NDN Technical Report NDN-0027, 2014. http://named-data.net/techreports.html
Revision 1: October 27, 2014

NDNFS: An NDN-friendly File System

Wentao Shang
UCLA
wentao@cs.ucla.edu

Zhe Wen
UCLA
wenzhe@cs.ucla.edu

Qiuhan Ding
Tsinghua University
dingqiuhan@gmail.com

Alexander Afanasyev
UCLA
afanasev@cs.ucla.edu

Lixia Zhang
UCLA
lixia@cs.ucla.edu

ABSTRACT

NDNFS is a file system designed for Named Data Networking (NDN) and supports efficient data access by both local and remote applications. It provides the standard file system interface for local file operations, but stores files internally as NDN Data packets, which can be directly sent out as responses to the incoming Interests, saving the overhead of encoding the packets and generating signatures on the fly. A metadata protocol is also designed to assist remote access to NDNFS by communicating directory contents and file metadata explicitly with data consumers. By providing consistent data naming and organizing across different layers, NDNFS implements storage, transmission and security protection functionalities with a single data unit: the NDN Data packet.

1. INTRODUCTION

Named Data Networking (NDN) [11, 7] proposes a future Internet architecture that shifts from "host-oriented" into "data-oriented" communication paradigm. In NDN network, every piece of data contains an NDN name as the unique identifier and a signature of the data producer, which secures the binding between the name and content. Since the signature generation process is expensive, NDN data storage services usually store pre-packaged Data packets, ready to be transmitted over the network without incurring additional overhead. This essentially unifies storage, process and transmission in the same data unit, bringing the benefits of integrity checking and tamper resistance.

Lots of applications on the NDN testbed make use of a permanent in-network storage provided by an NDN architectural component called *repo*, currently implemented by `ccnr` as part of the Project CCNx implementation [1]. However, this implementation has several critical limitations. First, `ccnr` only provides network-based read/write access to the data via Interest/Data packet exchange, which prohibits non-NDN applications (e.g., existing text editors or image viewers) from accessing the local repo data. Second, `ccnr` implements a persistent permanent storage, without the ability to remove any data after it is stored, which creates hurdles for applications that want to store large volumes of data temporarily in the repo.

In this paper, we designed and implemented an "NDN-friendly" file system called `NDNFS`, as a new type of data storage service for NDN, aiming to support applications that require both remote and local access to the files stored in the repo. NDNFS creates a customized file system that exposes POSIX file API, allowing basic operations like creating, modifying and deleting files and directories. Internally, it chops the file data into NDN-formated segments and stores them in a local database. A special NDNFS server daemon is employed to handle Interests for the data stored in NDNFS by pulling out the "network-ready" Data packets from the database and pushing them down to the wire. This makes NDNFS "NDN-friendly" by optimizing file data access through NDN protocol.

The rest of the paper is organized as follows. Section 2 introduces the design of NDNFS. Section 3 describes the implementation of our prototype. Section 4 analyzes the performance of the prototype by measuring the file access speed. Section 5 discusses previous works related to NDNFS. Finally, Section 6 concludes the paper and addresses future work.

2. SYSTEM DESIGN

This section describes the design of NDNFS. We first give an overview of the system architecture and then discuss specific design issues in detail.

2.1 Overview

NDNFS consists of two inter-dependent parts: the file system core and the server module, as is shown in Figure 1. The file system core manages the internal organization, NDN name assignment, and the back-end storage (using an embedded SQL database) of the file data. It is also responsible for chopping large files into small segments and encapsulating each segment into NDN wire format. The file system core exposes UNIX file operation interfaces to the operating system, so that local applications can access and manage the stored files

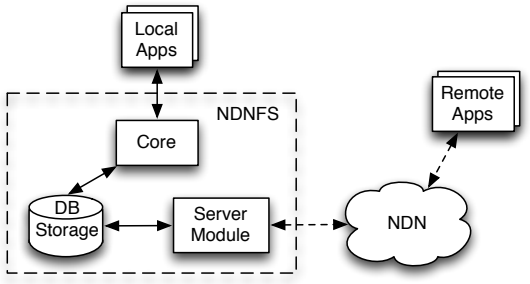

Figure 1: NDNFS basic architecture

via POSIX-style system calls (*open, close, read, write, unlink, etc.*).

The built-in NDNFS server provides remote access to the local file data over NDN. It parses the incoming Interests and searches the local storage for the matching Data packets. It also supports a metadata protocol that provides meta-information about the files and directories through RPC-style commands, allowing consumers to explore the NDNFS naming hierarchy remotely.

