Faster Content Distribution with Content Addressable NDN Repository

Junxiao Shi
Department of Computer Science
The University of Arizona
shijunxiao@email.arizona.edu

***Abstract*—The effectiveness of universal caching in Named Data Networking depends on data naming. NDN routers cannot identify duplicate contents published under different Names. This paper proposes an enhancement to NDN repository so that duplicate contents could be identified by their hash, in order to save bandwidth and shorten download completion time. The repository indexes Data packets not only by Name, but also by hash of payload. Client applications could retrieve chunks by hash from a nearby neighbor. Therefore, the same payload does not have to traverse the Internet twice. Our evaluation shows that total download time for two Linux Mint disc images is reduced by up to 38%.**

Keywords named data networking, content addressability

I. Introduction

Primary usage of Internet has shifted from point-to-point conversations to content distribution. TCP/IP, modeled after the telephone network, is not designed for today's Internet services and applications. Named Data Networking (NDN) [1] is an emerging future Internet architecture that makes data a first-class entity.

NDN network identifies data by *Name*. The *consumer* sends an *Interest* packet that contains a Name. The Interest is forwarded to potential producers, according to the Name rather than address. The *producer* replies with a *Data* packet that contains a matching Name, the payload. The Data packet goes back to the consumer, by following the reverse path of the Interest. Every Data packet must be signed by the producer, and could be verified by anyone.

NDN network supports universal caching. Every router may opportunistically cache Data packets, which could be used to satisfy future Interests with a matching Name. Any amount of caching can help save bandwidth and shorten data retrieval time. The signature in Data packet guarantees the authenticity of a Data packet, regardless of whether it comes from the producer or from a cache.

The effectiveness of universal caching relies on data naming. A good naming scheme should ensure that the same block in the same version of the same file have the same Name. When such Data packet is cached, a future Interest for the same block could be satisfied by the cache, without going all the way to the producer.

However, there are many cases that identical payload appears in Data packets under different Names: (1) When a file is modified, the name will contain a different version marker [2]. The new version typically shares a lots of common blocks with the previous version. (2) When a file is included in an uncompressed tape archive (.tar), its data blocks will appear in the tape archive which have a different name prefix. (3) When certain web content is published as several formats, the different representations (such as HTML / plain text / RSS) may share common blocks. Such duplication cannot be detected by routers by application-defined Name, so that identical payload has to be transmitted multiple times.

To identify duplicated payload under different Names and save bandwidth, this paper proposes an improvement to the NDN repository. Identical payload is identified by cryptographically secure hash, and fetched from a nearby neighbor if available. We implement Content-Addressable NDN Repository and related utility programs in C. We test the system over a virtual machine network that contains an emulated slow link, and find that it could shorten download completion time for up to 38%.

II. Design

The *repository* is a NDN application that provides persistent storage of NDN Data packets. A repository supports the network by preserving Data packets as instructed and responding to Interests requesting Data packets it holds [3].

Producer signs the Data packets, and asks the repository to store it. The repository stores Data packets in the filesystem, and maintains an index of Data Names as an on-disk balanced tree. When an Interest is forwarded to the repository, it looks up the Interest Name in the index, and replies with the original signed Data packet if a matching one exists. Data packets are immutable once stored, and cannot be overwritten or deleted; new Data packets must have a different Name.

A. Hash Index and Hash Request

We propose to have the repository maintain a *hash index*, in addition to the Name index. Whenever the repository stores a Data packet, it computes the SHA256 hash over its payload, and this hash is used as the key in hash index. It's important that the hash is computed over just the payload portion of the Data packet; Name, metadata, and signature are not covered by the hash. A stored Data packet could be retrieved not only by its Name, but also by the hash of its payload. With a cryptographically secure hash algorithm, hash collision is very unlikely, so the hash could uniquely identify a chunk of payload.

In our protocol, consumer retrieves a Data packet by hash using a *hash request*. A hash request is an Interest under the special namespace `%C1.R.SHA256` such as `/%C1.R.SHA256/`*`hash`*. The repository looks up the hash index, rather than the Name index, for hash requests. If a Data packet with that payload hash is found, a new Data packet is constructed, which takes Name from the hash request and contains the requested payload. We cannot simply send back the original Data packet, because its Name does not match the hash request Interest.

