Analysis of NDN Repository Architectures

Inchan Hwang[1], Dabin Kim[2], Young-Bae Ko[2]
[1]Graduate School of Software Engineering
[2] Graduate School of Computer Engineering
Ajou University, Suwon, South Korea
inchan@uns.ajou.ac.kr, dabin912@uns.ajou.ac.kr, youngko@ajou.ac.kr

POSTER PAPER

***Abstract*—Named data networking (NDN) is a newly proposed networking architecture, which a few network scientists believe that it is an appropriate protocol to distribute Big Data files in the network of supercomputing domain. This new scheme is organized to reduce high network congestion resulted from it. In addition to many current NDN applications, such as climate modeling data service, another supercomputing domain, its repository applications also have been developed. However, compared to the ordinary file systems, they now significantly underperform in writing operations. It will be highly likely to prevent scientists and engineers from applying NDN over their networks because it could turn to be as a system bottleneck in the performance critical area. In this paper, writing operations of those repositories will be examined and how performance can differ according to the software architectures placed in them. Finally, we will discuss optimization schemes to make it perform better in a large scaled computing system.**

Keywords— Named Data Networking(NDN); repository; file system

I. INTRODUCTION

In the era of the modern Internet, efficient content distribution has grown more significant than ever as well as the data server management. Some large companies like Google, Facebook, and other banking firms run datacenters where multiple server machines work to publish data when users request it. For an organized data management with better throughput, scheme like grouping data by category and put them in a certain storage is applied. For example, data accessed frequently are placed in a certain node with the highest bandwidth; large files are handled at a computer with bigger storage capacity. Administrative data are only accessed at a node surrounded by Intrusion Prevention System (IPS)/Intrusion Detection System (IDS), where strong security software is set up. Yet, data around those nodes must be easily retrieved and read by authorized personnel. Such highly flexible, hierarchical, and security demanding data management scheme have been first made possible with Network File System (NFS) services [1].

Alongside with the evolution of distributed file systems, the Internet traffic has dramatically increased for past years since the dawn of IoT, and Big Data. Its growth is unprecedented, all those data from various kinds of machines and mobile devices being connected to the Internet, has already reached its bandwidth limit. Many network scientists have been on alert against this traffic outburst and studied in order to solve these problems. A solution proposed to it is Named Data Networking (NDN) [2], one of the most prominent future Internet architectures, is able to cope with growing traffic demand. It integrates cache into the network architecture. Its each NDN router caches every travelling data packet and this packet is fetched again from a physically closer distance when a request to the identical data is made. It is expected to result in massive congestion reduction around the data producer and the network itself. This new architecture's applications have been researched across many domains [3-7]. However, due to its relatively short history of NDN compared with the conventional IP network architecture, there is limited knowledge of its application to the high performance computing field and Big Data processing in terms of NDN storage and performance effect on computer systems. Two NDN storage solutions have been suggested so far, such as NDNFS [8], and repo-ng [9]. They are the instances of NFS to serve contents over NDN so that average engineers and computer scientists research and take advantage of the brand-new network architecture on large file service at super computers.

In this paper, we focus on the analysis of two different NDN repositories; NDNFS, and repo-ng. In particular, we make those contributions. We are going to describe the full working experiences of repo-ng and NDNFS. In order to develop and research NDN over data intensive science and high performance computing, being aware of different repositories' strengths and weaknesses on it is essential for computer scientists and engineers to decide whether to adapt it to their systems. We provide the comparisons of those repositories in terms of performance, memory usage, and CPU use rates while working with difference size of Big Data files. Moreover, we make full-fledged analysis of the large data file insertion operations of those two repositories with the descriptions of their architectures. Lastly, we put forward to a few discussions and considerations on its implementation in high performance computing and data intensive science.

