NDN Libraries
Progress and Plans

March 24, 2019

Jeff Thompson, Jeff Burke

jefft0@remap.ucla.edu

Overview

- Common Client Libraries (CCL)
- PSync
- Common Name Library (CNL)
- NDN-RTC

What are the Common Client Libraries (CCL)?

- Enable client applications to use NDN in C++, Python, JavaScript, Java, .NET
- Common API across languages: http://named-data.net/doc/ndn-ccl-api
- Interest/Data, signatures, encryption, transports, app utilities, unit tests, examples
- Track ndn-cxx research (security, NAC, NDN protocols, NFD interaction)
- Backwards compatibility, platform flexibility for development stability
- Used in NDN-RTC, BMS, mHealth, neighborhood network, web page apps, ICE-AR
- Specialized libraries: NDN-CPP Lite (Arduino), Imp, Android, browser speedups
- Stats (total): 10,771 commits, 277 closed issues, 79 pull requests, 80 forks

Example

```
face = Face("memoria.ndn.ucla.edu")
name = Name("/ndn/edu/ucla/remap/demo/ndn-js-test/hello.txt/%FDU%8D%9DM")
def onData(interest, data):
    print(data.content.toRawStr())
face.expressInterest(name, onData)
```

CCL Features

- Certificate signing/validating – RSA, ECDSA, HMAC
- Configurable cert chain policies, regex name matching
- Flexible public/private key database API
- Signed Interests – verify with same API as certs
- Name-base access control (AES encryption, RSA key protection)
- MemoryContentCache, SegmentFetcher
- Optional thread-safe network I/O
- Configurable wire format (see below)
- ChronoSync, PSync (see below)
- Unit tests, example programs

CCL wire format abstraction

- API is not hard-wired to one wire format
- Enable backwards compatibility if running with old forwarders

```
WireFormat.setDefaultWireFormat(Tlv0_1WireFormat.get())
```

- Can specify on ad hoc basis if sending to a various networks

```
face.expressInterest(name, onData, Tlv0_1WireFormat.get())
```

- Was used for transition from CCN 0.x
- Plans to support other ICN wire formats

CCL – Next steps

- NDN wire format v0.3 (with backwards compatibility)
  - Typed name components
  - Removed (most) Interest selectors
  - Interest hop count
  - Interest defaults to exact name (optional CanBePrefix)
  - Extra application parameters in the Interest
  - Explicit fields for signed interests (instead of using name components)
- New wire formats
- Support new network autoconfig protocols

What is PSync?

- Developed as improvement to ChronoSync
- Used in NLSR to sync routes on the NDN test bed
- Part of the CCL
- Invertible Bloom filter of a set of hashed names
  - Send interest with my IBF, receive interests with others’ IBF
  - Stable state: Everyone sends the same IBF – Interest aggregation, no Data
  - Update: I receive a different IBF with missing names and provide in reply Data
  - IBF efficiently updates a set difference of ~275 names
- Eventual consistency from pairwise updates – broadcast not needed
- Option to subscribe to partial namespace updates

Example PSync app

```
face = Face()
def onNamesUpdate(names):
    print("Got names, starting with " + names[0].toUri())

updateSize = 80
pSync = FullPSync2017(updateSize, face, Name("/sync"), onNamesUpdate)
pSync.publishName(Name(”/edu/ucla/jefft/paper.txt"))
```

PSync – Next steps

- Implement in Python, JavaScript, Java (currently in C++)
- Use as native sync for the Common Name Library (see below)
- Stress test “eventual consistency” without broadcast
- Support partial PSync (waiting for use case)

What is the Common Name Library (CNL)?

- Library enabling applications to work with hierarchical, named data collections.
    - Namespace object (root and child nodes)
    - Application interacts with a Namespace node (attach handlers, receive notifications)
- Provides a lightweight way to integrate various:
    - Sync mechanisms (i.e., PSync, vector sync)
    - Data access patterns (i.e., Consumer/Producer API, fetch latest),
    - Publishing models (i.e., publish/subscribe, in-memory content cache),
    - Complex namespace queries / pattern matching (i.e., regexp, wildcards),
    - Triggered data generation (supporting security)
- Currently using in ICE-AR mobile client application
  (No interest-data exchange exposed to developers of that app.)
- Segmented content with a Meta packet and versioning
- Built-in encode/decode, encrypt/decrypt, sign/verify as part of the pipeline
- New names added to the Namespace tree through PSync, app is notified

CNL Motivation

- Provide tools for working with namespaces as they represent collections, in an *information-focused* rather than communication-oriented way
- Assume asynchronous network operations will be used to sync the namespace and consume/publish objects in the collection
- Insulate non-networking developers from communication details
- Make progress towards NDN as a middleware-replacement in terms of high-level, application-facing features, but try to stay as general as possible
- Work with aggregate application-specific objects, not (segmented) blobs in packets
- As a result, support namespace synchronization the way that is conceived / described at a high-level, and promote it as an application-level concept to explore

