NDN, Technical Report NDN-0006, 2012. http://named-data.net/techreports.html
Revision 2: July 2012

NDNLP: A Link Protocol for NDN

Junxiao Shi, Beichuan Zhang
The University of Arizona

1 Introduction

Current CCNx applications run on top of TCP/UDP/IP tunnels. It would be beneficial to be able to run directly on Ethernet, e.g., without having to configure IP addresses and without extra layer of processing. There are two major issues, though, namely (1) messages larger than Ethernet MTU cannot be sent, and (2) packet losses may degrade application performance.

In this work we propose a link protocol for delivering CCNx messages over a local one-hop link. This protocol, called NDNLP, defines a new message encapsulation and exchange protocol to provide the following two features:

- **Fragmentation and Reassembly** When a CCNx message is larger than the lower-layer's MTU, NDNLP will fragment the message into multiple MTU-compliant packets at the sender, and reassembly the original message at the receiver.
- **Acknowledgement and Retransmission** The receiver acknowledges the receipt of packets in a summary bitmap, and the sender can retransmit any lost packets.

NDNLP operates between the network layer (i.e., ccnd) and the "link", which can be any lower-layer delivery service, including TCP tunnels, UDP tunnels, Ethernet links, and so on. Its two features are optional: they can be turned on or off depending on the capability of lower-layer delivery. For example, both should be turned off when using TCP tunnels, but when using Ethernet, fragmentation/reassembly must be turned on.

We have made the initial design of NDNLP, implemented it on Linux with CCNx, and tested it on Ethernet links, and TCP/UDP tunnels. The rest of the paper will describe these in details.

2 Protocol Design

In this section we first describe packet formats of NDNLP, and then how it operates in fragmentation/reassembly and acknowledgement/retransmission.

A note about terminology: in this paper "messages" refers to what NDNLP receives from and sends to the upper network layer, and "packets" refers to what NDNLP receives from and sends to the lower link layers. Messages may be fragmented into multiple packets and get reassembled later. Messages are the payload of packets, whose header contain some extra information to facilitate NDNLP's functionality.

2.1 Packet Format

CCNx has two types of messages: Interest and Data. Each message is in XML under a well-known schema. Messages are encoded to a binary representation, ccnb [1], when transmitted either across hosts or between ccnd and local apps on the same host.

NDNLP has two types of packets: Link Data packets and Link Acknowledgement packets. Link Data packets are those carrying CCNx messages (Interest and Data) as payload, while Link Acknowledgement packets are used to detect packet loss. All NDNLP packets are XML documents encoded in ccnb too. This makes it easier to extend the packet format or be compatible with other link layer protocols developed in the future.

2.1.1 Link Data Packet

A Link Data packet carries a CCNx message or a fragment of it as the payload. It has the following XML structure:

```
<NdnlpData>
    <NdnlpSequence>sequence number</NdnlpSequence>
    <NdnlpFlags>flags</NdnlpFlags>
    <NdnlpFragIndex>fragment index</NdnlpFragIndex>
    <NdnlpFragCount>fragment count</NdnlpFragCount>
    <NdnlpPayload>payload</NdnlpPayload>
</NdnlpData>
```

These elements must appear in the given order. Their meanings are as follows.

1. **Sequence number**, 48 bits; required. A unique sequence number for each packet.
2. **Flags**, 16 bits; required. The following flags are defined:

   bit 0 RLA, requesting per-link acknowledgement
3. **Fragment index**, 16 bits; optional, default to '0' if not present. The 0-based index of current fragment within the message.
4. **Fragment count**, 16 bits; optional, default to '1' if not present. Total number of fragments of the message.
5. **Payload**, variable length; required. The payload.

2.1.2 Link Acknowledgement Packet

An acknowledgement packet encodes the receipt status of a collection of link data packets. It contains one or more acknowledgement blocks. Each block indicates the receipt status of a sequence of link data packets (i.e., they have continuous sequence numbers). Packet sequence numbers in different blocks don't have to be continuous. The packet's XML structure is as follows.

```
<NdnlpAck>
    <NdnlpAckBlock>
        <NdnlpSequenceBase>sequence base</NdnlpSequenceBase>
        <NdnlpBitmap>bitmap</NdnlpBitmap>
    </NdnlpAckBlock>
</NdnlpAck>
```

These elements must appear in the given order. There can be multiple NdnlpAckBlock elements. The meanings of the elements are as follows.

