# NDN specification Documentation

***Release 0.1a2***

**NDN Project Team**

March 27, 2014

## Contents

# 1 Acknowledgment

This NDN packet format specification is adapted from the CCNx specification described in http://www.ccnx.org/releases/latest/doc/technical/index.html as of October 2013. However we have made a number of packet format changes based on our deepened understanding about NDN design through experimentation with NDN over the last three years, and based on the inputs from NDN research community at large. In particular, this specification adopted a TLV encoding format that has been championed by cisco NDN project team. We list the major protocol format changes from the CCNx specification at the end of each section.

# 2 Introduction

This version 0.1 specification aims to describe the NDN packet format only, a much narrower scope than a full NDN protocol specification. Our plan is to circulate and finalize the packet format first, then write down the full protocol specification.

In addition to this protocol specification draft, we are also in the process of putting out a set of technical memos that document our reasoning behind the design choices of important issues. The first few to come out will address the following issues:

- Packet fragmentation: end-to-end versus hop-by-hop;
- Understanding the tradeoffs of (not) handling Interest selectors;
- NDN Name discovery: why do we need it?
- NDN naming convention; and
- Scaling NDN routing.

In the rest of the document, we assume readers are familiar with how NDN/CCN works in general. For a description of the current CCNx protocol definition, please refer to http://www.ccnx.org/releases/latest/doc/technical/CCNxProtocol.html.

# 3 Type-Length-Value (TLV) Encoding

Each NDN packet is encoded in a Type-Length-Value (TLV) format. NDN Interest and Data packets are distinguished by the type value in the first and outmost $TLV_0$.

An NDN packet is mainly a collection of TLVs inside $TLV_0$. Some TLVs may contain sub-TLVs, and each sub-TLV may also be further nested. A guiding design principle is to keep the order of $TLV_i$s deterministic, and keep the level of nesting as small as possible to minimize both processing overhead and chances for errors.

Note that NDN packet format does not have a fixed packet header nor does it encode a protocol version number. Instead the design uses the TLV format to provide the flexibility of adding new types and phasing out old types as the protocol evolves over time. The absence of a fixed header makes it possible to support packets of very small sizes efficiently, without the header overhead. There is also no packet fragmentation support at network level. Whenever needed, NDN packets may be fragmented and reassembled hop-by-hop. [1]

## 3.1 Variable Size Encoding for type (T) and length (L)

(Both the text below and that in *TLV encoding section* (page 4) are adopted from an earlier packet specification draft by Mark Stapp)

To minimize the overhead during early deployment and to allow flexibility of future protocol extensions to meet unforeseeable needs, both type (T) and length (L) take a variable size format. For implementation simplicity, both type and length take the same encoding format.

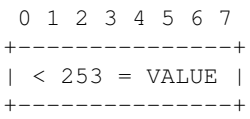
We define a variable-length encoding for numbers in NDN as follows:

```
VAR-NUMBER := BYTE+
```

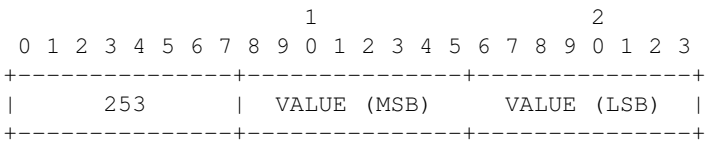
The first octet of the number either carries the actual numeric value, or signals that a multi-octet encoding is present, as defined below:

- if the first octet is < 253, the number is encoded in that octet;
- if the first octet == 253, the number is encoded in the following 2 octets, in net byte-order;
- if the first octet == 254, the number is encoded in the following 4 octets, in net byte-order;
- if the first octet == 255, the number is encoded in the following 8 octets, in net byte-order.

