ICNRG Working Group	C. Tschudin
Internet-Draft	University of Basel
Intended status: Informational	C. Wood
Expires: October 6, 2016	PARC, Inc.
	April 04, 2016

File-Like ICN Collection (FLIC)

draft-tschudin-icnrg-flic-00

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

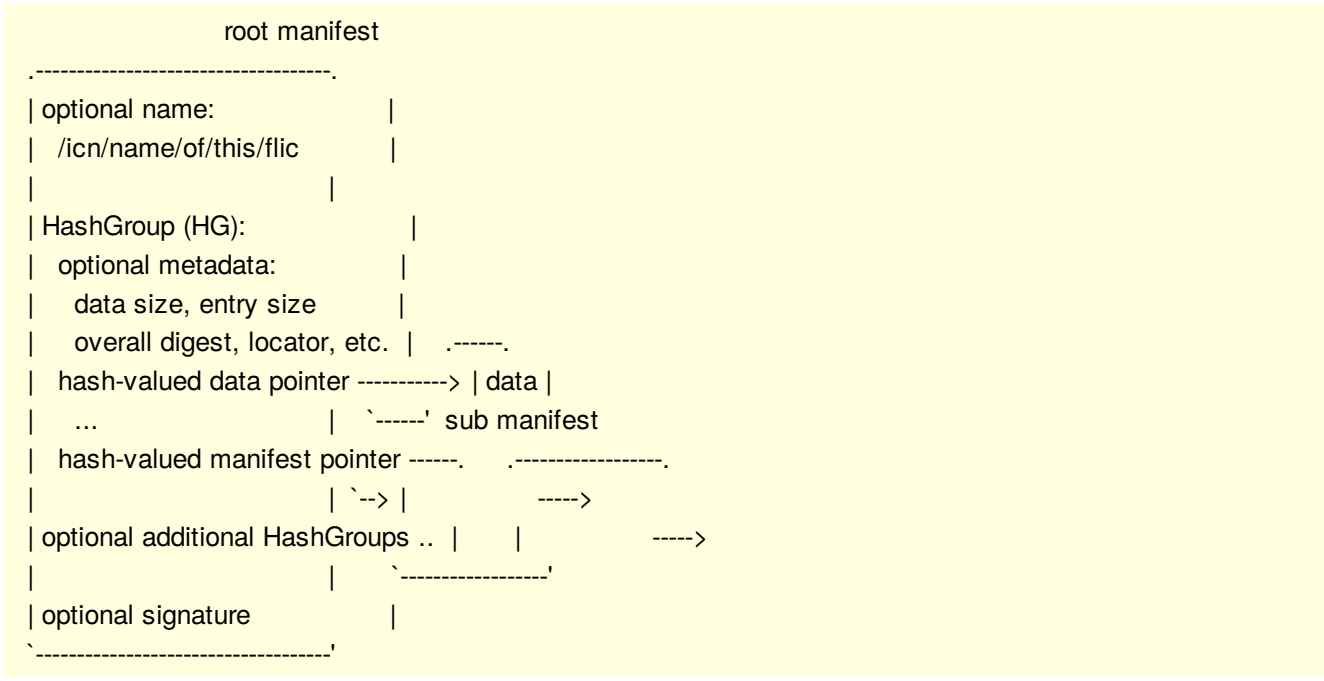


Figure 1: A FLIC manifest and its directed acyclic graph

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? (DataPointer | ManifestPtr)+
DataPointer := HashValue
ManifestPtr := HashValue
HashValue   := OCTET[32]

