NDN Technical Report NDN-0014, 2013. http://named-data.net/techreports.html
Revision 1: August 31, 2013

Development and Experimentation with NDN-JS, a JavaScript Library for Named Data Networking

Wentao Shang*, Jeff Thompson†, Jeff Burke† and Lixia Zhang*
*Department of Computer Science, UCLA. {wentao,lixia}@cs.ucla.edu.
†Center for Research in Engineering, Media and Performance, UCLA. {jefft0,jburke}@remap.ucla.edu.

***Abstract*—NDN-JS is a pure JavaScript implementation of a client-side library for Named Data Networking (NDN). The initial purpose of designing this library was to simplify the development and deployment of NDN applications. It enables developers to create Web applications using the NDN protocol and to deliver them to end users without the installation of the CCNx package. This report describes how NDN-JS implements the core protocol functions and provides a set of high-level APIs that abstract the communication primitives and packet processing operations. It also includes a performance evaluation which identifies a throughput bottleneck caused by the RSA public key operations implemented in JavaScript, and introduces some optimizations to mitigate them. The library has been successfully adopted inside and outside the NDN project team to create various Web applications shortly after it became available, which provides evidence for the usefulness and potential value of the library.**

I. Introduction

NDN [1]–[3] is a recent proposal for a future Internet architecture design. NDN treats data as first-class entities in network communication and replaces IP addresses with data names at the narrow waist of the 'hourglass' Internet architecture. NDN defines two packet types, *Interest* and *Data*, that carry data requests and replies, respectively. Since NDN directly names data rather than hosts on the network, applications issue an Interest for a specific data name or prefix, which is consumed by the returned data segment (also called a *Content Object*) that has a longest match with the requested name. In order to achieve flow balance, NDN mandates that one Interest can only retrieve one Content Object and each Content Object has a unique data name. If the requested name prefix matches multiple data segments, NDN uses the *Selector* fields in the Interest packet to determine which segment should be returned. Additionally, every Content Object contains a signature field that enables data consumers to verify the authenticity of the received piece.

Up to now researchers experiment with NDN by using the CCNx user-space package[1] that includes a router daemon ('ccnd'), command-line tools, and a set of client libraries for application development. In order to run NDN applications, users must first install the CCNx package and start 'ccnd' as the background service. This requirement has become an obstacle to large scale NDN experimentation for two reasons. First, the installation and configuration of 'ccnd' requires the understanding of the protocol design and significant skill in low-level system management. Second, the CCNx package is not well supported on some platforms, such as Microsoft Windows and iOS. Therefore, it is desirable to have a more 'lightweight' solution for bringing NDN services to average Internet users.

From a developer's perspective, the functionality of the current CCNx libraries is also rather limited. CCNx provides APIs in C and Java languages. The C API is optimized for performance but requires developers to work directly with low-level protocol details (e.g., the CCNx binary XML packet encoding format [4]). It also requires one to program explicit memory management for NDN packets, which can be complex and error-prone. On the other hand, the Java API hides the protocol features completely and only exposes data-stream oriented interfaces. As a result, applications using Java API have little control over the network communications (such as disabling the retransmission of Interest packets). Moreover, the stream-based operations are sometimes unsuitable for certain types of NDN applications.

To address the above issues, Derek Kulinski and Jeff Burke from UCLA developed a Python binding, called 'PyCCN' [5], on top of the CCNx C API. This Python API captures the key elements of the NDN protocol in a straightforward way and enables developers to create portable programs conveniently without touching the packet composition operations unless necessary. PyCCN obtains several important features as a client library, such as portability, developer-friendliness and code simplicity. However, it still binds to the C API and requires installation of the CCNx package for both application developers and users.

In this paper, we present NDN-JS, an NDN client library implemented in pure JavaScript. NDN-JS benefits from the experience of building and using the existing APIs, and it facilitates the spread of NDN usage by reducing the complexity of protocol deployment. It implements the NDN stack completely from scratch without dependency on the CCNx library or 'ccnd' service on the local host. It retains the programming interface design of PyCCN that simplifies the application development process. (Note that NDN-JS does not provide packet forwarding; at this time it relies on a remote 'ccnd'-style daemon to provide the forwarding function.)

The selection of JavaScript is based on the following considerations. First, JavaScript itself is a powerful language that provides inherent support for object-oriented and closure-based asynchronous programming styles, both of which are critical to NDN application development. Moreover, JavaScript runs directly in Web browsers, which enables NDN-JS applications to be easily deployed on many network-capable devices. Finally, an easy-to-program language like JavaScript can encourage

[1]The CCNx package is a prototype implementation of NDN, available at http://www.ccnx.org/.

development and experimentation with NDN protocol, and introduce NDN to a larger audience of developers.

After its release to the public at the end of 2012, NDN-JS quickly attracted wide interest from both inside and outside the NDN project team. Several Web applications have been created using the library, which provide valuable insight into the evolution of the NDN-based Web architecture and the design of the library itself. By the time of this writing, NDN-JS has been widely adopted in building lightweight NDN applications, creating Web interfaces for distributed NDN services and fast prototyping in NDN research projects. We believe NDN-JS will also play an important role in promoting NDN use outside of our project team and sharing NDN applications with existing Internet users.

The remainder of this paper is organized as follows: Section II describes the design and implementation of the NDN-JS library; Section III analyzes the performance of the library through a set of file fetching tests and proposes two algorithms that can boost the throughput; Section IV and V introduce new applications created with NDN-JS; Section VI discusses the current challenges and open questions in developing the library; finally, Section VII concludes the paper and discusses future work.

II. System Design

A. Design Goals

NDN-JS was designed with the following goals in mind:

1) Pure JavaScript Implementation: The library should be implemented in pure JavaScript, functioning without any dependency on the native code, including Java Applet modules or Flash plug-ins. The library should target compatibility with popular Web browsers.

2) CCNx Compatibility: The NDN packets generated by NDN-JS should be wire-format compatible with the CCNx library [3], allowing Web users to connect to existing NDN nodes. (Typically, an NDN-JS node will connect to some NDN router and use that router as the default gateway to access the NDN testbed.)