1. **Sequence base**, 48 bits. This is the sequence number of the first packet to which this acknolowedgment block applies.

2. **Bitmap**, variable length. It represents the receipt status of a block of packets with continuous sequence numbers. Each bit, when set to 1, indicates that a packet with the corresponding sequence number has been received. The first bit corresponds to the packet whose sequence number is the sequence base.

2.2 Fragmentation Operations

To send a ccnb-encoded message to another host, NDNLP layer slices it into N packets according to link MTU and send them. Packets belonging to the same message have continuous *sequence numbers*. The relative position of a fragment in the message is reflected by *fragment index* and *fragment count* fields. The receiver reassembles the message according to these three fields.

2.2.1 Sender Operation

To send a message of L octets on a lower layer with MTU, the sender first decides maximum payload length MPL, such that the total length of a resulting packet will not exceed MTU. Thus, the message needs to be sliced into $N = \lceil \frac{L}{MPL} \rceil$ fragments. N packets are created and assigned consecutive *sequence numbers*, *fragment index* from 0 to $N-1$, and have N as the *fragment count*. Each of the first $N-1$ packets contains MPL octets payload, and the last packet contains the remaining part.

For example, on $MPL = 1000$, a 2500-octet message is sent as the following 3 packets:

sequence	fragment index	fragment count	payload
600	0	3	0-999
601	1	3	1000-1999
602	2	3	2000-2499

2.2.2 Receiver Operation

The receiver maintains a **partial message store** data structure. Each item in the store represents a partially received message. Items are indexed by *message identifier*, which is the sequence number of the first packet in the message. Each partial message contains a list of received fragments. It also has a timestamp of latest packet arrival, so an incomplete message without new packets coming can be cleaned out after some time.

When a packet with *sequence number* S, *fragment index* I, and *fragment count* N is received, the receiver first calculates *message identifier* $M = S - I$. If item M does not exist in the store yet, it is created. The current fragment is added into the partial message, if it isn't a duplicate. When the partial message has all the fragments (as indicated by *fragment count*), the message is reassembled and delivered to upper layer, and the partial message object is removed from the store.

Two performance improvements can be made:

- If $N = 1$, the message is not fragmented, so it can be delivered right away without going through the partial message store.

- The receiver can pre-allocate the buffer when message length can be estimated, so that copying is not needed on reassembling.

However an attacker may inject many packets with very large fragment count as a Denial-of-Service attack. Thus, an implementation should limit buffer pre-allocation for large messages.

For example, if the packets in Section 2.2.1 are received in the order of 601, 600, 601, 602, the following will happen:

1. **601 arrival** $M = S - I = 601 - 1 = 600$. Create a partial message with message identifier 600 with buffer size $1000 \times 3 = 3000$, and copy the payload into octets 1000-1999.
2. **600 arrival** $M = S - I = 600 - 0 = 600$. A partial message with message identifier 600 exists, so copy the payload into octets 0-999.
3. **601 arrival** $M = S - I = 601 - 1 = 600$. A partial message with message identifier 600 exists, and fragment $I = 1$ already arrived, so this is a duplicate.
4. **602 arrival** $M = S - I = 602 - 2 = 600$. A partial message with message identifier 600 exists, so copy the payload into octets 2000-2499. Now the partial message has all the fragments, so the message is reassembled and delivered to upper layer, and removed from the store.

2.2.3 Discussions

The main difference between IPv4's design and NDNLP's design of fragmentation/reassembly is that in IPv4, reassembly only happens at the final destination, while in NDNLP it happens right after the fragmentation across a single hop. Reassembling messages at each hop is required because the receiver node needs to be able to cache the message at its entirety or verify its signature over the entire content. By including fragment count in the packet header, we allow receivers to improve its performance by allocating memory of appropriate size and avoiding copying things around.

2.3 Link Acknowledgement Operations

When per-link acknowledgement is enabled, NDNLP will set the *RLA* (requesting per-link acknowledgement) flag in link data packets to 1. The sender can store some recently sent packets for the purpose of retransmission, which is also optional and configurable. Acknowledgement and retransmission are performed on packet level.

