Workgroup: ICNRG
Internet-Draft: draft-irtf-icnrg-flic-04
Published: 22 October 2023
Intended Status: Experimental
Expires: 24 April 2024
Authors: C. Tschudin
University of Basel
C.A. Wood
Cloudflare
M.E. Mosko
PARC, Inc.
D. Oran, Ed.
Network Systems Research & Design

File-Like ICN Collections (FLIC)

Abstract

This document describes a simple "index table" data structure and its associated Information Centric Networking (ICN) data objects for organizing a set of primitive ICN data objects into a large, File-Like ICN Collection (FLIC). At the core of this collection is a *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

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 24 April 2024.

Copyright Notice

carefully, as they describe your rights and restrictions with respect to this document.

Table of Contents

1. Introduction

ICN architectures, such as Content-Centric Networking (CCNx) [RFC8569] and Named Data Networking [NDN], are well suited for static content distribution. Each piece of (possibly immutable) static content is assigned a name by its producer. Consumers fetch this content using said name. Optionally, consumers may specify the full name of content, which includes its name and a unique (with overwhelming probability) cryptographic digest of said content.

Note: The reader is assumed to be familiar with general ICN concepts from CCNx or NDN. For general ICN terms, this document uses the terminology defined in [RFC7927]. Where more specificity is needed, we utilize CCNx [RFC8569] terminology where a Content Object is the data structure that holds application payload. Terms defined specifically for FLIC are enumerated below in Section 3.1.

To enable requests with full names, consumers need a priori knowledge of content digests. A Manifest, a form of catalog, is a data structures commonly employed to store and transport this information. Typically, ICN manifests are signed content objects (data) which carry a collection of hash digests. As content objects, a manifest itself may be fetched by full name. A manifest may contain either hash digests of, or pointers to, either other manifests or content objects. A collection of manifests and content objects represents a large piece of application data, e.g., one that cannot otherwise fit in a single content object. Because a manifest contains a collection of hashes, it is by definition non-circular because one cannot hash the manifest before filling it in.

Structurally, this relationship between manifests and content objects is reminiscent of the UNIX inode concept with index tables and memory pointers. In this document, we specify a simple, yet extensible, manifest data structure called FLIC - *File-Like ICN Collection*. FLIC is suitable for ICN protocol suites such as CCNx and NDN. We describe the FLIC design, grammar, and various use cases, e.g., ordered fetch, seeking, de-duplication, extension, and variable-sized encoding. We also include FLIC encoding examples for CCNx and NDN.

The purpose of a manifest is to concisely name, and hence point to, the constiuent pieces of a larger object. A FLIC manifest does this

by using a *root* manifest to name and cryptographically sign the data structure and then use concise lists of hash-based names to indicate the constituent pieces. This maintains strong security from a single signature. A Manifest entry gives one enough information to create an *Interest* for that entry, so it must specify the name, the hash digest, and if needed, the locators.

FLIC is a distributed data structure illustrated by the following picture.

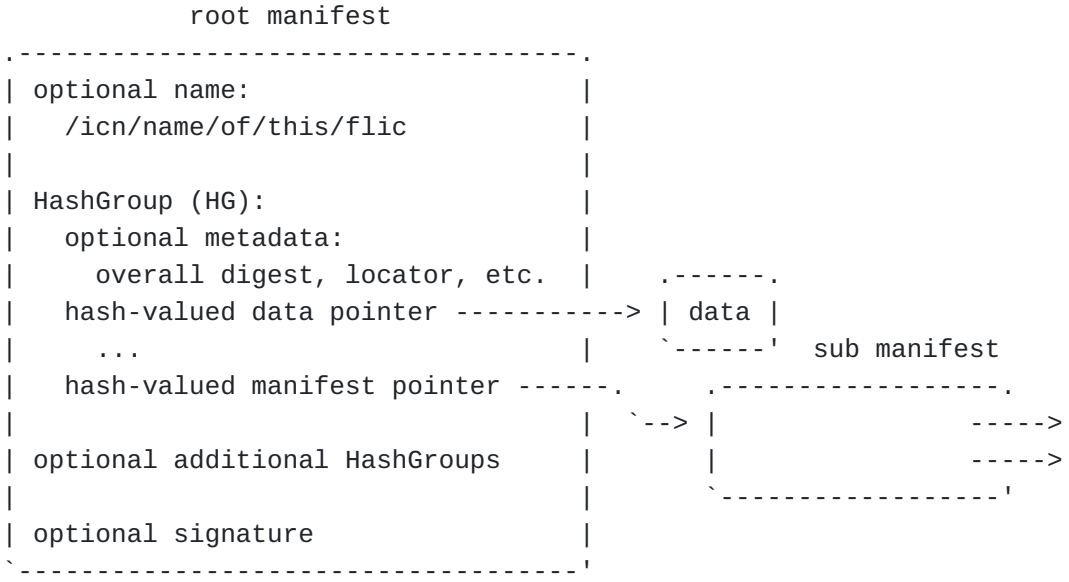


Figure 1: A FLIC manifest and its directed acyclic graph

A key design decision is how one names the root manifest, the application data, and subsidiary manifests. FLIC uses the concept of a Name Constructor. The root manifest (in fact, any FLIC manifest) may include a Name Constructor that instructs a manifest reader how to properly create Interests for the associated application data and subsidiary manifests. The Name Constructors allow interest construction using a well-known, application-independent set of rules. Some name constructor forms are tailored towards specific ICN protocols, such as CCNx or NDN; some are more general and could work with many protocols. We describe the allowed Name Constructor methods in Section 3.3. There are also particulars of how to encode the name schema in a given ICN protocol, which we describe in Section 3.9.

FLIC has encodings for CCNx (Section 3.9.1) as per RFC 8609 [RFC8609] and for NDN (Section 3.9.2).

An example implementation in Python may be found at [FLICImplementation].

1.1. FLIC as an ICN experimental tool

FLIC enables experimentation with how to structure and retrieve large data objects and collections in ICN. By having a common data structure applications can rely on, with a common library of code that can be used to create and parse manifest data structures, applications using ICN protocols can both avoid unnecessary reinvention and also have enhanced interoperability. Since the design attempts to balance simplicity, universality, and extensibility, there are a number of important experimental goals to achieve that may wind up in conflict with one another. We provide a partial list of these experimental issues in Section 4. It is also important for users of FLIC to understand that some flexibility and extensions might be removed if use cases do not materialize to justify their inclusion in an eventual standard.

1.2. 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 [RFC2119].

2. Design Overview

The FLIC design adopts the proven UNIX inode concept of direct and indirect pointers, but without the specific structural forms of direct versus indirect. FLIC is a collection of pointers, and when one de-references the pointer it could be an application object or another FLIC manifest. The pointers in FLIC use hash-based naming of Content Objects analogous to the function block numbers play in UNIX inodes.

Because FLIC uses hash-based pointers as names, FLIC graphs are inherently acyclic. Both CCNx and NDN support hash-based naming, though the details differ (see Section 3.9.1 and Section 3.9.2).

The FLIC datastructure is an acyclic digraph of Content Objects. In this document, our examples are trees, but that is not a requirement. For example, a de-duplication representation might have a common object with many 0s and that object could be references from multiple places in the tree. As another example, there could be a common sub-collection of objects organized in a Manifest, and that sub-manifest could be included in multiple places.

In FLIC terms, a direct pointer links to application-level data, which is a Content Object with application data in the Payload. An indirect pointer links to a Content Object with a FLIC Manifest in the Payload.

Note: A substantial advantage of using hash-based naming is that it permits block-level de-duplication of application data because two blocks with the same payload will have the same hash name.

The FLIC structure that is expected most applications would use consists of a root manifest with a strong cryptographic signature and then cryptographically strong (e.g. SHA256 [SHS]) hash names as pointers to other manifests. The advantage of this structure is that the single signature in the root manifest covers the entire data structure no matter how many additional manifests are in the data structure. Another advantage of this structure is it removes the need to use chunk (CCNx) or segment (NDN) name components for the subordinate manifests.