3) Easy-to-use API: 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 (such as packet format and encoding, etc.) while still reflecting the general model of NDN communications.

B. Architecture

Figure 1 shows the class stack of the NDN-JS library. The important entities in NDN, such as Interests, Content Objects and Names, are abstracted as JavaScript objects. Security libraries provide data signing and verification functionality required by the architecture[2]. The library also implements a set of encoding/decoding helpers that are responsible for converting protocol entity objects to/from 'ccnb' [4] formatted byte streams.

[2]The RSA cryptography utilities used by NDN-JS are also written in pure JavaScript.

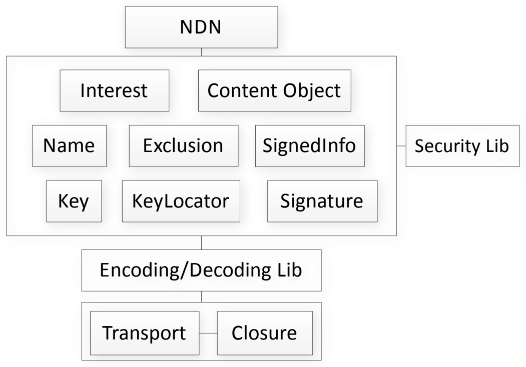


Fig. 1. NDN-JS library architecture

TABLE I. Tested browser support of NDN-JS

Browser	Version	Test Platform
Chrome	23.0	Windows / Mac OS X
Firefox	17.0.1	Windows / Mac OS X
Safari	6.0.2	Mac OS X
Internet Explorer	10.0	Windows
Firefox Mobile	17.0	Android
Safari Mobile	6.0.1	iOS

The NDN class (at the top of the diagram) exposes abstract interfaces for common communication operations such as issuing Interests to the network or registering prefixes in order to publish Content Objects. This class provides an object-oriented NDN protocol handle, akin to a file descriptor in traditional socket programming.

The transport class at the bottom of the stack is responsible for sending and receiving NDN packets on the network. NDN-JS employs a closure-based asynchronous communication style, which is familier to Web developers. The asynchronous Web communication was first introduced by the *XMLHttpRequest* [6] interface (better known as 'AJAX'), where the HTTP request is issued with a registered callback function. This function is invoked asynchronously upon the reception of incoming replies. NDN-JS uses the new WebSocket [7] protocol, which follows the AJAX communication paradigm but enables Web applications to set up bi-directional TCP communications with remote hosts.

Table I shows a list of browsers that support the NDN-JS library, including browsers running on smartphones[3]. The version number shown in the list reflects the version we used in our tests, which is not necessarily the supported minimum version.

In the next two subsections, we discuss the WebSocket transport and the NDN-JS API in detail.

C. WebSocket Transport Service

Traditional HTTP-based Web communication interfaces, such as synchronous GET/POST and asynchronous XML-HttpRequest, lack 'server push' capability. That is, an HTTP

[3]Apple currently does not allow third-party JavaScript engines to run on iOS. All the mobile browsers on iOS use the same Safari engine and thus support NDN-JS.

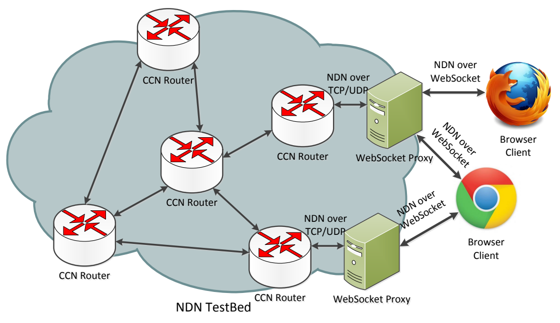


Fig. 2. WebSocket proxies enable browsers to connect to the NDN testbed.

server cannot send data to the client without being requested by the client first, nor can it send a request to the client. The WebSocket protocol provides a JavaScript interface to enable non-HTTP and full-duplex TCP connections from Web browsers to any remote host. As part of the next generation Web technologies loosely called HTML5, WebSocket support has been implemented in many popular Web browsers. Use of this transport enables NDN-JS to meet the design goal of a 'pure JavaScript' solution.

NDN-JS separates the transport implementation from the rest of the API, enabling developers to extend the library with their own transport libraries. One recent example has been support for raw TCP and UDP in the browser add-on, as described in Section V.

One drawback to the WebSocket approach is that the current implementation of CCN routers (running 'ccnd' daemon) does not support the WebSocket protocol. As a workaround, we developed a simple proxy that accepts WebSocket connections from NDN-JS instances and passes the NDN packets over TCP/UDP to CCN routers.

Figure 2 illustrates the relationship between the three entities: client Web browser, WebSocket proxy and CCN routers on the NDN testbed. A client browser runs NDN-JS and establishes WebSocket connection to a proxy. Upon receiving a WebSocket frame, the proxy extracts the original NDN packet and forwards the packet to the pre-configured CCN router via TCP or UDP. In this case, the router simply functions as the default gateway for the client browser. When an NDN packet is received from the router, the proxy encapsulates the packet into WebSocket frames and forwards to the browser.

The WebSocket proxy is implemented in straightforward JavaScript and runs on Node.js [8], a widely used JavaScript execution platform. The proxy listens on TCP port 9696, one greater than the CCN daemon (9695). This enables operators to easily deploy the WebSocket proxy on the same host as a CCN daemon. In the future, we plan to rewrite the proxy using native C code and/or integrate it into 'ccnd', in order to achieve better performance and eliminate the dependencies on Node.js and its modules.

D. Application Programming Interface

The design of the application programming interface in NDN-JS follows the lessons we learned from the design of PyCCN. The top-level NDN class in NDN-JS exports two important methods, *expressInterest()* and *registerPrefix()*, for data fetching and publishing, respectively. *expressInterest()* will compose an Interest packet based on the information provided by the caller and send this packet to the remote CCN router to which NDN-JS connects. *registerPrefix()* will register an NDN Name prefix by sending a special type of Interest packet to the remote CCN router and then wait for incoming Interests that request data under that prefix. Both APIs incorporate an event-driven asynchronous programming paradigm and require the caller to provide closures for event handling.

