parc®
A Xerox Company

Computing Science Laboratory

CCNx 1.0 Protocol Architecture

Marc Mosko[1], Ignacio Solis[1], Ersin Uzun[1], Christopher Wood[1]

Abstract

In contrast to today's IP-based host-oriented Internet architecture, Information-Centric Networking (ICN) emphasizes content by making it directly addressable and routable. Content-Centric Networking (CCNx) is an instance of ICN designed by PARC as a reference implementation that works in many environments from high-speed data centers to resource constrained sensors. By flexibly caching content within the network (in routers) and providing message-based, as opposed to channel-based security, CCNx is well-suited for large-scale content distribution, the Internet of Things and for meeting the needs of increasingly mobile and bandwidth-hungry applications that dominate today's Internet. This paper describes the CCNx 1.0 architecture and protocols, from message semantics to wire format to transmission rules.

[1] *Computing Science Laboratory, PARC*

Contents

1. Introduction

The functional goal of the Internet Protocol as conceived and created in the 1970s was to enable two machines, one comprising resources and the other desiring access to those resources, to have a conversation with each other [1]. The operating principle was to assign addresses to end-points, thereby enabling these end-points to locate and connect with one another. Since those early days, there have been fundamental changes in the way the Internet is used — from the proliferation of social networking services to viewing and sharing digital content such as videos, photographs, documents, etc. Instead of providing basic connectivity, the Internet has evolved largely into a distribution network with massive amounts of video and web page content flowing from content providers to viewers. Internet users of today are demanding faster, more efficient, and more secure access to content without being concerned with where that content might be located.

To address the Internet's modern-day requirements with a better fitting model, PARC has created a new networking architecture called Content-Centric Networking (CCNx), which operates by addressing and delivering Content Objects directly by name instead of merely addressing network end-points. In addition, the CCNx security model explicitly secures individual Content Objects rather than securing the connection or "pipe". Named and secured content resides in distributed caches automatically populated on demand or selectively prepopulated. When requested by name, CCNx delivers named content to the user from the nearest cache or content provider, thereby traversing fewer network hops, eliminating redundant requests, and consuming fewer overall resources.

CCNx is based on a few simple key tenets.

- **Access content by name, not machine address**: Networking and communications are better served by using **names** to access information as opposed to using machine addresses. Packets have no source or destination address, the name identifies a request and response.
- **Secure the content, not the connection**: Security must

be the foundation of any network architecture. Securing the data is more important (and useful) than securing the connections. In CCNx, Security does not rely on secure communication end-points or secure communication channels.

- **Add computing and memory into the network**: Computation and memory continue to decline in cost, making it feasible to add computation and memory to routers and, as a result, maintain state and object stores. CCNx may cache content throughout the network essentially creating a fully-managed peer-to-peer network without losing control of content.

A reference implementation of PARC's CCNx 1.0 is available from the CCNx website [2].

In the remainder of this document, we describe the protocols, mechanisms, methods, and formats used in CCNx 1.0 to deliver an efficient, flexible, ICN implementation.

2. Architecture Overview

The CCNx 1.0 architecture comprises a protocol relying on a set of well formatted basic messages, name formats, and packet formats. Higher-level value-adding protocols are layered upon the core protocol to provide additional functionality such as large object chunking, cache control, versioning etc. Definitions for the core protocols and several higher level protocols may be found on the IETF data tracker site [3] and on the CCNx website [4]. It is expected that additional protocols will be added by others in the research community.

A CCNx 1.0 network is populated by content producers, content publishers, and content consumers with traffic supported by Forwarders and in-network caches called Content Stores. Each of these actors fulfill specific roles. A content producer creates user data. This could be an individual taking photos, a family talking on the phone, a sensor producing a reading, or a media corporation creating movies. A content publisher converts user data into network objects with associated cryptographic identities, and publishes the item to the network using the CCNx 1.0 protocols. A content consumer uses the CCNx 1.0 protocols to retrieve publisher authenticated user data.

Content is published and located by Publisher-given Names, not physical addresses (see Section 3 for more detail). For very large pieces of content, the publisher may choose to chunk the content (see Section 9.1) or use Manifests (objects describing a collection of content - see Section 9.2).

The core protocol uses two message types: Interest and Content Object. An Interest message is used to request data from the network. The data is returned in the form of a Content Object. An Interest message is forwarded along routing paths, leaving reverse-path state with each Forwarder. A Content Object is a chunk of authenticated user data sent back along the reverse path of the Interest, consuming the state left by the Interest. Section 4 describes these two messages in detail and specifies how they are processed by nodes in the network.

Every piece of content on a CCNx network carries with it the means to verify its origin, integrity, and authenticity. Moreover, every Interest may carry optional trust-related information to help Forwarders make informed decisions about which content to trust and which content to quickly discard. Thus, content verification and trust management are tightly coupled, as will be shown in Section 6

CCNx 1.0 uses a nested Type-Length-Value (TLV) format to encode all messages on the wire as described fully in Section 8. This is a new general packet format comprising a fixed header, a set of hop-by-hop headers, and a CCNx protocol message.