NDNLP uses selective acknowledgement. Each acknowledgement block contains a bitmap where each bit indicates whether a packet is received. The receiver should accumulate received sequence numbers for a short time, pack them into one or more *acknowledgement blocks* and send back to the sender in one or multiple acknowledgement packets.

2.3.1 Sender Operation

When sending a message, the sender should determine whether per-link acknowledgement is needed for this message. This decision can be based on link quality, the importance of the message, and/or configuration. If the sender decides that per-link acknowledgement is needed, RLA flag is set to 1 for all packets associated with this message. Link acknowledgement packets themselves do not request per-link acknowledgement, nor can they be fragmented.

The sender maintains a **sent packet store** that stores a number of recently sent packets, indexed by their sequence numbers. These packets can be retransmitted if not acknowledged in time. The size of this store is configurable, and setting the size to zero will effectively disable retransmission.

When a packet requesting per-link acknowledgement is sent, it is inserted into the sent packet store along with a timestamp. When this packet is acknowledged, it is removed from the store.

A periodical process checks the store for packets that are sent some time ago, and retransmit them. The timeout period should be at least 4x link delay. Each packet can be retransmitted at most for a few times (e.g. twice), and within a limited period of time after the first transmission (e.g. 32x link delay).

The focus of this work is to provide the acknowledgment mechanism to detect packet losses. The details of how to deal with the losses, e.g., retransmission mechanisms, reporting to upper layers, etc. are subject to future work.

2.3.2 Receiver Operation

The receiver maintains an **acknowledgement queue** data structure, which is a list of sequence numbers of received packets that are requesting per-link acknowledgement. Once a packet with RLA set to 1 is received, its *sequence number* is appended into this queue.

A periodical process reads all sequence numbers from the acknowledgement queue, and packs them into one or more acknowledgement blocks that are sent in one or more packets. This process should be executed at least once per 2x link delay, to ensure every packet is acknowledged within 4x link delay.

Once an acknowledgement packet is sent, the sequence numbers are removed from the acknowledgement queue. If a later acknowledgement block happens to cover a previously acknowledged sequence number, the bit for that sequence number will be zero.

2.3.3 Discussions

TCP receivers by default acknowledge the last sequence number that has been successfully received without holes. The TCP Selective Acknowledgement Option (TCP SACK, [2]) allows a TCP receiver to acknowledge ranges of octets that have been successfully received, by specifying the sequence numbers of the beginning and the end of each block. NDNLP is similar to TCP SACK. This allows more accurate loss detection and retransmission. The differences are that NDNLP uses bitmaps to to encode the receipt status more efficiently, and acknowledgement and retransmission are done at the packet level, not octet.

3 Reference Implementation

To extend CCNx [3] with NDNLP protocol, we implement *ndnld*, a user-space daemon, as a link adaptor for ccnd. The ndnld maintains multiple "link connections", each can be Ethernet, IP, UDP, TCP, or other underlying protocols. Acting as a ccnd client program, ndnld creates a face for each connection on ccnd. From these faces, ndnld receives messages from local ccnd, perform its functions (e.g., fragmentation), and send them to remote hosts over the lower-level links/tunnels. A control program, called ndnldc, is also implemented to control the connections and faces of ndnld. The control is done using CCNx Interests, similar to how ccndc controls ccnd.

3.1 Architecture

There are two options implementing NDLP: (1) directly modify ccnd code to add the new functionality; (2) write a ccn client program to receive the messages and perform the functions outside of ccnd. At this early stage of development, we adopted the second approach so to test the protocol. In

the future when better performance or more functionality are needed, the current implementation can be integrated into ccnd code base.

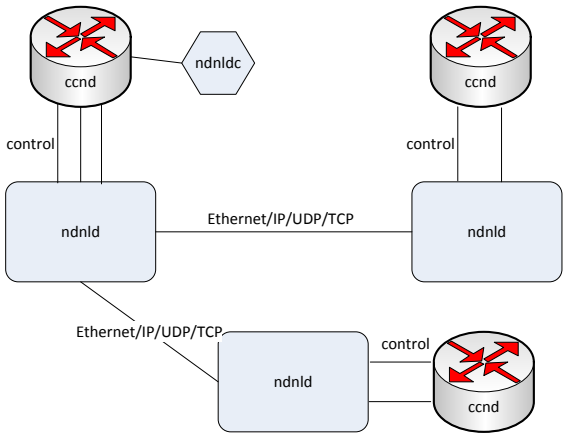

Figure 1: network architecture

The network architecture is shown in Figure 1. Each NDN router runs one ccnd process and one ndnld process. The ccnd process is responsible for routing and forwarding, and ndnld takes care of communicating with remote hosts via links/tunnels. A single ndnld process is capable of managing multiple connections, but each connection needs its own UNIX socket face on local ccnd, because routing and forwarding is done by ccnd. From the viewpoint of ccnd, messages sent into this face will be received by ccnd on another host through the corresponding face. To open a connection from host A to host B:

