A Few Arguments for Explicit Segment and Version Naming
Last updated 7/13/2014

Question 1

Lixia asks: *1/ You mentioned yesterday there may be cases where even the data consumers may not have a complete understanding of the data name structure -- can you offer an example how this may occur? Since we are not as familiar with application cases, we did not see how the consumers not knowing the name structures (as it'd be defined by the app)*

First, the general point and then a specific example:

NDN is focused on *data exchange* rather than host-to-host channel-based communication. That data exchange could have long delays between production and consumption, and there is no reason for a consumer to "contact" the original publisher if the content object itself is sufficient for its purposes. So, the more that can be determined directly and unambiguously from the name of a content object, the better—subject to reasonable engineering constraints. E.g., one consumer's live video segment is another's archival material. It seems likely that the more that can be determined directly and unambiguously from the name of the object, the better for exchange across apps and domains, especially for file-like objects.

If we believe in the death of "local files" in favor of NDN content objects, the more we must agree to not always know a priori what exactly is trying to consume these objects. (That's a nice feature of files with known structure, file systems that provide metadata, and should be of NDN name conventions in this case as well.) Who knows what application would try to load/use a networked content object? What tool might be trying to sync "the newest" objects in a namespace?

While we have so far been pretty focused on replacing channel-based approaches to, say, video, chat, IoT communication with producer/consumer pairs, a huge amount of content falls into the category where *the content object itself is the interface between disparate applications*. The most prominent example is file-like objects that are pretty much independent of what application is accessing them. (Think of how many apps implement FTP, SFTP, HTTP, AFP, SMB, NFS, etc. just to cover their bases about moving files around... But the files are "the thing" that matter.)

Retrieving a segmented, versioned content object is so basic of a function that it should be easy to get right -- making library support straightforward and, critically, enabling human publishers/consumers to simply share object prefixes in the human world.

To go one step further, carrying such critical information about data objects by implicit rather than explicit convention recreates in this domain a problem that we've cited in IP networks: There is really important knowledge that is unnecessarily inaccessible to the network application operating at the NDN layer... Knowledge of the namespace structure has to be transmitted through some other protocol or be coded implicitly in the app or library. Why not make it explicitly visible?

I have been considering Green's cognitive dimensions framework[1], often referenced in usability of programming languages, as it applies to IP and NDN, and this relates to the following aspects of usability (I think):

---

[1] Green, Thomas R. G., and Marian Petre. "Usability analysis of visual programming environments: a 'cognitive dimensions' framework." Journal of Visual Languages & Computing 7.2 (1996): 131-174. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.54.3584&rep=rep1&type=pdf

- "**Hard mental operations**: Are there places where the user [app developer] needs to resort to fingers or pencilled annotation to keep track of what's happening?"
- "**Error-proneness**: Does the design of the notation induce 'careless mistakes'?"
- "**Closeness of mapping**: What 'programming games' need to be learned?" (same reason to get rid of combining markers and numbers)", and even:
- "**Progressive evaluation**: Can a partially-complete program [namespace] be executed to obtain feedback on 'How am I doing'?"

Web Browser Example

For example, consider a straightforward port of web-browser object retrieval to NDN:

1. The publisher -- often an end-user -- names a pdf file `/edu/ucla/jburke/foo/1/0`

2. The OS implements segmenting and versioning on the network side and names segments accordingly, generating `/edu/ucla/jburke/foo/1/0/0/0`, etc.

3. The consumer -- also an end-user -- enters the prefix known to both (`foo/1/0`) in the URL bar of a browser.

How does the browser distinguish between the filename and versioned/segmented content itself? We could come up with a way to infer it based on the final components. But, the routine to infer things gets messier quickly, if, for example:

- Segmenting and versioning are optional, so we can have segmented, non-versioned content and vice versa, and have to detect those cases

- Parents (potentially directories, in this example) can also be versioned, or similarly

- Content is nested (i.e., `foo/1/0` corresponds to an object, but so does `foo/1`)

- The fetch routines enable particular versions to be also specified by the end-user or, say, a client-side script... i.e., `foo/1/0/42`

- The fetch routines enable particular segments to be specified by the end-user or a client-side-script

It seems to me that the namespace structure inference approach has to be unnecessarily complicated to support these cases, or be combined with other end-user input. We can invent ways around this, I suppose, but anytime there are user-supplied prefixes for content objects, it seems easy to run into trouble at the consumer (and perhaps even publisher).