II. Brief overveiw of NDN repositories

A. NDNFS

NDNFS is a NDN-friendly file system that is a tool that helps content owners publish contents over the NDN network. It is built upon Filesystem in User Space (FUSE) [10]. Without much effort to integrate it into the native OS file system, it allows users to mount NDN file system with just few commands. When users copy and paste the file from the source location into NDNFS file system to publish it, NDNFS chops it into many NDN segments. They are in the network-ready state after the segmentation process into the file system so that when an interest for a particular data gets in contact with NDNFS daemon, it is ready to be sent over NDN. It also provides a simple working example scenario over Web with Fire-fox and NDN.JS add-on [11]. Contents published by NDNFS over NDN network can be viewed and searched with Firefox web browser. In addition, all the files and directories in the file system can be viewed and searched in the terminal and the file manager of the OS even though they are actually split in NDN packets. Other average file operations, such as read, write, modify, and delete can be done over them. Features of NDNFS are summarized below:

- *FUSE file system*: Data is written in FUSE file system so that average OS file operations like “cp” and “mv” could be performed.
- *Metadata management*: The metadata of each segment is kept in SQLite across three tables.
- *Extra tools*: It works with NDN.JS Firefox add-on so that users navigate the file system and download the published contents.

B. Repo-ng

Repo-ng is an application of NDN persistent in-network storage complying with repo-protocol that allows data management operations of repo-ng, including file read, insertion, and deletion. Compared with the native Interest packet, repo-ng employed the concept of signed Interest [12], modified Interest packet for the authentication of a data consumer’s control command. Users insert contents at the repo with signed Interests that have encoded users’ commands to do the requested operations at repo-ng. Contrary to NDNFS, all the contents are stored in SQLite with names, and data itself. It also provides a few simple tools to insert and retrieve contents from repo-ng. “*ndngetfile*” and “*ndnputfile*” are the ones that are included in its package for insertion and retrieval operations respectively. Features of Repo-ng are summarized below:

- *Data management*: Data and its name are stored in SQLite blob data type.
- *Signed Interests*: Write, read, and delete operations are done by receiving signed command Interests. This one is an extended version of the standard Interests, which adds security features on it, authorizing the operating personnel.
- *Extra tool*: “*ndnputfile*”, and “*ndngetfile*” are provided with repo-ng to retrieve and write data in repo.

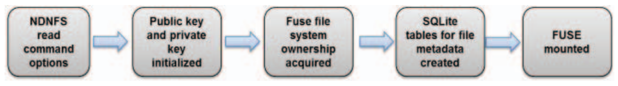
NDNFS read command options
Public key and private key initialized
Fuse file system ownership acquired
SQLite tables for file metadata created
FUSE mounted

Fig. 1. Initialization process of NDNFS.

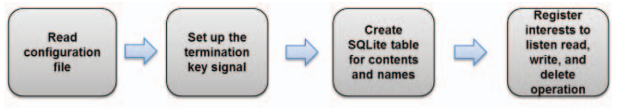
Read configuration file
Set up the termination key signal
Create SQLite table for contents and names
Register interests to listen read, write, and delete operation

Fig. 2. Initialization process of repo-ng.

III. Architecture Analysis and Comparison of NDNFS and Repo-ng

A. Initialization process

1) NDNFS

NDNFS undergoes the initialization process before any data is inserted into them. In this phase of process shown in Fig. 1, it sets up all the components required to operate themselves over the network, such as keychain, database table creation, and file system initialization. The investigation of this process is the first step to learn about the blueprint of its entire architecture. First, it reads options after NDNFS execution file for configuration, such as actual directory to store files, the mount point where users access the file system, the prefix name to publish the contents over NDN, the path of the log file, and the location of the metadata DB. Afterwards, it initializes key chain to sign packets later. Then, it acquires the current user and group IDs for NDNFS access privilege. Third, it creates SQLite tables, such as **‘file_system’**, **‘file_version’**, and **‘file_segments’**. The ‘file_system’ contains a list of files in the file system, its current version, its mime_type, the signature-encoding scheme in the column ‘*type*’. Another column ‘*ready_signed*’ is unused currently but is going to support asynchronous signature, which is now being researched.