MetaData    := Property*
Property    := Locator | DataSize | EntrySize | BlockSize |
               DataDigest | TreeDepth | ...
```

Description:

- The core of a manifest is the sequence of “hash groups”.
- A HashGroup (HG) consists of a sequence of data or manifest pointers.
- Data as well as manifest pointers are SHA256 digests (32 Bytes); their encoding assigns them distinct types.
- 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 harddisk 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.

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 (of equal size if blocksize is present in the metadata section, except for the last chunk). 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.

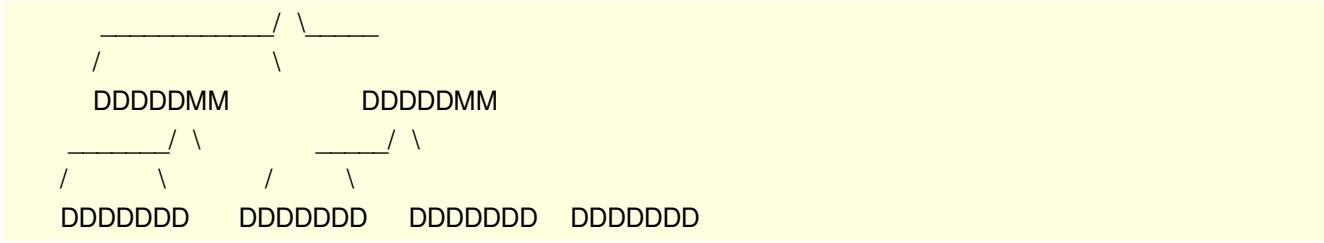
```
data1   <-- h1  - - - - - - - - - root mfst
...                              /
dataJ-1 <-- hJ-1                    /
dataJ   <-- hJ  - -  mfst1 <-- hN+1 /
...              /
dataN   <-- hN  - /

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
```



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 size per entry. 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.

* DataSize - indicates the total number of *application data bytes*
  contained in a single HashGroup. This does not include bytes consumed
  by child manifests.

* EntrySize - indicates the number of *application data bytes* contained
  by a single entry (pointer) in a HashGroup.

* BlockSize - indicates the size of each data and manifest node used
  when producing each entry of the HashGroup.

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

* Tree Depth - expresses the depth of each entry in the HashGroup and
  allows a receiver to predict the amount of memory needed when
  traversing this tree.
```

To give an example of how the DataSize and EntrySize values are used, consider the following example. Let HG be a HashGroup with n entries, DataSize S, and EntrySize E. It is required that the *first (n - 1) entries of HG* "contain" E bytes of application data and that the last (nth) entry has at most E bytes of application data. Thus, the following inequality always holds:

```
 n * E <= S
```

We will give an example of how to use these sizes for seeking in Section 3.1.

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 byte offset of the data pointed at by pointer P_i (in a HashGroup with EntrySize E and DataSize S) is computed as follows:

```
offset = (i * E)
```

Recall that the total size of each pointer (except the last) is equal to the EntrySize. If P_n is the last pointer in a HashGroup, its size is calculated as

```
size = S - ((n - 1) * E)
```

With these formulas, seeking is done as follows:

```
Input: seek_pos P, a FLIC manifest with a HashGroup having
       EntrySize E, DataSize S and N entries
Output: blockindex i and offset o, or out-of-range error
Algo:
  if (P >= S)
    return out-of-range
  i = floor(P / E)
  if (i >= N)
    return out-of-range # bad FLIC encoding
  o = S - (i * E)
  return (i, o)
```

Note: 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, and therefore manifests do not have consistent entry sizes, the HashGroup can be used to still convey to a reader the length of the chunks at the manifest level. 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(blocksize=12)) (dataptr=h1))
     (hashgroup((metadata(blocksize=1)) (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. An abbreviated example for the NDN Interest packet would be:

```
(Interest/0x5
  (Name/0x7 (NameComp=0x8 ...) ...)
  (Selector/0x9 ...)
  (Nonce/0xA BLOB)
  (Scope/0xB INT)
  (InterestLifeTime/0xC INT)
)
```

4.1. Encoding for CCNx1.0

```
[FIXED_HEADER OCTET[8]]
(ManifestMsg/0x6
  (Name/0x0 ...)
  (HashGroup/0x1
     (MetaData/0x1
        (HGLocator/0x0 (NameComp/0x ...))
        (HGDataSize/0x2 INT)
        (HGEntrySize/0x3 INT)
        (HGBlockSize/0x4 INT)
        (HGDataDigest/0x5 OCTET[32])
        (HGTreeDepth/0x6 INT)
     )
     (DataPtr/0x2 OCTET[32])
     (MfstPtr/0x3 OCTET[32])
  )
)
```

Interest: name is locator, use objHashRestriction as selector.

4.2. 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 ...))
        (TotalHash/0xC4 OCTET[32])
        (TotalSize/0xC5 INT)
        (BlockSize/0xC6 INT)
        (TreeDepth/0xC7 INT)
```

```
      )
      (DataPtr/0xC1 OCTET[32])
      (MfstPtr/0xC2 OCTET[32])
    )
  )
  (SignatureInfo/0x16 ...)
  (SignatureValue/0x17 ...)
)
```

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

Authors' Addresses

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

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