Why not component typing?

I think that using component typing to get around this is a Bad Idea because I fear the type explosion that Van mentioned on Monday. If we do allow component typing, I have a long list of types to propose as well as a type registration adjudication process that involves app developers. :)

Explicit Version / Segment Child Objects

Taking both of these into account, explicit child objects with names that are the markers (only) seem the best way to handle this as they are cleaner for discovery, scaling to include other metadata, etc.

Per Van's suggestion, why not use: `foo/1/0/_v/<version>/_s/<segment>` where `_v`, `_s` are explicit markers.

In the future, this scales nicely if we want to support string or binary data too, e.g., `foo/1/0/_m/<meta_data_objs>` for non-number metadata that would have been available from the MIME type or file extension, etc.

> Sidebar: Or, is it `/_v/<version>/_m` ? Explicit markers make both orderings possible. They even make individually versioned segments possible. Order-based approaches do not enable applications to explore such alternative semantics without (perhaps) breaking the conventions.

Elsewhere, Dave Oran has argued that apps should consider explicit `key=value` names, so that implicit ordering is not relied on for important decision making. I am not sure that is always needed. But this is a conceptually similar idea - we need basic object retrieval to be based on a dependable and straightforward approach that doesn't rely on information external to the namespace.

Why not directories for these objects?

Directories have a place here, too, for file-like objects, but envision a versioned, multi-segment ChronoShare directory object, for example. Consistent, low-level, name-based conventions are still needed to fetch that, too, in order to bootstrap a higher-level file access protocol.

Video has multiple versions and segments separated

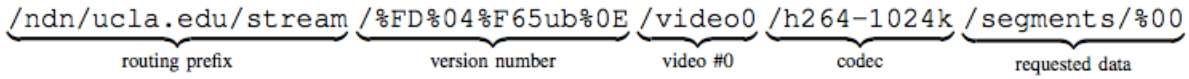

Figure 1. Example of data packet name

Question 2

Lixia asks: *2/ for the "middleman" cases like repo or even the library: wonder if you could also offer an example or two on how they may benefit from knowing the version and segment number components in the name.*

The best examples I can think of are repos that perform domain-wide ingest and republishing of content on behalf of entire ecosystems or organizations. Some structure is known, but most is not. In this case, retrieval might be prioritized based on version (for example), and/or explicit structure read during sync/transfer/ingest might be used to "hook" general routines for version control, segmented content retrieval, etc.

I believe but am not certain yet that there's a strong argument here in the mHealth and IoT applications, because the whole point is to support an ecosystem of applications that are all accessing data produced by other applications *that they would prefer not to know much about*.

In those applications, versioning could be used at several levels because of accumulating history and recursive retrieval. For example:

```
<person>/fitness_record/<version>/
      imaging/home_camera/0/<version>/_s/<segments>
```

Imagine SYNC'ing the `<person>/fitness_record` namespace as part of an electronic health record (EHR) ingest of patient-provided data or incorporation into a personal data cloud (PDC), then prioritizing retrieval of the latest content objects only for storage and immediate viewing by the physician.

Another application in the same ecosystem might use the very different structure:

```
<person>/hospital/org/kp/
      mri_images/<version>/<slice>/<segments>
```

We'd still want the repo / ingest routines to retrieve the "latest" content with the same SYNC and retrieval process, without having to know the structure. Sure, *other* applications in the system may know more about structure, but they'll probably only know about structure for namespaces of interest. It seems easier to manage the data transfer consistently with version and segment (and metadata) children, especially when you slice "versions" into layers as above. The ingest app could be provided a template for all possible namespace structures producing the content, but why do that at that layer, when it can be expressed explicitly in the name? At some point, templates may be needed for apps to find specific data, but we should not introduce that at ingest/retrieval/browsing, necessarily.

If there are thousands of unique data provider and consumer apps, we want to make basic object transfer as consistent and error-free as possible. Explicit segment and version children do this. Implicit component ordering, even with types, does not.