The following two pieces of sample code demonstrate basic the use of these two APIs for fetching and publishing NDN data.

```
var onData = function (interest, co) {
  // 'Data' event handler
  console.log(co.to_xml());
};

var onTimeout = function (interest) {
  // 'Timeout' event handler
  ndn.close();
};

var ndn = new NDN();   // NDN protocol wrapper

ndn.onopen = function () {
  var n = new Name('/ndn/ucla.edu/foobar');
  ndn.expressInterest(n, null, onData, onTimeout);
};

ndn.connect();
```

Listing 1. Data retrieval example.

```
var onInterest = function (inst) {
  // 'Interest' event handler
  var co = new ContentObject(inst.name,
    'Hello, NDN-JS.');
  co.sign(ndn.getDefaultKey());
  ndn.send(co);
};

var ndn = new NDN();   // NDN protocol wrapper

ndn.onopen = function () {
  var n = new Name('/ndn/ucla.edu/foobar');
  ndn.registerPrefix(n, onInterest);
};

ndn.connect();
```

Listing 2. Data publishing example.

NDN-JS exports a set of helper functions to manipulate the protocol entities, including converting NDN Names from 'ccnb'-formatted byte array to URI-formatted string representation and vice versa, and adding a component at the end of an existing Name object. The encoding/decoding libraries provide interfaces for processing 'ccnb'-encoded NDN packets in wire format, which enables developer to create applications that need to touch the low-level protocol details in JavaScript.

III. Evaluation

In this section, we benchmark the performance of NDN-JS library by measuring the throughput of content retrieval for

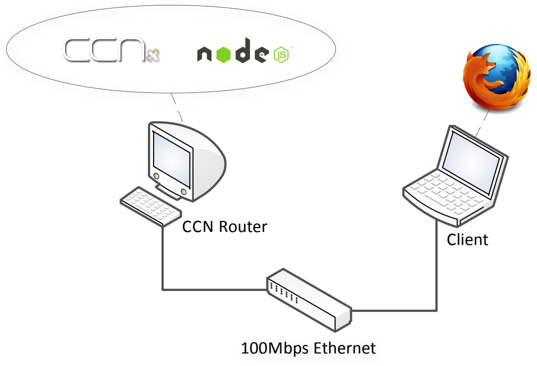


Fig. 3. Throughput test environment.

different web browsers. We also compare the throughput with that of the CCNx command-line utility implemented with the C library. Additionally, we explore the performance impact of the signature verification and propose several ways to speed up that process in the context of fetching large files segmented into multiple Content Objects.

A. *Methodology*

We conducted all the experiments in an isolated network environment with two test machines connected to each other via a 100 Mpbs Ethernet link as shown in Figure 3. The client machine executes NDN-JS test scripts in Web browsers, which retrieve a file from the server machine via NDN. The server hosts three different processes for the test purpose:

- *'ccnd' daemon*, which forwards NDN packets to the proper destination processes;
- *WebSocket proxy*, which directs NDN packets between the NDN-JS client and the 'ccnd' daemon;
- *'ccnr' daemon*, which stores the Content Objects that will be fetched by the NDN-JS client.

During the test, the client retrieved a JPEG image of 917 KB. The source file was pre-published into 'ccnr' using 'ccnputfile' utility, which automatically cuts the input file into 224 chunks (4096 bytes per chunk by default) with consecutive segment numbers. The segment number is appended to the end of the NDN name under which the file is published to give each each chunk a unique name under the common prefix. The publishing utility also signs each data segment and includes the RSA signature the Content Object.

We ran the same test scripts on three popular Web browsers available on Mac OS X: Firefox, Chrome and Safari. Each test was repeated twenty times and the average result used. Using NDN-JS, we implemented a special closure that automatically fetched all the chunks of the file sequentially. To compute the throughput, we recorded the start and stop time of the entire fetching process in JavaScript. To test the C library, we used the 'ccncatchunks2' command in CCNx to fetch the segmented file and then read the transmission time from the log information.

TABLE II. BASIC ALGORITHM THROUGHPUT (UNIT: MBPS)

Verification	NDN-JS (WebSocket)			CCNx C utility
	Chrome	Firefox	Safari	(Raw TCP)
Off	16.01	14.80	17.43	21.68
On	7.202	2.097	2.895	21.19

B. *Analysis*

1) Basic Fetching Algorithm: The first row of Table II shows throughput with signature verification turned off for both NDN-JS and the C utility. The test script used a naive algorithm that issues one Interest at a time and does not issue the next Interest until the Content Object for the previous request is returned (i.e. no pipelining). To be consistent, we also disabled the pipelining option in the C utility.

The performance of NDN-JS varies slightly across the browsers due to their different JavaScript engines. Interestingly, in this case, the performance of NDN-JS and the C utility are not far apart from each other. Both are much lower than the line-speed of the underlying 100Mbps network. Note that the direct point-to-point TCP throughput between the two test machines can easily achieve more than 90 Mbps (measured by the 'iperf' utility) because of the built-in pipelining feature of TCP.

2) Signature Verification Performance: Signature verification is a mandatory operation in NDN protocol and is critical to the NDN security model. To meet our design goals, NDN-JS implements the signature verification process with an open-source third-party RSA library written in pure JavaScript [9]. Since JavaScript is not optimized for big-integer computations (which is one of the basic operations in RSA cryptography), signature verification becomes a bottleneck in data packet processing.

To measure the performance impact of cryptographic computations in NDN-JS, we repeat the same throughput test with signature verification enabled. The experimental result is shown in the second row of Table II. We can see that the signature verification operations greatly reduce the performance of content fetching of NDN-JS. Chrome achieves about 45% of the throughput in the non-verification case, which is already the best performance among the three browsers. In clear contrast, the performance of the C utility is only slightly affected by the signature processing.

Currently there is no built-in cryptography API in JavaScript that performs signature verification in Web browser kernel using efficient, native code. In order to improve the performance, another method must be used to speed-up the fetching process. The application level transmission delay in NDN-JS is comprised of two parts: the network transmission delay and the JavaScript processing delay. In the next two subsections, we consider two possible optimization techniques that reduce these two types of delay, and we analyze their effects with experiment results.