Another usage is to have a signed Root Manifest with a single pointer to the Top Manifest. The Top Manifest maybe a CCNx Nameless object. This method allows an intermediary service to respond to client requests with its own signed Manifest that then points to a small Root manifest. The client trusts the intermediary's reponse because of the intermediary's signature, and then trusts the content because of the Root manifest. In some cases, the intermediary could embed the Root Manifest (because it is small) and avoid additional round trips before beginning download. This technique is used in a peer-to-peer sharing protocol [ProjectOrigin].

FLIC supports manifest encryption separate from application payload encryption (See Section 3.8). It has a flexible encryption envelope to support various encryption algorithms and key discovery mechanisms. The byte layout allows for in-place encryption and decryption.

A limitation of this approach is that one cannot construct a hash-based name for a child until one knows the payload of that child. In practical terms, this means that one must have the complete application payload available at the time of manifest creation.

FLIC's design allows straightforward applications that just need to traverse a linear set of related objects to do so simply, but FLIC has two extensibility mechanisms that allow for more sophisticated uses: manifest metadata, and pointer annotations. These are described in Section 3.4 and Section 3.5 respectively.

FLIC goes to considerable lengths to allow creation and parsing by application-independent library code. Therefore, any options used by applications in the data structure or encryption capabilities MUST NOT require applications to have application-specific Manifest traversal algorithms. This ensures that such application agnostic libraries can always successfully parse and traverse any FLIC Manifest by ignoring the optional capabilities.

The reader may find it useful to refer to Section Example Usages (Section 5) from time to time to see worked out examples.

3. FLIC Structure

3.1. Terminology

Data Object: a CCNx nameless Content Object that usually only has Payload. It might also have an ExpiryTime to limit the lifetime of the data.

Direct Pointer: borrowed from inode terminology, it is a CCNx link using a content object hash restriction and a locator name to point to a Data Object.

Hash Group: KA collection of pointers. A Manifest should have atleast one Hash Group. A Hash Group may have its own associated meta data and Name Constructor.

Indirect Pointer: borrowed from inode terminology, it is a CCNx link using a content object hash restriction and a locator name to point to a manifest content object.

Internal Manifest: some or all pointers are indirect. The order and number of each is up to the manifest builder. By convention, all the direct manifests come first, then the indirect.

Leaf Manifest: all pointers are direct pointers.

Locator: A routing hint in an Interest used by forwarding to get the Interest to where it can be matched based on its Name Constructor-derived name.

Manifest: a CCNx ContentObject with PayloadType 'Manifest' and a Payload of the encoded manifest. A leaf manifest only has direct pointers. An internal manifest has a mixture of direct and indirect pointers.

Manifest Waste: a metric used to measure the amount of waste in a manifest tree. Waste is the number of unused pointers. For example, a leaf manifest might be able to hold 40 direct pointers, but only 30 of them are used, so the waste of this node is 10. Manifest tree waste is the sum of waste over all manifests in a tree.

Name Constructor: The specification of how to construct an Interest for a Manifest entry.

Root Manifest: A signed, named, manifest that points to nameless manifest nodes. This structure means that the internal tree

structure of internal and leaf manifests have no names and thus may be located anywhere in a namespace, while the root manifest has a name to fetch it by.

Top Manifest: One useful manifest structure is to use a Root manifest that points to a single Internal manifest called the Top Manifest. The Top manifest the begins the structure used to organize manifests. It is also possible to elide the two and use only a root manifest that also serves in the role of the top manifest.

3.2. Locators

Locators are routing hints used by forwarders to get an Interest to a node in the network that can resolve the Interest's name. In some naming conventions, the name might only be a hash-based name so the Locator is the only available routing information. Locators exist in both CCNx and NDN, though the specific protocol mechanisms differ. A FLIC manifest represents locators in the same way for both ICN protocols inside Name Constructors (Section 3.3), though they are encoded differently in the underlying protocol. See Section 3.9 for encoding differences.

A manifest Node may define one or more Locator prefixes that can be used in the construction of Interests from the pointers in the manifest. The Locators are inherited when walking a manifest tree, so they do not need to be defined everywhere. It is RECOMMENDED that only the Root manifest contain Locators so that a single operation can update the locators. One use case when storing application payloads at different replicas is to replace the Root manifest with a new one that contains locators for the current replicas.

3.3. Name Constructors

A Manifest may define zero or more name constructors in NameConstructorDefinitions (NCD) located in the Manifest Node. An NCD associates a Name Constructor Id (NCID) to a Name Constructor. The NCID is used in other parts of the Manifest to refer to that specific definition.

A manifest organizes pointers inside Hash Groups. Each Hash Group uses an NCID to indicate what Name Constructor to use to fetch the pointers inside the group.

NCID 0 is the default name constructor. If it is not defined in an NCD, it is assumed to be a HashNamingConstructor. A Manifest may re-define the default as needed.

A Manifest MUST use locally unique NCIDs in the NCD.

NCDs and their associated NCIDs are inherited as one traverses a manifest. That is, a manifest consumer must remember the NCDs as it traverses manifests. If it encounters a HashGroup that uses an unknown NCID, the RECOMMENDED action is to report a malformed manifest to the user.

A Manifest may update an NCID. If a child manifest re-defines an NCID, the manifest consumer MUST use the new definition from that point forward under that Manifest branch.

It is RECOMMENDED that only the root or similar top-level manifest define NCDs and they not be re-defined in subsequent manifests.

We expect that an application constructing a Manifest will take one of three approaches to name constructors. The advantage of using, or re-defining, the default name constructor is that any hash groups that use it do not need to specify an NCID and thus might save some space.

* A manifest might define (or use) a default name constructor and mix subsequent Manifest and Data objects under that same namespace. The manifest only needs to use one Hash Group and can freely mix Manifest and Data pointers.

* A manifest might define (or use) a default name constructor for subsequent Manifests and define a second NCD for the application data. This places all subsequent manifests under the default constructor and places all application data under the second NCD. The Manifest must use at least two Hash Groups.

  There are a few options on how to organize the Hash Groups:

  (1) Manifest Hash Group followed by Data Hash group,

  (2) Data Hash Group followed by Manifest Hash Group,

  (3) Intermix multiple manifest and data hash groups for interleaved reading, or

  (4) use a data-on-leaf only approach: the interior manifests would use the manifest hash group and the leaves would use the data hash group. Other organizations are possible.

* Define multiple NCDs for subsequent manifests and data, or not use the default NCD, or use some other organization.

In this specification, we define the following four types of Name Constructors. Additional name constructor types may be specified in a subsequent revision of the specification. Here, we informally

define the name constructors. Section 3.6 specifies the encoding of each name constructor.

Type 0 (Interest-Derived Naming): Use whatever name was used in the Interest to retrieve this Manifest, less a hash component, and append the desired hash value.

Type 1 (Data-Derived Naming): Use the Manifest Name, less a hash component, as the Interest name, and append the desired hash value.

Type 2 (Prefix List): The NCD specifies a list of 1 or more name prefixes. The consumer may use any (or all) of those prefixes with the desired hash appended.

Type 3 (Segmented Naming): As in Type 2, but the consumer MUST track Segment Numbers. If the Hash Group provides Segment Number annotations for each pointer, it MUST use those numbers. Otherwise, the consumer MUST use a 0-based counter that follows the traversal order.

In Type 0, the consumer uses some name N to fetch a manifest. When the consumer receives the Manifest back, it begins issuing interests for the content using the same name N, but with the hash pointers from the manifest.

In Type 1, the consumer uses some name N to fetch a manifest. The consumer receives a manifest back with name M inside the Manifest Content Object. The consumer then uses the name M plus hash pointers from the manifest.

In Type 2, the consumer receives a manifest and begins traversing it. If it visits a Hash Group with a PrefixSchema Name Constructor, then that Name Constructor provides a list of 1 or more locators to use. The consumer may use any or all of the provided locators, plus the hash pointer, to fetch the contents.