1. ndnld on host A opens a UNIX socket face with local ccnd of host A
2. ndnld on host A initiates a connection to ndnld on host B
3. ndnld on host B accepts the connection
4. ndnld on host B opens a UNIX socket face with local ccnd of host B

In addition, a separate *control face* is opened for accepting control commands. A utility program, ndnldc, sends a control command by expressing an Interest of specific name, which is forwarded by local ccnd to ndnld's control face. Face management and prefix registration is taken care of by ndnldc, which subsumes ccndc.

3.2 Software Components

We implemented ndnld and ndnldc in C.

Figure 2 illustrates software components and their relationship.

Core event system is the main loop of the program. It makes use of `poll` system call and non-blocking I/O, and invokes other components periodically or when data becomes available.

CCN client abstraction is a wrapper of ccn_client. It talks with local ccnd over a UNIX socket face, and conducts message integrity verification and message bound detection.

Control command listener listens on the *control face* for control commands. These control commands are Interests generated by ndnldc, and are used to add or remove connections and set parameters. **CCN face manager** talks with ccnd to add or remove faces for such connections.

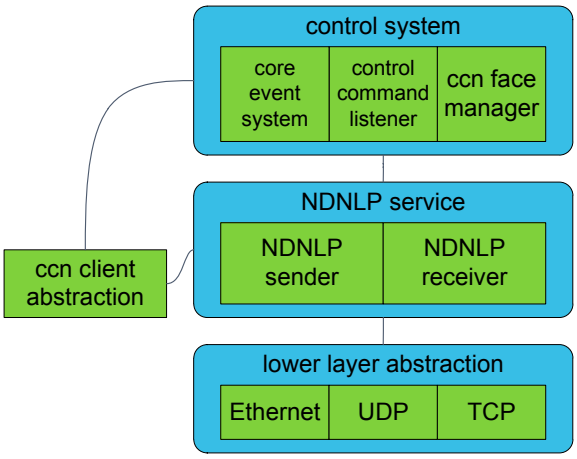

Figure 2: software components

Lower layer abstraction provides a generic lower layer interface out of different types of lower layers. **NDNLP sender** and **NDNLP receiver** implement NDNLP service above a generic lower layer.

3.3 Control Protocol

The ndnld control protocol provides a method for tools such as ndnldc to control the connections managed by ndnld.

3.3.1 Connection Management Protocol

The connection management protocol of ndnld supports "connect", "disconnect", and "listconnections" operations.

The design of connection management protocol borrows largely from CCNx Face Management Protocol [4], due to their similarity. A request operation is represented as a CCNx Interest with a CCNx ContentObject encoding the majority of the request parameters embedded as a component of the Interest name. A response is represented as a ContentObject for which the name matches the Interest, and the content encodes any necessary response data.

The ContentObject encoding requests and responses is NdnldConnection, that has the following structure:

```
<NdnldConnection>
    <Action>verb</Action>
    <FaceID>ccnd face ID</FaceID>
    <NdnldLowerProtocol>lower layer protocol</NdnldLowerProtocol>
    <Host>remote address</Host>
    <NdnldLocalInterface>local interface</NdnldLocalInterface>
    <ForwardingFlags>flags</ForwardingFlags>
    <NdnldSentPktsCapacity>sent packets store capacity</NdnldSentPktsCapacity>
    <NdnldRetransmitCount>retry count</NdnldRetransmitCount>
    <NdnldRetransmitTime>retransmit time</NdnldRetransmitTime>
```

```
    <NdnldAcknowledgeTime>acknowledge time</NdnldAcknowledgeTime>
    <StatusCode>connection state</StatusCode>
</NdnldConnection>
```

The elements must appear in the given order. Fields are described below:

