NCFS: On the Practicality and Extensibility of a Network-Coding-Based Distributed File System

Yuchong Hu[†], Chiu-Man Yu[‡], Yan Kit Li[‡], Patrick P. C. Lee[‡], John C. S. Lui[‡]
[†]The Institute of Network Coding, The Chinese University of Hong Kong, Hong Kong
[‡]Dept of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong
ychu@inc.cuhk.edu.hk, {cmyu, ykli7, pclee, cslui}@cse.cuhk.edu.hk

***Abstract*—An emerging application of network coding is to improve the robustness of distributed storage. Recent theoretical work has shown that a class of regenerating codes, which are based on the concept of network coding, can improve the data repair performance over traditional storage schemes such as erasure coding. However, there remain open issues regarding the feasibility of deploying regenerating codes in practical storage systems. We present NCFS, a distributed file system that realizes regenerating codes under real network settings. NCFS transparently stripes data across multiple storage nodes, without requiring the storage nodes to coordinate among themselves. It adopts a layered design that allows extensibility, such that different storage schemes can be readily included into NCFS. We deploy and evaluate our NCFS prototype in different real network settings. In particular, we use NCFS to conduct an empirical study of different storage schemes, including the traditional erasure codes RAID-5 and RAID-6, and a special family of regenerating codes that are based on E-MBR [16]. Our work provides a practical and extensible platform for realizing theories of regenerating codes in distributed file systems.**

***Keywords*—network coding, distributed file system, implementation and experimentation**

I. Introduction

With the increasing growth of data to be managed, *distributed storage systems* provide a reliable platform for storing massive amounts of data over a set of storage nodes that are distributed over a network. A real-life business model of distributed storage is *cloud storage* (e.g., Amazon S3 [3] and Windows Azure [21]), which enables enterprises and individuals to outsource their data backups to third-party repositories in the Internet.

One key feature of distributed storage is data reliability, which generally refers to the *redundancy* of data storage. Specifically, given the pre-determined level of redundancy, the distributed storage system must sustain normal I/O operations within a tolerable number of node failures. In addition, in order to maintain the required redundancy, the storage system must support *data repair*, which involves reading data from existing nodes and reconstructing essential data in the new nodes. It is critical that the repair process is *timely*, so as to minimize the probability of losing all data should more nodes be failed before the data repair process is completed.

Recent studies (e.g., [5], [11], [16], [18]) propose a class of fast data repair schemes based on *network coding* [2] for distributed storage systems. Such network-coding-based schemes, or called *regenerating codes*, seek to intelligently mix and combine data blocks in existing nodes, and regenerate data blocks at new nodes. It is theoretically shown that regenerating codes can improve the data repair performance over traditional redundancy approaches such as erasure codes (e.g., RAID-5, RAID-6).

However, there remain open issues regarding the feasibility of deploying regenerating codes in practical distributed storage systems. Most existing studies focus on theoretical analysis. They mainly assume that storage nodes are intelligent, in the sense that nodes can inter-communicate and collaboratively conduct data repair. Some regenerating codes (e.g., [15], [19], [20]) may even require the support of encoding/decoding functions. Such intelligence assumptions require that storage nodes be programmable, and hence will limit the deployable platforms for practical storage systems. Thus, the key motivation of this paper is to explore the deployment of regenerating codes in practical distributed storage systems.

In this paper, we present the design, implementation, and empirical experimentation of *NCFS*, a network-coding-based distributed file system. NCFS is a file system built on FUSE [8], a programmable user-space framework that provides interfaces for implementing file system operations. It acts like a proxy that interconnects multiple storage nodes. It relays regular read/write operations between user applications and storage nodes. It also relays data among storage nodes during the data repair process, so that storage nodes do not need the intelligence to coordinate among themselves. Thus, regenerating codes can be implemented in real storage systems without requiring additional functional changes in storage nodes.

NCFS supports a specific regenerating coding scheme called the *Exact Minimum Bandwidth Regenerating (E-MBR)* codes [16], which seek to minimize repair bandwidth. We adapt E-MBR, which is proposed from a theoretical perspective, into practical implementation. NCFS also supports RAID-based erasure coding schemes, so as to enable us to conduct a comprehensive empirical study of different classes of data repair schemes for distributed storage under real network settings. To our knowledge, NCFS is the first work that realizes regenerating codes in a practical distributed file system.