Lixia 6/3

1/ there is a general agreement on the need to mark version and seg# explicitly.

Ok.

2/ but exactly what is the best way to do this?

a) why not using name component types, given we have types already?
wouldn't the worry about type explosion extend to marker explosion as well?

The high-level point I would make is that opening up name type definition essentially creates a second namespace (for types) that is differentiated only in its encoding, call it a *nametypespace*. A type code is just a delimiting marker parsed differently in the library. So do we want to start providing type manipulation tools that mirror name manipulation tools?

Since a *nametypespace* is just a namespace used to describe the subsequent name component, I would suggest markers should just be put in the name and we use the existing namespace manipulation tools for this. "If it can be in the name, put it in the name" seems to continue to be a good mantra for reasons described further below.

b) if we decide to use markers, why make the marker as a separate component, as opposed to the CCNx way which just attach special characters in front of the version/seq# component?

I agree with the point that folks seem to be making that the CCNx way was awkward to decode. Though, if there is library support I don't see that as a fundamental problem.

The primary con to marker components seems to just be some lack of compactness...?

Some potential reasons I can see to use marker components, outside of just managing type explosion:

1. **Violating the name opaqueness principle through nametypes is likely to be error-prone given the coupling of app names with network forwarding, as well as in interest matching.**

   If nametypes are opened to application definition and any part of a prefix can be typed, then:

   a. Do we forward the interest for
      `<utf-8>{foobits}/<utf-16>{barbits}/<version>{7}`
      to the same place as
      `<utf-16>{foobits}/<utf-16>{barbits}/<version>{7}` ??

      If the types are not ignored in forwarding, then they 1) probably need to be exposed in representations to the end-user – suddenly we have things in the name presented to users that are not name components – and 2) seems like all havoc could break lose easily if the same component bits with different types are forwarded differently. 3) Perhaps an interesting attack vector, too.

   Continuing for the sake of argument, let's say by definition nametypes are ignored for forwarding, and opaqueness of the component bits themselves is preserved.

   b. Does library-level interest matching take into account component types? I'm not sure, but it seems like it should do what the network does, so for the sake of argument ignore types there too.

   c. Each application then is responsible for checking both type and value for each component that it cares about, in order to find/confirm versions and segments, which are ignored in forwarding and library interest matching.

      Result in this case: Useful information for the application that describes the data is hidden from interest matching and forwarding for the first time in the architecture.

   Neither option seems that appealing.

2. **Using marker-component delimited subtrees means that as new capabilities (like SYNC) are introduced that operate on names, they will automatically be available for working with versions, segments, metadata, etc.** This is not true in the case of types nor as straightforward in markers integrated with components.

   For example, once an app has been given or finds a version prefix, it can work with child names (versions) directly just like other names. You could do the same with types, but would then have to

rebuild library functions that already exists for names. Seems unnecessary and error-prone, with no additional expressiveness.

To me, this is a very straightforward reason for implementing markers with components.

3. **Enabling marker *hierarchy* by using names also provides additional expressiveness** that may be useful but we are not locked into.

   e.g., it is easier for apps to explore things like versioned metadata: `foo-object/_m/_v/47`
   How would you encode this with types? (Especially now that empty components are disallowed? :)

4. Visibility is also an important consideration. **Putting things in the name, especially with marker components, puts this information more readily in a human-readable domain**, if the app and/or libraries desire. I don't think this should be underestimated.

   Even if the marker name is not human-readable; common markers could be translated just as / is used for component boundaries. As suggested above, we can do this for types, but would have to invent a new delimiter and explain its meaning, potentially even to end-users.

   This capability of doing type-value pairs could become a commonly used feature of the architecture from the application perspective. For flexibility, it would be reasonable for app developers to demand more expressive (say string) types, and we'd be back to name components…

5. Putting markers into the name hierarchy helps to limit the object count in a given marker subtree, if an object has multiple marker-delimited components someday in the future.

6. Similarly, it is actually more compact overall if markers are considered as keys in key/value pairs (don't have to repeat the key in every child object name).