Example segmented content consumer app

```
face = Face("memoria.ndn.ucla.edu")
page = Namespace("/ndn/edu/ucla/remap/demo/ndn-js-test/named-data.net/project/ndn-ar2011.html/%FDX%DC5B")
page.setFace(face)

def onSegmentedObject(namespace):
    print("Got segmented object size " + str(namespace.obj.size()))

page.setHandler(SegmentedObjectHandler(onSegmentedObject)).objectNeeded()
```

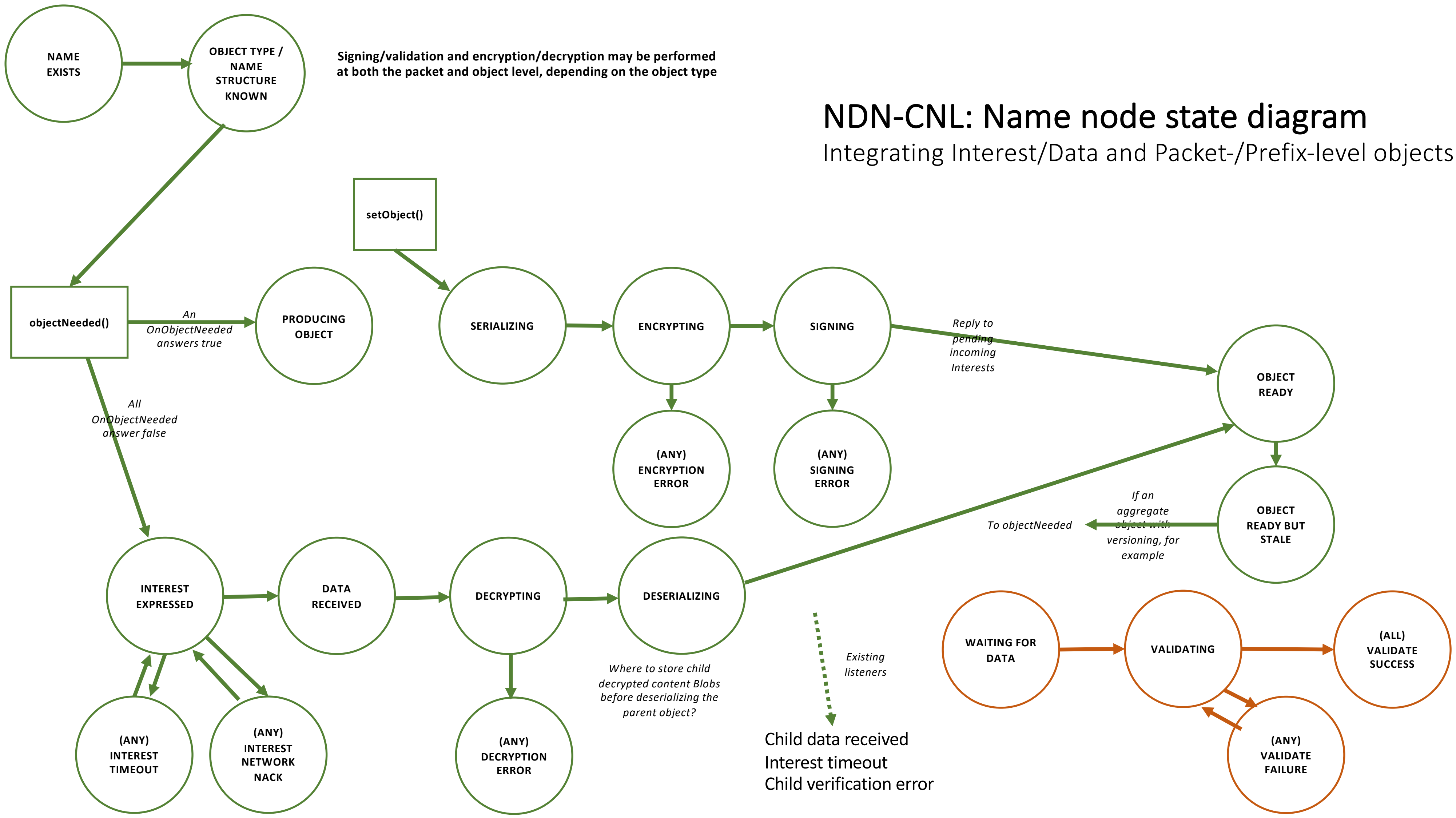
Unified publisher/consumer

- objectNeeded() – From application (producer) or network (consumer)
- Producer
  - CNL receives Interest, adds to PIT, calls OnObjectNeeded (if not already in cache).
  - Handler's OnObjectNeeded answers True.
  - CNL waits for application to produce data asynchronously.
  - Application calls setObject().
  - CNL does serialize/encrypt/sign and satisfies PIT.
- Consumer
  - Application calls OnObjectNeeded for a Namespace node.
  - (All handlers answer False.)
  - CNL does Face.expressInterest and waits for Data.
  - CNL receives Data, does verify/decrypt/deserialize and OnStateChanged(OBJECT_READY)

