Building Efficient and Available Distributed Transaction with Paxos-based Coding Consensus

Shenglong Li*, Quanlu Zhang†, Zhi Yang*, Hanyu Zhao* and Yafei Dai*
*Peking University, †Microsoft Research Asia
lishenglong@pku.edu.cn, Quanlu.Zhang@microsoft.com, {yangzhi, zhaohanyu, dyf}@pku.edu.cn

***Abstract*—Supporting distributed transaction is a key function for large-scale database systems. Conventional database systems build distributed transactions on the top of replication storage to provide high availability. However, replication induces a large amount of storage overhead. In this paper, we make the first attempt to build highly available distributed transactions over erasure coding to achieve high storage efficiency. We propose *Eunice*, an *E*fficient and available distributed transaction protocol that *Uni*fies *C*oncurrency control and *E*rasure coding. In our protocol, we first design a single-layered coding update mechanism to optimize transaction latency. Then we propose a Paxos-based coding consensus protocol to provide fault-tolerance and strong consistency for coding update operations. Compare with conventional distributed transaction protocol with replication, Eunice can save up to 41.9% storage consumption, while achieving comparable throughput and latency performance.**

***Index Terms*—Distributed Transaction; Erasure Coding; High Availability**

I. Introduction

Recent research works pay close attention to the data availability of distributed database [1–4]. Replicating each data partition among distributed nodes is a traditional way to guarantee data availability [5–7]. However, replication incurs a large amount of storage overhead, which requires at least $f+1$ times storage space to tolerate f node failures. This overhead is becoming a major bottleneck as the amount of data grows faster than datacenter infrastructure [8, 9]. Especially for increasingly prevalent in-memory databases [10, 11], a mass of memory consumption for replications becomes an extremely serious problem [12].

Erasure coding is an advanced redundant scheme integrated in many large-scale storage systems, such as Windows Azure [9], Facebook' HDFS [8], which contributes to saving storage consumption to a great extent. Currently, some emerging research works focus on applying erasure coding in in-memory key-value stores (KV-stores), which is derived from a key observation that high-performance CPUs and high-bandwidth network give an opportunity to perform fast coding and recovery on the critical path of online services [12–14].

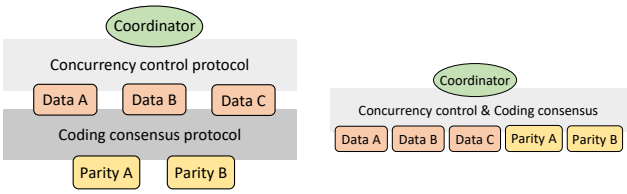
Although some KV-store databases move away from traditional full ACID properties [5, 6], supporting distributed transactions is still a crucial function for large-scale database systems, such as Google's Spanner [1] and Yahoo's Omid [15]. However, to the best of our knowledge, few research works attempt to support transaction processing over erasure coding. As the extreme growth of the database size and significance of distributed transaction, it is an urgent need to employ a more space-efficient redundancy scheme for transactional database system. Thus, we aim to fill the gap of applying erasure coding in transactional database systems.

However, supporting transaction over erasure coding faces several great challenges to meet the requirements for transaction latency and consistency. First, erasure coding has a unique redundancy pattern compared with replication, which performs different update operations on original data and redundant data (called *parity*). As a result, committing a transaction has to suffer extra coordination latency between data node and parity node. Besides, erasure coding requires *atomic update* on all the coding blocks in one *coding stripe* for each update operation (we call it *coding consensus*). Traditional implementation of coding consensus such as two-phase commit (2PC) can not provide fault tolerance for coding update operation, which leads to stalling transaction processing during node failures.

In this paper, we propose *Eunice*, an efficient distributed transaction protocol integrating erasure coding to achieve high data availability and high storage efficiency. To address the above challenges caused by the unique features of erasure coding, we first design a *single-layered coding update* mechanism to eliminate coordination between data node and parity node for coding update operation, which facilitates unifying transaction consistency (*e.g.,* concurrency control protocol) and coding consensus to achieve low-latency transaction processing. Second, we propose a *Paxos-based coding consensus* protocol to provide fault-tolerant coding update. Specifically, we define a *coding group* for each coding update operation to maintain *delta* (intermediate result for updating data and parity [12]), and leverage Paxos [16] to achieve fault-tolerate and consistent *delta* broadcast, which guarantees the coding consensus through delta recoverability. Further, we present on-demand recovery strategy to provide uninterrupted transaction processing during node failures. At last, we consolidate above designs into Eunice protocol and build a distributed in-memory distributed database system based on KV-store. In comparison with conventional transaction protocols with replication, Eunice can save up to 41.9% storage consumption and achieve comparable throughput and latency performance.