The initial design of NDNFS described in this paper provides full support for local *read/write* operations, while defining only *read* interface for the remote access. Enabling remote *write* access requires a careful design of access control scheme to handle security and privacy concerns.

2.2 Naming structure

File and NDN names are very similar to each other: they both are hierarchical and uniquely identify objects, either on the file system or in an NDN environment. The hierarchical structure of file names can be directly applied in NDN: a file with a path `/a/b/c.txt` can correspond to exactly the same NDN name `/a/b/c.txt`. However, due to file segmenting, the NDN name of each data segment (collectively representing a file) needs to contain at least a sequence number to indicate the order of the file block. Also, in order to distinguish different updates of a file, NDNFS adds a version number after the file name (but before the segment number, following `ccnr` naming conventions) to uniquely identify each edition of the same file (see Figure 2). Note that the version and segment information is invisible to local applications from the file system interface.

Although it is possible to simply map the root directory `./` of NDNFS into the NDN root prefix `/`, in the actual deployment it is often desirable (e.g., in order to facilitate Interests forwarding) to associate the root with some predefined global prefix so that all the data segments in NDNFS will be named under this prefix. For example, the NDNFS instance in the above figure is configured to use `/ndn/ucla.edu/irl/user/ndnfs/` as the

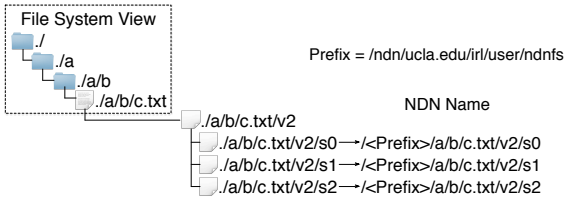

Figure 2: NDNFS naming structure

global root prefix, then the NDN name for the first segment of the file `./a/b/c.txt` will become `/ndn/ucla.edu/irl/user/ndnfs/a/b/c.txt/v2/s0` (here `v2` indicates the second version). This essentially avoids namespace conflict if multiple NDNFS instances are running on the network. It also makes NDNFS file block names routable on the global Internet.

2.3 Storing file segments

Just like Unix file system stores file data as blocks on disk, NDNFS stores files as a collection of segments in wire-formated NDN Data packets. The key motivation for segmenting file data is to fit file transmission into the packet switching communication model of NDN network. A properly selected segment size can adapt to the MTU (Maximum Transmission Unit) of the underlying network so that every file block can be directly transmitted on the network without fragmentation. It also allows data consumer to fast recover one or two lost blocks without retransmitting the entire file.

For each file segment, a segment number is appended to the end of the NDN name of that segment to support sequencing and assembly. The segment number simply starts from zero and grows sequentially, and the size of each segments in our prototype is fixed to 8192 bytes. When a file is *read* by local applications, NDNFS combines all the segments of that file in order, and returns the aggregated data back to the applications.[1] For local *write* operations, NDNFS splits the written file into fixed-size segments, packs them into 'network-ready' Data packets, signs and store them in the back-end database for future use. Because of this packetizing process (especially the data signing process), the local *write* operation becomes more expensive in NDNFS than in the normal file systems that store files as plain byte chunks.

When the file is accessed remotely through the network interface, NDNFS extracts the requested file path, version, and segment information from the incoming Interest, and perform a lookup in the back-end database. If the requested entry is found, the requested segment is

[1]The standard file system interface allows reading a particular part of a file (starting from some offset and ending at certain length), in which case only the segments related to that part are processed.

immediately forwarded to the network. Otherwise the incoming Interest is simply ignored.

2.4 Version control

In the NDN architecture design, Data packets always bind to a unique name and are immutable. This unique feature requires NDNFS to update the version number of all the file blocks when that file is modified (even though some of these blocks are actually unaffected by the change). NDNFS automatically performs this version control without user intervention since normal applications have no concept about NDN versioning and also lacks the capability to manage the NDNFS internal version information. NDNFS generates version numbers consistently based on the system clock which guarantees local chronological ordering.[2] This avoids the problem in `ccnr` where careless users may produce version numbers that lead to incorrect order of the file history, confusing applications and users.

