ICN "Begin-End" Hop by Hop Fragmentation draft-mosko-icnrg-beginendfragment-01

Abstract

This document describes a simple hop-by-hop fragmentation scheme for ICN and mappings to the CCNx 1.0 and NDN packet formats, called "begin-end fragmentation". This scheme may be used at Layer 3 when ICN packets are used natively over a Layer 2 media which does not reorder packets.

Status of this Memo

Copyright Notice

Table of Contents

TOC

1. Introduction

In the past, there were two known hop-by-hop fragmentation schemes for ICN packets: The one described in NDNLP (Shi, J. and B. Zhang, “NDNLP: A Link Protocol for NDN,” July 2012.) [NDNLP] as "indexed fragmentation" and the one implemented in CCN-lite (Mosko, M., Plass, M., and C. Tschudin, “CCN-Lite fragmentation,” Summer 2012.) [CCNLite], using the old ccnb encoding. In a first part, this document describes a third, hop-by-hop fragmentation protocol in an encoding-neutral way. In a second part, we show mappings of this "begin-end fragmentation scheme" to the CCNx Messages in TLV Format (Mosko, M., Solis, I., and C. Wood, “CCNx Messages in TLV Format (Internet draft),” 2016.) [CCNMessages] and the NDN TLV [NDN] encoding. Thirdly, possible extensions and their encodings are discussed, for example reporting link reliability or link ARQ schemes such as windowing protocols.

The proposed hop-by-hop "begin-end fragments" scheme may be used at Layer 3 when large ICN messages are to be natively sent over a Layer 2 media with a small MTU. In cases where ICN packets are carried over an existing Layer 3 protocol, such as IP, the Information Centric Network SHOULD use that protocol's native fragmentation.

This proposed fragmentation scheme is an adaptation of PPP Multilink PPP Multilink (Sklower, K., Lloyd, B., McGregor, G., Carr, D., and T. Coradetti, “The PPP Multilink Protocol (MP),” August 1996.) [RFC1990] fragmentation between peers identified by their Layer 2 identity. It is appropriate for standard Layer 2 media that guarantee in-order packet delivery.

Definitions:

- (Network Protocol) Packet: A layer 3 ICN datagram, such as a Content Object or Interest, which is too large to be transmitted over a given L2 technology.
- Fragment: The datagram containing all serialized data fields required by the proposed fragmentation protocol. Depending on the mapping, the fragment will contain these fragment protocol specific data but also, for example, a CCNx fixed header, optional Per-Hop-Headers and/or and validator fields like

checksums or signatures.
- Fragment Header: The serialized CONTROL data structures of the proposed fragmentation protocol plus mapping specific bits.
- Fragment Data (or payload): The portion of the original Network Protocol Packet that is carried in the Fragment.
- Frame: A layer-2 frame in which the Fragment will be transferred.

Fragments are represented as 32-bit wide words using ASCII art. Because of the Type-Length-Value encoding used (TLV) and optional fields or sizes, there is no concise way to represent all possibilities. We use the convention that ASCII art fields enclosed by vertical bars "|" represent exact bit widths. Fields with a forward slash "/" are variable bit widths, which we typically pad out to word alignment for picture readability.

TODO -- we have not adopted the Requirements Language yet.

TOC

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 (Bradner, S., “Key words for use in RFCs to Indicate Requirement Levels,” March 1997.) [RFC2119].

TOC

2. Abstract Description of the Begin-End-Fragment Protocol

A Fragment is defined as the following fields (plus any additional fields required by the wire formatting in which fragments are encoded):

```
Fragment       = FragProtocolID FragProtocolData
FragProtocolID = <BeginEndFragmentsProtocolID> | (other protocols' ids)
FragProtocolData = BeginEndFragmentsData | (other protocols' data)
BeginEndFragmentsData = Flags FragSeqNo FragLength FragData
Flags          = B / E / BE / I
B              = <begin flag>
E              = <end flag>
BE             = <begin and end flag>
I              = <idle flag>
FragSeqNo      = 1*OCTETS
FragLength     = <Octets of fragment data>
FragData       = <Continuous octets (portion of Packet)>
```

The fragmentation protocol is run between a sender and a "peer", which can be one or more, potentially passive, receivers. They execute first an Initialization Protocol (Initialization), then use a Sender Protocol (Sender Protocol) and Receiver Protocol (Receiver Protocol) to exchange frames.