One-octet value:

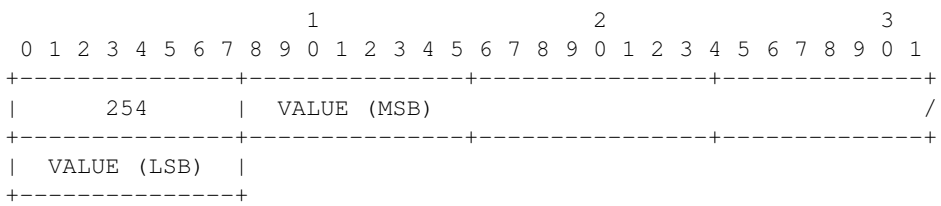



Two-octet value:

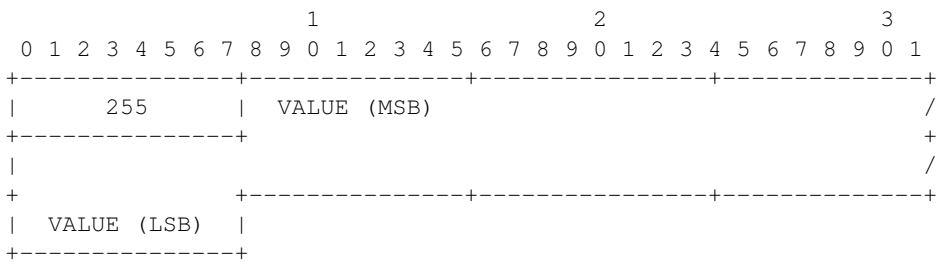



Four-octet value:

[1] Today's IP networks provide point-to-point packet delivery and perform end-to-end fragmentation. An NDN network, on the other hand, may fetch requested data from any in-network storage, thus the notion of data flowing along an end-to-end path does not apply.




Eight-octet value:




## 3.2 TLV Encoding

TLV encoding for NDN pakcets is defined as follows:

```
NDN-TLV := TLV-TYPE TLV-LENGTH TLV-VALUE?
TLV-TYPE := VAR-NUMBER
TLV-LENGTH := VAR-NUMBER
TLV-VALUE := BYTE+
```

TLV-TYPE SHOULD be unique at all nested levels. The TLV Type number space and initial assignments will be specified in the later revision of the current document. NDN packet design will try best to keep the length of T staying with a single byte.

The `TLV-LENGTH` value represents number of bytes that `TLV-VALUE` uses. It **does not** include number of bytes that `TLV-TYPE` and `TLV-LENGTH` fields themselves occupy. In particular, empty payload TLV will carry `TLV-LENGTH` equal to 0.

This encoding offers a reasonable balance between compactness and flexibility. Most common, standardized Type codes will be allocated from a small-integer number-space, and these common Types will be able to use the compact, single-byte encoding.

## 3.3 Non Negative Integer Encoding

A number of TLV elements in NDN packet format take a non-negative integer as their value, with the following definition:

```
nonNegativeInteger ::= BYTE+
```

Length value of the TLV element MUST be either 1, 2, 4, or 8. Depending on the length value, a nonNegativeInteger is encoded as follows:

- if the length is 1 (i.e. the value length is 1 octet), the nonNegativeInteger is encoded in one octet;
- if the length is 2 (= value length is 2 octets), the nonNegativeInteger is encoded in 2 octets, in net byte-order;
- if the length is 4 (= value length is 4 octets), the nonNegativeInteger is encoded in 4 octets, in net byte-order;
- if the length is 8 (= value length is 8 octets), the nonNegativeInteger is encoded in 8 octets, in net byte-order.

The following shows a few examples of TLVs that has nonNegativeInteger as their value component in hexadecimal format (where TT represents TLV-TYPE, followed by the TLV-LENGTH, then TLV-VALUE):

```
0     => TT0100
1     => TT0101
255   => TT01FF
256   => TT020100
65535 => TT02FFFF
65536 => TT0400010000
```

### 3.4 Changes from CCNx

- XML-based ccnb packet encoding is replaced by TLV encoding.

## 4 Name

An NDN Name is a hierarchical name for NDN content, which contains a sequence of name components.