NDN-CNL: Name node state diagram

Integrating Interest/Data and Packet-/Prefix-level objects

Signing/validation and encryption/decryption may be performed at both the packet and object level, depending on the object type

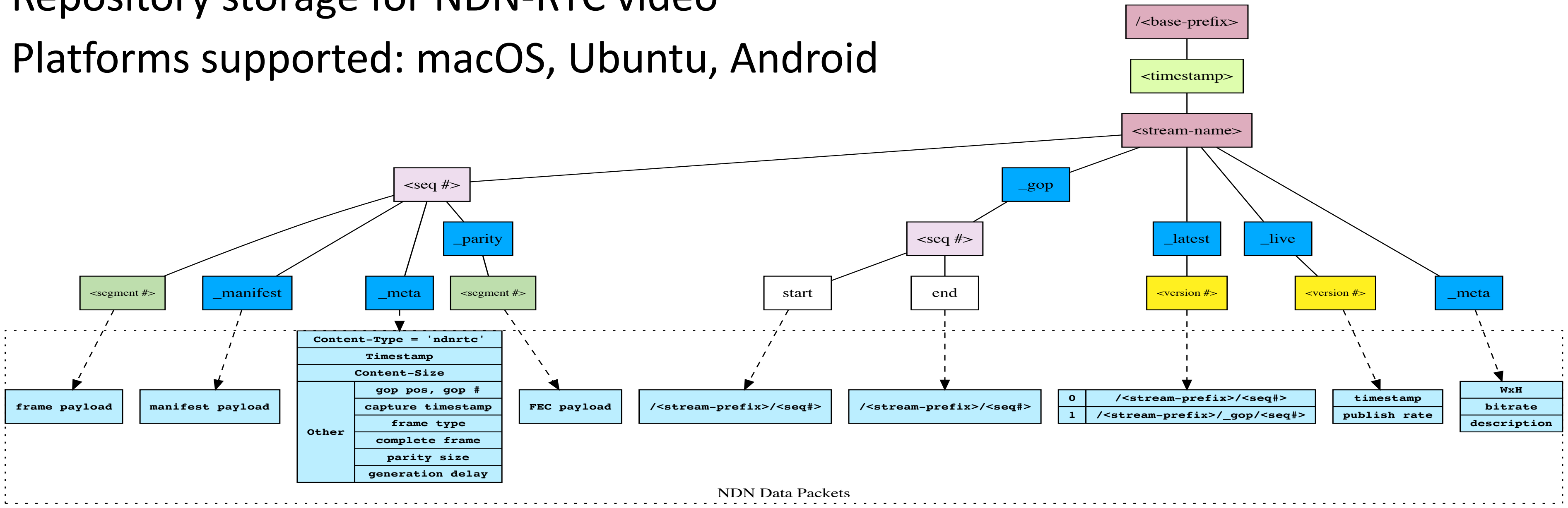


CNL – Next steps

- High-performance persistent storage
- Port to Java and JavaScript
- More applications
  - Currently used in augmented reality mobile client application

What is NDN-RTC?

- C++ video (HD) streaming library
- Sub-second (~150ms) latency
- VP9 video encoder
- Repository storage for NDN-RTC video
- Platforms supported: macOS, Ubuntu, Android



Applications

- ICE-AR (AR browser)
  - Offload phone POV video for edge processing (object, face, pose recognition)
  - Processed information delivered back to the phone to enrich phone’s environmental understanding (deep context)
- TouchNDN (theatrical live systems)
  - Based on the TouchDesigner media IDE https://www.derivative.ca
  - Live video dissemination over L2 to multiple nodes for simultaneous processing & storage
  - Nodes may perform “historical” streaming from a repo data, seamlessly with live streaming

NDN-RTC – Next Steps

- Incorporating VP9 SVC layers in the namespace
- Support Region-of-Interest-based fetching (360^{o} video use case)
- Volumetric video streaming

How to learn more

- Common Client Library (CCL)
  - C++: https://github.com/named-data/ndn-cpp
  - Python: https://github.com/named-data/PyNDN2
  - JavaScript: https://github.com/named-data/ndn-js
  - Java: https://github.com/named-data/jndn
  - C# (.NET Framework): https://github.com/named-data/ndn-dot-net
- PSync: Scalable Name-based Data Synchronization for Named Data Networking
  - https://named-data.net/publications/scalable_name-based_data_synchronization/
- Common Name Library (CNL)
  - C++: https://github.com/named-data/cnl-cpp
  - Python: https://github.com/named-data/PyCNL
- NDN-RTC: https://github.com/remap/ndnrtc