JEPSEN

TigerBeetle 0.16.11

Kyle Kingsbury
2025-06-06

TigerBeetle is a distributed OLTP database oriented towards financial transactions. We tested TigerBeetle 0.16.11 through 0.16.30. We discovered seven client and server crashes, including a segfault on client close and several panics during server upgrades. Single-node failures could cause significantly elevated latencies for the duration of the fault, and requests were intentionally retried forever, which complicates error handling. We found only two safety issues: missing results for queries with multiple predicates, and a minor issue with a debugging API returning incorrect timestamps. TigerBeetle offered exceptional resilience to disk corruption, including damage to every replica's files. However, it lacked a way to handle the total loss of a node's data. As of version 0.16.30, TigerBeetle appeared to meet its promise of Strong Serializability. As of 0.16.45, TigerBeetle had addressed every issue we found, with the exception of indefinite retries. TigerBeetle has written a companion blog post to this work. This report was funded by TigerBeetle, Inc., and conducted in accordance with the Jepsen ethics policy.

1 Background

TigerBeetle is an Online Transactional Processing (OLTP) database built for double-entry accounting with a strong emphasis on safety and speed. It builds on the Viewstamped Replication (VR) consensus protocol to offer Strong Serializable consistency. Unlike general-purpose databases, TigerBeetle stores only accounts and transfers between them. This data model is well-suited for financial transactions, inventory, ticketing, or utility metering. To store other kinds of information, users typically pair TigerBeetle with other databases, linking them through user-defined identifiers.

TigerBeetle optimizes for high-contention and high-throughput workloads, such as central bank switches or brokerages. A central bank exchange might have only a half-dozen to a few hundred account records—one for each partner bank—and process hundreds of millions of transactions per day between 647 banks. A large brokerage, after the trading day closes, might need to settle the entire day's trades as quickly as possible.[1] These trades also tend to be concentrated on a small number of popular stocks. Under high contention, per-object concurrency control mechanisms can be the limiting factor in throughput. Instead, TigerBeetle funnels all writes through a single core on the primary VR node. This limits throughput to whatever a single node can execute: TigerBeetle is firmly scale-up, not scale-out. To make that single node as fast as possible, TigerBeetle makes extensive use of batching, IO parallelization, a fixed schema, and hardware-friendly optimizations—such as fixed-size, cache-aligned data structures.

Refreshingly, TigerBeetle stresses fault tolerance in their marketing and documentation. They offer explicit models for memory, process, clock, storage, and network faults. ECC RAM is assumed to be correct. Processes may pause or crash. Clocks may jump forward and backward in time. Disks are assumed to not only fail completely, but to tear individual writes or corrupt data. Networks may delay, drop, duplicate, misdirect, and corrupt messages. To mitigate these faults, TigerBeetle combines Viewstamped Replication with techniques from Protocol-Aware Recovery, uses extensive checksums stored separately from data blocks, and for critical data, writes and reads multiple copies. TigerBeetle also makes extensive use of runtime correctness assertions to identify and limit the damage from faults and bugs alike.

Unlike most distributed systems, TigerBeetle claims to keep running without data loss if even a single replica retains a copy of a record:

> A record would need to get corrupted on all replicas in a cluster to get lost, and even in that case the system would safely halt.

To test safety under faults, TigerBeetle employs deterministic simulation testing: tests which perform reproducible, pseudo-random operations against the system and ensure that some property holds.[2] The Viewstamped Operation Replicator (VOPR) test simulates an entire TigerBeetle cluster, including clock,

[1] To give some idea of the rates involved—India's Unified Payments Interface processes roughly 16 billion transfers per month, which works out to about 6,000 per second, on average. Clear Street, a brokerage in New York, indicates that they process on the order of 30,000 debit-credit transfers per second after the market closes.

[2] Deterministic simulation testing is essentially property-based testing with techniques to turn non-deterministic systems into deterministic ones. The clock, disk state, scheduler, network delivery, external services, and so on are controlled to ensure reproducibility. For more on this approach, you might start with PULSE, Simulant, FoundationDB, and Antithesis.

disk, and network interfaces. It simulates clock skew, corrupts reads and writes, loses and reorders network messages, and so on. There are other simulation tests which stress specific subsystems, as well as a variety of more traditional integration and unit tests.

TigerBeetle also offers a noteworthy approach to upgrades. Each TigerBeetle binary includes the code not just for that particular version, but several previous versions. For example, the 0.16.21 binary can run 0.16.17, 0.16.18, and so on through 0.16.21. To upgrade, one simply replaces the binary on disk. TigerBeetle loads the new binary, but continues running with the current version. It then coordinates across the cluster to smoothly roll out each successive version, until all nodes are running the latest version available. This approach does not require operators to carefully sequence the upgrade process. Instead, upgrades are performed automatically, and coupled to the replicated state machine. This also allows TigerBeetle to ensure that an operation which commits on version x will never commit on any other version—guarding against state divergence.

1.1 Time

TigerBeetle defines an explicit model of time. Viewstamped Replication forms a totally ordered sequence of state transitions, and its view and op numbers can be used as a totally ordered logical clock. Financial systems usually prefer wall clocks, so most TigerBeetle timestamps are in "physical time," which, like Hybrid Logical Clocks, approximate POSIX time with stronger ordering guarantees. Specifically, TigerBeetle leaders collect POSIX timestamps from all replicas[3] and try to find a time which falls within a reasonable margin of error across a quorum of nodes.[4] Those timestamps are incorporated into the VR-replicated state machine, and constrained to be strictly monotonic. When no quorum of clocks falls within a twenty-second window for longer than sixty seconds, the cluster refuses requests until clocks come back in sync.

As of October 2024, TigerBeetle's documentation described TigerBeetle timestamps as "nanoseconds since UNIX epoch". This is not quite true: POSIX time is presently twenty-seven seconds less than the actual number of seconds since the epoch. During leap seconds or other negative time adjustments, TigerBeetle's clock slows to a crawl until values from `CLOCK_REALTIME` catch up.

1.2 Data Model

TigerBeetle's data model is specifically intended for double-entry bookkeeping. It has no way to represent arbitrary rows, objects, graphs, blobs, and so on.[5] Instead TigerBeetle stores two types of data: *accounts*, and *transfers* between them. All fields are fixed-size,[6] and numbers are generally unsigned integers. All values are, with limited exceptions, immutable.