### 4.1 NDN Name Format

We use a 2-level nested TLV to represent a name. The Type in the outer TLV indicates this is a Name. All inner TLVs have the same Type indicating that they each contain a name component. There is no restriction on the Value field in a name component and it may not contain any bytes:

```
Name ::= NAME-TYPE TLV-LENGTH NameComponent*
NameComponent ::= NAME-COMPONENT-TYPE TLV-LENGTH BYTE+
```

### 4.2 NDN URI Scheme

For textual representation, it is often convenient to use URI to represent NDN names. Please refer to RFC 3986 (URI Generic Syntax) for background.

- The scheme identifier is ndn.
- When producing a URI from an NDN Name, only the generic URI unreserved characters are left unescaped. These are the US-ASCII upper and lower case letters (A-Z, a-z), digits (0-9), and the four specials PLUS (+), PERIOD (.), UNDERSCORE (_), and HYPHEN (-). All other characters are escaped using either the percent-encoding method of the URI Generic Syntax or a ndn scheme specific hexadecimal string escape starting with the EQUALS (=) and an even number of characters from the set of hex digits. Once an EQUALS has been encountered in a component the hexadecimal encoding persists until the end of the component. The hex digits in these escaped encodings should always use upper-case letters, i.e., A-Z.
- To unambiguously represent name components that would collide with the use of . and .. for relative URIs, any component that consists solely of zero or more periods is encoded using three additional periods.
- The authority component (the part after the initial // in the familiar http and ftp URI schemes) is not relevant to NDN. It should not be present, and it is ignored if it is present.

### 4.3 Implicit Digest Component

The Name of every piece of content includes as its final component a derived digest that ultimately makes the name unique. This digest may occur in an Interest Name as an ordinary Component (the last one in the name). This final

component in the name is never included explicitly in the Data packet when it is transmitted on the wire. It can be computed by any node based on the Data packet content.

The **implicit digest component** consists of the SHA-256 digest of the entire Data packet without the signature component. Having this digest as the last name component enables us to achieve the following two goals:

- Identify one specific Data packet and no other.
- Exclude a specific Data packet in an Interest (independent from whether it has a valid signature).

### 4.4 Canonical Order

In several contexts in NDN packet processing, it is useful to have a consistent ordering of names and name components. NDN names consist of a sequence of NameComponents, and each NameComponent is a sequence of zero or more 8-bit bytes. The ordering for components is such that:

- If *a* is shorter than *b* (i.e., has fewer bytes), then *a* comes before *b*.
- If *a* and *b* have the same length, then they are compared in ASCII lexicographic order (e.g., ordering based on memcmp() operation.)

For Names, the ordering is just based on the ordering of the first component where they differ. If one name is a proper prefix of the other, then it comes first.

### 4.5 Changes from CCNx

- The name encoding is changed from binary XML to TLV format.
- The discussions on naming conventions and the use of special markers inside NameComponents are removed from packet specification, and will be covered by a separate technical document
- Deprecated zero-length name component.

## 5 Interest Packet

NDN Interest packet is TLV defined as follows:

```
Interest ::= INTEREST-TYPE TLV-LENGTH
               Name
               Selectors?
               Nonce
               Scope?
               InterestLifetime?
```

`Name` and `Nonce` are the only two two required elements in an Interest packet. Selectors are optional elements that further qualify Data that may match the Interest. They are used for discovering and selecting the Data that matches best to what the application wants. Selectors are placed right after the Name to facilitate implementations that may use continuous memory block of Name and Selectors TLVs together as the index for PIT lookup. By using a TLV to group all the Selectors, an implementation can easily skip them to find Nonce, which is used together with Name to identify looping Interests. If Selectors TLV is present in the Interest, it MUST contain at least one selector.

The two other optional elements, Scope and InterestLifetime, are referred to as *Guiders*. They affect Interest forwarding behavior, e.g., how far the Interest may be forwarded, and how long an Interest may be kept in the PIT. They are not grouped.

## 5.1 Name

The Name element in an Interest is synonymous with the term *prefix*. See *Name section* (page 5) for details.

