ICNRG Working Group	C. Tschudin
Internet-Draft	University of Basel
Intended status: Informational	C. Wood
Expires: January 7, 2017	PARC, Inc.
	July 06, 2016

File-Like ICN Collection (FLIC)

draft-tschudin-icnrg-flic-01

Abstract

This document describes a bare bones “index table”-approach for organizing a set of ICN data objects into a large, File-Like ICN Collection (FLIC).

At the core of this collection is a so called manifest which acts as the collection’s root node. The manifest contains an index table with pointers, each pointer being a hash value pointing to either a final data block or another index table node.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

1.1. FLIC as a Distributed Data Structure

One figure

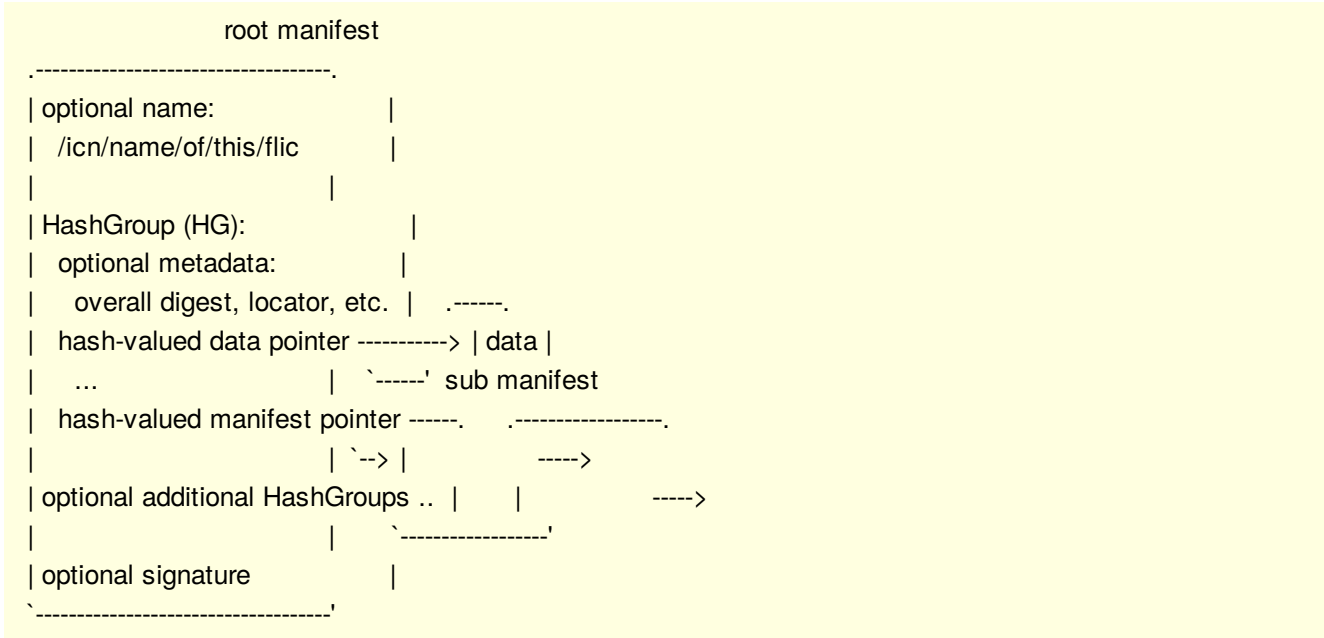

Figure 1: A FLIC manifest and its directed acyclic graph

1.2. Design goals

- Copy the proven UNIX inode concept:
  - index tables and memory pointers
- Adaption to ICN:

    - hash values instead of block numbers, unique with high probability
- Advantages (over non-manifest collections):
    - single root manifest signature covers all elements of the full collection, including intermediate sub manifests
    - eliminate reference to chunk numbering schemata (hash values only)
    - supports block-level deduplification (can lead to a directed acyclic graph, or DAG, instead of a tree)
- Limitations
    - All data leafs must be present at manifest creation time (otherwise one cannot compute the pointers)
- Potential extensions (for study):
    - Enhance the manifest such that it can serve as a “database cursor” or as a cursor over a time series, e.g. having entries for “previous” and “next” collections.

