NDN, Technical Report NDN-0024, 2014. http://named-data.net/techreports.html
Revision 1: September 5, 2014

NDN Common Client Libraries

Jeff Thompson, Jeff Burke
{jefft0, jburke}@remap.ucla.edu

1 Introduction

This technical report supplements and refers to the online documentation for NDN-CCL located at `http://named-data.net/doc/ndn-ccl-api`.

The NDN Common Client Libraries (CCL) provide a common application programming interface (API) across several languages for building applications that communicate using Named Data Networking (NDN) [2]. Currently, the CCL is implemented in C++, Python, JavaScript and Java. This technical report describes general design goals, the CCL API and specific design choices for each language implementation.

NDN development is supported by the Future Internet Architecture program of the National Science Foundation. The proposal for the current stage of research states that it is crucial to develop and release libraries and tools, created based on application development experience, that promote ongoing team and community experimentation with NDN.

Historically, the API design began with a library called PyCCN that provided an object-oriented interface to the underlying C code of the CCNx library with the goal of representing important architectural primitives of Named Data Networking in classes, such as the `Name`, `Interest`, `Data`, and `Key` now found in NDN-CCL. This path of development continued with early versions of the JavaScript library (NDN-JS) which added more data manipulation functions needed to support browser applications. Then, NDN-CPP was introduced as an implementation of NDN-CCL using C++ that intends to combine higher performance with the Python and Javascript libraries' ease-of-use. Finally, the C++ library (NDN-CPP) added the Blob class (see below), a formal security API to sign and verify packets, and is the reference library for developing higher-level functionality like Sync and the in-memory content cache.

See Section 5 for a list of applications currently using NDN-CCL.

1.1 Relationship to NDN-CXX

In addition to NDN-CPP, the C++ implementation of the Common Client Library, the NDN project team also maintains the NDN-CXX library, "C++ with eXperimental eXtensions". This library is used in the NFD forwarder and other core implementations. It does not have the constraint of supporting multiple languages and makes extensive use of the Boost C++ libraries. It is also used for prototyping of new architectural features, which are then incorporated into the NDN-CCL.

2 Design Goals

To support as many developers of NDN applications as possible, the libraries were designed with the following goals.

1. *Multiple software languages.* Support multiple languages as needed by the NDN research project and community. NDN-CCL currently supports:
   - *C++.* C++ as a strongly typed reference implementation and library for high-performance applications. It also serves as the target language for importing experimental extensions.
   - *Python.* Python to make experimentation and rapid prototyping easy.
   - *JavaScript.* JavaScript because it provides inherent support for asynchronous programming styles natural to NDN application development, because it runs directly in Web browsers which enables applications to be easily deployed on many network-capable devices and, as with the ease of programming in Python, to encourage development and experimentation with NDN for a larger audience of developers.

- *Java.* Java to enable mobile applications on Android devices.

2. *Keep the API consistent across languages.* While different software languages need some unique features such as the type of byte array, the names of classes, methods and overall design should be consistent. New developments in the library for one language should be quickly migrated to the other languages.

3. *Keep dependencies to a minimum.* This makes for easier installation and less likelihood of a conflict with a client application's other dependencies. Since we use open source software, if depenent code is small then its source can be incorporated directly into the library, with proper license and author attribution.

4. *Provide an easy-to use API.* (See below.) The library should provide a developer-friendly API that eases application development with the NDN architecture. The interface should provide a level of abstraction that hides protocol details (packet format and encoding, etc.) while still reflecting the general model of NDN communication.

5. *Follow standard release practices.* In standard practice, obsolete functionality is deprecated in minor releases and only removed at a later major release. This stability allows developers to focus on their NDN research and not on rewriting unrelated parts of their code.

6. *Use a release "heartbeat".* This stimulates interoperability testing and discussion of how the various moving parts work together.