3) Optimization Method 1: Pipelining: The throughputs of both NDN-JS and the C utility are significantly lower than line-speed (100 Mbps). One reason for such low performance is that the previous tests used a naive stop-and-wait fetching algorithm. To improve fetching efficiency, we can borrow the pipelining technique from TCP and allow the client to issue

TABLE III. PIPELINING ALGORITHM THROUGHPUT (UNIT: MBPS)

Verification	NDN-JS (WebSocket)			CCNx C utility
	Chrome	Firefox	Safari	(Raw TCP)
Off	51.33	61.60	67.77	72.84
On	10.85	2.352	3.187	72.50

several requests in a row and keep the underlying network 'pipeline' full of packets.

We implemented a pipelined fetching algorithm similar to the TCP 'slow start' algorithm. In this case, NDN-JS maintains a sliding window of outstanding Interests. The window size starts from 1 and is bounded by a maximum value. Every time a segment is returned within the Interest lifetime, the window size is increased by 1 to allow more Interests to be sent. When the retransmission timer expires, the window size is shrunk back to 1. The C utility 'ccncatchunks2' we used in the test also implements a similar algorithm, which can be enabled with a '-p' command option. The maximum pipeline size is set to 32 for both NDN-JS scripts and the C utility in our test.

Table III shows the throughput of the pipelining algorithm. We can see that, for the non-verification case, both NDN-JS and the C utility achieve significant increase in throughput, and the NDN-JS throughput still closely follows that of the C utility. However, when verification is enabled, pipelining did not improve NDN-JS throughput much, leading to the conclusion that JavaScript processing delay led to the low throughput in the previous test.

4) Optimization Method 2: Delayed Verification: With the understanding that the performance bottleneck comes from JavaScript processing, we used the JavaScript profiler integrated in three Web browsers to analyze the execution time of the test scripts. The profiling result indicates that a large portion of the running time is spent on functions that perform RSA signature and SHA hash computations. This inspires us to consider the possibility of reducing the number of times that the cryptography library is invoked in order to improve throughput in this case.

In the current CCNx implementation of the 'ccnputfile' utility, the signature for different segments of the same file is generated via a Merkle hash tree [10] when the file is fragmented and published into the repo. A Merkle hash tree is constructed by placing n Content Objects at the leaf nodes of a binary tree with a depth of $\lceil log(n) \rceil + 1$. Each internal or leaf node has a hash value; the hash of a leaf node is simply the hash of the data bytes stored in that node, while the hash of a internal node is the hash on the concatenation of the hash values from its two children. The Merkle hash tree ensures that every internal node has two children. Otherwise the single child is 'pushed' up and merged with its parent. Figure 4 shows a simple Merkle hash tree with seven leaf nodes.

When signing multiple segments at the same time using a Merkle hash tree, the signature is computed over the hash of the root node of the tree and shared by all the segments. Due to the consideration of flexibility, the protocol used by CCNx was designed to be able to verify each segment independently, by including a 'Witness' component in the data packet that encodes the position of the leaf in the hash tree as well as

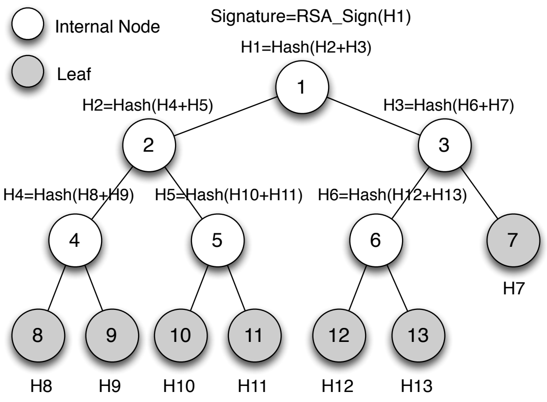


Fig. 4. Merkle hash tree with 7 leaf nodes

the hash values of all the sibling nodes along the path to the root. Upon receiving a Data packet, one can walk up along the path to compute the hash of the root node using the hash of the received data and the sibling node information, and then verifies the signature of the root hash. The default verification algorithm in the NDN-JS library is implemented in this way.

Consider a full Merkle hash tree with n leaf nodes: the Merkle hash tree signing scheme was originally proposed to reduce the signing cost as the publisher only needs to compute $2n$ hashes and 1 signatures for the entire hash tree. However signature verification per leaf node will invoke $log(n)$ calls to the SHA hashing function and 1 call to the RSA signature computation. Thus to verify each of the n leaf nodes on the hash tree, the library needs to compute total $nlog(n)$ SHA hashes and n RSA signatures. In the case of multi-segment file fetching, given the consumer eventually fetches all the segments, it can elect to delay the signature verification until it have collected all the segments and then construct the entire hash tree only once. In that case, the complexity of computing $O(nlog(n))$ hashes and $O(n)$ signatures can be reduced to just $O(n)$ hashes and $O(1)$ signatures.

To evaluate the performance of a delayed verification algorithm based on this observation, we disabled the default signature verification process of NDN-JS and implemented the new approach for both the basic and the pipelining fetching algorithm. The throughput results of the new tests are shown in Table IV[4]. We can see that delayed verification strategy can indeed improve the performance of NDN-JS for multi-segment files. For example, the file fetching throughput in Firefox reaches about 32 Mbps with the pipelining turned on, which is a 10-fold improvement from the previous result. Also note that since the JavaScript execution bottleneck is alleviated, ttransport pipelining shows more significant impact; the throughput at least doubles across all browser platforms compared to the non-pipelining case. Keep in mind, however, that this delayed verification scheme may place limitations on progressive display or delay of file usage, given that early segments are not verified immediately upon arrival.

[4]We did not implement the delayed verification for the C utility since the cryptographic computations in C code do not cause much performance penalty.

