A Practical Study of Exact-BASIC Codes at the MSR Point in Distributed Storage Systems

Yumeng Zhang, Hui Li, Tai Zhou, Jun Chen, Hanxu Hou

***Abstract*—Regenerating codes have been proven a class of optimal distributed storage codes in the tradeoff between storage capacity and repair bandwidth. However, existing regenerating codes rely on expensive computations such as finite field multiplication. The high coding complexity makes regenerating codes unsuitable for practical distributed storage systems. BASIC codes, standing for Binary Addition and Shift Implementable Convolutional codes, are proposed to reduce the computational complexity, as well as to keep the benefits of regenerating codes. In this paper, we implement an exact-repair BASIC code at the minimum-storage (MSR) point in a practical distributed storage system and compare it to Cauchy Reed-Solomon (CRS) code atop a cluster testbed with 20 storage nodes. The results show that minimum-storage BASIC code outperforms CRS code in terms of computational complexity and achieves a significant reduction both of repair bandwidth and disk I/O.**

***Index Terms*—distributed storage system, regenerating codes, implementation, experimentation**

I. Introduction

TO provide high storage reliability, large-scale distributed storage systems [3], [4], [5] are transforming from replication to erasure coding techniques, and Reed-Solomon (RS) code [6] is a typical erasure code. RS code divides original file into k blocks, encodes them into n ($n > k$) coded blocks, and stores in n different storage nodes. Such that the original file can be reconstructed from any set of k nodes. We term this property as (n, k) maximum distance separable (MDS) property. When a block is lost, RS code will download k coded blocks from the surviving nodes, reconstruct the entire file, and encode again to obtain the lost block. Term this process as *data recovery*, the amount of data read from disks as *repair disk I/O* and the amount of data transferred over the network as *repair bandwidth*. Data recovery is mainly performed in two cases. One is to *repair* from permanent failures (e.g., disk crash, device replacement, long-term network disruption) where data is permanently lost. The other is to *degradedly read* the temporarily unavailable data during transient failures or before the permanent failures are restored. The reads are degraded as the unavailable data needs to be regenerated form the available data of other surviving nodes. High-performance recovery is necessary in both cases. The repair bandwidth and repair disk I/O of RS code in both cases are k times size of the lost block, which results in a waste of I/O operations and network bandwidth. In large-scale multi-tiered data centers, the background network traffic due to degraded reads and repairs can become prohibitive for massive amounts of data stored.

Regenerating codes apply network coding to storage systems to lower the network bandwidth upon data recovery, while offer the same properties as erasure codes with respect to storage and reliability. As explained in [7], regenerating codes can be parameterized to achieve two extreme points: the *minimum-storage regenerating* (MSR) codes and the *minimum-bandwidth regenerating* (MBR) codes. However, existing regenerating codes rely on complex parameters and expensive coding computational operations, such as finite field multiplication, which make them difficult to understand, parameterize, and limit their applications in practical storage systems [8], [13], [14]. Binary Addition and Shift Implementable Convolutional (BASIC) codes, introduced in [1], can achieve all the advantages of regenerating codes with only addition and shift operations involved in coding process. The data recovery of BASIC codes performs in the use of ZigZag decoding algorithm [9]. An constructional instance of exact- repair BASIC codes at the MSR point is provided in [10].

In this paper, we provide a practical study of exact minimum-storage BASIC (MS-BASIC) codes, and compare it to Cauchy Reed-Solomon (CRS) code [11], which is a class of optimized RS codes widely used in storage systems. We theoretically show that MS-BASIC code provides the same level of reliability and storage overhead as CRS code with much lower coding calculation, repair bandwidth and disk I/O. We implemented MS-BASIC code and CRS code in a practical distributed storage system and experimented on a cluster testbed with up to 20 storage nodes. Our experiments take into account several significant metrics, including computation overhead, repair bandwidth and repair disk I/O. From the results, we find out that minimizing the data read (repair disk I/O) and transferred (repair bandwidth) during data recovery plays a crucial role in improving the overall recovery performance. Our experiments verify that MS-BASIC code conforms to our theoretical findings and outperforms CRS code in terms of computational complexity. More attractively, it achieves recovery throughput up to 2 times in the case of a single failure compared to the

Manuscript received January 8, 2015; revised January 16, 2015. This work is supported by National Basic Research Program of China (973 Program, No.2012CB315904), the National Natural Science Foundation of China (No.NSFC61179028), the Natural Science Foundation of Guangdong (GDNSF, No.S2013020012822), and the Basic Research of Shenzhen (No.JCYJ20130331144502026, No.JCYJ20140417144423192).