7. *Implement the NDN-TLV wire format.* This of course allows an application to communicate with the NFD forwarder [1] as well as ndn-tlv [2]. As of writing the libraries implement NDN-TLV version 0.1.1. When version 0.2 is released, the libraries will implement this with the option to revert as needed. This stability again allows developers to focus on their NDN research and not on rewriting unrelated parts of their code.

8. *Incorporate advances from NDN research projects.* (For example, security, Sync and autodiscovery.) This allows devopers to more easily pull these capabilities into their applications.

9. *Follow the NDN Platform Development Guidelines*[3] These guidelines for documentation, code style and licensing encourage consistency across the libraries and related projects.

10. *Provide installation packages.* (For example, easy_install for Python.) This lowers the entry barrier for end users of applications which depend on the libraries.

3 The CCL API

The application programming interface (API) of the NDN Common Client Libraries (CCL) is documented online at `http://named-data.net/doc/ndn-ccl-api`. The important entities in NDN, such as Interest packets, Data packets Keys, and Names, are abstracted as API objects. A security API provides data signing and verification functionality required by the architecture, and trust management support based on the ptototype NDN-CXX security library is under development.

The API documentation provides a unified reference for all the implemented languages: C++, Python, JavaScript and Java. The class names, method names and parameters are the same in all the languages. So, for example, a developer can prototype an application in Python and later port it to C++ (for performance improvements) with minimal effort. The unified API documentation also provides a place to describe library support for higher-level and experimental NDN functionality such as Sync and an in-memory cache. (The experimental status of an API is emphasized with a red bar on the left side of the documentation page.)

Figure 1 shows the CCL library architecture. Each box in the diagram represents a class which implements part of the NDN architecture. The connecting lines show how objects of a class use objects of another class. The main box contains classes which represent objects in the NDN Packet Format Specification.

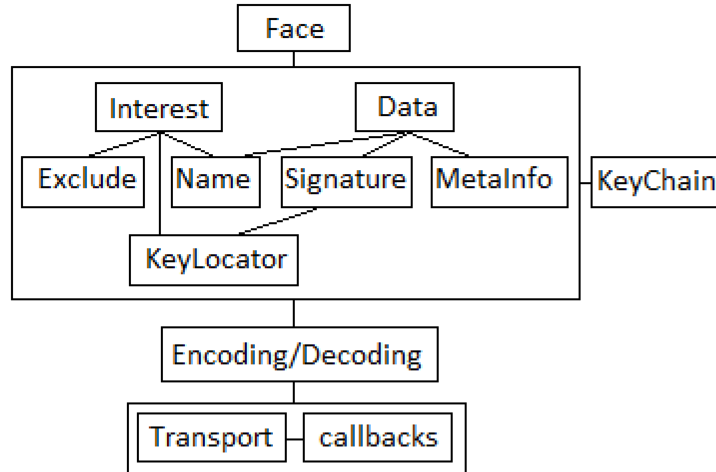

Figure 1: Library architecture

3.1 Interest / Data Exchange

The `Face` class (at the top of figure 1) provides the top-level interface to the library. It holds a connection to a (typically local) forwarder and supports Interest / Data exchange. The `Face` constructor can take optional connection parameters such as a host name or Unix socket address. The `Face` class also provides the interface to create command interests (section 3.7).

`Face` has two important methods for Interest / Data exchange: `expressInterest()` for data fetching and `registerPrefix()` for publishing. Both methods incorporate an event-driven asynchronous programming paradigm and require the caller to provide callbacks for event handling.

`expressInterest()` composes an Interest packet based on the information provided by the caller and sends this packet to the forwarder to which the `Face` object is connected. When the library receives an incoming Data packet which matches the Interest, it calls the `onData` callback provided to `expressInterest()` to supply the Data packet to the application.

`registerPrefix()`, registers an NDN Name prefix by sending a *Command Interest* packet to the forwarder and then waits for incoming Interests that request the data under that prefix. When an Interest is received, the library calls the `onInterest` callback provided by the application to `registerPrefix()` and the application can send the related Data packet.