An account represents an entity which sends and receives something. For example, a "Gross Revenues" account might accrue dollars, "Meadow Lake Wind Farm" might generate kilowatt-hours of electricity, and "Beyoncé" would obviously hold an ever-growing number of Grammy awards. Accounts are uniquely identified by a user-defined 128-bit `id`, a `ledger` which determines which accounts can interact with each other, a bitfield of `flags` controlling various behaviors, a creation `timestamp`, a user-defined `code`, and three custom fields of varying sizes: `user_data_32`, `user_data_64`, and `user_data_128`. There are also four derived fields which represent the current sum of transfers into (*credits*) and out of (*debits*) the account: `debits_pending`, `debits_posted`, `credits_pending`, and `credits_posted`.

A transfer is an immutable record which represents an integer quantity moving from one account to another. Like accounts, transfers have a unique, user-specified, 128-bit `id`, a `code`, a `ledger`, a bitfield of `flags`, and three custom fields: `user_data_32`, `user_data_64`, and `user_data_128`. Transfers also include the `debit_account_id` and `credit_account_id` of the two accounts involved, and the integer `amount` transferred between them.

A single-phase transfer takes effect, or *posts*, immediately. A transfer can also be executed in two phases, represented by two transfer records. The first phase, *pending*, reserves capacity in the debit and credit account for the given amount. The second phase posts the pending transfer, transferring at most the pending amount. Pending transfers can be explicitly *voided*, which cancels them, or automatically *expire*, which is controlled by a `timeout` field. Posting and voiding transfers use a flag and a `pending_id` field to indicate which pending transfer they resolve. Pending transfers resolve at most once.

A special kind of transfer can *close* an account, preventing it from participating in later transfers. Closing transfers are always pending. Account closures can be "un-done" by voiding the closing transfer.

Accounts are immutable with five exceptions: a `closed` flag, which is derived from closing and reopening transfers, and four balance fields, which are derived from the sum of pending and posted transfers.

[3] Timestamps are derived from `CLOCK_REALTIME`, which is presumably synchronized via NTP, PTP, etc. The primary uses `CLOCK_BOOTTIME` to estimate and compensate for network latency in clock messages.

[4] Many consensus systems use a majority of nodes as a quorum. As Heidi Howard showed in 2016, Paxos can use *different* quorums for its leader election and replication phases; these two quorums must intersect, but one may be less than a majority. TigerBeetle applies this "flexible quorum" approach to Viewstamped Replication. It requires only half, not a majority, of clocks to agree.

[5] TigerBeetle's core is designed to replicate arbitrary state machines, so this may change in the future.

[6] This representation is unusual: most databases allow user-defined schemas, a variety of types, and variable-size data. However, TigerBeetle's domain is well-understood: the broad shape of financial record-keeping has not changed in centuries. Moreover, a rigid, fixed-size schema provides significant performance advantages: efficient encoding and decoding, zero-copy transfer of structures between network and disk, prefetcher/branch prediction friendliness, cache line alignment, and so on.

Transfers are always immutable. One alters or undoes a transfer by creating a new, compensating transfer.

1.3 Operations

TigerBeetle clients make *requests* to update or query database state. Each request represents a single kind of logical operation, like creating accounts or querying transfers. Requests and their corresponding responses usually involve a batch of up to 8190 *events*, all of the same type. For example, a create-transfers request includes a batch of transfers to create, and logically[7] returns a batch of results, one per transfer. Read operations generally take a list of IDs, or a query predicate, and return a batch of matching records.

From a database perspective, each TigerBeetle request is a single transaction: an ordered group of micro-operations which execute atomically. Events within a request are executed in order. Each event observes a unique, strictly increasing timestamp.[8] There are no interactive transactions, mixed read-write transactions, or indeed any kind of multi-request transactions.

TigerBeetle's home page promises Strong Serializability, and the documentation is consistent with this promise. Requests execute at most once, and events within a request "do not interleave with events from other requests." TigerBeetle also promises several session safety properties: a session "reads its own writes" and "observes writes in the order that they occur on the cluster." These are guaranteed by Strong Session Serializability, which in turn is implied by Strong Serializability.

There are two kinds of write requests. The `create_accounts` and `create_transfers` requests add a series of accounts or transfers to the database. There are also six read requests. Users look up specific accounts or transfers by ID using `lookup_accounts` and `lookup_transfers`. To query accounts or transfers matching a predicate, one uses `query_accounts`, `query_transfers`, and `get_account_transfers`. Finally, `get_account_balances` reads historical balance information.

Requests are atomic in the sense that either all or none of a request's events execute. However, specific events in a committed request can *logically* fail, returning error codes. For example, a `create_transfers` request might try to create two transfers, the first of which fails due to a balance constraint, and the second of which succeeds. This request can still commit, even though only one of its two transfers was added to the database.

To make one event contingent on another, TigerBeetle offers a sort of logical sub-transaction within a request, called a *chain*. Each event in a chain succeeds if and only if all others succeed. This allows users to express complex, multi-step transfers that succeed or fail atomically.

2 Test Design

We built a test suite for TigerBeetle using the Jepsen testing library, which combines property-based testing with fault injection. We tested TigerBeetle versions 0.16.11 through 0.16.30, including several development builds. Our tests ran on clusters of three to six[9] Debian nodes, both in LXC containers and on EC2 VMs.

TigerBeetle offers only a "smart" client which connects to every node in the cluster. These clients can mask concurrency errors by routing all requests to a single server.[10] In addition to testing this smart-client behavior, we also ran tests with each client restricted to a single node, by passing invalid addresses for the other nodes. Since TigerBeetle followers do not proxy `register` requests to the leader, most clients spend their time attempting futile requests against inexorable followers. This is fine for safety testing, so long as they time out quickly enough to keep up with leader elections.

TigerBeetle's domain-specific data model poses a challenge for validation. Jepsen has well-established tricks for checking Strict Serializability of lists, sets, and registers, but TigerBeetle has no direct analogue to these structures.

As in our 2022 work on Radix DLT, we considered interpreting each account as a list of transfers. Creating a transfer would be interpreted as a pair of appends to the debit and credit accounts. A balance read could, with the help of a constraint solver, often be mapped to a read of a specific set of transfers. However, this leaves account creation and most queries untested. It also makes it difficult to validate the rich semantics of TigerBeetle transfers. For example, TigerBeetle supports balancing transfers, which adjusts the amount of a transfer to ensure the debit (and/or credit) account maintains certain invariants, like a positive or negative balance.