Yumeng Zhang, Hui Li, Tai Zhou, Jun Chen, Hanxu Hou are with the Shenzhen Engineering Lab of Converged Networks Technology, Shenzhen Graduate School, Peking University, Shenzhen, Guangdong, 518055, China (email: zhangyumeng06@126.com, huilihuge@163.com, zhoutai_1989@qq.com, chenjun_9003@163.com, houhanxu@163.com).

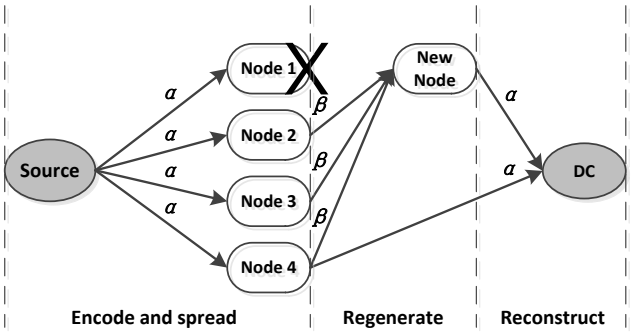


Fig. 1. An illustration of reconstruction and regeneration with $n = 4, k = 2$ and $d = 3$. On failure of node 1, data from nodes 2, 3 and 4 is used for regeneration.

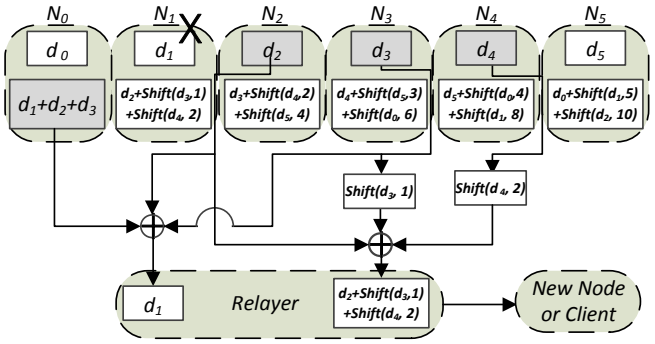


Fig. 2. An illustration of MS-BASIC code with $n = 6$, $k = 3$, and $d = 4$. On failure of node 1, download 4 slices d_2, d_3, d_4 and $d_1 + d_2 + d_3$ from nodes 0, 2, 3 and 4 to regenerate the lost data stored in node 1.

traditional recovery paradigms based on erasure codes.

The rest of the paper is structured as follows. Section II first reviews the background and formulates our code construction. Section III presents our theoretical and analysis findings. Section IV describes the detailed implementation of MS-BASIC code in a practical distributed storage system. Section V shows our experimental results, and Section VI concludes this paper.

II. Background and Code Construction

A. Regenerating Codes

As a generation of erasure-correcting codes, regenerating codes are proposed in [7] to significantly lower the network bandwidth and disk I/O upon data recovery. Consider an $(n, k, d, \alpha, \beta, B)$ regenerating code that the file is divided into B slices, which are encoded into $n\alpha$ coded slices and stored in n storage nodes, where each node stores a group of α slices. As shown in Fig. 1, A *data collector* (DC) connecting to any set of k nodes should be able to reconstruct the original file. We refer to this process as *reconstruction*. Moreover, regenerating codes rely on an additional parameter d referred to as *repair degree*, which is the number of helper nodes involved in data recovery. When a storage node fails, to maintain the same level of redundancy, it will be replaced by a newcomer which downloads β slices each from any d surviving nodes. This process is termed as *regeneration* and the total repair bandwidth is $\gamma = d\beta$. It is shown that regenerating codes lead to an optimal trade-off curve between the amount of data stored and transferred. As explained in [7], regenerating codes can be parameterized by the value B and α to achieve two extreme points: MSR codes and MBR codes, where MSR codes are corresponding to the point with

$$(\alpha_{MSR}, \gamma_{MSR}) = (\frac{B}{k}, \frac{Bd}{k(d-k+1)}). \quad (1)$$

B. Coding Framework

Unlike traditional RS codes and regenerating codes, BASIC codes only involve the *binary additions* and *byte-wise shift operations* to generate coded information, which can reduce the computational complexity by a wide margin [1]. Thus, we define two novel types of functions: *Shift*() and *xor*().