Each file’s information is recorded in metadata tables, such as size of each file. Lastly, ‘file_segments’ table is intended to accommodate the information of stored segments, for example, signature, segment sequential number, and file path. After the creation of those tables, it returns the OS with *fuse_main()* so that FUSE will work in NDNFS way with its own defined methods for Linux file operations.

2) Repo-ng

Unlike NDNFS, repo-ng reads the configuration file to set up its environment. Then, it sets a key input to terminate the application. Otherwise, it waits for command Interests to arrive at the repository daemon. Afterwards, it initializes the SQLite DB to store data and its corresponding names. It also saves information on public keys in ‘*keyLocatorHash*’ column about the key selection in the public key chain so that the data consumer may decrypt the signature to verify the receiving data packets. This process is visualized in Fig. 2.

B. File insertion

1) NDNFS

When a user wants a file to be published over NDN, one copies it onto the file system. When it is pasted into the mount

TABLE I. NDN_REPO DB SCHEMA OF REPO-NG

Attribute	Type	Description
id	integer	primary key of the table.
name	blob	NDN name for contents
data	blob	contents in binary format
keyLocatorHash	blob	public key information

TABLE II. FILE_SEGMENTS DB SCHEMA OF NDNFS

Attribute	Type	Description
path	text	file path(equivalent to "name" in repo-ng)
version	integer	version information of each segment
segment	integer	number for each segment
signature	blob	signature of each segment

point of the FUSE, NDNFS investigates whether there is already a file identical to it. If not, it reads the list of files from the target directory. If there is not a file to write, it creates a new file to append segments being copied from the source. Next, it makes sure the target file's existence, and reads the file's extension and decides the mime type and saved it into the SQLite database.

Finally, it begins writing to the file, the target file is incremented by 4kB, and the increment continues until it reads the end of the source. Each step, it writes version information into SQLite table. After it completely finishes writing the file, here comes the most expensive operation next, sign each segment, and store it into the SQLite. This step is indispensable. NDN specification demands all data packets to be signed for stronger security. And then, write operation is finished.

2) Repo-ng

File insertion into the repo-ng is different from NDNFS. Instead of the utilization of OS file operation calls, repo file insertion tool like '*ndnputfile*' issues file insertion Interests to the repo-ng. After the Interests arrive at it, it recognizes the authentication on the command Interest. Only authorized Interest insertion command allows a file to be stored into it. This Interest also has information on the '*StartBlockID'* and '*EndBlockID*' of the desired data, or it has the selector information to retrieve it.

Once repo-ng has the enough information to retrieve the target data, it expresses Interests to the target node where the target contents are placed. It receives data packets in response to the previously issued Interests. After the last data packet landed, '*ndnputfile*' sends insertion status check Interest to the repo-ng. If there has been no error occurred at repo-ng, it answers with insertion complete message.

C. Observations on SQLite tables

As depicted in Table I, repo-ng has three blob typed attributes, and one column with integer, which functions as the primary key of the table. Data attribute reserves all data packets, and name attribute contains its corresponding name. '*keyLocatorHash*' attribute has the information on a key to decrypt the signature in the data packet. All of these attributes' type is blob.

However, as shown in Table II, '*path*' attribute of NDNFS that is identical to '*name*' attribute in repo-ng is a text type. There is no blob data type in it except for '*signature*' attribute, which seem to result in higher performance than that of repo-ng, described in next section.