Instead, we decided to take advantage of TigerBeetle's explicit total order of transactions. In broad strokes, our checker splits the problem into two interlocking parts. First, we check that the apparent timestamps of operations are Strong Serializable. Second, we

[7] For efficiency, TigerBeetle omits successful results from the actual response messages, and returns only errors, if present.

[8] Datomic, FaunaDB, and TigerBeetle are all Strong Serializable temporal databases. However, they choose varying semantics for the flow of time and effects within a transaction. Datomic evaluates the parts of a transaction concurrently, and assigns them all a single timestamp. Fauna executes them sequentially, but all operations observe a single timestamp. TigerBeetle executes sequentially *and* gives each micro-operation a distinct timestamp.

[9] Unlike most consensus systems, which use majority quorums and work best with an odd number of nodes, TigerBeetle uses flexible quorums which allows some operations to commit with (e.g.) just three out of six nodes.

[10] Imagine a write w is acknowledged by node a, but node b lags behind, such that a read sent to b would not observe w. This would be a stale read—a violation of Strong Serializability. Smart clients tend to route all requests to *either* a or b, rather than balancing requests between them. A test suite using such a client would likely miss the stale read.

check that the *semantics* of those operations, when executed in timestamp order, make sense.

2.1 Timestamp Order

Verifying timestamp order was relatively straightforward. TigerBeetle added a new client API which allowed us to read the timestamp assigned to every successful request. For operations which failed or timed out, we inferred their timestamps from the timestamp assigned to any of their effects. For instance, if the creation of account 3 timed out, but we later read account 3 with timestamp 72, we assumed that write executed at timestamp 72. TigerBeetle's promise that timestamps are strictly ordered both within and between requests means that this inference should yield an order compatible with the request timestamps. We ignored any failed reads, whether definite or indefinite—this is safe as reads have no semantic side effects.

Timestamp inference required that we eventually observe the effect of every attempted write. We divided the test into two phases: a *main phase* involving writes and reads, and a *final read phase* where we tried to read any unseen writes until TigerBeetle definitively responded "yes, this write exists", or "no, it does not (yet) exist." Our goal was to infer exactly which operations executed during the main phase, and the timestamps of those operations.[11]

If a write was observed, and its inferred timestamp fell before the timestamp of the last successfully acknowledged write,[12] we inferred that it executed during the main phase. If a write was *not* observed, we assumed that it did not execute during the main phase. There are two possible scenarios:

- TigerBeetle is Strong Serializable. If the write had executed during the main phase, Strong Serializability would have ensured its visibility during the final read phase. Our inference is correct.
- TigerBeetle is not Strong Serializable. If the write did not execute during the main phase, our inference is correct. If it *did* execute during the main phase, our inference is incorrect. We might encounter false positives or false negatives—but in either case, TigerBeetle has failed to maintain a promised invariant.

If TigerBeetle were Strong Serializable, our checker would not falsely report an error. If TigerBeetle were to (e.g.) exhibit a stale read or another violation of Strong Serializability, we might detect it indirectly. It could, for example, manifest as a model-checker error on a different operation much earlier in the history. This non-locality is not ideal, but we found it an acceptable tradeoff in exchange for excellent coverage.

Having inferred a set of operations executed during the main phase, and timestamps for each, we used Elle to construct a graph over operations. We linked operations by real-time edges when operation A ended before operation B began.[13] We also linked operations in ascending timestamp order. A violation of Strong Serializability (as far as timestamps were concerned) would manifest as a cycle in this graph. Elle checks for cycles in roughly linear time, and constructs compact exemplars of consistency violations.

2.2 Model Checking

To verify that the semantics of TigerBeetle's requests and responses were correct, we built a detailed, single-threaded model of the TigerBeetle state machine based on the documentation. This model is essentially a datatype with an initial state $init$ and a transition function $step(state,\ invoke,\ complete) \rightarrow state'$, which takes a state, the invocation of a request, and the completion of that request, and returns a new state. Illegal transitions (for instance, a read whose completion value does not agree with the state) returned a special *invalid* state. Given the inferred list of timestamp-sorted operations from the main phase, we stepped through each operation in order. Any invalid state was reported as an error.

We modeled the state as an immutable data structure including the current timestamps,[14] maps of IDs to accounts and transfers, transient errors[15], a set of indices to support efficient querying, and a few internal statistics. To model the flow of clocks, we provided each state with a pre-computed map of IDs to timestamps, derived from the reads performed during the test. Whenever one of those IDs was created, we advanced the clock to that timestamp.

The state machine is surprisingly complex, involving over 1,600 lines of Clojure and an extensive test suite. A broad array of error conditions had to be handled, including duplicate IDs, non-monotonic timestamps, balance constraints, incompatible flags, and more. Linked chains of events required speculative execution and rollback of the state—made simpler by our pure, functional approach. We made extensive use of Zach Tellman's Bifurcan, a thoroughly tested library of high performance persistent data structures.

Modeling the full state machine takes time, but allows extraordinarily detailed verification of correctness. To make checking computationally tractable,

[11] This technique does not work for imported events, where reads tell us the *imported* timestamp, rather than the *execution* timestamp. When testing imports, we used very long timeouts, and required that every operation succeed in order to check the history.

[12] We used the timestamp of the last successful write as the upper bound on the main phase. Writes may have been executed during the final read phase (e.g. due to network delays), but we ignored them for safety checking.

[13] For efficiency, we actually computed a transitive reduction of the real-time dependency graph.

[14] TigerBeetle has three internal timestamps that constrain clock values: the "current" timestamp, and two separate clocks for imported accounts and transfers, whose timestamps are still monotonic, but lag behind the current time.

[15] TigerBeetle guarantees that certain classes of errors, called *transient errors*, ensure that a transfer will *always* fail, even if resubmitted under conditions where it would otherwise succeed. These errors are transient in the sense that they are caused by (potentially) short-lived conditions in the database state, but persistent in the sense that the database must remember them for all time.

typical Jepsen tests use only a handful of carefully selected data types and operations on them. The implicit assumption is that if the database's concurrency control protocol handles that selected example correctly, it is likely correct for other workloads as well. Modeling the state machine in detail allowed us to check almost[16] every[17] operation TigerBeetle can perform. We verified that observed results of queries matched exactly, down to specific error codes. As discussed in this report, this approach found bugs we would have otherwise missed.