TABLE IV. DELAYED VERIFICATION ALGORITHM THROUGHPUT (UNIT: MBPS)

Pipeline	NDN-JS (WebSocket)		
	Chrome	Firefox	Safari
Off	8.476	12.88	11.00
On	15.38	32.08	24.79

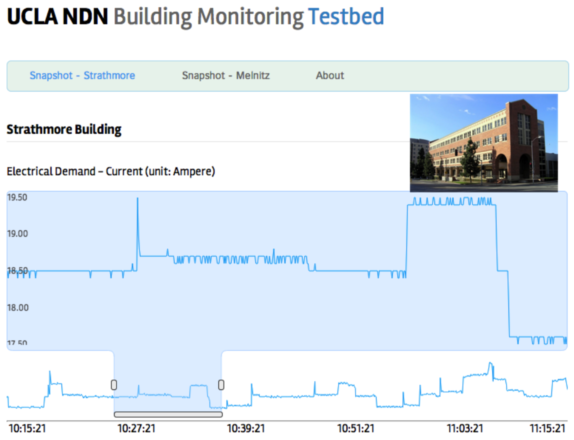


Fig. 5. Building monitoring platform page.

IV. APPLICATION I: WEB INTERFACES

Since its release the NDN-JS library has been used in several projects to build Web applications with NDN. These projects explore the capability and potential benefits from NDN-based Web services, and provide valuable feedback on the library design and implementation. In this section, we introduce a few representative projects powered by the NDN-JS library, which are either publicly available on the Internet or integrated with local applications.

A. Building Monitoring Platform Website

The NDN building monitoring platform is being developed by Wentao Shang from UCLA as part of a larger project exploring the design of a NDN-based Building Automation System. In this platform, background processes collect sensor data indicating, for example, electrical demand, and it as Content Objects into a repository. When users load the front-end web page in their browsers, NDN-JS is used to fetch the data and render it using JavaScript visualization libraries.

The building monitoring platform tests NDN Interest exclusion filter support in NDN-JS, which enables the client to fetch only the Content Objects under a prefix whose child name component falls in a specific range. The platform encodes the data acquisition timestamp as the last component in the NDN name. The NDN-JS client then specifies in the Exclude filter the time range it is interested in and issues Interests to retrieve the data in that range directly.

Figure 5 shows a snapshot of the platform running inside a browser.

B. NDN Lighting Control Web Interface

Alex Horn and Wentao Shang from UCLA developed the Web interface for the NDN lighting control system [11],

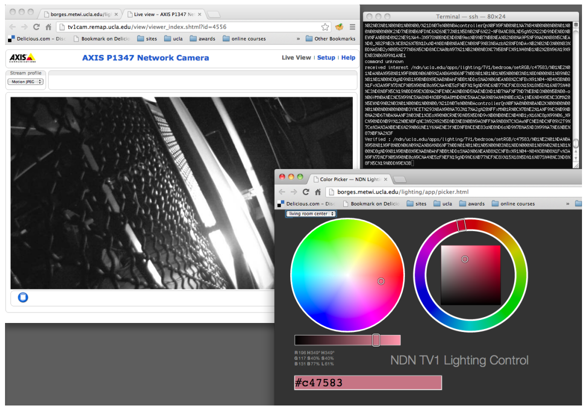


Fig. 6. Lighting control Web interface.

another pilot component of the NDN Building Automation System project. Lighting controllers receive Interest packets that serve as authenticated commands. The control commands are expressed as NDN names that include a signature as the last name component, which serves as the access control authenticator. The operators sign their commands and then send the authenticated Interest packets to the lighting controllers, which verify the signature and carry out operations (change lighting levels) accordingly.

The lighting control system was originally implemented by Alex Horn using PyCCN. A Web control interface was implemented via a Python-based CGI on the Apache server. Wentao Shang ported the Interest authentication approach to JavaScript and used it, together with NDN-JS, to implement a new version of the control interface which issues commands directly from Web browsers to the light controllers, without the server indirection. The prototype system is deployed in UCLA TV#1 studio in Melnitz Hall. Figure 6 shows a screen capture of both the control and monitoring interface[5].

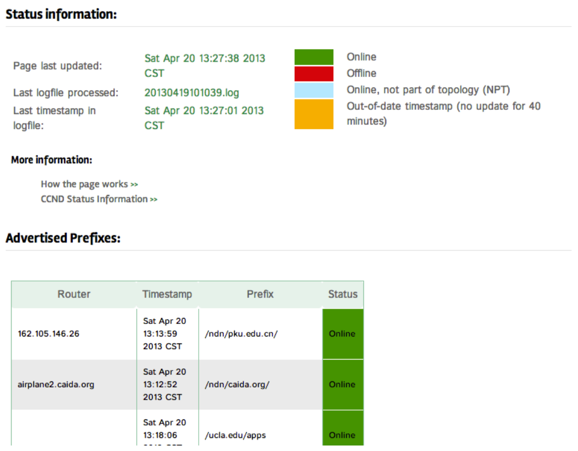
C. NDN Testbed Router Status Website

Adam Alyyan from the University of Memphis used the NDN-JS library to build a monitoring website that reports the status of NDN routers on the testbed. This application uses the ‘ccnping’ protocol [12], which is ported to JavaScript by Jeff Burke from UCLA. ‘ccnping’ is similar to the traditional ‘ping’ utility that is based on ICMP Echo/Echo Reply messages. The ‘ccnping’ client issues an Interest with a special name containing a random number as the last component. The ‘ccnping’ server will simply reply with a Content Object that includes the same random number as the data content. The NDN router status website implements the ‘ccnping’ protocol with NDN-JS and uses it to detect NDN router failures or reachability problems. Figure 7 shows a snapshot of the router status page in a browser. This monitoring website is publicly available at http://netlab.cs.memphis.edu/script/htm/status.htm.

D. ChronoShare File Change History Page

Alexander Afanasyev and Zhenkai Zhu from UCLA incorporated an NDN-JS powered Web service in their recently-developed distributed application called ‘ChronoShare’ [13],

[5]Due to privacy consideration, the sensing and control pages described above are not publicly available.