Internally, NDNFS maintains two version numbers for each file entry: the *current* version and a *temporary* version. When the file is opened with *write* access, NDNFS will generate a temporary version and all the subsequent write operations will simply edit this temporary version without creating a new one. After the file descriptor is closed, the temporary version is committed and becomes the "current version", while the previous version becomes historical. The automatic version control in NDNFS offers a straightforward solution to file system journaling: by default, NDNFS will remove the old version when a new version is staged (which is suitable for lightweight service deployment); but when configured to run in journaling mode, NDNFS will keep the entire file change history (subject to local disk capacity) so that previous versions can be recovered at any time in the future.

However, the versioning algorithm described above only supports "open-after-close" consistency, but not "read-after-write" consistency. That is, until the writer closes its file descriptor, any changes made to the temporary version will be invisible to the readers who access the file concurrently with the writer. We could have implemented NDNFS to update the file version for *every* write operations (such as `write` and `truncate`). But that would incur significant performance penalty since we would have to update the NDN name and signature of every file block for each single write. Here we decided to trade fine-grained consistency for performance boost. This issue actually reflects a deeper problem in the general file system design: the applications and storages use different data units (file versus segment), which causes conflicts in the semantics of various functionalities.

[2]Here we made the assumption that the system clock will not rewind.

2.5 Deletion

Deleting files and directories is a common operation on file systems. In NDNFS, however, there is a subtlety in the semantics of data removal: due to the pervasive caching in the NDN network, the file blocks may still reside in some routers' Content Store after that file is deleted from NDNFS local storage. `ccnr` implements data removal by publishing a special type of Data packet (called `GONE` type [2]) under the same NDN name to indicate that this piece of data is no longer available.[3] This forces the size of the `ccnr` local storage to grow monotonically, which becomes a major drawback of `ccnr`. NDNFS takes an simpler approach by removing the data from local storage and relying on the *FreshnessSeconds* setting to purge the router cache. Future Interests asking for the removed data (possibly carrying Exclude filters to bypass cached versions in the network) will be ignored by NDNFS server, eventually causing a timeout at the consumer side.

2.6 Metadata protocol

To assist remote file fetching via NDN network, NDNFS employs a metadata protocol to communicate information about the file system with remote consumers. The functionalities of the metadata protocol resembles the Network File System (NFS) [5] services, the most basic ones including getting file attributes and listing entries in a directory. The metadata protocol implements RPC-style request/reply communication through Interest/Data exchange, where the RPC command is encoded in the name of the Interest. The metadata packets are signed by the same key that signs the normal file data and are verifiable at the consumer side.

Since NDNFS remote access is read-only, the current metadata protocol design only supports reading file/directory attributes and directory contents, which is similar to the NFS operations `GETATTR` and `READDIR` respectively.[4] The name of the metadata is constructed by appending the RPC command (in the form of an NDNFS-specific name component) at the end of the corresponding file/directory name so that the requests for the metadata can be routed to the NDNFS producer. In the current implementation, the `GETATTR` command is expressed by the special name component `NDNFS.GETATTR`, while the `READDIR` command is expressed by the special name component `NDNFS.READDIR`.

For `GETATTR` command, the returned information contains not only the general file meta-info such as size and access time, but also the NDN-specific information like

[3]This special type of `GONE` packet is only used in `ccnr` Sync protocol, which helps maintain the monotonic growth property of the set of NDN names synced between the repos. The old versions are still cached inside the network until timeout.

[4]Reading file data is already supported implicitly by fetching the data block through basic NDN protocol.

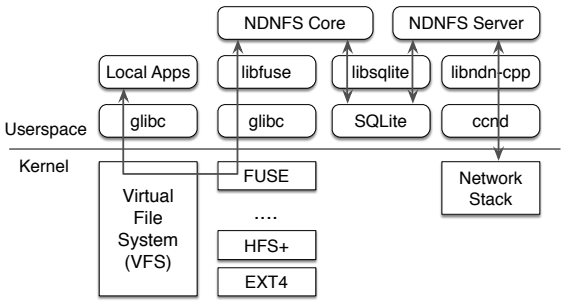

Figure 3: NDNFS system implementation

current version number and total segments count. For `READDIR` command, the returned data packet contains a list of file/sub-directory names under the directory described by this metadata. These information is critical for the consumers to be able to issue Interests with exact file data names (including accurate version and segment numbers), reducing the usage of Interest selectors which will likely cause Interest forwarding overhead at NDN routers.

Since files are versioned by NDNFS, the corresponding metadata also needs to be versioned to keep track of the file changes. The version number of file metadata is identical to that of the corresponding file and is appended after the command component. A simple mechanism for bootstrapping version information is to explicitly ask for the metadata (with the command component attached but no version component) and use Exclude filter to bypass unwanted versions.[5] Another possible solution is to rely on *FreshnessSeconds* field in the NDN Data packet to purge the outdated copies in the network.