2.3 Generating Operations

The downside of testing so much of TigerBeetle's state machine is that we must then generate requests which *exercise* it. Generating syntactically valid requests is easy, but generating requests which often succeed, or queries which return non-empty results, is surprisingly hard.

Our generator maintained extensive in-memory state throughout each test, including probabilistic models of which account and transfer IDs were likely to exist, which transfers were likely pending, what timestamps were likely extant, and what each worker process was currently doing. The generator updated this state with each operation's invocation and completion.

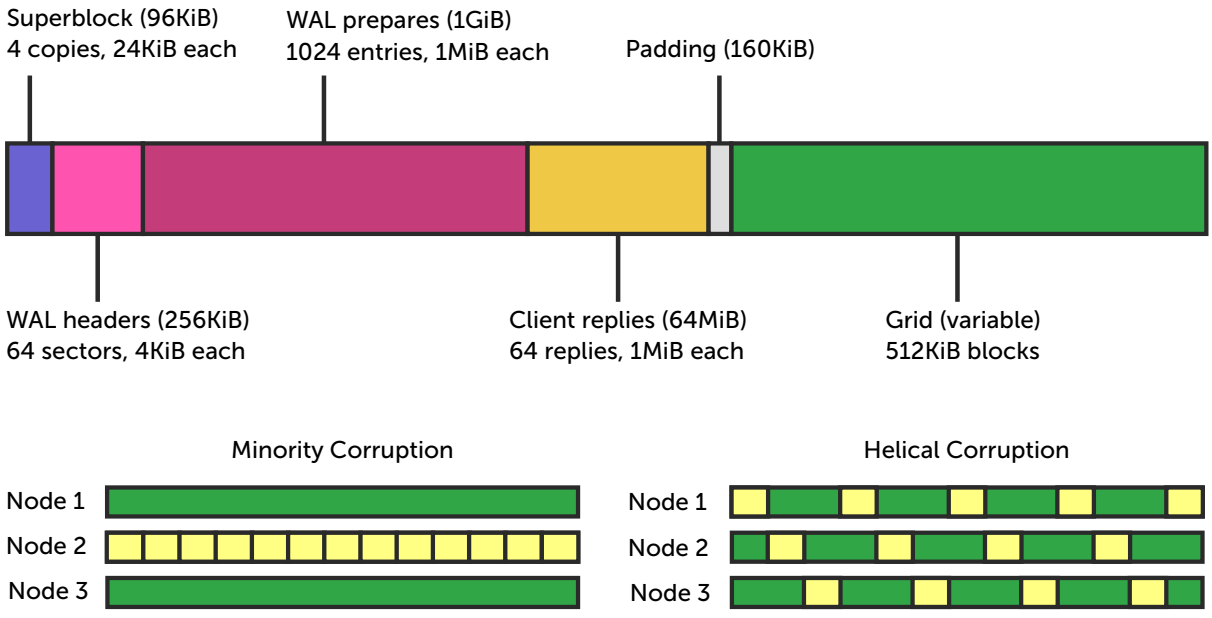
We selected Zipfian distributed IDs, ledgers, codes, and so on, ensuring a mix of very hot and very cool objects. We used a broad set of parameters to guide stochastic choices of request types, account and transfer IDs, chain lengths, flags, queries, and probabilistic state updates. These parameters were carefully tuned across a variety of concurrencies, request rates, hardware environments, and fault conditions to find a reasonable balance of successes and failures, non-empty query results, attempted invariant violations, and so on.

2.4 Fault Injection

Jepsen provides several kinds of faults "out of the box." We stressed TigerBeetle with process crashes (`SIGKILL`), pauses (`SIGSTOP`), a variety of transitive and non-transitive network partitions, and clock changes ranging from milliseconds to hundreds of seconds, as well as strobing the clock rapidly back and forth. We also upgraded nodes through several versions during tests.

We also introduced a variety of storage faults via a new file corruption nemesis. We flipped random bits to simulate (e.g.) cosmic ray interference. We replaced chunks of the file with other chunks, in an attempt to simulate misdirected writes. We also took snapshots of chunks of the file, then restored them later, to simulate lost writes.

Each TigerBeetle node has a single data file, which is divided into *zones* at predictable offsets. Each zone stores a single kind of fixed-sized record. We scoped our faults to specific zones—corrupting, for example, only the write-ahead-log (WAL)'s headers, or restoring a snapshot of just one of the four redundant copies in the superblock zone. In many tests we corrupted multiple zones, or the entire file.



[16] Our strategy does require that a single ID is never written twice. We complemented the main workload with a dedicated idempotence workload which verifies that duplicate attempts to write the same data never succeed, and never lead to divergent values for the same ID.

[17] TigerBeetle includes an automatic timeout mechanism for pending writes, but timeouts are not exactly deterministic, which makes it difficult to model-check.

We also targeted a variety of nodes for file corruption. In one scenario, we corrupted data throughout (e.g.) the superblock, but only on a minority of nodes. In a second scenario, we corrupted every node's data, but selected different chunks of the file for each node. For example, one node in a three-node cluster might corrupt the first, fourth, seventh, and tenth chunks of the grid; another would corrupt the second, fifth, eighth, and so on. We called this a *helical* disk fault. If you imagine arranging the cluster's nodes into a ring, and drawing their file offsets along the ring's symmetry axis, the corrupted chunks "spin" around the ring, forming a helix. Because TigerBeetle's file layout is (generally speaking) bit-for-bit identical between up-to-date replicas, this avoids corrupting any single record in the database beyond repair.[18]

3 Results

3.1 Requests Never Time Out (#206)

Our first tests of TigerBeetle routinely stalled forever. For example, in this test run the very first request never returned, which prevented the test from ever completing. This turned out to be a consequence of an unusual design decision: TigerBeetle actually guaranteed that requests would never time out:

> Requests do not time out. Clients will continuously retry requests until they receive a reply from the cluster. This is because in the case of a network partition, a lack of response from the cluster could either indicate that the request was dropped before it was processed or that the reply was dropped after the request was processed.

