An Efficient Fault Tolerance Framework for Distributed In-memory Caching Systems

Shuaibing Zhao, Lu Shen, Yusen Li, Rebecca J. Stones, Gang Wang*, Xiaoguang Liu*
Nankai-Baidu Joint Lab, College of Computer Science,
Nankai University, Tianjin, China
Email:{zhaoshb,shenlu,liyusen,rebecca.stones82,wgzwp,liuxg}@nbjl.nankai.edu.cn

***Abstract*—With the development of the information age, many large database applications have introduced distributed in-memory object caching systems, of which Memcached is one of the most typical. However, Memcached does not have fault-tolerant capabilities. In order to make Memcached enable fault tolerance, Cocytus introduced Reed-Solomon codes and distributed protocols into Memcached. Cocytus saves significant memory compared to primary-backup replication when tolerating the same number of failures. However, the relatively complex finite-field calculations used by RS codes and the high network transmission cost during data reconstruction are becoming new bottlenecks.**

This paper introduces RDP codes into distributed Memcached to optimize the calculation performance in Cocytus. In addition, this paper adopts RDOR scheme and Collective Reconstruction Read to speed up the data reconstruction. Compared with Cocytus, which uses RS codes for fault tolerance, the new distributed Memcached with 4 data nodes and 2 check parity nodes reduces reconstruction overhead by up to 31%.

***Index Terms*—Memcached, erasure code, optimal recovery, parallel recovery**

I. Introduction

Relational database management systems struggle to keep pace with modern increases in performance requirements [1] due to limitations of their storage structures [2]. In order to address this issue, some distributed in-memory caching systems have been proposed, such as Memcached [3] and Redis [4]. They put the most frequently accessed data in memory so that user requests are processed without disk operations, thereby improving overall performance. Distributed in-memory caching systems are consequently widely used in large Internet companies such as Facebook [5] and Twitter [6].

In distributed in-memory caching systems, when server nodes are crashed(or are temporarily unavailable), it is desirable to have a mechanism to recover the data on erased nodes via the non-erased nodes. Otherwise, if the lost data is reloaded from the disk, the long loading time hurts the system performance. A traditional approach for fault tolerance is through primary-backup replication (PBR) [7]. In this approach, each primary node has some backup nodes to store the data replicas for fault tolerance. When a primary node crashes, one of the backup nodes acts as the new primary node. Although this approach provides continuous services in the presence of node failures, the data redundancy is high.

Erasure codes [8] (such as the well-known Reed-Solomon codes [9]) have been proven very efficient in providing higher levels of fault-tolerance with less cost, and are widely used in today's distributed systems [10]–[12]. In erasure-coded distributed systems, server nodes are classified into *data nodes* (where the raw data is stored) and *check nodes* (where the parity check data is stored). The parity check data is computed using the raw data. Data nodes and check nodes are organized into *coding groups*. The raw data and parity check data are both divided into equal-sized data units. If some data units (either raw data units or parity check data units) in a coding group are lost due to node failures, the lost data units can be recovered using other data units belonging to a common coding group.

Based on Reed-Solomon codes (RS codes) [9], [13], Zhang et al. [14] proposed Cocytus, which applied erasure codes to distributed in-memory caching systems for the first time. For a RS coding system with k data nodes and $n-k$ check nodes (normally denoted by RS (n,k)), at most $n-k$ node failures can be handled [13]. However, in order to recover one data unit in a RS (n,k) coding system, k units of data from different nodes need to be fetched to perform the finite-field computation. The overheads in both computation and data transmission are huge, which inhibit the throughput of the system.

In this paper, we propose a efficient fault-tolerance framework for a distributed in-memory caching systems that utilizes:

1) Row-Diagonal Parity (RDP) codes [15]. RDP codes only use XOR operations for computing parity check units and during recovery, which are faster than finite-field operations in RS codes.
2) Row-Diagonal Optimal Recovery (RDOR) scheme [16]. The amount of data required for single failure recovery using RDOR is less than that using naive recovery schemes for RDP codes and RS codes.
3) Collective Reconstruction Read (CRR) decoding scheme [17] which carries out the recovery process in a distributed and parallel manner.

We design encoding and decoding protocols for the proposed fault-tolerance framework. We implement this as a modification of Cocytus [14], which is implemented on the Memcached [18] caching system.

The structure of this paper is as follows. Section 2 reviews the related work of this paper. Section 3 introduces RDP codes. Section 4 describes the overall design of our Memcached system, which includes data updating and data recovery. Section

5 presents the experimental evaluations. Finally, Section 6 summarizes this paper and describes a future research idea.

II. Related work

Replication and erasure coding are the two most commonly used approaches for fault tolerance in distributed systems. Replication, where multiple copies (or *replicas*) of each data unit are stored, is a simple approach which is widely used in distributed file systems [19], [20]. When data becomes unavailable, it is accessed through its replicas. However, replication has a large space overhead, requiring $M+1$ copies of all the data to tolerate M failures.

Memcached [18], [21] is a high-performance, distributed memory object caching system, which stores user data in memory so that cached read operations are processed without database access which reduces the system load. To provide real-time, precise and high-throughtput services, an increasing number of large database applications adopt in-memory caching systems to store data [18], [22]–[24]. Relavant to this paper is Cocytus by Zhang et al. [14], who applied RS codes to in-memory caching systems for the first time.

Memagent [25] is a simple but useful proxy program, giving Memcached replication-based fault tolerance. If one data server is down, data can be obtained from the corresponding backup server.