The initialization protocol uses a reliable messages exchange to reset the the FragSeqNo to 0 on both peers. This ensures that when one or another peer restarts both peers will reset their state.

The sender breaks a packet P (typically an Interest- or Content packet in the embedding wire encoding) into one or more fragments which are tagged with monotonically increasing sequence numbers. The B, E and BE flags are used to signal the start of a fragment series (B), its end (E), or a single fragment (BE) for the given packet P.

It is advisable that the 'B' fragment contains enough information in its Fragment Data to let the receiver know the total length of the packet to be reconstructed (and size of the reassembly buffer to be allocated) and the type of the expected packet.

The receivers listen to the fragment stream and reconstruct from a valid fragment series the original packet, and reject fragments with invalid sequence numbers, flags, or validation data.

The 'I' flag allows the sender to send idle frames that do not contain any Fragment Data, but do increment the fragment sequence number. This is useful on lossy links to indicate that the sender is past the end of the previous packet in case the 'E' fragment was lost. Moreover, as a possible extension of the protocol, this allows for periodic keepalives, measuring for example link quality when there is no other traffic to send.

TOC

2.1. Initialization

Whenever a peer begins operation, it must reliably reset the sequence number space. If the underlying link already ensures a complete reset of both peers, that method MAY be used. Otherwise, the method presented here SHOULD be used.

1. For each peer, a node tracks its local state, S_LOCAL[peer], and its peer's state S_REMOTE[peer]. A state may be INIT or OK.
2. RESET messages carry a sequence number denoted as N_LOCAL[peer] and N_REMOTE[peer], for the messages in each direction. The sequence number MAY be a positive random number.
3. A node tracks its FragSeqNo as FSN_LOCAL[peer] and the next expected FragSeqno from its peer as FSN_REMOTE[peer].
4. We will drop the [peer] subscript from the state variables in the following with the understanding that this protocol executes per pair of peers.
5. If S_LOCAL or S_REMOTE is in INIT state, the node MUST only send RESET or RESET_ACK messages. Note that two sides need to both be in the same set of states. One side could be in OK/OK and another in INIT/OK. In that case the side in OK/OK may begin sending data to the peer, though the peer cannot reply with data until it progresses to OK/OK.
6. Upon initialization, a peer sets its FSN_LOCAL and FSN_REMOTE to 0, and both S_LOCAL and S_REMOTE to INIT. It sets N_LOCAL to a new sequence number (i.e. a positive random number) and N_REMOTE to 0.
7. A first peer, called A, sends {RESET, N_LOCAL} to its peer, called B.
8. Upon receiving a message {RESET, N}:
    1. If N does not equal N_REMOTE, set N_REMOTE to N, set both FSN_LOCAL and FSN_REMOTE to 0. If S_REMOTE is not INIT, set S_LOCAL to INIT. Set S_REMOTE to OK. The node MUST discard any partially received packet, and MUST clear its current

partially transmitted packet (if any), though it MAY re-queue it.
    2. Send {RESET_ACK, N} to the peer
9. Upon receiving a message {RESET_ACK, N}:
    1. If N does not equal N_LOCAL, discard the message.
    2. Else, set N_LOCAL to OK. If this sets both N_LOCAL and N_REMOTE to OK, this node may begin sending data. It cancels any pending RESET timer.
10. Upon sending a first RESET message, the sending node starts a retransmit timer for the peer. It begins at RESET_TIMOUT (50 msec).
11. At each RESET timeout, a node sets a new timeout as twice the previous timeout. If it is less than MAX_TIMEOUT (4 seconds), it sends {RESET, N_LOCAL} again and starts a reset timer with the new timeout. If it is not less than MAX_TIMEOUT, it reports an error for the peer.

