NDN.JS: A JavaScript Client Library for Named Data Networking

Wentao Shang*, Jeff Thompson‡, Meki Cherkaoui*, Jeff Burke† and Lixia Zhang*
*Department of Computer Science, UCLA. {wentao,meki,lixia}@cs.ucla.edu.
‡Los Angeles, California. jeff@thefirst.org.
†Center for Research in Engineering, Media and Performance, UCLA. jburke@ucla.edu.

***Abstract*—NDN.JS is the first JavaScript implementation of a client library for Named Data Networking (NDN). It facilitates NDN experimentation and usage by enabling end nodes to interact with an NDN network without installing the CCNx code package. It is also a first step towards exploring an NDN-based Web architecture. NDN.JS is wire format compatible with CCNx and supports the basic NDN functions of content fetching and publishing using Interest/Data exchange. NDN.JS works with modern Web browsers, including some browsers on mobile devices, that support JavaScript and HTML5 WebSocket. The client communicates with existing CCN routers via a simple Websocket proxy. As a use case study, we create a Firefox Add-On over NDN.JS to enable content fetching using an 'ndn:/' URI scheme and identify several research issues in bringing NDN into existing browsers.**

I. Introduction

Named Data Networking (NDN) [1] [2] is a recently-proposed networking architecture that shifts the "thin waist" of the Internet from IP's host-centric model to a data-centric model, together with two important consequences. First, data are named by applications, and the network routes directly on these data names, rather than host addresses. Second, each name is associated with a cryptographic key, which is used to secure data directly. NDN is one of the major research efforts in the broader area of information-centric networking (ICN).

Web services today provide global data dissemination, yet its implementation is still based on the TCP/IP architecture developed over 30 years ago, creating a disconnect between the host-based addressing of the underlying TCP/IP stack and the data-focused naming schemes of URIs. NDN is a promising solution to address this mismatch in the current Web architecture. However, a major obstacle along the road towards an NDN-based Web is that current Web browsers do not have inherent NDN support. To effectively address this need in as many browsers as possible, we have created a pure JavaScript NDN library.

In this paper, we describe the NDN.JS project, which addresses both the high-level goal of exploring an NDN-based web and a practical need for browser support in NDN research. Pragmatically, it aims to enable more widespread dissemination of the NDN protocol and applications by reducing the complexity of usage for both users and developers. Because it is implemented in JavaScript, it enables researchers to easily build NDN-based applications that can be delivered into one of the most ubiquitous platforms on the Internet – the Web browser.

NDN.JS provides a JavaScript API that can be used to embed NDN data access and publishing at the client-side of existing Web applications. It requires only the addition of WebSockets [3] support in NDN routers, which is currently achieved by a simple JavaScript-based proxy. NDN.JS is a first step in our exploration of how the Web may evolve through the use of data-centric protocols. It does not assume a particular overarching approach to Web applications, but is intended to bridge the Web services of today with NDN application experimentation. One of the first tools built with NDN.JS was a Mozilla Firefox Add-On, which enables Firefox to process URIs in the NDN scheme (e.g. "ndn:/foo/bar/file.html") and brings NDN features "up to the user".[1]

This paper is organized as follows: Section II briefly reviews NDN; Section III introduces the NDN.JS system design; in Section IV we present the performance evaluation of the current NDN.JS release in various Web browsers. Section V introduces the Firefox Add-On. Finally Sections VI and VII discuss remaining challenges and our future plan.

II. NDN Background and Available Libraries

NDN retrieves data based on application-defined names rather than host addresses. To retrieve data, the consumer must know the name, instead of location, of that data. NDN communication involves two packet types: *Interest* and *Data*. An Interest packet is issued by the consumer to express what set of data is needed. A Data packet is returned by the data producer in response to an Interest. Both Interest and Data packets use names to identify the data being exchanged. An Interest is "satisfied" when a Data packet is received with a Content Name that falls within the name prefix specified in the Interest packet. For more detail, see [1].