The session documentation reaffirmed this stance: a TigerBeetle client "will never time out" and "does not surface network errors". This is particularly surprising since most systems do expose network errors, whether Strong Serializable or otherwise.

> With TigerBeetle's strict consistency model, surfacing these errors at the client/application level would be misleading. An error would imply that a request did not execute, when that is not known[.]

There are, broadly speaking, two classes of errors in distributed systems. A *definite* error, like a constraint violation, signifies that an operation has not and will never happen. An *indefinite* error, like a timeout, signifies that the operation may have already happened, might happen later, or might never happen.[19] Consistent with the documentation, TigerBeetle tries to conceal both kinds of error through an unbounded internal retry loop.

However, TigerBeetle clients actually *can* produce timeout errors. The Java client's asynchronous methods, like `createTransfersAsync`, return `CompletableFutures`. `CompletableFuture` usually represents operations, like network requests, which are subject to indefinite failures. Indeed, timeouts are integral to the datatype: one awaits a future using `.get(timeout, timeUnit)`, or wraps it in `.orTimeout(seconds, timeUnit)` to throw a timeout automatically. Likewise, the .Net client's `createTransferAsync` and friends return `Task` objects which offer timeout-driven `Wait()` methods.

Even if users constrain themselves to synchronous calls, applications rarely have unbounded time to run. It seems likely that applications will wrap TigerBeetle calls in their own timeouts. If they do not, the application may eventually terminate, which is a worse kind of indefinite failure. Even when applications can wait, their clients (or the human beings waiting for an operation), may give up at any time. The challenge of indefinite errors is intrinsic to asynchronous networks and cannot be eliminated.

Because TigerBeetle clients handle all failures through a silent internal retry mechanism, they unnecessarily convert definite errors into indefinite ones. For example, imagine a common fault: a TigerBeetle server has crashed. An application makes a `createTransfer` request. Its client attempts to open a TCP connection to submit the request, and receives `ECONNREFUSED`. The client knows internally that this request cannot possibly have executed: it has a definite failure. However, it refuses to inform the caller, and instead retries again and again. The caller's only sign of an error is that the client appears to have stalled. When the caller times out or eventually shuts down, that definite failure becomes indefinite. Instead of making indefinite errors impossible, TigerBeetle's client design *proliferates* them.

This is an ongoing discussion within TigerBeetle (#206). Jepsen recommended that TigerBeetle develop a first-class representation for definite and indefinite errors, and return those errors to callers when problems occur. It is perfectly fine to keep automatic retries—perhaps even unbounded ones—but this behavior should be configurable. TigerBeetle clients should take options controlling the maximum time allowed for opening a connection, and for awaiting responses from a submitted request. Users can request unbounded timeouts if desired.

[18] One notable exception to this rule is the WAL. The write-ahead log is built as a ring buffer. The head of the write-ahead log is critical: if the head of the WAL is corrupted on one node, that node cannot trust its own data file and must ask the other nodes to help it repair the damage. Because some nodes may lag behind others, it is possible that the head of the WAL could be at different file offsets on different nodes. A helical fault could corrupt the head on a majority of nodes, preventing the cluster from recovering.

[19] To be clear: a definite failure can be retried, and that retry operation might succeed. When we say a definite error means an operation will "never happen," we refer to the original operation, not retries.

3.2 Client Invalid Pointer Dereference (#2435)

Because synchronous client operations never timed out, our early tests generally failed to terminate. To avoid this problem, we tried wrapping calls to TigerBeetle clients in two kinds of timeouts. In the first, we spawned a new thread to make a synchronous call, and interrupted that thread if it did not complete within a few seconds.[20]

```
(let [worker (future (.createAccounts
                       client accounts))
      ret    (deref worker 5000 ::timeout)]
  (if (= ret ::timeout)
    (do (future-cancel worker)
        (throw {:type :timeout}))
    retval))
```

In 0.16.11, this immediately segfaulted the entire JVM. TigerBeetle's Java client is implemented via a JNI binding to a client library written in Zig, and a Zig panic crashes the JVM as well. Concerned that our use of multiple threads or a thread interrupt might be at fault, we tried an alternate approach, using the client's asynchronous methods which returned a `CompletableFuture`. If that future did not produce a result within a few seconds, we closed the client.

```
(let [future (.createAccountsAsync
               c (account-batch accounts))
      ret    (deref future 5000 ::timeout)]
  (if (= ret ::timeout)
    (do (close! client)
        (throw+ {:type :timeout}))
    ret))
```

—this too segfaulted the JVM.

In a closely related issue, calling `client.close()` in 0.16.11, on a freshly opened client, caused the JVM to panic with a `reached unreachable code` error in `tb_client.zig:122`.

TigerBeetle traced these problems to unset fields in the client's request data structure (#2435). These fields were normally initialized during request submission. However, if the client was closed between request creation and submission, it would dereference the default `0xaaa...` address: a Zig language default. This issue was fixed in 0.16.12.

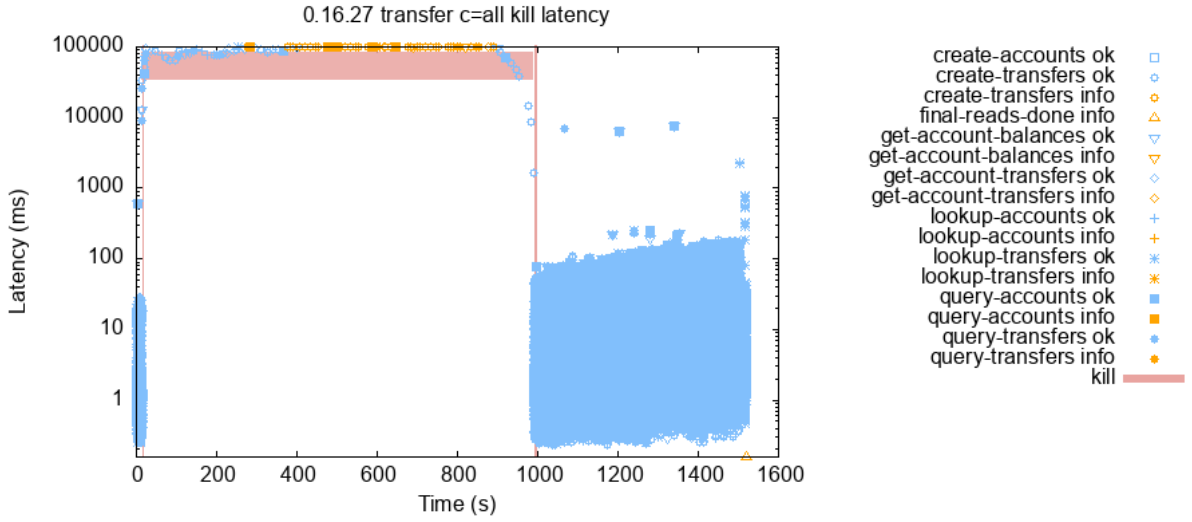
3.3 Client Crash on Eviction (#2484)

The official TigerBeetle clients crashed the entire process when a server informed them that their session had been evicted. TigerBeetle allows only 64 concurrent sessions by default, making it relatively easy to hit this limit.[21] TigerBeetle also evicts clients which use a newer client version than the server.