Erasure coding is an efficient approach for redundancy reduction for fault tolerance, which has been adopted in many large scale distributed systems, e.g., Windows Azure Storage [26]. Reed-Solomon (RS) codes [9], [13] are the most frequently used erasure codes in the storage systems. For example, Facebook uses RS codes in their Hadoop Distributed File System to improve the storage efficiency [27]. However, RS codes are based on finite field, so encoding and decoding process need complex computations. Even some studies have been performed to optimize the computations of RS codes [28], [29], the overhead is still high.

Row-Diagonal Parity (RDP) [15] code is a more efficient erasure code compared to RS code. RDP code is a double-erasure correcting array code which uses only exclusive-or operations to compute parity and the decoding process is also much faster than RS code. The Row-Diagonal Optimal Recovery (RDOR) [16] scheme is an optimization scheme for data recovery in RDP single-node failures by reducing the number of blocks read in the reconstruction process. Collective Reconstruction Read (CRR) [17] instead optimizes the reconstruction process by parallelizing the reading of data. The fault-tolerance framework proposed in this paper is based on RDP codes, and we discuss this in detail in Section III.

III. RDP Codes

In this paper, we adopt RDP codes instead of RS codes for fault tolerance to improve the computing efficiency of Memcached system.

An RDP array is defined by a controlling parameter p, which must be a prime number greater than 2 (but this can be relaxed by deleting some nodes and imagining they contain only zeros). In the RDP array, there are $p+1$ nodes in total, of which $p-1$ data nodes which store raw data units, and 2 check nodes which store the parity check units. The data units on each node are divided into $p-1$ equal-sized blocks. One check node is called the *row check node*, which stores the XOR results of the raw data blocks in the same row. The other check node is called the *diagonal check node*, which stores the XOR results of the diagonal data blocks in the same diagonal.

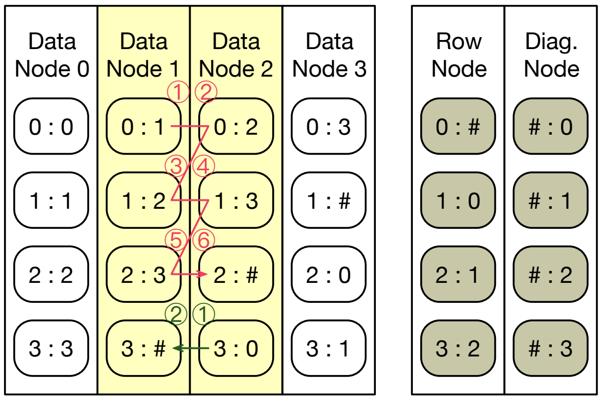


Fig. 1. The RDP coding system with $p=5$. Blocks marked with numbers indicate how the check node are defined. For example, the row check node (abbreviated "Row Node") block marked 0:# is the XOR of the four data blocks marked 0:0, 0:1, 0:2, and 0:3, and the diagonal check node (abbreviated "Diag. Node") block marked #:1 is the XOR of the four blocks marked 1:1, 0:1, 3:1, and 2:1. The numbers in circles indicate the order in which blocks should be recovered when Data Node 1 and Data Node 2 be erased (odd numbers use the diagonal check, while even numbers use the row check).

Figure 1 shows an RDP array with $p=5$, i.e., there are 4 data nodes and 2 check nodes in the array. The data units in each node are divided into 4 equal-sized blocks. Each block belongs to one row check group and one diagonal check group. In this figure, the two numbers marked on each data blocks indicate which row and diagonal that the block belongs to. For example, 1:2 indicates that the block belongs to row 1 and diagonal 2. Each block in the row check node is the XOR result of the raw data blocks in that row (not including the diagonal check block). Each block in the diagonal check node is the XOR result of the raw data blocks as well as the row check blocks in the same diagonal. Note that one of the diagonal check groups is ignored in the computation [15]. The symbol *"#"* indicates a block which is ignored, which means it doesn't participate in the computation of the check group.

The data recovery process of a single node error in RDP codes is straightforward: if one data node has crashed, each data block in the failed data node can be recovered as the XOR of the data blocks in the remaining data nodes and row check node. Figure 1 shows an example of the data recovery process with two node failures. In this example, data node 1 and 2 crash. As diagonal 1 and diagonal 0 only miss one block each (block 0:1 and block 3:0 respectively), we first recover these two missing blocks as we handle single-node failure. Then, the recovery process alternately uses row check and diagonal check to recover the remaining missing blocks [15].

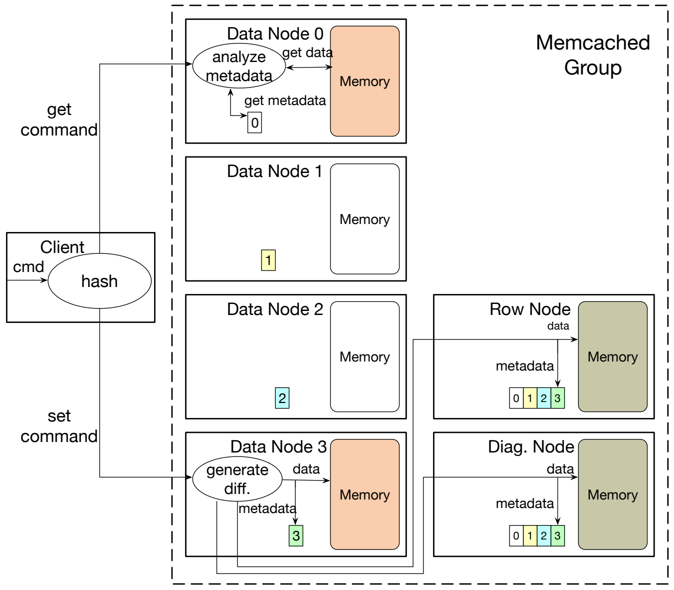


Fig. 2. An coding group in Memcached with RDP codes

Specifically, block 0:2 is recovered according to row 0, then block 1:2 is recovered according to diagonal 2 and so on.