TABLE III. INSERTION PERFORMANCE COMPARISON

Metrics (%)	200MB		700MB		1GB	
	NDNFS	repo-ng	NDNFS	repo-ng	NDNFS	repo-ng
CPU	60%	100%	59.3%	100%	63%	100%
Memory	0.1%	16.3%	0.1%	55.3%	0.1%	77%

IV. PERFORMANCE EVALUATION

We compared the performance of those two repositories while they put files into the storages. All the experiments are carried out on an Ubuntu 15.10 system with Intel Xeon® CPU 3.4Ghz 8 Cores, 14GB RAM, 50GB hard drive. The NDN platform we used were NFD v0.4.0, NDN-cxx v0.4.0-beta2, NDNFS v0.3, NDN-CPP v0.4.0, and repo-ng released on Nov. 21st, 2015. We selected 5 sample files weighing 200MB, 700MB, 1GB, 2GB, and 3GB respectively. We did not modify the default packet size set in NDN-cxx so that data transmission between '*ndnputfile*' and repo-ng was facilitated in 8K. To collect benchmark datasets, a bash script, running "ps" commands per second to read memory consumption and CPU resource, was made and it wrote the readings into text files

The statistics in Table III were collected while several large files were inserted into repo-ng and NDNFS. Generally, repo-ng performs worse than NDNFS. In case of 1GB file insertion, repo-ng took up more memory space than NDNFS. It took approximately 77% of the total system memory while NDNFS never took more than 0.1% of the total space. It also used nearly 100% of CPU resource whereas NDNFS consumed around 60% of its total. This memory consumption indicated because repo-ng stores all the data, such as contents, names, and the signature in memory until it finishes the reading. The reason for low memory consumption of NDNFS comes from its size of data writing into the file system at each cycle. It is always fixed to be 4KB, which is nearly 0% of the system memory.

Fig. 3 shows time taken values across a variety of sample Big Data files. In general, repo-ng takes longer to insert a file into the repository. In case of 1GB file, repo-ng took 2215.4 seconds in average to finish insertion. However, only 39.8s elapsed with NDNFS for the same file. This trend is repeated with other cases. 1130.4 seconds in average passed while repo-ng was inserting 700MB, but it only consumed 22 seconds for NDNFS. As for 3GB and 2GB files, repo-ng was unable to complete the insertion. It kept consuming the system memory and space in the swap file. The system-monitoring tool showed that repo-ng used up all of 14GB RAM and the virtual memory, and the system crashed for inadequate memory space. Thus, those values were detached from the table because it was unable to record the values for those 2GB and 3GB files. Time consumption is due to its one-to-one NDN communication nature. "*ndnputfile*" and repo-ng exchange interest and data one-by-one in NDN manner. It is an extremely slow process