2. File-Like ICN Collection (FLIC) Format

We first give the FLIC format in EBN notation:

```
ManifestMsg := Name? HashGroup+

HashGroup   := MetaData? (SizeDataPtr | SizeManifestPtr)+
BlockHashGroup := MetaData? SizePerPtr (DataPtr | ManifestPtr)+

DataPtr := HashValue
ManifestPtr := HashValue
SizeDataPtr := Size HashValue
SizeManifestPtr := Size HashValue

SizePerPtr    := Size
HashValue     := See {{CCNxMessages}}
Size          := OCTET[8]
HashAlgorithm := T_SHA256 | T_SHA512 | ...
HashDigest    := OCTET+

MetaData    := Property*
Property    := Locator | OverallByteCount | OverallDataDigest | ...
```

Description:

- The core of a manifest is the sequence of “hash groups”.
- A HashGroup (HG) consists of a sequence of “sized” data or manifest pointers.
- A BlockHashGroup (BHG) consists of a sequence of data or manifest pointers and a mandatory field that lists the total size of each pointer. These HashGroups should be used when each pointer (except the last) contains an identical number of application bytes.
- Sizes are 64-bit unsigned integers.
- Data and manifest pointers are cryptographic HashValues encoded according to the mechanism listed in [CCNxMessages]. Specifically, a HashValue specifies the cryptographic hash algorithm and the actual digest.
- A HashGroup can contain a metadata section to help a reader to optimize content retrieval (block size of leaf nodes, total size, overall digest etc).
- None of the ICN objects used in FLIC are allowed to be chunked, including the (sub-) manifests. The smallest possible complete manifest contains one HashGroup with one pointer to an ICN object.

2.1. Use of hash-valued pointers

FLIC's tree data structure is a generalized index table as it is known from file systems. The pointers, which in an OS typically are hard disk block numbers, are replaced by hash values of other ICN objects. These ICN objects contain either other manifest nodes, or leaf nodes. Leafs contain the actual data of the collection. Each pointer explicitly indicates the amount of application data bytes contained by the referred object. For example, the size of a data pointer (to a leaf) represents the size of the leaf's content object payload. Conversely, the size of a manifest pointer represents the total size of all pointers contained in that manifest.

FLIC makes use of "nameless ICN object" where the network is tasked with fetching an object based on its digest only. The interest for such an object consists of a routing hint (locator) plus the given digest value.

2.2. Creating a FLIC data structure

Starting from the original content, the corresponding byte array is sliced into chunks. Each chunk is encoded as a data object, according the ICN suite. For each resulting data object, the hash value is computed. Groups of consecutive objects are formed and the corresponding hash values collected in manifests, which are also encoded. The hash values of the manifest objects replace the hash values of the covered leaf nodes, thus reducing the number of hash values. This process of hash value collection and replacement is repeated until only one (root) manifest is left.

```
data1 <-- h1  - - - - - - - - - - - - \
data2 <-- h2 \                              root mfst
...           mfst 1 <-- hN+1  \              /
dataJ <-- hJ /                  mfst2 <-- hN+2
...                          /
dataN <-- hN  - - - - - - /
```

Of special interest are "skewed trees" where a pointer to a manifest may only appear as last pointer of (sub-) manifests. Such a tree becomes a sequential list of manifests with a maximum of datapointers per manifest packet. Beside the tree shape we also show this data structure in form of packet content where D stands for a data pointer and M is the hash of a manifest packet.

```
data1   <-- h1  - - - - - - - - - root mfst
...                              /
dataJ-1 <-- hJ-1                    /
dataJ   <-- hJ  - -  mfst1 <-- hN+1 /
...              /
dataN   <-- hN  - /

DDDDDDM--> DDDDDDM--> ..... DDDDDDM--> DDDDDDD
```