Directories are special in NDNFS in the sense that they represent prefixes in the NDN namespace without any Data packet associated with their names. In `ccnr` convention, the Interest with a name pointing to a general prefix is satisfied by any data under the sub-namespace of that prefix (that could match the Interest selectors). However, since prefixes in NDNFS have concrete semantics (directory paths in the file system), NDNFS takes a different approach by always returning the directory meta-info for Interests that exactly match a directory name. That is, all such Interests are treated as an implicit `GETATTR` RPC call. This is both meaningful under the file system semantics and also removes the NDNFS server's burden of searching data under the directory and matching selectors.

3. IMPLEMENTATION

In this section we introduce our proof-of-concept implementation of NDNFS [8]. The prototype system is written in C++ and tested on Mac OS X platform. Figure 3 shows the general architecture of the current prototype.

3.1 File System Core

We built the NDNFS as a user-space file system on top of the FUSE [3] kernel module, which is a virtual file system service that links the file system interface with user-defined callbacks. The following list shows the file and directory operations implemented in NDNFS, which are necessary to support basic UNIX commands such as `ls`, `rm`, `cp`, and `mkdir`:

- File operations: `create, open, read, write, truncate, release, unlink`
- Directory operations: `mkdir, rmdir, readdir`
- Meta-info operations: `chmod, getattr`

The file system core runs as a single-threaded daemon service and mounts the file storage to a user-specified directory (usually referred to as the "mount point"). When local applications access files or sub-directories inside the mount point folder through file system APIs, the FUSE kernel service redirects the requests to the NDNFS core daemon, which will serve the request by looking up the data in the back-end storage.

The file data blocks and associated meta-info are stored in an SQLite [4]-based local database. The database maintains three tables: a *file entries* table (indexed by the file paths) that contains the meta-info about files and directories, similar to the *inode* table in Unix file systems; a *file versions* table (indexed by the file paths and version numbers) that keeps track of the version numbers for the files; and a *file segments* table (indexed by the file paths, version and segment numbers) that stores actual file data blobs. The queries from the FUSE callbacks are usually performed as SQL `JOINs` among multiple tables.

3.2 Server Module

The NDNFS server is implemented as an NDN data producer that runs alongside the NDNFS core daemon. Once started, it registers the NDNFS global prefix to local `ccnd` and waits for incoming Interests. Unlike other local applications that access NDNFS through the standard file interface, the server module directly interacts with and retrieve data from the back-end database. A more complex approach could map the entire NDN name hierarchy into file path tree, allowing each individual Data chunk to be accessed as Unix files. In that case the NDNFS server may also access the internal data through standard file system calls.

When an Interest packet arrives, NDNFS server first extracts NDNFS local file path (and possibly version

[5]Therefore the usage of Interest selectors is not completely avoided by the metadata protocol.

and segment numbers) from the Interest name, and then uses this path as the search key to find possible matching data in the SQLite database. An Interest with no version or segment number in the name is satisfied by the metadata (regardless of whether the metadata protocol command component is present), which is generated on the fly by the NDNFS server.[6] An Interest whose name contains at least a version number but no metadata protocol command is either satisfied by a matching segment or discarded if the requested version is not the latest one.[7] Table 1 summaries the rules for interpreting NDNFS names under any combination of command, version and segment components. Since there is no ambiguity for names under the NDNFS semantics, the server ignores any selectors in the Interest packet.

4. EVALUATION

In this section we present the performance evaluation results of the prototype implementation. We focus on the speed of the *read* and *write* access, which are the most critical file operations. All the tests are conducted on an iMac with 3.2 GHz Intel Core i3 processor, 4 GB DDR3 memory and 7200 rpm SATA hard drive. The hosting operating system is Mac OS X version 10.7.

4.1 Local Access

To measure the local *write* speed, we use the built-in `cp` command to copy a 5.7 MB image from HFS+ (the native file system on Mac OS X) into NDNFS. Then we use the *time* command to measure the running time of the copy process and compute the throughput. We did not measure the local *read* speed in this test due to the interference of file caching mechanism in Mac OS X.

Since the RSA signature computation operation is the main bottleneck in the NDNFS file writing, using larger file block size will effectively reduce this overhead by producing less segments to be signed per file. In this test, we repeated the same measurement under two different NDNFS configurations: one with 8 kB file block size and the other 64 kB. The result is shown in Table 2.