IV. System Design

A. System Architecture

In this section, we present the detailed design of the fault tolerance framework based on RDP codes. Figure 2 shows the overall architecture of the framework with $p = 5$. The Memcached servers are divided into data nodes and parity check nodes, which store raw data and parity check in their memory respectively. The memory space of each node is divided into several equal-sized *units* (4KB by default), which are numbered sequentially starting from 0. The memory units with the same number form a coding group.

We assume that *get(key)* and *set(key, value)* are the two basic operations in the in-memory caching systems. A *get* operation does not modify any data, and thus only the relevant data node manages the request. A *set* operation updates data on the associated data node and the two check nodes. Moreover, similar to Cocytus, we also assume the metadata are stored by primary-backup replication (PBR) [7]. Each data node saves their own metadata and the check nodes save the metadata for all the data nodes in the same coding group.

B. Data Update

In the proposed fault-tolerance framework, a data update is processed as follows. The data node first issues the update locally, then the data node calculates the new row check and the new diagonal check, and sends them to the row check node and diagonal check node, respectively.

In order to minimize data transmission, instead of sending the entire new check, we just transmit the XOR delta between the new check and the original check which, i.e., differential coding [30]. We give a toy example to show how this approach works.

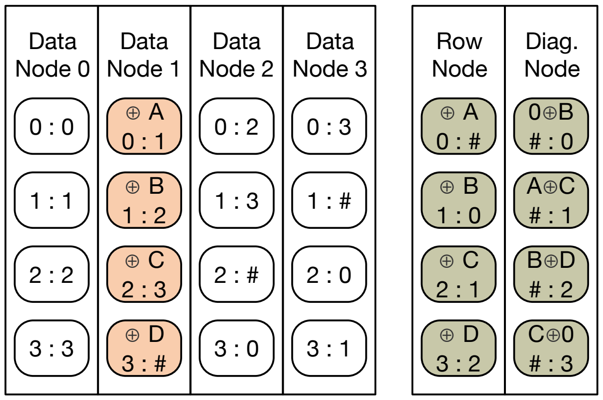


Fig. 3. The RDP differential encoding system with $p = 5$

Figure 3 shows an RDP array with $p = 5$ (i.e., there are 4 data nodes and 2 check nodes). Suppose the data blocks d_i (with $i \in \{0, 1, 2, 3\}$) in data node 1 are to be updated (the process is similar for the other nodes). The raw data units in data node 1 are changed according to

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} \longmapsto \begin{bmatrix} d_0 \oplus A \\ d_1 \oplus B \\ d_2 \oplus C \\ d_3 \oplus D \end{bmatrix}$$

where A, B, C, D determine the change. As per the RDP code, we also update the row node units R_i:

$$\begin{bmatrix} R_0 \\ R_1 \\ R_2 \\ R_3 \end{bmatrix} \longmapsto \begin{bmatrix} R_0 \oplus A \\ R_1 \oplus B \\ R_2 \oplus C \\ R_3 \oplus D \end{bmatrix}.$$

We next observe that the delta between the new diagonal check and the original diagonal check is the XOR result of Δ_{node1} and Δ_{row} after a rotate operation. We can update the diagonal check blocks D_i:

$$\begin{bmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \end{bmatrix} \longmapsto \begin{bmatrix} D_0 \oplus \mathbf{0} \oplus B \\ D_1 \oplus A \oplus C \\ D_2 \oplus B \oplus D \\ D_3 \oplus C \oplus \mathbf{0} \end{bmatrix}$$

where $\mathbf{0}$ denotes the all-0 unit.

Based on the above observations, both delta vectors, which need to be sent to the row check node and diagonal check node, can be calculated at the data node. Once the delta vectors are received by the check nodes, the parity checks can be updated accordingly.

The main challenge when updating data is maintaining data consistency in the presence of possible node failures. When updating a data node, we also update the row check node and the diagonal check node. So if a failure occurs during this process, we may lose consistency. To this end, we use similar approach to Cocytus.

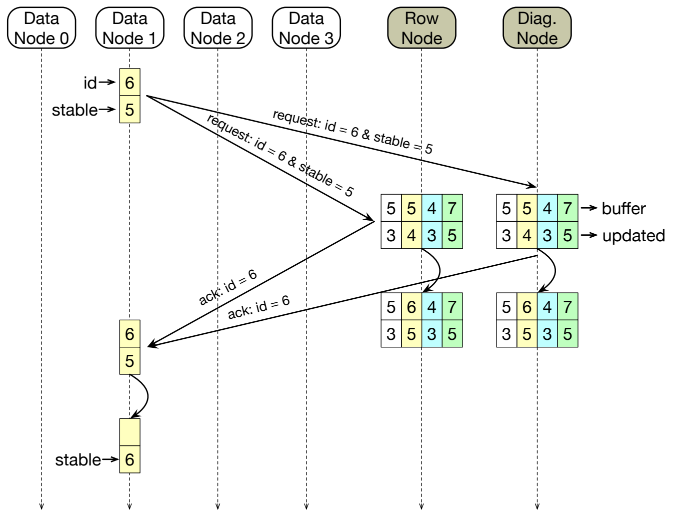


Fig. 4. Piggyback update of distributed Memcached with RDP code

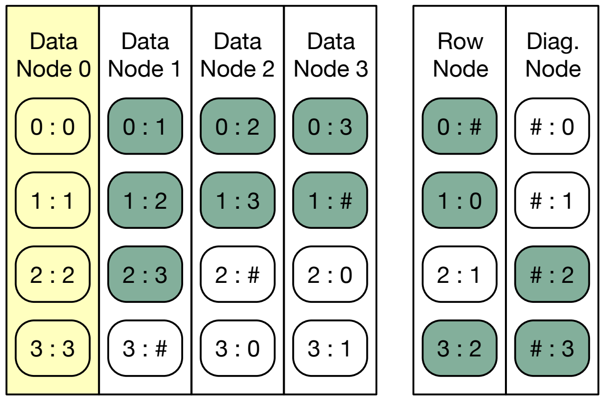


Fig. 5. RDOR decoding scheme when data node 0 crashs (the data node 0 can be recovered according to only 12 data units which are marked green)