The contributions of this paper are summarized as follows:

- We design NCFS, a distributed file system that supports general read/write operations in a distributed storage setting, while enabling data repair during node failures.

The work of Y. Hu was partially supported by a grant from the University Grants Committee of the Hong Kong Special Administrative Region, China (Project No. AoE/E-02/08).

- NCFS adopts a layered design that enables *extensibility*. Specifically, we can implement different storage schemes without changing the file system logic. Also, we can have NCFS connect to different types of storage nodes without affecting the file system design and storage schemes.
- We implement a proof-of-concept prototype of NCFS and deploy it different local area network settings. We then empirically compare the performance of read, write, and repair operations of different storage schemes including RAID-5, RAID-6, and E-MBR. This enables us to understand the overall practical performance of a storage scheme in real deployment.

The rest of the paper proceeds as follows. Section II describes the background and related work on regenerating codes. Section III presents the design and implementation of NCFS. Section IV reports empirical results of different storage schemes based on NCFS. Section V concludes.

II. Background and Related Work

A. Definitions

We consider the design of a general distributed file system, which can be realized as an array of n storage nodes. The file system organizes data into fixed-size *blocks*. We consider a stream of blocks, also called *native blocks*, that are to be written to the file system. We divide the stream into groups, each with m native blocks. The m native blocks are encoded to form c *code blocks*. In some storage schemes (e.g., E-MBR [16], see Section II-B), we may also create a duplicate copy for each native/code block. We define a *segment* as the collection of m native blocks, c code blocks, and any of their duplicate copies. The entire file system is a collection of segments.

In this paper, we consider the storage schemes that are based on a class of *maximum distance separable (MDS) codes*. An MDS code is mainly defined by the parameters n and k, such that any k ($< n$) out of n nodes can be used to reconstruct original native blocks. Given an MDS(n, k) code, the *repair degree* d is introduced for data repair, such that the repair for the lost blocks of one failed node is achieved by connecting to d nodes and regenerating the lost blocks in the new node.

B. MDS Codes in NCFS

In NCFS, we consider the following MDS codes: RAID-5 [14] and RAID-6 [12], and exact minimum bandwidth regenerating (E-MBR) codes [16]. Both RAID-5 and RAID-6 are traditional erasure codes in distributed file systems, while E-MBR uses the concept of network coding to minimize the repair bandwidth. We summarize the relationships of the parameters (i.e., n, k, d, m, c) for each of the MDS codes as follows, while Figure 1 illustrates the data layout for a special case n = 4.

RAID-5 [14]. In RAID-5, the corresponding parameters are $k = d = m = n - 1$, and c = 1. RAID-5 can tolerate at most a single node failure. In each segment, the single code block (or *parity*) is generated by the bitwise XOR-summing of the $m = n - 1$ native blocks. To recover a failed node, each lost block can be repaired from the blocks of the same segment in other surviving nodes via the bitwise XOR-summing.

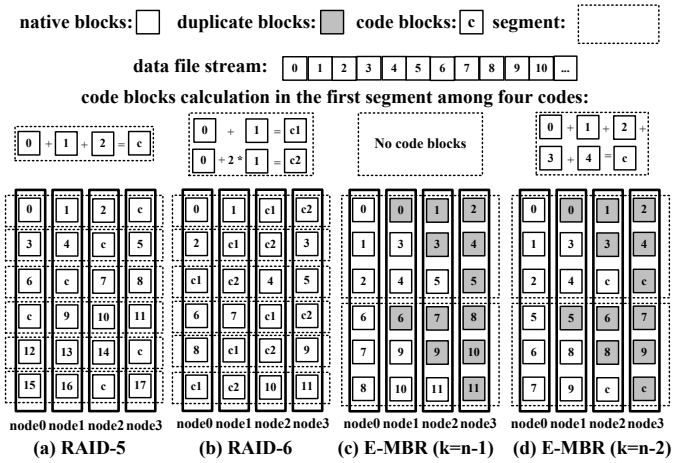


Fig. 1. The layout of a file system with different implementations of the MDS codes where n = 4.

RAID-6 [12]. In RAID-6, the corresponding parameters are $k = d = m = n - 2$, and c = 2. RAID-6 can tolerate at most two node failures with two code blocks known as the *P and Q parities*. The P parity is generated by the bitwise XOR-summing of the $m = n - 2$ native blocks similar to RAID-5, while the Q parity is generated based on Reed-Solomon coding [12]. Similar to RAID 5, if one or two nodes are failed, then each lost block can be repaired from the blocks of the same segment in other surviving nodes.