1. **Verb**, UDATA; required. The verb "connect", "disconnect", or "listconnections".
2. **ccnd face ID**, UDATA; required in disconnect request and all responses, ignored otherwise. An integer representing the connection. Since each connection has a face on ccnd, it's sufficient to identify a connection by the ccnd face ID.
3. **Lower layer protocol**, UDATA; required in connect request and all responses, ignored otherwise. The lower layer protocol.
   - Ethernet: "ether"
   - UDP: "udp"
4. **Remote address**, UDATA; required in connect request and all responses, ignored otherwise. The host address of the lower layer protocol.
   - Ethernet: standard hex-digits-and-colons notation; must be parsable by ether_aton.
   - UDP: string representation of IPv6 or IPv4 address; must be parsable by inet_pton as a AF_INET6 or AF_INET address.
5. **Local interface**, UDATA; required when using Ethernet in connect request and all responses, ignored otherwise. The name of local Ethernet interface, such as "eth1".
6. **Flags**, BLOB 16 bits; required in connect request and all responses, ignored otherwise. The following flags are defined:

   bit 0 RLA, whether to request per-link acknowledgement on each outgoing packet
7. **Sent packets store capacity**, UDATA; optional, default to '100' if not present, ignored in requests other than connect. The maximum number of packets to be kept in sent packets store. The recommended value of this parameter is the estimated number of packets being sent in 2x round trip time.
8. **Retry count**, UDATA; optional, default to '5' if not present, ignored in requests other than connect. The maximum number of retries for each packet. The initial transmission is not counted within this limit.
9. **Retransmit time**, UDATA; optional, default to '1000' if not present, ignored in requests other than connect. The timeout, in milliseconds, after which a packet should be retransmitted if not acknowledged. The recommended value of this parameter is 3x round trip time.
10. **Acknowledge time**, UDATA; optional, default to '300' if not present, ignored in requests other than connect. The maximum time, in milliseconds, a sequence number can be kept in the acknowledge queue before being sent in an acknowledgement packet. The recommended value of this parameter is 1x round trip time.

11. **Connection state**, UDATA; required in all responses, ignored otherwise. The connection state, 'normal' or 'error'.

ndnldc then expresses an interest in `/ccnx/ndnld/`*`CCNDID`*`/control/`*`VERB`*`/`*`NFBLOB`* , where

- CCNDID is the ccndid of the local ccnd
- VERB is the operation verb
- NFBLOB is the signed content object

VERB occurs redundantly in both the interest prefix and in the NFBLOB. Its presence in the prefix is for dispatching the request. It is also in the NFBLOB, so that it is signed.

The listconnections response contains zero or multiple NdnldConnection objects, enclosed in a <Collection> element.

3.3.2 Prefix Registration Protocol

ndnld does not need a separate prefix registration protocol. Once ndnldc receives the ccnd face ID of a connection from the connection management protocol, it can use CCNx Prefix Registration Protocol [4] to talk with ccnd and register or unregister prefixes on the face.

3.4 The ndnldc Utility

ndnldc is a command line utility for managing the connections on local ndnld, and registering prefixes on the ccnd faces of those connections. It works by using ndnld Connection Management Protocol and CCNx Prefix Registration Protocol.

3.4.1 Creating a Connection

Creating a UDP connection

```
ndnldc -c -p udp -h 192.0.2.1
ndnldc -c -p udp -h 2001:DB8::1
```

-h IP address of destination host, either IPv6 or IPv4.

Creating an Ethernet connection

```
ndnldc -c -p ether -h 08:00:27:01:01:01 -i eth1
```

-h MAC address of destination host, in standard hex-digits-and-colons notation.

-i Name of local network interface.

Specifying parameters for a new connection

```
ndnldc -c ... -a -S 100 -C 5 -R 1000 -A 300
```

-a Whether to enable per-link acknowledgement. If this argument is present, RLA flag is set for packets sent from local ndnld; otherwise RLA flag is not set. Local ndnld will acknowledge received packets with RLA flag set, regardless of this setting.

-S Sent packets store capacity. This number of packets will be kept in sent packets store, so that they can be retransmitted if not acknowledged.

-C Retry count. An unacknowledged packet may be retransmitted for this number of times.

-R Retransmit time. A packet may be retransmitted if it is not acknowledged within this amount of time since last (re)transmission.

