NDN-CNL: A Hierarchical Namespace API for Named Data Networking

Jeff Thompson
UCLA REMAP
Los Angeles, California, USA
jefft0@remap.ucla.edu

Peter Gusev
UCLA REMAP
Los Angeles, California, USA
peter@remap.ucla.edu

Jeff Burke
UCLA REMAP
Los Angeles, California, USA
jburke@remap.ucla.edu

ACM Reference Format:
Jeff Thompson, Peter Gusev, and Jeff Burke. 2019. NDN-CNL: A Hierarchical Namespace API for Named Data Networking. In *ICN '19: Conference on Information-Centric Networking, September 24–26, 2019, Macao, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3357150.3357400

1 INTRODUCTION

The Named Data Networking Common Name Library (NDN-CNL) is a high-level library that enables applications to work with hierarchical, named data collections as an abstract interface to NDN's request-response protocol. This approach foregrounds the importance of application-named data in NDN applications and is intended as an alternative to socket APIs. It aims to simplify programming of asynchronous applications that use a variety of data-centric approaches at the same time, including naming conventions for versioning, segmenting, synchronization, name-based access control, name confidentiality, schematized trust, as well as standard features needed in many applications such as interest pipelining and latest data retrieval. This paper introduces the rationale and design of the library, shows its use through a series of examples, and concludes with a brief discussion of future work. Our emphasis is on introducing the new abstraction, so the library's internal implementation is not discussed in detail; source code is available on Github.[1]

2 BACKGROUND

Named Data Networking (NDN) [9] is a proposed ICN network architecture that forwards data directly based on application-defined names. As a replacement for the TCP/IP architecture,[2] it provides request-response semantics at the network layer that are similar to web semantics, but at packet granularity. It does this without requiring host addressing or name-to-address mappings, such as those provided by the Domain Name System (DNS). Each Data packet is cryptographically bound to its name by a cryptographic signature or similar mechanism. Stateful forwarding is used to route packets through the network without source and destination addresses [6].

NDN makes named data the new "thin waist", or common interoperability layer. It communicates using application-level naming, which leads to systems that are semantically interconnected at the network layer.

APIs for NDN application development have evolved over the last eight years. The early CCNx API focused on wire format packet assembly. PyCCN[3] abstracted away the wire format details to provide an Interest-Data exchange API that was adopted and expanded in *ndn-cxx*[4] and NDN-CCL (Common Client Library)[5]. The Consumer/Producer API introduced in [4] used a socket-style API to provide higher-level functionality including reliability, verification, and other features without requiring application developers to code packet-level Interest-Data exchange directly. The NDN-CNL is written on top of the NDN Common Client Library and thus occupies an evolutionary branch adjacent to and inspired by the functionality of the Consumer/Producer API, while introducing a new and different abstraction that emphasizes the data namespace.

3 RATIONALE

The motivation for developing a namespace-oriented library is to foreground a central abstraction of NDN and explore how to make it available to software developers. In NDN, application objects are represented as *collections of immutable named data packets*. For example, an immutable file-like object is expressed as a sequence of packets, each containing a chunk of the file; a mutable file is, in turn, expressed as a collection of versions of those collections, perhaps along with manifests and other metadata. Typically, these objects and their constituent packets are hierarchically organized. The names of *prefixes* correspond to the application objects made up from the lower-level collection of packets. Our primary objective was to foreground this concept. While many of NDN's networking benefits can be explored with packet granularity APIs, we propose that its broader benefits to application authors (and effective use of the underlying network capabilities) requires a data-centric API abstraction, such as the NDN-CNL.

Practically, we are also interested in NDN becoming more competitive as an application-level technology as the ICN research field matures. Higher-level APIs are a necessity for this to happen. We are seeking an NDN-based middleware replacement for our own applications that provides the ability to write higher-level data-centric networking code operating on application objects with packet-level details left to reusable libraries. In our experience, it is

[1] github.com/named-data/cnl-cpp , github.com/named-data/PyCNL
[2] NDN can also run as an overlay on top of TCP/IP networks and, in fact, on top of any medium that can carry bits.