TOC

2.1.1. Examples

The first example, Figure 1 shows an example message sequence diagram of the initialization process. The "T" column is Time, The "S" column is for S_LOCAL and S_REMOTE, the N column is for N_LOCAL and N_REMOTE, and the FSN column is for FSN_LOCAL and FSN_REMOTE. We show both peers, one on the left side of the figure and one on the right side; call them nodes "A" and "B". We only show an entry in the table when state changes.

- T=0: A initializes and sets its state values as shown, picking the message sequence number 5. It sends the message {RESET, 5} to B. It starts a RESET timer.
- T=2: Node B initializes, picking message sequence number 7. It sends {RESET, 7} to A. It starts a RESET timer.
- T=3: Node A receives {RESET, 7}. It sets S_REMOTE to OK and records N_REMOTE as 7. It sets FSN_LOCAL and FSN_REMOTE to 0.
- T=4: Node A sends {RESET_ACK, 7} to node B.
- T=5: Node B reeives {RESET_ACK, 7}, so it sets S_LOCAL to OK. It cancels its RESET timer.
- T=6: Node A's RESET timer expires and it re-sends {RESET, 5} to B.
- T=7: Node B receives {RESET, 5}. It records N_REMOTE as 5 and sets S_REMOTE to OK. At this point, Node B is in OK/OK state and may begin sending data.
- T=8: Node B sends {RESET_ACK, 5}.
- T=9: Node B begins sending data to A.
- T=9: Node A reveives {RESET_ACK, 5} and sets S_LOCAL to OK. It is now in OK/OK state and may begin sending data to B.
- T=10: Node A begins sending data to B.

```
|---|-----|-----|-----|                             |-----|-----|-----|
| T |  S  |  N  | FSN |                             |  S  |  N  | FSN |
|   | L R | L R | L R |                             | L R | L R | L R |
|---|-----|-----|-----|                             |-----|-----|-----|
| 0 | I I | 5 0 | 0 0 | >-{RESET, 5}--------->X|    not ready      |
|   |     |     |     |                             |     |     |     |
| 2 |     |     |     | <---------{RESET, 7}-< | I I | 7 0 | 0 0 |
| 3 |   OK| 5 7 | 0 0 |                             |     |     |     |
| 4 |     |     |     | >-{RESET_ACK, 7}-----> |     |     |     |
```

```
| 5 |     |     |     |                          |OK    | 7 0 | 0 0 |
| 6 |     |     |     | >-{RESET, 5}---------> |      |     |     |
| 7 |     |     |     |                          |   OK|   5 | 0 0 |
| 8 |     |     |     | <-----{RESET_ACK, 5}-< |      |     |     |
| 9 |OK   |     |     | <-------------{data}-< |      |     |     |
| 10|     |     |     | >-{data}-------------> |      |     |     |
|---|-----|-----|-----|                          |-----|-----|-----|
```

Figure 1

TOC

2.2. Sender Protocol

1. A sender maintains a separate state machine for each peer.
2. When a peering is established, the FragSequenceNumber begins at 0.
3. After sending a Fragment, FragSequenceNumber is incremented by one.
4. In the first fragment for a packet, set the B bit to '1'.
5. In the last fragment for a packet, set the E bit to '1'.
6. Both the B and E bits must be set to '1' for a single fragment.
7. If both the B and E and I bits are not set, the fragment is in the middle of a series.
8. When not sending a fragment (with fragment data), the sender may send an Idle fragment with only the 'I' bit set. This indicates that the sender has no packet to send. Idle frames may only be sent in between E and B frames.