E-MBR [16]. In this paper, we focus on a particular case where $d = n - 1$, while a recent study [15] also considers all feasible values of n, k, and d. The number of native blocks in each segment is $m = k(2n - k - 1)/2$. For each native block, we create a duplicate copy, so the number of duplicate blocks in each segment is also m. By encoding the native blocks of a segment, we form $c = (n - k)(n - k - 1)/2$ code blocks. We also make duplicated copies of these c code blocks. Thus, each segment corresponds to $2(m + c)$ blocks, including the native and code blocks and their duplicate copies.

In order to compare E-MBR with RAID-5 and RAID-6 under the same level of fault tolerance, we select two values of parameter k in our implementation of the E-MBR code: (i) $k = n - 1$ and (ii) $k = n - 2$, while we point out that E-MBR can be generalized to other feasible values of k. Note that for $k = n - 1$, we must have $c = 0$, so there is no code block. On the other hand, for $k = n - 2$, we must have $c = 1$ code block, which is generated as in RAID-5, i.e., by the bitwise XOR-summing of all native blocks in the segment.

We now explain the block allocation mechanism of E-MBR for $k = n - 1$ or $k = n - 2$. We consider a segment of m native blocks M_0, M_1, $\cdots M_{m-1}$ and c code blocks C_0, C_1, $\cdots C_{c-1}$, and their duplicate copies $\overline{M}_0$, $\overline{M}_1$, $\cdots \overline{M}_{m-1}$ and $\overline{C}_0$, $\overline{C}_1$, $\cdots \overline{C}_{c-1}$, respectively. Thus, the total number of blocks in one segment is $2(m + c) = n(n - 1)$, implying that each storage node stores $(n - 1)$ blocks for each segment. To store a segment of blocks over n nodes, NCFS first allocates a segment size of free space, represented as $(n - 1) \times n$ block entries, where each row corresponds to the block offset within a segment of a node, and each column corresponds to a node. For each block B_i (either a native or code block), we search

TABLE I
THEORETICAL RESULTS OF RAID- AND E-MBR CODES [14], [16], WITH M ORIGINAL NATIVE BLOCKS BEING STORED.

	Total storage cost	Repair traffic in single-node failure
RAID-5	$M/(1-1/n)$	M
RAID-6	$M/(1-2/n)$	M
E-MBR $k=n-1$	$2M$	$2M/n$
E-MBR $k=n-2$	$\frac{2Mn(n-1)}{(n-2)(n+1)}$	$\frac{2M(n-1)}{(n-2)(n+1)}$

for a free entry from top to bottom in a column-by-column manner, starting from the leftmost column; for its duplicate copy $\overline{B}_i$, we search for a free entry from left to right in a row-by-row manner, starting from the topmost row. The allocation for each B_i starts with the native blocks $M_0, \cdots, M_{m-1}$, followed by the code blocks $C_0, \cdots, C_{m-1}$. To illustrate the block allocation mechanism, Figures 1(c) and 1(d) show the examples of ($n = 4$, $k = 3$, $m = 6$, $c = 0$) and ($n = 4$, $k = 2$, $m = 5$, $c = 1$), respectively.

To repair lost blocks during a single-node failure (for $n = k - 1$ or $n = k - 2$), we note that each native/code block has a duplicate copy, and both the block and its copy are stored in two different nodes. Thus, for each lost block, we retrieve its duplicate copy from another survival node and write it to the new node. Note that based on the block allocation mechanism, each survival node contributes exactly one block for each segment.

To repair lost blocks during a two-node failure (for $n = k - 2$), we consider two cases. If the duplicate copy of a lost block resides in a surviving node, we directly use it for repair; if both the lost block and its duplicate copy are in the two failed nodes, then we use the same approach as in RAID-5, i.e., the lost block is repaired by the bitwise XOR-summing of other native/code blocks of the same segment residing in other surviving nodes.

Theoretical results. In general, E-MBR trades off a higher storage cost for a smaller repair bandwidth as compared to the traditional RAID schemes. To understand this, suppose that M native blocks have been stored in the file system. Based on the studies in [14], [16], Table I presents the total storage cost (i.e., the total number of blocks stored) and the amount of repair traffic in a single-node failure (i.e., total number of blocks retrieved from other $d = n-1$ surviving nodes) for the above MDS codes. For $n = k - 1$, E-MBR incurs less repair traffic than RAID-5, but has higher storage cost. Similar observations are made between E-MBR and RAID-6 for $n = k - 2$.