A pseudo code description for producing a skewed tree follows below.

```
Input:
   Application data D of size |D| (bytes)
   Block size B (in bytes)
Output:
   FLIC root node R
Algo:
   n = number of leaf nodes = ceil(|D| / B)
   k = number of (encoded) hash values fitting in a block of size B
   H[1..n] = array of hash values
     initialized with the data hash values for data chunks 1..n
   While n > k do
     a) create manifest M with a HashGroup
     b) append to the HashGroup in M all hash values H[n-k+1..n]
```

```
    c)  n = n - k + 1
    d)  H[n] = manifest hash value of M
  Create root manifest R with a HashGroup
  Add to the HashGroup of R all hash values H[1..n]
  Optionally: add name to R, sign manifest R
  Output R
```

Obtaining with each manifest a maximum of data pointers is beneficial for keeping the download pipeline filled. On the other hand, this tree doesn't support well random access to arbitrary byte positions: All data pointers coming before that offset have to be fetched before locating the block of interest. For random access, binary trees (where both subtrees of a node cover half of the content bytes) are better suited. This can be combined with the "skewed tree" approach: Manifests of intermediate nodes are filled with data pointers except for the last two slots. The second last slot points to a manifest for the "first half" of the left content, the last slots then points to a manifest for the rest.

```
root manifest=     DDDDDMM
                  _____________/  \_____
                 /                     \
                 DDDDDMM                 DDDDDMM
               _______/  \             _____/  \
              /          \           /        \
              DDDDDDD      DDDDDDD     DDDDDDD    DDDDDDD
```

This can be generalized to k-ary trees by allocating k pointers per manifest instead of 2.

2.3. Reconstructing the collection's data

To fetch the data associated with a given FLIC (sub-) manifest, the receiver sequentially works through all entries found in the HashGroups and issues corresponding hash-based interests. In case of a data hash pointer, the received content object is appended. In case of a manifest hash pointer, this procedure is called recursively for the received manifest. In other words, the collection data is represented as the concatenation of data leaves from this *pre-order* depth-first search (DFS) traversal strategy of the manifest tree. (Currently, pre-order DFS is the only supported traversal strategy.) This procedure works regardless of the tree's shape.

A pseudo code description for fetching is below.

```
Input:
   Root manifest R
Output:
   Application data D
Algo:
   global D = []
   DFS(R)
   Output D

where:

procedure DFS(M)
{
L:
  H = sequence of hash valued pointers of M
  foreach p in H do:
    if p is a data pointer then
      data = lookup(p)
```

```
    Append data to D
  else
    M = lookup(p)
    if p is last element in H then
      goto L;   // tail recursion
    DFS(M)
}
```

The above DFS code works for FLIC manifest trees of arbitrary shape. In case of a skewed tree, no recursion is needed and a single instance of the DFS procedure suffices (i.e., one uses tail recursion).

2.4. Metadata in HashGroups

In FLIC, metadata is linked to HashGroups and permits to inform the FLIC retriever about properties of the data that is covered by this hash group. Examples are overall data bytes or the overall hash digest (this is akin to a Merkle hash). The intent of such metadata is to enable an in-network retriever to optimize its operation - other attributes linked to the collection as a whole (author, copyright, etc.) is out of scope.

The list of available metadata is below.

```
* Locator - provides a new routing hint (name prefix) where the
  chunks of this hash group can be retrieved from. The default is to
  use the locator of the root manifest.

* OverallByteCount - indicates the total number of *application
  data bytes* contained in a single HashGroup. This does not include
  bytes consumed by child manifests. This value is equal to the sum of
  all pointer sizes contained in the HashGroup.

* OverallDataDigest - expresses the overall digest of all application
  data contained in the HashGroup.
```

BlockHashGroups contain a mandatory piece of metadata called the SizePerPtr. This value indicates the total number of application bytes contained within each pointer in the hash group *except for the last pointer.* Normal HashGroups do not require this piece of metadata; Instead, each pointer includes their size explicitly.