Suppose we have an original data chunk C of size L, containing B source slices, labeled by $d_0, d_1, \cdots, d_{B-1}$, each of which consists of $L' = L/B$ bytes. The structure of d_i can be represented as:

$$d_i \Leftrightarrow \{c_{i1}, c_{i2}, \cdots, c_{iL'}\}, \quad i = 0,1,...,B-1, \quad (2)$$

where c_{ij} is the $(j+1)$-th byte of source slice d_i.

1) We define a $Shift(d_i, l)$ function as

$$Shift(d_i, l) \Leftrightarrow \{\overbrace{\underbrace{0,0,\cdots,0}_{l}, \underbrace{c_{i1}, c_{i2} \cdots c_{iL'}}_{L'}, \underbrace{0,0,\cdots,0}_{r-l}}^{L'+r}\} \quad (3)$$

which shifts all bytes in data slice d_i to right l bytes and returns a new slightly larger slice, with the size of $L' + r$, of which the left l bytes and the right $(r - l)$ bytes are padded with zero and r is the maximal degree of the global encoding coefficients.

2) An $xor(d_0, d_1, \cdots, d_{k-1})$ function can be written in an addition form:

$$xor(d_0, d_1, \cdots, d_{k-1}) \Leftrightarrow d_0 + d_1 + \cdots + d_{k-1} \Leftrightarrow \sum_{i=0}^{k-1} d_i \quad (4)$$

which applies the exclusive or operations in a bit-wise manner among data slices and returns the result in the form of a parity slice.

C. Construction of MS-BASIC Codes

As shown in [10], an (n, k, d) MS-BASIC code is composed of n storage nodes, denoted by $\{N_0, N_1, \cdots, N_{n-1}\}$, satisfying the following two conditions:

$$d = k + 1 \text{ and } B = n. \quad (5)$$

Therefore, by (1), they will also satisfy that $\alpha_{MS} = 2$ and $B_{MS} = n = 2k$. Algorithm 1 presents the encoding process and placement policy of MS-BASIC code.

Algorithm 1 Encoding and Placement

Step1: Fragment a data chunk, with the size of L, into $B = n$ equal-size data slices labeled by d_i $(0 \le i \le n-1)$,with each size of L/n.

Step2: Construct n parity slices p_i $(0 \le i \le n-1)$ by

$$p_i = \sum_{j=(i+1) \bmod n}^{(k+i) \bmod n} Shift(d_j, g_{i,j}),\ i = 0, 1, \cdots, n-1 \quad (6)$$

Step3: Let $s_{i,0} = d_i$ *and* $s_{i,1} = p_i$, then store $(s_{i,0}, s_{i,1})$ as a

ISBN: 978-988-19253-2-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

TABLE I
COMPARISON SUMMARY OF CRS CODE AND MS-BASIC CODE

Scheme	Repair computational complexity	Repair bandwidth	Repair disk I/O	Repair degree
(n, k) CRS code	$O(k^2 w M_{Lost})$	kM_{Lost}	kM_{Lost}	k
(n, k, d) MS-BASIC code	$O(kM_{Lost})$	$(k+1)M_{Lost}/2$	$(k+1)M_{Lost}/2$	$k+1$

strip in the node N_i for $i = 0, 1, \cdots, n-1$. Each node stores $\alpha = 2$ slices.

In Algorithm 1, $g_{i,j}$ stands for the number of bytes shifted to the right of the data slice d_j for the parity slice p_i, which is chosen to be a positive integer and satisfies the increasing difference property, such that the original object can be reconstructed from any k storage nodes by ZigZag decoding method [9]. The coded slices include *data slices* and *parity slices*. In general, the *parity slices* are generated by (6).

When a node fails, the lost data can be regenerated by the following Algorithm 2 in the use of a relayer model, which easily fits into practical distributed storage systems, and has been used in prior studies [3], [4], [5].

Algorithm 2 Regeneration

Step1: The relayer fetches the data slices from the next k nodes in the sequence. Note that the next node of N_{n-1} is N_0. From these data slices, the parity slices of the failed node can be calculated.