[3] github.com/named-data/PyCCN
[4] named-data.net/doc/ndn-cxx/current
[5] named-data.net/codebase/platform/ndn-ccl

ICN '19, September 24–26, 2019, Macao, China

ACM ISBN 978-1-4503-6970-1/19/09…$15.00
https://doi.org/10.1145/3357150.3357400

Listing 1: Segmented object example in Python.

```
face = Face()
image = Namespace("/foo/someimage/42")
image.setFace(face)
def onSegmentedObject(handler, obj):
    print("Got image")
SegmentedObjectHandler(image, onSegmentedObject).objectNeeded()
```

challenging for developers to use libraries focused on supporting Interest-Data exchange (including our NDN-CCL) in applications where they wish to simultaneously incorporate namespace synchronization, schematized trust, and name-based access control, on top of basic abstractions such as segmentation and versioning and performance-motivated optimizations such as pipelined Interests.

Arguably, such higher-level interfaces are achieved (or could be achieved) in the socket-inspired Consumer/Producer API [4]. However, that API does not expose or allow manipulation of the hierarchical relationships between various types of application data objects, and its affordances for succinctly composing new combinations of features from basic capabilities are limited.

The intersection of our conceptual and practical goals led to the approach discussed here, an abstraction that *harmonizes handling prefixes representing higher-level data objects with data-centric, packet-level operations*. This approach is inspired by and focused on the importance of names in the architecture. At a high level, we assumed that applications would often 1) use sync[5] to keep namespaces updated; 2) use app conventions or standard metadata to identify object types and other information necessary to know how to process named branches; and 3) then publish or fetch data of for relevant prefixes, using standard mechanisms for versioning, segmenting, and so on. With this in mind, we designed an API[6] to enable working with namespaces as if they represent hierarchical collections of objects. An added advantage of this approach that address another long-standing desire is the ability to use the in-memory namespace knowledge to enable developers to specify names using wildcards and regular expressions that are not yet practical to be applied at wire speed.

4 DESIGN

As discussed above, the NDN-CNL's primary goal is to provide a collection-oriented interface to NDN data that enables manipulation of both application-level objects and data packets. We aimed to achieve this with a small set of core features and minimizing loss of generality relative to NDN-CCL. The library holds an in-memory tree of Namespace nodes, where each node represents a name, its parent represents the prefix of this name, and each child represents the name with one appended name component. Thus, the "namespace hierarchy" typically shown in an NDN application design document is directly represented.

In the rest of this section, we describe the details of the design, which is inspired by a number of APIs and query languages that allow direct access to and manipulation of hierarchically-organized data, from the XPath query language for XML documents to the Google Cloud Firestore, which supports hierarchical key names for its cloud-native NoSQL database. While the CNL does not itself provide a detailed query language for namespaces (yet) or the features of a distributed database, the examples in the next section show how the availability of local namespace knowledge makes it possible to quickly create applications whose code is similarly organized around a distributed dataset itself, rather than around communication details, while maintaining the lower-level benefits of the NDN architecture, such as intrinsic multicast, data-centric security, etc. To repeat the point made at the end of the last section, this approach will typically require the availability of robust, efficient namespace synchronization protocols, which is an open research topic.

A key observation that informs the NDN-CNL design is that, in NDN, either leaf or parent nodes (or both) in a named data hierarchy can correspond to application data objects. And, though names corresponding to packets refer to immutable data, prefixes often refer to mutable application-level objects. Consider the simple example: /foo/someimage/<version>/. There may be many immutable NDN objects with fully specified names of this format in an application. In addition, from the application's point of view /foo/someimage is a mutable object that has a latest version, and /foo/someimage/42 is one such specific (immutable) version. (Listing 1 shows a CNL example that fetches all the segments of this version, assembles them in order and notifies the application which prints "Got image".) An application which only cares about high-level objects may monitor direct children of /foo/someimage to be alerted when new versions arrive. But an application can also monitor changes in deeper levels of the Namespace, e.g. individual segment packets, to measure progress in assembling the higher-level object. The Namespace tree provides a unified API for these different tasks.

