ICNRG
Internet-Draft
Intended status: Experimental
Expires: May 7, 2020

C. Tschudin
University of Basel
C. Wood
University of California Irvine
M. Mosko
PARC, Inc.
D. Oran, Ed.
Network Systems Research & Design
November 4, 2019

File-Like ICN Collections (FLIC)
draft-irtf-icnrg-flic-02

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

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 May 7, 2020.

Table of Contents

1. Introduction

ICN architectures such as Content-Centric Networking (CCN)[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. (See [I-D.irtf-icnrg-terminology] for a formal definition of "full name".)

To enable requests with full names, consumers need a priori knowledge of content digests. Manifests, or catalogs, are data structures commonly proposed to transport this information. Typically, manifests are signed content objects (data) which carry a collection of hash digests. However, as content objects, manifests themselves may be fetched by full name. Thus, manifests may contain hash digests of, or pointers to, 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.

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 ICNs such as CCN and NDN. We describe the FLIC design, grammar, and various use cases, e.g., seeking, de-duplication, extension, and variable-sized encoding. We also include FLIC encoding examples for CCN and NDN.

The purpose of a manifest is to concisely name the constiuent pieces of a larger object. A FLIC manifest does this by using a first manifest to name and cryptographically sign the data structure and then use more 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 (routing hints).

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

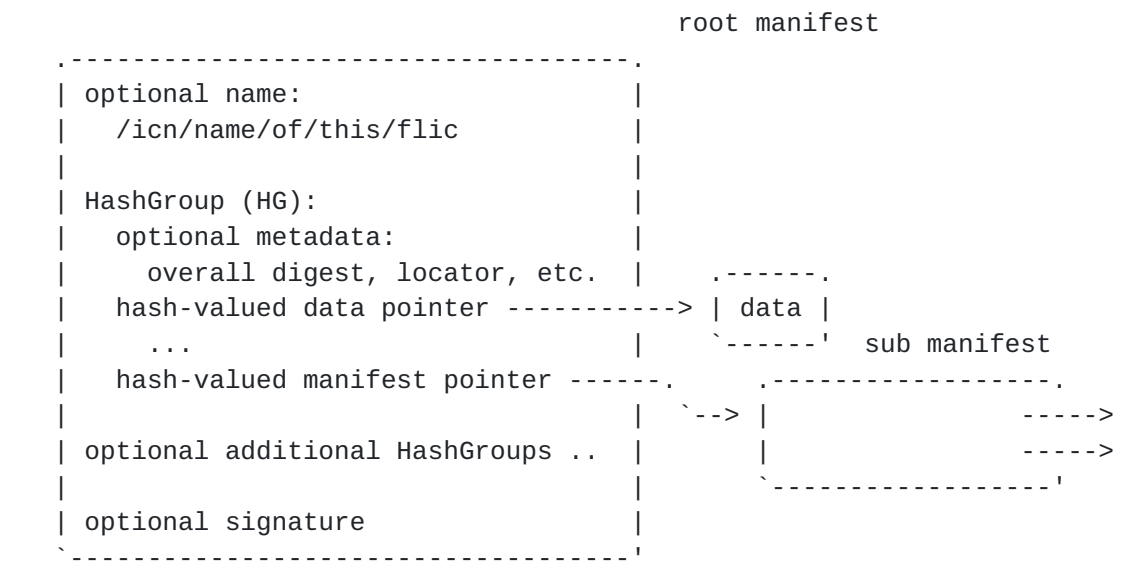

Figure 1: A FLIC manifest and its directed acyclic graph

A key question is how one names the root manifest, the application data, and other subsequent manifests. The question of namespaces is specific to the names of each Content Object (CCNx) or Data (NDN), and is separate from the question of Locators. FLIC allows one to use a first namespace for the manifests and a second namespace for the application data. A given namespace may use one of three schemas: hash-based naming, single-prefix naming, or segmented naming. We describe the allowed 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.

