Fault-Aware Messaging for Deterministic Sequencing in Caerus

Brian Lam[1] Hideyuki Kawashima[1]

Abstract: Caerus orders geo-replicated transactions by merging per-region partial sequences; a merger places a transaction only after all of its parts have arrived, which is safe only over links that never lose a message. Across a broken TCP connection, the original implementation's messaging layer lost messages without any error in 23 of 40 runs. We reimplemented Caerus's sequencing pipeline in C++ over gRPC with a per-link delivery contract. On one machine over loopback, the contract delivered every message exactly once in all 100 break runs that recovered, recovered from an unsignaled connection failure within 1.91 s using gRPC keepalive, and added about 3 μs to median and 99th-percentile transport latency over a bare gRPC stream.

1. Introduction

Deterministic databases such as Calvin [2] agree on a transaction order before execution, so replicas reach the same state without coordinating during execution. Caerus [1] makes this agreement cheaper for geo-replicated data: each region sequences the parts of a transaction that touch keys it masters, and every region's merger combines the partial sequences into a conflict-equivalent order within one wide-area round trip. A merger places a transaction only after the parts from all of its participant regions have arrived. This completeness gate is sound only over links that are lossless, FIFO and duplicate-free, which TCP guarantees within one connection but not across a break.

We tested the original's messaging classes (ZeroMQ PUSH/PULL), compiled unmodified from the authors' code, in our own sender and receiver programs. We put a TCP relay between them, killed it mid-stream and restarted it 2 s later. In two series, the original lost messages in 14 of 20 and 9 of 20 runs (869 messages in all), and neither side logged an error. By our reading of its code, one lost part keeps a completeness gate open forever: that transaction, and every transaction ordered after it, stalls with no error.

We reimplemented the pipeline (Batcher, Partial Sequencer, Merger) in C++ over gRPC so that such failures are repaired, or reported when they cannot be. It adds (1) a per-link delivery contract with dead-link detection, (2) runtime and offline checks of the determinism invariants, and (3) measurements of recovery latency and of the contract's no-fault cost.

[1] Keio University

2. Design

Sequencing pipeline. A Batcher gives each transaction a 64-bit ID from its node's range; no later stage changes it, and mergers use it to break ties inside conflict cycles. One shared function derives a transaction's participant regions from its keys, and every stage that needs them calls it. A merger builds a conflict graph from the parts, condenses strongly connected components (Tarjan), and emits a component once it is complete and has no pending predecessor, members in ascending ID order. Orders may differ between mergers with arrival timing, so the verifier checks conflict-equivalence key by key: the same writer order for each key, and each read after the same write. With 2 and 3 regions (each with its own Batcher, Partial Sequencer and merger), all mergers' orders were conflict-equivalent, in small runs built to force a conflict cycle and different orders, and in three 8,010-transaction runs with 3 regions, in which any two mergers ordered 505–596 pairs of transactions differently. Reading the original's code found a defect: its Partial Sequencer never projects the write set, so blind writes get no conflict edges (no original workload issues blind writes; our tests fail if we reintroduce the defect).

Delivery contract. Each directed link runs over one persistent gRPC bidirectional stream, with sequence numbers, asynchronous watermark acknowledgments, and a resend buffer that keeps each message until an acknowledgment covers it. When a stream is re-established, the receiver first advertises its watermark, and the sender resends from there. The receiver discards duplicates. Across connection breaks, no transaction is dropped: it is ordered, or an alarm names the stuck link or the completeness gate that stays open and the region that is missing. Acknowledgments are sent every 100 ms, and neither side waits for one.

© 2026 Information Processing Society of Japan

Table 1 Signaled failure: mean wait (ms) after the relay's restart, attempts in the 8 s outage, and messages lost in 30 runs.

Configuration	0.5/2/8 s wait	Attempts	Lost
Contract, 1 s	527/558/690	8	0
Contract, 100 ms	73/102/186	76–84	0
Contract, backoff	383/576/618	10–11	0
Original	61/93/54	—	604

Dead-link detection. A connection failure is *signaled* if TCP reports it to its user (a reset or a close) and *unsignaled* otherwise. A signaled failure ends the stream with an error. In an unsignaled one the connection stays open but no data arrives, for example behind a frozen relay whose kernel still acknowledges every byte; only a timeout finds it. We enable gRPC v1.83.0's HTTP/2 keepalive on both sides, with a ping every $K = 1$ s and a timeout $T = 1$ s. The client reconnects after a fixed delay or with exponential backoff from 100 ms to a 1 s cap.

3. Evaluation

A sender and a receiver exchange 512-byte messages through a TCP relay on one machine (a Docker Desktop Linux VM with 10 virtual CPUs, arm64). Each configuration ran 10 times. Over a wide-area network, the contract's added microseconds would be a much smaller share of the latency. The original's code pins every node's message thread to CPU 0; in the break tests both stayed there, and we did not check whether this changes those results.