A second observation is that an application may become aware of a new name (at any depth) in a number of ways, including: 1) it may create a new object; 2) it may need a new object and know some of the name; 3) it may opportunistically overhear a packet with the name from another node, e.g., in a wireless multicast environment; 4) it may use synchronization to retrieve names of interest.

Based on these two observations, along with the concept that packet-level data objects on the network and in storage are immutable while application objects represented by prefixes in the hierarchy may be considered mutable by the application, we designed the node state diagram shown in Figure 1. This state diagram is based on commonly discussed and implemented steps in the applications that we have been developing and are aware of. Again, it reflects that both leaf and parent nodes may represent data objects from an application's perspective, but can potentially use the same API.

[6] named-data.net/doc/ndn-ccl/latest/PyCNL

Typically, a higher-level object[7] goes through states such as "Serializing" and "Signing" based, ultimately, on the states of descendant leaf nodes which hold the individual immutable Data packets that are transmitted. One could, though, provide a specific handler to support manifest-level signatures and other approaches as well.

An application interacts with the CNL through the nodes in the Namespace tree, either by calling methods on the node objects, or by registering callbacks which notify the application of a change of state of Namespace nodes. Other APIs for NDN applications strictly separate the API for producer and consumer. But the CNL Namespace tree provides a common workspace where objects that are produced both by the application or received from the network are attached to the Namespace node with the object name, removing the strict separation between producer and consumer.

The state machine in Figure 1 shows that both the "producer arc" on the top and "consumer arc" on the bottom begin with calling the objectNeeded() method of a Namespace node. When the CNL receives an Interest, it calls objectNeeded() on the node with the Interest name. The application can register to respond when objectNeeded() is called, produce the object and attach it to the Namespace. The CNL then uses it to answer the Interest, thus acting like a producer. If the CNL receives another Interest which is satisfied by an object already stored in the Namespace tree, it responds immediately, thus acting like a content cache. Likewise, the application can call objectNeeded() on a Namespace node. If the Namespace tree does not already have the object, the CNL sends an Interest to the network. When it receives the matching Data packet, the CNL attaches it to the Namespace (possibly decrypting and deserializing) and calls the application callback, thus acting like a consumer. Furthermore, one part of an application can call objectNeeded() and another part of the application can respond to this by producing the object, allowing "producer" and "consumer" communication within the application.

The CNL also provides a suite of handlers (see examples below) to handle typical data patterns such as segmented content, versioned objects, or latest data retrieval. A handler is assigned to a portion of the Namespace tree and calls methods or registers callbacks to respond to changes (the same way an application would). A handler can also serialize one application object into one or more Data packets, and deserialize multiple Data packets into one application object. This is why the final state in Figure 1 says "Object ready" instead of "Data packet ready". The CNL allows the application to be abstracted away from the binary blobs in network-level Data packets and to interact with a meaningful object managed by a handler, e.g. a JSON object.

Multiple handlers can be assigned to combine functionality. The developer can write an application to handle a binary segmented object under one child namespace (e.g., /foo/%00, etc.) and later assign another handler which serializes/deserializes as JSON under a different child namespace (e.g., /foo/json). Testing full composability and how to manage potential conflicts of multiple handlers is still a research topic.

[7] One current limitation to the design is that a name cannot function as a prefix and data object at the same time. Once a leaf node has a Data packet, a packet with the same name but different content cannot be attached, nor can a child node have a Data packet; this simplifies the implementation and may be relaxed in the future.

As objects are produced or consumed and attached to Namespace nodes, the application can traverse the objects or perform more complex operations on names, such as regular expressions or transformations, which are not directly supported by the network. This convenience comes at the price of memory usage that may be significant for large numbers of objects. The handlers provided by the CNL partially mitigate this by deleting intermediate results (or auxiliary handlers) when the final object is reported to the application.