This behavior made it impossible for clients to cleanly recover from eviction, or to back off under load. TigerBeetle changed this behavior in #2484. As of 0.16.13, clients return errors to their callers on eviction, rather than crashing the entire process.

3.4 Elevated Latencies on Single-Node Faults (#2739)

When a single node failed, we often saw client latencies jump by three to five orders of magnitude. In this test of a five-node cluster, with clients constrained to a single node each, killing a single node caused minimum latencies to rise from less than one millisecond to ten seconds. There were fluctuations down to one second, but in general elevated latencies lasted for the full duration of a fault.

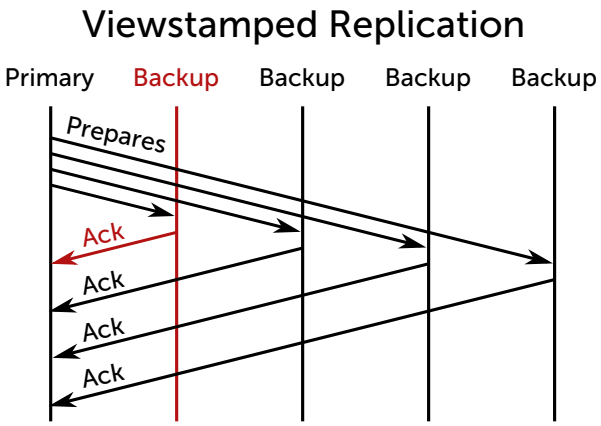


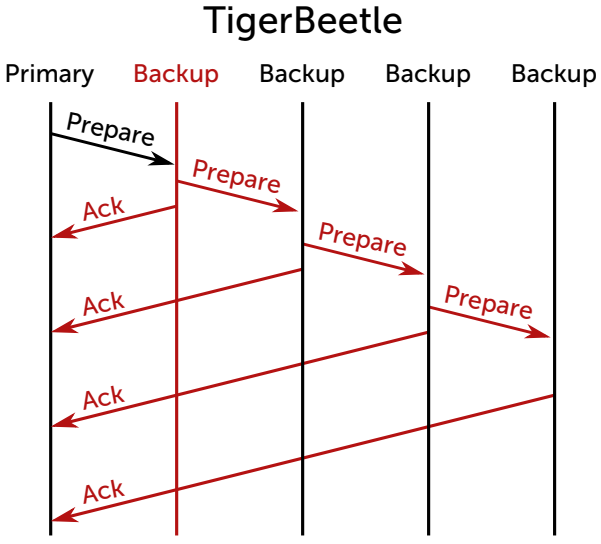
[20] This is Jepsen's standard timeout macro, used when clients don't time out reliably on their own. For clarity, we've omitted some error handling.

[21] This low session limit is an intentional design choice: TigerBeetle benefits from large batches of requests, and enforcing a smaller number of clients nudges users towards designs which can perform client-side batching efficiently. One imagines a PgBouncer-style proxy might also come in handy here.

Under higher load, the situation could become considerably worse. Consider this test run with a three-node cluster, where each client was allowed to connect to all three nodes. A few seconds into the test, we killed node `n3`. This drove latencies on every client from between 1–50 milliseconds to roughly a hundred seconds per request. This situation persisted for almost a thousand seconds, until we restarted `n3`.

In the original Viewstamped Replication and Viewstamped Replication Revisited, a primary sends a *prepare* message to every secondary when it wants to perform an operation. The secondaries send acknowledgements back to the primary, which can commit once f replicas have acknowledged. The failure of any single node (shown in red) causes a single acknowledgement to be lost, but does not affect any of the other nodes or their acknowledgements. The system as a whole is relatively insensitive to single-node failures.





TigerBeetle approaches prepares differently. Nodes are arranged in a ring, and the primary sends a single prepare message to the next secondary in the ring. That secondary sends a prepare to the following secondary, and so on, until all nodes have received the message. Acknowledgements are sent directly to the primary. This approach reduces the bandwidth requirements for any single node, but creates a weakness: if the primary must receive f acknowledgements to commit, the failure of any one of the next f replicas in the ring will prevent commit entirely. The effect of a single-node failure cascades through the rest of the ring. We opened issue #2739 to track this issue.

Version 0.16.30 includes a new tactic to mitigate this problem. Sending every other prepare message *backwards* around the ring allows half of prepares to bypass the faulty node. Since prepares must be processed in order, the replicas which receive these counter-ringward prepares must repair the ringward prepare messages they missed. Repairing takes time, but the overall effect is significant. Rather than hundred-second latencies during a single-node fault, 0.1.16 delivered 1–30 second latencies in our tests.

After our collaboration, TigerBeetle continued work on single-node fault tolerance, adding a new series of deterministic performance tests to their simulation framework. As of version 0.16.43, TigerBeetle includes a host of performance improvements. Nodes replicate in both directions around the ring, which reduces latencies and the impact of single failures. The ring topology is now dynamic: and the cluster continually adjusts the order of nodes to minimize latency based on network conditions and faults.