Step2: The relayer fetches the parity slice from the previous node in the sequence. Solving an easy system of equations, the lost data slice can be repaired.

Step3: The relayer sends the regenerated data to a newcomer for repairs from the permanent failures or to the client who requests the data for degraded reads.

It is clear that at MS-BASIC code achieves bandwidth optimality for parameter $d = k + 1$, by (5) and (6). Both the repair bandwidth and repair disk I/O are approximately $(k+1)/2$ times size of the lost data. The failed node have to be regenerated by a specific subset of $d = k + 1$ nodes, not any d node.

Note that MS-BASIC code optimizes data regeneration in scenarios when only one storage node is unavailable. If multiple correlative nodes are unavailable, MS-BASIC code performs reconstruction in a manner identical to RS codes. We can check that a DC can reconstruct the original object by downloading data of any k storage nodes. For any k nodes, we can retrieve k data slices and k parity slices, each of which is a linear combination of k data slices. The other k data slices can be retrieved using the ZigZag decoding method because of the corresponding shifting coefficients satisfying the increasing difference property [9].

D. Example

We present an example of the MS-BASIC code with $n = 6$, $k = 3$, and $d = 4$ using Fig. 2. In the construction process, the original object is fragmented into $B = n = 6$ data slices $d_0, d_1, \cdots, d_5$. Then, obtain $n = 6$ parity slices $p_0, p_1, \cdots, p_5$ using (6).

For (6, 3, 4) MS-BASIC code, we can repair one failure node by connecting to 4 nodes and downloading one slice from each helping node. Suppose node N_1 is failed. Our goal is to regenerate its lost data. As shown in Fig. 2, d_1 and $d_2 + Shift(d_3, 1) + Shift(d_4, 2)$ stored in node N_1 are unavailable. A relayer, in which a daemon coordinates the recovery operation, downloads d_2, d_3, d_4 and $d_1 + d_2 + d_3$ from nodes N_0, N_2, N_3 and N_4 respectively, then performs some simple additions and right-shift operations to recover the unavailable data. The parity slice $d_2 + Shift(d_3, 1) + Shift(d_4, 2)$ can be regenerated by adding the data slice d_2, one-shift data slice d_3 and two-shift data slice d_4. The data slice d_1 can be repaired by the following equation:

$$d_1 = (d_1 + d_2 + d_3) + d_2 + d_3. \qquad (7)$$

In this data recovery, the total amount of data read and transferred is $2/3$ size of the original object. The helper nodes need no coding to repair a failure node. The coding operations is only performed in the relayer. Such kind of code is called *repair-by-transfer* code [12].

III. PERFORMANCE ANALYSIS

In this section, we study the recovering performance over single failure by performing an analysis focusing on four key metrics: *repair computational complexity*, *repair bandwidth*, *repair disk I/O* and *repair degree*. CRS code was proposed in [11] to simplify the computational complexity of regular erasure codes, involving only *exclusive or* operations by matrix representation of finite fields. Hence, we select CRS code as a contrast item. The results are shown in Table I.

Suppose that a storage node is failed, the amount of data stored in which is M_{Lost} bytes. Recall that in (n, k, d) MS-BASIC code, each storage node consists of 2 slices, such that the size of each slice is $M_{Lost}/2$ bytes. An (n, k) CRS code have a storage node containing w slices, each size of M_{Lost}/w bytes. Evaluating the repair computational overhead, we compute the amount of binary additions during recovery for a single failure, since the binary addition is the main expense in the coding framework of CRS code and BASIC code. In the repairing process of (n, k, d) MS-BASIC code, both the data slice and the parity slice can be regenerated by summing k slices. Such that, the repairing process needs $2k$ additions of $M_{Lost}/2$ bytes. The repair computational complexity is $O(kM_{Lost})$. Likewise with the conventional recovery of erasure codes, (n, k) CRS code have to first compute the inverse matrix of the encoding

ISBN: 978-988-19253-2-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

matrix, then multiply the inverse matrix by k surviving coded strips to decode the integrated original object [11]. Compared to the time of adding large amounts of data, the time of computing the inverse matrix is too small to be considerable. The multiplication of the inverse matrix and k coded strips takes $kw \times kw$ additions of M_{Lost}/w bytes. So, the repair computational overhead of (n,k) CRS code is $O(k^2 w M_{Lost})$. Both the repair bandwidth and repair disk I/O of (n,k,d) MS-BASIC code are approximately $(k+1)/2$ times size of the lost data, namely $(k+1)M_{Lost}/2$ bytes. Moreover, the repair degree is $d = k+1$. For (n,k) CRS code, both the repair bandwidth and repair disk reads are at least k times of M_{Lost} bytes, and the repair degree $d = k$.