5 IMPLEMENTATION

The NDN-CNL is built on the NDN Common Client Library (CCL). The CCL is implemented in C, C++, Python, JavaScript, Java, C# and Squirrel. The CNL is currently implemented in C++, Python and C#, with plans for the others. The CNL relies on the application to supply callbacks which are called when the state of the Namespace tree changes, and uses the standard callback mechanism of each language. As with the CCL, the API for the CNL remains the same across languages, to enable experimentation in different development environments.

The CNL has been used in an Augmented Reality application [1] to communicate video frame annotations (e.g. object recognition). A C++ application receives video, processes the frames and adds the annotations to its CNL Namespace tree. The consumer application in Python creates a CNL GeneralizedObjectStreamHandler (see below) which fetches the stream of annotations and adds them to its Namespace tree, calling the application callback when each arrives. If the consumer application needs older annotations, it simply browses the Namespace tree.

The CNL has also been used in a data repository application (work in progress)[8]. The repository enables CNL's sync capability (see Section 6.3 below). Producer applications also enable sync and add objects to the CNL Namespace tree that they want to store. The repository receives the announcements of new names, fetches the objects and stores them.

The initial CNL implementation allowed the application to receive separate callbacks for each state change event, such as "Name added", "Interest expressed" and "Object ready". But since the state diagram (Figure 1) is well-defined, we found it simpler for the application to receive one callback for "state changed" with a parameter for the state. Also, since the application mainly interacts with the CNL through handlers, in most cases it is easier to use handler-specific callbacks such as "received generalized object".

6 EXAMPLES

We use a series of simple examples to further illustrate the basic abstractions and how the library is used. In these examples, we introduce how a developer might write code using pre-existing handlers for common object types and functionality; a detailed discussion of how handlers are written is outside the scope of this paper, but the interested reader can review the source code referenced in the introduction.

[8] github.com/remap/fast-repo

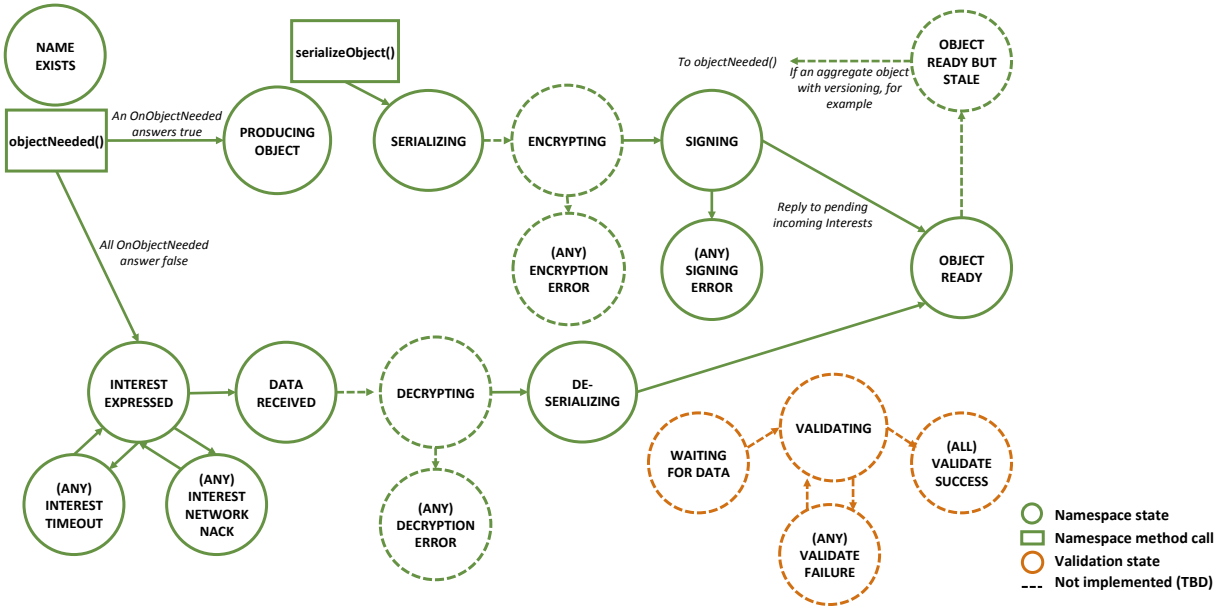

Figure 1: NDN-CNL Name node state diagram. Signing/validation and encryption/decryption may be performed at the packet and/or object level, depending on the object type implemented by associated handlers.