Status information:

Page last updated:	Sat Apr 20 13:27:38 2013 CST	🟩	Online
		🟥	Offline
Last logfile processed:	20130419101039.log	🟦	Online, not part of topology (NPT)
Last timestamp in logfile:	Sat Apr 20 13:27:01 2013 CST	🟧	Out-of-date timestamp (no update for 40 minutes)

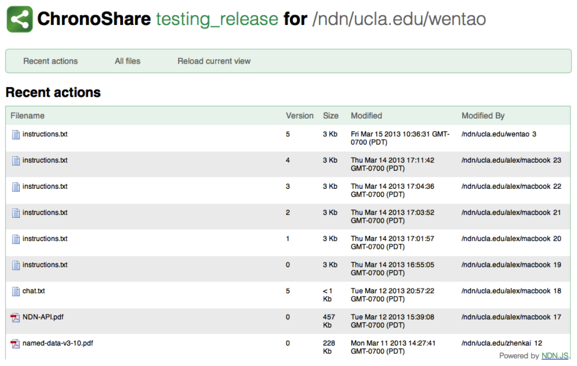
More information:

How the page works »
CCND Status Information »

Advertised Prefixes:

Router	Timestamp	Prefix	Status
162.105.146.26	Sat Apr 20 13:13:59 2013 CST	/ndn/pku.edu.cn/	Online
airplane2.caida.org	Sat Apr 20 13:12:52 2013 CST	/ndn/caida.org/	Online
	Sat Apr 20 13:18:06	/ucla.edu/apps	Online

Fig. 7. NDN testbed router status page.



ChronoShare testing_release **for** /ndn/ucla.edu/wentao

Recent actions All files Reload current view

Recent actions

Filename	Version	Size	Modified	Modified By
instructions.txt	5	3 Kb	Fri Mar 15 2013 10:36:31 GMT-0700 (PDT)	/ndn/ucla.edu/wentao 3
instructions.txt	4	3 Kb	Thu Mar 14 2013 17:11:42 GMT-0700 (PDT)	/ndn/ucla.edu/alex/macbook 23
instructions.txt	3	3 Kb	Thu Mar 14 2013 17:04:36 GMT-0700 (PDT)	/ndn/ucla.edu/alex/macbook 22
instructions.txt	2	3 Kb	Thu Mar 14 2013 17:03:52 GMT-0700 (PDT)	/ndn/ucla.edu/alex/macbook 21
instructions.txt	1	3 Kb	Thu Mar 14 2013 17:01:57 GMT-0700 (PDT)	/ndn/ucla.edu/alex/macbook 20
instructions.txt	0	3 Kb	Thu Mar 14 2013 16:55:05 GMT-0700 (PDT)	/ndn/ucla.edu/alex/macbook 19
chat.txt	5	< 1 Kb	Tue Mar 12 2013 20:57:22 GMT-0700 (PDT)	/ndn/ucla.edu/alex/macbook 18
NDN-API.pdf	0	457 Kb	Tue Mar 12 2013 15:39:08 GMT-0700 (PDT)	/ndn/ucla.edu/alex/macbook 17
named-data-v3-10.pdf	0	228 Kb	Mon Mar 11 2013 14:27:41 GMT-0700 (PDT)	/ndn/ucla.edu/zhenkai 12

Powered by NDN.JS.

Fig. 8. ChronoShare history page

which provides 'Dropbox'-like functionalities to synchronize file folders on different devices using NDN protocol. ChronoShare provides users a Web page that can display the history of file changes in the shared folders. To view the history in a Web browser, users just need to click a button on the application's menu bar, which will automatically load the web page in the default Web browser. Figure 8 shows a snapshot of the ChronoShare file change history page.

E. *CCNx Federated Wiki*

Ryan Bennett et al. ported an open source wiki implementation called 'Smallest Federated Wiki' to run over NDN. It enables authors to publish wiki content as NDN Content Objects over the NDN network in a fully distributed, peer-to-peer fashion, while enjoying the benefits of intrinsic security support and in-network data caching. User-generated content is stored inside the client browser using HTML5's 'IndexedDB' API [14]. The client code calls the '*registerPrefix*' API in NDN-JS to register the publisher's prefix to a remote NDN router. The Interests for data falling under that prefix will be forwarded by the NDN router to the client browser, which then replies with the matching content.

CCNx Federated Wiki shows a typical example of using Web browsers as NDN Content repositories. It also demonstrates how the browser-based local storage can be used to manage user-generated or fetched NDN data, using the latest HTML5 technologies, and motivates our interest in browser-based forwarding discussed below. Currently this project is still under development; the prototype source code is publicly available [15].

V. Application II: Firefox NDN Protocol and Toolbar Add-on

A. *Basic Design*

Jeff Thompson employed NDN-JS to create a Firefox NDN add-on in JavaScript that implements an 'ndn:' URI scheme, which can be entered in the browser location bar or used in HTML anchor tags. Its goals are to exercise the library and to provide a familiar browser interface for experimentation with NDN. Taking advantage of the modularity of NDN-JS, the add-on implements a new transport service using the Firefox XPCOM interface [16], enabling direct connection to NDN routers via raw TCP or UDP sockets without the need for WebSockets proxy.

For this add-on, an Interest is converted into a URI using the following naming conventions.

- The Name field of the Interest (including content version and segment number) is encoded in the URI according to the CCNx URI scheme. For example, *ndn:/ucla.edu/contact.html/%00%01* refers to the second segment of */ucla.edu/contact.html*.
- Interest selector fields, such as the ChildSelector, AnswerOriginKind, etc. are appended to the URI in the form of *?ndn.SelectorField=value*. For example, *ndn:/ucla.edu/maps.html?ndn.ChildSelector=1* selects the rightmost child of the corresponding content. This exposes significant features of NDN.

When processing a URI beginning with 'ndn:', Firefox automatically calls the add-on to retrieve the content. The add-on converts the URI to a Name and requests the content via NDN. If the content is fragmented (i.e., the name of the first packet returned contains a segment number) while no segment number is specified in the original Interest name, all the segments of that content will be fetched sequentially until the last segment is met. This approach works directly with files stored in the CCN Repo using its standard naming conventions.

B. *Additional features of the Firefox Add-On*

In addition to enabling content fetching via NDN and its benefits of multicast delivery and in-network caching, the add-on also helps us explore how to provide various application-level possibilities to content publishers and consumers via NDN. A few such features have been implemented in JavaScript using NDN-JS as described below.