Our contributions can be summarized as follows:

- We are the first to integrate erasure coding with distributed transaction to provide high data availability with low database storage consumption.
- We design a novel coding update mechanism to facilitate unifying concurrency control protocol and coding consensus efficiently.



(a) A naive implementation. (b) Eunice implementation.

Fig. 1: A comparison between two implementation of integrating erasure coding with transaction protocol.

- We combine Paxos protocol and erasure coding creatively to achieve consistent and fault-tolerate coding update.
- We present on-demand recovery strategy to process transactions uninterruptedly during node failures.
- We build an in-memory distributed database integrating above designs and evaluate our system comprehensively. Compared with different transaction protocols, Eunice can achieve high storage efficiency, comparable throughput and latency performance.

II. Eunice Design

A. Single-layered Coding Update

We first aim at integrating concurrency control protocol with coding consensus efficiently to provide low-latency transaction processing. A naive implementation is adding a coding consensus layer over concurrency control protocol (as shown in Figure 1(a)). Specifically, transaction coordinator first communicates with data node to execute concurrency control protocol and then data node runs another coordination with parity nodes to achieve coding consensus. We notice that these two coordination phases need to be executed serially to guarantee the consistency of concurrent transactions and coding stripes, which incurs high transaction processing latency. Thus, we aim to eliminate the extra coding consensus coordination latency and merge coding consensus with concurrency control in a single-layer (as shown in Figure 1(b)).

In the traditional implementation of coding storage, one coding stripe consists of k data blocks and m parity blocks, which are distributed across k data nodes and m parity nodes respectively. Parity blocks are calculated by a linear combination of data blocks (each object can be viewed as a data block). To complete coding update operation, coordinator needs send the written data to data node first, then data node computes a *delta* between the new object and its old version, after that it sends the delta to all the parity nodes to update parity blocks. In our design, we can allow the transaction coordinator to obtain the *delta* before sending update requests, thus coordinator can update all the data nodes and parity nodes in one round-trip by broadcasting the delta.

Based on this idea, we propose a *single-layered coding update* mechanism which leverages coordinator to compute delta and broadcast delta instead of data node. We illustrate this mechanism in Figure 2. Specifically, in our transaction protocol, we employ Optimistic Concurrency Control (OCC), which executes transactions without locking and validates

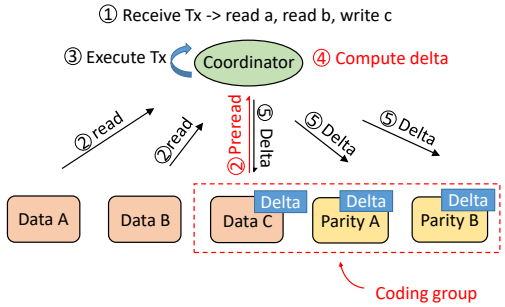


Fig. 2: A sample of transaction processing based on Eunice protocol. *The sample transaction requests data a and b from data node A and B. After transaction execution, coordinator obtains the new data c and writes c to data node C.*

transaction conflictions after execution. We use coordinator as transaction executor to issue distributed read and write requests. Different from traditional OCC protocol, we allow the coordinator to read the objects of *write-set* (*i.e.,* a set of keys to be written) of the transaction during the transaction execution phase (we call this operation as *preread*). After coordinator executes the transaction, it can compute the deltas between the new objects and old objects of write-set locally. Then it can broadcast the delta to one data node and all the parity nodes (we call them *coding group*) to update data blocks and parity blocks for each write operation in one round-trip.

B. Paxos-based Coding Consensus

Then we aim to provide fault-tolerant and atomic coding update. To guarantee the consistency of coding stripe, the coding blocks need to be updated *atomically*. Otherwise, when some nodes fail, the survived nodes may maintain incomplete coding stripes, which leads to decoding incorrect data during recovery. Classic implementation of atomic coding update is two-phase commit [12]. However, two-phase commit requires receiving the responses from all the nodes in coding group successfully. Thus, it could not tolerate node failures, which leads to the failure of coding update operation. Further, transaction processing will be stalled during node failures.