In Type 3, if a Hash Group has a SegmentedSchema Name Constructor, then the consumer uses the same mechanism as Type 2, but with the addition of a Segment Number in the name. Segmented naming is only compatible with deterministic traversal orders or if the Manifest provides Segment Number annotations for each pointer. If the Hash Group provides hints about other traversal orders, then it must also provide Segment Number annotations for each prefix.

3.4. Manifest Metadata

The FLIC Manifest may be extended by defining TLVs that apply to the Manifest as a whole, or alternatively, individually to every data object pointed to by the Manifest. This basic specification does not

specify any, but metadata TLVs may be defined through additional RFCs or via Vendor TLVs. FLIC uses a Vendor TLV structure identical to [RFC8609] for vendor-specific annotations that require no standardization process.

For example, some applications may find it useful to allow specialized consumers such as *repositories* (for example [repository]) or enhanced forwarder caches to pre-place, or adaptively pre-fetch data in order to improve robustness and/or retrieval latency. Metadata can supply hints to such entities about what subset of the compound object to fetch and in what order.

Note: FLICs ability to use separate namespaces for the Manifest and the underlying Data allows different encryption keys to be used, hence giving a network element like a cache or repository access to the Manifest data does not as a side effect reveal the contents of the application data itself.

3.5. Pointer Annotations

FLIC allows each manifest pointer to be annotated with extra data. Annotations allow applications to exploit metadata about each Data Object pointed to without having to first fetch the corresponding Content Object. This specification defines one such annotation. The *SizeAnnotation* specifies the number of application layer octets covered by the pointer.

An annotation may, for example, give hints about a desirable traversal order for fetching the data, or an importance/precedence indication to aid applications that do not require every content object pointed to in the manifest to be fetched. This can be very useful for real-time or streaming media applications that can perform error concealment when rendering the media.

Additional annotations may be defined through additional RFCs or via Vendor TLVs. FLIC uses a Vendor TLV structure identical to [RFC8609] for vendor-specific annotations that require no standardization process.

3.6. Manifest Grammar (ABNF)

The manifest grammar is mostly, but not entirely independent of the ICN protocol used to encode and transport it. The TLV encoding therefore follows the corresponding ICN protocol, so for CCNx FLIC uses 2 octet length, 2 octet type and for NDN uses the 1/3/5 octet types and lengths (see [NDNTLV] for details). There are also some differences in how one structures and resolves links. [RFC8569] defines HashValue and Link for CCNx encodings. The NDN ImplicitSha256DigestComponent defines HashValue and NDN Delegation

(from Link Object) defines Link for NDN. Section 3.9 below specifies these differences.

The basic structure of a FLIC manifest comprises a security context, a node, and an authentication tag. The security context and authentication tag are not needed if the node is unencrypted. A node is made up of a set of metadata, the NodeData, that applies to the entire node, and one or more HashGroups that contain pointers.

The NodeData element defines the namespaces used by the manifest. There may be multiple namespaces, depending on how one names subsequent manifests or data objects. Each HashGroup may reference a single namespace to control how one forms Interests from the HashGroup. If one is using separate namespaces for manifests and application data, one needs at least two hash groups. For a manifest structure of "MMMDDD," (where M means manifest (indirect pointer) and D means data (direct pointer)) for example, one would have a first HashGroup for the child manifests with its namespace and a second HashGroup for the data pointers with the other namespace. If one used a structure like "MMMDDDMMM," then one would need three hash groups.

```
TYPE = 2OCTET / {1,3,5}OCTET ; As per CCNx or NDN TLV
LENGTH = 2OCTET / {1,3,5}OCTET ; As per CCNx or NDN TLV

Manifest = TYPE LENGTH [SecurityCtx] (EncryptedNode / Node) [AuthTag]

SecurityCtx = TYPE LENGTH AlgorithmCtx
AlgorithmCtx = AEADCtx / RsaKemCtx
AuthTag = TYPE LENGTH *OCTET ; e.g. AEAD authentication tag
EncryptedNode = TYPE LENGTH *OCTET ; Encrypted Node

Node = TYPE LENGTH [NodeData] 1*HashGroup
NodeData = TYPE LENGTH [SubtreeSize] [SubtreeDigest] [Locators]
           0*Vendor 0*NcDef
SubtreeSize = TYPE LENGTH INTEGER
SubtreeDigest = TYPE LENGTH HashValue

NcDef = TYPE LENGTH NcId NcSchema
NcId = TYPE LENGTH INTEGER
NcSchema = InterestDerivedSchema / DataDerivedSchema /
           PrefixSchema / SegmentedSchema
InterestDerivedSchema = TYPE LENGTH [ProtocolFlags]
DataDerivedSchema = TYPE LENGTH [ProtocolFlags]
PrefixSchema = TYPE LENGTH Locators [ProtocolFlags]
SegmentedSchema = TYPE LENGTH Locators [ProtocolFlags]

Locators = TYPE LENGTH 1*Link
HashValue = TYPE LENGTH *OCTET ; As per ICN Protocol
Link = TYPE LENGTH *OCTET ; As per ICN protocol

ProtocolFlags = TYPE LENGTH *OCTET
                ; ICN-specific flags, e.g. must be fresh

HashGroup = TYPE LENGTH [GroupData] (Ptrs / AnnotatedPtrs)
Ptrs = TYPE LENGTH *HashValue
AnnotatedPtrs = TYPE LENGTH *PointerBlock
PointerBlock = TYPE LENGTH *Annotation Ptr
Ptr = TYPE LENGTH HashValue

Annotation = SizeAnnotation / SegmentIdAnnotation / Vendor
SizeAnnotation = TYPE LENGTH Integer
SegmentIdAnnotation = TYPE LENGTH Integer
Vendor = TYPE LENGTH PEN *OCTET

GroupData = TYPE LENGTH [NcId] [LeafSize] [LeafDigest]
            [SubtreeSize] [SubtreeDigest] [StartSegmentId]
LeafSize = TYPE LENGTH INTEGER
LeafDigest = TYPE LENGTH HashValue
StartSegmentId = TYPE LENGTH Integer

AEADCtx = TYPE LENGTH AEADData
```

```
AEADData = KeyNum AEADNonce Mode
KeyNum = TYPE LENGTH INTEGER
AEADNonce = TYPE LENGTH 1*OCTET
AEADMode = TYPE LENGTH (AEAD_AES_128_GCM / AEAD_AES_256_GCM /
           AEAD_AES_128_CCM / AEAD_AES_128_CCM)

RsaKemCtx = 2 LENGTH RsaKemData
RsaKemData = KeyId AEADNonce AEADMode WrappedKey LocatorPrefix
KeyId = TYPE LENGTH HashValue; ID of Key Encryption Key
WrappedKey = TYPE LENGTH 1*OCTET
LocatorPrefix = TYPE LENGTH Link
```

Figure 2: FLIC Grammar

SecurityCtx: information about how to decrypt an EncryptedNode. The structure will depend on the specific encryption algorithm.

AlgorithmId: The ID of the encryption method (e.g. preshared key, a broadcast encryption scheme, etc.)

AlgorithmData: The context for the encryption algorithm.

EncryptedNode: An opaque octet string with an optional authentication tag (i.e. for AEAD authentication tag)

Node: A plain-text manifest node. The structure allows for in-place encryption/decryption.

NodeData: the metadata about the Manifest node

SubtreeSize: The size of all application data at and below the Node or Group

SubtreeDigest: The cryptographic digest of all application data at and below the Node or Group

Locators: An array of routing hints to find the manifest components

HashGroup: A set of child pointers and associated metadata

Ptrs: A list of one or more Hash Values

GroupData: Metadata that applies to a HashGroup

LeafSize: Size of all application data immediately under the Group (i.e. via direct pointers)

LeafDigest: Digest of all application data immediately under the Group

StartSegmentId:
If using

Ptr: The ContentObjectHash of a child, which may be a data ContentObject (i.e. with Payload) or another Manifest Node.