Locators are routing hints to find a Content Object / Data. They exist in both CCNx and NDN, though the specifics differ. A FLIC manifest encodes locators the same for both ICN protocols, though they are encoded differently in the underlying protocol. See Section 3.9 for encoding differences.

We follow the CCNx [RFC8569] terminology where a Content Object is the data structure that holds application payload. It is made up of an optional Name, a PayloadType, a Payload, and an optional Signature.

FLIC has encodings for CCNx encoding (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. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Design Goals

The preferred FLIC structure copies the proven UNIX inode concept of direct and indirect pointers, but without the specific structural forms of direct versus indirect.

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.

Links in FLIC use hash-based naming of Content Objects, rather than inode block numbers. Both CCNx and NDN support hash-based naming, though the details vary. See Section 3.9.1 and Section 3.9.2. Another advantage of using hash-based naming is it permits block-level de-duplication of application data because two blocks with the same payload will have the same hash name.

Because FLIC uses hash-based naming, FLIC graphs are inherently acyclic.

The preferred FLIC structure includes a root manifest with a strong cryptographic signature and then strong hash names to other manifests (e.g. SHA256). The advantage of this structure is 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.

FLIC supports manifest encryption separate from application payload encryption. 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.

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.

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.

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 manifests.

Leaf Manifest: all pointers are direct pointers.

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.

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.

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: A preferred 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.

Namespace: The prefix and object name that goes inside a Content Object. It may include typed name components specifying a version and/or chunk/segment number.

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

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.

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 usecase 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. Namespaces

A FLIC manifest may define zero or more namespaces. If none are defined, FLIC uses the default Hash Naming approach. If using namespaces, typically there are two defined: one for the manifest namespace and one for the application data namespace. If the two are the same, they can share a namepace. There may be more than two namespaces.

A namespace follows a naming convention. The naming convention governs how FLIC creates the ICN Name that goes in an Interest's Name and must match a Content Object / Data Name. The conventions are: (1) Hash Naming, (2) Single Prefix, and (3) Segmented Prefix. The default is to use Hash Naming. Hash Naming does not include anything besides a hash name in the Interest's name and relies on the Locator to forward the Interest. Single Prefix uses the same name differntiated only by a Content Object's implicit hash. Segmented Prefix keeps a counter for the namespace, starting with 0, and increments it after each use of the namespace.

The namespace definitions may be inherited from the Root manifest or the Top manifest, or any prior manifest. It is RECOMMENDED that the namespace definitions appear in the Root manifest so they can be updated by a single operation. Because Segmented Prefix namespaces

use a counter, it is RECOMMENDED to only define them in the Root manifest or Top manifest and not elsewhere, as it may confuse the counters.

In the NodeData, there may be zero or more NSDef contains. Each NSDef defines a namespace identifier (octet string) and its naming convention. For the Hash Naming convention, no further information is required. For the Single Prefix and Segmented Prefix conventions, the NSDef specifies the ICN Name prefix used by the namespace.

A HashGroup may have an NSRef container that indicates which namespace it is using, and by implication which naming convention and the corresponding prefix. If there is no NSRef, the hash group uses Hash Naming convention.

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 similar 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 of delay performance. We note in passing that FLICs use of separate namespaces for the Manifest and the underlying Data allows different encryption keys to be used, hence giving a 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 preferred 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 similar to [RFC8609] for vendor-specific annotations that require no standardization process.

3.6. Manifest Grammar (ABNF)

The manifest grammar is mostly independent of the transport ICN protocol. 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. 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. The Section 3.9 section below specifies these differences.