Hash requests are limited to neighbor scope (1-hop) due to scalability concerns. Although the requested payload may exist 2 or 3 hops away, we must either flood the hash request, or flood an advertisement for the existence of a payload. Flooding the hash request consumes too much network resource, and may take longer time to retrieve a chunk than downloading directly from the remote repository. Flooding the advertisement has to be done for each individual chunk, because SHA256 hash is a flat namespace; the cost on routing is prohibitive.

B. File Chunking

Regular NDN applications, such as `ndnputfile`, chunk file at fixed offsets. While fixed chunking could work with content-addressable repository, a single-byte insertion at the front of a file causes all chunks to be different, defeating the benefit of hash index.

We chunk file with Rabin fingerprints [4]. A rolling hash is computed on every overlapping 31-octet sliding window; his hash is not a cryptographically secure hash, instead it's designed for fast computation over a sliding window. A chunk boundary is claimed when this rolling hash ends with a specified number of zeros. Therefore, where to chunk the file depends on the content inside the sliding window, rather than the byte offset from beginning of the file.

In Rabin fingerprint chunking, the *expected* chunk size could be adjusted by choosing the required number of tailing zeros to claim a chunk boundary. For example, requiring the rolling hash to end with 12 zeros causes the probability of any sliding window becoming a chunk boundary to be $\frac{1}{2^{12}}$, or 1/4096. This in turn sets the expected chunk size to be 4096 octets. However, the *actual* chunk size might be far from the expectation. If the content is repetitive (eg. a long run of space characters), it's possible that there is no chunk boundary in the content, resulting in very large chunk; it's also possible that every sliding window becomes a chunk boundary, generating very small chunks.

Unbounded chunk size is not acceptable in our system, because chunks are enclosed in Data packets. Data packets cannot be too large, because large packets are inefficient or infeasible to transmit over the network. NDN currently recommends a maximum packet size of 8800 octets [5]. Data packets should not be too small, because NDN routers must maintain states for Interests used in retrieving Data packets, and small Data packets means higher network overhead per payload byte. These size restrictions on Data packets are reflected on the chunking scheme.

We limit every chunk to have at least 1024 octets, and at most 8192 octets. If a chunk boundary is found within 1024 octets since the last chunk boundary, it is ignored. If no chunk boundary has been found within 8192 octets since the last chunk boundary, one is inserted regardless of the rolling hash. Therefore, Data packets would have a reasonable size.

C. Hash List and Trust Model

After chunking the file, the producer generates a *hash list* for the files. The hash list is an XML document that contains the length and hash of every chunk of the file. It has the following format:

```
<Collection>
  <Count>n</Count>
  <Entry>
    <Length>len</Length>
    <ContentHash>hash</Length>
  </Entry>
  ... more Entry elements
</Collection>
```

n and *len* are encoded as binary numbers, and *hash* is encoded as a BLOB. The hash list XML is encoded in NDNB format, chunked as fixed size blocks, signed by the publisher, and published as in the metadata namespace as `SHA256` [2].

The hash list is signed by the publisher, and contains hashes of every chunk. Therefore, the file chunks could be verified by the hashes, which is in turn verified by the signature on the hash list. Strong signatures are no longer necessary on Data packets of file chunks. If the consumer trusts the signature on the hash list, it can trust the chunks after verifying their hashes.

III. Implementation

We implement the Content-Addressable NDN Repository in C, based on NDNx 0.2 [6]. Code is published on [7] under BSD license.

The implementation contains three programs: `caput`, `car`, and `caget`.

A. caput - publisher

`caput` is the publisher program. It chunks a file using Rabin fingerprint chunking, and publishes file segments along with hash list into local repository.

`caput` could publish into content-addressable repository `car`, as well as regular repository `ndnr`. However, `ndnr` will not serve hash request, therefore this setup is only recommended on a remote server.

Data packets created by `caput` do not carry RSA signatures, because the payload could be verified by hash. `ndngetfile` cannot retrieve these Data packets because it expects RSA signatures. If backward compatibility is desired, a command line option could instruct `caput` to RSA-sign every Data packet.

B. car - repository

`car` is the content-addressable repository program. It is modified from regular repository `ndnr`, and should be used in place of `ndnr`.

`car` is fully compatible with `ndnr`. In addition, it maintains a hash index of all Data packets (even if they are not generated by `caput`), and serves hash requests.