3.7. Manifest Trees

3.7.1. Traversal

FLIC manifests use a pre-order traversal. This means they are read top to bottom, left to right. The algorithms in Figure 3 show the pre-order forward traversal code and the reverse-order traversal code, which we use below to construct such a tree. This document does not mandate how to build trees. Appendix A provides a detailed example of building inode-like trees.

If using Annotated Pointers, an annotation could influence the traversal order.

```
 preorder(node)
    if (node = null)
       return
    visit(node)
    for (i = 0, i &lt; node.child.length, i++)
       preorder(node.child[i])

reverse_preorder(node)
   if (node = null)
       return
   for (i = node.child.length - 1, i &gt;= 0, i-- )
       reverse_preorder(node.child[i]) visit(node)
```

Figure 3: Traversal Pseudocode

In terms of the FLIC grammar, one expands a node into its hash groups, visiting each hash group in order. In each hash group, one follows each pointer in order. Figure 4 shows how hash groups inside a manifest expand like virtual children in the tree. The in-order traversal is M0, HG1, M1, HG3, D0, D1, D2, HG2, D3, D4.

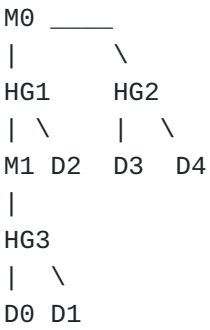


Figure 4: Node Expansion

Using the example manifest tree shown in Figure 6, the in-order traversal would be: Root, M0, M1, D0, D1, D2, M2, D3, D4, D5, M3, D6, D7, D8.

3.8. Manifest Encryption Modes

This document specifies two encryption modes. The first is a preshared key mode, where the parties are assumed to have the decryption keys already. It uses AES-GCM or AES-CCM. This is useful, for example, when using a key agreement protocol such as CCNxKE [I-D.wood-icnrg-ccnxkeyexchange]. The second is an RSA key encapsulation mode (RsaKem [RFC5990]), which may be used for group keying.

Additional modes may be defined in subsequent specifications. We expect that an RSA KemDem mode and Elliptic Curve mode should be specified.

All encryption modes use standard encryption algorithms and specifications. Where appropriate, we adopt the TLS 1.2 standards for how to use the encryption algorithms. This section specifies how to encode algorithm parameters or ICN-specific data.

For group key based encryption, we use RsaKem. This specification only details the pertinent aspects of the encryption. It describes how a consumer locates the appropriate keys in the ICN namespace. It does not specify aspects of a key manager which may or may not be used as part of key distribution and management, nor does it specify the protocol between a key manager and a publisher. In its simpliest form, the publisher could be the key manager, in which case there is no extra protocol needed between the publisher and key manager.

While the preshared key algorithm is limited in use, the AES encryption mode described applies to the group key mechanisms too. The group key mechanism facilitates the distribution of the shared key without an on-line key agreement protocol like (the expired draft) CCNxKE [I-D.wood-icnrg-ccnxkeyexchange].

3.8.1. AEAD Mode

This mechanism uses AES-GEM [AESGCM] or AES-CCM [RFC3310] for manifest encryption. A publisher creating a SecurityCtx SHOULD use the mechanisms in [RFC6655] for AES-CCM Nonce generation and [RFC5288] for AES-GCM Nonce generation.

As these references specify, it is essential that the publisher creating a Manifest never use a Nonce more than once for the same key. For keys exchanged via a session protocol, such as CCNx, the publisher MUST use unique nonces on each Manifest for that session. If the key is derived via a group key mechanism, the publisher MUST ensure that the same Nonce is not used more than once for the same Content Encryption Key.

The AEAD Mode uses [RFC5116] defined symbols AEAD_AES_128_CCM, AEAD_AES_128_GCM, AEAD_AES_256_CCM and AEAH_AES_256_GCM to specify the key length and algorithm.

The KeyNum identifies a key on the receiver. The key MUST be exactly of the length specific by the Mode. Many receivers may have the same key with the same KeyNum.

When a Consumer reads a manifest that specifies a KeyNum, the consumer SHOULD verify that the Manifest's publisher is an expected one for the KeyNum's usage. This trust mechanism employed to ascertain whether the publisher is expected is beyond the scope of this document, but we provide an outline of one such possible trust mechanism. When a consumer learns a shared key and KeyNum, it associates that KeyNum with the publisher ID used in a public key signature. When the consumer receives a signed manifest (e.g. the root manifest of a manifest tree), the consumer matches the KeyNum's publisher with the Manifest's publisher.

Each encrypted manifest node has a full security context (KeyNum, Nonce, Mode). The AEAD decryption is independent for each manifest so Manifest objects can be fetched and decrypted in any order. This design also ensures that if a manifest tree points to the same subtree repeatedly, such as for deduplication, the decryptions are all idempotent.

To encrypt a Manifest, the publisher:

1. Removes any SecurityCtx or AuthTag from the Manifest.

2. Creates a SecurityCtx and adds it to the Manifest.

3. Treats the Manifest TLV through the end of the Node TLV Length as unencrypted authenticated Header. That includes anything

from the start of the Manifest up to but not including the start of the Node's body.

4. Treats the body of the Node to the end of the Manifest as encrypted data.

5. Appends the AEAD AuthTag to the end of the Manifest, increasing the Manifest's length

6. Changes the TLV type of the Node to EncryptedNode.

To decrypt a Manifest, the consumer:

1. Verifies that the KeyNum exists and the publisher is trusted for that KeyNum.

2. Saves the AuthTag and removes it from the Manifest, decreasing the Manifest length.

3. Changes the EncryptedNode type to Node.

4. Treats everything from the Manifest TLV through the end of the Node Length as unencrypted authenticated Header. That is, all bytes from the start of the Manifest up to but not including the start of the Node's body.

5. Treats the body of the Node to the end of the Manifest as encrypted data.

6. Verifies and decrypts the data using the key and saved AuthTag.

7. If the decryption fails, the consumer SHOULD notify the user and stop further processing of the manifest.

3.8.2. RSA-OAEP Key Transport Mode

The RSA-OAEP mode uses RSA-OAEP (see RFC8017 Sec 7.1 [RFC8017] and [RSAKEM]) to encrypt a symmetric key that is used to encrypt the Manifest. We call this RSA key the Key Encryption Key (KEK) and each group member has this private key. A separate key distribuiton system is responsible for distributing the KEK. For our purposes, it is reasonable to assume that the KEK private key is available at a Locator and that group members can decrypt this private key.

The symmetric key MUST be one that is compatible with the AEAD Mode, i.e. a 128-bit or 256-bit random number. Further, the symmetric key MUST fit in the OAEP envelope (which will be true for normal-sized keys).

Any group key protocol and system needed are outside the scope of this document. We assume there is a Key Manager (KM) and a Publisher (P) and a set of group members. Through some means, the Publisher therefore has at its disposal:

- A Content Encryption Key (CEK), i.e. the symmetric key.
- The RSA-OAEP wrapped CEK.
- The KeyId of the KEK used to wrap the CEK.
- The Locator of the KEK, which is shared under some group key protocol.

This Manifest specification requires that if a group member fetches the KEK key at Locator it can decrypt the WrappedKey and retrieve the CEK.

In one example, a publisher could request a key for a group and the Key Manager could securely communicate (CEK, Wapped_CEK, KeyId, Locator) back to the publisher. The Key Manager is responsible for publishing the Locator. In another example, the publisher could be a group member and have a group private key in which case the publisher can create their own key encryption key, publish it under the Locator and proceed. The publisher generates CEK, Wrapped_CEK, KeyId, and a Locator on its own.

To create the wrapped key using a Key Encryption Key:

1. Obtain the CEK in binary format (e.g. 32 bytes for 256 bits)
2. RSA encrypt the CEK using the KEK public key with OAEP padding, following RFC8017 Sec 7.1 [RFC8017]. The encryption is not signed because the root Manifest must have been signed by the publisher already.