TOC

2.3. Receiver Protocol

1. A receiver maintains one reassembly queue per sender.
2. Discard Idle fragments.
3. Discard fragments until a 'B' fragment is received. Store the received sequence number for this sender.
4. If an out-of-order fragment is received next, discard the reassembly buffer and go to step (2).
5. Continue receiving in-order fragments until the first 'E' fragment. At this time, the fragmented packet is fully re-assembled and may be passed on to the next layer.
6. The receiver cannot assume it will receive the 'E' fragment or a subsequent 'I' frame, so it should use a timeout mechanism appropriate to the link to release preallocated memory resources.

TOC

3. Ethernet as a common use case

We expect that Ethernet will be the most common L2 technology where the proposed ICN fragmentation will be used, therefore we briefly elaborate on how fragmentation functions with the broadcast and multi-protocol nature of Ethernet.

When the fragmentation protocol is used with Ethernet, each participant uses the tuple {source mac, destination mac, ethertype} to identify a send or receive buffer and FragSequenceNumber number space.

If the fragmentation protocol is using a group address destination, each group address is considered a "peer" with its own FragSequenceNumber. For example, the MAC address 0x01005E0017AA on EtherType 0x0801 is the CCNx assigned group address for its 224.0.23.170 IP multicast address. Each sender would maintain a FragSequenceNumber for that peer. Each receiver would maintain a separate reassembly buffer for that group address based on the sender and ethertype.

If using other Ethernet encapsulations, such as 802.1AE MacSec, one could use a security identifier in place of the {source, destination, ethertype} tuple.

TOC

4. CCNx 1.0 Fragment Protocol Description

The hop-by-hop fragmentation protocol introduces a new CCNx 1.0 Packet Type called PT_FRAG and uses new fields in the Fixed Header. The hop-by-hop headers of a CCNx 1.0 fragment may be used for purposes like link quality reporting or a reliable ARQ scheme, which are out-of-scope of this document.

We describe a basic hop-by-hop fragmentation header, using bits in the Fixed Header for the fragment encoding. We also describe an extended version with variable sequence number size that puts the fragmentation header in the body of the CCNx message. This allows the fragmentation header to be signed or covered by a MIC. The extended encoding sets the 'X' flag to 1 in the Fixed Header, otherwise it is the basic encoding.

The "hop-by-hop fixed header" follows the normal conventions: The Version, PacketLength, and HeaderLength fields are as per CCNx Messages in TLV Format (Mosko, M., Solis, I., and C. Wood, “CCNx Messages in TLV Format (Internet draft),” 2016.) [CCNMessages]. The PacketType is set to PT_FRAG. However, in the packet-type dependent fields, we reserve 4 bits for flags and 20 bits for a sequence number.

The "message part" of the CCNx 1.0 fragment carries the fragment data in its own TLV block. The message part may also contain standard CCNx validation algorithm and validation bits in subsequent TLV blocks. In this way, the fragment can be covered by a CRC32C checksum or stronger validation methods.

```
                    1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+---------------+---------------+
|    Version    |    PT_FRAG    |         PacketLength          |
+---------------+---------------+---------------+---------------+
|X|B|E|I|          FragSequenceNumber           | HeaderLength  |
+---------------+---------------+---------------+---------------+
/ Optional Hop-by-hop header in TLV format                      /
+---------------+---------------+---------------+---------------+
| Fragment Data TLV                                             /
+---------------+---------------+---------------+---------------+
/ Optional CCNx ValidationAlgorithm TLV                         /
+---------------+---------------+---------------+---------------+
/ Optional CCNx ValidationPayload TLV (ValidationAlg required)  /
+---------------+---------------+---------------+---------------+
```

- X: Extended Format (X=0 shown above)
- B: Begin flag.
- E: End flag.
- I: Idle flag.
- FragSequenceNumber: a 20-bit sequence number to identify the fragment (see below).