6.1 Generalized Objects

Through writing applications for NDN for several years, we have identified some commonalities in network data objects, and attempted to generalize them in newer applications. We are currently experimenting with a "generalized object" type and have implemented support in the NDN-CNL to test both the applicability of the object format and the ability of the CNL's Namespace abstraction to implement simple ways of working with it.

Figure 2 shows the namespace of a generalized object, which has content defined by a content type and timestamp, where the content is segmented only if needed. (object_prefix)/_meta is a signed packet with the content type and timestamp. In many cases, the content is small enough and is placed in the "other" section of the _meta packet, with no further processing. Otherwise, the content is segmented into (object_prefix)/%00%00, (object_prefix)/%00%01, etc. To save processing time and bandwidth, the segment packets are not signed, but their digests are placed in the signed _manifest packet. As the consumer fetches the segments, it computes and saves their digests. Finally it verifies the signature on the _manifest packet and verifies that the segment digests are the same as it computed. Overall, the generalized object aims at achieving flexibility in fetching names by separating different types of metadata and payload, and allows consumer applications to decide on what needs to be fetched and when. For example, a consumer may opt not to verify data segments right away and not fetch the _manifest until later.

Listing 2 shows an example generalized object producer. Lines 1-3 create the system default communication Face and a KeyChain for signing packets with the default identity. Line 4 creates a CNL Namespace object with the object prefix. Line 5 tells the CNL to use the created Face to exchange data within this namespace, and to register to receive Interests with its prefix. Finally, line 7 creates a GeneralizedObjectHandler and uses it to set the object for the namespace to have the given string and a content type of "text/html". The handler (not shown) creates the _meta packet and segment packets, if necessary, as child Namespace objects following the namespace design in Figure 2. The CNL will use these to answer Interests.

Listing 3 shows the related generalized object consumer. Line 1 creates the default communication Face. Lines 2 and 3 create a CNL Namespace object and tell the CNL to use the Face. Line 4 sets a callback which the library calls on validation failure. When the GeneralizedObjectHandler finishes fetching the generalized object, it will call the callback defined on line 6, where contentMetaInfo is an object containing the content type and timestamp from the _meta packet, and objectNamespace is the CNL Namespace object to which the retrieved object is attached. Line 9 creates a GeneralizedObjectHandler to use this callback, and to attach to the Namespace as its handler. Finally, it calls objectNeeded() so that the CNL begins fetching and alerting the handler to process. When the handler has assembled the object (possibly from segments), it calls the callback.

6.2 Generalized Object Stream

The generalized object can be extended to a stream of objects, each with an incremented sequence number. For example, an applications may produce annotations for a video at regular intervals. Figure 3 shows the generalized object stream namespace. It is similar to the generalized object, but instead of a fixed object prefix, the stream of generalized objects have the prefixes (stream_prefix)/(sequence N), (stream_prefix)/(sequence N+1), ...