The two key network elements in CCNx 1.0 are Forwarders and in-network caches called Content Stores. A Forwarder uses the Forwarding Information Base (FIB) table for routing and the Pending Interest Table (PIT) to record Interest state such that Content Objects may follow the reverse Interest path. Optional Forwarder Content Object caches called Content Stores are used to temporarily store Content Objects. Using a temporary Content Object store allows the Forwarder to repair lost packets if a downstream node requests the same Content Object or to reduce upstream traffic by caching popular content. These are fully described in Section 7.

3. Names

A core feature of CCNx 1.0 is that units of data, called Content Objects, have Publisher-given Names. The Name serves two purposes: it is used to match against entries in the Forwarding Information Base (FIB) to forward the Interest message in its search for Content Objects, and it is used to identify the correct Content Object when it is found.

In many respects, a CCNx Name looks like a routable URI as shown in Name (1) below. A CCNx name is an absolute URI without an authority (hostname). Following the nomenclature of URIs [5], we use the *scheme* "lci" and the *hier-part* is a *path-absolute* [5, Sec 3.3]. The absolute path is composed of a "/" followed by zero or more *segments*. CCNx 1.0 follows the convention that each URI name segment has a label and a value, which leads to the scheme name Labeled Content Identifier (lci). The label identifies the purpose of the segment.

Currently, three types of Name segments are supported: generic Name segments, Interest Payload ID segments, and application-specific Name segments which may encapsulate application-specific Payload in the Name. The Interest Payload ID is an identifier that represents the Interest Payload field. As an example, the Payload ID might be a hash of the Interest Payload. This provides a way to differentiate between Interests based on their payloads without having to parse all the bytes of the Payload itself; instead using only this Payload ID Name segment. If no label is present, it is assumed to be a general name segment. Other protocols such as Chunking (see [6]) and Serial Versioning (see [7]) add Name segment types to support their functionality as shown below.

(1) `lci:/Name=foo/Name=bar/SerialNumber=7/ChunkNumber=30`

For example, in Name (1), the prefix `lci:/Name=foo/Name=bar` corresponds to a publisher-given Name. The segment `SerialNumber=7` indicates the revision of the document using a serial number. The segment `ChunkNumber=30` indicates that the user data was split into multiple chunks with this chunk being the 30th in the series. These labels, shown in human-readable URI form, are encoded to TLV types. Labels and names may be binary, rather than human-readable. The current specification for Names may be found at [8].

4. Network Messages

This section describes the CCNx 1.0 messages used to transfer user data: a request message called an Interest, a response message called a Content Object, and an Interest Return message returned by network elements to indicate Interest processing errors.

4.1 Interest Message

An Interest carries a Name, an optional publisher identifier called the KeyId, and an optional Content Object identifier called a ContentObjectHash - a secure cryptographic hash of a specific Content Object message.

While Interest looping is not prevented in CCNx, an Interest traversing loops is eventually discarded using the hop-limit field of the Interest. CCNx uses a 1-byte HopLimit, so at most 255 hops are allowed. When a Forwarder receives an Interest with a HopLimit, it decrements the value. It may not forward an Interest with a HopLimit of 0 out an external interface; it may only satisfy it from the built-in Content Store or forward it to a local application.

An Interest also carries a few directives that affect how it should be handled by the network. The Lifetime field specifies an upper limit on how long the application is willing to wait for a response. The requester should not need to re-express the Interest before the Lifetime has elapsed, unless it receives a NACK. The network, however, may store unanswered Interests for less than the specific Lifetime.

An Interest message sometimes carries state. For example, a Name segment could encode data such as a session identifier. In this case an application-specific type known to the application as "SessionId" could be added as a Name segment to identify the session, for example:

(2) `lci:/Name=foo/Name=bar/App:SessionId=0x5512334`

A Name segment could also identify a computation task to be done by the authoritative source, such as retrieving the current account balance. In this case, the Interest message is used to trigger behavior at a source and the computation result is returned in the Content Object.

(3) `lci:/Name=foo/Name=bar/App:SessionId=0x5512334/App:Task=AccountBalance`

An Interest may carry even more complex state. One could serialize a complex data structure into a Name segment. For example, a JSON data record could be encoded as a string in a Name segment, or a binary record could be included as-is in a binary path segment as in:

(4) `lci:/Name=foo/Name=bar/App:State={SessionId:"0x5512334",Task:"AccountBalance",...}`

An Interest message typically has a Message Integrity Check (MIC) Validator appended to it. The Validator specifies the ValidationAlgorithm (e.g., CRC32C) and ValidationPayload (i.e., the checksum). This allows end-to-end validation of unauthenticated messages. An Interest message could have a stronger Validator, such as a Message Authentication Code (MAC) or digital signature.

