecture can provide useful features. One such benefit is in live streaming, where the consumer typically wants the most recent data. In IP, the publisher simply pushes the most recent stream, and the consumer plays it as fast as possible, so the playback delay is equal to the latency of the network and size of player's buffer.

This simple problem in IP becomes more complex in NDN. Not only does the video consumer need to pipeline the data and fetch it quickly and efficiently, it should not fetch the data too quickly and request data that does not yet exist. To solve this problem, the consumer must determine what the latest data are and the rate at which it is supposed to request that data.

To determine the rate at which the consumer is supposed to issue Interests, in addition to the timestamp of the buffer, each individual packet contains a local time at which the packet was generated. While the consumer is not guaranteed to have a clock in sync with the publisher, the information is still useful because it can yield the time difference between the packets. The consumer can then estimate the mean time interval using a low pass filter: $T_s = (1 - \alpha) * T_s + \alpha * T$ with $\alpha = \frac{1}{8}$, and dynamically adjust the rate of Interests. The playback starts from the "end" of the live stream determined, as described in section IV-C3.

Another difference in live streaming, as opposed to prerecorded playback, is the need to maintain the playback rate. If the player does not receive data at a specific time, it must drop the outdated data and move to the next instead of pausing the playback for buffering. Solving this problem requires calculating the local time at which given content is supposed to be played back, as well as determining whether to send the data to the decoder or not. The consumer utilizes the receiver's local time and the buffer time (i.e., nanosecond timestamp for every frame/sample). Upon retrieving the latest segment as described in section IV-C3, the consumer calculates local time when the streaming started: $t_{start} = t_{local} - t_{buffer}$. Using this information, the consumer calculates the time it is supposed to play given the frame by simply adding the buffer's timestamp to the t_{start}. The NDNVideo consumer uses buffer time as opposed to segment time because specific frames and not data packets are dropped if information is late.

2) Previously recorded files: Compared to the complexities in live video stream seeking and pipelining, archival / recorded playback is straightforward; the duration is fixed. The only additional information needed for streaming from a pre-recorded file (or a live stream that completes) is a marker of the end of the stream. The publisher sets the value of `FinalBlockID` in the ContentObjects to the last segment number to signal the end of the stream. If the player uses multiple streams (e.g. audio and video), it stops playback after receiving the (End of Stream) EOS signal from all the streams.

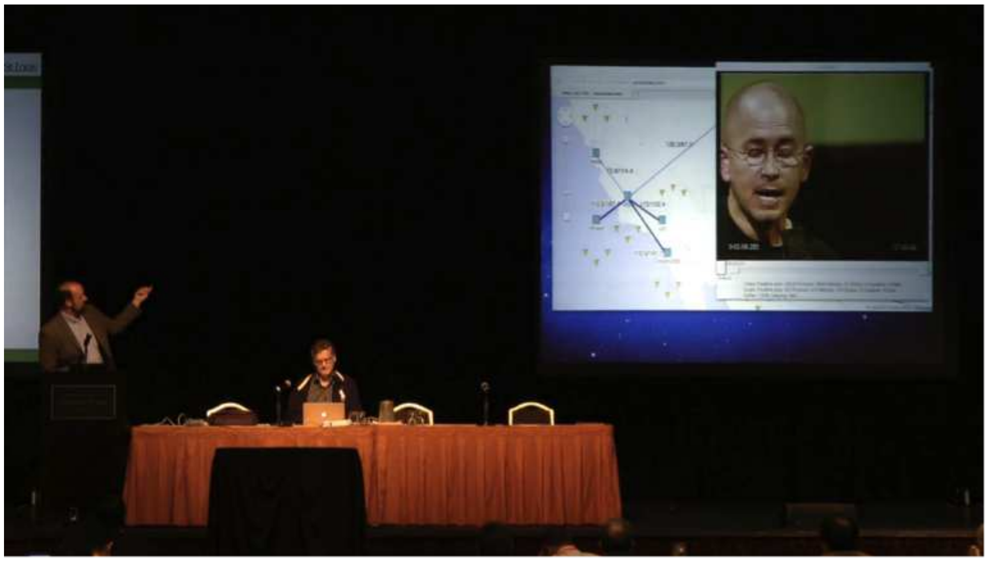


Figure 9. Consumer playback screenshot from NDNVideo broadcast at GENI.