3.2 Representing Interest and Data Packets

The `Interest` class represents an Interest packet. As detailed in the NDN Packet Format Specification, an Interest holds a `Name` and its selectors can hold an `Exclude` object and a `KeyLocator`. (Other values such as Nonce or Scope are represented by a byte array Blob (section 4.1.1) or a simple number.)

The `Data` class represents a Data packet. As detailed in the NDN Packet Format Specification, a Data packet holds a `Name`, a `MetaInfo`, a `Signature` object and a Content (represented by a byte array Blob (section 4.1.1)).

As detailed in the NDN Packet Format Specification, the Signature is further specified based on the signature algorithm, for example `Sha256WithRsaSignature` class which extends the `Signature` class. (This is one of the few places in the CCL API that uses class inheritance. Generally, the class hierarhy is very flat.) In this case, Sha256WithRsaSignature can hold a `KeyLocator`. The `Interest` and `Data` make use of the same `Name` and `KeyLocator` classes, as shown by the common lines in figure 1.

The APIs for `Interest` and `Data` are independent of a specific wire format encoding, but it is possible to explicitly call `Interest.wireEncode()`/`Data.wireEncode()` or `Interest.wireDecode()`/`Data.wireDecode()` to manipulate an encoding. These are also implicitly called by the `Face` object when using the wire Transport interface (at the bottom of figure 1).

3.3 Signing and Verification

In NDN, every Data packet is signed. Interests can be optionally signed in application-specific ways; for example, *Command Interests* used to communicate with the NFD forwarder are signed by adding signatures bits as a name component. Signing and verification is handled by the `KeyChain` class of the security API (at the right of figure 1). The default `KeyChain` constructor uses the same public and private key stores as the system's local NFD forwarder.

An application may create a new `Data` object, for example within the `onInterest` callback provided to `registerPrefix()`. Before sending, it must be signed by calling `KeyChain.sign()`. The `sign()` method takes a certificate name parameter which specifies the name of the key to fetch from the private key store for signing, as explained in the Security Library Tutorial[4]. This name is also used to create the KeyLocator information in the Data packet's Signature. Although there is a `KeyChain.sign()` method for Interests, a Command Interest is typically signed by `Face.makeCommandInterest()`. (See section 3.7).

The signature on a Data packet is not automatically verified when the library receives it. Rather, if the application wants to verify then it must explicitly call `KeyChain.verifyData()`, typically within the `onData` callback provided to `expressInterest()`. Because verification may require communication to fetch certificates, `verifyData()` itself is asynchronous and requires `onVerified` and `onVerifyFailed` callbacks to indicate a successfull or failed verification. Likewise, if the application wants to verify a Command Interest then it must explicitly call `verifyInterest()`. (See section 3.7).

3.4 Client configuration file

For compatibility with NDN-CXX and NFD, the NDN-CCL respects the relevant settings of */.ndn/client.conf*, including the keystore type and unix socket location, for its own defaults.

3.5 Sync

Sync is a higher level protocol of the NDN architecture for efficient reconciliation of namespace knowledge across many peers. The protocol definition is still an active research topic, which may yield multiple forms of synchronization. To support developers in experimenting with the concept, NDN-CCL implements a variation of Sync compatible with Chronosync, described in [5], as an experimental API.

3.6 Name Convention Support

3.6.1 Versioning, Segmenting and Timestamping

NDN-CCL provides helper methods in each language for the versioning, segmenting, and timestamping name conventions defined in [3]. The `Name` class has methods `appendSegment`, `appendSegmentOffset` (for segment byte offset), `appendSequenceNumber`, `appendTimestamp` and `appendVersion`, each of which take an integer and append a name component with the appropriate marker followed by the encoded integer. These methods let an application construct a name with the needed special name components.