-A Acknowledge time. A sequence number cannot stay in acknowledge queue for more than this amount of time.

FaceID FaceID is written to stdout. A calling script should capture this FaceID in order to destroy the conneciton, and (un)register prefixes. If ndnldc -c is called with the same protocol, address, and local interface, the same FaceID will be returned.

Accepting a connection Currently ndnld does not listen for new connections. To make a working connection, you need to run ndnldc -c on both hosts, setting the address of the remote host.

3.4.2 Destroying a Connection

```
ndndlc -d -f 11
```

-f FaceID of the connection. This must be the ccnd FaceID of a connection managed by local ndnld; you cannot use this command to kill the control face, or a face of another application.

3.4.3 Listing Existing Connections

```
ndndlc -l
```

A human-readable list of existing connections is written to stdout. It looks like:

```
face=277 ether 08:00:27:01:01:01 on eth1
face=280 udp 2001:DB8::1
face=282 udp 192.0.2.1
```

3.4.4 Prefix Registration

Registering a prefix

```
ndnldc -r -f 11 -n ccnx:/example
```

-f FaceID of the connection. This should be the ccnd FaceID of a connection managed by local ndnld. It's possible to operate on the control face, or a face of another application, but this is not the intended use of this command.

-n The prefix. "ccnx:" can be omitted.

Unregistering a prefix

```
ndnldc -u -f 11 -n ccnx:/example
```

-f FaceID of the connection. This should be the ccnd FaceID of a connection managed by local ndnld. It's possible to operate on the control face, or a face of another application, but this is not the intended use of this command.

-n The prefix. "ccnx:" can be omitted. It must precisely match the prefix used in registration.

3.4.5 Error Handling

ndnldc returns 0 on success.

ndnldc returns 1 on failure, and writes any error messages to stderr. Some common failure reasons include:

- Local ccnd is not running.
- Local ndnld is not running.
- You are trying to operate on a FaceID that does not exist in local ndnld.
- The format of MAC address or IP address is incorrect.
- The local interface does not exist.

4 Testing

Testing is an important part of every software project. I conducted unit test, functionality test, and performance test for ndnld.

4.1 Unit Test and Debugging

I made use of CUnit [5] unit testing framework to test various components individually. These unit tests can help ensure code quality and the correctness of implementation.

ndnlink is a light-weight version of ndnld. It is a command line program that can manage one NDNLP connection, and it does not contain the control system. This program is used to test the NDNLP service components.

Debugging tools such as gdb and valgrind are used to fix logical errors, segment faults, and memory leaks in the program.

4.2 Test Environment

The test environment consists of four Ubuntu 11.10 64-bit machines, on a VirtualBox virtualization environment. Four Intel Xeon E5645 @2.40GHz CPU cores are available to each machine. Each machine has two network interfaces, and eth1 of all four machines are connected to the same Ethernet switch. All machines are running ccnx-0.6.0 stable release.

ndnld, ndnldc, and ndnlink programs are installed to /usr/local/bin on each machine. ndnld and ndnlink have 'SetUID' enabled so that they are able to create Ethernet sockets.

A network-layer topology shown in Figure 3 is configured with ndnldc. Each machine is assigned a name prefix, and `ccnpingserver` [6] is running on this name prefix. FIB is populated statically with assigned name prefixes by using ndnldc.

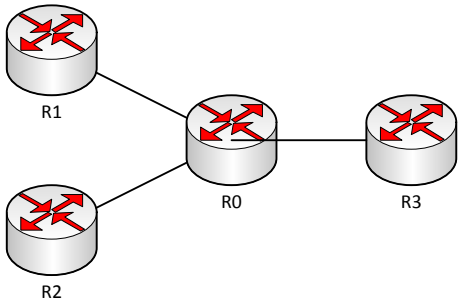

Figure 3: test environment topology

4.3 Wireshark NDNLP Dissector

ndnlp.lua is a Wireshark plugin which implements a dissector for NDNLP. It is able to identify NDNLP packets, and display certain header fields.

Although this dissector is not able to find out every NDNLP field, it is sufficient for debugging and testing this NDNLP implementation.