3.5 Incorrect Header Timestamps in Java Client (#2495)

To support Jepsen's tests, TigerBeetle added a new, experimental API in 0.16.13 for obtaining the execution timestamp for each request from a header included in response messages. In the Java client, this API routinely returned incorrect and duplicate timestamps. For example, both of these `create-transfers` operations returned identical timestamps.

```
{:index     5827,
 :type      :ok,
 :process   1,
 :f         :create-transfers,
 :value     [:ok],
 :timestamp 1736185975365035812}
{:index     5829,
 :type      :ok,
 :process   11,
 :f         :create-transfers,
 :value     [:ok :ok :ok :ok],
 :timestamp 1736185975365035812}
```

TigerBeetle traced this bug to a mutable singleton response object in the Java client: `Batch.EMPTY`. Every empty response used the same instance of this object, updating its header to reflect that response's timestamp. As responses overwrote each other's headers, callers observed incorrect timestamps. Since TigerBeetle represents entirely-successful responses as empty batches, this happened quite often.

This bug (#2495) was fixed in 0.16.14, just seven days after 0.6.13's release. It did not affect the correctness of the actual data involved, only request timestamps from the Java client's header API. We believe the impact to users was likely nil.

3.6 Missing Query Results (#2544)

In version 0.16.13, responses for `query_accounts`, `query_transfers`, and `get_account_transfers` routinely omitted some or all results.[22] Missing results were always at the end: each response was a (possibly empty) prefix of the correct results. This behavior occurred frequently in healthy clusters. For example, take this test run, where 281 seconds into the test, a client called `query_transfers` with the following filter:

```
{:flags  #{:reversed}
 :limit  9
 :ledger 3
 :code   289}
```

This query returned a single result:

```
[{:amount             34N,
  :ledger             3,
  :debit-account-id   3137N,
  :pending-id         0N,
  :credit-account-id  1483N,
  :user-data          9,
  :id                 327610N,
  :code               289,
  :timeout            0,
  :timestamp          1733448783658756894,
  :flags              #{:linked}}]
```

However, our model expected eight additional transfers which TigerBeetle omitted. For instance, transfer `326112` had ledger 3 and code 289, and was successfully acknowledged five seconds before this query began. It should have been included in these results, but was not.

```
{:amount             21,
 :ledger             3,
 :debit-account-id   123076N,
 :pending-id         0N,
 :credit-account-id  51358N,
 :user-data          2,
 :id                 326112N,
 :code               289,
 :timeout            0,
 :timestamp          1733448782536800935,
 :flags              #{}}
```

Note that this query asked for transfers matching both `ledger = 3` and `code = 289`. Queries which filtered on only a single field did not exhibit this problem. TigerBeetle traced the cause to a bug in the zig-zag merge join between multiple indices (#2544). When traversing an index, a bounds check prevented scanning the same chunk of records twice. During a join between two indices, the scans informed one another that some records can be safely skipped. This process could push the highest (or lowest) key outside the bounds check in the wrong direction, causing the scan to terminate early. The issue was fixed in version 0.16.17.

This bug went undetected by all four TigerBeetle fuzzers which perform index scans. Two fuzzers, `fuzz_lsm_tree` and `fuzz_lsm_forest`, did not perform joins. The other two, `vopr` and `fuzz_lsm_scan`, generated objects which happened to appear consecutively in each index—the "zig-zag" part of the merge join was never executed. Rewriting the scan fuzzer to generate unpredictable objects helped it reproduce this bug quickly.

3.7 Panic! At the Disk 0 (#2681a)

Occasionally, tests with single-bit file corruption in the superblock, WAL, or grid zones caused TigerBeetle 0.16.20 to crash on startup. The process would print `panic: reached unreachable code`, then exit.[23]

These crashes were caused by three near-identical bugs in checking sector padding. For example, each superblock header in TigerBeetle's data file contains an unused padding region normally filled with zeroes. Similarly, entries in the WAL and grid blocks may have zero padding bytes at the end. TigerBeetle's checksums cover the data stored in each chunk, but exclude the padding. If a bit in the padding flipped from zero to one, the chunk's checksum would still pass. Then, TigerBeetle checked to make sure the padding bytes were still zeroed. When it encountered the flipped bit, that failed assertion caused the server to crash. This is perhaps worth logging, but damage to padding bytes does not compromise safety. The corrupted padding could be re-zeroed or repaired from other replicas.

TigerBeetle's internal testing with the VOPR did not discover this bug because it corrupted entire sectors, rather than single bits. Corrupting a sector caused the checksum to fail and triggered the repair process. The zero-padding assertion was never reached! TigerBeetle revised the VOPR (#2681) to introduce single-byte errors, which reproduced the bugs. As of 0.16.26, TigerBeetle repairs sectors with corrupt padding, instead of crashing.

3.8 Panic Due to Superblock Bitflips (#2681b)

In a closely related bug, TigerBeetle could crash with an identical `panic: reached unreachable code` error, when we flipped bits in the superblock's region rather than padding. Each of the superblock's four copies includes a unique two-byte `copy` number, so that TigerBeetle can detect if a write or read of the superblock was misdirected by the disk. However, each copy of the superblock was supposed to have an identical checksum. Those checksums therefore skipped over the copy number.

When writing a superblock back to disk, TigerBeetle checked to make sure (#2681) that the copy number was between 0 and 3. If the copy number had been corrupted on disk, and that corrupted version

[22] This also occurred with `get_account_balances`, but our test harness didn't yet cover that API call.

[23] All TigerBeetle assertion failures printed `reached unreachable code` then exited—there was no error message to tell them apart. Debugging builds offered a stacktrace.

read into memory, that assertion would fail at write time, causing a panic. TigerBeetle resolved this in version 0.16.26 by resetting the copy number, rather than crashing.

3.9 Checkpoint Divergence on Upgrade (#2745)

When testing upgrades from 0.16.25 and before to 0.16.26 and higher, we observed repeated TigerBeetle crashes with log messages like `panic: checkpoint diverged`. For example, this one-minute test upgraded a five-node cluster from 0.16.25 to 0.16.26, with no other faults. Node n5 detected the new binary at 21:48, and switched to executing 0.16.26 at 22:01. Immediately after starting, it panicked at `replica.zig:1766`:

```
2025-02-13 21:22:06.159Z error(replica):
4: on_prepare: checkpoint diverged (op=23040
expect=3779fc8a6a13bf5cf9f995b8895c2609
received=05383d884c680d15e726071358854f67
from=2)
thread 227936 panic: checkpoint diverged
```

TigerBeetle traced this crash to a change in the `CheckpointState` structure in 0.16.26. Between checkpoints, TigerBeetle tracks a set of released blocks in the file. In 0.16.26, TigerBeetle changed when that set was flipped to empty. The old version of `CheckpointState` did not need to track released blocks, because that set was always empty at checkpoint time. The new version included released blocks. To ensure older replicas could still sync state from newer ones, nodes running 0.16.26 could send both the old and new versions of `CheckpointState`. This allowed a node running 0.16.26 to send a backwards-compatible `CheckpointState` with an empty set of released blocks to a node running 0.16.25. If that node then restarted on 0.16.26, it would be missing the released blocks which other replicas knew about. Thankfully, the assertion detected this divergence and crashed the node, rather than allowing clients to observe inconsistent data.

