NDNts API Design

Junxiao Shi, yoursunny.com
Presented at IRTF interim-2021-icnrg-01, 2021-12-10

(NDNts is a personal project; this talk reflects personal opinions)
NDNts: NDN Libraries for the Modern Web

- Modern JavaScript libraries.
- Works in Node.js and browsers.
  - >90% test coverage.
  - Automated & manual browser tests on desktop / Android / iOS.
- Standalone without forwarder.
  - Or connect to NFD / NDN-DPDK.
- Actively maintained.
  - New features added regularly.
  - Support latest NDN specs.
What this talk is about

• My personal thoughts on NDN low-level API design.
  ✓ Low-level: packet decoding, fragmentation, "face", retransmission logic, etc.
  ○ Not low-level: "data centric toolkit", "common name library", etc.

• The unique challenges in building an NDN library for the web.
  • Code size is a primary concern.
  • The browser is like an OS, but it differs from a traditional OS.
Low-Level API is boring?

• Probably true.
  • Application developers are encouraged to use the high-level APIs, which abstracts NDN complexity away from the applications.

• Interacting with low-level API is unavoidable.
  • Developers who build high-level APIs would have to use low-level API.
  • High-level APIs do not cover all possible application needs.

• Therefore, it's still important to design a good low-level API.
Opportunities of NDNts

• NDNts is not the first library. I'm rarely the first to implement a feature. Instead, I prefer to:

  1. Write applications with the existing libraries.
  2. Look at how other developers are using the existing libraries.
  3. Feel the pain points of the existing libraries.
     • Which APIs are cumbersome to use?
     • Which code snippets are copy-pasted in multiple places because it's not in the library?
  4. Improve those areas in NDNts.

• NDNts is a personal project, so I can have the freedom.
  • I don't promise backwards compatibility.
  • I take my time to refactor, without worrying about deadlines.
  • I ask people to watch my push-ups over NDN testbed and collect metrics to improve NDNts congestion control implementation.
TLV Decoding

with TLV evolvability considerations
Example: NLSR LsaInfo structure

$LsaInfo = \text{LSA-TYPE TLV-LENGTH}$
Name
SequenceNumber
ExpirationTime

NDN spec: **considerations for evolvability of TLV-based encoding**

- If the decoder encounters an unrecognized or out-of-order element, the behavior should be as follows:

<table>
<thead>
<tr>
<th>TLV-TYPE number</th>
<th>expected behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>0~31</td>
<td>abort decoding and report error</td>
</tr>
<tr>
<td>least significant bit is 1</td>
<td>ignore TLV element and continue decoding</td>
</tr>
<tr>
<td>least significant bit is 0</td>
<td>ignore TLV element and continue decoding</td>
</tr>
</tbody>
</table>
NDNts: semi-declarative

```javascript
const EVD = new EvDecoder<Lsa>("LsaInfo", 0x80)
  .add(TT.Name, (t, { value }) => t.originRouter = new Name(value))
  .add(0x82, (t, { value }) => t.sequenceNum = NNI.decode(value, { big: true }))
  .add(0x8B, (t, { text }) => t.expirationTime = text);
```

**Evolvability-aware TLV decoder** (EvDecoder)

1. Declare each sub-TLV via `.add()` function.
2. Decode each sub-TLV with a lambda function.
   - It may include extra logic, such as saving signed portion boundary.
3. EvDecoder automatically handles evolvability considerations.
m_originRouter.clear();
m_seqNo = 0;

ndn::Block baseWire = wire;
baseWire.parse();

auto val = baseWire.elements_begin();

if (val != baseWire.elements_end() &&
    val->type() == tlv::Name) {
    m_originRouter.wireDecode(*val++);
} else {
    throw Error("OriginRouter: Missing required Name field");
}

if (val != baseWire.elements_end() &&
    val->type() == tlv::SequenceNumber) {
    m_seqNo = readNonNegativeInteger(*val++);
} else {
    throw Error("Missing required SequenceNumber field");
}

if (val != baseWire.elements_end() &&
    val->type() == tlv::ExpirationTime) {
    m_expirationTimePoint = time::fromString(readString(*val++));
} else {
    throw Error("Missing required ExpirationTime field");
}

• This decoding function does not support TLV evolvability.
python-ndn: declarative, reflection-based

class LsaInfo(TlvModel):
    originRouter = NameField()
    sequenceNum = UintField(0x82)
    expirationTime = BytesField(0x8B, is_string=True)

✓ Shorter than NDNts.
• Less flexible: cannot easily add extra logic.
• Class structure must follow TLV structure:
  • Application is exposed to encoding details.

Not yet in NDNts, but it's a direction to explore.
Endpoint, a better "face"
Traditional "face" vs NDNts Endpoint

- **Consumer:**
  - ✓ Interest retransmission
  - ✓ Data verification
- **Producer:**
  - ✓ prefix announcement
  - ✓ Data buffering
  - ✓ Data signing
  - ✓ multiple uplinks
  - ✓ connect to other NDNts apps
  - ✓ connect to forwarders
  - ✓ connect to IoT gadgets
  - ✓ automatic reconnecting

- **Handles manually:**
  - transport errors
  - Interest retransmission
  - signing & verification
  - InMemoryStorage

- **App:**
  - ✓ Interest-Data matching
  - ✓ prefix announcement

- **mgmt**

- **Face**

- **Packet demultiplexer**

- **Transport**

- **mgmt**
Consumer: Interest retransmissions

• NDNts: enable Interest retransmissions with one option.
  
  ```javascript
  try {
    const data = await endpoint.consume(interest, { retx: 2 });
    /* use retrieved Data */
  } catch { /* handle retrieval error */ }
  ```

• Other libraries: developer implements this flowchart manually.
Producer: Data buffering

- Use case: prepare a multi-segment response to one Interest.
  - Example: NFD management protocol dataset publisher.