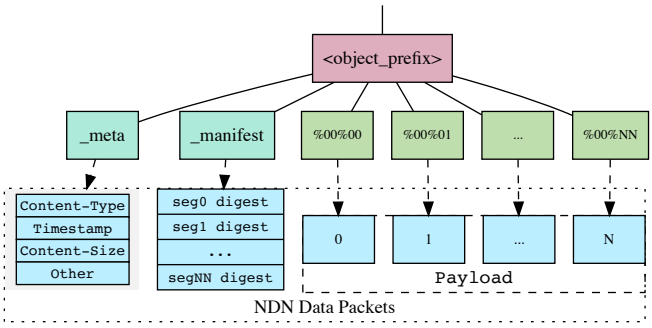

Figure 2: Generalized Object Namespace

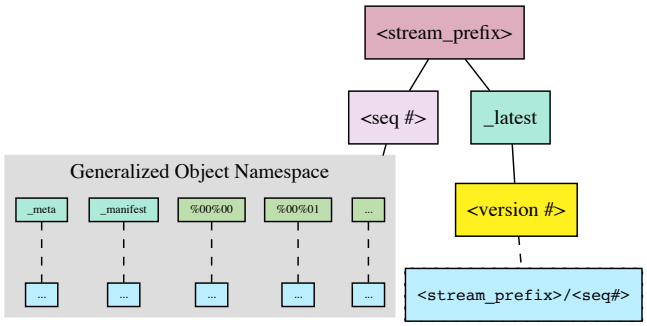

Figure 3: Generalized Object Stream Namespace

Similar to the GeneralizedObjectHandler, a GeneralizedObjectStreamHandler can be attached to a CNL Namespace for the stream prefix. The producer application repeatedly calls the handler's addObject(object). This increments the sequence number and internally uses a GeneralizedObjectHandler to place the packets for the object in the children of the Namespace. As before, the CNL will use these to answer Interests.

When the GeneralizedObjectStreamHandler in the consumer application initializes, it needs to know the latest sequence number to start fetching. It uses the real-time data retrieval (RDR) protocol [3], sending an Interest for (stream_prefix)/_latest which returns a signed, versioned Data packet with a short freshness period whose content is the name of the latest sequence, e.g. (stream_prefix)/(sequence N). To fetch the stream, the handler uses an Interest pipeline by always keeping a fixed number outstanding Interests for the next sequence numbers. It internally attaches a GeneralizedObjectHandler to the sequence number Namespace object to fetch the object and call the application's callback when it arrives. If the Interest for the highest sequence number times out, the handler restarts by repeatedly using RDR to try to get the latest sequence number.

Some applications produce a stream of generalized objects but at sporadic intervals, so an Interest pipeline is not appropriate. In this case, the handler is configured to repeatedly use RDR to ask for and fetch the latest sequence number.

6.3 Name Synchronization with PSync

With the CNL, applications may know and operate on namespaces for which they do not yet have some or all of the data. This is partially inspired by the concept of "sync" in the NDN architecture, which uses efficient set reconciliation techniques to synchronize namespaces across multiple nodes, after which Interest-Data exchange can optionally be used by each node to fetch named data of interest. [2]

The CNL currently supports the PSync protocol [10] to synchronize namespaces between instances. When an application creates a CNL Namespace object, it can enable sync at a point in the Namespace tree. The CNL will join the sync group. Any new names created by the application in the Namespace to a specified depth are announced to other users in the sync group. (The depth limitation allows, for example, announcing the name of a new version, but not the child names containing segment numbers.) Likewise, when new names are announced by other users, the CNL adds the names to the Namespace tree and the application which monitors the Namespace can take action, e.g. to fetch the content.

The CNL has a test_sync[9] example, which is ostensibly run by two different users. Each instance creates a CNL Namespace for the prefix /test/app, and enables sync. Each instance also creates a user prefix, e.g. /test/app/alice or /test/app/bob, and then creates child Namespace objects under this prefix, e.g. /test/app/bob/1, test/app/bob/2. Because both instances are in the sync group, they receive the announced names from the other user which are added to its own Namespace tree. The application monitors changes to the Namespace, detects the other user's new names and displays them.

The CNL uses the PSync protocol at this time because it can efficiently represent large sets of names through the use of Invertible Bloom Filters (IBF). A full discussion of PSync is outside of the scope of this paper, but in practice, in the CNL, an application can announce up to 275 names on each update. (If the set difference is larger than this, then the application sends a "recover" message with all of its names.)

6.4 Schematized Trust & Name-based Access Control

The CNL supports schematized trust [7] through the capabilities of the underlying CCL library, and as shown in the state diagram of Figure 1. As the code listings cited above show, a producer application assigns an NDN-CCL KeyChain object to a Namespace node which is used to sign Data packets created for it and child nodes. (The KeyChain can be configured for different signing identities and algorithms.) Likewise, a consumer application can assign a CCL Validator, where packets are validated in parallel to other processing. An application callback can receive a "Validate failure" notification and take appropriate action. At the implementation level, the CCL provides a specialized Validator for schematized trust which the consumer configures with a policy to enforce relationships between the names of a packet and its signer.