The basic structure of a FLIC manifest is 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 HashGroups. 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 hash group 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 HashGroups.

```
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 = PresharedKeyCtx / RsaKemCtx / RsaKemDemCtx
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*NSDef
SubtreeSize = TYPE LENGTH INTEGER
SubtreeDigest = TYPE LENGTH HashValue
NSDef = TYPE LENGTH NsId NsSchema
NsId = TYPE LENGTH INTEGER
NsSchema = HashSchema / SinglePrefixSchema / SegmentedPrefixSchema
HashSchema = TYPE 0
SinglePrefixSchema = TYPE LENGTH Name
SegmentedPrefixSchema = TYPE LENGTH Name

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

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

GroupData = TYPE LENGTH [LeafSize] [LeafDigest] [SubtreeSize] [SubtreeDigest]
[NsId]
LeafSize = TYPE LENGTH INTEGER
LeafDigest = TYPE LENGTH HashValue

PresharedKeyCtx = TYPE LENGTH PresharedKeyData
PresharedKeyData = KeyNum IV Mode
KeyNum = TYPE LENGTH INTEGER
IV = TYPE LENGTH 1*OCTET
Mode = TYPE LENGTH (AES-GCM-128 / AES-GCM-256)

RsaKemCtx = 2 LENGTH RsaKemData
RsaKemData = KeyId IV Mode WrappedKey LocatorPrefix
KeyId = TYPE LENGTH HashValue; ID of Key Encryption Key
WrappedKey = TYPE LENGTH 1*OCTET
LocatorPrefix = TYPE LENGTH Link

RsaKemDemCtx = 3 LENGTH RsaKemDemData
RsaKemDemData = KeyId IV Mode WrappedKey LocatorPrefix
```

Figure 1: 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

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 2 show the in-order forward traversal code and the reverse-oder 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 < node.child.length, i++)
        preorder(node.child[i])

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

Figure 2: 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 Figure 3 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.

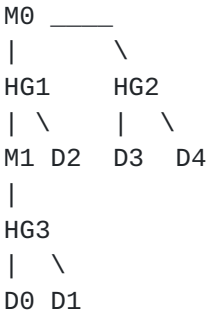

Figure 3: Node Expansion

Using the example manifest tree shown in Figure Figure 9, the in-order traversal would be: Root, M0, M1, D0, D1, D2, M2, D3, D4, D5, M3, D6, D7, D8.

3.8. Manifest Encryption

This document specifies three encryption methods. The first is a preshared key algorithm, where the parties are assumed to have the decryption keys already. This is useful, for example, when using a key agreement protocol such as CCNxKE. The second is an RSA key encapsulation mechanisms (RsaKem). The third is a standard RSA KEM-DEM mechanism that uses a shared group key (RsaKemDem).

For group key based encryption, RsaKem and RsaKemDem, this specification only details the pertinent aspects of the encryption. 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 it's simpliest form, the publisher could be the key manager, so there is no extra protocol needed between the publisher and key manager. This specification does describe how a consumer locates the appropriate keys.

While the preshared key algorithm is limited in use, the AES encryption mechanisms described apply to the group key mechanisms too. They group key mechanisms simply facilitate distribution of the shared key without an on-line key agreement protocol like CCNxKE.

A fourth encryption mechanism based on elliptic curve key distribution is forthcoming.

3.8.1. Preshared Key Algorithm

The KeyNum identifies a key on the receiver. The key must be of the correct length of the Mode used. If the key is longer, use the left bits. Many receivers many have the same key with the same KeyId. A publisher creates a signed root manifest with a security context. A consumer must ensure that the root manifest signer is the expected publisher for use with the pre-shared key, which may be shared with many other consumers. The publisher may use either method 8.2.1 (deterministic IV) or 8.2.2 (RBG-based IV) [NIST 800-38D] for creating the IV.

Each encrypted manifest node (root manifest or internal manifest) has a full security context (KeyNum, IV, Mode). The AES-GCM 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.

The functions for authenticated encryption and authenticated decryption are as given in Sections 7.1 and 7.2 of NIST 800-38D: GCM-AE_K(IV, P, A) and GCM-AD_K(IV, C, A, T).

```
EncryptNode(SecurityCtx, Node, K, IV) -> GCM-AE_K(IV, P, A) -> (C, T)
    Node: The wire format of the Node (P)
    SecurityCtx: The wire format of the SecurityCtx as the Additional
Authenticated Data (A)
    K: the pre-shared key (128 or 256 bits)
    IV: The initialization vector (usually 96 or 128 bits)
    C: The cipher text
    T: The authentication tag
```