C. Related Work on Regenerating Codes

Regenerating codes (e.g., see survey in [6]) are a class of storage schemes based on network coding for distributed storage systems. With regenerating codes, when one storage node is failed, we can repair data at a new storage node by downloading data from surviving storage nodes. There exists an optimal tradeoff spectrum between repair bandwidth and storage cost in regenerating codes, where this tradeoff has been analyzed (e.g., see [5], [22]). *Minimum storage regenerating (MSR)* codes occupy one end of the spectrum that corresponds to the minimum storage, and *minimum bandwidth regenerating (MBR)* codes occupy another end of the spectrum that corresponds to the minimum repair bandwidth.

There are generally three data repair approaches [6]: (i) *exact repair*, which regenerates the exact copies of the lost blocks of the failed node in the new node, (ii) *functional repair*, which may regenerate different copies from the lost blocks so long as the MDS property is maintained, and (iii) a hybrid of both. In general, with functional repair, some native blocks may no longer be kept after repair, so we need to access all blocks in a segment to decode a native block. This may not be desirable for general file systems as the read accesses will be slowed down.

To achieve fast read/write operations in a file system, it is important to maintain the code in *systematic form* (i.e., a copy of each native block is kept in storage). Thus, exact repair has received attention in literature, including the exact MSR (E-MSR) code [19], [20] and the exact MBR (E-MBR) code [15], [16]. The above studies focus on one-loss repair, while multi-loss repair is studied (e.g., [11], [18]). There is another repair model called *exact repair of the systematic part* [6], which is a hybrid of exact repair and functional repair while keeping the storage in systematic part. On the other hand, among all the above codes, only E-MBR (with the repair degree $d = n - 1$) does not require storage nodes be programmable to support encoding/decoding operations. As a starting point, we adopt E-MBR as a building block in our current NCFS prototype.

Most existing studies for network-coding-based distributed storage are theoretical. Several studies (e.g., [7], [9], [13]) evaluate random linear codes for peer-to-peer storage from a practical perspective, but they are mainly based on simulations without actually deploying a real storage system. RACS [1] and DEPSKY [4] are cloud storage proxies that interconnect multiple cloud storage providers, and they are built on traditional erasure codes. To our knowledge, this paper is the first work that evaluates the empirical performance of regenerating codes using a practical distributed file system.

III. Design and Implementation

A. Architectural Overview

NCFS is designed as a distributed file system that interconnects multiple storage nodes. Figure 2 shows the architecture of NCFS. Our current NCFS implementation does not require storage nodes be programmable to support encoding/decoding functions. Thus, the connected storage nodes can be of various types, so long as each storage node provides the standard interface for reading and writing data. For instance, a storage node could be a regular PC, network-attached storage (NAS) device, or even the repository of a cloud storage provider (e.g., Amazon S3 [3] or Windows Azure [21]). NCFS transparently stripes data across different storage nodes, without requiring the storage nodes to coordinate among themselves during the repair process as assumed in existing theoretical studies (e.g., see [6]). Thus, NCFS can be made compatible with most today's storage frameworks.

NCFS connects to storage nodes over the network (e.g., a local area network or the Internet), while we assume that NCFS is deployed locally as a file system on the client

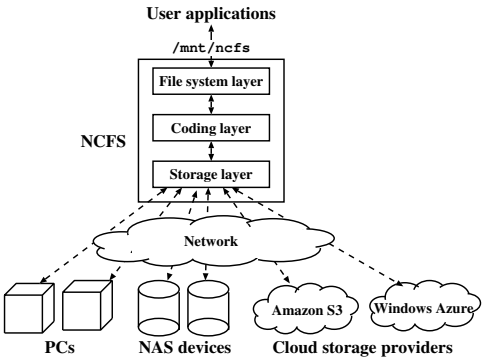


Fig. 2. Architectural overview of NCFS. It is presented as a logical drive to user applications.

machine. Thus, our goal is to improve the performance of read/write operations between NCFS and the storage nodes.

B. Layered Design of NCFS

NCFS adopts a layered design, as shown in Figure 2. The layered design enables *extensibility*, in which each layer can be extended for other functionalities without affecting the logic of other layers. We introduce the layers below and explain how each layer accommodates extensibility.