Figure 4 shows how the data is updated and how the consistency is maintained. Specifically, we assign each *set(key, value)* operation an *id*, which monotonously increases at each data node like a logical clock. Upon receiving a parity check update from a data node, the check node first puts the operation in a buffer and then immediately sends an acknowledgement to the data node. After the data node receives the acknowledgements from all the check nodes, the operation is marked *stable* and the data node performs the write operation (i.e., update the data) locally. We regard the *id* of the latest stable operation as the *version number* of the data node. When the data node sends the next parity check update, the version number is piggybacked to the check nodes. When the latest version number is received by a check node, it sequentially processes all the operations in the buffer whose *id* is smaller than the latest version number. Once a failure occurs, the corresponding operations that are not received by all check nodes are discarded. In this manner, the consistency in the recovery process is guaranteed.

We next illustrate how data are recovered in the proposed framework after node failure(s). In the standard decoding process for RDP codes, in order to recover one data unit, the other data units and the parity checks in the same coding group are fetched from the corresponding nodes, which produces comparable data-transmission overhead as RS codes.

However, different from RS codes, one data unit in RDP codes belongs to two coding groups (one row coding group and one diagonal coding group). RDOR scheme is a recovery optimization model of RDP codes with a single node failure. The idea of RDOR decoding scheme is to reduce data transmission in the recovery. In contrast, the CRR decoding scheme tries to reduce the recovery time by transmitting and computing data in a distributed and parallel manner. Based on these techniques, we design two data recovery framework to improve the reconstruction efficiency.

C. Data Recovery

1) RDOR-based Data Recovery: RDP codes tolerate (at most) two node failures. However, the probability of two node failures is much smaller than that of single-node failure in practice [31]. Therefore, the optimization of decoding a single node can significantly improve the efficiency of the Memcached system. Based on this fact, we design a data recovery approach for the proposed fault tolerance framework utilizing the RDOR decoding scheme, which incurs far fewer data transmissions during reconstruction.

We first use an example to illustrate how RDOR decoding works. Figure 5 illustrates an RDP array with $p = 5$, where we suppose node 0 crashes. In order to recover all the data blocks in node 0, the standard decoding process for RDP codes fetches all the data blocks of all the surviving data nodes and the row check node (16 blocks in total). However, in RDOR decoding, it only fetches 12 data blocks (the data blocks colored green in Figure 5). The data block marked 0:0 is recovered using the data blocks marked 0:1, 0:2, 0:3, and 0:#, the data block marked 1:1 is recovered using the data blocks marked 1:2, 1:3, 1:#, and 1:0, the data block marked 2:2 is recovered using the data blocks marked 1:2, 0:2, 3:2, and #:2 of the diagonal node, and the data block marked 3:3 is recovered using the data blocks marked 2:3, 1:3, 0:3, and #:3 of the diagonal node.

Xiang et al. [16] proved that when reconstructing any data node using RDP coding, at least $\frac{3}{4}(p-1)^2$ blocks need to be read, and any recovery combination which consists of $\frac{1}{2}(p-1)$ row check sets and $\frac{1}{2}(p-1)$ diagonal check sets will achieve this minimum. Notice that data blocks i:# can only be recovered using the i-th row set. So RDOR first chooses $\frac{1}{2}(p-1)$ blocks (data i:j where $j \neq \#$) to be recovered according to the diagonal check sets (2:2 and 3:3 in this example) which uses $\frac{1}{2}(p-1)^2$ blocks from the remaining nodes, and then chooses the other $\frac{1}{2}(p-1)$ blocks to be recovered according to the row check sets (0:0 and 1:1 in this example) which uses $\frac{1}{2}(p-1)^2$ blocks from the remaining nodes. However, the former $\frac{1}{2}(p-1)^2$ blocks and the latter $\frac{1}{2}(p-1)^2$ blocks overlap, so RDOR only needs $\frac{3}{4}(p-1)^2$

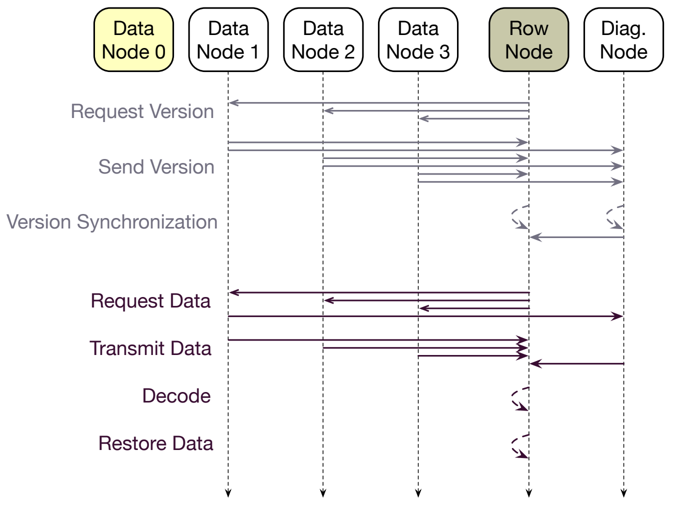


Fig. 6. Data recovery timeline when Data Node 0 is crashed under RDOR decoding architecture