V. Implementation and Testing

The implementation is written in Python, and uses the GStreamer [6] multimedia framework. It employs PARC's CCNx [4] implementation of the NDN architecture and our PyCCN [7] bindings. Both NDNVideo and PyCCN are open source (as are CCNx and GStreamer) and can be retrieved from GitHub. The URLs for the project's code repositories are listed at the end of the document.

After a variety of tests using pre-recorded files and live sources, the system was demonstrated "live" to an audience. A live, standard definition H.264-encoded stream (@ 1Mbit/sec) from a musical performance in a UCLA School of Theater, Film and Television studio was published over the NDN testbed to the Washington University team's demonstration for the GENI Engineering Conference in Los Angeles (March 2012), as shown in figure 2 and figure 9. Broadcast quality audio and video feeds from three cameras were mixed live and published to a CCN repo at UCLA. The WUSTL team retrieved the video using the player. In these and other tests with standard definition, H.264 video, the streaming works well to end-users. However, repeated (periodic) packet timeouts have occurred, described below.

Additionally, we have deployed webcams connected to application servers at several geographic locations on the NDN testbed, which use NDNVideo to provide live video content. Consumer screenshots are shown in figure 1.

VI. Scalability Evaluation

To evaluate scalability, multiple instances of a headless version of the video consumer were instantiated on Amazon EC2 and connected to the NDN project testbed. The test topology was designed as shown in figure 10. The node `hydra.remap.ucla.edu` is the data publisher, the Amazon nodes are the data consumers, and `borges.metwi.ucla.edu` is the CCNx router. The results are shown in figure 11 and figure 12, and are as expected. The first graph shows the benefits of caching. In this measure, about 2-3 minutes of the video was already cached on

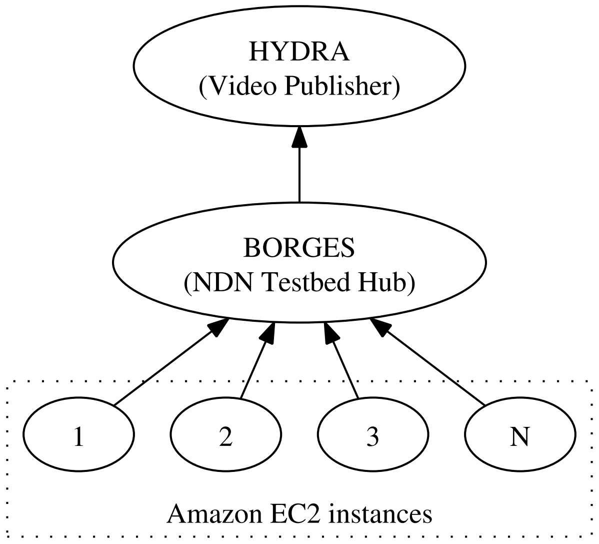


Figure 10. Amazon Web Services testing topology, in which each EC2 instance runs one or more headless video consumers.