- NDNts: automatic Data buffering.
  - Insert multiple Data packets to the buffer.
  - Subsequent Interests are satisfied from the buffer automatically.

```javascript
endpoint.produce("/prefix", async (interest, { dataBuffer }) {
  if (interest.name.at(-1).as(Segment) === 0) {
    /* generate all segments */
    await dataBuffer.insert(seg0, seg1, seg2);
  }
});
```

- Other libraries: developer queries InMemoryStorage for every Interest.
Data signing & verification

• NDNts: automatically sign outgoing Data and verify incoming Data.
  
  ```javascript
  const endpoint = new Endpoint({
    dataSigner: signer,
    verifier: verifier,
  });
  ```

• Both signer and verifier can be either:
  • a fixed key, or
  • a trust schema that chooses a key based on Data packet name.

• Other libraries: developer calls KeyChain & Validator manually.
Code size is a primary concern on the Web

Every KB of code must be downloaded over the network.
- Visitors expect the webpage to load within 5 seconds "time to interactive".
- Code size budget: 170KB minified & gzipped.

How I'm solving this problem in NDNts?
- Reduce core features that are always loaded.
- If an app needs an extra feature, import the module and pay the cost:
  ```javascript
  const endpoint = new Endpoint({
    dataBuffer: new DataBuffer(new DataStore(memdown())),
  });
  ```
- Trade-off between API simplicity and webpage performance.
## Transport support matrix

<table>
<thead>
<tr>
<th>libraries</th>
<th>protocol</th>
<th>forwarders and more</th>
</tr>
</thead>
<tbody>
<tr>
<td>ndn-cxx</td>
<td>Unix socket</td>
<td>NFD</td>
</tr>
<tr>
<td>python-ndn</td>
<td>memif</td>
<td>●</td>
</tr>
<tr>
<td>NDNts (Node.js)</td>
<td>Ethernet</td>
<td>●</td>
</tr>
<tr>
<td>NDNts (browser)</td>
<td>UDP</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>TCP</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>WebSocket</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>HTTP/3</td>
<td>○</td>
</tr>
<tr>
<td></td>
<td>WebRTC</td>
<td>○</td>
</tr>
<tr>
<td></td>
<td>Bluetooth</td>
<td>*</td>
</tr>
</tbody>
</table>
KeyChain & Crypto
KeyChain: Web Crypto & IndexedDB

```javascript
const keyChain = KeyChain.open("homecam");

const [pvt, pub] = await generateSigningKey(keyChain, subjectName);

const cert = await requestCertificate({
    profile: caProfile,
    publicKey: pub,
    privateKey: pvt,
    validity: ValidityPeriod.MAX,
    challenges: [new ClientNopChallenge()],
});

await keyChain.insertCert(cert);
```

- Open IndexedDB for storing keys and certificates.
- Generate non-extractable keys via Web Crypto API and store in IndexedDB.
- Request a certificate from a remote certificate authority, using NDNCERT (NDN certificate management protocol).
- Save the received certificate.
Web Crypto requires Secure Context

• Webpage must be delivered over HTTPS to use Web Crypto.
  • Required by Web Crypto spec.
  • Enforced in Chrome.
  • Not enforced in Firefox.

• Why bother with plain HTTP?
• "Coffee shop hotspots" are still popular in less developed countries.
  • The locals are sharing files and chatting over those hotspot networks.
  • No Internet, no DNS, cannot obtain trusted TLS certificates.

• NDNts security features will not work in this environment.
Web Crypto has limited algorithms

✓ SHA-256
✓ ECDSA, ECDH
✓ RSA PKCS#1, RSA-OAEP
✓ AES-CBC, AES-GCM
✓ PBKDF2
  o BLAKE2b, required in Pollere DCT
  o EdDSA, required in Pollere DCT
  o AES-CCM, an option of FLIC rev03
  o ChaCha20-Poly1305, an option of ndn-ind

Despite being an option, if an existing application chooses an algorithm, NDNts needs to have the algorithm to be able to interoperate with that app.
Alternatives to Web Crypto

- asmcrypto.js and other JavaScript crypto libraries
- Rust crypto compiled as WebAssembly module

**Drawbacks:**
- Code size concerns.
- Keys are unprotected (vs. non-extractable keys in Web Crypto).
- No effective way to cleanse memory.

**Drawbacks, when delivered over plain HTTP:**
- Code can be modified by MITM attacker, completely compromising security.
- Lack of secure random number generator.

- So far, NDNts is limited to Web Crypto only.
Naming a Browser

"Name is the secret sauce of NDN"
Naming a browser for anonymous users

```javascript
await generateSigningKey(keyChain, subjectName);
```

Where does this name come from?

- My current webapps use random names:
  1. Generate a random identity name during the first visit.
  2. Request a certificate and store it in the KeyChain.
  3. Reuse the same identity name during subsequent visits.
  4. Start over if the certificate expires or the KeyChain is deleted.

- This only works for anonymous users.
Naming a browser with user authentication

• Username+password / "Email me a magic link".
  • Obtains a short-lived certificate from a server-controlled CA.

• OpenID / OAuth / WebAuthn, but do it over NDN.
  • Interacts with a downloadable or self-hosted "NDN authenticator" app, which contains a user-controlled CA.

• User experience must be streamlined.
  • Visitors do not care whether the webpage is using NDN.
Start Coding with NDNts

• NDNts homepage: https://yoursunny.com/p/NDNts/
• Getting Started tutorials on yoursunny.com blog
• API documentation available in Visual Studio Code IDE

• NDN Play https://play.ndn.today
  • Web simulator for NDN, built with NDNts