An existing reference implementation of the NDN protocol is the CCNx package [4] from PARC, which is written in C. It is used by the CCN routers and low-level utilities. Intended for developers and experts, it requires the skills to compile, configure and run from the command line.

The CCNx Java libraries [5] provide useful higher-level abstractions for manipulating segmented files, exploring the CCN Repo and enforcing profiles for versioning and meta-data conventions. However, in our experience, the abstractions

[1]The source code for NDN.JS and the Firefox Add-On is available at http://github.com/remap/ndn-js.

of the Java API have not always been appropriate for our applications research, which often explores specific protocol features. Additionally, running NDN-based Java Applets at the browser-side (one of our goals) requires installation of Java, download of the library, explicit authorization from the Web users, and integration with HTML page delivery – a long series of steps which we believe will limit uptake and experimentation.

The Python API (PyCCN) [6] provides a middle ground: it offers a set of easy-to-use wrappers on top of the high-performance C libraries. It provides object-oriented classes for Interest, ContentObject, etc., and uses callback functions to implement asynchronous communications. However, PyCCN still requires manual installation of the CCNx package and lacks native support in Web browsers, which makes it unsuitable for Web development. We expect to use a combination of PyCCN and the NDN.JS libraries to do rapid prototyping of NDN applications.

III. System Design

A. Design Goals

NDN.JS is created with the following design goals:

- *Pure (and Compatible) JavaScript*: In order to run on the widest number of browsers and machine types, without user intervention, no native code or code that required user authorization (such as Java) should be used.
- *Developer-friendly API*: The API should clearly represent the basic protocol components directly (Name, Interest, ContentObject, and Key), while not burdening everyday development with wire format details.
- *Content Signing and Verification*: The library must support content signing and verification as currently implemented in CCNx using RSA signatures and SHA-256 hashes.
- *CCNx Compatibility*: The library should be wire format compatible with CCNx routing and forwarding, facilitate use in the NDN project testbed, and enable performance comparisons.
- *Lightweight*: Because it is intended to be used in browsers, the library should be as simple and low-overhead as possible, while still implementing key NDN features.

B. Use Cases

We describe below a few basic use cases envisioned for NDN.JS. We are making NDN.JS package publicly available and hope many more to be developed by others.

- *Web content fetching and publishing*: Web users can use NDN.JS to communicate with CCN routers to fetch and/or publish content conveniently using the browser. The Firefox Add-On, described later in this paper, explores the fetching of content using an "ndn:" scheme, which can be integrated into existing web pages.
- *User interfaces to NDN applications*: Browser integration enables easy development of rich, cross-platform user interfaces for NDN applications that currently exist. These may run natively on the same host as the browser, or run elsewhere in the network and communicate with the browser through NDN Interest/Data exchanges.
- *Peer-to-peer chatting and file sharing*: NDN.JS allows Web users to act as both content producer and consumer, which is expected to enable NDN-based P2P communication among Web users.
- *Mobile Web applications*: NDN-based mobile applications can be easily developed in JavaScript and quickly delivered into Web browsers on a wide variety of mobile devices without native application installation or manual configuration.

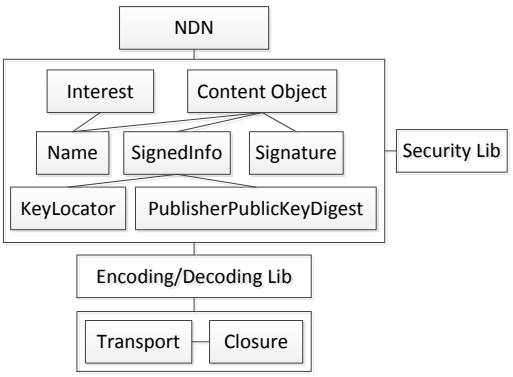

Fig. 1. NDN.JS library architecture

C. Architecture

NDN.JS includes most of the core functionality of the CCNx client libraries, including data fetching and publishing, content signing and verification, exclusion filters, and binary name storage.