We observe that an essential cause of inconsistency of coding stripe is derived from updating data and parity blocks *in-place* without maintaining delta history version. If we can obtain delta history during node failures, all the nodes in coding group can apply the delta eventually to guarantee the consistency of coding stripe. Therefore, we can allow the majority of nodes in coding group to maintain the delta. During recovery, deltas which are prepared successfully (received by the majority) can be synchronized in the coding group to recover the consistent delta state. Thus, we can tolerate the failures of minority nodes and guarantee the consistency of coding stripe through delta recoverability.

Based on this observation, we propose *Paxos-based coding consensus* protocol to achieve fault-tolerant delta broadcast in the coding prepare phase. Specifically, we utilize *delta instance* as Paxos instance to maintain delta value and corresponding transaction metadata. We define each coding group

as Paxos group which executes Paxos protocol to maintain the consistent delta instance. We allow transaction coordinator to act as Paxos proposer and allow nodes in the coding group to act as Paxos acceptors. Each node maintains a *delta queue* to receive delta instances. If transaction coordinator receives the successful responses from the majority of nodes in coding group, the coding prepare phase can be returned successfully. After all the nodes complete applying the delta instance, they can remove the delta instance safely.

During the node failure, delta consistency among coding group is first recovered. When some nodes fail, each survived node broadcasts its delta instances in the delta queue to the other nodes in the same coding group. Then each survived node finds out the lost delta instances compared with other nodes and puts them into its own delta queue. After coding group recovers a consistent delta state, it can request the result of transaction prepare phase from coordinator and apply the delta on the data blocks or parity blocks selectively.

Then we analyze the fault-tolerance ability of Paxos-based coding consensus. Based on the property of erasure coding, we can tolerate at most m node failures (m is the number of parity nodes) among all the data nodes and parity nodes. Simultaneously, in each coding group, we can tolerate failures of minority nodes (f node failures among $2f$+1 nodes).

C. Holistic Transaction Protocol

We first introduce the OCC conflict validation algorithm deployed in each data node. Each data node first acquires all the locks of objects in the transaction's write-set. If any lock operation fails, data node will return ABORT to the coordinator and coordinator informs all the participant nodes in the write-set to release all the locks. After all the write locks are acquired successfully, data node examines all the objects in the transaction's *read-set* (*i.e.,* a set of keys that were read). If any object has a different timestamp compared with the object read in the execution phase, or is locked by other transactions, the data node returns ABORT. If all the examinations are passed, the data node will return PREPARE_OK to the coordinator.

Then we present the overall distributed transaction protocol.First, in the execution phase, coordinator reads all the data with timestamp in the read-set and write-set. Then coordinator executes the transaction based on stored procedures and obtains the new objects of write-set. Next coordinator computes deltas between new objects and old objects of write-set.

In the prepare phase, coordinator sends the keys with timestamp in read-set and write-set to corresponding nodes to execute OCC validation algorithm. Simultaneously, coordinator broadcasts the delta instances to each coding group. If coordinator receives ABORT from any data node, it returns transaction abort to client and persists the validation result in the local log. Then coordinator notifies the participants in the write-set to release the object locks and each coding group to remove delta instances. Otherwise, if coordinator receives PREPARE_OK from all the participant data nodes and the successful responses from the majority of nodes in each coding group for each transaction update operation, it writes the result to the local log and continues executing the commit phase. To tolerate coordinator failure, we use a group of $2f$+1 coordinators to persist preparing result. After confirming the preparing results from all the involved nodes, coordinator writes the results (PREPARED_OK or ABORT) to the coordinator group through Paxos protocol.

In the commit phase, coordinator informs all the nodes of each coding group to apply delta instance.Data node adds the delta on the data blocks directly and updates data timestamp. Parity node adds the product of a coding coefficient and delta on the parity blocks. Locks can be released immediately after the object and timestamp are updated successfully.

D. Recovery

At last, we design on-demand recovery strategy to process transactions uninterruptedly during node failures, which includes coordinator failure, data node failure and parity node failure (we call data node and parity node failure as *storage node failure*). We handle only omission node failures where a node is fail-stop and does not impact other nodes. Commission or Byzantine failures are also not considered in our scenario.