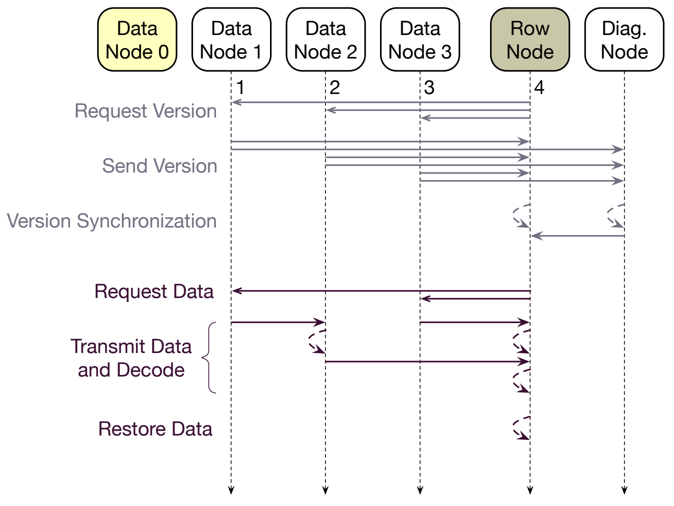


Fig. 7. Data recovery timeline when Data Node 0 is crashed under CRR decoding architecture

to reconstruct a single node. Thus, reconstruction of a single node using the RDOR scheme only requires fetching $\frac{3}{4}(p-1)^2$ blocks, which is 25% fewer blocks than RDP codes($(p-1)^2$).

We now present the RDOR-based data recovery process in our framework. Once a node failure is detected, a check node will be selected as the *recovery leader* to reconstruct the missing data units and one check node will be selected as the *substitution node* to temporarily take over the crashed data node (the recovery leader and the substitution node are usually the same check node when there is only one node failure). Recall that we maintain a version number for each data node to guarantee the consistency. At first, the recovery leader synchronizes the version number to the stable version number of the crashed data node. During the recovery, the missing data is recovered unit by unit. The default size of the unit is 4KB in our practical implementation. When recovering a data unit, the recovery leader fetches the data units of the same *unit_id* from the remote nodes in the same group.

Figure 6 depicts the work flow of the data recovery process for an RDP array with $p = 5$ when data node 0 is crashed.

1) A check node, either the row check node or diagonal node, is selected as the recovery leader. The recovery leader sends a *version synchronization request* to the non-erased data nodes.
2) Upon receiving a version synchronization request from the recovery leader, a data node sends its latest version number to both check nodes.
3) When a check node receives the latest version number from a data node, it performs the data update operations (which are stored in the buffer) whose $unit\_id$ is not higher than the latest version number. Once a non-leader check node receives the latest version number from all the data nodes, it then sends the synchronization complete message to the recovery leader.
4) When the recovery leader receives the synchronization complete message from all the check nodes, the version synchronization procedure terminates. After that, the recovery leader sends a *data request* to each involved node to fetch the data required in the following recovery.
5) When a node receives the request from the recovery leader, it sends the desired data to the recovery leader.
6) When the recovery leader receives all the desired data, it recovers the data and sends the recovered data to the substitution node.

Basically, the data recovery process splits into two phases: the version synchronization phase(the first three steps) and the data recovery phase(the last three steps).

Notice that in the transmission process, the row check node acts as the recovery leader and the substitution node, and doesn't need to transmit the decoded results to others. Therefore, the actual number of blocks transmitted using RDP codes is $(p-1)(p-2)$. And the actual minimum transmission blocks of RDOR scheme is $\frac{3}{4}(p-1)^2 - p + 2$, which is at least 25% fewer data transmission than for RDP codes.

2) CRR-based Data Recovery: Traditionally, data units are sent to the node performing the recovery process one by one in a serial manner, and after all the required data units are received, the recovery process computes the missing data using the received data. Since the network bandwidth is limited at the node where the recovery process takes place, the bandwidth may become the performance bottleneck. In order to address this issue, Li et al. [17] proposed the Collective Reconstruction Read (CRR) decoding scheme, which carries out the recovery process in a distributed and parallel manner.

We use an example to illustrate how CRR decoding works with RDP codes. As illustrated in Figure 7, there are 4 nodes involved in the reconstruction process when data node 0 crashes in RDP_CRR decoding with $p = 5$, which are labeled from 1 to 4. In this example node 4 fetches data from the other non-crashed data nodes for recovery. CRR decoding runs for multiple *rounds*. and the operations in the same round are performed in parallel. In the first round, each node i such that $i \equiv 1 \pmod 2$ (i.e., nodes 1 and 3) sends the data to its next node $i+1$ (i.e., nodes 2 and 4). After the data is received, each node $i \equiv 0 \pmod 2$ computes and stores a partial result (i.e, the XOR of its local data and the received data). In the second round, half of the remaining nodes, those indexed $i \equiv 2 \pmod 4$ (i.e., node 2), sends the partial result to the next remaining node $i+2$ (i.e., node 4). In general, this process repeats recursively until the final result are computed by the node leading the recovery (i.e., node 4 in this example).

In the example in Figure 7, the decoding process takes two rounds. More generally, for a coding group with k nodes, in the first round $\lfloor k/2 \rfloor$ nodes transmit data, which is repeatedly halved in each subsequent round. Therefore, the total number of rounds is at most $\lceil \log_2 k \rceil$. Since in each round a node transmits k blocks (raw data blocks or partial result), the total data transmission time is $\Theta(k \lceil \log_2 k \rceil)$. Note that serial transmission requires $k-1$ rounds to transmit all the data, so the total data transmission time is $\Theta(k(k-1))$, indicating that CRR decoding is theoretically much faster than the standard decoding of RDP codes for a large number of coding groups k. However, even for small k the difference is still meaningful (e.g. when $k=4$, which arises in the $p=5$ case, we have $k \lceil \log_2 k \rceil = 8$ vs. $k(k-1) = 12$).