7. Finally, protocol-specified types for delimiting TLV fields that are part of the architecture and are very slowly changing should not, in my opinion, be conflated with application-specified types of names or data. **This raises at minimum a terminology problem that need to be solved.** Not sure both can be called types and have users follow what's going on easily.

c) although the type space is a universal resource shared by all and needs standardization, but marker definitions also need agreement
- unrecognized type and unrecognized markers seem causing the same problem
- so do the conflicting types/markers

Perhaps this is true but using types in this way raises their prominence (to app developers and perhaps even end users) in the architecture and requires us to both provide functions to manage the typespace separately from the namespace.

Again, since a *nametypespace* is an application-controlled namespace, non-protocol related types should be expressed as names and managed like other namespaces. To make a distinction for only a marginal efficiency increase I think is a bad idea. Brooks[2] suggests the principle of "propriety" – not introducing what is [essentially] immaterial…

[2] Brooks, F. *The Design of Design*. (2010)

As an aside, I think introducing a limited set of markers as name conventions will not be interpreted by developers in the same way as managing a "type".

d) we do see 2 differences between using types and using marker:
- Jeff's way of adding markers make names longer/more components,
maybe that could be a factor to discourage liberal use of markers?

In practice I think length may provide benefits (expressiveness) as well as limits, as with any name.

If we go with Van's suggestion of short (one symbol?) marker names, I really don't see the length as an issue... and it's insufficient to overcome the limitations above, at least as I understand so far.

- marker space may be unlimited, while type space is not.

Yes.

Further thoughts (JeffT) –

If the TLV spec defines typed name components, then it opens questions of

- What defines if an NDN application is compliant?
- What should applications do when encountering unrecognized types?
- Is it an error for an application to use a type component in a way not described in the TLV spec?
- If they define a UTF8 type and a String type, then can two different encodings be equivalent for name matching?
- Critically, how to define canonical ordering in the presence of name component types?

From Alex, 7/12 with comments by JB inline:

Hi Van, Jeff, and everybody else,

There is one urgent problem that we must resolve before we can make the public release of NFD. For a long time we were stuck with the conversation about encoding segment and version numbers and we still don't have any decision. Unfortunately, this caused inconsistency problem within our libraries.

In particular, CCL libraries still use CCNx naming conventions for segment and version encoding, while all ndn-cxx based applications (NFD, repo-ng, NLSR, and other small tools like ndncatchuncks3) use just number encoding ("nonNegativeInteger" encoding in terms of our NDN-TLV spec). This simple encoding of numbers was defined in NFD spec for the management protocols as part of http://redmine.named-data.net/projects/nfd/wiki/StatusDataset and http://redmine.named-data.net/projects/nfd/wiki/Notification specs.

So, now we got to problem that was "discovered" by Steve recently: "segment number" encoded in one library (ndn-cxx) cannot be "decoded" in another (pyndn) and vice versa. This has to be fixed, but given that we still don't have a decision on how to mark segments/versions, we don't yet have the "right" way to fix it. So, we must make some decision right now or at least decide what to do in the short term, to fix consistency problem within the first release.

JB: My thoughts are basically the same as in the doc I passed around earlier - a slightly updated version is attached. I am still in favor of option #2. Before proceeding with #1 I would want to see some response to the concerns on p5-7 about name interpretation ambiguity. Also, the type registration/explosion concern is to me mostly a question of whether we really want/need a type registry for apps and an "invisible" namespace for component names that would need a URI representation, etc, etc.

Let me list here all the options for the "right" solution that we have, giving my personal comments about them. All of them assume explicit marking that component is version or segment.

1. Designate TLV type for version and segment (SegmentNameComponent, VersionNameComponent).

Advantages:
+ the most minimal wire representation
+ non-ambiguous, guaranteed semantics for the name component

Concerns:
- We would need to make several new definition for name components: ordering between different types of components, how to make URI representation of the typed component, how to handle unrecognized types (e.g., how to represent them in URI when needed)
- Common types needs to be clearly defined in the spec and there should be a way to "reseve" types.
*- There could be "type explosion" problem, but I don't clearly see what exactly is this problem. **Van, can you comment on this more?***
Even though our type space is not as infinite as name components space, it is really huge (2^64).