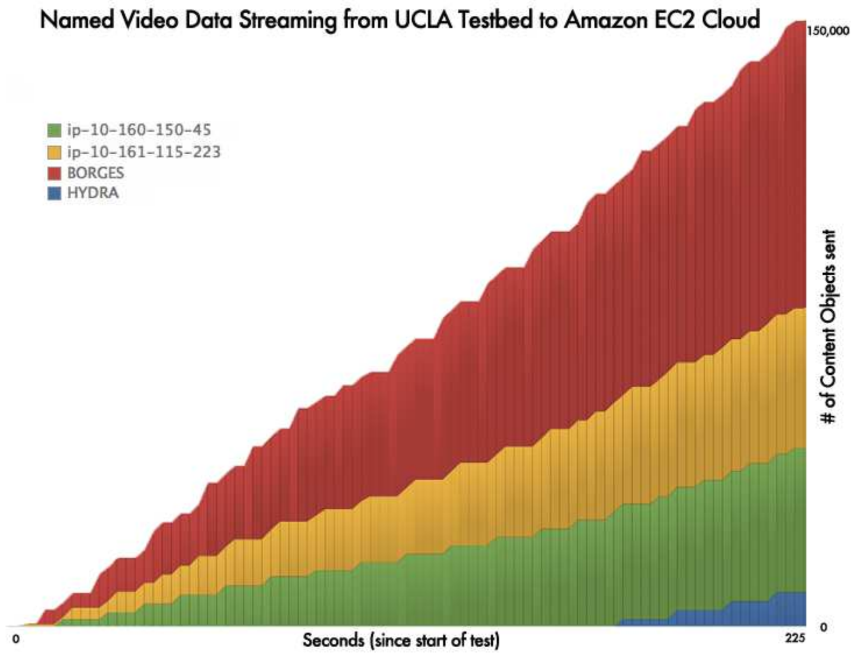


Figure 11. Illustration of caching while streaming to multiple consumers, where `hydra` is the source and `borges` is the content router.

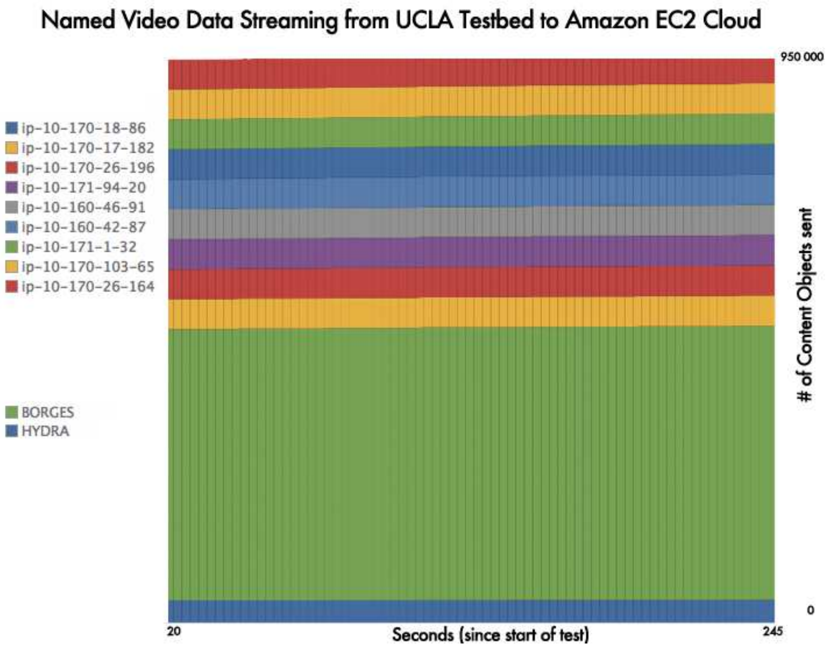


Figure 12. Number of ContentObjects sent per node, where `hydra` is the source and `borges` is the content router, and `ip-*` are the connecting clients.

`borges`. Once the cache data are used up, `borges` (red color) starts requesting new data from `hydra` (blue).[5]

Shown in figure 12 are how many data packets each node has sent after the nodes are run for a bit longer. `hydra` sends only one copy of the video, and `borges`, which acts as a single router in the network, duplicates the data to every EC2 node connected to it. Notice that the consumer nodes send slightly more data than the `hydra` provides. This is due to the issue described in the next paragraph, which causes the nodes to reissue Interests for the same content. Due to NDN design, this does not affect `hydra` at all.