File system layer. The file system layer is responsible for general file system operations, such as handling the requests of read, write and delete made by users. Each read/write/delete request specifies a data block to be accessed on the storage nodes (see the storage layer below). We also enhance the file system layer to support the data repair operation. That is, if a node is failed, then the repair operation will (i) read data from survival nodes, (ii) regenerate lost data blocks, and (iii) write the regenerated blocks to a new node.

The file system layer is built on FUSE [8], an open-source, user-space framework for implementing file system operations. It mounts different storage nodes as a logical drive (e.g., `/mnt/ncfs`). This allows user applications to access data on the storage nodes through the mounted drive, without worrying about how to organize data in different storage nodes.

Coding layer. The coding layer is responsible for the encoding/decoding functions of fault-tolerant storage schemes based on MDS codes. In the current implementation of NCFS, we implement traditional erasure codes RAID 5 and RAID 6, and regenerating codes E-MBR($k = n-1$) and E-MBR($k = n-2$) (assuming $d = n-1$) (see Section II-B). With the above codes, the current NCFS prototype does not require programmability of storage nodes. On the other hand, if this assumption can be relaxed and storage nodes are programmable (e.g., all storage nodes are regular PCs), then the coding layer can be extended to support other erasure/regenerating codes if necessary. For example, we can implement a class of MSR codes (see Section II-C) in the coding layer as well as the storage nodes, so that we can explore the tradeoffs between the storage cost and repair bandwidth as studied in [5], [22]. Other layers remain unaffected with such extensions.

Storage layer. The storage layer provides a common interface for the file system to access different types of storage nodes. Since the file system organizes data into fixed-size blocks, each block can be uniquely identified by the mapping (`node`, `offset`), where `node` identifies a particular storage node, while `offset` specifies the location of the block within the storage node. The storage layer can then access a data block using the mapping provided by the file system, while the access method is transparent to the file system. For example, the storage layer can access regular PCs or NAS devices over the Ethernet and IP networks via protocols like ATA over Ethernet [10] or iSCSI [17], respectively; it can also access the repositories of different cloud storage providers based on their own semantics.

Extensions. We can also make extensions atop the existing design of NCFS to improve its performance. In current NCFS, each read/write request directly accesses on storage nodes. One extension is to include a *cache layer*, which caches recently accessed blocks in main memory. If the read/write requests preserve data locality, then they can directly access the blocks via memory without accessing the storage nodes. The cache layer can reside between the coding layer and the storage layer (see Figure 2), and it is transparent to the file system layer. We pose the design issue of the cache layer as future work.

IV. Experiments

~~We implement our NCFS prototype in C and deploy it in a Linux machine with Quad-Core 2.4GHz.~~ Using our NCFS prototype, we compare the empirical performance of different storage schemes, including the traditional erasure codes (i.e., RAID-5 and RAID-6) and regenerating codes (i.e., E-MBR($k = n-1$) and E-MBR($k = n-2$)). Note that the overall empirical performance depends on different factors, such as data transmissions over the network, I/O accesses within storage nodes, and block encoding/decoding operations within NCFS.

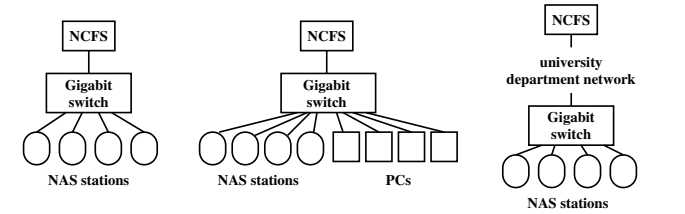
Topologies. We deploy NCFS on an Intel Quad-Core 2.66GHz machine with 4GB RAM and conduct our experiments based on three local area network topologies as shown in Figure 3:

- Figure 3(a) shows the basic setup, in which we interconnect NCFS via a Gigabit switch with four network-attached storage (NAS) stations (i.e., n = 4).
- Figure 3(b) considers a larger-scale setup and studies the scalability of NCFS. NCFS is interconnected with eight storage nodes (i.e., n = 8), including the four NAS stations and four regular PCs, via a Gigabit switch.
- Figure 3(c) considers a relatively more bandwidth-limited network setting, in which NCFS interconnects with the four NAS stations (i.e., n = 4) over a university department network.