Based on the CRR decoding scheme, we propose a parallel data recovery approach for our framework. Consider an RDP array with parameter p (i.e., there are $p-1$ data nodes and 2 parity nodes). Suppose one data node has crashed and the row check node is selected to be the recovery leader. According to the standard decoding of RDP codes, the recovery leader fetches data from the $p-2$ non-crashed data nodes. The entire recovery process comprises two steps: In the first step, the recovery process synchronizes the version number. The version synchronization process is the same as that based on the RDOR decoding. After the version synchronization, in the second step, the recovery process computes the to-be-recovered data according to CRR decoding scheme. Figure 7 illustrates the work flow of the data recovery process for an RDP array with $p=5$ where Data Node 0 has crashed.

V. Evaluation

In our experiments, we focus on updating data and single-node failure recovery. For each experiment, we compare our framework with Cocytus. RDP codes with $p=5$ have 4 data nodes and 2 check nodes in our framework, so we compare to $\mathrm{RS}(6,4)$ in Cocytus, which also has 4 data nodes and 2 check nodes. Similarly, we compare RDP with $p=7$ with $\mathrm{RS}(8,6)$ in Cocytus. We just evaluate two scenarios with $p=5$ and $p=7$ because if the stripe is too long, the reconstruction time will increase significantly and the system performance will be reduced. Moreover, the two scenarios with $p=5$ and $p=7$ have met the reliability needs of the distributed in-memory caching systems. In the experiments, the unmodified Cocytus measurements are labeled “RS codes”. The measurements for the RDP codes are labeled “RDP codes”. The measurements for the RDOR method are labeled “RDOR” and the measurements for the CRR method are labeled “RDP_CRR”.

We implement the proposed framework in Memcached. The system is built on a small cluster consisting of 9 servers. Each server in the cluster is configured with four Intel Xeon CPU cores, six 7.2K RPM disks (each has a size of 320 GB) and 2GB memory.

We use iproute2 [32] to control the network bandwidth among servers. In the experiments, we simulate the scenario where the data recovery process can only use a part of network bandwidth capacity because the system resources must be used to guarantee QoS as a priority. We simulate three types of network environments, which are low available bandwidth (100 Mbps), middle available bandwidth (500 Mbps) and high available bandwidth (1Gbps).

We use the YCSB benchmark [33] to generate workloads for Memcached. We vary the item size from 4KB to 64KB. We generate 2000 items as the input for each experiment.

A. Generation overhead

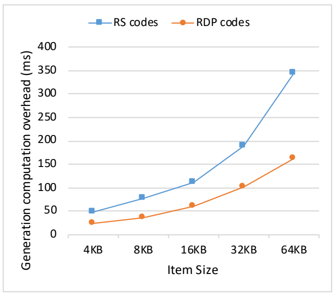
In the experiments in this section, we begin with an empty database, then one-by-one we add 2000 items generated by the YCSB benchmark of some specified item size (which we vary). Thus, the total raw data we generate is around 2000 times the item size. Data is updated using the method described in Section IV-B, by both Cocytus (i.e., “RS codes”) and RDP codes.

Figure 8 plots the *generation computation overhead* of Memcached, i.e., the computation time required to perform the 2000 updates. Figure 9 plots the *generation overhead* of Memcached, which includes all contributions to the generation overhead, including the data transmission overhead, the generation computation overhead (as in Figure 8) and so on. We observe that RDP codes consistently incur significantly less update computation overhead than RS codes (Figure 8), but this saving is swamped by the transmission overhead (Figure 9).

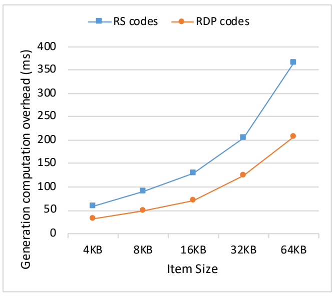
Generally speaking, the RDP code based on XOR encodes much faster than the RS code whose calculation is based on finite fields. However, both of the new Memcached and Cocytus use differential coding to set items, which drastically reduces computation time. And they both use Jerasure libary to process parallel computing [34]. So the encoding time of RDP codes is about half of that for RS codes. Moreover, Figure 8 and Figure 9 highlight how the generation computation time is only a small part of the generation time (about 2%).

B. Decoding

In this paper, RS codes used by Cocytus are replaced by RDP codes to make the distributed Memcached more efficient. Then we adopt two schemes, RDOR scheme and CRR decoding scheme, to reduce the reconstruction time of RDP code. The following uses profiling data to quantify the effect of these optimizations.