The FragSequenceNumber follows Serial Number Arithmetic (Elz, R. and R. Bush, “Serial Number Arithmetic,” August 1996.) [RFC1982] for a 20-bit serial number. This means we have 19 bits of "valid" sequence number space, or 524,288 fragments. The packets per second for a 10 Gbps link with 1500 bytes Ethernet frames is 833,333 packets per second. Therefore, the 20-bit sequence number space allows for 629 milliseconds of frames.

In CCNx 1.0, the maximum encapsulated length is 64 KB -- which requires under 50 PT_FRAG frames of 1500 bytes, depending on the HeaderLength and validation options. So the valid sequence number space (when e.g. used over Ethernet) is approximately 10,500 maximum size (network protocol) packets.

We do not specify a link negotiation protocol for CCNx 1.0 at this time. Such a protocol could negotiate certain starting or operation parameters of the protocol, such as the starting fragment number or the size of the sequence number field.

If a PT_FRAG packet has optional hop-by-hop headers, the implementation should pass the fragment to the appropriate subsystem to process those headers before discarding the fragment.

The Extended encoding (X=1) moves the fragment header fields (= flags and sequence number) to the CCNx packet's message part, so they are covered by any ValidationAlgorithm used on the packet. It also allows for variable length sequence numbers. In the example shown below, there is a 7-byte (56-bit) sequence number.

The Extended encoding also allows different fragmentation protocols to co-exist by changing the opening TLV type from T_FRAG_HEADER to a new type (specified in an external document).

The first 8 bytes of the first fragment are the FixedHeader of the encapsulated Packet, so one may learn the overall length of the Packet from that FixedHeader.

```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+---------------+---------------+
|    Version    |    PT_FRAG    |         PacketLength          |
+---------------+---------------+---------------+---------------+
|1|0|0|0|                Reserved               | HeaderLength  |
+---------------+---------------+---------------+---------------+
/ Optional Hop-by-hop header in TLV format                      /
+---------------+---------------+---------------+---------------+
|          T_FRAG_HEADER        |              8                |
+---------------+---------------+---------------+---------------+
|1|B|E|I|0|0|0|0|          FragSequenceNumber                   /
/                                                               /
+---------------+---------------+---------------+---------------+
| Fragment Data TLV                                             /
+---------------+---------------+---------------+---------------+
/ Optional CCNx ValidationAlgorithm TLV                         /
+---------------+---------------+---------------+---------------+
/ Optional CCNx ValidationPayload TLV (ValidationAlg required)  /
+---------------+---------------+---------------+---------------+
```

TOC

4.1. Example

We present a complete example of the basic fragment encoding for a 2KB Content Object for 1500 byte frames according to the protocol described in this draft (with clear X-flag). The original 2KB packet also has an RSA signature, but this cannot easily be used for integrity checking as the receiver may not have the appropriate key and it is an expensive operation. We therefore chose to use a CRC32C validator on each fragment. The Content Object has the name ccnx:/abcd. First, the original 2000 byte packet is shown in entirety.

```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+---------------+---------------+
|       1       |       2       |             2000              |
+---------------+---------------+---------------+---------------+
|            Reserved           |     Flags     |       20      |
+---------------+---------------+---------------+---------------+
|          T_CACHETIME          |               8               |
+---------------+---------------+---------------+---------------+
|                                                               |
|                    Recommended Cache Time                     |
+---------------+---------------+---------------+---------------+
|        T_CONTENTOBJECT        |             1508              |
+---------------+---------------+---------------+---------------+
|            T_NAME             |               8               |
+---------------+---------------+---------------+---------------+
|          T_NAMESEGMENT        |               4               |
+---------------+---------------+---------------+---------------+
|       a               b              c              d         |
+---------------+---------------+---------------+---------------+
|           T_PAYLOAD           |             1492              |
+---------------+---------------+---------------+---------------+
/                        Payload Contents                       /
+---------------+---------------+---------------+---------------+
|        T_VALIDATION_ALG       |             204               |
+---------------+---------------+---------------+---------------+
|          T_RSA-SHA256         |             200               |
+---------------+---------------+---------------+---------------+
|             T_KEYID           |              32               |
+---------------+---------------+---------------+---------------+
/                              KeyId                            /
/---------------+---------------+-------------------------------+
|          T_PUBLICKEY          |             160               |
+---------------+---------------+---------------+---------------+
/                 Public Key (DER encoded SPKI)                 /
+---------------+---------------+---------------+---------------+
|      T_VALIDATION_PAYLOAD     |             256               |
+---------------+---------------+---------------+---------------+
/                          RSA Signature                        /
+---------------+---------------+---------------+---------------+
```