Signaled failure (Table 1). The relay is killed 4 s into a stream of 400,000 messages at 20,000 messages/s, and restarted after 0.5, 2 or 8 s plus a seeded 0–1 s jitter. In all 90 runs of the contract, every message was delivered exactly once. Every retry wait stayed within its bound: the delay or cap, plus the time of refused attempts, plus a 25 ms timer tolerance that we added after the sweep (without it, 2 runs were 1.3 and 9.6 ms over, each after a late timer). The original lost 604 messages in 18 of 30 runs. A 100 ms delay gave the shortest wait but the most attempts; backoff's wait did not differ significantly from a fixed 1 s delay's.

Unsignaled failure (Table 2). A blackhole relay keeps reading both directions but discards the bytes and never closes a socket. New connections relay normally. The rate is 1,000 messages/s, with a 60 s cutoff. Without detection, the contract did not recover within the cutoff. It lost nothing: at the cutoff its resend buffer held about 59,700 messages, and the sender's resident memory had grown from 13.8 MiB to 98.8 MiB (means of 10 runs). With keepalive, the contract recovered in 1.46–1.91 s, within its 3.08 s bound ($K+T+cap$ plus timer tolerances), with no loss. The original did not recover within the cutoff. With ZeroMQ heartbeats at the same 1 s/1 s setting, it reconnected in 2.05–2.08 s, but lost the 1,869–1,892 messages sent between the break and the new connection.

Cost with no fault (Fig. 1). The relay forwards 200,000 messages at 20,000 messages/s, and the first second is excluded. The configurations are interleaved over 10 rounds; the original's row is from a later rerun. The bare gRPC stream uses the contract's write rule. Against the bare stream, the contract's medians were 3 μs higher at both p50 and p99 of transport latency, and each send call took 2.0 μs longer, 0.8 μs of it writing the sequence log to a file. Two contract runs had a p99 of 171 and 232 μs, for unknown reasons. The original is a reference only: its send call only queues the message, our harness checks its receiver every 20 μs, as the original's merger does, and we moved its receiver's message thread to CPU 1.

Table 2 Unsignaled failure: recovery time, and messages never delivered of 70,000 per run (heartbeat: 19 valid of 20 runs).

Configuration	Recovery (s)	Never delivered
Contract, no keepalive	none in 60 s	≈66,000, all buffered
Contract, keepalive	1.46–1.91	0
Original	none in 60 s	≈66,000
Original, heartbeat	2.05–2.08	1,869–1,892

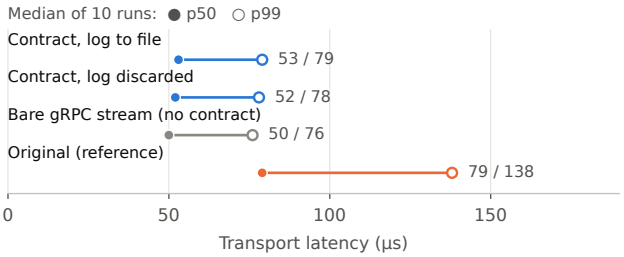


Fig. 1 No fault: send call to delivery, keepalive off.

4. Limitations and Future Work

The contract recovers connection breaks only. When a peer process restarts, its sequence numbers restart from zero, and we measured that the surviving side then discards the new process's first messages as duplicates, with no error. A stalled peer is only reported: after 5 s the stuck-link alarm names the link, but not whether the peer or the network stalled, and messages wait in the resend buffer meanwhile. The resend buffer is unbounded. Next, we will detect and refuse peer restarts, then continue after a crash from local durable logs.

Acknowledgments This paper is based on results obtained from the project "Research and Development Project of the Enhanced Infrastructures for Post-5G Information and Communication Systems (JPNP20017)" and JPNP25014 by the New Energy and Industrial Technology Development Organization (NEDO), and from JSPS KAKENHI Grant Number 25H00446, and from JST CREST Grant Number JPMJCR24R4 and from SECOM Science and Technology Foundation and from JST COI-NEXT SQAI (JPMJPF2221) and JST Moonshot R&D Grant Number JPMJMS2215.

References

[1] Hildred, J., Abebe, M. and Daudjee, K.: Caerus: Low-Latency Distributed Transactions for Geo-Replicated Systems, *PVLDB*, Vol.17, No.3, pp.469–482 (2023).
[2] Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P. and Abadi, D.J.: Calvin: Fast Distributed Transactions for Partitioned Database Systems, *Proc. SIGMOD*, pp.1–12 (2012).

© 2026 Information Processing Society of Japan