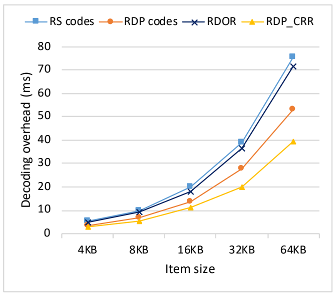


(a) $p = 5$

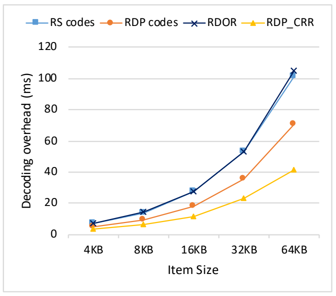


(b) $p = 7$

Fig. 8. Memcached generation computation overhead as item size varies

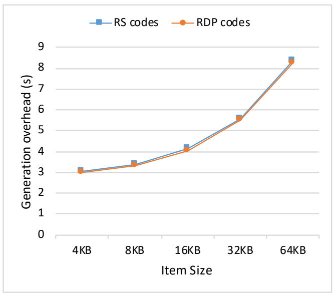


(a) $p = 5$

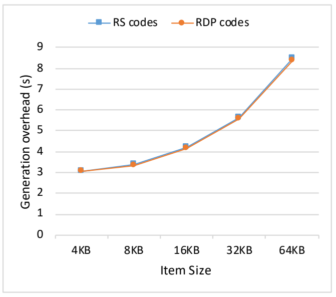


(b) $p = 7$

Fig. 10. Memcached decoding overhead for reconstructing a crashed node

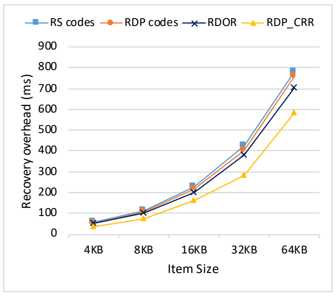


(a) $p = 5$

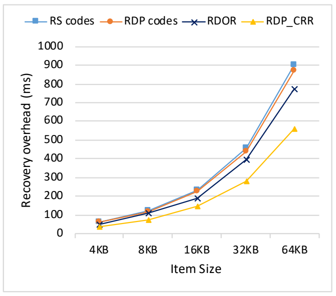


(b) $p = 7$

Fig. 9. Memcached generation overhead as item size varies



(a) $p = 5$



(b) $p = 7$

Fig. 11. Memcached recovery overhead when a node crashes

In this section, we experiment on the data generated in Section V-A. We simulate the erasure of one single node and its recovery process.

Figure 10 plots the *decoding overhead* of Memcached for the erasure of one data node, i.e., the time spent on computation during recovery. Figure 11 plots the *recovery overhead* of Memcached for the erasure of one data node, which includes the version synchronization overhead, the data transmission overhead, the decoding overhead (as in Figure 10), and so on. Figure 12 plots the *throttled recovery overhead* of Memcached for the erasure of one data node, where we throttle the bandwidth: we simulate three bandwidth available for recovery, 1 Gb/s (high), 500 Mb/s (middle), and 100 Mb/s (low).

We observe that the decoding time of RS codes is about 45% more than that for RDP codes (Figure 10). The RDOR method reduces the concurrency of decoding due to the reuse of data blocks which leads to the decoding efficiency similar to the RS codes. Moreover, CRR spreads the decoding operation to different nodes which reduces about 30% decoding time compared with RDP codes. Note that the observed decoding overhead takes up less than 10% of the recovery overhead (Figure 11).

From Figure 11, we observe that the recovery overhead for RDP codes is similar to that of RS codes (and Figure 12 indicates this remains true when the bandwidth is throttled). We also observe from Figure 11 that RDOR reduces the recovery overhead by around 8% to 14% compared to RS codes, while CRR reduces the recovery overhead by 31% to 36% compared to RS codes, but when the bandwidth is throttled, we observe from Figure 12 that RDOR reduces the recovery overhead by around 25% to 35% compared to RS codes. RDOR outperforms the other approaches with low network bandwidth, where the overhead of data transmission is larger, which we attribute to RDOR being designed to reduce the total number of reads (thereby reducing transmission size).

Figure 11 indicates that the proposed RDP_CRR is the overall best performer in terms of single-node recovery, but Figure 12 indicates that RDOR becomes the best performer with low and medium available network bandwidth.

It has been proved that under the same system configuration, the mean time to data loss (MTTDL) is inversely proportional to the failure repair time (MTTR) squared [35].

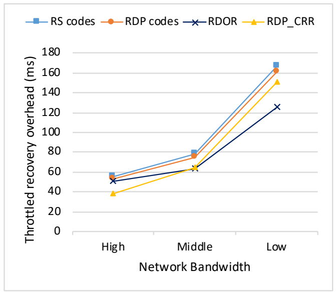
$$\text{MTTDL} = \frac{\text{MTTF}^3}{N \times (G-1) \times (G-2) \times \text{MTTR}^2}$$

where MTTF is the mean time to node failure. G is the stripe size, and N is the total number of nodes in the systems.

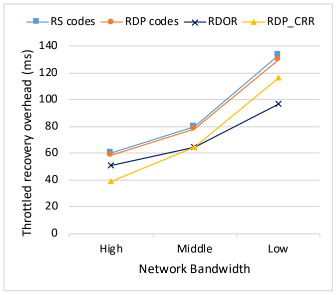
From this formula, we can see that since the efficient fault tolerance framework reduces the recovery time by 31%, so MTTDL will increase to 210%, which significantly improves the reliability of the system. On the other hand, if we maintain a comparable reliability, RDP $(8, 6)$ can be used instead of RS $(6, 4)$, which reduces the storage cost by 11%.

VI. Conclusion And Future Work

For a distributed storage system especially a in-memory KV-store system, efficiency is critical. When a data node is erased, the missing data need to be recovered as quickly as possible.



(a) $p = 5$



(b) $p = 7$

Fig. 12. Memcached throttled recovery overhead when a node crashes

In this paper, we utilize RDP codes instead of RS code used in Cocytus to make the distributed system more efficient. We adopt two schemes, RDOR and CRR decoding architecture, to reduce the recovery time of RDP codes which significantly improve the user experience and the system reliability. Finally, this paper evaluates the performance of the two optimization schemes and point out the different application scenarios of the two methods.

In the future, we plan to look for more efficient, more scalable solutions to optimize availability for distributed Memcached. For example, it would be possible to develop a "node avoiding" single-failure recovery scheme for RDP codes: when a system is online, a node may be busy serving user requests, and utilizing it for the recovery of another node may hurt this node's responsiveness. Given that RDP is tolerant of 2 erasures, we can recover an erased node while avoiding the busiest non-erased node.