4.4 Functionality: Fragmentation

R1 acts as the publisher; a file of 6600 bytes is published under R1's prefix with `ccnpoke` command. R0 acts as the consumer; it expresses an interest for that file with `ccnpeek` command.

```
 3 0.024903   CadmusCo_a7:ef:fa  CadmusCo_23:b0:11  NDNLP   1514 Data 8a4810d0f08b 0/5
 4 0.024912   CadmusCo_a7:ef:fa  CadmusCo_23:b0:11  NDNLP   1514 Data 8a4810d0f08c 1/5
 5 0.024922   CadmusCo_a7:ef:fa  CadmusCo_23:b0:11  NDNLP   1514 Data 8a4810d0f08d 2/5
 6 0.024932   CadmusCo_a7:ef:fa  CadmusCo_23:b0:11  NDNLP   1514 Data 8a4810d0f08e 3/5
 7 0.024941   CadmusCo_a7:ef:fa  CadmusCo_23:b0:11  NDNLP   1248 Data 8a4810d0f08f 4/5
 8 0.133674   CadmusCo_23:b0:11  CadmusCo_a7:ef:fa  NDNLP     43 Ack  8a4810d0f08b
```

Figure 4: fragmented message

A Wireshark capture on R0 (Figure 4) shows that the ContentObject message is fragmented into 5 packets.

4.5 Functionality: Acknowledgement and Retransmission

The same experiment is repeated. This time, R1 is running ndnlink instead of ndnld, and a lossy option is enabled in ndnlink so that it drops outgoing packets with 5% probability.

A Wireshark capture on R0 (Figure 5) shows that the data packet with sequence 'aaaf95d90680' was lost during the first transmission, and it is retransmitted after about 1000ms (the default retransmission time).

4.6 Performance: Delay

To measure the delay of ndnld and compare to ccnd's native UDP/TCP face, `ccnping` is invoked on R2 to ping the `ccnpingserver` located on R1. Interests and Contents will traverse two hops (R2-R0 and R0-R1). 20000 Interests are sent at an interval of 5 milliseconds (200 Interests per seconds; experiment lasts for 100 seconds).

```
25 21.313683  CadmusCo_23:b0:11  CadmusCo_a7:ef:fa  NDNLP     94 Data ce992d232b89
26 21.345770  CadmusCo_a7:ef:fa  CadmusCo_23:b0:11  NDNLP   1514 Data aaaf95d9067e 0/5
27 21.345820  CadmusCo_a7:ef:fa  CadmusCo_23:b0:11  NDNLP   1514 Data aaaf95d9067f 1/5
28 21.345832  CadmusCo_a7:ef:fa  CadmusCo_23:b0:11  NDNLP   1514 Data aaaf95d90681 3/5
29 21.345843  CadmusCo_a7:ef:fa  CadmusCo_23:b0:11  NDNLP   1248 Data aaaf95d90682 4/5
30 21.496909  CadmusCo_23:b0:11  CadmusCo_a7:ef:fa  NDNLP     43 Ack  aaaf95d9067e
31 22.031574  CadmusCo_a7:ef:fa  CadmusCo_23:b0:11  NDNLP     43 Ack  ce992d232b89
32 22.396187  CadmusCo_a7:ef:fa  CadmusCo_23:b0:11  NDNLP   1514 Data aaaf95d90680 2/5
33 22.446651  CadmusCo_23:b0:11  CadmusCo_a7:ef:fa  NDNLP     43 Ack  aaaf95d90680
```