Encryption/decryption is similarly built on the CCL, which provides an API for Name-based Access Control[10] (NAC)[8]. Each Data

[9] github.com/named-data/PyCNL/blob/master/examples/test_sync.py
[10] github.com/named-data/name-based-access-control/blob/new/docs

packet is encrypted with a random content key, independent of the user. Each content key is encrypted with the public key of an RSA key pair. Then an "access manager" creates Data packets whose name is based on the names of the content and of a consumer who needs access to the associated private key (hence "name-based" control). The API provides a Decryptor object for the consumer which uses this naming convention to fetch this packet and recover the content key to decrypt the Data packet. (Details omitted for brevity.)

In the CNL, decryption is one of the steps of the standard Namespace node state machine (Figure 1). To enable decryption, the consumer creates a Decryptor object with its name and connected to its RSA keypair in the KeyChain. Then, the consumer calls setDecryptor() on the CNL Namespace object which will use it to decrypt incoming Data packets for this and child nodes. This is shown in the test_nac_consumer[11] example. It uses the same segmented object handler as the simple segmentation example, but the CNL uses the supplied Decryptor to automatically decrypt the segment Data packets from the producer[12]. These examples illustrate how the CNL can compose different functionalities.

6.5 Wildcards & Regular Expressions

Two common questions that we have received from application developers are 1) how to enumerate objects available in a namespace and 2) how to retrieve objects matching a certain pattern. There is currently no direct architectural or forwarder support for either operation. However, building blocks for providing these features within a given application context are available. For example, an application could use a synchronization protocol, such as that described in Section 6.3 above, to share knowledge of a namespace among many instances– which is enabled with a single method call for a Namespace in the CNL. Then, it can run filtering or matching operations on the local knowledge of the namespace with only a few lines of code. See Listing 4 for a simple example. The CNL does not yet provide extensive name matching and iteration methods, but we expect that these can be built relatively simply because of its approach of holding namespace knowledge locally.

7 CONCLUSION

The NDN-CNL is an application-inspired, collection-based API for data-centric networking. The features and examples discussed in the previous section have all been implemented in C++, C#, and Python. The library has had some initial use in research applications in our lab, including a mixed reality production described in [1]. The CNL implicitly captures some design strategies that we have used repeatedly in NDN applications, including: 1) use of prefixes to represent mutable application-level objects; 2) synchronization to gather local knowledge of a namespace; 3) an application-level cache that enables publish-and-forget behavior, allowing the library to independently respond to incoming Interests whenever possible. It enables features we have long sought, including wildcard and regular expressions on names, and attempts to ease a key pain point in NDN development: the integration of asynchronous, data-centric approaches to authenticity and confidentiality, processing of common data types, and lower-level features such as reliability and real-time data retrieval.

In addition to limitations and areas of future work identified earlier, other design issues include how to best propagate events from individual low-level packets, such as timeouts, validation failure, expired freshness, etc., to higher-level objects, as well as approaches to integrating storage for persistence and lowering memory requirements. A variety of implementation opportunities also remain, such as maintaining statistics on prefixes (e.g., Interest retransmission, RTT, etc.) and optimization of the performance of the library as an application-level cache. Even at this early stage of development, we hope this API and implementation will be valuable to the ICN community and generate other approaches to support application development over ICN architectures with higher-level APIs without the loss of benefits from NDN's packet-level data-centric design.

[11] github.com/named-data/PyCNL/blob/master/examples/test_nac_consumer.py
[12] github.com/named-data/PyCNL/blob/master/examples/test_nac_producer.py

ACKNOWLEDGMENTS

The authors thank Lixia Zhang and Alex Afanasyev for their input on the CNL and Ashlesh Gawande for help integrating PSync. Portions of this work were supported by NSF Award Nos. CNS-1719403 and CNS-1629922 and the Intel ICN-WEN program.