In all topologies, NCFS communicates with the storage nodes via the ATA over Ethernet protocol [10].

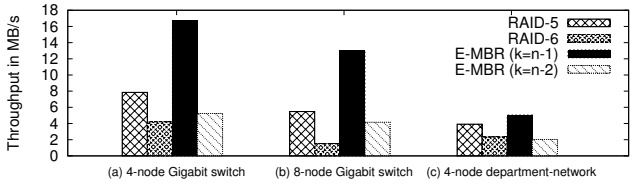
Metrics. We consider the throughput (in MB/s) of different operations: (i) normal upload/download operations with no failure, (ii) degraded download operations with node failures, and (iii) repair operations during node failures. Each throughput measurement is obtained over the average of five runs.

Experiment 1 (Normal upload/download operations). Suppose that there is no node failure. This experiment studies the

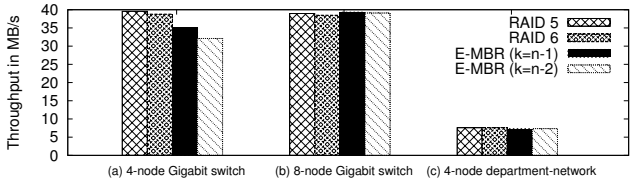


(a) 4-node, Gigabit switch (b) 8-node, Gigabit switch (c) 4-node, deparment network

Fig. 3. Topologies used in our experiments.



(a) Normal upload



(b) Normal download

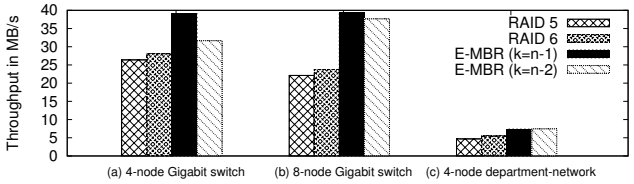
Fig. 4. Experiment 1: Throughput of normal upload/download operations.

throughput of the normal upload/download operations. Here, we upload/download a file of size 256MB to/from the storage nodes. Note that when we upload a 256-MB file, different storage schemes have different actual storage sizes based on how they introduce redundancy (see Table I). For instance, when n = 4, the actual storage sizes of different codes are: 341MB for RAID-5, 512MB for RAID-6, 512MB for E-MBR($k = n - 1$), and 614MB for E-MBR(k = $n - 2$).

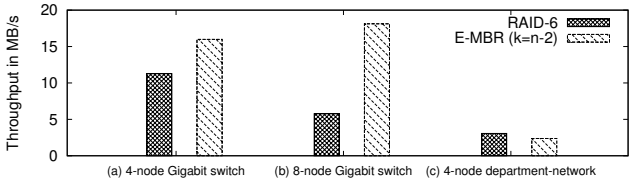
Figure 4(a) shows the upload throughput. E-MBR($k = n - 1$) has the largest throughout among all codes, since it does not need to access any code blocks. For other codes, when NCFS is about to upload a native block, it needs to read and update the corresponding code block(s) of the same segment (see Figure 1 in Section II) on the storage nodes, and this introduces additional read accesses. RAID-5 has the second largest upload throughput as it transmits fewer blocks than RAID-6 and E-MBR($k = n - 2$). E-MBR($k = n - 2$) outperforms RAID-6 in both 4-node and 8-node Gigabit-switch settings, but the difference becomes small in the department network setting. The reason is that RAID-6 uses Reed-Solomon coding to compute the Q-parity code blocks (see Section II), so the computation overhead dominates the transmission overhead when the topology has high network capacity (e.g., a Gigabit-switch setting), but becomes less significant over the more bandwidth-limited setting (e.g., a department network).

It is interesting to see that the upload throughput is smaller in the 8-node Gigabit-switch setting than in the 4-node one. We conjecture that this is related to disk locality of I/O accesses. We plan to analyze the impact of I/O accesses in future work.

Figure 4(b) shows the download throughput. In each topology, all storage schemes have similar download throughput. Note that download operations generally have higher throughput than upload operations, mainly because NCFS can only download one copy of each native block without the need of accessing other code blocks, or duplicate blocks (in E-MBR).



(a) Download throughput during a single-node failure



(b) Download throughput during a two-node failure

Fig. 5. Experiment 2: Degraded download throughput.