1) Long-term Secure Links: NDN support in the browser may provide the option for consumers to verify that a piece of named (static) content retrieved a long time after its creation is indeed the content originally linked with a URI. This can be achieved by including a content digest in the URI. Our implementation supports this by enabling an application to create names with a ContentDigest after the version, using the "guid" special marker "%C1.M.G" [17]. E.g. a user may express an interest for a license file *ndn:/example.com/license.html* which matches:

ndn:/example.com/license.html/%FD%05%0BZ%94%B4l/
%C1.M.G%C1<binary-XML-encoded ContentDigest>

In this case, the add-on detects the special name component, computes the digest of the received file, compares this to the ContentDigest in the name, and shows an error if they do not match.

This could be used not only by web-based applications but also by the browser itself. Say, for example, the user views some content from the network and bookmarks the URI, the browser can at that time append a ContentDigest to the URI. Much later, the user can view the same content again by clicking the bookmark to retrieve the same file, perhaps from a caching repository. The digest can be verified against the one in the bookmark, and the user will know if she is viewing the exact same content, even if the signer's private key has been compromised in the meantime. (This, of course, assumes that the hash algorithm used to generate the digest is secure at the time of retrieval.)

2) Get Latest Version: In NDN, application-level protocols often require more than one Interest/Data exchange to retrieve the desired data. Retrieving the latest version of named content is an example case: it is typically implemented by iterative requests for named data using the ChildSelector and Exclusion fields in the Interests to get the most recent version of the content[6]. The add-on implements this design pattern in JavaScript and exposes it to the user of the browser. If a data name (URI) uses the CCNx versioning strategy, the user can click `Get Latest` in the NDN toolbar to request the latest, which issues the appropriate requests, and still verifies the overall ContentDigest if used. Such common routines may likely be backported to the NDN-JS library.

3) Representing NDN Semantics to the End User: Through the add-on, we plan to explore how best to convey NDN semantics and common patterns to browser users and test the direct application of NDN as an HTTP alternative. For example, if the user puts a prefix in the address bar that is matched with a longer name, the browser updates the address bar with the full name (without segment number) after retrieval. This is particularly important to show the version number of content retrieved, whenever applicable.

VI. Discussion

NDN-JS is a young project and continues to evolve with both the NDN architecture and latest Web technologies. During the development of this project, we experienced a number of challenges and open questions, which we discuss briefly here.

A. Local storage

Limitations in JavaScript enforced by the security model of modern Web browsers are faced by any web application. The important role of storage in NDN makes local file system accessibility a key example: Current Web browsers prohibit JavaScript from accessing local disk storage, which forces NDN-JS to store all the fetched or self-generated content objects in memory, either as raw data or organized into structural storage (such as the IndexedDB used in the CCNx Wiki project). The HTML5 standard provides a File API [19] for Web applications to access a sand-boxed file system, which is, however, not yet widely supported. The lack of permanent storage capability makes it difficult to implement what we expect to be standard NDN functionality, such as writing fetched data into local disk or publishing local files as Content Objects. This forces otherwise distributed applications like CCNx Wiki to reply on remote repositories.

The lack of local file accessibility also creates an issue with security key configuration. In other NDN client APIs such as PyCCN, it is convenient and easy to load a user-provided '.pem' key file and use it for data signing. In NDN-JS, however, a user cannot access a key file from a local disk. The temporary solution used by NDN-JS now is to hardcode the default RSA key bytes in the library. We plan to implement more elegant key configuration methods like fetching key data via secured links from Web servers or retrieving the key data as an NDN Content Object from the NDN repository. However, those methods are less flexible and require a trust relationship between the end-user and key management server. Currently, we are still investigating other possible solutions to the Web-based key configuration issue.

B. Cryptographic performance

A significant performance challenge we encountered earlier is the low efficiency of current JavaScript engines for cryptographic computations. We have already seen in Section III that the signature verification process is the main burden on the throughput. Since JavaScript is not designed to support big-integer operations, it is generally hard to optimize cryptography operations in pure JavaScript implementations. Recently, there have been discussions about the possibility of extending the JavaScript native API with cryptographic features and its implication on the Web security model. For example, W3C is already working on a draft proposal for a Web Cryptography API [20]. If such security-related APIs were added to JavaScript, we would be able to offload the computation-intensive tasks into browser kernels and boost the performance of NDN-JS, likely catching up with the C implementations.

C. Forwarding support

There has been debate inside the NDN-JS project team about whether to integrate the full 'ccnd' functionality into the library. In particular, like other client libraries, NDN-JS does not yet implement the forwarding functionality of the NDN architecture. As a result, NDN-JS clients must connect to a pre-configured NDN hub and Websocket proxy in order to access an NDN network. This is equivalent to setting a *default route* for the browser-based NDN nodes. In the future, we may want clients to have a more robust forwarding strategy (e.g., favoring shorter paths in local communication instead of going through a remote hub at all times). We are still exploring the implications of providing this functionality, which would effectively turn Web browsers into NDN routers.

D. Application conventions

An application-related design challenge is the integration of NDN protocol into Web services. Both NDN and the current

[6]For an example, see the discussion of live video streaming in [18].

Web architecture share the same communication paradigm driven by data request and retrieval. Therefore it is quite easy to convert the current Ajax-style Web services into NDN-based applications. However, since NDN Interests support more expressive features directly, such as Interest selectors, there is still an open question about how to best employ these additional features to build more powerful Web applications.

Additionally, how to render NDN ContentObjects in Web browsers is another issue to be addressed. Current Web standards extend HTTP with MIME to express content type, which is likely to be adapted by NDN-JS to specify content type. Though this is application-specific, we still need some convention to guide the Web development with NDN-JS.

E. Data-centric security

Another design challenge in NDN-JS is to construct a new Web security model using NDN's inherent security features. Traditional Web security relies heavily on secured transport support (e.g. HTTPS). While the current solutions attempt to secure the underlying communication channel, the NDN architecture secures data directly [21], which can provide a more flexible and scalable security approach. The data-centric nature of NDN eliminates the need for traditional origin-based Web access control model, including the widely-adopted 'Same-Origin' Policy (SOP) and the negotiation-based Cross-Origin Resource Sharing (CORS) [22][7]. However, new standards that can be easily deployed must be developed. We believe that Web access control policy could be enforced through the NDN convention of encryption-based control scheme (e.g. using authenticated Interest [11]), which provides finer granularity of data access management, although this is still an ongoing research topic.