In testing the consumer under nominal network conditions, many packet drops were observed, often happening in bursts. Measuring the round-trip time (RTT) showed that while the average RTT is quite low, there are large spikes of 1, 2 or even more seconds in duration. (While measuring the values, the Interest timeout was set to 2, so beyond that time, the packet was assumed lost.) The spikes were larger and happened more often when the consumer had more Interests pending. Even more surprising, as shown in figure 13 and figure 14, the large spikes happened even when the content was already cached on a local machine.[6]

[5]In these figures, consumers are seen sending data because the CCNx stack counts the local library's return of ContentObjects to the requesting application.

[6]The two graphs show RTT for two different approaches to Interest pipelining: size-based, in figure 13, which keeps a pipeline of N interests outstanding at any given time, and interval-based, which is shown in figure 14. The latter is explained further in section IV-D1; it attempts to determine the appropriate rate to issue interests to maintain playback under network conditions. This approach is still under development and may have had other bugs affecting this test, while it did seem to reduce these RTT spikes.

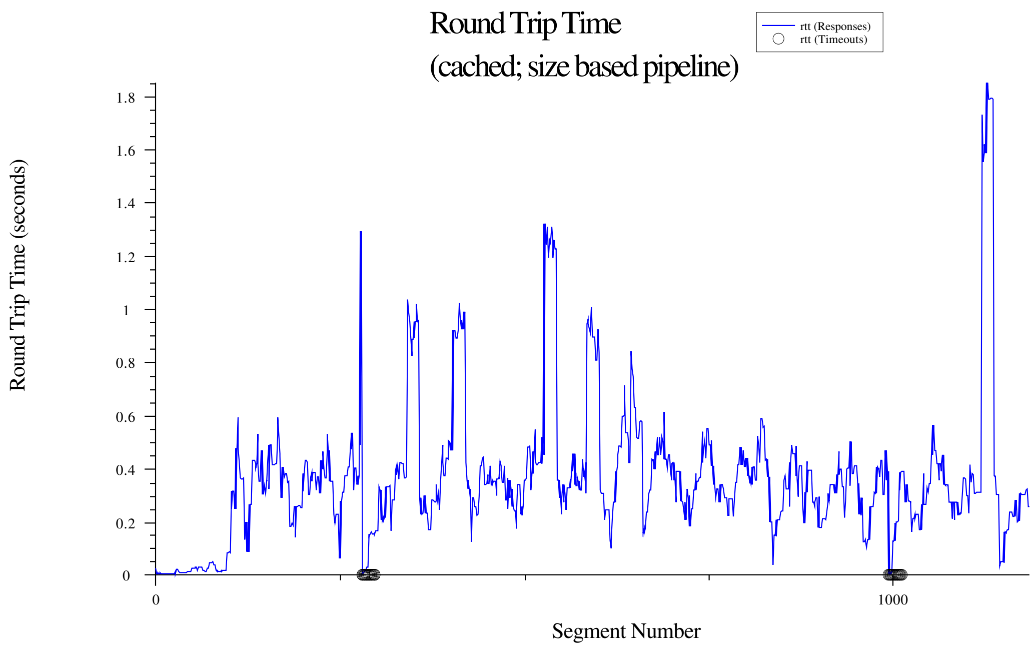


Figure 13. Round Trip Time in size-based pipeline.