2.5. Locating FLIC leaf and manifest nodes

The optional name of a manifest is a mere decoration and has no locator functionality at all: All objects pointed to by a manifest are retrieved from the location where the manifest itself was obtained from (which is not necessarily its name). Example:

```
Objects:
  manifest(name=/a/b/c, ptr=h1, ptr=hN)  - has hash h0
  nameless(data1)                        - has hash h1
  ...
  nameless(dataN)                        - has hash hN

Query for the manifest:
  interest(name=/the/locator/hint, implicitDigest=h0)
```

In this example, the name “/a/b/c” does NOT override “/the/locator/hint” i.e., after having obtained the manifest, the retriever will issue requests for

```
  interest(name=/the/locator/hint, implicitDigest=h1)
```

```
...
interest(name=/the/locator/hint, implicitDigest=hN)
```

Using the locator metadata entry, this behavior can be changed:

```
Objects:
  manifest(name=/a/b/c,
           hashgroup(loc=/x/y/z, ptr=h1)
           hashgroup(ptr=h2)          - has hash h0
  nameless(data1)                     - has hash h1
  nameless(data2)                     - has hash h2

Queries:
  interest(name=/the/locator/hint, implicitDigest=h0)
  interest(name=/x/y/z, implicitDigest=h1)
  interest(name=/the/locator/hint, implicitDigest=h2)
```

3. Advanced uses of FLIC manifests

The FLIC mechanics has uses cases beyond keeping together a set of data objects, such as: seeking, block-level de-duplification, re-publishing under a new name, growing ICN collections, and supporting FLICs with different block sizes.

3.1. Seeking

Fast seeking (without having to sequentially fetch all content) works by skipping over entries for which we know their size. The following expression shows how to compute the byte offset of the data pointed at by pointer P_i, offset_i. In this formula, let P_i represent the Size value of the i-th pointer.

```
offset_i = \sum_{i = 1}^{i - 1} P_i.size
```

With this offset, seeking is done as follows:

```
Input: seek_pos P, a FLIC manifest with a hash group having N entries
Output: pointer index i and byte offset o, or out-of-range error
Algo:
   offset = 0
   for i in 1..N do
      if (P < P_i.size)
         return (i, P - offset)
      offset += P_i.size
   return out-of-range
```

Seeking in a BlockHashGroup is different since offsets can be quickly computed. This is because the size of each pointer P_i except the last is equal to the SizePerPtr value. For a BlockHashGroup with N pointers, OverallByteCount D, and SizePerPointer L, the size of P_i is equal to the following:

```
D - ((i - 1) * L)
```

In a BlockHashGroup with k pointers, the size of P_k is equal to:

```
D - L * (k - 1)
```

Using these, the seeking algorithm can be thus simplified to the following:

```
Input: seek_pos P, a FLIC manifest with a hash group having
       OverallByteCount S and SizePerPointer L.
Output: pointer index i and byte offset o, or out-of-range error
Algo:
   if (P > S)
       return out-of-range
   i = floor(P / L)
   if (i > N)
       return out-of-range # bad FLIC encoding
   o = P mod L
   return (i, o)
```

Note: In both cases, if the pointer at position i is a manifest pointer, this algorithm has to be called once more, seeking to seek_pos o inside that manifest.

3.2. Block-level de-duplification

Consider a huge file, e.g. an ISO image of a DVD or program in binary form, that had previously been FLIC-ed but now needs to be patched. In this case, all existing encoded ICN chunks can remain in the repository while only the chunks for the patch itself is added to a new manifest data structure, as is shown in the picture below. For example, the venti archival file system of Plan9 uses this technique.

```
old_mfst -  - > h1 --> oldData1  <-- h1 < -  -  new_mfst
        \  - > h2 --> oldData2  <-- h2 < -  - /
         \           replace3  <-- h5 < -  -/
          \- > h3 --> oldData3             /
           \ > h4 --> oldData4  <-- h4 < - /
```

3.3. Growing ICN collections

A log file, for example, grows over time. Instead of having to re-FLIC the grown file it suffices to construct a new manifest with a manifest pointer to the old root manifest plus the sequence of data hash pointers for the new data (or additional sub-manifests if necessary). Note that this tree will not be skewed (anymore).

```
old data < -  -  -  mfst_old <-- h_old -  - mfst_new
                                        /
new data1 <-- h_1 -  -  -  -  -  -  -  -  -/
new data2                                /
...                                    /
new dataN <-- h_N -  -  -  -  -  -  -  -/
```

3.4. Re-publishing a FLIC under a new name

It can happen that a publisher's namespace is part of a service provider's prefix. When switching provider, the publisher may want to republish the old data under a new name. This can easily be achieved with a single nameless root manifest for the large FLIC plus arbitrarily many per-name manifests (which are signed by whomever wants to publish this data):

```
  data < - nameless_mfst() <-- h  < - mfst(/com/parc/east/the/flic)
                               < - mfst(/com/parc/west/old/the/flic)
                               < - mfst(/internet/archive/flic234)
```

Note that the hash computation (of h) only requires reading the nameless root manifest, not the entire FLIC.