The 2000 byte packet will be fragmented into two pieces. In the first fragment, there is 28 bytes of overhead (fixed header 8, T_STD_FRAGMENT 4, validation 16), so the fragment's payload size is 1472 bytes. In the second packet, the T_FRAGMENT block carries the remaining data 528 bytes, hence the overall packet size is 556 bytes due to the same 28 bytes of overhead. We used FragSequenceNumber "0" and "1" for the two fragments.

```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+---------------+---------------+
|       1       |    PT_FRAG    |             1500              |
+---------------+---------------+---------------+---------------+
|0|1|0|0|                 0x00000               |       8       |
+---------------+---------------+---------------+---------------+
|          T_FRAGMENT           |             1472              |
+---------------+---------------+---------------+---------------+
|       1       |       2       |             2000              |
+---------------+---------------+---------------+---------------+
|            Reserved           |     Flags     |       20      |
+---------------+---------------+---------------+---------------+
|          T_CACHETIME          |               8               |
+---------------+---------------+---------------+---------------+
|                                                               |
|                    Recommended Cache Time                     |
+---------------+---------------+---------------+---------------+
|        T_CONTENTOBJECT        |             1508              |
+---------------+---------------+---------------+---------------+
|            T_NAME             |               8               |
+---------------+---------------+---------------+---------------+
|          T_NAMESEGMENT        |               4               |
+---------------+---------------+---------------+---------------+
|       a                b              c              d        |
+---------------+---------------+---------------+---------------+
|           T_PAYLOAD           |             1492              |
+---------------+---------------+---------------+---------------+
/             Payload Contents  (1432 out of 1492 bytes)        /
+---------------+---------------+---------------+---------------+
|       T_VALIDATION_ALG        |               4               |
+---------------+---------------+---------------+---------------+
|            T_CRC32            |               0               |
+---------------+---------------+---------------+---------------+
|      T_VALIDATION_PAYLOAD     |               4               |
+---------------+---------------+---------------+---------------+
|                         CRC32C Value                          |
+---------------+---------------+---------------+---------------+

                    1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+---------------+---------------+
|       1       |    PT_FRAG    |             556               |
+---------------+---------------+---------------+---------------+
|0|0|1|0|                 0x00001               |       8       |
+---------------+---------------+---------------+---------------+
|          T_FRAGMENT           |             528               |
+---------------+---------------+---------------+---------------+
/                Payload Contents  (last 60 bytes)              /
+---------------+---------------+---------------+---------------+
|        T_VALIDATION_ALG       |             204               |
+---------------+---------------+---------------+---------------+
|          T_RSA-SHA256         |             200               |
+---------------+---------------+---------------+---------------+
|             T_KEYID           |              32               |
+---------------+---------------+---------------+---------------+
/                            KeyId                              /
/---------------+---------------+-------------------------------+
|          T_PUBLICKEY          |             160               |
+---------------+---------------+---------------+---------------+
```

```
/                Public Key (DER encoded SPKI)                  /
+---------------+---------------+---------------+---------------+
|     T_VALIDATION_PAYLOAD      |             256               |
+---------------+---------------+---------------+---------------+
/                        RSA Signature                          /
+---------------+---------------+---------------+---------------+
|      T_VALIDATION_ALG         |              4                |
+---------------+---------------+---------------+---------------+
|           T_CRC32             |              0                |
+---------------+---------------+---------------+---------------+
|     T_VALIDATION_PAYLOAD      |              4                |
+---------------+---------------+---------------+---------------+
|                       CRC32C Value                            |
+---------------+---------------+---------------+---------------+
```