Figure 5: packet retransmission

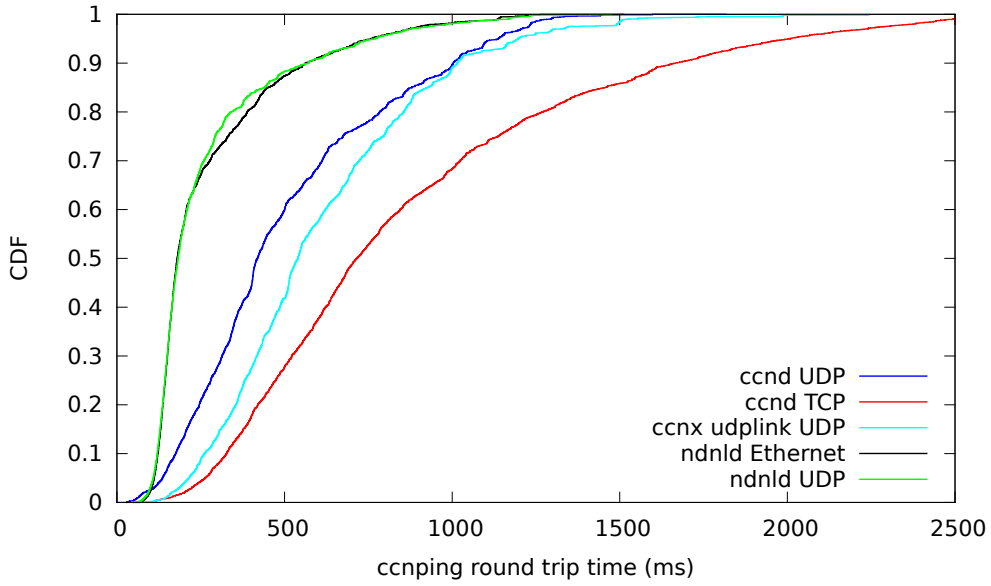

Figure 6: ccnping round trip time on various link implementations

link program	lower-layer protocol	timeouts	round trip time (ms)		
			min	median	max
ccnd	UDP	2.03%	29.545	419.130	1816.55
ccnd	TCP	0.00%	57.739	715.417	3102.440
ccnx udplink	UDP	0.35%	56.395	539.820	2241.140
ndnld	Ethernet	0.00%	51.283	179.651	1511.120
ndnld	UDP	1.11%	46.587	182.490	1475.110

The result shows that, in terms of delay, ndnld has better performance than ccnd's native UDP/TCP face.

4.7 Performance: Resource Consumption

This experiment measures the system resource consumption of ndnld. R1, R2, R3 each runs two `ccnping` processes that continuously ping two other hosts. The CPU consumption of ndnld on R0 is observed by `top`.

ccnping interval (ms)	msg/sec on R0	ndnld CPU	ccnd CPU
1000	12	1%	1%
100	120	8%	9%
8	1500	28%	33%

The result shows that, ndnld consumes about 85% of the CPU cycles consumed by ccnd. Most servers are equipped with multiple CPU cores, and both ccnd and ndnld are single-threaded so each can take one CPU core. Thus, as long as ndnld consumes less CPU cycles than ccnd, the CPU consumption of ndnld is not a big concern.

A Protocol Numbers

lower layer	field	protocol number
Ethernet	Ethertype	0x8624
IP	protocol	150
UDP	port	9695
TCP	port	9695

B ccnb DTAG Codes

CCNx project assigned a block of 16 DTAGs starting with 20653248 for use with NDNLP. [7]

Element	DTAG
NdnlpData	20653248
NdnlpSequence	20653249
NdnlpFlags	20653250
NdnlpFragIndex	20653251
NdnlpFragCount	20653252
NdnlpPayload	20653253
NdnlpAck	20653254
NdnlpAckBlock	20653255
NdnlpSequenceBase	20653256
NdnlpBitmap	20653257

B.1 DTAGs in Control Protocol

The ndnld control protocol is using a private range of DTAGs. This is allowed because those DTAGs only appear in messages to or from ndnld, so the dictionary can be inferred from context [1].

Element	DTAG
NdnldConnection	20653264
NdnldLowerProtocol	20653265
NdnldLocalInterface	20653266
NdnldSentPktsCapacity	20653267
NdnldRetransmitCount	20653268
NdnldRetransmitTime	20653269
NdnldAcknowledgeTime	20653270

References

[1] "CCNx Binary Encoding (ccnb)." [Online]. Available: http://www.ccnx.org/releases/latest/doc/technical/BinaryEncoding.html

[2] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgment Options," RFC 2018 (Proposed Standard), Internet Engineering Task Force, Oct. 1996. [Online]. Available: http://www.ietf.org/rfc/rfc2018.txt

[3] "CCNx." [Online]. Available: http://www.ccnx.org/

[4] "CCNx Face Management and Registration Protocol." [Online]. Available: http://www.ccnx.org/releases/latest/doc/technical/Registration.html

[5] "CUnit, A Unit Testing Framework for C." [Online]. Available: http://cunit.sourceforge.net/

[6] "ccnping." [Online]. Available: https://github.com/NDN-Routing/ccnping

[7] "CCNx - Feature #100710: Assign DTAGs for experimentation with link-level protocols." [Online]. Available: http://redmine.ccnx.org/issues/100710