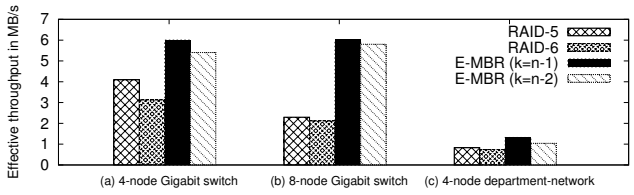
Experiment 2 (Degraded download operations). We now consider the performance of download operations when some storage nodes are failed. In the experiment, we first upload a 256MB file to all storage nodes. We then pick one/two nodes to disable, and then evaluate the throughput of downloading the 256MB file. Here, we pick the leftmost nodes in the array (see Figure 1) to disable, while our observations are similar if we disable other nodes.

Figure 5(a) shows the download throughput during a single-node failure. We observe that the E-MBR codes have higher download throughput than RAID codes. The reason is that for each lost native block, there must be a corresponding duplicate copy (see Figure 1), which could be used for download. On the other hand, RAID codes need to additionally access the corresponding code block of the same segment to recover each lost native block.

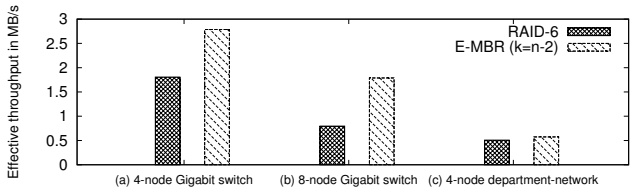
Figure 5(b) shows the download throughput during a two-node failure (for RAID-6 and E-MBR($k = n - 2$) only). E-MBR($k = n - 2$) outperforms RAID-6 in the Gigabit-switch settings, mainly because RAID-6 uses Reed-Solomon coding to recover lost native blocks and incurs higher computation overhead than E-MBR. Using the same reasoning as in Experiment 1, E-MBR($k = n - 2$) has higher throughput than RAID-6 in well-connected settings.

Experiment 3 (Repair operations). Recall in Section III-B that the repair operation of a failed node includes three steps: (i) transmission of the existing blocks from survival nodes to NCFS, (ii) regeneration for lost blocks of the failed node in NCFS, and (iii) transmission of the regenerated blocks from NCFS to a new node. If there is more than one failed node, then we apply the repair operation for each failed node one-by-one. In this experiment, we evaluate the performance of the repair operation (i.e., from step (i) to step (iii)). For the single-node failure case, we consider the throughput of repairing the failed node. For the two-node failure case, we only consider the throughput of repairing the *first failed node*, since after we repair the first failed node, repairing the second failed node is reduced to the single-node failure case.

Note that each segment contains both original native blocks



(a) Repairing a single-node failure



(b) Repairing the first failed node in a two-node failure

Fig. 6. Experiment 3: Repair throughput.

as well as redundant blocks (e.g., code blocks, or duplicate blocks). For fair comparison, we here consider the *effective throughput* of repair that we define as follows. If each segment contains a fraction f (where $0 < f < 1$) of redundant blocks and the time to repair a total of N-MB all lost blocks (including both original native blocks and redundant blocks) of a failed node is Ts, then the effective throughput of repair is defined as $(1-f)N/T$ (in MB/s).

Figure 6(a) shows the repair throughput of a single-node failure. We observe that in the Gigabit-switch settings, E-MBR codes achieve significantly higher repair throughput than RAID codes. For example, the repair throughput of E-MBR($k = n-1$) is 1.91× and 2.61× over that of RAID-5 in 4-node and 8-node Gigabit switch settings, respectively. The main reason is that E-MBR codes retrieve fewer blocks than RAID codes for repair. On the other hand, in the department network setting, the throughput improvement of E-MBR codes over RAID codes becomes less significant. The reason is that the performance bottleneck now lies on the transmission of regenerated blocks from NCFS to the new node. Since E-MBR stores more redundant blocks than RAID codes in a storage node, it needs more time to transmit blocks from NCFS to the new node. This reduces the effective throughput of E-MBR.

Figure 6(b) shows the repair throughput for the first failed node during a two-node failure (for RAID-6 and E-MBR($k = n-2$) only). We make similar observations as in the two-node failure degrade download case (See Figure 5(b)).