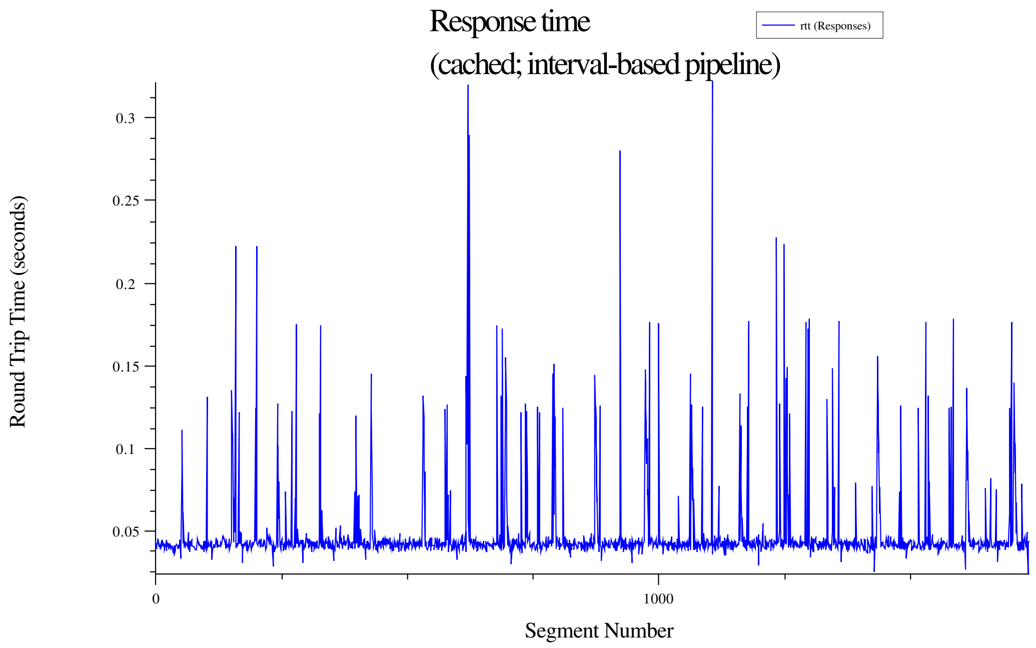


Figure 14. Round Trip Time in interval-based pipeline.

CCNx developers suggested the issues were caused by bufferbloat[7], CCNx Interest reordering, or a combination of the two. Because the issue also occurs with the locally cached content, while bufferbloat is still possible, the more likely cause is Interest reordering, which is necessary for CCNx's support of multicasting. The CCNx development team has plans for replacing it with a different approach in the future. They suggested that the interval-based pipeline, which spaces out the Interests expressed by the consumer, may partially mitigate the issue in the meantime. The test results appear to agree with this assessment.

Several client-side improvements were made to combat the RTT spikes. First, the consumer was modified to react more quickly to the spikes by estimating right timeout value (as described in section IV-C4). It was also modified to retry issuing an Interest once after a timeout before giving up. That approach works well, as it permits watching video streaming without any frames being dropped.

VII. Current issues and future work

A. *Current implementation issues*

Currently, the interval-based pipeline is still not reliable enough to be used widely. It is being refined for deployment in the next series of demonstrations and tests. Additionally, the CCNx repository does not yet implement data deletion, which can be problematic for long-running live streams. This feature will be available in an upcoming version of CCNx, and probably will not require any changes to the video code (only to repository configuration).

A few content verification features have not yet been added. For example, the current implementation does not check that the signatures on successive media packets are consistent, or confirm the publisher's key is the desired one. These are straightforward and will be added in a future version.

Finally, the goal is to enable the consumer to switch codec based on bandwidth or performance. For example, when the consumer detects that it cannot receive data at the desired rate, it could downgrade playback to a lower bit rate by simply changing a component of the Interest name it is requesting. An elegant way to do this would be to provide H.264 Scalable Video Coding (SVC) in the video namespace.

B. *Future support for Scalable Video Coding*

NDNVideo as tested used H.264/AVC (Advanced Video Coding) for video encoding, since it provides good quality per transmitted bit at a reasonable encoding complexity and is widely deployed in current video applications. In fact, the publisher code is capable of handling any video format that is contained in one stream and is supported by GStreamer. One interesting possible extension is H.264 Scalable Video Coding (SVC), though it is not yet supported by GStreamer [9]. SVC enables a single video stream to support multiple devices with various screen resolutions, bandwidth, and playback quality without encoding multiple versions of the same video. SVC instead uses multiple data layers providing progressive enhancement of the stream. Each SVC stream has a mandatory base layer, which has the lowest temporal, spatial and quality representation of the video stream, and several optional enhancement layers that can increase video frame rate, resolution or overall picture quality. The player can then fetch the base layer and additional enhancement layers based on need. SVC has the potential to be even more beneficial

[7]Bufferbloat is an interesting phenomenon that occurs when adding too much buffering actually reduces the speed of communication. A good discussion can be found on Jim Getty's blog[8].

in an NDN network than in IP, since less data duplication (i.e. separate streams of the same video) results in better utilization of data already cached on the routers.