2. Jeff's proposal to define segment, version (and all other) markers as separate components (.../_s/1, .../_v/12, ...)

+ no need to any new NDN-TLV spec definition
+ name components kept opaque
+ "more clear" designation: _s (or _segment) is clearer than "Type=128" in option 1 and %00 prefix in option 3. However, if wire optimization is being considered, this advantage no longer applies.

Concerns:
- The most non-optimal wire encoding and most complicated processing by applications and 3rd parties.

[jb] "Most" of our two/three options. :) What is the performance improvement we would really expect to see? Also, I'm not sure that it's the most complicated processing if we have to handle in the forwarder / expose to the developer and user how to handle displaying, forwarding, and generally working with the same component bits that have different types.

- Marker components can be ambiguous: an application/data producer may inadvertently use "_s"/"_v" as part of the data. In such cases, more problems could follow if the next component is being decoded as "number".

[jb] Yes - but one could argue this of any naming convention. The combination of well known / unusual markers (I was not necessarily suggesting the literal string "_s", just that it is parsed/printed that way, though this is debatable) seems to make this reasonable... and the assumption is that there are certain ordering/position conventions that help.

- There is inter-dependency between name components, and I don't see clearly how this can/should be handled in the applications

[jb] Do you mean the hierarchical relationship? If so, I agree but I think it could actually be conceptually helpful in some circumstances as mentioned in the doc.

- Given that two components are required to represent version/segment, trust models would need to be updated (it is simple, I listed here just as a reminder that we need to do it)

3. Old CCNx-style naming convention.

I would say we should not completely take out CCNx-based naming convention out of the picture. The actual encoding of the number could be optimized (e.g., we can use a slightly modified version of nonNegativeInteger with prefix in front), but the idea of coupling markers with actual data within opaque name component is appealing.

Advantages:
+ no need to any new NDN-TLV spec definition
+ name components kept opaque

Concerns:
- Marker components can be ambiguous: an application/data producer may inadvertently use "%00"/"%FD" prefixes as part of the data. In such cases, a problem may occur if the rest of the component is decoded as number
- Common types needs to be clearly defined in the spec and there should be a way to "rerseve" types.

Is there "type explosion" here? If not, why it exists in option 1?

For me, this option is about the same as option 1, but with a little bit less optimal wire format (~extra 1-2 bytes) and "problem" of marker ambiguity.

From Lan, 7/13:

Just a clarification question: it seems to me that as long as the encoding applies to all applications, there needs to be some process to standardize the encoding, whether it is encoded as a separate name component (in your proposal), or in the type, or in the first few bits of the name (CCNx). Is my understanding correct?

For versioning and segmenting, this may be true. But, setting aside other concerns expressed in the document I sent earlier, I think that "type codes" vs. "name conventions" would be handled differently enough that the choice has significant impact.

Defining a type system seems to create a "global" way of describing component bits that I understand to be expected to be followed by every application. From a previous message by Alex I interpreted that there would be a list of types and their definitions as part of the protocol, which range from bit encodings - like "Number" - to semantic definitions, like "Segment" - that would be coordinated through a registration process for the sake of interoperability. To avoid confusing type code collisions, this coordination is useful. This is trivial now, but non-trivial in the future if NDN is widespread.

Defining name conventions one by one, on the other hand, does not make a global change to the protocol nor require a registry of anything but the conventions considered relevant by the architects. Conventions can come into being organically based on agreement within a particular application community, and if technical governance structures are needed to create standards that resolve ambiguity or promote interoperability, they can arise - but all of this dialogue happens around namespaces, not the weird intersection of typespaces and namespaces. The equivalent can be coordinated in types by assigning blocks of types to particular groups/communities, but that sounds suspiciously like the equivalent of IP/port block assignments. In any case, the coordination requirement is not "designed in" in the same way that seems to be happening with types. I am really fond of the notion that names are opaque for this reason.

The type space **can** be managed, but I am having a hard time understanding the motivation for opening this can of worms, if the main benefit is a minor performance increase and a parsing approach that is simpler in the short-term. At a technical level, my other comments about forwarding and representational ambiguity are the ones I am actually the most concerned about.