4.2 Content Object Message

A Content Object carries a Name and a Payload. Commonly, a Validator comes along with the Content Object. The Validator includes the Validation Algorithm used and the Validation Payload (e.g. a signature or HMAC) produced against which to validate. The Validation Algorithm also carries a KeyId, which identifies the publisher authenticating the Content Object. If an Interest carries a KeyId in addition to a Name, it limits the universe of Content Objects that match the Name to those validated with that KeyId. The ContentObjectHash is a cryptographic hash of the entire Content Object. If used in an Interest in addition to a Name, it selects the one specific Content Object of that Name and cryptographic hash.

The ContentObjectHash is the SHA-256 hash of the Content Object's wire format, from the opening Content Object tag to the end of the Content Object including the Validation Algorithm, but not the Validation Payload. It does not include the fixed or optional packet headers. The ContentObjectHash is not an explicit field in the packet; it must be calculated. This calculation provides the assurance that, for correctly behaving nodes within the network, if a user requests a Content Object by hash, the network delivers the correct packet.

A Content Object may not include a Validator or it may use a Validator that is only an integrity check (e.g. CRC32C). Such a Content Object should only be retrieved by an Interest with a specified ContentObjectHash. This type of "bare" Content Object is appropriate when used in conjunction with a Manifest, where the Manifest is a full Content Object with a KeyId and cryptographic signature and enumerates a set of bare Content Objects. The bare Content Objects, without the overhead of the KeyId or Signature, may consist of only a small Name and Payload and thus be pre-generated to a network Maximum Transmission Unit (MTU) and not require fragmentation.

4.3 InterestReturn Message

An Interest Return message may be returned by a network element to a previous hop if there is an error processing the Interest. The returned Interest may be further processed at the previous hop or returned towards the Interest origin. When a node returns an Interest it indicates that the previous hop should not expect a response from that node for the Interest – i.e. there is no PIT entry left at the returning node.

5. Matching

Matching in CCNx include two operators: Equals and ComputeContentObjectHash. These two operators work on the three fields of an Interest message to match against a Content Object. One Interest message receives at most one Content Object message, so there is flow balance in messages.

For a Content Object to match an Interest, the Name of the Content Object must be equal to the Name of the Interest. If the Interest has a KeyIdRestriction, then it must exactly equal the KeyId of the key that validates a Content Object (this is not a cryptographic operation). If the Interest carries a ContentObjectHashRestriction, then the Forwarder must compute the SHA-256 digest of the Content Object and match it for equality to the Interest's restriction.

6. Verification and Trust Management

Both Interests and Content Objects may optionally carry information about how to validate the CCNx message. This information is contained in two sections of the message: the ValidationAlgorithm section and the ValidationPayload section. The ValidationAlgorithm field specifies the verification mechanism to be used and any validation dependent data (VDD) needed to perform the verification. For example, if digital signature verification is needed to verify a Content Object, then the ValidationAlgorithm field will specify the digital signature algorithm (e.g., RSA, DSA, ECDSA, etc.) and related parameters (e.g., key size, mode of operation, etc.). The VDD will then contain the public key (possibly in a certificate) used for verification or a pointer (CCNxLink, CCNxName) to obtain said key. The ValidationPayload section contains the validation output, such as the CRC32C code or the RSA signature. Validation is computed over the CCNx Message portion of the packet through the Validation Algorithm section (see (Fig. 4), Validation Range).

There are three types of validation algorithms: Message Integrity Checks (MIC), private-key Message Authentication Codes (MAC), and public-key signatures. A MIC is the weakest form of verification, typically only used in Interests. This requires no additional keys or information and simply serves as a form of error detection. It is easily forgeable, and therefore should not be used when strong authenticity guarantees are required. MACs are useful for communication between two trusting parties who have already shared private keys. Any CCN-compliant key exchange protocol may be used to establish these private keys. Moreover, a Content Object should include a KeyId of the corresponding private key that is used to verify the MAC at the consumer.

Lastly, a signature type Validator specifies a digest mechanism and a signing algorithm to verify the message. Examples include RSA signatures, Elliptic Curve signature with SECP-256K1 parameters, etc. Signature validators require a KeyId so consumers can identify the correct public key to use when verifying content, as well as a mechanism for locating the publisher's public key. Once the key is retrieved, its KeyId is matched against the8 KeyId provided in the content object. The publisher's public key may be included directly in the VDD, may be embedded in a certificate in the VDD, or may be pointed to by a PublicKeyLocator or KeyName in the VDD.

In order for public key verification to be effective, the consumer has to be able to trust the public key they have been given. In the case of a direct key or a key retrieved via a PublicKeyLocator or KeyName, the consumer needs to check whether or not the key is in its collection of trusted keys, or trusted hashes of keys; in the case of a certificate, the user will have to traverse the certificate hierarchy until s/he finds a trusted root.