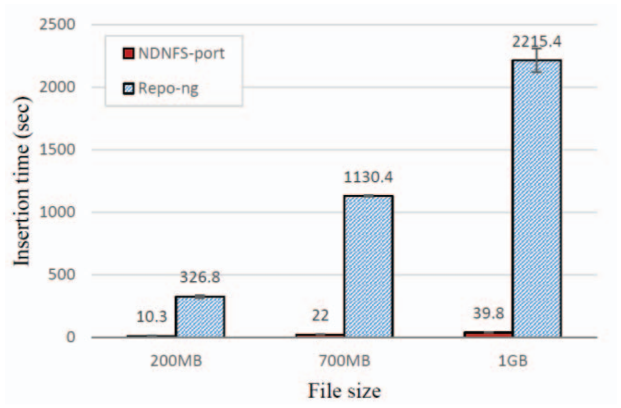

Fig. 3. Insertion time comparison for 200MB, 700MB, and 1GB files

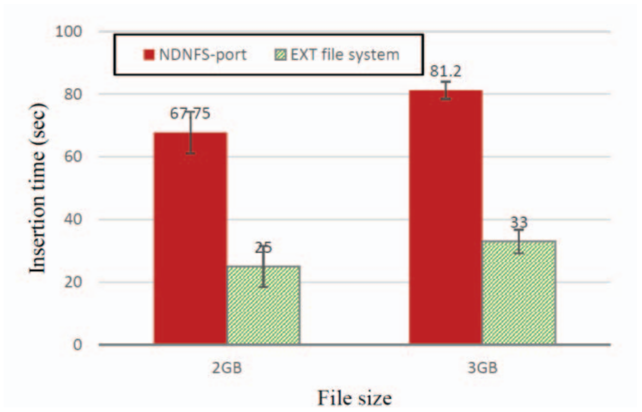

Fig. 4. Insertion time comparison for 2GB and 3GB files

Fig. 4 is a graph of time measurements while 2GB and 3GB files were being inserted into a destination folder and the NDNFS. Although file operations of NDNFS works quicker than the other one, It still operates slowly in comparison with ordinary Linux file system, Ext4. This bottleneck is believed to be caused by SQLite operations' frequent write of metadata into SQLite per each write cycle [8].

V. Discussion and Conclusion

Both NDN repositories make use of SQLite. However, their performance statistics show significantly lower performance than the average Linux file system. Their differences could be due to the uses of blob data types in SQLite tables [13]. Repo-ng's data storage algorithm totally relies on SQLite, but NDNFS also exploits FUSE with it. For better performance of SQLite, repo-ng stores all the data in memory first like contents, names, and the signature itself. This algorithm could hurt overall system performance when it inserts a large size file, like Big Data and other data intensive science files, resulting in as drawback to the application to high performance computing. Data shown in the table III depicts that repo-ng's write operation takes up a large portion of the system memory due to those reasons. In addition, NDNFS also suffers performance degradation because of SQLite. When it is compared with a file operation of ext4, severe bottleneck is observed. It is caused by frequent write operations of signature and other data into SQLite blob data type.

In conclusion, a new NDN data-storing scheme must be suggested to apply NDN storage in the high performance-computing field, which makes no use of blob data type. It is expected to improve system performance in overall, especially if one inserts a file over gigabyte.

Acknowledgement

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education(NRF-2015R1D1A1A01059049)

References

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

[2] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in ACM CoNEXT, Rome, Italy, 2009.

[3] C. Fan, S. Shannigrahi, S. DiBenedetto, C. Olschanowsky, C. Papadopoulos, and H. Newman, "Managing scientific data with named data networking," in ACM SIGHPC Workshop on Network-aware Data Management (NDM), Austin, TX, 2015.

[4] G. Grassi, D. Pesavento, G. Pau, R. Vuyyuru, R. Wakikawa, and L. Zhang, "VANET via named data networking," in IEEE INFOCOM Workshop on Name Oriented Mobility (NOM), Toronto, Canada, 2014.

[5] M. Meisel, V. Pappas, and L. Zhang, "Ad hoc networking via named data," in ACM MobiArch, Chicago, IL, 2010.

[6] V. Jacobson, D. K. Smetters, N. H. Briggs, M. F. Plass, P. Stewart, J. D. Thornton, and R. L. Braynard, "VoCCN: Voice-over content-centric cetworks," in ACM CoNEXT Workshop on ReArch, Rome, Italy, 2009.

[7] J. Burke, A. Horn, and A. Marianantoni, "Authenticated lighting control using named data networking," Technical Report, the NDN Project Team, 2012.

[8] W. Shang, Z. Wen, Q. Ding, A. Afanasyev, and L. Zhang, "Ndnfs: An ndn-friendly file system," NDN Technical Report NDN-0027, Revision 1, Oct. 2014.

[9] Repo-ng: Next generation of NDN repository, *http://redmine.named-data.net/projects/repo-ng/wiki*

[10] FUSE: Filesystem in User Space. *http://fuse.sourceforge.net/*.

[11] W. Shang, J. Thompson, M. Cherkaoui, J. Burke, and L. Zhang. "NDN.JS: A javascript client library for named data networking," in IEEE INFOCOM Workshop on NOMEN, Turin, Italy, 2013.

[12] Signed Interest, *http://redmine.named-data.net/projects/ndn-cxx/wiki/SignedInterest*

[13] Internal Versus External BLOBs in SQLite. *https://www.sqlite.org/intern-v-extern-blob.html*