Unfortunately, there is no known Open Source implementation of SVC; therefore, this feature could not be implemented. (It is suspected that transport complexity might play a role in the lack of implementations, which would be mitigated by an NDN-based approach that mapped available layers into the video namespace.) Outlined below are three options for a design based on the SVC specification:

1) A straightforward approach is to provide each layer as a separate stream. NDNVideo would treat SVC's enhancement streams the same as any other versions of the video in its naming hierarchy. For example, in figure 4, the enhancement layers would be on the same level as `h264-1024k` and `h264-512k`. The consumer would fetch their data as desired. This approach does not seem to have any obvious issues, except it is slightly wasteful. For example, the publisher will need to generate a separate index namespace for each layer and, when the consumer performs a seek during the playback, it will need to do a lookup for every layer used. This overhead may be small enough to be acceptable.
2) The stream hierarchy could be changed to put the layer id after the segment number (e.g. `stream/segments/%00%01/layer1`). Without specifying the last component, the consumer could retrieve base layer as usual, and then issue additional Interests for enhancement layer data as desired. However, the enhancement layers might not have the same amount of data as the base layer; most of the time, they will be smaller (i.e. the optional layer would be contained in fewer segments than the base layer), so to keep playback in sync, the publisher would need to place layer info every few segments. If the layer would turn out to be bigger than the base (needing larger number of segments than the base layer), the publisher would need to segment them, creating one more level by having segment of a segment (e.g. `stream/segments/%00%01/layer1/%00)`. Since it will be unlikely that the optional layers would be of the same size as the base layer, the consumer would need hints inside of the base layer, as to whether the given segment has enhancement layer information or not. Such information would complicate or eliminate Interest pipelining, as the consumer would need to receive the base segment prior to knowing if there was more data to pull. This approach seems to become complicated quickly.
3) Another reasonable approach may be to put one more level in the stream hierarchy and introduce layer names within the `segments` namespace as illustrated in figure 15. The `base` namespace would work exactly the same as in the original design. Layers with separate segment numbers allows packing a maximum number of data into packets and transmitting them efficiently without the issues described in the previous paragraph. Since each layer would have segments numbered independently, the index would need to be modified to contain a segment number for every available layer. This could add a few extra bytes to the index data packet, the size of which is still significantly smaller than the segment's data packet. Additionally, during playback, the consumer could dynamically enable and disable specific layers. Meanwhile, during seeking, it could find the right segment number of every layer by looking up the current location using the `index`. If extra lookups are not acceptable (perhaps due to high latency), the base segments could periodically update the consumer with that information.
4) All of the above approaches require the consumer to issue additional Interests for enhancement layer data. An alternative would be to provide a special namespace that, depending on the Interest name, would combine the desired enhancement layers. This would make