REFERENCES

[1] Peter Gusev, Jeff Thompson, and Jeff Burke. 2019. Data-centric video for mixed reality. In *2019 28th International Conference on Computer Communication and Networks (ICCCN)*. IEEE.
[2] Tianxiang Li, Wentao Shang, Alex Afanasyev, Lan Wang, and Lixia Zhang. 2018. A Brief Introduction to NDN Dataset Synchronization (NDN Sync). In *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*. IEEE, 612–618.
[3] Spyridon Mastorakis, Peter Gusev, Alexander Afanasyev, and Lixia Zhang. 2018. Real-Time Data Retrieval in Named Data Networking. In *2018 1st IEEE International Conference on Hot Information-Centric Networking (HotICN)*. IEEE, 61–66.
[4] Ilya Moiseenko, Lijing Wang, and Lixia Zhang. 2015. Consumer/producer communication with application level framing in named data networking. In *Proceedings of the 2nd ACM Conference on Information-Centric Networking*. ACM, 99–108.
[5] Wentao Shang, Yingdi Yu, Lijing Wang, Alexander Afanasyev, and Lixia Zhang. 2017. *A Survey of Distributed Dataset Synchronization in Named Data Networking*. Technical Report NDN-0053. NDN.
[6] Cheng Yi, Alexander Afanasyev, Ilya Moiseenko, Lan Wang, Beichuan Zhang, and Lixia Zhang. 2013. A Case for Stateful Forwarding Plane. *Computer Communications: ICN Special Issue* 36, 7 (April 2013), 779–791.
[7] Yingdi Yu, Alexander Afanasyev, David Clark, kc claffy, Van Jacobson, and Lixia Zhang. 2015. Schematizing Trust in Named Data Networking. In *Proc. of ACM ICN*.
[8] Yingdi Yu, Alexander Afanasyev, and Lixia Zhang. 2016. *Name-Based Access Control*. Technical Report NDN-0034. NDN.
[9] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, kc claffy, Patric Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. 2014. Named Data Networking. *ACM Computer Communication Review* (July 2014).
[10] Minsheng Zhang, Vince Lehman, and Lan Wang. 2016. Partialsync: Efficient synchronization of a partial namespace in ndn. *Technical report, Technical Report NDN-0039, NDN* (2016).

Listing 2: Generalized object producer.

```
face = Face()
keyChain = KeyChain()
face.setCommandSigningInfo(keyChain, keyChain.getDefaultCertificateName())
objectPrefix = Namespace("/ndn/eb/run/28/description", keyChain)
objectPrefix.setFace(face,
  lambda prefixName: dump("Register failed for prefix", prefixName))
GeneralizedObjectHandler().setObject(
  objectPrefix, Blob("EB run #28. Ham and oats"), "text/html")
```

Listing 3: Generalized object consumer.

```
face = Face()
objectPrefix = Namespace("/ndn/eb/run/28/description")
objectPrefix.setFace(face)
objectPrefix.addOnValidateStateChanged(lambda ns, changedNS, state, ID: print(
  "Validate failure" if state == NamespaceValidateState.VALIDATE_FAILURE else ""))
def onGeneralizedObject(contentMetaInfo, objectNamespace):
    print("Got generalized object, content-type " +
          contentMetaInfo.contentType + ": " + str(objectNamespace.obj))
GeneralizedObjectHandler(objectPrefix, onGeneralizedObject).objectNeeded()
```

Listing 4: Simple wildcard example.

```
applicationPrefix = Namespace(Name("/test/app/users"), keyChain)
applicationPrefix.setFace(face,
    lambda prefix: dump("Register failed for prefix", prefix))
applicationPrefix.enableSync()  # Sync with other instances using this namespace
# ... Since the Namespace object childComponents is iterable, enumerate simply elsewhere -
regex = re.compile("Bob.*")
for child in filter(lambda c: regex.match(str(c)), applicationPrefix.childComponents):
    applicationPrefix[child].objectNeeded(True)  # generate interests to retrieve
```