Figure 4: Preshared Key Encrypt

The pair (C,T) is the OpaqueNode encoded as a TLV structure: (OpaqueNode (CipherText C) (AuthTag T)).

```
DecryptNode(SecurityCtx, C, T, K, IV) -> GCM-AD_K (IV, C, A, T) -> (Node,
FailFlag)
Node: The wire format of the decrypted Node
FailFlag: Indicates authenticated decryption failure (true or false)
```

Figure 5: Preshared Key Decrypt

If doing in-place decryption, the cipher text C will be enclosed in an EncryptedNode TLV value. After decryption, change the TLV type to Node. The length should be the same. After decryption the AuthTag is no longer needed. The TLV type should be changed to T_PAD and the value zeroed. The SecurityCtx could be changed to T_PAD and zeroed or left as-is.

3.8.2. RSA Key Encapsulation

See also RFC 5990, NIST SP 800-56B Rev. 2 and https://lists.w3.org/Archives/Public/public-xmlsec/2009May/att-0032/Key_Encapsulation.pdf

In this system, a key manager (KM) (which could be the publisher) creates a symmetric Content Encryption Key (CEK) and a key wrapping pair with a Key Encryption Key (KEK) and Key Decryption Key (KDK). Each publisher and consumer has its own public/private key pair, and the KM knows each publisher's and consumer's identity and its public key (PK_x).

We do not describe the publisher-key manager protocol to request a CEK. The publisher will obtain the (CEK, E_KEK(Z), KeyId, Locator), where each element is: the content encryption key, the CEK precursor, Z, encrypted with the KEK (an RSA operation), and the KeyId of the

corresponding KDK, and the Locator is the CCNx name prefix to fetch

the KDK (see below). The precursor Z is chosen randomly z < n-1, where n is KEK's public modulus. Note that CEK = KDF(Z). Note that the publisher does not see KEK or Z.

We use HKDF (RFC 5869) for the KDF. CEK = HKDF-Expand(HKDF-Extract(0, Z), "CEK", KeyLen), where KenLen is usually 32 bytes (256 bits).

In the ABNF grammar, the RsaKemData includes a KeyId, IV, Mode, WrappedKey, and LocatorPrefix. The KeyId is the ID (sha256) of the KEK. The IV and Mode are as per preshared key, and describe how the manifest is encrypted with AES-GCM. The WrappedKey is the AES key to decrypt the manifest. The LocatorPrefix is used to construct an Interest to fetch the KDK.

To fetch the KDK, a consumer with public key PK_c constructs an Interest with name /LocatorPrefix/{KeyId}/{PK_c keyid} and a KeyIdRestriction of the KM's KeyId (from the LocatorPrefix Link). It should receive back a signed Content Object with the KDK wrapped for the consumer, or a NAK from the KM. The payload of the ContentObject will be RsaKemWrap(PK, KDK). The signed ContentObject must have a KeyLocator to the KM's public key. The consumer will trust the KM's public key because the publisher, whom the consumer trusts, relayed that KeyId inside its own signed Manifest.

The signed Content Object should have an ExpiryTime, which may be shorter than the Manifest's, but should not be substantially longer than the Manifest's ExpiryTime. The KM may decide how to handle the Recommended Cache Time, or if caching of the response is even permissible. The KM may require on-line fetching of the response via a CCNxKE encrypted transport tunnel.

```
RsaKemWrap(PK, K, KeyLen = 256):
    choose z < n-1, where n is PK's public modulus
    encrypt c = z^e mod n
    prk = HKDF-Extract(0, Z)
    kek = HKDF-Expand(prk, "RsaKemWrap", KeyLen)
    WK = E_KEK(K) # [AES-WRAP, RFC 3394]
    output (c, WK)
```

Figure 6: RSA KEM Wrap

A consumer must verify the signed content object's signature against the Key Manager's public key. The consumer then unwraps the KDK from the Content Object's payload using RsaKemUnwrap(). The KeyLen is taken from the WrapMode parameter.

```
RsaKemUnwrap(SK, c, WK, KeyLen = 256):
    Using the consumers private key SK, decrypt Z from c.
    prk = HKDF-Extract(0, Z)
    kek = HKDF-Expand(prk, "RsaKemWrap", KeyLen)
    K = D_KEK(WK) # [AES-UNWRAP, RFC 33940]
    output K
```

Figure 7: RSA KEM Unwrap

The consumer then unwraps the CEK precursor by using the KDK to decrypt Z. It then derives CEK as above. Manifest encryption and decryption proceed as with PresharedKey, but using the CEK.