As is expected, using 64 kB block size almost increased the file writing speed by 4 times compared to the 8 kB case. However, such a large segment size is impractical for transmission on the NDN network. A more sophisticated implementation would employ a two-level segmenting scheme: at file system level, the file blocks are managed in large data size; at NDN level, the large file blocks are further chopped into smaller segments, which can be grouped together using Merkle hash tree to reduce signing overhead.[8]

Table 2: Local file writing speed (Mbps)

Block-size	Mean	Dev
8 kB	1.34	0.23
64 kB	5.28	0.37

Table 3: NDN file fetching throughput

Unit: kbps		NDNFS	`ccnr`
Sequential	mean	121.725	204.316
	dev	0.237	1.192
Pipeline	mean	284.079	430.329
	dev	1.107	1.924

4.2 Network Access

We evaluate the network *read* access speed by fetching a pre-installed 100 MB file from NDNFS server and `ccnr` (using the same segment size of 8 kB) and comparing the downloading throughput. We implement two versions of file fetching client: the first version fetches all the file segments sequentially, while the second version employs a fixed-size pipeline mechanism that always keeps eight outstanding Interests.

To capture the performance of the network interface without introducing additional overhead, the test client simply discards the fetched data rather than saving it to local disk. Since `ccnr` does not support the trust model used by NDNFS, we also skipped data verification in the test client and only measure the data transfer speed. A special flag in the Interest packet (called `AnswerOriginKind`) is set to bypass the local `ccnd` cache (so that the result really reflects the through at the server). Moreover, the test client runs on the same machine with the server and communicates through local socket interface in order to remove interference on the network.

Table 3 summarizes the measurement results, where each test case is repeated 100 times. In sequential fetching, NDNFS server is roughly 40% slower than `ccnr`. Pipelining improves both performances but NDNFS is still about 34% slower than `ccnr`. The difference in the data throughput is easy to understand: for every request, NDNFS has the extra overhead of parsing and executing SQL commands in the database while `ccnr` uses a customized indexing scheme for its data; also, NDNFS server is implemented with a C++ NDN library, which is less efficient than the "pure C" imple-

[6]Currently we keep the file system core and the server as two separate modules. A more sophisticated design would integrate the two modules and perform both local and remote queries through the file system interface. In that case, the metadata packet will also be pre-generated and stored in NDNFS alongside normal data blocks.

[7]In journaling mode, NDNFS may optionally return the historical version tracked by the journal, which is not implemented in the current prototype.

[8]We cannot use Merkle hash tree across file segments because the file system semantics requires each file block to be independent and complete.

Table 1: Interpreting Interest names in the NDNFS server

Special Components			Action
command	version	segment	
×	×	×	Return the latest attribute meta-info for the requested file or directory.
√	×	×	Execute the command and return the latest metadata.
×	√	×	Return the first segment of the requested data or discard if version is stale.
√	√	×	Return the requested metadata or discard if version is stale.
×	√	√	Return the requested data or discard if version or segment does not exist.
–	–	–	Discard in any other cases.

mentation of `ccnr`.

5. RELATED WORKS

There is a rich collection of literatures on file system design and implementation, especially for networked and distributed file systems. The traditional NFS [5] design focuses on providing consistent interface with the same control granularity across local and remote file systems. It relies on IP-style security mechanism to protect data authenticity and integrity. Most distributed file systems deployed in the data centers (such as Google File System [6] and Hadoop Distributed File System [9]) assume well-managed clusters where distributed storages are inter-connected. Therefore security is not a major concern in such systems and researches in this area usually focus on data partition, system reliability and performance optimization.

A closely related research work is Iris [10], an authenticated file system with inherent security support. It assumes an enterprise-scale file system hosted in untrusted public cloud (and therefore requires strong security guarantee). It is similar to NDNFS in many aspects: it relies on heavy caching at the so-called "Iris portal" (deployed near the enterprise customers to reduce file fetching latency); it provides data integrity protection using Merkle hash tree and ensures freshness through versioning. However, Iris solely relies on the portal as the trust anchor to provide the correct data checksum and version numbers without using any cryptographic signature. Another difference is that Iris performs per-block versioning rather than using the same version across the whole file. This allows the decoupling of updates among different file blocks but loses the semantic consistency between data version and file modification time.

The NDNFS design is unique in that it directly uses NDN Data packet as the file system data unit. Therefore it inherits all the security properties from the NDN architecture and allows applications and the network to share the same security model. It also saves the overhead of converting data units between different layers, making the file system "network-friendly". The current simple design addresses the middleground between local and network file systems (i.e., a local file system designed with network accessibility in mind) and can be extended into RPC-style network file system or distributed file system with data-oriented security support.