The architecture of the NDN.JS library is shown in Fig. 1. The "NDN" class is the top level abstraction, which interacts with other components of the protocol (Interest, ContentObject, etc.). NDN.JS provides a set of security libraries, based on open source implementations of RSA/SHA algorithms, to perform data signing and verification. It also implements encoding and decoding of Interests and Content Objects to and from a CCNx-compatible wire format. Transport service classes lie in the bottom of the stack, which communicate with encoding/decoding libraries to process NDN packets on the wire.

Table I shows a list of browsers that can run NDN.JS successfully. The version number of the browsers we used in the compatibility test is also shown in the list (earlier versions may also work).

D. WebSockets Transport Service

Conventional HTTP-based Web implementations, such as synchronous GET/POST or asynchronous XMLHttpRequest, are not suited for NDN communications due to the lack

TABLE I
TESTED BROWSER SUPPORT OF NDN.JS

Brower	Version	Test Platform
Chrome	23.0	Windows / MacOSX
Firefox	17.0.1	Windows / MacOSX
Safari	6.0.2	MacOSX
Internet Explorer	10.0	Windows
Firefox Mobile	17.0	Android
Safari Mobile	6.0.1	iOS

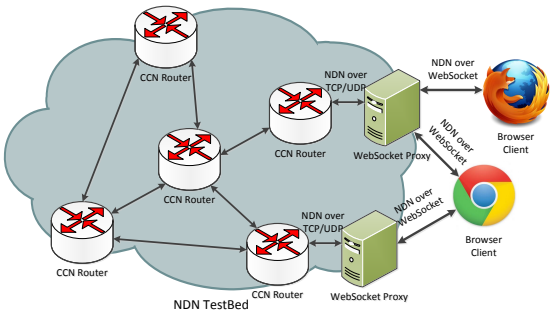

Fig. 2. WebSocket proxies connecting browsers to the NDN testbed

of "server push" capability.[2] The WebSocket protocol [3], on the other hand, provides a generic JavaScript interface to enable full-duplex TCP connections to any remote host. NDN.JS adopts WebSocket as the default transport for NDN packet exchange. It also provides a flexible interface for developers to implement their own transport services using other technologies. For example, a XPCOM-based transport, described in Section V, provides optional, higher-performance native TCP and UDP functionality for Firefox Add-Ons.

Since current CCN routers do not support the WebSocket protocol, we have developed a WebSocket proxy that accepts WebSocket connections from NDN.JS instances and passes the packets over TCP/UDP to CCN routers. We expect the WebSocket transport to be supported by CCNx in near future to provide a more efficient and higher-performance solution.

Fig. 2 illustrates the relationship between the three entities: client Web browsers, WebSocket proxies and CCN routers in an NDN testbed. The client browser establishes WebSocket connection to the proxy, which in turn maintains the TCP/UDP connection to the router. Upon receiving a WebSocket frame, the proxy extracts the original NDN packet and forwards the packet to the CCN router via TCP or UDP. When an NDN packet is received from the router, the proxy encodes the packet into WebSocket frames before forwarding to the browser. The proxy is also responsible for fragmentation and reassembly if the NDN packet cannot fit into a single WebSocket frame.

The WebSocket proxy is implemented in straightforward JavaScript code running on Node.js [7], a widely used JavaScript execution platform, and using an open source WebSocket library [8]. The proxy listens on TCP port number 9696, one port number above the CCN daemon (9695). This allows operators to deploy the WebSocket proxy and the CCN daemon on the same device.

[2]A HTTP server cannot initiate communication to a client: it cannot send data to the client without the client sending a request first, nor can it send a request to the client.