TOC

4.2. CCNx 1.0 Frame Packing

A sender/receiver pair may multiplex non-fragmentation frames on the same link. For example, in CCNx 1.0, there may be some PacketType PT_FRAG frames and some plain PT_INTEREST or PT_CONTENTOBJECT frames on the same link between the same pairs. PT_FRAG frames are considered independently of other frames between the pair.

Because each CCNx 1.0 datagram with a Fixed Header has all information needed for framing, two peers may pack multiple CCNx 1.0 datagrams in to one Layer 2 frame. For example, if there are several small Interests queued back-to-back, they could be encapsulated in a single Ethernet frame, up to the maximum Ethernet payload.

At the extreme, a peer may use fragmentation for all packets and completely pack each Layer 2 frame. The tail fragment would be cut off at whatever byte length fits the remaining Layer 2 frame.

Example: Assume that the outgoing queue for a specific peer has the following four packets to be sent:

```
interest1(200B)
content1(3500B)
interest2(200B)
content2(500B)
```

With 12 bytes of overhead per fragment these four packets could be fragmented and packeted into three Ethernet frames as:

```
[frag(interest1,200B), frag(content1, 1276B)]
[frag(content1, 1488B)]
[frag(content1, 236B), frag(interest2, 200B), frag(content2, 500B)]
```

TOC

4.3. Assigned Numbers for CCNx 1.0 Begin-End fragmentation

```
PT_FRAG             0x03
T_FRAGMENT          0x05
T_FRAG_HEADER       0x10
```

TOC

5. NDN Fragment Protocol Description

The Begin-End fragmentation protocol described in this draft extends the NDN link protocol v2 packet format (NDNLPv2). Note that this extension is not an official part of the NDN suite, at this point in time.

NDNLPv2 packets have a start type NDNLP-TYPE which distinguishes them from the classic Interest and Data packets. Inside the NDNLPv2 TLV structure, a sequence of NDNLPv2 header fields precede the payload (fragment data) which is introduced by the type value NND-FRAGMENT-TYPE.

```
NDNpacket       ::= Interest | Data | NDNLP
NDNLP           ::= NDNLP-TYPE TLV-LENGTH
                      NDNLPhdrFields*
                      NDNLPfragment?
NDNLPhdrFields ::= BeginEndField | (other NDNLP header fields)
BeginEndField   ::= BEGIN-END-FIELD-TYPE TLV-LENGTH
                      BYTE BYTE+
NDNfragment     ::= NDN-FRAGMENT-TYPE TLV-LENGTH
                      BYTE+
```

The extension for the "begin-end" fragmentation scheme relies on a new header field with type value NDN-BEGIN-END-FIELD-TYPE: The presence of this field marks a NDNLPv2 packet as a "begin-end" fragment. The field's value is 1 to 8 bytes long and consists of 2 flag bits (most-significant bits) plus a sequence number (remaining less-significant bits).

For a sender/receiver pair and for a given direction, the value of the BeginEndField is of constant size. But depending on the start configuration, different sizes can be chosen for operations, both in time and for the different directions.

In the smallest possible setup (e.g. sensor network with very small MTUs), the BeginEndField can have a one-byte value (2 flag bits plus 6 sequence number bits). For Ethernet, it it is recommended to use a 3-byte value (2 flag bits plus 22 sequence number bits).

An idle fragmentation frame is encoded as a NDNLP packet with a Begin-End Field but no NDNfragment element. Both the B- and the E-flags should be set to 1 in this case.

The frist bytes of the first fragment are the outermost NDN TLV of the encapsulated Packet. One may learn the overall length from the the outermost TLV length.