In some cases, a consumer knows exactly which Content Object they want. In this case, they send an Interest with a ContentObjectHashRestriction. Technically, no verification is required when this is returned since the ContentObjectHashes are exactly the same; however, we note that a consumer should always verify the packet, whereas routers may not always perform this verification.

Finally, the router's Content Store should validate all Content Objects prior to them being served in response to incoming interests. This means that verification can be done when content is initially cached or when content is served from the cache for future Interest messages.

7. Forwarding

A Content Consumer issues an Interest request for data over the network. The Interest is transmitted through a set of Forwarders until the Content Object is found or the Interest's Lifetime elapses.

A Forwarder consists of three notional tables: Forwarding Information Base (FIB), Pending Interest Table (PIT), and Content Store (CS). An actual implementation may use a different information organization as long as the collective behavior is the same. The FIB is a longest-matching-prefix name table populated by a routing protocol or with static routes. The Forwarder records Interest state in the PIT such that Content Objects that satisfy the Interest may follow the reverse Interest path back to the requester as shown in Fig. 1.

A Forwarder implements Interest aggregation, so that multiple similar Interests do not get forwarded upstream. Aggregation is based on Interests that have the same Name, KeyIdRestriction (if present) and ContentObjectHashRestriction (if present) and that fit within the same Interest Lifetime, where InterestLifetime is defined as the amount of time the consumer is willing to wait for the Content Object response.

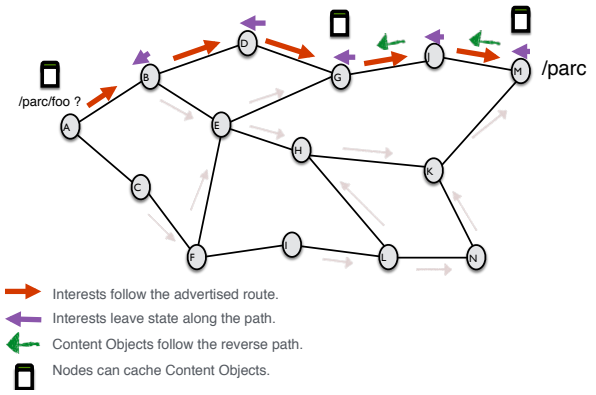

Figure 1. Forwarder Behavior

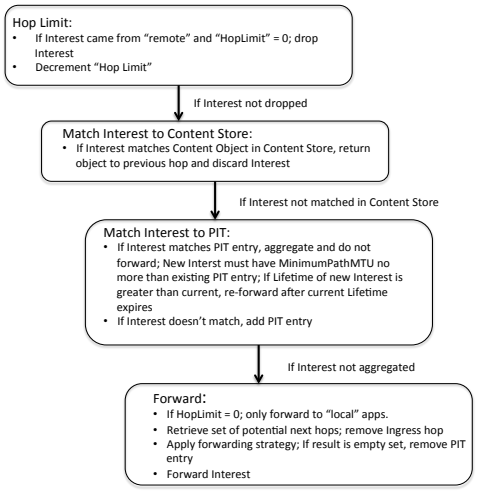

Figure 2. Forwarder Behavior: Receive Interest

If a new, similar, Interest comes from the same previous hop then:

- if the new Lifetime would extend PIT entry lifetime, update the PIT entry to the max of (old Lifetime, new Lifetime) and forward.
- if the new Lifetime would not extend PIT entry, forward (no PIT update required).

If a new, similar, Interest comes from a different previous hop then:

- if the new Lifetime would not extend the PIT entry, update PIT with new previous hop and don't forward.
- if the new Lifetime would extend the PIT entry, update PIT entry to the max of (old Lifetime, new Lifetime), add the new previous hop, and forward.

The Content Store is an optional in-network Content Object cache that may be used to satisfy Interest messages with fewer hops. Depending on its role in the network, a Forwarder may also run additional protocols such as service and content discovery protocols.

As a Content Object moves through the "fast path" of the network, it matches pending Interests at each node according to the following rule:

Satisfy Interest: *A Content Object satisfies an Interest if and only if (a) the Content Object name exactly matches the Interest name, and (b) the Content Object Hash equals the Interest Content Object hash restriction, if given, and (c) the Content Object KeyId exactly equals the Interest KeyId restriction, if given.*

Only end systems (or Content Stores) need to verify cryptographic signatures, which decreases the computational load on each node and therefore increases throughput. However, each hop may need to compute the ContentObjectHash, if the pending Interest includes a ContentObjectHashRestriction.