On all nodes in the local area network, a forwarding entry for the Name prefix `/%C1.R.SHA256` should be created and point to a multicast group that can reach all NDN nodes running `car`.

C. caget - consumer

`caget` is the consumer program.

To separate responsibility, `caget` does not fetch the hash list by itself. The caller should fetch and verify the hash list with another tool (such as **ndngetfile**).

`caget` reads the hash list, and builds a set of pending requests for each unique chunk. For each unique chunk, `caget` first sends a *hash request* to local neighbor scope. When the content-addressable repository on the local node or a nearby neighbor holds an identical chunk, it will respond to this hash request. Otherwise, if the hash request is not fulfilled within a short time, `caget` sends a *name request* with the versioned filename to retrieve those chunks from producer or caches along the route.

TABLE I: hash request vs name request

	hash request	name request
Interest Name	`/%C1.R.SHA256 /`*`hash`*	*/filename/version /segment*
scope	neighbor	global
max outstanding	30	10
timeout	500ms	4000ms
after timeout	send name request	retry twice, then fail

Table I shows a comparison between hash request and name request. Pipelining is enabled for both types of requests, by allowing a number of requests outstanding at a time.

`caget` benefits the most when many of the chunks are sitting in a nearby repository so that hash requests could be fulfilled. Even if all chunks have to retrieved from the remote repository, `caget` still benefits from intra-file duplications: if several chunks within the file are identical, only one chunk needs to be retrieved.

After a file is retrieved with `caget`, the file should be published into local `car` with `caput`, so that its chunks could be used to serve hash requests in the local area network.

IV. Workload Analysis

The benefit of content-addressable NDN repository depends on the amount of duplicate chunks in the workload. Both intra-file similarity and inter-file similarity are beneficial. Inter-file similarity enables `caget` to fetch identical chunks from local area network, if a neighbor happens to have a similar file. Intra-file similarity allows `caget` to reduce the number of Name requests even if no neighbor has a similar file, because it needs to download every unique chunk only once, using any segment number in the Name request.

A. CCNx source code

CCNx is the reference implementation of Content Centric Networking, the predecessor of NDN. CCNx project has released 29 versions since 2009 [8]. We are using these 29 versions of CCNx source code as the workload. Every file is an *uncompressed* TAR archive of one CCNx version.

Our analysis shows that, on average, 97.4% chunks are unique within a file, which means content-addressable NDN repository could same 2.6% bandwidth without using hash requests. Figure 1 shows inter-file similarity. On average, 60.3% chunks need to be downloaded if one immediate prior version exists in local area network, and 55.3% chunks need to be downloaded if all prior versions are available locally.

The percentage of inter-file duplication varies by version. Understandably, a stable version is very similar to its release candidate. For example, only 39.7% chunks are new in ccnx-0.5.1, compared to ccnx-0.5.1rc1.

We also analysis the intra-file and inter-file similarity of *compressed* TAR.GZ archives. Intra-file similarity is always zero, because DEFLATE, the algorithm underlying `gzip`, eliminates duplicate strings within 32KB distance [9]. Inter-file similarity is also very low: 99.2% or 98.2% chunks must be downloaded with one prior version or all prior versions in local area network. This means content-addressable NDN repository is not suitable for compressed workload.

B. Linux Mint images

Linux Mint is a Ubuntu derived linux distributed for desktop systems. Each release of Linux Mint contains multiple editions.

TABLE II: Linux Mint workload

	MATE 64-bit	MATE no-codecs 64-bit
filename	linuxmint-15-mate-dvd-64bit.iso	linuxmint-15-mate-dvd-nocodecs-64bit.iso
size	1000MB	981MB
media	DVD	DVD
package base	Ubuntu Raring	Ubuntu Raring
desktop	MATE 1.6	MATE 1.6
video playback	included	not included

We are using two editions of Linux Mint "Olivia" [10], as shown in Table II. The difference between these two editions is that *MATE 64-bit* contains video codecs (packages used for video playback), while *MATE no-codecs 64-bit* does not include them.