Coordinator recovery. When coordinator fails, we use view change protocol based on Paxos to elect the new coordinator in coordinator group. The new coordinator first reads the transaction log persisted in coordinator group, then it recommits the ongoing transactions which are labeled PREPARED_OK and aborts the ongoing transactions which are labeled ABORT. For the unprepared ongoing transactions, the new coordinator will inform all the participants to release the involved locks, then retry the execution phase and prepare phase.

Storage node recovery. When storage node fails, the coordinator first reads the transaction log to examine the prepare state. Then we launch a *background recovery process* in the coordinator to recover the lost data of the failure node. Specifically, for successful prepared transactions, we synchronize the delta in each coding group and apply the delta to the original data block or parity block. After reaching a consistent coding state among survival storage nodes, the coordinator communicates with any k storage nodes of survived nodes to request all the data blocks or parity blocks. Then it can decode the lost blocks according to coding formula.

Besides, to deal with ongoing transactions and incoming transactions which requests the data on the failed node, we use a *foreground recovery process* in the coordinator to recover the requested data online after reaching the consistent delta state in each coding group. Foreground recovery process first locates the coding stripe based on requested key. Then it requests coding blocks of located coding stripe from any k nodes, and recovers the lost data in the new storage node preferentially. For prepared and uncommitted transactions, the coordinator will inform the new storage node to apply the delta on the data block or parity block after data recovery. For unprepared transactions, coordinator informs all the transaction's participants to release locks and remove delta first, and then retry the execution phase and prepare phase.

III. Evaluation

A. Setup

System and cluster configurations. We implement a distributed in-memory KV-store integrating Eunice protocol, which is developed based on asynchronous RPC framework. Values are coded in our system, while metadata (*e.g.,* keys and coding stripe index) is replicated across distributed storage nodes for metadata availability. We run all the experiments on Amazon EC2 platform. We launch several m4.large instances as storage nodes and coordinator nodes in multiple datacenters. The machines are distributed in Oregon (us-west), Seoul (ap-northeast), Ireland (ec-west), Virginia (us-east) and California (us-west). For replication experiments, we set replication factor as three and use Paxos to achieve replication consensus. For comparison, we define the number of parity nodes as two to achieve the same Paxos quorum as replication. We implement RS(3,2) coding scheme by default (RS(*k*,*m*) denotes Reed-Solomon codes [17] with *k* data nodes and *m* parity nodes).

In the single datacenter experiments, we deploy all the nodes in Oregon datacenter. For geo-distributed datacenter experiments, we deploy data nodes and coordinator nodes in Oregon, Virginia and California datacenter. We deploy replication nodes and parity nodes in Seoul and Ireland datacenter. We also deploy coordinator group to tolerate coordinator failure. One coordinator is elected as coordinator leader to manage log persistence. Each coordinator can issue transaction requests and act as the transaction executor. By default, we utilize one coordinator to issue transaction requests. We launch different numbers of client processes in the coordinator. Each client process issues the transaction requests with a fixed request rate. Transaction read operations are performed on data nodes for both replication and erasure coding configuration. Besides, we set a flow control window and message timeout to limit the number of on-flying transactions. When transaction request is overtime, we abort the transaction first and retry it.

Targets of comparison. We evaluate different transaction protocols for comparison with Eunice. We implement a two-layered coding consensus protocol called *LayeredCE* which layers OCC and coding consensus protocol. Specifically, in the prepare phase, coordinator first communicates with data nodes to execute conflict validation, then data node computes delta and broadcasts it to all the parity nodes. In the commit phase, coordinator first commits the transaction on data nodes, then data node informs parity nodes to apply the delta.

Besides, we also implement a two-layered transaction protocol with replication called *LayeredCR*, which combines OCC and Paxos (as Percolator [2] implementation). Specifically, in the prepare phase, coordinator first communicates with replication leader to validate confliction, then replication leader communicates with backups to confirm the validation result. In the commit phase, coordinator also writes the data to the replication leader first and then replication leader writes the data to all the backup nodes in another round-trip. At last, we implement a single-layered transaction protocol with replication called *UnifiedCR* which unifies OCC and Paxos (as TAPIR [3] implementation). In this protocol, coordinator communicates with all the replication nodes directly to complete prepare phase and commit phase in one round-trip instead of communicating with replication leader.

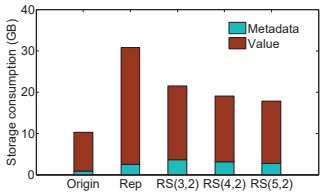


Fig. 3: Storage consumption for different redundancy schemes.