To sum up, MS-BASIC code cuts down the repair computational complexity by several orders of magnitude and provides reasonable overheads, $O(kM_{Lost})$. The theoretical analysis as well as shows that MS-BASIC code makes a reduction of approximately $2\times$ on both repair bandwidth and repair disk I/O compared to CRS code.

IV. Implementation

We complement our theoretical analysis with prototype implementation. As a proof of concept, we implemented MS-BASIC code atop of the Hadoop Distributed File System (HDFS) [2]. We modified the source code of HDFS and augmented several new modules. The relevant modules and the communication flows for relevant operations are depicted in Fig. 3.

A. Integration into HDFS

HDFS stores each file by dividing it into blocks of a certain size. By default, the size of each block is 64MB, and this is also the value that is typically used in practice. In HDFS, 3 replicas of each block are stored in the system by default to achieve data reliability. There are a single NameNode and multiple DataNodes in the cluster. The NameNode manages the metadata for HDFS blocks, while the DataNodes actually store HDFS blocks. To integrate our model of MS-BASIC code into HDFS, we augmented several new modules, including the RelayerNode, BlockFixer, CodedFS and CodeLibrary. We deploys a relayer daemon in RelayerNode and client node for failure recovery and degraded reads, respectively.

On top of HDFS, we adds a new node named RelayerNode, which mainly handles data recovery operation, if needed, recovers the corrupted blocks in order to ensure the reliability of the system. It periodically asks NameNode to check any lost blocks and keeps a list of blocks that are missing and needed to be recovered. The RelayerNode delegates the recovery task to the BlockFixer, which periodically goes through the corrupted blocks list and regenerates the blocks with locally recovery process in small scale or via MapReduce jobs in large scale.

CodedFS, in short of Coded File System, runs above HDFS as a wrapper and handles all read/write requests for coded data stored in HDFS. It creates CodedOutputStreams for writing requests and CodedInputStreams for reading requests. When writing a file to HDFS, CodedFS first performs encoding of the file locally, then spreads the coded data across the cluster by means of CodedOutputStreams. If a corrupted block is requested, then CodedFS opens CodedInputStreams, performs degraded read operations via a locally running decoding process and responds to the read request. Both RelayerNode and CodedFS rely on an underlying component: CodeLibrary, which implements the encoding and decoding functionalities, referred as Encoder and Decoder.

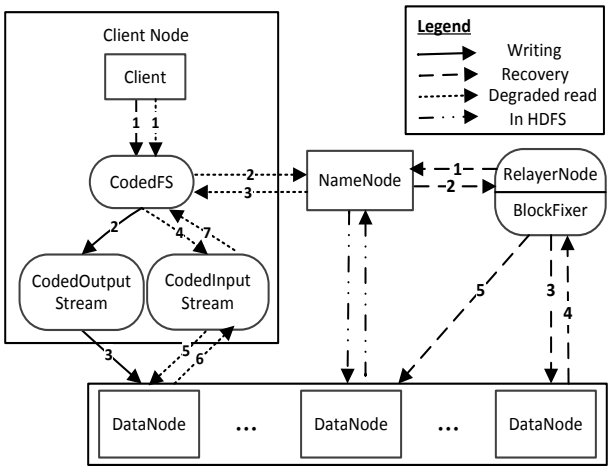


Fig. 3. An overview of our practical storage system. The communication flows for writing, degraded read and recovery operations are shown.