This example points out the problem of HashGroups having locator metadata elements: A retriever would be urged to follow these hints which are “hardcoded” deep inside the FLIC but might have become outdated. We therefore recommend to name FLIC manifests only at the highest level (where these names have no locator function). Child nodes in a FLIC manifest should not be named as these names serve no purpose except retrieving a sub-tree’s manifest by name, if would be required.

3.5. Data Chunks of variable size

If chunks do not have regular (block) sizes, the HashGroup can be used to still convey to a reader the length of the chunks at the manifest level. (This can be computed based on the size of pointers, but the metadata field makes this determination simpler.) Example use cases would be chunks each carrying a single ASCII line as entered by a user or a database with variable length records mapped to chunks.

```
M = (manifest
      (hashgroup((metadata(SizePerPtr=4096)) (dataptr=h1))
      (hashgroup((metadata(SizePerPtr=1500)) (dataptr=h2))
      ...
    )
```

4. Encoding

We express the packet encoding of manifests in a symbolic expression style in order to show the TLV structure and the chosen type values. In this notation, a TLV’s type is a combination of “SymbolicName/Tvalue”, Length is not shown and Values are sub-expressions. Moreover, we populate the data structure with all possible entries and omit repetition.

4.1. Example Encoding for CCNx1.0

```
[FIXED_HEADER OCTET[8]]
(ManifestMsg/T_MANIFEST
  (Name/T_NAME ...)
  (HashGroup/T_HASHGROUP
     (MetaData/T_HASHGROUP_METADATA
        (HGLocator/T_HASHGROUP_METADATA_LOCATOR (T_NAME ...))
        (HGOverallByteCount/T_HASHGROUP_METADATA_BYTECOUNT INT)
        (HGOverallDataDigest/T_HASHGROUP_METADATA_DATADIGEST OCTET[32])
     )
     (SizeDataPtr/T_HASHGROUP_SIZEDATAPTR OCTET[8] (T_HASH ...))
     (SizeMfstPtr/T_HASHGROUP_SIZEMANIFESTPTR OCTET[8] (T_HASH ...))
  )
  (BlockHashGroup/T_BLOCKHASHGROUP
     (MetaData/T_HASHGROUP_METADATA (...))
     (DataPtr/T_HASHGROUP_DATAPTR OCTET[32] (T_HASH ...))
     (MfstPtr/T_HASHGROUP_MANIFESTPTR OCTET[32] (T_HASH ...))
  )
)
```

Interest: name is locator, use objHashRestriction as selector.

4.2. Example Encoding for NDN

The assigned NDN content type value for FLIC manifests is 1024 (0x400).

```
(Data/0x6
```

```
 (Name/0x7 ...)
 (MetaInfo/0x14
   (ContentType/0x18 0x0400)
 )
 (Content/0x15
   (HashGroup/0xC0
     (MetaInfo/0x14
       (LocatorNm/0xC3 (NameComp/0x8 ...))
       (OverallDataDigest/0xC4 OCTET[32])
       (OverallByteCount/0xC5 INT)
     )
     (DataPtr/0xC1 OCTET[8] OCTET[32])
     (MfstPtr/0xC2 OCTET[8] OCTET[32])
     (SizeDataPtr/0xC3 OCTET[32])
     (SizeMfstPtr/0xC4 OCTET[32])
   )
 )
 (SignatureInfo/0x16 ...)
 (SignatureValue/0x17 ...)
)
```

Interest: name is locator, use implicitDigest name component as selector.

5. Security Considerations

None.

6. Normative References

[CCNxMessages] PARC, ., PARC, . and . PARC, "CCNx Messages in TLV Format", n.d..

Authors' Addresses

Christian Tschudin
University of Basel
EMail: christian.tschudin@unibas.ch

Christopher A. Wood
PARC, Inc.
EMail: christopher.wood@parc.com