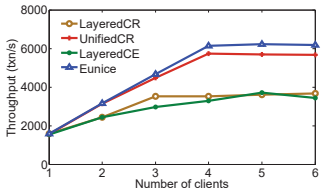


Fig. 4: Throughput of different transaction protocols for microbenchmark in single datacenter.

Workload. We evaluate our system with microbenchmark and TPCC benchmark [18]. To initiate database, we generate one hundred million keys based on TPCC workload. We partition the database tables into three data nodes uniformly. In replication experiments, keys and values are replicated among data nodes and replication nodes. In erasure coding experiments, we first calculate parity blocks based on values and generate stripe metadata. Then we send values and parity blocks with stripe metadata to data nodes and parity nodes respectively. For microbenchmark, each transaction performs one read-write operation on a random key of TPCC tables. For TPCC benchmark, we run write-intensive New-order benchmark.

B. Storage Consumption

First, we evaluate storage consumption of different redundancy schemes. We implement different RS coding schemes in our system including RS(3,2), RS(4,2) and RS(5,2). As shown in Figure 3, Eunice performs notable storage saving compared with replication protocol (*i.e., Rep* in the figure). With the increase of the number of data nodes, Eunice achieves higher storage efficiency which is up to 41.9% storage saving for RS(5,2) scheme. For RS(3,2) coding scheme, the expected storage overhead of erasure coding should be 1.67x while actual storage overhead compared with original storage is 2.09x. Because Eunice maintains the coding stripe metadata and replicates metadata across different nodes. However, the size of metadata is much less than the value, which takes up only 8.3% of database size. Thus erasure coding on values dominates the whole storage saving.

C. System Performance

Then we test system throughput and latency performance in both single datacenter and geo-distributed datacenters.

Single datacenter. Figure 4 shows the average throughput of different transaction protocols for microbenchmark workload. As the number of clients increases, the throughput performance of all the protocols increases. The throughput climbs and stabilizes at a peak throughput. The increase of the number of requests leads to network congestion, which incurs more transaction timeouts. Single-layered protocols can

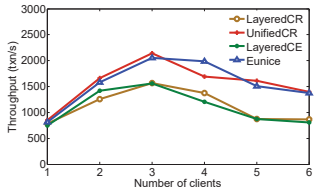


Fig. 5: Throughput of different transaction protocols for TPCC workload in single datacenter.

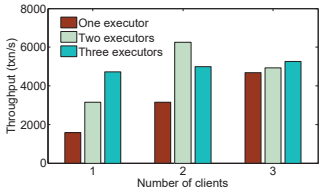


Fig. 6: Throughput of Eunice with different executors for microbenchmark in single datacenter.

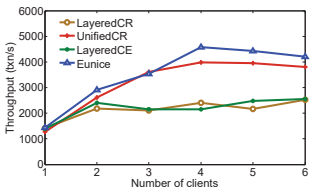


Fig. 9: Throughput of different transaction protocols for microbenchmark in geo-distributed datacenters.

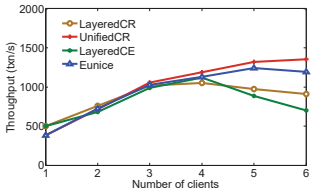


Fig. 10: Throughput of different transaction protocols for TPCC workload in geo-distributed datacenters.

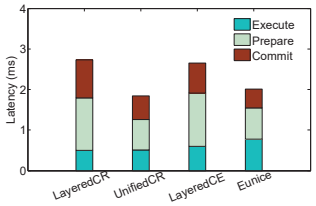


Fig. 7: 90% latency of different transaction protocols for microbenchmark in single datacenter.

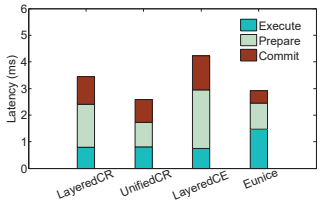


Fig. 8: 90% latency of different transaction protocols for TPCC workload in single datacenter.

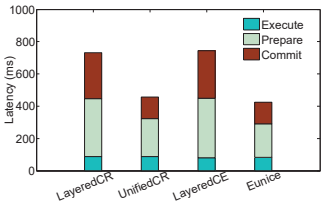


Fig. 11: 90% latency of different transaction protocols for microbenchmark in geo-distributed datacenters.