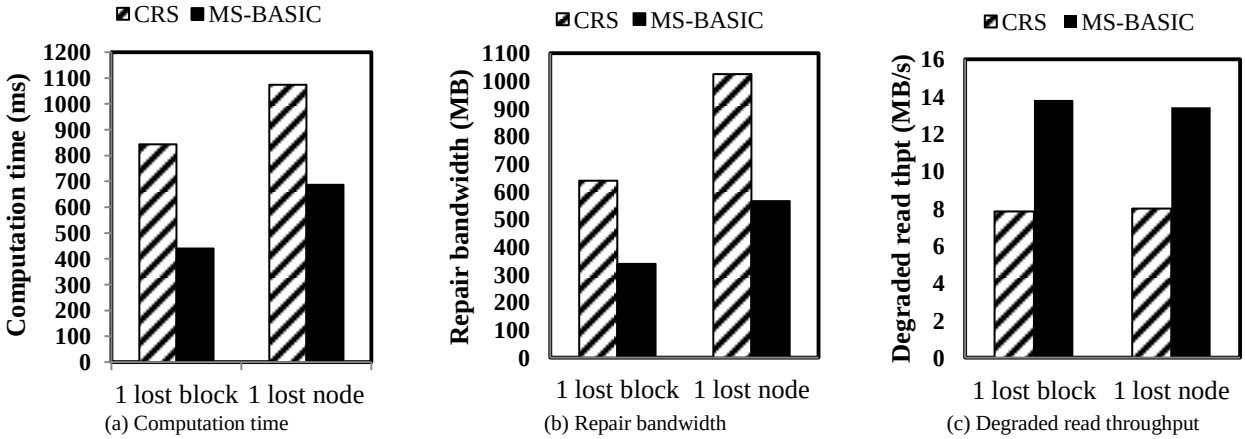
B. Writing

Once the client sends a write request of a file to HDFS, CodedFS launches the Encoder (Step 1). The Encoder initially fragments the file into several data chunks, each size of L. Depended on the file size, the last chunk, which is smaller than L , is considered as "zero-padded" full-chunk as far as the parity calculation is concerned. The Encoder iteratively loads a chunk, divides it into B data slices, then encodes them into $n\alpha$ coded slices and constructs α slices into n strips based on the specific encoding algorithms (see Section II). During each iteration, the Encoder writes n strips into n local temporary files (Step 2). When the size of each file achieves the block size, namely 64MB, CodedFS uses n CodedOutputStreams to upload the local data to HDFS across n different nodes (Step 3).

C. Recovery

The RelayerNode periodically asks NameNode to check any lost blocks and keeps a list of blocks that are missing and needed to be recovered. Once a single failure needs to be recovered, the RelayerNode requests relevant metadata from NameNode and delegates the recovery task to the BlockFixer (Step 1-2). BlockFixer fetches data from d helping nodes (Step 3-4) and regenerates the lost block with locally recovery process in small scale or via MapReduce jobs in large scale according to the specific regenerating algorithms (see Section II). Finally, RelayerNode sends the repaired data to a newcomer to ensure the redundancy of the system (Step 5).

ISBN: 978-988-19253-2-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)



(a) Computation time (b) Repair bandwidth (c) Degraded read throughput

Fig. 4. Measurements from HDFS cluster during a single failure: (a) computation time for regeneration, (b) repair bandwidth, and (c) degraded read throughput.

D. Degraded Read

Once a corrupted block is requested, CodedFS calls Decoder to perform degraded read operation (Step 1). The Decoder inquires the NameNode to get the coding information and the locations of recovering related blocks (Step 2-3), and then opens several CodedInputStreams to read the data required for recovery from HDFS (Step 4-5). Similarly to recovery operation, the block is reconstructed locally based on the specific regenerating algorithm (see Section II), and then responded to the read request (Step 6).

V. Evaluation

In this section, we provide details on our experiments to evaluate the performance of MS-BASIC code and compare it with CRS code in a cluster of one NameNode and 20 DataNodes. Each node runs on a quad-core PC quipped with 4GB RAM, 3.6GHz CPU and 1Gb/s Ethernet card. All nodes are interconnected over a 1Gb/s Ethernet switch and run Linux CentOS5.6. For all evaluations, we consider the encoding parameter ($n = 20$, $k = 10$), a buffer size of 1MB, and a system block size of 64MB, which is default value used in storage systems. Since hadoop is implemented in Java, we newly implement CRS code and MS-BASIC code directly in Java, avoiding the fragile of JNI.

In a system perspective, we consider two levels of failures, *block failures* and *node failures*, since block is the basic unit of storage function in practical system and node is the basic unit of physical devices. Both of them may influence the performance of recovery in a practical system. We measure the *degraded read throughput*, defined as the amount of data being requested divided by the response time. All of our results are averaged over 5 runs.