Likewise, when the application receives a name, it can one of the name components and call the method for `toSegment`, `toSegmentOffset`, `toSequenceNumber`, `toTimestamp` or `toVersion`. Each of these checks that the component has the expected marker, parses the encoded integer value and returns it. These methods let an application interpret a name with special name components which was produced by another application using the previous "append" methods.

3.7 Command Interests

NDN-CCL implements the specification for command interests used by the NFD forwarder and defined in "Signed Interest". To enable creating command interest, an application calls the `Face` method `setCommandSigningInfo` to supply the `KeyChain` object and related certificate name used to sign interests. Once the command signing info is set, each time an application needs to sign an interest it calls `makeCommandInterest` which takes an interest name with existing command components and appends components for a timestamp, a random value and the SignatureInfo. Then it signs all the name components so far and appends a final signature component. (The `Face` class also calls `makeCommandInterest` internally to sign the command interest for `registerPrefix`.)

When the application calls `makeCommandInterest`, the interest name must already have the command components which may be a simple string like "register" or a TLV encoding of a more complex structure like

a FibEntry. To create a custom TLV encoding, an application can use the support for the protocol buffer definition format described in section 3.9.

To verify a received command interest, an application calls the `KeyChain` method `verifyInterest`, similarly to `verifyData` described above (section 3.3).

3.8 Memory Content Cache

We have observed that many NDN applications follow a pattern in which the application would like to "publish" data once, and then have some other entity be responsible for sending it out in response to Interests. NDN Repositories provide persistence beyond the lifetime of the application process but require a separate process and may add performance overhead. For data that does not necessarily need to persist after the application process ends or needs to be sent quickly while a repo write operation may still be in progress, the NDN-CCL provides the `MemoryContentCache` class, an in-memory content store that responds asynchronously to requests received by the library.

From the time that a data packet is added to the `MemoryContentCache`, it is retained based on its freshness period, after which it expires and is removed from the cache and not used to answer interests. This is useful for data packets representing timely information such as the frames of a streaming video. Since expired data is removed from the cache, an interest's "must be fresh" selector does not apply and is ignored. (If the optional freshness period is not specified, the data packet is retained indefinitely.)

When the `MemoryContentCache` receives an interest without a "child selector", it simply returns the first data packet whose name matches based on the "min suffix components", "max suffix components" and "exclude" selectors (if present). If the interest has a "child selector", it is applied after filtering based on the other selectors.

This cache may duplicate entries in a local forwarder's content store. In the future, the NFD forwarder may provide an API for its own content store, which could be wrapped by this class to provide a more memory-efficient option.

3.9 Protobuf-TLV Messages

To provide a language-independent way of defining messages, the NDN-CCL enables developers to describe messages in the well-known *protocol buffer*[5] definition format, and then creates encoding/decoding helper classes in any of the NDN-CCL languages. Instead of encoding/decoding in the protocol buffer format encoding, NDN-CCL provides the `ProtobufTlv` utility to take a protocol buffer definition and encode/decode in NDN-TLV. This allows an application to define custom structured parameters of a signed command interest, or a structured payload of a data packet. The application only needs to write a protocol buffer definition, populate an in-memory object using the protocol buffer API, and call the library converter to/from NDN-TLV.

4 Implementation Design Choices

This section describes the design choices for API features implemented in all the software languages and provides details for each implemented software language. In general, design choices are made to meet the goals of portability and minimal dependencies on external libraries.

4.1 All Languages

4.1.1 Generic Byte Arrays with `Blob`

The Common Client Libraries API for all languages uses a `Blob` class which holds an immutable byte array and hides the details of how the byte array is implemented in the language. (For example, in Python version 2 a byte array is the same as a string, but Python 3 implements a true byte array since a string object was changed to hold Unicode characters.)