3.8.3. RSA KEM-DEM

In this scheme a Key Manager (KM), who could be the publisher, creates a Key Encryption Key (KEK) and Key Decryption Key (KDK) key pair. The publisher obtains the KEK. The KM distributes the KDK to each group member by encrypting it under each member's public key. To encrypt data, the publisher generates a symmetric Content Encryption Key (CEK), wraps it with the KEK, then encrypts the manifest with the CEK. It places the wrapped CEK in the manifest.

The KM communicates the KEK to the publisher through an unspecified means particular to the KM.

The KM distributes the KDK to each group member. It uses a name /{km-prefix}/{publisher-prefix}/{KDK KeyId}/{member KeyId} to publish the encrypted KDK under a member's public key. It uses RSA-OAEP for the encryption.

The publisher creates a random symmetric CEK of an appropriate bit length. It uses the KEK to wrap the CEK using RSA-OAEP. It places the wrapped key in the manifest's RsaKemDemData along with the KeyId set to the KDK's KeyId and the KeyLocator prefix /{km-prefix}/{publisher-prefix}/. Each member appends the KDK KeyId and their public key KeyId to the name to attemt to fetch the KDK. When forming the Interest to fetch the key, a consumer should also use a KeyIdRestriction of the KM's KeyId, which it can retrieve from the KeyLocator.

3.9. Protocol Encodings

3.9.1. CCNx Encoding

In CCNx, all Manifest content objects use a PayloadType of T_PYLDTYPE_MANIFEST, while all application data content objects use a PayloadType of T_PYLDTYPE_DATA.

```
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_PYLDTYPE_MANIFEST ; Value TBD
Payload : TYPE LENGTH *OCTET ; the serialized Manifest object
```

Figure 8: CCNx Embedding Grammar

3.9.1.1. CCNx Hash Naming

The Hash Naming namespace uses CCNx nameless content objects.

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 HashSchema. It's 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

The Single Prefix schema uses the same name in all Content Objects and distinguishes them via their ContentObjectHash. Note that in CCNx, using a SinglePrefix name means that we do not use Locators.

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 SinglePrefix and the SinglePrefixSchema element specifies the SinglePrefixName.

- The Top Manifest has the name SinglePrefixName. 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 SinglePrefixName, possibly with cache control directives. It's 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

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 we do not use Locators. OPTIONAL: Use AnnotatedPointers to indicate the segment number of each hash pointer to avoid needing to infer the segment numbers.

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 will track the chunk number of each content object within the NsId. Objects will be numbered in their traversal order. Within each manifest, the name will 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. It's 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 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 are named as appropriate for their 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 ForwardingHints.

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. It's 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. It's GroupData elements must have an NsId that references the NsDef.

- An Internal or Leaf manifest has the name SinglePrefixName, possibly with cache control directives. It's 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.

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 will track the segment number of each Data object within a SegmentedPrefix NsId. Data will be numbered in their traversal order. Within each manifest, the name will be 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. It's 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. It's 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 are named as appropriate for their naming schema.

3.10. Example Structures

3.10.1. Leaf-only data

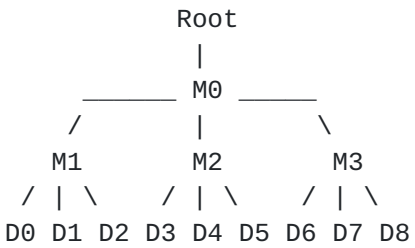

Figure 9: Leaf-only manifest tree

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. Usage Examples

4.1. 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 10: 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 11: 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 12: Using Locators

4.2. 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, 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
```

Figure 13: 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_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)
```

Figure 14: 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.