The Content Store has more restrictive matching rules in order to prevent persistent failures resulting from incorrect content getting into the persistent store:

Content Store Matching: *If an interest has a Content Object Hash restriction, then the Content Store must verify a content object's hash prior to returning it to the requester. If an Interest has a KeyId restriction, then the Content Store must verify a cached object's signature before returning it to the requester. This means that either the Interest must carry the desired public key or the Content Object must carry its own public key.*

The CCN 1.0 node behavior is as follows. The CCN packet (Fig. 4) arrives either in direct layer 2 encapsulation – such as an Ethernet frame with a specific CCN ethertype – or via a layer 3 tunnel. The Forwarder determines which processing rules to use based on the Fixed Header PacketType field (Fig. 4).

CCN distinguishes between "local" and "remote" next hops. A local next hop is a directly attached application running locally on the system. A remote next hop is not local to the current system. These conventions along with how we use HopLimit result in expected behavior. If a local application sends an Interest with a HopLimit 0, that Interest will only go to other applications on the system. If it sends an Interest with a HopLimit 1, it will go to local applications plus the 1-hop neighbors of the system. If a system receives a remote Interest with a HopLimit 1, it will be decremented to 0 and then only forwarded to local applications.

As shown in Fig. 2, a Forwarder goes through a set of steps to determine the appropriate action to take upon receiving an Interest message:

- It examines (and decrements) the HopLimit of the incoming Interest to determine whether the Interest may be forwarded to remote or local nodes.

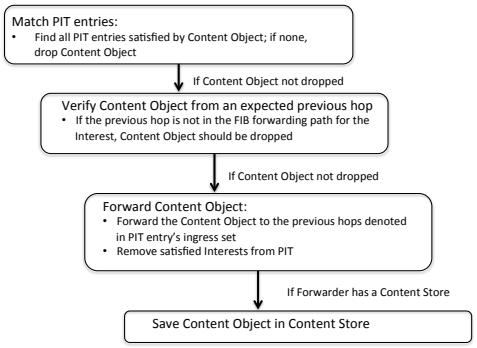

Figure 3. Forwarder Behavior: Receive Content Object

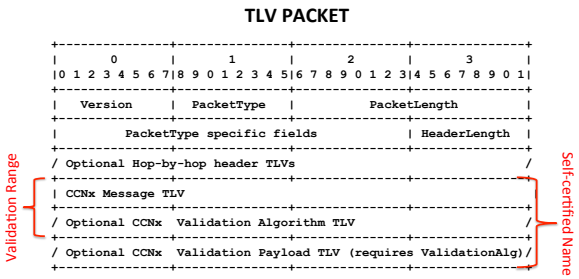

Figure 4. Overall packet format

- It checks the ContentStore (if present) for a matching Content Object.
- It checks the PIT to see if Interest aggregation is possible. If not, it adds a new PIT entry.
- It forwards the Interests appropriately to the set of next hops.

As shown in Fig. 3, a Forwarder goes through a set of steps to determine the appropriate action to take upon receiving an Content Object message:

- It checks the PIT to find all entries satisfied by the Content Object (see Definition 1). If none are satisfied, it drops the Content Object.
- It forwards the content Object to the previous hops denoted in the PIT entry's ingress set and removes the satisfied PIT entries.
- If the node has a Content Store, it may save the Content Object in the Content Store.

8. Packet Format

CCNx 1.0 uses a TypeLengthValue format to encode messages into a TLV Packet. This is a new general packet format based on a fixed header, a set of hop-by-hop headers in TLV format, and a CCNx message containing nested TLV encoded content. The Type and Length fields are both exactly two bytes. Using a fixed type and length size avoids issues with aliases and is simple for high speed equipment to parse. In this section, we describe the TLV format, the fixed packet header, and the skeletons of Interest and Content Object messages.

Each CCNx 1.0 packet, as shown in Fig. 4, begins with a fixed header followed by a set of hop-by-hop headers in TLV format. The hop-by-hop headers are not considered part of the CCNx message, but rather communicate in-network state such as a DSCP class. The headers are followed by the CCNx message - an Interest message, a Content Object message, an InterestReturn, or any future message types. The CCNx message is followed by optional ValidationAlgorithm and ValidationPayload TLVs. The ValidationAlgorithm TLV specifies the mechanism to use to verify the CCNx message. Examples include verification with a Message Integrity Check (MIC), a Message Authentication Code (MAC), or a cryptographic signature. The ValidationPayload field contains the validation output, such as the CRC32C code or the RSA signature. A full specification of the CCNx 1.0 Packet Format may be found at [9].

Following the CCNx Message with optional ValidationAlgorithm and ValidationPayload fields has several desirable features. In this scheme, the validation of a CCNx message is modular and the same for all packet types. It is important that all messages, including Interests, carry at least a MIC. In studies on TCP, UDP, and DNS messages [10, 11], researchers observe a packet error rate (PER) on the order of 10^{-5}, which is far greater than what one would expect from an Ethernet Mean Time To False Packet Acceptance (MTTFPA), which is usually at least as long as the age of the universe for properly functioning hardware and cabling. Therefore, we allow Interests and other messages to carry any one of a MIC, MAC, or Signature for transport-level validation.