If an object uses mutable byte arrays (for example the component values of a name), then the library would need to make a copy of the arrays when it stores the object (for example in the internal pending interest table). This is because the application may change the values in the byte arrays causing unpredictable behavior. But if the object uses immutable byte arrays, then the library does not need to make copies. It is more efficient to avoid a data copy (especially for large byte arrays like the content payload of a data packet) and cleaner semantics (for the same reasons that a String is immutable in languages like Java and Python).

Furthermore, even though an object like `Name` is mutable, it is relatively cheap to make a copy of it since each name component is an immutable `Blob` so it is only necessary to do a shallow copy.

The `Blob` class also provides a common API for methods like `size`, `isNull` and `toHex` which make it easier to port an NDN application from one software language to another.

4.1.2 Event processing with `processEvents`

JavaScript comes with a built-in event loop which automatically checks for incoming network data and to provide a timer to call a function after a delay. But the other software environments (C++, Python and Java) don't have native support for an event loop. So, to make the libraries as portable as possible, the API has the function `processEvents` which the application must call frequently so that the library can check for incoming network data and call any delayed functions (such as an interest timeout).

Also, JavaScript is single-threaded so it inherently avoids cross-threading problems. But the other languages (C++, Python and Java) support multible threads (but C++ and Python support it differently based on the implementation). So, requiring the application to call `processEvents` helps with this problem too. The application is required to call `processEvents` in the same thread that it calls `expressInterest` or other methods which manipulate the same data structures as `processEvents` (such as the pending interest table).

Although C++, Python and Java don't have native support for an event loop, there are prominent support utilities. For example, C++ can use Boost ASIO and Python can use asyncio (or Trollius in Python 2). These would change the core of the library since the choice of event loop must be allowed to "replace" the underlying network I/O and timer code. We are looking for a way to support this while still retaining the option for the `processEvents` approach.

4.2 C++ (NDN-CPP)

4.2.1 C core

While NDN-CPP provides an object-oriented C++ API for advanced applications, its core functionality is implemented in C. The core C code makes few assumptions about memory management or linked library support in order to promote use on a range of platforms from embedded processors up to high-performance routers.

4.2.2 C++ language features

In addition to the normal C++ class definitions for `Name`, `Interest`, etc., NDN-CPP supports the following C++-specific features.

- *Name as an array of components.* The `Name` class supports `operator[]` so that you can more conveniently access a name component with `name[i]` instead of the CCL `name.get(i)`.
- *Name and Name.Component equality and comparison operators.* The `Name` and `Name.Component` classes support equality operators `operator==` and `operator!=` to check `Name.equals` and `Name.Component.equals`, and comparison operators `operator<`, etc. for `Name.compare` and `Name.Component.compare` so that you can more conveniently write, for example, `if (name1 == name2)` or `if (name1[3] < name2[3])`.
- *Name stream output.* The stream output operator `operator<<` is overloaded for `Name` to call `toUri` so that you can more conveniently write, for example, `cout << name1` to print the URI.
- *Exclude as an array of entries.* The `Exclude` class supports `operator[]` so that you can more conveniently access an exclude entry with `exclude[i]` instead of the CCL `exclude.get(i)`.

4.2.3 Shared pointers

There are many places in the CCL API where the application needs to allocate an object and pass it to the library for the it to use and delete, or vice versa. For example when the library receives a data packet for an interest, it allocates a Data object and passes it to the application through a callback function, where the application is responsible for deleting it. Since an object is referred to in multiple places, it is not enough to simply transfer "ownership" to the part of the application which needs to delete it since the library would need to waste time and memory to make copies of the object if it still needs to use it. Therefore, NDN-CPP makes use of `shared_ptr`, which is a reference-counting utility class designed to solve this common problem.

The `shared_ptr` class is defined in three possible places: If the compiler is compliant with C++ 11, then the standard library defines `std::shared_ptr`. If the developer has installed Boost, then it defines

boost::shared_ptr. Finally, NDN-CPP includes the extracted files from Boost with the renamed namespace ndnboost::shared_ptr.