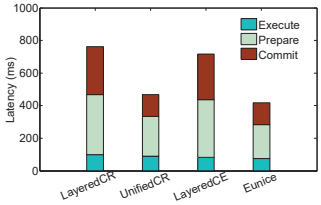


Fig. 12: 90% latency of different transaction protocols for TPCC workload in geo-distributed datacenters.

commit transaction with lower latency than two-layered protocols, which induces less transaction timeouts under high request load. Thus UnifiedCR and Eunice can achieve higher throughput than LayeredCR and LayeredCE. We can also see that erasure coding protocols can achieve similar performance as replication protocols, which results from the fast coding operations with powerful CPUs.

Figure 5 indicates the throughput performance under TPCC workload. The throughput performance shows a similar trend as the results of microbenchmark. The throughput first climbs with the increase of request load and then drops with high request load. In addition to the cause of network bottleneck The throughput degradation is also derived from transaction conflictions. As the request load increases, the transaction abort rate increases for all the protocols. The high transaction latency of two-layered protocols results in lengthy resource locking time, thus LayeredCR and LayeredCE has potential to induce more transaction conflictions than UnifiedCR and Eunice under high request load.

Then we show the throughput performance of Eunice using different numbers of executors in Figure 6. Each executor is distributed in different coordinators. We deploy different numbers of clients in each executor. With the increase of the number of executor, the network on the server side becomes the bottleneck, which leads to transaction timeouts and limits the increase of throughput. With the same number of executors, the throughput climbs with the increase of the number of clients until reaching the network bottleneck.

Next we compare transaction latency between different transaction protocols. Figure 7 and Figure 8 show 90-percentile latency under the same request rate for microbenchmark and TPCC workload. We analyze different latency components in three transaction phases including execute phase, prepare phase and commit phase. Specifically, prepare latency includes the cost of transaction validation and log persistence on coordinator group. LayeredCR incurs high latency because of extra coordinations between replication leader and backups in prepare phase and commit phase. LayeredCE performs similar latency as LayeredCR due to the extra coordination between data node and parity node. Particularly, LayeredCE spends higher latency than LayeredCR on prepare phase because of delta computing overhead. Eunice can achieve lower latency in both prepare and commit phase because we merge the coding consensus protocol with the concurrency control protocol. Besides, Eunice increases the latency of execute phase slightly compared with UnifiedCR because of extra delta computing overhead. In the single datacenter experiment, even though the latency saving on extra coding coordination is not a great proportion of the whole latency, Eunice can also achieve up to 26.3% latency saving compared with LayeredCR and 31% latency saving compared with LayeredCE.

Geo-distributed datacenters. Then we deploy our system in geo-distributed datacenters and test system performance using different transaction protocols. As shown in Figure 9 and Figure 10, transaction throughput preforms the same trend as the single datacenter experiment. With the increase of number of clients, the amount of transaction conflictions and resource consumption increases, which results in the throughput saturation. Figure 11 and Figure 12 show the 90-percentile latency of different transaction protocols for microbenchmark and TPCC workload. Different from single datacenter experiment, the latency between geo-distributed nodes dominates the whole latency cost, which is far greater than computing overhead. Taking advantage of unifying concurrency control and coding consensus, Eunice can achieve up to 45.2% latency saving compared with LayeredCR and 42.8% latency saving compare with LayeredCE. Besides, Eunice can achieve similar latency performance as UnifiedCR under both of microbenchmark and TPCC workloads, because computation overhead in execute phase can be ignored compared with coordination latency between geo-distributed datacenters.

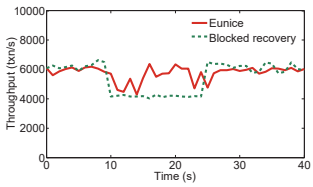


Fig. 13: Real-time throughput during data failure.

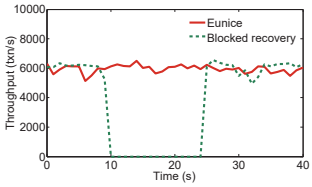


Fig. 14: Real-time throughput during parity failure.

D. Recovery Efficiency

First, we initial a small size of in-memory database and run microbenchmark with a fixed request rate and monitor the system throughput. During the system running, we emulate a data node failure and a parity node failure respectively at 10s. We launch a new coordinator node to process background data recovery. We compare on-demand recovery method applied in our system with traditional blocked data recovery, which blocks the transaction requests until completing data recovery.