To decrypt the wrapped key using a Key Encryption Key:

1. RSA decrypt the WrappedKey using the KEK private key with OAEP padding, following RFC8017 Sec 7.1 [RFC8017].
2. Verify the unwrapped key is a valid length for the AEADMode.

To encrypt a Manifest, the publisher:

1. Acquires the set of (CEK, Wrapped_CEK, KeyId, Locator).
2. Creates a SecurityCtx and adds it to the Manifest. The SecurityCtx includes an AEADNonce and AEADMode, as per AEAD mode.

3. Encrypts the Manifest as per AEAD Mode using the RSA-OAEP SecurityCtx and CEK.

To decrypt a Manifest, the consumer:

1. Acquires the KEK from the Key Locator. If the consumer already has a cached copy of the KeyId in memory, it may use that cached key.

2. SHOULD verify that it trusts the Manifest publisher to use the provided key Locator.

3. Decrypts the WrappedKey to get the CEK. If the consumer has already decrypted the same exact WrappedKey TLV block, it may use that cached CEK.

4. Using the CEK, AEADNonce, and AEADMode, decrypt the Manifest as per AEAD Mode, ignoring the KeyNum steps.

3.9. Protocol Encodings

3.9.1. CCNx Encoding

In CCNx, application data content objects use a PayloadType of T_PAYLOADTYPE_DATA. In order to clearly distinguish FLIC Manifests from application data, a different payload type is required. Therefore this specification defines a new payload type of T_PAYLOADTYPE_FLIC.

```
ManifestContentObject = TYPE LENGTH [Name] [ExpiryTime]
                        PayloadType Payload
Name = TYPE LENGTH *OCTET ; As per RFC8569
ExpiryTime = TYPE LENGTH *OCTET ; As per RFC8569
PayloadType = TYPE LENGTH T_PAYLOADTYPE_FLIC ; Value TBD
Payload : TYPE LENGTH *OCTET ; the serialized Manifest object
```

Figure 5: CCNx Embedding Grammar

3.9.1.1. CCNx Hash Naming Strategy

The Hash Naming Strategy uses CCNx nameless content objects. This means that only the Root Manifest should have a name embedded in the Content object. All other are CCNx nameless objects. The Manifest should provide a set of Locators that the client may use to form the Interests.

It proceeds as follows:

- The Root Manifest content object bound to a name assigned by the publisher and signed by the publisher. It also may have a set of Locators used to fetch the remainder of the manifest. The root manifest has a single HashPointer that points to the Top Manifest. It may also have cache control directives, such as ExpiryTime.

- The Root Manifest has an NsDef that specifies HashSchema. Its GroupData uses that NsId. All internal and leaf manifests use the same GroupData NsId. A Manifest Tree MAY omit the NsDef and NsId elements and rely on the default being HashSchema.

- The Top Manifest is a nameless CCNx content object. It may have cache control directies.

- Internal and Leaf manifests are nameless CCNx content objects, possibly with cache control directives.

- The Data content objects are nameless CCNx content objects, possibly with cache control directives.

- To form an Interest for a direct or indirect pointer, use a Name from one of the Locators and put the pointer HashValue into the ContentObjectHashRestriction.

3.9.1.2. CCNx Single Prefix Strategy

The Single Prefix strategy uses a named Root manifest and then all other data and sub-manifest objects use the same Name. They are differentiated only by their hash.

It proceeds as follows:

- The Root Manifest content object has a name used to fetch the manifest. It is signed by the publisher. It has a single Locator used to fetch the remainder of the manifest using the commong Single Prefix name. It has a single HashPointer that points to the Top Manifest. It may also have cache control directives, such as ExpiryTime.

- The Root Manifest has an NsDef that specifies PrefixSchema with the Locator for the single prefix.

- The Top Manifest has the name SinglePrefixName. It may have cache control directies. Its GroupData elements must have an NsId that references the NsDef.

*An Internal or Leaf manifest has the name SinglePrefixName, possibly with cache control directives. Its GroupData elements must have an NsId that references the NsDef.

*The Data content objects have the name SinglePrefixName, possibly with cache control directives.

*To form an Interest for a direct or indirect pointer, use SinglePrefixName as the Name and put the pointer HashValue into the ContentObjectHashRestriction.

3.9.1.3. CCNx Segmented Prefix Strategy

The Segmented Prefix schema uses a different name in all Content Objects and distinguishes them via their ContentObjectHash. Note that in CCNx, using a SegmentedPrefixSchema means that only the Root Manifest has a Locator for the Segmented Prefix (minus the segment number).

Optional: Use AnnotatedPointers to indicate the segment number of each hash pointer to avoid needing to infer the segment numbers.

Optional: Use StartSegmentId in GroupData to indicate the segment number of for each group. The producer must ensure that each subsequent GroupData starts at the correct offset.

It proceeds as follows:

*The Root Manifest content object has a name used to fetch the manifest. It is signed by the publisher. It has a set of Locators used to fetch the remainder of the manifest. It has a single HashPointer that points to the Top Manifest. It may also have cache control directives, such as ExpiryTime.

*The Root Manifest has an NsDef that specifies SegmentedPrefix and the SegmentedPrefixSchema element specifies the SegmentedPrefixName.

*The publisher tracks the chunk number of each content object within the NsId. Objects are be numbered in their traversal order. Within each manifest, the name can be constructed from the SegmentedPrefixName plus a Chunk name component.

*The Top Manifest has the name SegmentedPrefixName plus chunk number. It may have cache control directies. It's GroupData elements must have an NsId that references the NsDef.

*An Internal or Leaf manifest has the name SegmentedPrefixName plus chunk number, possibly with cache control directives. Its GroupData elements must have an NsId that references the NsDef.

*The Data content objects have the name SegmentedPrefixName plus chunk number, possibly with cache control directives.

*To form an Interest for a direct or indirect pointer, use SegmentedPrefixName plus chunk number as the Name and put the pointer HashValue into the ContentObjectHashRestriction. A consumer must track the chunk number in traversal order for each SegmentedPrefixSchema NsId.

3.9.1.4. CCNx Hybrid Strategy

A manifest may use multiple schemas. For example, the application payload in data content objects might use SegmentedPrefix while the manifest content objects might use HashNaming.

The Root Manifest should specify an NsDef with a first NsId (say 1) as the HashNaming schema and a second NsDef with a second NsId (say 2) as the SegmentedPrefix schema along with the SegmentedPrefixName.

Each manifest (Top, Internal, Leaf) uses two or more HashGroups, where each HashGroup has only Direct (with the second NsId) or Indirect (with the first NsId). The number of hash groups will depend on how the publisher wishes to interleave direct and indirect pointers.

Manifests and data objects derive their names according to the application's naming schema.

3.9.2. NDN Encoding

In NDN, all Manifest Data objects use a ContentType of FLIC (1024), while all application data content objects use a PayloadType of Blob.

3.9.2.1. NDN Hash Naming

In NDN Hash Naming, a Data Object has a 0-length name. This means that an Interest will only have an ImplicitDigest name component in it. This method relies on using NDN Forwarding Hints.

It proceeds as follows:

*The Root Manifest Data has a name used to fetch the manifest. It is signed by the publisher. It has a set of Locators used to fetch the remainder of the manifest. It has a single HashPointer that points to the Top Manifest. It may also have cache control directives.

*The Root Manifest has an NsDef that specifies HashSchema. Its GroupData uses that NsId. All internal and leaf manifests use the

same GroupData NsId. A Manifest Tree MAY omit the NsDef and NsId elements and rely on the default being HashSchema.

*The Top Manifest has a 0-length Name. It may have cache control directies.

*Internal and Leaf manifests has a 0-length Name, possibly with cache control directives.

*The application Data use a 0-length name, possibly with cache control directives.