TOC

5.1. Example

We present an example of the basic fragment encoding for a payload of size larger than 253 Bytes and less than 64KB.

```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+---------------+---------------+
| type=NDNLP    | len=N+9                                       |
+---------------+---------------+---------------+---------------+
| type=BeginEnd | len=3         |B|E|       FragSequence...     |
+---------------+---------------+---------------+---------------+
| ..Number      | type=Fragment | len=N                         |
+---------------+---------------+---------------+---------------+
|               | FragmentData (N bytes) ...                    /
+---------------+---------------+---------------+---------------+
/                                                               /
+---------------+---------------+---------------+---------------+
/               |
+---------------+
```

- B: Begin flag.
- E: End flag.
- FragSequenceNumber: a 22-bit sequence number to identify the fragment.

TOC

5.2. NDN Frame Packing

A sender/receiver pair may multiplex non-fragmentation frames on the same link. For example, in NDN, there may be some NDNLP frames and some plain Interest or Data frames on the same link between the same pairs. NDNLP frames are considered independently of other frames between the pair.

NDNLP does not allow for frame packing: A frame contains only one out of the three Interest, Data and NDNLP packet types.

TOC

5.3. Assigned Numbers for NDN Begin-End fragmentation

```
NDNLP-TYPE             0x64   // official, might change
BEGIN-END-FIELD-TYPE   0x5c   // inofficial
NDN-FRAGMENT-TYPE      0x52   // official, might change
```

TOC

6. Acknowledgements

TOC

7. IANA Considerations

This memo includes no request to IANA.

All drafts are required to have an IANA considerations section (see Guidelines for Writing an IANA Considerations Section in RFCs (Narten, T. and H. Alvestrand, “Guidelines for Writing an IANA Considerations Section in RFCs,” May 2008.) [RFC5226] for a guide). If the draft does not require IANA to do anything, the section contains an explicit statement that this is the case (as above). If there are no requirements for IANA, the section will be removed during conversion into an RFC by the RFC Editor.

TOC

8. Security Considerations

This protocol has no security mechanisms and is vulnerable to injection attacks by other devices on the same physical link as the fragmentation peers. One should use a secure Layer 2 protocol, such as 802.1AE (MacSec) to prevent such attacks.

TOC

9. References

TOC

9.1. Normative References

[RFC2119] Bradner, S., “Key words for use in RFCs to Indicate Requirement Levels,” BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997.

TOC

9.2. Informative References

[CCNLite] Mosko, M., Plass, M., and C. Tschudin, “CCN-Lite fragmentation,” Summer 2012.

[CCNMessages] Mosko, M., Solis, I., and C. Wood, “CCNx Messages in TLV Format (Internet draft),” 2016.

[CCNx] PARC, Inc., “CCNx Open Source,” 2007.

[NDN] “NDN specification Documentation, Release 0.1a2,” March 2014.

[NDNLP] Shi, J. and B. Zhang, “NDNLP: A Link Protocol for NDN,” NDN Technical Report NDN-0006, July 2012.

[RFC1982] Elz, R. and R. Bush, “Serial Number Arithmetic,” RFC 1982, DOI 10.17487/RFC1982, August 1996.

[RFC1990] Sklower, K., Lloyd, B., McGregor, G., Carr, D., and T. Coradetti, “The PPP Multilink Protocol (MP),” RFC 1990, DOI 10.17487/RFC1990, August 1996.

[RFC3552] Rescorla, E. and B. Korver, “Guidelines for Writing RFC Text on Security Considerations,” BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003.

[RFC5226] Narten, T. and H. Alvestrand, “Guidelines for Writing an IANA Considerations Section in RFCs,” BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008.

TOC

Authors' Addresses

Marc Mosko
PARC
Palo Alto, California 94304
USA
Phone: +1 650-812-4405
Email: marc.mosko@parc.com

Christian Tschudin
University of Basel
Basel 4051
Switzerland
Phone: +41 61 267 0557
Email: christian.tschudin@unibas.ch