E. Application Programming Interface

As discussed above, currently there are three versions of NDN API available to developers. The C API provided in CCNx package exposes low-level details of the protocol (e.g. parsing wire format ccnb), while the Java API in that package is, in our experience, abstracts away too far from the core features of the protocol for many application experiments. PyCCN [6], the Python wrapper of the C API, has the key elements of the protocol represented in a straightforward way, facilitating experimentation without requiring one to deal with the wire format in applications unless necessary. NDN.JS follows the PyCCN approach, aiming to further reduce the development and deployment effort needed to use NDN. We feel this is especially important during this early stage of NDN experimentation, even given the performance implications. Note the Web community is making continuous efforts to improve the performance of JavaScript engines, that this project can leverage.

The programming interface in NDN.JS is designed to be consistent with PyCCN. The library is first initialized with the hostname or IP address of a WebSocket proxy or other transport provider. By default, the library will select an available router from the NDN testbed. For the time being this is done by randomly selecting a running proxy while we are waiting for NDN autoconfiguration to be deployed that can provide a most appropriate selection.

After connection to the NDN network, the top level "NDN" class provides two important methods:

- expressInterest: fetch the named data from CCN routers.
- registerPrefix: publish local content by registering the content name to CCN routers.

These two methods are asynchronous, which is appropriate both for NDN Interest/Data exchange and JavaScript programming. Application developers provide callback functions (encapsulated as 'Closure' objects) to handle the responses. The following two pieces of JavaScript codes shows skeleton examples of using NDN.JS to fetch and publish contents.

```
var ndn = new NDN(); // Use default proxy selection
var DataReceivedClosure = function
    DataReceivedClosure() { Closure.call(this); };
DataReceivedClosure.prototype.upcall =
function(kind, upcallInfo) {
  if (kind == Closure.UPCALL_CONTENT) {
    var content = upcallInfo.contentObject;
    console.log(content);
  }
  return Closure.RESULT_OK;
};

ndn.onopen = function() { // 'open' event callback
  ndn.expressInterest(new Name('/ucla.edu/foobar'),
      new DataReceivedClosure()); };
```

```
ndn.transport.connectWebSocket(ndn);
```

Listing 1. Content retrieval example

```
var ndn = new NDN(); // Use default proxy selection
var ReturnDataClosure = function ReturnDataClosure()
     { Closure.call(this); };
ReturnDataClosure.prototype.upcall =
function(kind, upcallInfo) {
  if (kind == Closure.UPCALL_INTEREST) {
    // Respond to interest that generates the upcall
    var name = upcallInfo.interest.name;
    var co = new ContentObject(name,
      'Hello, world! This is NDN.JS.',
      new SignedInfo(), new Signature());
    co.sign();
    upcallInfo.contentObject = co;
    // Content will be sent out by NDN.JS
    return Closure.RESULT_INTEREST_CONSUMED;
  }
  return Closure.RESULT_OK;
};

ndn.onopen = function() { // 'open' event callback
  ndn.registerPrefix(new Name('/ucla.edu/foobar'),
      new ReturnDataClosure()); };
ndn.transport.connectWebSocket(ndn);
```

Listing 2. Content publishing example

IV. Evaluation

In this section, we analyze the performance of NDN.JS by measuring the throughput of content retrieval in different Web browsers. We compare the throughput with that of HTTP asynchronous GET and the C implementation of the CCNx command-line utilities. We also study the impact of the signature verification functionality in NDN.JS.

A. Methodology

To conduct the throughput test, we set up an isolated network environment with two Macintosh machines, the CCN router and the CCN client, directly connected to each other via a 100Mbps link. We configured the WebSocket proxy to run on the CCN router together with ccnd, which is the common practice on the current NDN testbed. To test the HTTP throughput, we also ran an Apache HTTP daemon on the router machine.

During the test, the client pulls down two images of different sizes (742KB and 28.7 MB). These files were pre-installed on the router using 'ccnputfile' utility, which automatically cuts the input file into small chunks (4096 bytes per chunk by default) with consecutive segment numbers.