*To form an Interest for a direct or indirect pointer, the name is only the Implicit Digest name component derived from a pointer's HashValue. The ForwardingHints come from the Locators. In NDN, one may use one or more locators within a single Interest.

3.9.2.2. NDN Single Prefix

In Single Prefix, the Data name is a common prefix used between all objects in that namespace, without a Segment or other counter. They are distinguished via the Implicit Digest name component. The FLIC Locators go in the ForwardingHints.

It proceeds as follows:

*The Root Manifest Data object has a name used to fetch the manifest. It is signed by the publisher. It has a set of Locators used to fetch the remainder of the manifest. It has a single HashPointer that points to the Top Manifest. It may also have cache control directives.

*The Root Manifest has an NsDef that specifies SinglePrefix and the SinglePrefixSchema element specifies the SinglePrefixName.

*The Top Manifest has the name SinglePrefixName. It may have cache control directies. Its GroupData elements must have an NsId that references the NsDef.

*An Internal or Leaf manifest has the name SinglePrefixName, possibly with cache control directives. Its GroupData elements must have an NsId that references the NsDef.

*The Data content objects have the name SinglePrefixName, possibly with cache control directives.

*To form an Interest for a direct or indirect pointer, use SinglePrefixName as the Name and append the pointer's HashValue into an ImplicitDigest name component. Set the ForwardingHints from the FLIC locators.

3.9.2.3. NDN Segmented Prefix

In Segmented Prefix, the Data name is a common prefix plus a segment number, so each manifest or application data object has a unique full name before the implicit digest. This means the consumer must maintain a counter for each SegmentedPrefix namespace.

Optional: Use AnnotatedPointers to indicate the segment number of each hash pointer to avoid needing to infer the segment numbers.

Optional: Use StartSegmentId in GroupData to indicate the segment number of for each group. The producer must ensure that each subsequent GroupData starts at the correct offset.

It proceeds as follows:

*The Root Manifest Data object has a name used to fetch the manifest. It is signed by the publisher. It has a set of Locators used to fetch the remainder of the manifest. It has a single HashPointer that points to the Top Manifest. It may also have cache control directives.

*The Root Manifest has an NsDef that specifies SegmentedPrefix and the SegmentedPrefixSchema element specifies the SegmentedPrefixName.

*The publisher tracks the segment number of each Data object within a SegmentedPrefix NsId. Data is numbered in traversal order. Within each manifest, the name is constructed from the SegmentedPrefixName plus a Segment name component.

*The Top Manifest has the name SegmentedPrefixName plus segment number. It may have cache control directies. Its GroupData elements must have an NsId that references the NsDef.

*An Internal or Leaf manifest has the name SegmentedPrefixName plus segment number, possibly with cache control directives. Its GroupData elements must have an NsId that references the NsDef.

*The Data content objects have the name SegmentedPrefixName plus chunk number, possibly with cache control directives.

*To form an Interest for a direct or indirect pointer, use SegmentedPrefixName plus segment number as the Name and put the pointer HashValue into the ImplicitDigest name component. A consumer must track the segment number in traversal order for each SegmentedPrefixSchema NsId.

3.9.2.4. NDN Hybrid Schema

A manifest may use multiple schemas. For example, the application payload in data content objects might use SegmentedPrefix while the manifest content objects might use HashNaming.

The Root Manifest should specify an NsDef with a first NsId (say 1) as the HashNaming schema and a second NsDef with a second NsId (say 2) as the SegmentedPrefix schema along with the SegmentedPrefixName.

Each manifest (Top, Internal, Leaf) uses two or more HashGroups, where eash HashGroup has only Direct (with the second NsId) or Indirect (with the first NsId). The number of hash groups will depend on how the publisher wishes to interleave direct and indirect pointers.

Manifests and data objects derive their names according to the application's naming schema.

3.9.3. Segmented Schema Details

When using CCNx Segmented Prefix Strategy or NDN Segmented Prefix strategy, the consumer must determine the segment number to use in the name. There are two methods.

* If fetching a pointer with a SegmentIdAnnotation, the consumer MUST use that segment number for the pointer. A pointer with SegmentIdAnnotation does not increment the SegmentId used by the GroupData case.

* If the GroupData has a StartSegmentId parameter, then that segment number MUST be used for the first in-order pointer of the group. The consumer then increments the segment number for each in-order pointer of that group.

Every group of a segmented NsId MUST have either a GroupData with a StartSegmentId, or use annotated pointers with SegmentIdAnnotation.

A segment number MUST indicate exactly one data item. That is, the producer MUST NOT duplicate the segment number in an object name for different objects. The object hash MUST be the same for the same segment number of a name.

It is allowed to have multiple manifest entries with the same segment number (see below).

While a producer is allowed to mix using GroupData StartSegmentId and SegmentIdAnnotation, we in general do not consider that a good idea. It is up to the manifest producer to ensure that every segment

may be fetched, and fetch in the right order. Segments, when fetched in the **manifest order** reconstruct the original data.

Let us make this clear, the original data is constructed by the in-order manifest retrieval, not the segment number order. We recommend that the manifest in-order sequence SHOULD correspond to the segment number sequence.

A consumer is not required to fetch every segment. A consumer may fetch segments in any order it chooses. It may skip around or omit segments.

It is allowed to have multiple pointers to the same segment number. This can be used for data de-duplication, e.g. multiple occurances of the same binary string within the reconstructed data object. If the producer uses this method, then the original data cannot be reconstructed by simply fetching the sequence numbers in order.

3.10. Example Structures

3.10.1. Leaf-only data

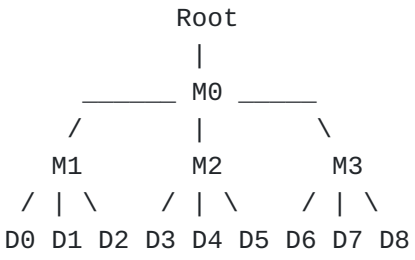


Figure 6: Leaf-only manifest tree

3.10.2. Linear

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.

```
Root -> M0 ----> M1 ----> ...
|->DDDD  |->DDDD
```

4. Experimenting with FLIC

FLIC is expected to enable a number of salient experiments in the use of ICN protools by applications. These experiments will help not only to inform the desirable structure of ICN applications but

reflect back to the features included in FLIC to evaluate their usefulness to those applications. While many interesting design aspects of FLIC remain to be discovered through experience, a number of important questions to be answered through experimentation include:

- *use for just files or other collections like directories
- *use for particular applications, like streaming media manifests
- *utility of pointer annotations to optimize retrieval
- *utility of the encryption options for use by repositories and forwarders
- *need for application metadata in manifests

5. Usage Examples

5.1. Locating FLIC leaf and manifest nodes

The names of manifest and data objects are often missing or not unique, unless using specific naming conventions. In this example, we show how using manifest locators is used to generate Interests. Take for example the figure below where the root manifest is named by hash h0. It has nameless children with hashes with hashes h1 ... hN.

```
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=/a/b/c, implicitDigest=h0)
```

Figure 7: Data Organization

After obtaining the manifest, the client fetches the contents. In this first instance, the manifest does not provide any Locators data structure, so the client must continue using the name it used for the manifest.

```
interest(name=/a/b/c, implicitDigest=h1)
...
interest(name=/a/b/c, implicitDigest=hN)
```

Figure 8: Data Interests

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=/a/b/c, implicitDigest=h0)
interest(name=/x/y/z, implicitDigest=h1)
interest(name=/a/b/c, implicitDigest=h2)
```

Figure 9: Using Locators

5.2. 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, call it offset_i. In this formula, let P_i.size represent the Size value of the *i*-th pointer.