## 5.2 Selectors

```
Selectors ::= SELECTORS-TYPE TLV-LENGTH
                MinSuffixComponents?
                MaxSuffixComponents?
                PublisherPublicKeyLocator?
                Exclude?
                ChildSelector?
                MustBeFresh?
```

### MinSuffixComponents, MaxSuffixComponents

```
MinSuffixComponents ::= MIN-SUFFIX-COMPONENTS-TYPE TLV-LENGTH
                          nonNegativeInteger

MaxSuffixComponents ::= MAX-SUFFIX-COMPONENTS-TYPE TLV-LENGTH
                          nonNegativeInteger
```

When needed, `MinSuffixComponents` and `MaxSuffixComponents` allow a data consumer to indicate whether the Name in the Interest is the full name including the digest, or the full name except for the digest, or the content it is seeking has a known range of legitimate component counts. These two parameters refer to the number of name components beyond those in the prefix, and counting the implicit digest, that may occur in the matching Data. The default for `MinSuffixComponents` is 0 and for `MaxSuffixComponents` is effectively infinite, meaning that any Data whose name starts with the prefix is a match. Often only one of these will be needed to get the desired effect.

### PublisherPublicKeyLocator

```
PublisherPublicKeyLocator ::= KeyLocator
```

This element specifies the name of the key which is used to sign the Data packet that the consumer is requesting. This is a way for the Interest to select answers from a particular publisher.

See *KeyLocator* (page 13) section for more detail.

### Exclude

```
Exclude ::= EXCLUDE-TYPE TLV-LENGTH Any? (NameComponent (Any)?)+
Any ::= ANY-TYPE TLV-LENGTH(=0)
```

The `Exclude` selectors allows requester to specify list and/or ranges of names components that MUST NOT appear as a continuation of the Name prefix in the responding Data packet to the Interest. For example, if Interest is expressed for `/ndn/edu` and Exclude specifies one name component `ucla`, then nor data producer nor conforming NDN routers are allowed to return any Data packet that has prefix `/ndn/edu/ucla`.

Exclude filter applies only to a name component of the Data packet name that is located at a position that numerically equals to the number of name components in the Interest packet, assuming 0 is the first name component.

The Components in the exclusion list MUST occur in strictly increasing order according to the canonical NDN name component ordering (*Name Section* (page 5)), with optional leading, trailing, and interleaved `Any` components. The following defines processing of `Any` components:

- If none of the `Any` components are specified, the filter excludes only to the names specified in the Exclude list.
- If a leading `Any` component is specified, then the filter excludes all names that are smaller or equal (in NDN name component canonical ordering) to the first NameComponent in the Exclude list.
- If a trailing `Any` component is specified, then the filter excludes all names that are larger or equal (in NDN name component canonical ordering) to the last NameComponent in the Exclude list.
- If `Any` component is specified between two NameComponents in the list, then the filter excludes all names from the range from the right NameComponent to the left NameComponent, including both ends.

Exclude filter MUST not consist of a single `Any` component or one NameComponent with leading and trailing `Any` components.

### ChildSelector

```
ChildSelector ::= CHILD-SELECTOR-TYPE TLV-LENGTH
                    nonNegativeInteger
```

Often a given Interest can match more than one Data within a given content store. The `ChildSelector` provides a way of expressing a preference for which of these should be returned. If the value is 0, the leftmost child is preferred. If 1, the rightmost child is preferred. Here leftmost and rightmost refer to the least and greatest components according to the canonical NDN name component ordering (*Name Section* (page 5)). This ordering is only done at the level of the name hierarchy one past the name prefix.

For example, assuming in the name hierarchy the component immediately after the name prefix is the version number, whose next level is the segment number, then setting ChildSelector to be 1 will retrieve the rightmost version number (i.e., the latest version) and the leftmost segment number (i.e., the first segment). However, this selection is only done with respect to a single content store, not globally. Additional rounds that exclude the earlier versions may be used to explore other content stores for newer versions. In this case, the use of ChildSelector does not change the multi-round outcome, but it decreases the number of rounds needed to converge to an answer.