TABLE III: Linux Mint analysis

	MATE 64-bit	MATE no-codecs 64-bit
number of chunks	238436	233852
intra-file unique chunks	235509	231270
inter-file unique chunks	254276	

Table III shows the number of chunks and unique chunks in these two files. We can see that unique chunks across two files

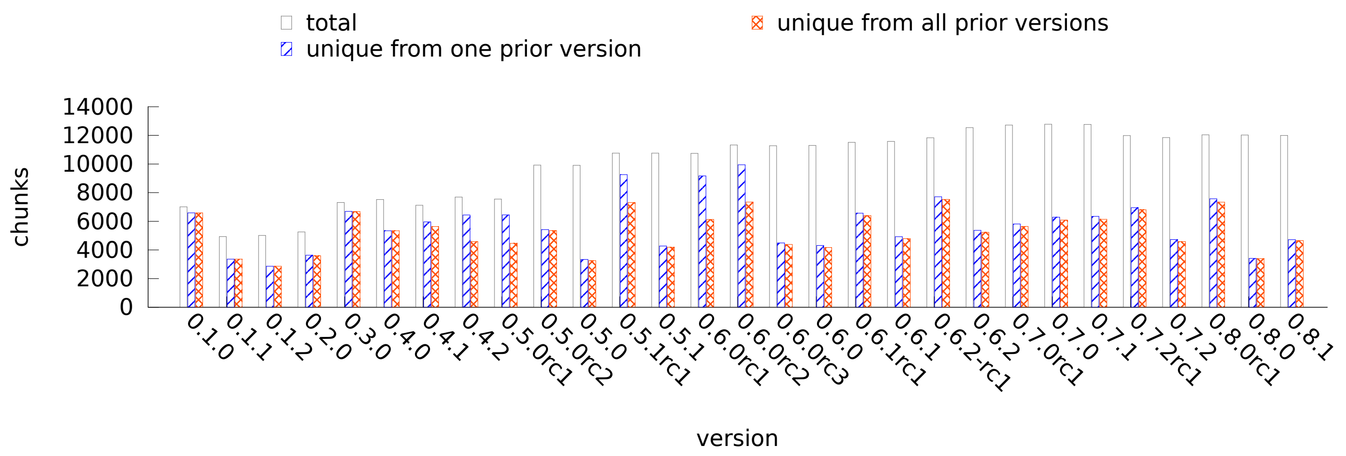


Fig. 1: inter-file similarity in uncompressed CCNx source code

are only slightly more than unique chunks in either files, which means the two files are similar to each other. In particular, if a client already has *MATE 64-bit* locally, only 18767 chunks need to be downloaded in order to construct *MATE no-codecs 64-bit*, which would be a 91.9% bandwidth saving.

Figure 2 shows the count, total bytes, and unique percentage of chunks with different chunk sizes in two files combined. The data points for 8192-octet chunk size (the maximum) are much higher than usual and are not shown in the figures: 81446 chunks (17.2%) are reaching the maximum chunk size, and they attribute for 44721 inter-file unique chunks (17.6%). 53.8% chunks are unique overall, while 54.9% 8192-octet chunks are unique. This shows that imposing a limit of maximum chunk size has little impact on similarity.

V. Performance Evaluation

The system is evaluated in a virtual machine network. Three systems are compared in the evaluation:

- *carepo* content-address repository
- *ndn* regular NDN repository
- *tftp* Trivial File Transfer Protocol

A. Environment

A network is created on VirtualBox virtualization platform. The NDN-layer topology is shown in Figure 3. Each node is allocated four Intel Xeon E5645 CPU cores, and 1GB memory. Ubuntu Server 12.04 and NDNx 0.2 are running on all nodes.

Gateway and clients are connected via high-speed local area network. Server and gateway are connected via a slow Internet link simulated by netem [11] on an additional virtual machine acting as an IP router. On the slow link, the download rate (server to gateway) caps at 2.5Mbps, and the upload rate (gateway to server) caps at 0.5Mbps; 20ms delay is added in both directions.

Table IV shows the software used in three setups. In *carepo*, a regular NDN repository `ndnr` is running on the server node, and files are published into this repository with `caput`; `caget` is executed on the client nodes to download files via gateway and the slow link, and downloaded files are published into local `car`. In *ndn*, `ndnr` is running on the server node, and files are published with `ndnputfile`; `ndngetfile` is executed on the client nodes to download files via gateway and the slow link. In *tftp*, `tftpd-hpa` is running on the server; only one client is used; `atftp` is executed on the client node to download files via the slow link, and it is configured to use 8000-octet block size.