```
offset_i = \sum_{k=1}^{i-1} &gt; P_k.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
Algorithm:
offset = 0
for i in 1..N do
   if (P &gt; offset + P_i.size)
      return (i, P - offset)
   offset += P_i.size
return out-of-range
```

Figure 10: Seeking Algorithm

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_N is equal to the following:

```
D - ((N - 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
Algorithm:
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)
```

Figure 11: Seeking Algorithm

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.

5.3. Block-level de-duplication

Consider a huge file, e.g. an ISO image of a DVD or program in binary 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 diagram below. For example, the venti archival file system of Plan9 [venti] uses this technique.

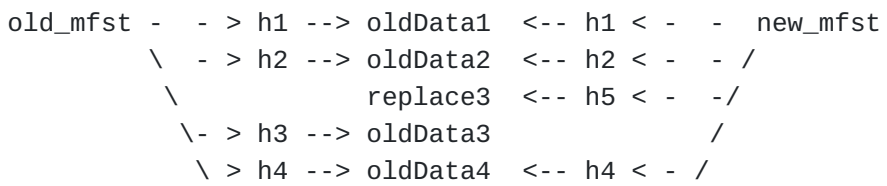


Figure 12: De-duplication

5.4. 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).

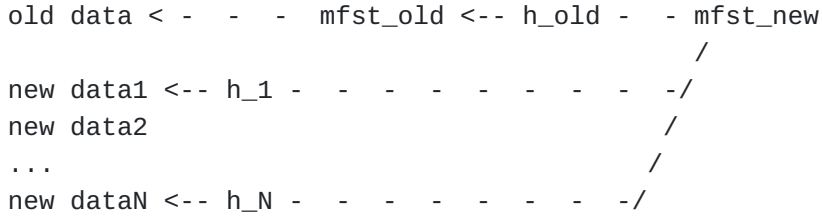


Figure 13: Growing A Collection

5.5. Re-publishing a FLIC under a new name

There are several use cases for republishing a collection under a new namespace, or having one collection exist under several namespaces:

- 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.

- A publishes wishes to distribute its content to several repositories and would like a result to be delivered from the repository for consumers who have good connectivity to that repository. For example, the publisher /alpha wishes to place content at /beta and /gamma, but routing only to /alpha would not send a request to either /beta or /gamma. The operators of of /beta and /gamma could create a named and signed version of the root manifest with appropriate keys (or delegate that to /alpha) so the results are always delivered by the corresponding repository without having to change the bulk of the manifest tree.

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):

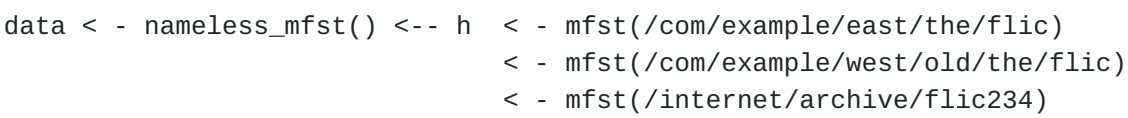


Figure 14: Relocating A Collection

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 their own 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.

6. IANA Considerations

IANA is requested to perform the actions in the following sub-sections.

IANA should also note that FLIC uses the definitions of AEAD_AES_128_GCM, AEAD_AES_128_CCM, AEAD_AES_256_GCM, AEAD_AES_256_CCM from [RFC5116].

6.1. FLIC Payload Type

Register FLIC as a Payload Type in the *CCNx Payload Types* Registry referring to the description in Section 3.9.1 as follows:

Type	Name	Reference
TBA	T_PAYLOADTYPE_FLIC	Section 3.9.1 and Section 3.6.2.2.1 of [RFC8609]

Table 1: FLIC CCNx Payload Type

6.2. FLIC Manifest Metadata and Annotation TLVs

Create the following registry to be titled *FLIC Manifest Metadata and Annotation TLVs* Manifest Metadata is described in Section 3.4; Pointer Annotations are described in Section 3.5. The registration procedure is **Specification Required**. The Type value is 2 octets. The range is 0x0000-0xFFFF. Allocate a value for the single *SizeAnnotation* TLV.

Type	Name	Reference
TBA	T_SIZE_ANNOTATION	Size (Section 3.5)

Table 2: FLIC Manifest Metadata and Annotation TLVs

7. Security Considerations

TODO Need a discussion on:

* signing and hash chaining security. (**Note: Did I cover this adequately below?**)

* republishing under a new namespace. (**Note: need help here - is this to reinforce that you can re-publish application data by creating a new root Manifest and signing that, requiring only one signature to change?**)

* encryption mechanisms. (**Note: did I cover this adequately below?**)

* encryption key distribution mechanisms.(**Note: not sure what needs to be said here**)

* discussion of privacy, leaking of linkability information - **could really use some help here**.

Anything else?????.

7.1. Integrity and Origin Authentication of FLIC Manifests

A FLIC Manifest is used to describe how to form Interests to access large CCNx or NDN application data. The Manifest is itself either an individual content object, or a tree of content objects linked together via the corresponding content hashes. The NDN and CCnx protocol architectures directly provide both individual object integrity (using cryptographically strong hashes) and data origin authentication (using signatures). The protocol specifications, [NDN] and CCNx [RFC8609] respectively, provide the protocol machinery and keying to support strong integrity and authentication. Therefore, FLIC utilizes the existing protocol specifications for these functions, rather than providing its own. There are a few subtle differences in the handling of signatures and keys in NDN and CCNx worth recapitulating here:

* NDN in general adds a signature to every individual data packet rather than aggregating signatures via some object-level scheme. When employing FLIC Manifests to multi-packet NDN objects, it is expected that the individual packet signatures would be elided and the signture on the Manifest used instead.

* In contrast, CCNx is biased to have primitive objects or pieces thereof be "nameless" in the sense they are identified only by their hashes rather than each having a name directly bound to the content through an individual signature. Therefore, CCNx depends heavily on FLIC (or an alternative method) to provide the name and the signed binding of the name to the content described in the Manifest

A FLIC Manifest therefore gets integrity of its individual pieces through the existing secure hashing procedures of the underlying protocols. Origin authentication of the entire Manifest is achieved through hash chaining and applying a signature **only** to the root Manifest of a manifest tree. It is important to note that the Name

of the Manifest, which is what the signature is bound to, need not bear any particular relationship to the names of the application objects pointed to in the Manifest via Name Constructors. This has a number of important benefits described in Section 3.3.

7.2. Confidentiality of Manifest Data

ICN protocol architectures like CCNx and NDN, while providing integrity and origin authentication as described above, leaves confidentiality issues entirely in the domain of the ICN application. Therefore, since FLIC is an application-level construct in both NDN and CCNx, it is incumbent on this specification for FLIC to provide the desired confidentiality properties using encryption. One could leave the specification of Manifest encryption entirely in the hands of the individual application utilizing FLIC, but this would be undesirable for a number of reasons:

* The sensitivity of the information in a Manifest may be different from the sensitivity of the application data it describes. In some cases, it may not be necessary to encrypt manifests, or to encrypt them with a different keying scheme from that used for the application data

* One of the major capabilities enabled by FLIC is to allow repositories or forwarding caches to operate on Manifests (see in particular Section 3.4). In order to allow such intermediaries to interpret manifests without revealing the underlying application data, separate encryption and keying is necessary

* A strong design goal of FLIC is *universality* such that it can be used transparently by many different ICN applications. This argues that FLIC should have a set of common encryption and keying capabilities that can be delegated to library code and not have to be re-worked by each individual application (see Section 2, Paragraph 11)

Therefore, this specification directly specifies two encryption encapsulations and associated links to key management, as described in Section 3.8. As more experience is gained with various use cases, additional encryption capabilities may be needed and hence we expect the encryption aspects of this specification to evolve over time.

7.3. Privacy of names and linkability of access patterns

What to say here, if anything?

8. References

8.1. 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, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3310] Niemi, A., Arkko, J., and V. Torvinen, "Hypertext Transfer Protocol (HTTP) Digest Authentication Using Authentication and Key Agreement (AKA)", RFC 3310, DOI 10.17487/RFC3310, September 2002, <https://www.rfc-editor.org/info/rfc3310>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <https://www.rfc-editor.org/info/rfc5116>.

[RFC5288] Salowey, J., Choudhury, A., and D. McGrew, "AES Galois Counter Mode (GCM) Cipher Suites for TLS", RFC 5288, DOI 10.17487/RFC5288, August 2008, <https://www.rfc-editor.org/info/rfc5288>.

[RFC5990] Randall, J., Kaliski, B., Brainard, J., and S. Turner, "Use of the RSA-KEM Key Transport Algorithm in the Cryptographic Message Syntax (CMS)", RFC 5990, DOI 10.17487/RFC5990, September 2010, <https://www.rfc-editor.org/info/rfc5990>.

[RFC6655] McGrew, D. and D. Bailey, "AES-CCM Cipher Suites for Transport Layer Security (TLS)", RFC 6655, DOI 10.17487/RFC6655, July 2012, <https://www.rfc-editor.org/info/rfc6655>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/info/rfc8017>.

[RFC8569] Mosko, M., Solis, I., and C. Wood, "Content-Centric Networking (CCNx) Semantics", RFC 8569, DOI 10.17487/RFC8569, July 2019, <https://www.rfc-editor.org/info/rfc8569>.

[RFC8609] Mosko, M., Solis, I., and C. Wood, "Content-Centric Networking (CCNx) Messages in TLV Format", RFC 8609, DOI 10.17487/RFC8609, July 2019, <https://www.rfc-editor.org/info/rfc8609>.

8.2. Informative References

[AESGCM]
Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", National Institute of Standards and Technology SP 800-38D, 2007, <https://doi.org/10.6028/NIST.SP.800-38D>.

[FLICImplementation] Mosko, M., "FLIC Implementation in Python", various, <https://github.com/mmosko/ccnpy>.

[I-D.wood-icnrg-ccnxkeyexchange] Mosko, M., Uzun, E., and C. A. Wood, "CCNx Key Exchange Protocol Version 1.0", Work in Progress, Internet-Draft, draft-wood-icnrg-ccnxkeyexchange-02, 6 July 2017, <https://datatracker.ietf.org/doc/html/draft-wood-icnrg-ccnxkeyexchange-02>.

[NDN] "Named Data Networking", various, <https://named-data.net/project/execsummary/>.

[NDNTLV] "NDN Packet Format Specification.", 2016, <http://named-data.net/doc/ndn-tlv/>.

[ProjectOrigin] Mosko, M., "Peer-to-Peer Sharing with CCNx 1.0", 2014, <https://github.com/PARC/CCNxReports/blob/master/SelectedTopics/p2pshare.pdf>.

[repository] "Repo Protocol Specification", Various, <https://redmine.named-data.net/projects/repo-ng/wiki/Repo_Protocol_Specification>.

[RFC7927]
Kutscher, D., Ed., Eum, S., Pentikousis, K., Psaras, I., Corujo, D., Saucez, D., Schmidt, T., and M. Waehlisch, "Information-Centric Networking (ICN) Research Challenges", RFC 7927, DOI 10.17487/RFC7927, July 2016, <https://www.rfc-editor.org/info/rfc7927>.

[RSAKEM] Barker, E., Chen, L., Roginsky, A., Vassilev, A., Davis, R., and S. Simon, "Recommendation for Pair-Wise Key-Establishment Using Integer Factorization Cryptography", National Institute of Standards and Technology SP 800-56B Rev. 2, 2019, <https://doi.org/10.6028/NIST.SP.800-56Br2>.

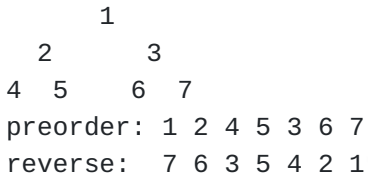
[SHS] Technology, N. I. O. S. A., "Secure Hash Standard, United States of American, National Institute of Science and Technology, Federal Information Processing Standard (FIPS) 180-4", National Institute of Standards and Technology SP 180-4, 2012, <https://csrc.nist.gov/publications/fips/fips180-4/ fips180-4_final.pdf>.

[venti]
"Venti: a new approach to archival storage", Bell Labs Document Archive /sts/doc, 2002, <http://doc.cat-v.org/plan_9/4th_edition/papers/venti/>.

Appendix A. Building Trees

This appendix describes one method to build trees. It constructs a pre-order tree in a single pass of the application data, going from the tail to the beginning. This allows us to work up the right side of the tree in a single pass, then work down each left branch until we exhaust the data. Using the reverse-order traversal, we create the right-most-child manifest, then its parent, then the indirect pointers of that parent, then the parent's direct pointers,then the parent of the parent (repeating). This process uses recursion, as it is the clearest way to show the code. A more optimized approach could do it in a true single pass.

Because we're building from the bottom up, we use the term 'level' to be the distance from the right-most child up. Level 0 is the bottom-most level of the tree, such as where node 7 is:



The Python-like pseudocode build_tree(data, n, k, m) algorithm creates a tree of n data objects. The data[] array is an array of Content Objects that hold application payload; the application data has already been packetized into n Content Object packets.An interior manifest node has k direct pointers and m indirect pointers.