The Fixed Header begins with the Version field which indicates the overall protocol version for the TLV encoding. The PacketType field indicates the type of packet (or CCNx message) that follows the hop-by-hop headers. Current options for a PacketType are 0 (an Interest message), 1 (a Content Object Message) or 2 (an InterestReturn message). The HopLimit field is used in Interest messages as a hop limit to prevent loops. HeaderLength is the number of bytes of header, including the fixed and all hop-by-hop headers. The CCNx Message begins at HeaderLength into the overall packet.

Fig. 5 shows an Interest message in TLV format. The value T_INTEREST is a protocol constant used to identify an Interest message. The value T_NAME is a protocol constant used to identify a CCNx 1.0 Name, which is a nested TLV structure. Each message type carries optional message TLVs. The optional KeyIdRestriction and ContentObjectHashRestriction are as described previously. The Content Object Hash of a Content Object is the SHA-256 hash of the CCNx Message, ValidationAlgorithm, and ValidationPayload.

An Interest may also have a Payload. The Payload carries state about the the Interest, but is not directly used to

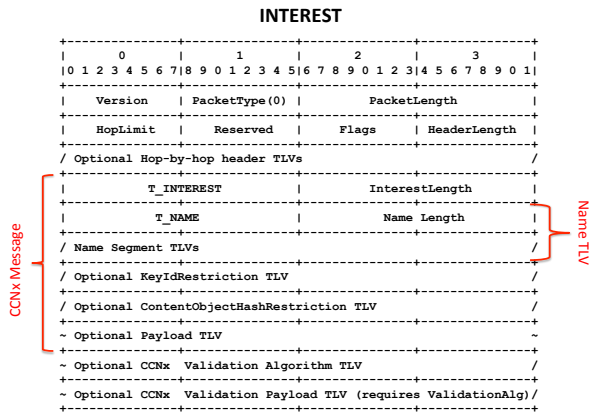

Figure 5. TLV Interest

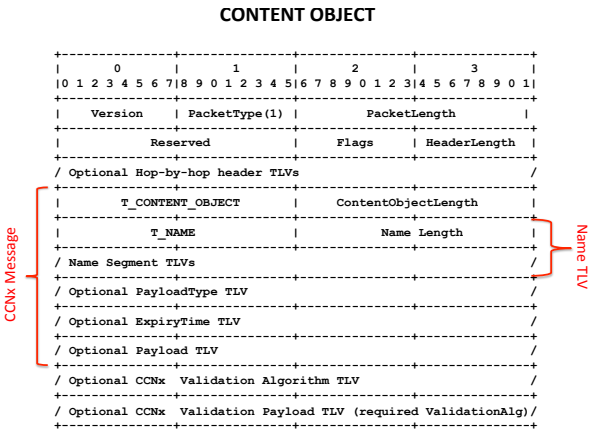

Figure 6. TLV Content Object

match a Content Object. The Interest Payload carries data that a publisher might use to generate a Content Object or information to help select among multiple Content Objects that would otherwise match. It is important that an Interest with a Payload have a Name that is specific to that payload – otherwise it may be aggregated with other Interests with the same Name. We suggest that a Name segment be set to the hash of the Interest Payload to differentiate payloads.

Fig. 6 shows a TLV encoded Content Object. The field ContentObjectLength is the length of the Content Object through the payload; it does not include the Validation TLVs. The packet begins in the same way as an Interest. It starts with a fixed header, a set of hop-by-hop headers in TLV format, the message container (T_CONTENT_OBJECT) and the length of the unsigned Content Object. The first field in the unsigned Content Object is the Name TLV. The remaining TLV containers are specific to a Content Object. Current optional TLVs specific to a ContentObject include a PayloadType TLV indicating the type of Payload contents (for example data (default), key, link, or manifest), and an ExpiryTime which expresses the time at which the Payload expires. The Payload TLV contains an opaque value that is the Payload of the Content Object.