The header file ndn-cpp/common.hpp defines the alias ptr_lib which is used throughout NDN-CPP as ptr_lib::shared_ptr. By default, if the compiler has C++ 11, then ptr_lib is defined as std. Otherwise if Boost is installed it is defined as boost. The C++ 11 standard library takes precedence over Boost because the developer may have installed Boost for utilities other than boost::shared_ptr. Finally, if neither is available then ptr_lib is defined as the fallback ndnboost which means that NDN-CPP will compile even if neither C++ 11 or Boost is installed. The default for ptr_lib can be changed with the ./configure options --with-std-shared-ptr and --with-boost-shared-ptr.

4.2.4 Function objects

The CCL API uses callback functions. For example, expressInterest takes a parameter onData which is called when the library receives a data packet for an interest. For a callback in C, an application will typically pass a simple function pointer, but this requires it to also pass an extra pointer to a "context" containing the dynamic data needed by the callback function. C++ solves this common problem with the bind utility which creates a "function object" that automatically binds a function pointer to its context in a single parameter.

As with shared_ptr above, the bind utility is defined in three possible places: If the compiler is compliant with C++ 11, then the standard library defines std::bind. If the developer has installed Boost, then it defines boost::bind. Finally, NDN-CPP includes the extracted files from Boost with the renamed namespace ndnboost::bind.

And simiar to shared_ptr above, the header file ndn-cpp/common.hpp defines the alias func_lib which is used throughout NDN-CPP as func_lib::bind. By default, if the compiler has C++ 11, then func_lib is defined as std. Otherwise if Boost is installed it is defined as boost. Finally, if neither is available then it is defined as the fallback ndnboost. The default for func_lib can be changed with the ./configure options --with-std-function and --with-boost-function.

Although the support for shared_ptr and func_lib is similar, they are kept separate because an application may start out using Boost for both, then as it upgrades to use C++ 11, it may for example update the application code to use bind from the standard library but continue to use Boost for shared_ptr. Therefore, NDN-CPP supports these separately.

4.2.5 Obtaining NDN-CPP

NDN-CPP can be obtained from http://github.com/named-data/ndn-cpp.

4.3 Python (PyNDN)

The PyNDN implementation in CCL aims to promote rapid, iterative development and better support for new users of the architecture. It has evolved from Python wrappers around C code to be a pure Python implementation, which aids cross-platform development. In the future, a higher performance version of the library may be introduced that wraps NDN-CPP.

4.3.1 Python language features

In addition to the normal Python class definitions for Name, Interest, etc., PyNDN supports the following Python-specific features.

- *Name as an array of components.* The Name class supports __getitem__ (where the key is an int) so that you can more conveniently access a name component with name[i] instead of the CCL name.get(i).
- *Name and Name.Component equality and comparison operators.* The Name and Name.Component classes support equality operators __eq__ and __ne__ to check Name.equals and Name.Component.equals, and comparison operators __lt__, etc. for Name.compare and Name.Component.compare so that you can more conveniently write, for example, if (name1 == name2) or if (name1[3] < name2[3]).
- *Name length.* The Name class supports the Python idiom for __len__ so that you can more conveniently get the name's number of components with len(name) instead of the CCL name.size().
- *Exclude as an array of entries.* The Exclude class supports __getitem__ (where the key is an int) so that you can more conveniently access an exclude entry with exclude[i] instead of the CCL exclude.get(i).
- *Exclude length.* The Exclude class supports the Python idiom for __len__ so that you can more conveniently get the exclude's number of entries with len(exclude) instead of the CCL exclude.size().

- *Blob length.* The `Blob` class supports the Python idiom for `__len__` so that you can more conveniently get the blobl's size with `len(blob)` instead of the CCL `blob.size()`.
- *Blob to string.* Because PyNDN supports both Python 2 and 3 (see below), the `Blob` class supports the string operator `__str__`. In Python 2, a `str` is simply an byte array, so this directly converts the blob to a raw string. In Python 3, a `str` is a Unicode string, so this decodes the byte array as UTF8 and returns the Unicode string.