6. CONCLUSION AND FUTURE WORK

In this paper, we described the design and implementation of NDNFS, an NDN-friendly file system that provides both local and network access to the file data. We discussed the performance of NDNFS through comparison with native file system and `ccnr`.

The current prototype implements only a limited set of functionalities in support of basic NDN operations, which leaves room for future research effort. Regarding the NDNFS core service, we plan to implement more file interfaces, such as file locking and renaming, to turn NDNFS into a full-featured file system. We will also explore new types of back-end data storage, such as storing all Data packets as individual plain files on the local disk. On the other hand, the evaluation result drives us to seek potential improvements in file writing speed, such as optimizing the NDN C++ library currently used in NDNFS or devising more complex file segmenting scheme.

For NDNFS server, we will consider how to integrate access control in order to implement remote *write* functionality, which is missing in the current prototype. It is also possible to turn NDNFS into a distributed file-sharing service like ChronoShare [12] by adding synchronization protocols on top of it. We encourage wider adoption of NDNFS in other applications to explore the potential usage of this new type of NDN data storage and also welcome contributions to this newly-started research project.

7. REFERENCES

[1] Ccnr(1) manual page. Available at `http://www.ccnx.org/releases/latest/doc/manpages/ccnr.1.html`.

[2] Ccnx content object. Available at `http://www.ccnx.org/releases/latest/doc/technical/ContentObject.html`.

[3] FUSE: Filesystem in userspace. Available at `http://www.named-data.net/`.

[4] Sqlite home page. Available at `http://www.sqlite.org/`.

[5] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification. RFC 1813 (Informational), June 1995.

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[7] V. Jacobson et al. Networking named content. In *Proc. of CoNEXT*, 2009.

[8] W. Shang et al. NDNFS source code repository. `http://github.com/named-data/NDNFS`, 2013.

[9] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[10] E. Stefanov, M. van Dijk, A. Juels, and A. Oprea. Iris: A scalable cloud file system with efficient integrity checks. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 229–238. ACM, 2012.

[11] L. Zhang et al. Named data networking (NDN) project. TR 0001, NDN, 2010.

[12] 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.

AFTERWORD

The NDNFS project was originally inspired by a closely related application called `ChronoShare` [12], which provides Dropbox-like, but de-centralized file sharing service over NDN. ChronoShare has the specific requirement of providing both local and network access to the files in the sharing folder, which becomes the motivation of building an "NDN-friendly" file system. However, after NDNFS was implemented, we realized that it did not meet ChronoShare's functional requirements and was too complex to be integrated into the ChronoShare codebase. Here is a list of major limitations:

- First, NDNFS does not implement all the necessary file system interfaces (such as file renaming and locking), which leads to usability problems.
- Second, the FUSE-based file system requires extra handling during application startup, such as mounting the file system to the correct path, which adds more complexity to the applications on top of it.
- Third, the implementation suffers from a performance problem as explained below, which significantly affects the I/O speed and hurts the user experience.

The first version of NDNFS used a *NoSQL* database called `MongoDB` as its backend data store. Preliminary performance measurement showed significant slowdown in read/write speed as compared to the `ccnr` repository. We then switched to use the embedded database `SQLite`, with the hope that it would mitigate the bottleneck in the backend storage. However, as shown in the evaluation results in this paper, the performance is still worse than that of `ccnr`. Another related project, `repo-ng` (which aims to create a replacement for ccnr with additional desired features), also ran into similar performance issues when it tried to use `SQLite` as the backend. Through further discussion and analysis, we concluded that the performance bottleneck is due to the delay of the database I/O operations. Consequently `repo-ng` implemented an in-memory name index to avoid multiple DB lookup when executing Interest/Data matching.

It has been over a year since this paper was drafted in late 2013. During this time period, the NDN protocol specification has undergone significant change, with the packet format being moved from the old binary XML format to the new Type-Length-Value (TLV) format. The NDN team also developed a new forwarder, NDN Forwarding Daemon (NFD), to fully support the new packet format. Since the original NDNFS implementation was based on the old XML format, it is no longer usable. Considering that NDNFS did not fit into *ChronoShare* and we did not find other use cases where NDNFS would apply, we decided not to port NDNFS to the new NDN packet format. However we feel it is worthwhile to publish this work as a technical report for future reference, and for sharing with others what we learned from the NDNFS exercise.