### MustBeFresh

```
MustBeFresh ::= MUST-BE-FRESH-TYPE TLV-LENGTH(=0)
```

This selector is encoded with Type and Length but no Value part. When it is absent from an Interest packet, the router can respond with a Data packet from its content store whose FreshnessPeriod is either still valid or expired. When it is present in an Interest packet, the router should not return Data packet from its content store whose FreshnessPeriod has expired.

The FreshnessPeriod carried in each Data packet (*Data Section* (page 10)) is set by the original producer. It starts counting down when the Data packet arrives at a node. Consequently if a node is N hops away from the original producer, it may not consider the Data stale until N *X* FreshnessPeriod after the Data is produced.

## 5.3 Nonce

Nonce defined as follows:

```
Nonce ::= NONCE-TYPE TLV-LENGTH(=4) BYTE{4}
```

The Nonce carries a randomly-genenerated 4-octet long byte-string. The combination of Name and Nonce should uniquely identify an Interest packet. This is used to detect looping Interests.

## 5.4 Guiders

### Scope

```
Scope ::= SCOPE-TYPE TLV-LENGTH nonNegativeInteger
```

This value limits how far the Interest may propagate. Scope 0 prevents propagation beyond the local NDN daemon (even to other applications on the same host). Scope 1 limits propagation to the applications on the originating host. Scope 2 limits propagation to no further than the next node. Other values are not defined at this time, and will cause the Interest packet to be dropped.

Note that Scope is not a hop count—the value is not decremented as the Interest is forwarded.

### InterestLifetime

```
InterestLifetime ::= INTEREST-LIFETIME-TYPE TLV-LENGTH nonNegativeInteger
```

`InterestLifetime` indicates the (approximate) time remaining before the Interest times out. The value is the number of milliseconds. The timeout is relative to the arrival time of the Interest at the current node.

Nodes that forward Interests may decrease the lifetime to account for the time spent in the node before forwarding, but are not required to do so. It is recommended that these adjustments be done only for relatively large delays (measured in seconds).

It is the application that sets the value for `InterestLifetime`. If the `InterestLifetime` element is omitted, a default value of 4 seconds is used (4000). The missing element may be added before forwarding.

## 5.5 Changes from CCNx

- `Nonce` is changed from optional to required.
- `PublisherPublicKeyDigest` is replaced by `PublisherPublicKeyLocator`.
- `AnswerOriginKind` is simplified from 4bits to a 1-bit `MustBeFresh`.
- `FaceID` has been removed.
- `InterestLifetime` changes the unit to the number of milliseconds.
- Removed Bloom Filter from Exclude.
- Changed default semantics of staleness.

  Specifically, NDN-TLV Interest without any selectors will bring any data that matches the name, and only when `MustBeFresh` selector is enabled it will try to honor freshness, specified in Data packets. With Binary XML encoded Interests, the default behavior was to bring "fresh" data and return "stale" data only when `AnswerOriginKind` was set to 3.

  Application developers must be aware of this change, reexamine the Interest expression code, and enable `MustBeFresh` selector when necessary.

# 6 Data Packet

NDN Data packet is TLV defined as follows:

```
Data ::= DATA-TLV TLV-LENGTH
           Name
           MetaInfo
           Content
           Signature
```

The Data packet represents some arbitrary binary data (held in the Content element) together with its Name, some additional bits of information (MetaInfo), and a digital Signature of the other three elements. The Name is the first element since all NDN packet processing starts with the name. Signature is put at the end of the packet to ease the implementation because signature computation covers all the elements before Signature.

## 6.1 Name

See *Name section* (page 5) for details.

## 6.2 MetaInfo

```
MetaInfo ::= META-INFO-TYPE TLV-LENGTH
               ContentType?
               FreshnessPeriod?
               FinalBlockId?
```

Compared with CCNx, four fields are removed: PublisherPublicKeyDigest, ExtOpt, Timestamp, and FinalBlockID for the following reasons.