To evaluate the computational performance of data recovery of CRS code and MS-BASIC code, we measured the time taken for computations during degraded reads and repairs. Fig. 4(a) presents the computation time for regenerating one block failure and one node failure. MS-BASIC code performs much faster regeneration than CRS code. Fig. 4(b) depicts the repair bandwidth during recovering a block and a node respectively. MS-BASIC code significantly reduces the repair bandwidth, the averaged gain is about 2 times compared to CRS code. In fact, the repair disk I/O of MS-BASIC code during a single failure recovery is the same as the repair bandwidth, since the code is repair-by-transfer regenerating code with the minimal I/O cost.

TABLE II
Time Comparisons for Different Recovery Steps in the Two Schemes

Time (s)	Download	Regenerate	Upload
CRS code	13.084	0.82	1.147
MS-BASIC code	7.021	0.784	1.404

Recall form Section II that a recovery operation can be decomposed into three steps. We evaluate the expected performance of each recovery step to identify the bottleneck. Table II illustrates the experiment results of time cost by different recovery steps. We can see that the download step uses the most time among all operations. This justifies the need of minimizing the repair bandwidth to optimize the data recovery performance in distributed storage systems. Fig. 4(c) shows the results of degraded read throughput. Due to the big reduction of download step, MS-BASIC code accelerates the degraded read performance, the rate of which is close to the twice of CRS code.

VI. Conclusion

In this paper, we focus on exploring the feasibility of deploying regenerating codes in a practical distributed storage system. We studied an exact minimum-storage BASIC code, implemented it in a practical distributed storage system and compared it to CRS code atop a cluster testbed with 20 storage nodes. The results demonstrate that MS-BASIC code outperforms CRS code in repairing cost and coding cost and achieves an approximately 2 times reduction both of repair bandwidth and disk I/O. In addition, we find out that minimizing the data transferred (*repair bandwidth*) during data recovery plays a crucial role in improving the overall recovery performance. Owing to the reduction of repair bandwidth, the degraded read throughput of MS-BASIC code is boosted notably.

ISBN: 978-988-19253-2-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

REFERENCES

[1] H. Hou, *et al*, "BASIC Regenerating Code: Binary Addition and Shift for Exact Repair," in *Proc. IEEE ISIT*, 2013.
[2] K. S., *et al*, "The Hadoop Distributed File System," in *MSST ' 10*, 2010.
[3] HDFS-RAID wiki. http://wiki.apache.org/hadoop/HDFS-RAID.
[4] C. Huang, H. Simitci, Y. Xu, *et al*, "Erasure Coding in Windows Azure Storage," in *Proc. of USENIX ATC*, Jun. 2012.
[5] R. Bhagwan, *et al*, "Total recall: System support for automated availability management," in *Symp. Networked Systems Design and Implementation (NSDI)*, 2004.
[6] Reed and Solomon, "Polynomial codes over certain finite fields," in *Journal of the Society for Industrial and Applied Mathematics*, 1960.
[7] A. Dimakis, *et al*, "Network Coding for Distributed Storage Systems," in *IEEE Trans. on Information Theory*, 56(9):4539–4551, Sep. 2010.
[8] Jiekak, *et al*, "Regenerating codes: A system perspective," in *ACM SIGOPS*, 2013.
[9] C. W. Sung and X. Gong, "A Zigzag-Decodable Code with the MDS Property for Distributed Storage Systems," in *Proc. IEEE Int. Symp. Inf. Theory*, Istanbul, July 2013, pp. 341–345
[10] H. Hou, *et al*, "Construction of Exact BASIC codes for Distributed Storage System at the MSR point," in *IEEE Bigdata Workshop*, 2013.
[11] J. S. Plank, "Optimizing Cauchy Reed-Solomon codes for fault-tolerant storage applications," in *Technical Report CS-05-569*, University of Tennessee, 2005.
[12] N. B. Shah, *et al*, "Distributed storage codes with repair-by-transfer and non-achievability of interior points on the storage-bandwidth tradeoff," in *IEEE Trans. on Information Theory*, vol. 58, no. 3, pp. 1837–1852, Mar. 2012.
[13] Y. Hu, Henry C. H. Chen, *et al*. NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-Clouds. In *FAST '12*, 2012.
[14] Y. Hu, *et al*, "NCFS: On the Practicality and Extensibility of a Network-Coding-Based Distributed File System," in *Proc. of NetCod*, 2011.

ISBN: 978-988-19253-2-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)