```
build_tree(data[0..n-1], n, k, m):
  # data is an array of Content Objects (Data in NDN) with app data.
  # n is the number of data items
  # k is the number of direct pointers per internal node
  # m is the number of indirect pointers per internal node

  segment = namedtuple('Segment', 'head tail')(0, n)
  level = 0

  # This bootstraps the process by creating the right most child
  # manifest. A leaf manifest has no indirect pointers, so k+m
  # are direct pointers
  root = leaf_manifest(data, segment, k + m)

  # Keep building subtrees until we're out of direct pointers
  while not segment.empty():
    level += 1
    root = bottom_up_preorder(data, segment, level, k, m, root)

  return root

bottom_up_preorder(data, segment, level, k, m, right_most_child=None):
  manifest = None
  if level == 0:
    assert right_most_child is None
    # build a leaf manifest with only direct pointers
    manifest = leaf_manifest(data, segment, k + m)
  else:
    # If the number of remaining direct pointers will fit
    # in a leaf node, make one of those. Otherwise, we need to be
    # an interior node
  if right_most_child is None and segment.length() &lt;= k + m:
    manifest = leaf_manifest(data, segment, k+m)
  else:
    manifest = interior_manifest(data, segment, level,
                                 k, m, right_most_child)
  return manifest

leaf_manifest(data, segment, count):
  # At most count items, but never go before the head
  start = max(segment.head(), segment.tail() - count)
  manifest = Manifest(data[start:segment.tail])
  segment.tail -= segment.tail() - start
  return manifest
```

```
interior_manifest(data, segment, level, k, m, right_most_child)
  children = []
  if right_most_child is not None:
    children.append(right_most_child)

  interior_indirect(data, segment, level, k, m, children)
  interior_direct(data, segment, level, k, m, children)

  manifest = Manifest(children)
  return manifest, tail

interior_indirect(data, segment, level, k, m, children):
  # Reserve space at the head of the segment for this node's
  # direct pointers before descending to children.  We want
  # the top of the tree packed.
  reserve_count = min(k, segment.tail - segment.head)
  segment.head += reserve_count

  while len(children) &lt; m and not segment.head == segment.tail:
    child = bottom_up_preorder(data, segment, level - 1, k, m)
    # prepend
    children.insert(0, child)

    # Pull back our reservation and put those pointers in
    # our direct children
    segment.head -= reserve_count

interior_direct(data, segment, level, k, m, children):
  while len(children) &lt; k+m and not segment.head == segment.tail:
    pointer = data[segment.tail() - 1]
    children.insert(0, pointer)
    segment.tail -= 1
```

Authors' Addresses

Christian Tschudin
University of Basel

Email: christian.tschudin@unibas.ch

Christopher A. Wood
Cloudflare

Email: caw@heapingbits.net

Marc Mosko
PARC, Inc.

Email: marc.mosko@parc.com

David Oran (editor)
Network Systems Research & Design

Email: daveoran@orandom.net