VII. Conclusion and Future Work

In this report we described NDN-JS, a JavaScript client library created to facilitate the development and deployment of NDN applications for the Web. We analyzed the library's throughput performance with different optimization strategies in the context of file fetching. We also discussed several running Web applications powered by NDN-JS and the challenges faced in further development of the library. The original design goal of creating a lightweight and easy-to-use NDN client library has proven to provide usability and efficiency in application development.

NDN-JS serves as a first exploration of an NDN-based Web architecture and how the data-centric semantics of NDN can provide secure, efficient and scalable Web services. Our future plan for the NDN-JS project includes implementing the key configuration functionality, refactoring APIs based on the experience in using the library and improving the quality of the source code. We expect this to be a continuous effort and welcome contributions from all interested parties.

Apart from improving the library itself, we also plan to make extensive use of the library in future projects during the next phase of NDN research. One of the new projects that we have in mind is to design the cyber-physical system architecture on top of NDN protocol. Both the lighting control application and the building monitoring platform mentioned in this paper are part of the initial exploration of this broad field. We expect NDN-JS to be one of the core technologies in creating user-friendly interfaces for complex industrial systems, such as sensor network monitoring or building automation systems.

[7]These models still bundle the data with its location (or 'origin'), while NDN-JS effectively enables the client to fetch desired data from any reachable NDN nodes.

Acknowledgment

This technical report is an extended version of an earlier publication which was presented at INFOCOMM'13 NOMEN workshop [23]. This report includes both the previous results and the more recent ones since the submission. Sec. III covers new test results from recent experiments and extensive analysis on the performance comparison with different optimization strategies, which is not covered in the NOMEN paper; Sec. IV introduces several new NDN-JS-powered applications, which are developed after the NOMEN paper submission; Sec. VI summarizes our cumulated thoughts about the library development since its release in Dec. 2012.

NDN-JS effort was originally started by A. C. de Verdiere and Meki Cherkaoui from UCLA as a term project for CS217B in spring 2012. They finished the initial implementation of JavaScript classes to abstract the NDN protocol entities and the encoding/decoding operations. Derek Kulinski from UCLA and Junxiao Shi from University of Arizona offered valuable feedback on the library's design and implementation, along with assistance from Michael Plass and Nicholas Briggs of the CCNx team at PARC.

This work was supported by NSF grants CNS-1040868 and CNS-1248049.

References

[1] "Named data networking." [Online]. Available: http://www.named-data.net/

[2] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, ser. CoNEXT '09. New York, NY, USA: ACM, 2009, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/1658939.1658941

[3] "Ccnx." [Online]. Available: http://www.ccnx.org/

[4] "Ccnx binary encoding (ccnb)." [Online]. Available: http://www.ccnx.org/releases/latest/doc/technical/BinaryEncoding.html

[5] "Pyccn." [Online]. Available: https://github.com/remap/PyCCN

[6] J. Aubourg, J. Song, and H. R. M. Steen, "XMLHttpRequest," World Wide Web Consortium, Dec. 2012. [Online]. Available: http://www.w3.org/TR/2012/WD-XMLHttpRequest-20121206/

[7] I. Fette and A. Melnikov, "The WebSocket Protocol," RFC 6455 (Proposed Standard), Internet Engineering Task Force, Dec. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6455.txt

[8] "Node.js." [Online]. Available: http://nodejs.org/

[9] "jsrsasign - rsa signing and verification in javascript." [Online]. Available: http://kjur.github.io/jsrsasign/

[10] "Ccnx signature generation and verification." [Online]. Available: http://www.ccnx.org/releases/latest/doc/technical/SignatureGeneration.html

[11] J. Burke, A. Horn, and A. Marianantoni, "Authenticated Lighting Control Using Named Data Networking," Technical Report, The NDN Project Team, Oct. 2012.

[12] "Ccnping utility." [Online]. Available: https://github.com/NDN-Routing/ccnping

[13] Z. Zhu, A. Afanasyev, and L. Zhang, "ChronoShare: a new perspective on effective collaborations in the future Internet," Poster, UCLA Tech Forum 2013, May 2013. [Online]. Available: http://www.engineer.ucla.edu/techforum/index.html

[14] N. Mehta, J. Sicking, E. Graff, A. Popescu, J. Orlow, and J. Bell, "Indexed Database API," World Wide Web Consortium, May 2013. [Online]. Available: https://dvcs.w3.org/hg/IndexedDB/raw-file/tip/Overview.html

[15] "Ccnx federated wiki." [Online]. Available: https://github.com/WardCunningham/Smallest-Federated-Wiki

[16] "Xpcom transport." [Online]. Available: https://developer.mozilla.org/en-US/docs/XPCOM_Interface_Reference/nsISocketTransportService

[17] "Ccnx technical documentation." [Online]. Available: http://www.ccnx.org/releases/latest/doc/technical

[18] D. Kulinski and J. Burke, "NDN Video: Live and Prerecorded Streaming over NDN," Technical Report, The NDN Project Team, Sep. 2012.

[19] E. Uhrhane, "File API: Directories and System," World Wide Web Consortium, Apr. 2012. [Online]. Available: http://www.w3.org/TR/file-system-api/

[20] D. Dahl and R. Sleevi, "Web Cryptography API," World Wide Web Consortium, Jan. 2013. [Online]. Available: http://www.w3.org/TR/2013/WD-WebCryptoAPI-20130108/

[21] D. Smetters and V. Jacobson, "Securing Network Content," Technical Report, PARC, 2009.

[22] A. van Kesteren, "Cross-Origin Resource Sharing," World Wide Web Consortium, Apr. 2012. [Online]. Available: http://www.w3.org/TR/cors/

[23] W. Shang, J. Thompson, M. Cherkaoui, J. Burke, and L. Zhang, "NDN.JS: A javascript client library for Named Data Networking," in *Proceedings of IEEE INFOCOMM 2013 NOMEN Workshop*, April 2013.