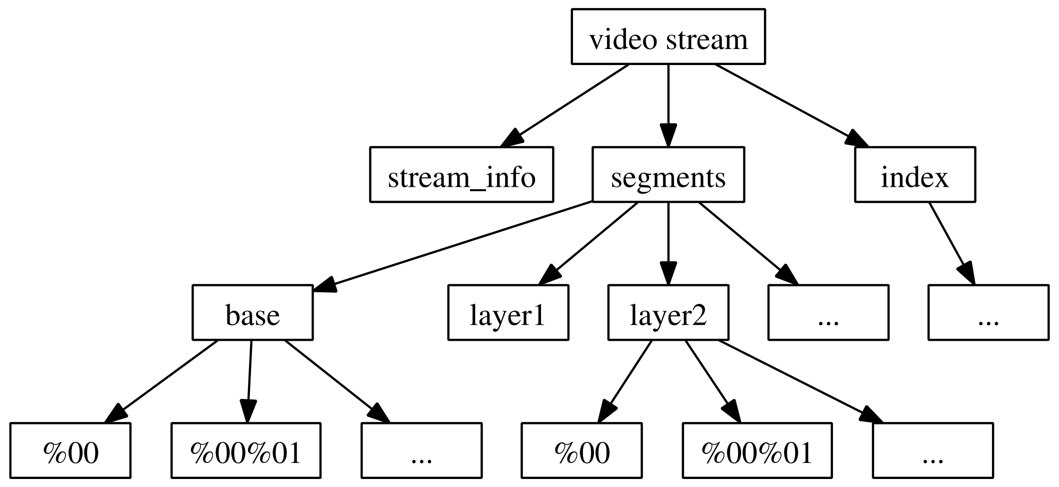


Figure 15. A possible namespace for SVC support.

playback by a single consumer more efficient (due to fewer Interests issued), but reduce cache utilization and nullify many benefits that the network provides. If such a feature is still needed, it might be best to implement it in a form of proxy that fetches data using described protocol, and converts it to a the simple stream namespace of the original design.

Exploring these design variations may be the objectives of a future project.

VIII. Conclusion

We were able to design, implement, and test a live (and pre-recorded) video streaming application that was conceived with named data network architecture in mind. While our approach posed its own challenges in terms of consumer complexity, it was still relatively simple to implement. There were significant benefits demonstrated, especially the ability to provide reliable playback with no session semantics or negotiation necessary between the consumer and the producer. The approach leverages Content Stores in the network, which makes the content distribution of the video much more efficient. With NDNVideo, video and audio streaming uses the intrinsic features of the network to scale to many consumers and provide efficient random-access, even to live streams, without loading the original publisher. We believe this will enable a better user experience with less strain on the publisher when compared to TCP/IP streaming, and that the approach could be used to enable serverless video publishing from resource-constrained mobile devices.

IX. Acknowledgments

This work would not have been possible without the support of many people. The authors thank Alex Horn for supplying Amazon EC2 measurements, webcam deployment, and code testing, as well as the CCNx developers, especially Michael Plass, for their assistance. Thanks also go to Alessandro Marianantoni for producing the GENI demonstration, and to Lixia Zhang, Van Jacobson, Deborah Estrin, Mario Gerla, and Giovanni Pau for their advice and discussion. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1040868.

X. Project Source

- NDNVideo - https://github.com/remap/ndnvideo
- PyCCN - https://github.com/remap/PyCCN

References

[1] L. Zhang, D. Estrin, J. Burke, J. V. Jacobson, D. Smetters, B. Zhang, G. Tsudik, D. Massey, C. Papadopoulos, T. Abdelzaher *et al.*, “Named data networking (ndn) project,” 2010.

[2] “CCNx vlc plugin,” part of the CCNx package. [Online]. Available: https://github.com/ProjectCCNx/ccnx/tree/master/apps/vlc

[3] J. Letourneau, “CCNxGST - GStreamer plugin used to transport media traffic over a CCNx network.” [Online]. Available: https://github.com/johnlet/gstreamer-ccnx

[4] “CCNx.” [Online]. Available: http://www.ccnx.org

[5] V. Paxson and M. Allman, “Computing TCP’s retransmission timer,” Internet Engineering Task Force, RFC 2988, Nov. 2000. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2988.txt

[6] “GStreamer - open source multimedia framework.” [Online]. Available: http://gstreamer.freedesktop.org

[7] “PyCCN - python bindings for CCNx.” [Online]. Available: https://github.com/remap/PyCCN

[8] J. Getty, “Jim Getty’s blog.” [Online]. Available: http://gettys.wordpress.com

[9] H. Schwarz, D. Marpe, and T. Wiegand, “Overview of the scalable video coding extension of the h.264/avc standard,” vol. 17, pp. 1103–1120, 2007. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4317636