Fig. 7 shows the ValidationAlgorithm TLV. The type (Val-

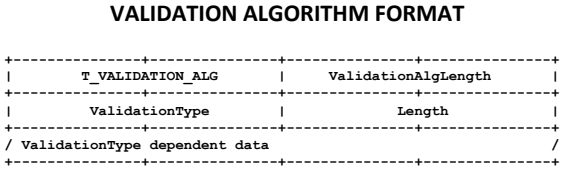

Figure 7. ValidationAlgorithm TLV Format

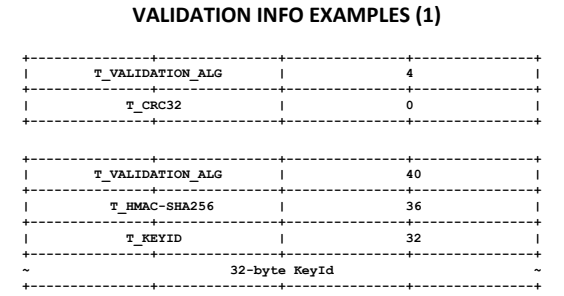

Figure 8. ValidationAlgorithm TLV Format Examples

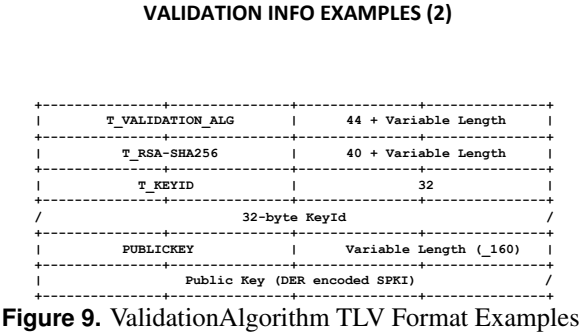

Figure 9. ValidationAlgorithm TLV Format Examples

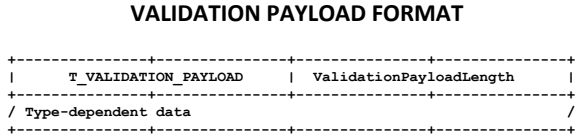

Figure 10. Validation Payload TLV Format

idationType) defines the type of validator. Each MIC, MAC, or signature has its own TLV type, which in turn has requires its own validation dependent data. The use of CRC32, as in Ethernet, is not suggested because it duplicates the link layer Frame Check Sequence; we recommend using a different MIC though the analysis of what would make the best compliment is beyond the scope of the present work. Fig. 8 shows examples for a CRC32 and an HMAC with SHA-256 hash. The CRC32 MIC requires no additional specification, so it has a 0 length. The HMAC-SHA256 MAC requires a KeyId parameter so the receiver knows which shared key to use. Fig. 9 shows an example of a Signature type validation using an RSA public key signing with a SHA256 digest and public key. Fig. 10 shows an example of a Validation Payload.

9. Large Object Options

Publisher data such as large text documents, audio files, and very large movie files, may be too big to fit in a single Content Object or MTU. A Content Object is limited to a maximum of 64 KB by the TLV wire format (see Sec. 8) and a typical MTU for Ethernet is only 1500 bytes. When this occurs, the system will automatically split the data into pieces in order to fit into the network MTU size using a fragmentation process (see Sec. 9.3). However, a more efficient way of handling this is through the use of Chunking (see Sec. 9.1) or Manifests (see Sec. 9.2). The chunking mechanism informs the consumer about the data breakup through the use of chunk name segments and chunking metadata objects while the Manifest structure offers aggregated signing across a set of constituent hashes.

9.1 Chunking

Chunking, as defined in the CCNx Chunking specification (see [6]), means serializing user data into one or more chunks, each encapsulated in a CCNx Content Object. Each data chunk is identified by a Name segment with label T_CHUNK and a value with the chunk number as an unsigned integer in network byte order without leading zeros. The first chunk number is %x00.

As an example, the second Content Object in a stream of chunks for picture.jpg would be:

(5) `lci:/Name=parc/Name=csl/Name=picture.jpg/Chunk=1`

An additional Content Object Message TLV with type = T_ENDChunk is defined for the EndChunkNumber. This may be included in the earliest chunk Content Object in which it is known; it must be included in the last chunk Content Object. Subsequent chunk Content Objects may increase the EndChunkNumber as long as the end has not yet been reached, but it should never decrease the number. An example of the Names for a piece of chunked data is shown in Fig.fig:ChunkNameEx.

```
lci:/Name=parc/Name=csl/Name=picture.jpg/Chunk=0
      Protocol Info: No EndChunk
lci:/Name=parc/Name=csl/Name=picture.jpg/Chunk=1
      Protocol Info: EndChunk="3"
lci:/Name=parc/Name=csl/Name=picture.jpg/Chunk=2
      Protocol Info: No EndChunk
lci:/Name=parc/Name=csl/Name=picture.jpg/Chunk=3
      Protocol Info: EndChunk="3"
```

Figure 11. Chunked Name Examples

A more detailed description of the chunking mechanism including the packet format may be found in [6].