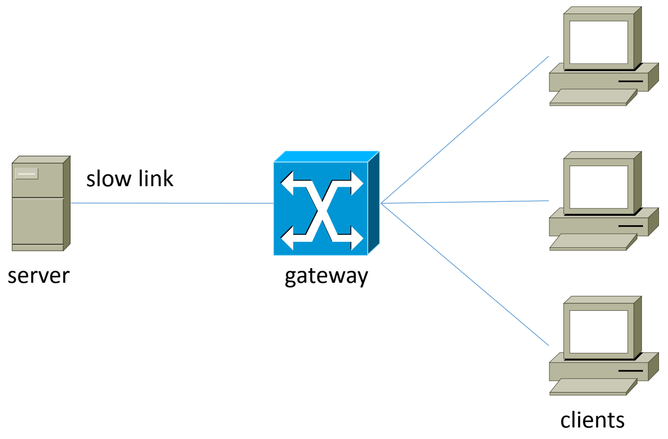


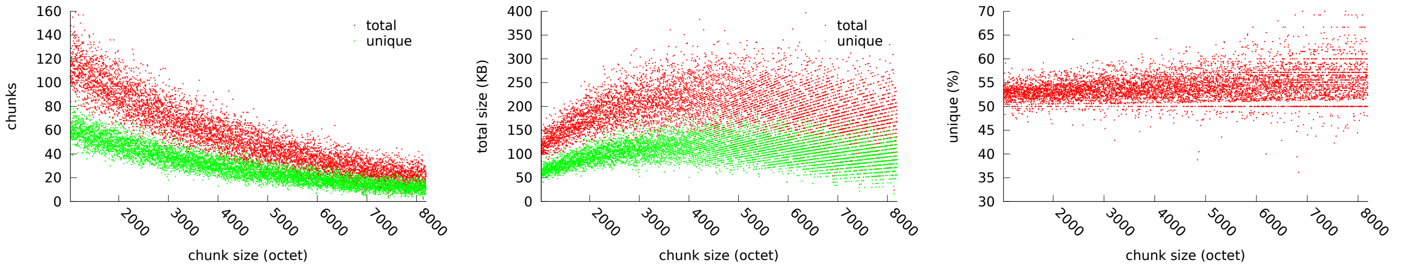
Fig. 3: network topology

TABLE IV: Software Environment

	server	gateway	client
carepo	ndnd ndnr caput	ndnd	ndnd car caget
ndn	ndnd ndnr ndnputfile	ndnd	ndnd ndngetfile
tftp	tftpd-hpa		atftp

B. Download Time

a) CCNx source code: We download three versions of uncompressed CCNx source code onto clients.



(a) number of chunks by chunk size (b) total bytes by chunk size (c) unique percentage by chunk size

Fig. 2: Linux Mint analysis by chunk size

1) ccnx-0.6.0.tar is downloaded onto client1.
2) ccnx-0.6.1.tar is downloaded onto client2.
3) ccnx-0.6.2.tar is downloaded onto client3.

In *carepo*, file is published onto local `car` after each download.

TABLE V: CCNx source code download time

	download time (s)			
	0.6.0 52300KB	0.6.1 53530KB	0.6.2 58750KB	total 164580KB
carepo	196	141	165	502
ndn	206	210	231	648
tftp	336	343	379	1058

Table V shows CCNx soure code download times in three setups. *carepo* is a clear win here. It is faster than *ndn* even for the first file due to the bandwidth saving from intra-file similarity.

b) Linux Mint images: We download two Linux Mint images onto clients.

1) We first download MATE 64-bit onto client1. After the first download completes, for *carepo* we publish the file onto local `car`; publishing time is not included in download time.
2) Then we download MATE no-codecs 64-bit onto client2.

TABLE VI: Linux Mint download time

	download time (s)		
	MATE 64-bit 1000MB	MATE no-codecs 64-bit 981MB	total 1981MB
carepo	4108	792	4900
ndn	4041	3942	7984

Table VI shows Linux Mint download times in *carepo* and *ndn* setups; the *tftp* implementation being tested limits file size to 500MB, so it could not complete the download. We can see that *carepo* is slightly slower than *ndn* in downloading the first file, because intra-file similarity is low (Table III) and `caget` takes time waiting for hash requests to time out. But *carepo* is significantly faster in downloading the second file, because only 9% of the chunks need to be downloaded over the slow link. The total download time for two files in *carepo* is 38% less than *ndn*.