Lessons learned. We study the empirical performance of different storage schemes in different network settings. In repair, E-MBR significantly outperforms RAID codes in the Gigabit-switch settings, mainly because it downloads fewer blocks and has lower coding complexity. This conforms to the findings in existing theoretical studies. On the other hand, it is important to mitigate the transmission bottleneck between NCFS and the new storage nodes, which can degrade the repair throughput as shown in the department-network setting.

E-MBR seeks to minimize repair bandwidth with a tradeoff of higher storage overhead. It is interesting to explore other classes of regenerating codes, such as MSR codes (e.g., [19], [20]) that seek to minimize storage overhead, with the relaxed assumption that storage nodes are programmable to support encoding/decoding functions.

V. Conclusions

We present NCFS, a distributed file system that realizes storage schemes based on traditional erasure codes and regenerating codes in practice. NCFS complements existing theoretical studies on network coding for distributed storage from a practical perspective. It adopts a layered design that allows extensibility of new functionalities. We use NCFS to evaluate different storage schemes under real network settings, in terms of the throughput of upload, download, and repair operations. NCFS provides a practical and extensible platform for researchers to evaluate the empirical performance of various storage schemes. We believe that researchers can benefit from NCFS in conducting applied research in network-coding-based distributed storage systems.

The source code of NCFS is published for academic use at: **http://ansrlab.cse.cuhk.edu.hk/software/ncfs**.

References

[1] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: A case for cloud storage diversity. In *Proc. of ACM SOCC*, 2010.
[2] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung. Network information flow. *IEEE Trans. on Information Theory*, 46(4):1204–1216, Jul 2000.
[3] Amazon S3. http://aws.amazon.com/s3, 2010.
[4] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DEPSKY: Dependable and Secure Storage in a Cloud-of-Clouds. In *Proc. of ACM EuroSys*, 2011.
[5] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *IEEE Transactions on Information Theory*, 56(9), Sep 2010.
[6] A. G. Dimakis, K. Ramchandran, Y. Wu, and C. Suh. A survey on network codes for distributed storage. In *arXiv:1004.4438v1 [cs.IT]*, 2010.
[7] A. Duminuco and E. Biersack. A practical study of regenerating codes for peer-to-peer backup systems. In *Proc. of IEEE ICDCS*, 2009.
[8] FUSE. http://fuse.sourceforge.net/, 2010.
[9] C. Gkantsidis and P. Rodriguez. Network coding for large scale content distribution. In *Proc. of INFOCOM*, 2005.
[10] S. Hopkins and B. Coile. AoE (ATA over Ethernet). http://support.coraid.com/documents/AoEr11.txt, Feb 2009.
[11] Y. Hu, Y. Xu, X. Wang, C. Zhan, and P. Li. Cooperative recovery of distributed storage systems from multiple losses with network coding. *IEEE JSAC*, 28(2):268–276, Feb 2010.
[12] Intel. Intelligent RAID6 Theory Overview and Implementation, 2005.
[13] M. Martaló, M. Picone, M. Amoretti, G. Ferrari, and R. Raheli. Randomized Network Coding in Distributed Storage Systems with Layered Overlay. In *Information Theory and Application Workshop*, 2011.
[14] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proc. of ACM SIGMOD*, 1988.
[15] K. V. Rashmi, N. B. Shah, and P. V. Kumar. Optimal exact-regenerating codes for distributed storage at the msr and mbr points via a product-matrix construction. In *arXiv:1005.4178v1 [cs.IT]*, 2010.
[16] K. V. Rashmi, N. B. Shah, P. V. Kumar, and K. Ramchandran. Explicit construction of optimal exact regenerating codes for distributed storage. In *Proc. of Allerton Conference*, 2009.
[17] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet small computer systems interface (iscsi), Apr 2004. RFC 3720.
[18] K. W. Shum. Cooperative regenerating codes for distributed storage systems. In *Proc. of IEEE ICC*, 2011.
[19] C. Suh and K. Ramchandran. Exact-repair mds codes for distributed storage using interference alignment. In *Proc. of IEEE ISIT*, 2010.
[20] C. Suh and K. Ramchandran. Exact regeneration codes for distributed storage repair using interference alignment. In *Proc. of IEEE ISIT*, 2011.
[21] Windows Azure. http://msdn.microsoft.com/en-us/windowsazure/default.aspx, 2010.
[22] Y. Wu, A. G. Dimakis, and K. Ramchandran. Deterministic regenerating codes for distributed storage. In *Allerton Conference on Control, Computing and Communication*, 2007.