4.3.2 Unified support for Python 2 and 3

Python 3 has enough differences from Python 2 that some Python projects provide different libraries for applications developed with Python version 2 vs. 3. However, the CCL API provides a relatively high level of abstraction from the lower-level details where differences between Python versions 2 and 3 matter. Therefore, the CCL for Python (PyNDN) provides a single library to support both versions, using runtime checks where needed to handle the low-level differences.

Futhermore, the CCL API already provides the `Blob` class (section 4.1.1) which provides an abstraction for a byte array which is one of the differences between Python versions 2 and 3. The Blob constructor accepts many types of array and converts to a consistent internal representation based on the Python version.

4.3.3 Obtaining PyNDN

PyNDN can be obtained from `http://github.com/named-data/PyNDN2`, either directly from the repository or via instructions for package installation using `easy_install`.

4.4 JavaScript (NDN-JS)

NDN-JS is a pure JavaScript client library, previously called "lwNDN". As mentioned above, it was originally intended to support browser-based experimentation with NDN but has also grown into an important tool for developing user interfaces to NDN applications. It is described in more detail in [1] and [4].

4.4.1 Javascript language features

In addition to the normal JavaScript class definitions for `Name`, `Interest`, etc., NDN-JS supports the following JavaScript-specific features.

- *Face constructor parameters.* In addition to the constructor which takes an explicit transport and connectionInfo, the `Face` class supports the JavaScript constructor idiom of a single parameter which is an associative array of optional parameters. See the JavaScript-only section of the constructor for a default Transport.

4.4.2 Buffer wrapper for the browser

For JavaScript, we want a single library to support applications in both the browser and in Node.js. The main differences are in socket communication and in the type of object to represent a byte array. For socket communication, the CCL API already provides the `Transport` class abstraction so we use a different subclass for communication in the browser and in Node.js.

But for byte array objects, the browser uses `Uint8Array`[6] and Node.js uses `Buffer`[7] which are incompatible. Therefore, in the browser we use a wrapper class with the same API as the `Buffer` class, but uses a `Uint8Array` underneath. Since JavaScript is loosely typed, we can always refer to `Buffer` throughout the JavaScript library code, whether it is used in the browser or Node.js.

4.4.3 Support for Node.js

The NDN-JS library can also be used in to create standalone Javascript applications outside of the browser through Node.js.

4.4.4 Obtaining NDN-JS

NDN-JS can be obtained from `http://github.com/named-data/ndn-js`, either directly from the repository or via instructions for package installation on Node.js using `npm`.

4.5 Java (jNDN)

The Java library, jNDN, aims to provide a native Java implementation of the CCL. It is currently motivated by the need to provide support for Android development as part of the NDN research project.

4.5.1 ByteBuffer

Java provides many types of object to represent an array of numbers, such as `byte[]`, `int[]`, `ByteArrayOutputStream` and `ByteBuffer`. For efficiency reasons, we want a type of object which can map directly to an underlying block of memory, which rules out `byte[]` and `int[]`. `ByteArrayOutputStream` has the option to use an underlying block of memory but is restricted to stream-style operations and doesn work with NIO. Therefore, we use `ByteBuffer`.

`ByteBuffer` also has the nice feature of a slice operation which shares the memory of a subsequence, which can safely be used with the immutable `Blob` class (section 4.1.1). It is OK to pass a new `Blob` which shares a subsequence with the original `Blob` since the bytes won't be changed.

4.5.2 Obtaining jNDN

jNDN can be obtained from `http://github.com/named-data/jndn`.