- PublisherPublicKeyDigest is supposed to be used in selecting data packets signed by a particular key. We replace PublisherPublicKeyDigest with KeyLocator, which is part of the Signature block (see *Signature Section* (page 12)), due to the following consideration. First, it requires data consumer to acquire a *valid* public key, as opposed to the key locator, before sending Interest out. Second, if a router is to verify the content objects, it must have other means to locate the keys first. Further, it may require publishers to maintain their public keys and certificates by their public key digests instead of names.
- ExtOpt was intended for extending XML-based ccnb format. Since we are now using TLV, ExtOpt is no longer needed.
- Timestamp and FinalBlockID can be useful meta information for applications, but do not need to be processed at the network layer. Therefore, if desired, applications should encode such meta information as part of the content.

### ContentType

```
ContentType ::= CONTENT-TYPE-TYPE TLV-LENGTH
                  nonNegativeInteger
```

Three ContentTypes are currently defined: default (=0), LINK (=1), and KEY (=2). The **default** type of content is a BLOB (=0), which is the actual data bits identified by the data name. The textbf{LINK} type of content is another name which identifies the actual data content. The `KEY` type of content is a public key.

Compared with CCNx, three types, ENCR, GONE, and NACK are removed. ENCR means the content is encrypted, and since the network layer should not care whether content is encrypted or not, this type is not needed. GONE was a placeholder for implementing cache purging, however the research is yet to be carried out on how to accomplish this

goal, if it is feasible to achieve, it is not included in this 0.1 version of NDN specification. NACK is used to signal a downstream node that the upstream node is unable to retrieve a matching data. Since the actual NACK mechanism is still under active investigation, we do not include it in this version of specification, but may add it back in a future version.

### FreshnessPeriod

```
FreshnessPeriod ::= FRESHNESS-PERIOD-TLV TLV-LENGTH
                      nonNegativeInteger
```

The optional FreshnessPeriod indicates how long a node should wait after the arrival of this data before marking it as stale. The encoded value is number of milliseconds. Note that the stale data is still valid data; the expiration of FreshnessPeriod only means that the producer may have produced newer data.

Each content store associates every piece of Data with a staleness bit. The initial setting of this bit for newly-arrived content is "not stale". If the Data carries FreshnessPeriod, then after the Data has been residing in the content store for FreshnessPeriod, it will be marked as stale. This is per object staleness and local to the NDN node. Another possible way to set the staleness bit of a local content is for a local client to send a command to the local NDN daemon.

If an Interest contains MustBeFresh TLV, a Data that has the staleness bit set is not eligible to be sent in response to that Interest. The effect is the same as if that stale Data did not exist (i.e., the Interest might be matched by some other Data in the store, or, failing that, get forwarded to other nodes). If an exact duplicate of a stale Data arrives, the effect is the same as if the stale Data had not been present. In particular, the Data in the store is no longer stale. As a practical matter, a stale Data should be ranked high on the list of things to discard from the store when a storage quota has been reached.

### FinalBlockId

```
FinalBlockId ::= FINAL-BLOCK-ID-TLV TLV-LENGTH
                   NameComponent
```

The optional FinalBlockId indicates the identifier of the final block in a sequence of fragments. It should be present in the final block itself, and may also be present in other fragments to provide advanced warning of the end to consumers. The value here should be equal to the last explicit Name Component of the final block.

## 6.3 Content

```
Content ::= CONTENT-TYPE TLV-LENGTH BYTE*
```

## 6.4 Changes from CCNx

- The structure of Data packet is changed.
- SignedInfo is renamed to MetaInfo and its content is changed.
- PublisherPublicKeyDigest and ExtOpt are removed.
- Timestamp and FinalBlockID are removed.
- KeyLocator is moved to be inside the Signature block.
- Three content types, ENCR, GONE, and NACK are removed.
- FreshnessSeconds is renamed to FreshnessPeriod and is expressed in units of milliseconds.