C. Publishing Overhead

The faster download of *carepo* comes with a price: longer publishing time.

TABLE VII: comparison of publishing overhead

	carepo		*ndn*	
	`caput`	`car`	`ndnputfile`	`ndnr`
chunking	Rabin		fixed	
SHA256	payload	payload	Data packet	
RSA-sign	hash list		all chunks	
index		Name index & hash index		Name index

Table VII compares the publishing overhead of *carepo* and *ndn*. `caput` uses Rabin chunking algorithm, which is more expensive than fixed size chunking. Both `caput` and `ndnputfile` compute SHA256 hash on part of the Data packet, and the overheads are roughly the same. However, `car` must also compute the hash, because even if the hash is carried in the Data packet, it could not be trusted because there's no signature; this is an added overhead. `caput` saves the cost of RSA-signing every Data packet, because chunks are verified by hash. `car` must maintain two indexes, while `ndnr` has only one index.

TABLE VIII: Linux Mint publishing time

	publisher	repo	publishing time (s)		
			MATE no-codecs 64-bit	MATE 64-bit	total
baseline	ndnputfile	ndnr	110	128	238
+Rabin	caput-sign	ndnr	271	276	547
-sign	caput	ndnr	146	145	291
+hashindex	caput	car	507	372	879

Table VIII shows the publishing time under different configuration. We publish Linux Mint MATE no-codecs 64-bit followed by MATE 64-bit into the same repository.

The baseline uses regular NDN repository and publisher. Then we substitute `ndnputfile` with `caput`, and force `caput` to RSA-sign every Data packet; this adds the overhead of Rabin fingerprints. Then we run `caput` without forcing RSA signatures, to get the benefit of not RSA-signing every chunk. Finally we substitute `ndnr` with `car`, adding the overhead of computing hash at repository and maintaining hash index.

We can see that publishing time of *carepo* (`caput+car`) is 3.8x publishing time of *ndn* (`ndnputfile+ndnr`). Computing SHA256 and maintaining hash index is the biggest overhead of *carepo*.

However, we argue that the increased publishing time is not a big problem in our scenario. On the server, a published file will be served to many clients, and average publishing cost for each download is low. Furthermore, the server could run `ndnr` instead of `car` which has lower overhead, if no client exists in neighbor scope.

On a client node, the file is available for use as soon as download completes. Publishing time does not affect user experience. Downloaded files should be published into local *car* to help neighbors, but this can be done in the background.

VI. Conclusion

NDN universal caching is not capable of suppress duplicate payload under different Names. We use content-addressability to solve this problem by identifying identical payload by hash. A hash index is added to NDN repository, and the downloader finds duplicate chunks on nearby nodes by hash. The system is tested with two workloads, and we find that download time is reduced by 38% for two similar DVD images, but publishing overhead is increased.

An alternative design is to maintain a hash index on the ContentStore in some or all routers, and attach a hash as part of Interest. We will explore this alternative design as our future work.

References

[1] 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

[2] "NDNx Basic Name Conventions." [Online]. Available: http://named-data.net/doc/0.2/technical/NameConventions.html

[3] "NDNx Repository Protocols." [Online]. Available: http://named-data.net/doc/0.2/technical/RepoProtocol.html

[4] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 174–187, Oct. 2001. [Online]. Available: http://doi.acm.org/10.1145/502059.502052

[5] N. Briggs, "Re: [ccnx-users] ccnsendchunks vs ccnseqwriter." [Online]. Available: http://www.ccnx.org/pipermail/ccnx-users/2013-October/001355.html

[6] "NDN Platform." [Online]. Available: http://named-data.net/codebase/platform/

[7] J. Shi, "carepo git repository." [Online]. Available: https://github.com/yoursunny/carepo

[8] "Index of /releases, ccnx.org." [Online]. Available: http://www.ccnx.org/releases/

[9] P. Deutsch, "DEFLATE Compressed Data Format Specification version 1.3," RFC 1951 (Informational), Internet Engineering Task Force, May 1996. [Online]. Available: http://www.ietf.org/rfc/rfc1951.txt

[10] "Editions for Linux Mint 15 "Olivia"." [Online]. Available: http://www.linuxmint.com/release.php?id=20

[11] "netem." [Online]. Available: http://www.linuxfoundation.org/collaborate/workgroups/networking/netem