We ran the test scripts on the latest versions of three popular Web browsers available in MacOSX: Firefox, Chrome and Safari. Each test is repeated for 10 times in order to take the average result. To test NDN.JS, we implemented a Closure that automatically fetches all the chunks of the file. For HTTP, we simply issued an XMLHttpRequest (XHR) call that fetches the image file from the Web server. To compute the throughput, we recorded the start and stop time of the entire fetching process in JavaScript. To test the C library, we use the 'ccncatchunks2' command in CCNx to fetch the segmented file, and then read the throughput from the log information. To keep consistent with NDN.JS and HTTP, content dumping (which introduces file I/O overhead) in this utility is disabled.

Both the NDN.JS closure and the C command implement the same pipelining algorithm similar to TCP slow start to issue multiple outstanding Interests. The Closure maintains a sliding window that constrains the number of un-replied Interests. The window size starts from 1 and is bounded by a maximum value of 32 segments. 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 a timeout happens, the window is shrinked to 1 to slow down the transmission speed.

TABLE III
Performance Impact of Content Verification in NDN.JS

Browser Type	Throughput (Mbps) with Verification Disabled	Throughput (Mbps) with Verification Enabled
Chrome	46.01	10.15
Firefox	48.66	2.313
Safari	65.66	3.319

B. Results

1) Transport Throughput: We first show the transport performance with signature verification turned off (for both NDN.JS and C utility). Table II summarizes the test results. The performance of NDN.JS varies across browsers due to different efficiency of the JavaScript engines. It is interesting to see that the performances of NDN.JS and C library are comparable, especially in the large file case. The throughput of fetching small file versus large file with NDN.JS has noticeable difference because in longer data transmission the initial 'slow start' phase has less impact on the overall performance. Note that since the test machines are on the same Ethernet link, there is no Interest timeout throughout the fetching process.

As expected, XHR (HTTP) generally achieves better performance than NDN.JS and the C utility, The most significant factor for NDN's much lower performance than HTTP is the parsing of CCNx packet header for *every packet*, as opposed to HTTP transfer where image file is considered one object and the browser only parses HTTP header once. In the current untuned implementation, CCNx header parsing incurs significant overhead.

2) Signature Verification Speed: We repeated the small file throughput test with signature verification turned on. The comparison with the previous result is shown in Table III. We can see that the signature verification operations greatly reduces the performance of content fetching. Among the three browsers, Chrome gives the best performance when verification is turned on. However, the throughput is still about 5 times slower than in the non-verification case. This is due to the fact that JavaScript is not optimized for computation-intensive tasks, which makes signature verification the bottleneck in content processing.

TABLE II
THROUGHPUT TEST RESULTS (UNIT: MBPS)

File Size	NDN.JS (WebSocket)			Native HTTP (XHR)			ccncatchunks2
	Chrome	Firefox	Safari	Chrome	Firefox	Safari	(C utility)
742 KB	46.01	48.66	65.66	83.6	84.73	82.51	71.22
28.7 MB	62.26	71.07	74.75	88.71	89.33	89.02	75.41

V. APPLICATION EXAMPLE: FIREFOX NDN PROTOCOL AND TOOLBAR ADD-ON

A. Basic Design

We leverage the core library of 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 provide a familiar browser interface for experimenting with NDN.

For this Add-On, an Interest is converted into a URI using the following 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 an Interest packet and requests the content from upstream. 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 immediately enable content fetching via NDN with potential benefit of multicast delivery and in-network caching, the Add-On also helps us to explore how to provide various application-level possibilities to content publishers and consumers via NDN. A few such features are described below, implemented in JavaScript using NDN.JS.

1) Long-term Secure Links: NDN support in the browser may provide the option for consumers to verify that (static) named 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" [9]. 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. If, much later, the user would like to view the same content again, she can click the bookmark and retrieve the 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 for data retrieval often require more than one Interest/Data exchange. Retrieving the latest version of named content is an example; 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.[3] 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 Other NDN Semantics in the UI: Through the Add-On, we will also 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, where applicable.

4) XPCOM Transport Service: Taking advantage of the modularity of NDN.JS, the add-on implements a new transport service using the Firefox XPCOM interface [11]. The new

[3]For an example, see the discussion of live video streaming in [10].

transport allows the add-on to directly connect to CCN routers via raw TCP or UDP socket, which eliminates the need of proxy indirection.