Figure 13 shows the real-time throughput of two recovery methods during data node failure. On-demand recovery method can recover the lost data requested by incoming transactions in real-time. For blocked data recovery, new node needs to recover the whole lost data and then serves transaction requests. During the data recovery, transaction requests onto the lost data node will be stalled. Figure 14 shows the throughput during parity node failure. We can see that our system can maintain the system throughput because of fault-tolerate coding update. The coordinator can recover the delta state of coding group and reconstruct the lost parity block in background. However, blocked recovery will stop the services due to the failure of atomic update. After the parity node recovers all the parity blocks, the system can continue to provide transaction processing.

IV. Related Work

Erasure Coding. Erasure coding is widely used in storage systems in both academic research and industry [8, 9] to achieve high data availability and space efficiency. MemEC [13] and Cocytus [12] uses erasure coding to maintain the data availability in the in-memory KV-store to achieve speedy data recovery. Different from existing erasure coding systems, we integrate erasure coding with concurrency control protocol efficiently to provide highly available distributed transactions.

Available Distributed Database. Many recent database systems apply replication to guarantee data availability and provide different degrees of replication consistency and transaction support. Some NoSQL databases, such as Dynamo [5], Apache's Cassandra [6] leverage weak consistent replications to achieve high scalability and system performance, but do not support transaction processing. Google Spanner [1] and Percolator [2] builds concurrency control protocols on top of replication consensus. TAPIR [3] and MDCC [4] unify the concurrency control protocol and replication protocol into one layer to reduce the overhead of two-layered coordination. Different from existing database systems, we aim to support serializability of transaction processing, while integrating erasure coding to achieve high storage efficiency.

V. Conclusion and Future Work

In this paper, we propose Eunice, a highly available distributed transaction protocol over erasure coding. We unify concurrency control and coding consensus to achieve low transaction processing latency. We propose Paxos-based coding consensus to provide fault-tolerant coding update. In the future work, we plan to explore deterministic transaction execution to optimize the latency of log persistence. Second, we plan to investigate the load balance between coding nodes and improve the resource utilization of the whole cluster.

References

[1] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Googles globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, p. 8, 2013.

[2] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications." in *OSDI*, vol. 10, 2010, pp. 1–15.

[3] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. Ports, "Building consistent transactions with inconsistent replication," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 263–278.

[4] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete, "Mdcc: Multi-data center consistency," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 113–126.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[6] A. Lakshman and P. Malik, "Cassandra: structured storage system on a p2p network," in *Proceedings of the 28th ACM symposium on Principles of distributed computing*. ACM, 2009, pp. 5–5.

[7] P. Membrey, E. Plugge, and T. Hawkins, *The definitive guide to MongoDB: the noSQL database for cloud and desktop computing*. Springer, 2010.

[8] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "Xoring elephants: Novel erasure codes for big data," in *Proceedings of the VLDB Endowment*, vol. 6, no. 5. VLDB Endowment, 2013, pp. 325–336.

[9] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci *et al.*, "Windows azure storage: a highly available cloud storage service with strong consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 143–157.

[10] B. Fitzpatrick, "Distributed caching with memcached," *Linux journal*, vol. 2004, no. 124, p. 5, 2004.

[11] J. Zawodny, "Redis: Lightweight key/value store that goes the extra mile," *Linux Magazine*, vol. 79, 2009.

[12] H. Zhang, M. Dong, and H. Chen, "Efficient and available in-memory kv-store with hybrid erasure coding and replication." in *FAST*, 2016, pp. 167–180.

[13] M. M. Yiu, H. H. Chan, and P. P. Lee, "Erasure coding for small objects in in-memory kv storage," in *Proceedings of the 10th ACM International Systems and Storage Conference*. ACM, 2017, p. 14.

[14] W. Litwin, R. Moussa, and T. Schwarz, "Lh* rs—a highly-available scalable distributed data structure," *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 3, pp. 769–811, 2005.

[15] E. Bortnikov, E. Hillel, I. Keidar, I. Kelly, M. Morel, S. Paranjpye, F. Perez-Sorrosal, and O. Shacham, "Omid, reloaded: Scalable and highly-available transaction processing." in *FAST*, 2017, pp. 167–180.

[16] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[17] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[18] "TPC-C Benchmark," http://www.tpc.org/tpcc/.