5 Usage

The NDN-CCL is currently used by the following applications and application research projects:

- **CCNx Federated Wiki**, an NDN port of the Smallest Federated wiki (NDN-JS)
- **Chronochat-js**, a javascript implementation of the ChronoChat demonstration application (NDN-JS)
- **Matryoshka**, an experimental multi-player online game using NDN and the Unity3D game engine. (jndn as the basis of the .NET port of CCL used in this project.)
- **ndn-bms**, a building management system prototype being developed as part of the NDN-NP project (PyNDN, NDN-JS)
- **ndn-lighting**, lighting control application using NDN (PyNDN, NDN-JS)
- **ndn-protocol**, a firefox browser plug-in supporting an ndn:/ retrieval scheme (NDN-JS)
- **NDNEx**, an NDN-based mobile health application being developed as part of the NDN-NP research project. (jndn)
- **ndnfs** and **ChronoShare**, NDN file sharing platforms (PyNDN)
- **NDNoT**, the Named Data Network of Things toolkit for the Raspberry PI (PyNDN, NDN-JS)
- **ndnrjs**, a javascript implementation of an NDN repository (NDN-JS)
- **ndnrtc**, a peer-to-peer multiparty audio, video, and chat application over NDN. (NDN-CPP, NDN-JS)
- **ndnstatus**, the NDN routing status web page (PyNDN, NDN-JS)
- **NDNVideo**, a video playout application for NDN (PyNDN)
- **OpenPTrack-NDN** an open source person tracking system that will add NDN support in Fall 2014. (NDN-CPP)

Please contact the authors to add other applications to this list.

6 Conclusion

The NDN-CCL is an actively developed set of libraries for writing Named Data Networking applications. The developers invite feedback on the approach, API, and implementation.

6.1 Future Work

Future work under discussion includes .NET framework support, a prototype for which has been developed to support the Matryoshka[8] multiplayer online game project.

Additionally, NDN-CCL will continue to incorporate new research in the NDN project, including architectural evolution, security and applications results, and API concepts.

6.2 Acknowledgements

Many people have contributed to the NDN-CCL, including: Ryan Bennett, Alex Horn, Felix Rabe, Anmol Rajpurohit, Wentao Shang, and Zhehao Wang. The library also makes heavy use of features first introduced in NDN-CXX and the NDN Security Library, whose primary authors include Alex Afanasyev, Junxiao Shi, and Yingdi Yu. PyNDN is based on code by Derek Kulinski and Jeff Burke. NDN-JS is based on code by Meki Cherkaoui and Axel Colin de Verdiere, later updated by Wentao Shang.

Notes

[1] http://named-data.net/doc/NFD/current/
[2] http://redmine.named-data.net/projects/ndnd-tlv/wiki
[3] http://named-data.net/codebase/platform/documentation/ndn-platform-development-guidelines/
[4] http://redmine.named-data.net/projects/ndn-cxx/wiki/SecurityLibrary
[5] https://developers.google.com/protocol-buffers
[6] https://developer.mozilla.org/en-US/docs/Web/API/Uint8Array
[7] http://nodejs.org/api/buffer.html
[8] https://github.com/zhehaowang/NDNMOG-live

References

[1] Wentao Shang, Jeff Thompson, Meki Cherkaoui, Jeff Burke, and Lixia Zhang. NDN.JS: A javascript client library for Named Data Networking. In *Proceedings of IEEE INFOCOMM 2013 NOMEN Workshop*, April 2013.

[2] NDN Project Team. Named data networking (ndn) project. Technical Report NDN-0001, Revision 1, NDN, October 2010.

[3] NDN Project Team. Naming conventions. Technical Report NDN-0022, Revision 1, NDN, July 2014.

[4] Jeff Burke Wentao Shang, Jeff Thompson and Lixia Zhang. Development and experimentation with ndn-js, a javascript library for named data networking. Technical Report NDN-0014, Revision 1, NDN, August 2013.

[5] Zhenkai Zhu, Alexander Afanasyev, and Lixia Zhang. Let's ChronoSync: Decentralized dataset state synchronization in Named Data Networking, 2013. under submission.