# 7 Signature

NDN Signature is defined as two consecutive TLV blocks: `SignatureInfo` and `SignatureValue`. The following general considerations about SignatureInfo and SignatureValue blocks that apply for all signature types:

1. `SignatureInfo` is **included** in signature calculation and fully describes the signature, signature algorithm, and any other relevant information to obtain parent certificate(s), such as *KeyLocator* (page 13)
2. `SignatureValue` is **excluded** from signature calculation and represent actual bits of the signature and any other supporting signature material.

```
Signature ::= SignatureInfo
              SignatureBits

SignatureInfo ::= SIGNATURE-INFO-TYPE TLV-LENGTH
                    SignatureType
                    ... (SignatureType-specific TLVs)

SignatureValue ::= SIGNATURE-VALUE-TYPE TLV-LENGTH
                    ... (SignatureType-specific TLVs and
                    BYTE+
```

## 7.1 SignatureType

```
SignatureType ::= SIGNATURE-TYPE-TYPE TLV-LENGTH
                    nonNegativeInteger
```

This specification defines the following SignatureType values:

| Value | Reference | Description |
|---|---|---|
| 0 | *DigestSha256* (page 12) | Integrity protection using SHA-256 digest |
| 1 | *SignatureSha256WithRsa* (page 12) | Integrity and provenance protection using RSA signature over a SHA-256 digest |
| 2-200 | | reserved for future assignments |
| >200 | | unassigned |

### DigestSha256

`DigestSha256` provides no provenance of a Data packet or any kind of guarantee that packet is from the original source. This signature type is indended only for debug purposes and limited circumstances when it is necessary to protect only against unexpected modification during the transmition.

`DigestSha256` is defined as a SHA256 hash of the *Name* (page 5), *MetaInfo* (page 10), *Content* (page 11), and *SignatureInfo* (page 12) TLVs:

```
SignatureInfo ::= SIGNATURE-INFO-TYPE TLV-LENGTH(=3)
                    SIGNATURE-TYPE-TYPE TLV-LENGTH(=1) 0

SignatureValue ::= SIGNATURE-VALUE-TYPE TLV-LENGTH(=32)
                     BYTE+(=SHA256{Name, MetaInfo, Content, SignatureInfo})
```

### SignatureSha256WithRsa

`SignatureSha256WithRsa` is the basic signature algorithm that MUST be supported by any NDN-compliant software. As suggested by the name, it defines an RSA public key signature that is calculated over SHA256 hash of

the *Name* (page 5), *MetaInfo* (page 10), *Content* (page 11), and *SignatureInfo* (page 12) TLVs.

```
SignatureInfo ::= SIGNATURE-INFO-TYPE TLV-LENGTH
                    SIGNATURE-TYPE-TYPE TLV-LENGTH(=1) 1
                    KeyLocator

SignatureValue ::= SIGNATURE-VALUE-TYPE TLV-LENGTH
                     BYTE+(=RSA over SHA256{Name, MetaInfo, Content, SignatureInfo})
```

---

**Note:** SignatureValue size varies (typically 128 or 256 bytes) depending on the private key length used during the signing process.

---

This type of signature ensures strict provenance of a Data packet, provided that the signature verifies and signature issuer is authorized to sign the Data packet. The signature issuer is idenfified using *KeyLocator* (page 13) block in *SignatureInfo* (page 12) block of `SignatureSha256WithRsa`. See *KeyLocator section* (page 13) for more detail.

---

**Note:** It is application's responsibility to define rules (trust model) of when a specific issuer (KeyLocator) is authorized to sign a specific Data packet. While trust model is outside the scope of the current specification, generally, trust model needs to specify authorization rules between KeyName and Data packet Name, as well as clearly define trust anchor(s). For example, an application can elect to use hierarchical trust model [1] (page 15) to ensure Data integrity and provenance.