VI. Discussion

NDN.JS is both a practical solution to facilitate NDN deployment by enabling the protocol in browsers, and an exploratory step toward building an NDN-based Web architecture. As such, the project raises many questions and challenges.

An important challenge at the library level is to develop appropriate security models in support of real applications, and to establish sandboxing guidelines similar to those used by JavaScript itself. A simple example is in the case of storage, a critical feature of NDN nodes. Current Web browsers prohibit Web pages from accessing local disk storage, which forces NDN.JS to store all the fetched or published contents in the memory. Although HTML5 provides a set of File API [12] for Web APPs to access a sand-boxed file system, this API is not yet widely supported. We envision that with the popularization of the new File API it would be feasible to implement persistent local storage for NDN.JS, to support both publishing and caching of data when browsers are disconnected.

Another key challenge is how to interpret NDN ContentObjects for rendering by the browser. MIME types are the current standard for recognizing and rendering content "on the Web", and NDN.JS likely should integrate such content typing (perhaps through a well-defined metadata naming scheme) so that typical content can be rendered properly in the browser. This issue is application-specific but a convention is still necessary to guide the Web development with NDN.JS.

An open question arises from introducing NDN to existing applications such as the Firefox Add-on, which exposes NDN semantics directly to the user: how one should enable proper specification of various components and selectors (content versioning, Interest, segment number, etc.) of the NDN Interest. The Add-On makes an initial attempt by encoding Interests into a single URI string and providing different features in the UI (toolbar) to represent NDN capabilities.

A fourth challenge comes from JavaScript itself, which as a scripting language does not achieve the high performance of the C library nor provide direct memory management and other opportunities for optimization. One of the major performance hurdle is the signature verification functionality, which requires JavaScript to manipulate large integer objects. We expect that new standard API (such as the Web Crypotography API [13] under development at W3C) may appear in the future to offload some of the computational complexity to the browser kernel.

Finally, we plan to explore how NDN offers certain revolutionary, rather than evolutionary, changes for Web applications – in particular through the availability of per-packet signatures and typical use of encryption for access control. Traditional Web security model enforces the "same origin policy (SOP)", which requires that the scripts and Web pages loaded from one domain can normally only access resources published from that domain. Modern JavaScript transports, such as the WebSocket protocol, allow cross-origin resource access via 'Origin' header negotiation between client and server [14]. The NDN convention of encryption-based access control (e.g. using signed Interest [15]) provides a more flexible and scalable alternative to the origin-based security model. Currently we have only scratched the surface in this research field.

VII. Conclusion and Future Work

This paper described NDN.JS, a JavaScript client library for Named Data Networking, to facilitate the development and deployment effort for NDN-based applications on the browser platform. We also created a Firefox add-on as a first use case of NDN.JS. This exercise helps shed new insights on how to migrate existing platforms onto NDN data transport, as well as remaining open issues.

We believe NDN.JS can also serve as the first step toward redesigning the Web architecture with the data-centric semantics provided by NDN. Our planned future work includes completing the implementation of content verification and key management functionalities, the development of applications that explore the possibilities of an NDN-based Web, addressing the challenges discussed above, and improving the NDN.JS design. The release of NDN.JS code serves as our invitation to all interested parties to join us in this new exciting pursuit.

References

[1] "Named Data Networking (NDN) Project," Oct. 2010. [Online]. Available: http://www.named-data.net/techreport/TR001ndn-proj.pdf

[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] 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

[4] "Content-centric networking in c documentation." [Online]. Available: http://www.ccnx.org/releases/latest/doc/ccode/html/

[5] "Content-centric networking in java documentation." [Online]. Available: http://www.ccnx.org/releases/latest/doc/javacode/html/

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

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

[8] "ws: a node.js websocket implementation." [Online]. Available: http://einaros.github.com/ws/

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

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

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

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

[13] 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/

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

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