4.3. Block-level de-duplication

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 [1] archival file system of Plan9 uses this technique.

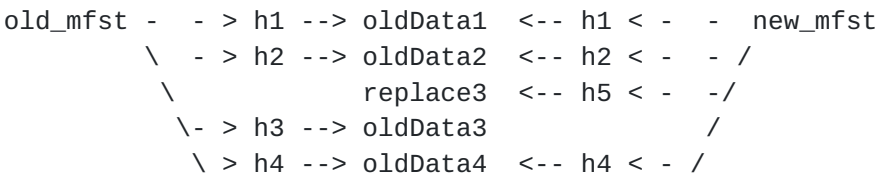

Figure 15: De-duplication

4.4. 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).

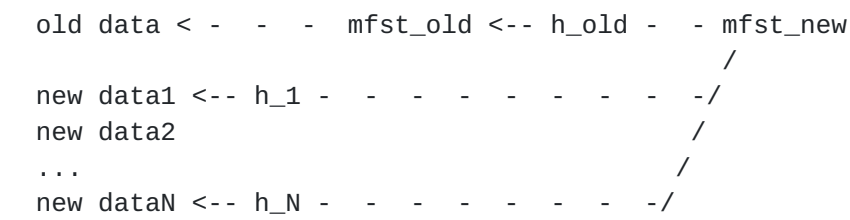

Figure 16: Growing A Collection

4.5. Re-publishing a FLIC under a new name

There are several usecases 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 caches and would like a local result to appear. For example, the publisher /alpha wishes to place content at /beta and /gamma, but routing to /alpha would not send a request to either of those sites. Each of /beta and /gamma could create a locally named and signed version of the root manifest with appropriate keys (or delegate that to /alpha) so the results are always local without having to change the bulk of the maniest 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):

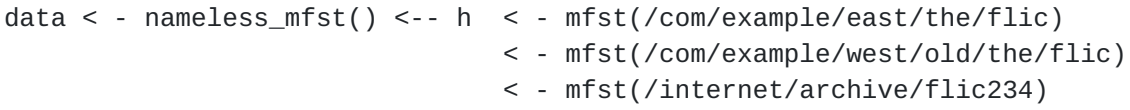

Figure 17: Relocating A Collection

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 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.

5. IANA Considerations

TODO Need IANA actions:

- Create a registry for Manifest Data and Annotation TLVs
- Register the SizeAnnotation TLV

Also TODO: If this document is submitted as an official RG draft, this section must be updated to reflect the IANA registries described in [RFC8609]

6. Security Considerations

TODO Need a discussion on:

- signing and hash chaining security.
- republishing under a new namespace.
- encryption mechanisms.
- encryption key distribution mechanisms.

7. Appendix A: Building Trees

This section 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:

```
         1
     2      3
   4  5    6  7
   preorder: 1 2 4 5 3 6 7
   reverse:  7 6 3 5 4 2 1
```

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 application
payload.
    # 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() <= 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(m, segment.tail - segment.head)
    segment.head += reserve_count

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

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>.

8.2. Informative References

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

[I-D.irtf-icnrg-terminology]
Wissingh, B., Wood, C., Afanasyev, A., Zhang, L., Oran, D., and C. Tschudin, "Information-Centric Networking (ICN): CCNx and NDN Terminology", draft-irtf-icnrg-terminology-07 (work in progress), November 2019.

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

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

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

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, <https://www.rfc-editor.org/info/rfc3552>.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", RFC 5226, DOI 10.17487/RFC5226, May 2008, <https://www.rfc-editor.org/info/rfc5226>.

[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>.

Authors' Addresses

Christian Tschudin
University of Basel

Email: christian.tschudin@unibas.ch

Christopher A. Wood
University of California Irvine

Email: woodc1@uci.edu

Marc Mosko
PARC, Inc.

Email: marc.mosko@parc.com

David Oran (editor)
Network Systems Research & Design

Email: daveoran@orandom.net