Acknowledgement

This work is partially supported by NSF of China (61602266,61872201), Science and Technology Development Plan of Tianjin (17JCYBJC15300, 16JCYBJC41900) and the Fundamental Research Funds for the Central Universities and SAFEA:Overseas Young Talents in Cultural and Educational Sector.

References

[1] S. John Walker, *Big data: A revolution that will transform how we live, work, and think*. Taylor & Francis, 2014.
[2] T. Bakuya and M. Matsui, "Relational database management system," 1997, uS Patent 5,680,614.
[3] B. Fitzpatrick, "Distributed caching with Memcached," *Linux J.*, vol. 2004, no. 124, p. 5, 2004.
[4] J. Zawodny, "Redis: Lightweight key/value store that goes the extra mile," *Linux Magazine*, vol. 79, 2009.
[5] P. Saab, "Scaling Memcached at Facebook," 2008.
[6] C. Aniszczyk, "Caching with Twemcache," *Twitter Blog, Engineering Blog*, pp. 1–7, 2012.
[7] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The primary-backup approach," in *Distributed Systems*, 1993, pp. 199–216.
[8] J. Plank and C. Huang, "Tutorial: Erasure coding for storage applications," in *Slides presented at FAST-2013: 11th Usenix Conference on File and Storage Technologies*, 2013.
[9] S. B. Wicker and V. K. Bhargava, *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.
[10] H. Y. Lin and W. G. Tzeng, "A secure erasure code-based cloud storage system with secure data forwarding," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 6, pp. 995–1003, 2012.
[11] D. Shankar, X. Lu, and D. K. Panda, "High-performance and resilient key-value store with online erasure coding for big data workloads," in *IEEE International Conference on Distributed Computing Systems*, 2017, pp. 527–537.
[12] J. Harshan, F. Oggier, and A. Datta, "Sparsity exploiting erasure coding for resilient storage and efficient i/o access in delta based versioning systems," vol. 664, no. 4, pp. 798–799, 2014.
[13] B. Sklar, "Reed-Solomon codes," *Downloaded from URL http://www. informit. com/content/images/art. sub.–sklar7. sub.–reed-solomon/elementLinks/art. sub.–sklar7. sub.–reed-solomon. pdf*, pp. 1–33, 2001.
[14] H. Zhang, M. Dong, and H. Chen, "Efficient and available in-memory KV-store with hybrid erasure coding and replication," in *Proc. FAST*, 2016, pp. 167–180.
[15] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar, "Row-diagonal parity for double disk failure correction," in *Proc. FAST*, 2004, pp. 1–14.
[16] L. Xiang, Y. Xu, J. C. S. Lui, and Q. Chang, "Optimal recovery of single disk failure in RDP code storage systems," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 1, pp. 119–130, 2010.
[17] P. Li, X. Jin, R. J. Stones, G. Wang, Z. Li, X. Liu, and M. Ren, "Parallelizing degraded read for erasure coded cloud storage systems using collective communications," in *Proc. ISPA*, 2017.
[18] J. Jose, H. Subramoni, M. Luo *et al.*, "Memcached design on high performance RDMA capable interconnects," in *Proc. ICPP*, 2011, pp. 743–752.
[19] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 29–43, 2003.
[20] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. MSST*, 2010, pp. 1–10.
[21] B. Fitzpatrick and A. Vorobey, "Memcached: a distributed memory object caching system," 2011.
[22] J. Petrovic, "Using memcached for data distribution in industrial environment," in *International Conference on Systems*, 2008, pp. 368–372.
[23] W. Cheng, F. Ren, W. Jiang, and T. Zhang, "Modeling and analyzing latency in the memcached system," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 2017, pp. 538–548.
[24] P. Garefalakis, P. Papadopoulos, and K. Magoutis, "Acazoo: A distributed key-value store based on replicated lsm-trees," in *Reliable Distributed Systems (SRDS), 2014 IEEE 33rd International Symposium on*. IEEE, 2014, pp. 211–220.
[25] D.-c. QU and Y.-s. LIU, "Design of a distributed mem-agent system," *Journal of Beijing Institute of Technology*, vol. 10, p. 010, 2005.
[26] C. Huang, H. Simitci, Y. Xu *et al.*, "Erasure coding in Windows Azure storage," in *Proc. USENIX ATC*, 2012, pp. 15–26.
[27] D. Borthakur, R. Schmidt, R. Vadali, S. Chen, and P. Kling, "HDFS RAID," in *Hadoop User Group Meeting*, 2010.
[28] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman, "An xor-based erasure-resilient coding scheme," *Proc Acm Sigcomm*, 1995.
[29] J. S. Plank and L. Xu, "Optimizing cauchy reed-solomon codes for fault-tolerant network storage applications," in *Network Computing and Applications, 2006. NCA 2006. Fifth IEEE International Symposium on*. IEEE, 2006, pp. 173–180.
[30] C. Canudas-de Wit, F. R. Rubio, J. Fornes, and F. Gómez-Estern, "Differential coding in networked controlled linear systems," in *American Control Conference, 2006*. IEEE, 2006, pp. 6–pp.
[31] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster," *Usenix Hotstorage*, 2013.
[32] V. Dogaru, O. Purdilă, and N. Ţăpuş, "Network interface grouping in the Linux kernel," in *Proc. ICNC*, 2011, pp. 131–135.
[33] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. SOCC*, 2010, pp. 143–154.
[34] J. S. Plank, "Jerasure: A library in C/C++ facilitating erasure coding for storage applications," 2007.
[35] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "Raid: High-performance, reliable secondary storage," *ACM Computing Surveys (CSUR)*, vol. 26, no. 2, pp. 145–185, 1994.