9.2 Manifests

A Manifest may be used to describe the collection of Content Objects that constitute one logical entity. A Manifest is a Content Object with a well-known Payload format and a PayloadType of "Manifest," rather than the default PayloadType of "Data". The Manifest provides meta information about the collection and enumerates the ordered, hash-based names of every constituent piece of the collection. Manifests are hierarchical to enable a single root Manifest to represent an unbounded length Content Object through chaining. The use of hierarchical Manifests means that only the root manifest needs a cryptographic signature as all other Content Objects and subsequent hierarchical Manifests are requested via a Name and a ContentObjectHash, forming a trusted hash chain from the original signature in an aggregated signing mechanism. Manifests allow the amortization of a single public key operation over a very large object thereby keeping the individual constituent Content Object sizes under the MTU and preventing fragmentation. A draft specification for Manifests may be found on the CCNx website [4]

9.3 Fragmentation

A CCNx 1.0 message may be larger than a specific networking technology allows. The limitation, known as the Maximum Transmission Unit (MTU), typically varies from 1280 octets (for some IPv6 tunnels) to 1500 octets (Ethernet) to 9000 octets (Ethernet Jumbo frames), with 1500 octets being the most common. Both Interest and Content Object messages, however, need to accommodate potentially large Names and have key and signing overhead. Additionally, a Content Object may be as large as 64 KB. Therefore, CCNx messages often must be fragmented over specific network media.

End-to-end fragmentation avoids the complexity and buffer space requirements of the hop-by-hop option. It is based on the principle that intermediate systems should not have to fragment packets. A proposed specification for an end-to-end

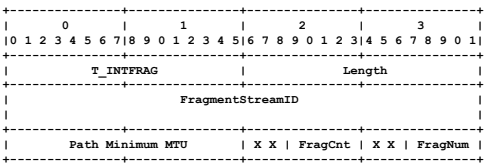

Figure 12. Interest Fragment Header

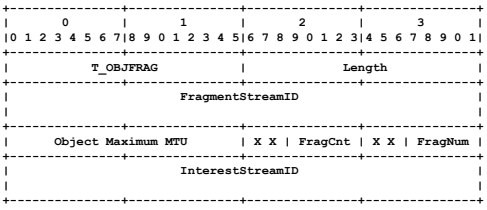

Figure 13. Content Object Fragment Header

fragmentation mechanism may be found at [12]. To achieve this, an Interest is always fragmented to the minimum MTU required by the path and the forward path's MTU is recorded in the Interest, so that a system sending back a Content Object will know what the required fragment size needs to be. An intermediate system's Content Store may store only pre-fragmented objects, responding only if an Interest's MTU is no smaller than that stored, or it may re-assembly complete objects and fragment on demand.

Because the Interest's path serves as the path discovery mechanism for Content Objects, all Interests must carry an InterestFragmentationHeader so the reverse path MTU is known by the node responding with a Content Object. The minimum MTU required is 1280 octets. Any system implementing a physical layer with a smaller MTU must implement link fragmentation, for example using a PPP layer over the small MTU network.

Systems must create fragment streams in the most compressed packing. That is, all fragments except the last must be fragmented to the same size. This means that all Interests are fragmented to 1280 byte fragments, except for the last fragment. A Content Object may be fragmented to any size that is no more than the path MTU discovered, but all fragments except the last must be the same size. This ensures that any two fragment streams of the same Content Object with the same MTU have the same representation.

When an end system creates a fragment stream, it generates a random 64-bit number for the FragmentStreamID. This number identifies a contiguous stream of fragments. A system at the other end uses the FragmentStreamID for reassembly. An intermediate system uses the FragmentStreamID of a Content Object to ensure that only one stream of Content Object fragments follows a reverse PIT entry. If the Maximum Path MTU of a Content Object fragment is larger than the supported MTU on an egress interface, the fragment stream should be dropped on that interface, even if some fragments fit within the MTU. Examples of an Interest Fragment Header and a Content Object Fragment Header are shows in Figs. 12 and 13.

References

[1] J. Postel. Internet Protocol. RFC 791 (INTERNET STANDARD), September 1981. Updated by RFCs 1349, 2474, 6864.

[2] Ccnx downloads. `http://www.ccnx.org/downloads`, 2015.

[3] Ietf data tracker. `https://datatracker.ietf.org`, 2015.

[4] Content centric networking specification. `http://www.ccnx.org/specifications`, 2015.

[5] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (INTERNET STANDARD), January 2005. Updated by RFC 6874.

[6] Content centric networking chunking specification. `http://tools.ietf.org/html/draft-mosko-icnrg-ccnxchunking-00`, 2015.

[7] Content centric networking serial versioning specification. `http://tools.ietf.org/html/draft-mosko-icnrg-ccnxserialversion-00`, 2015.

[8] Labeled content information specification. `https://datatracker.ietf.org/doc/draft-mosko-icnrg-ccnxlabeledcontent/`, 2015.

[9] Content centric networking specification. `https://datatracker.ietf.org/doc/draft-irtf-icnrg-ccnxmessages/`, 2015.

[10] Duane Wessels. Observations on checksum errors in DNS UDP messages. In *Domain Name System Operations Analysis and Research DNS-OARC*, October 2011.

[11] Jonathan Stone and Craig Partridge. When the CRC and TCP checksum disagree. In *SIGCOMM '00*, pages 309–319, 2000.

[12] Content centric networking serial versioning specification. `http://tools.ietf.org/html/draft-mosko-icnrg-ccnxfragmentation-00`, 2015.