---

## 7.2 KeyLocator

A `KeyLocator` specifies a name that points to another Data packet containing certificate or public key, or can be used by the specific trust model in another way to verify the the content.

```
KeyLocator ::= KEY-LOCATOR-TYPE TLV-LENGTH KeyLocatorValue

KeyLocatorValue ::= Name |
                    KeyLocatorDigest |
                    ...

KeyLocatorDigest ::= KEY-LOCATOR-DIGEST-TYPE TLV-LENGTH BYTE+
```

---

**Note:** KeyLocator has meaning only for specific trust model and the current specification does not imply or suggest use of any specific trust model. Generally, KeyLocator should point to another Data packet which is interpreted by the trust model, but trust model can allow alternative forms of the KeyLocator.

For example, one can define a trust model that does not interpret KeyLocator at all (KeyLocator MUST be present, but TLV-LENGTH could be 0) and uses naming conventions to infer proper public key or public key certificate for the name of the Data packet itself. Another possibility for the trust model is to define digest-based KeyLocatorValue (`KeyLocatorDigest`), where RSA public key will be identified using SHA256 digest, assuming that the trust model has some other means to obtain the public key.

---

## 7.3 Changes from CCNx

- `Signature` is moved to the end of Data packet.
- `KeyLocator` is moved to be a part of the `SignatureInfo` block, if it is applicable for the specific signature type. The rationale for the move is to make Signature (sequence of `SignatureInfo` and `SignatureValue` TLVs) self-contained and self-sufficient.

- Signature type (or signing method information) is expressed as an assigned integer value (with no assumed default), rather than OID.
- Added support for cheaper signatures
- The current specification does not define Merkle Hash Tree Aggregated Signatures, but it is expected that such (or similar) signatures will be defined in future version of this specification.

# 8 Type value assignment

| Type | Assigned value (decimal) | Assigned value (hexadecimal) |
|---|---|---|
| **Packet types** | | |
| Interest | 5 | 0x05 |
| Data | 6 | 0x06 |
| **Common fields** | | |
| Name | 7 | 0x07 |
| NameComponent | 8 | 0x08 |
| **Interest packet** | | |
| Selectors | 9 | 0x09 |
| Nonce | 10 | 0x0a |
| Scope | 11 | 0x0b |
| InterestLifetime | 12 | 0x0c |
| **Interest/Selectors** | | |
| MinSuffixComponents | 13 | 0x0d |
| MaxSuffixComponents | 14 | 0x0e |
| PublisherPublicKeyLocator | 15 | 0x0f |
| Exclude | 16 | 0x10 |
| ChildSelector | 17 | 0x11 |
| MustBeFresh | 18 | 0x12 |
| Any | 19 | 0x13 |
| **Data packet** | | |
| MetaInfo | 20 | 0x14 |
| Content | 21 | 0x15 |
| SignatureInfo | 22 | 0x16 |
| SignatureValue | 23 | 0x17 |
| **Data/MetaInfo** | | |
| ContentType | 24 | 0x18 |
| FreshnessPeriod | 25 | 0x19 |
| FinalBlockId | 26 | 0x1a |
| **Data/Signature** | | |
| SignatureType | 27 | 0x1b |
| KeyLocator | 28 | 0x1c |
| KeyLocatorDigest | 29 | 0x1d |

### 8.1 Type value reservations

| Values | Designation |
|---|---|
| 0-4, 30-79 | Reserved for future assignments (1-byte encoding) |
| 80-100 | Reserved for assignments related to local link data processing (NDNLP header (http://redmine.named-data.net/projects/nfd/wiki/NDNLP-TLV), LocalControlHeader (http://redmine.named-data.net/projects/nfd/wiki/LocalControlHeader), etc.) |
| 101-127 | Reserved for assignments related to forwarding daemon |
| 128-252 | For application use (1-byte encoding) |
| 253-32767 | Reserved for future assignments (3-byte encoding) |
| >32767 | For application use (3-byte encoding) |

## References

[1] Chaoyi Bian, Zhenkai Zhu, Alexander Afanasyev, Ersin Uzun, and Lixia Zhang. Deploying key management on NDN testbed. Technical Report NDN-0009, Revision 2, NDN, February 2013. http://named-data.net/techreports.html.