We found this bug after several later versions had already been released. Because it requires state synchronization, rather than the normal replication path, we believe healthy and non-lagging clusters shouldn't encounter this bug. The impact should also be limited to a minority of nodes. Based on these factors, and a lack of test coverage for upgrades in general, TigerBeetle opted not to release a patched version of 0.16.26. Instead, the team updated the changelog (#2745) to inform customers of the hazard. Operators should pause all clients and wait for replicas to catch up before upgrading to (or past) 0.16.26.

3.10 Panic in `release_transition` on Multiple Upgrades (#2758)

In upgrade tests from 0.16.16 to 0.16.28, we found that TigerBeetle could crash with an assertion failure in `replica.zig`'s `release_transition` function. This happened when we executed multiple upgrades within ~20 seconds of one another, or when nodes crashed or paused during the upgrade process. We could reproduce this bug reliably–with process pauses, it manifested a few times per minute.

TigerBeetle traced this problem to an over-zealous assertion in the upgrade code (#2758).

Imagine a node currently runs version A, and an operator replaces its binary with version B. The node detects that version B is available, opens the new binary with a `memfd`, and uses `exec()` to replace the running process with that new code. Meanwhile, an operator replaces the binary with version C. The replica starts up with B, and as a safety check, asserts that the binary on disk (not the `memfd`!) has version header B. This assertion fails, since the binary is actually version C.

TigerBeetle resolved this issue in 0.16.29 by replacing the assertion with a warning message; running a different version than the binary on disk does not actually break safety.

3.11 Panic on Deprecated Message Types (#2763)

We encountered another occasional crash in the upgrade from 0.16.26 to 0.16.27. In this test run, two nodes crashed shortly after the upgrade. Both logged `panic: switch on corrupt value`, originating in `message_header.zig`'s `into_any` function. The crashed nodes recovered after a restart.

This crash was caused by a switch expression which dispatched based on the type of a message. Prior to 0.16.28, Tigerbeetle removed deprecated message types from these switch expressions. An older node could send a network message of a type that the newer node would have no corresponding `switch` case for. TigerBeetle resolved the issue in version 0.16.29 by adding the deprecated message types back into the switch statements, and simply ignoring them.

3.12 No Recovery Path for Single-Node Disk Failure (#2767)

TigerBeetle offers exceptional resilience to data file corruption. However, disk failure, fires, EBS volume errors, operator errors, and more can cause a node to lose its entire data file, or to corrupt that data beyond repair. TigerBeetle is fault tolerant and can continue running safely with a minority of nodes offline. However, failed nodes do need to be replaced eventually, and most distributed systems have a mechanism for doing so. In systems which support membership changes, the best path is often to add a new, replacement node to the cluster, and to remove the failed node. Others have dedicated replacement procedures.

TigerBeetle, surprisingly, has no story for replacing a failed node. The documentation says nothing on the matter. There is an undocumented recovery

procedure: users can run `tigerbeetle format` to reinitialize the node with an empty data file, and allow TigerBeetle's automatic repair mechanisms to transfer the data from other nodes. Since our tests often damaged data files beyond repair, we used this reformat approach regularly.

Reformatting nodes works most of the time, but as TigerBeetle explained to Jepsen, it may be unsafe. For example, imagine a committed operation `op` is present on two out of three nodes, and one of those nodes is reformatted. The cluster now has a two-thirds majority which can execute a view change *without* observing `op`; the operation is then lost. In our testing, data loss was infrequent, and limited to just a few operations. For example, this run lost five acknowledged transfers which were created in two separate requests. Another problem arises when nodes are upgraded. If a node is formatted using a newer binary, but the cluster has not yet completed the transition to that version, the node will crash on startup during `replica.zig/open`.

TigerBeetle had been aware of this issue #2767 for some time, and planned to add a safe recovery path for the loss of a node. However, it took time to design, build, and document. After our collaboration, TigerBeetle completed this work. Version 0.16.43 incorporates a new `tigerbeetle recover` command to recovera node which has suffered catastrophic data loss.

№	Summary	Event Required	Fixed in
206	Requests never time out	None	Unresolved
2435	Client uninitialized memory access on close	Interrupt or close a client	0.16.12
2484	Client crash on eviction	Newer, or too many, clients	0.16.13
2739	Elevated latencies on single-node fault	Pause or crash	0.16.43
2495	Incorrect header timestamp from Java client	None	0.16.14
2544	Missing query results	None	0.16.17
2681a	Panic on bitflips in chunk padding	Bitflip and restart	0.16.26
2681b	Panic on bitflips in superblock copy number	Bitflip and restart	0.16.26
2745	Checkpoint divergence	0.16.26 upgrade during sync	Documented
2758	Panic in `release_transition` on upgrades	Upgrades in quick succession	0.16.29
2763	Panic on deprecated message types	Upgrade	0.16.29
2767	No recovery path for single-node disk failure	Single-node disk failure	0.16.43

4 Discussion

We found two safety issues in TigerBeetle.[24][25] Prior to version 0.16.17, TigerBeetle often omitted results from queries with multiple filters, even in healthy clusters. We also found a very minor issue in which a debugging API in the Java client, added specifically for our tests, returned incorrect and duplicate timestamps for operations. As of 0.16.26 and higher, our findings were consistent with TigerBeetle's claims of Strong Serializability—one of the strongest consistency models for concurrent systems. TigerBeetle preserved this property through various combinations of process pauses, crashes, network partitions, clock errors, disk corruption, and upgrades.

We also found seven crashes in TigerBeetle. Two affected the Java client: an uninitialized memory access caused by a shared mutable data structure, and a design choice to crash the entire process when a server evicted a client. Both were fixed by 0.16.13. Five involved servers: two panics on disk corruption, and three more involving upgrades. All crashes were resolved by 0.16.29, with the exception of #2745, which is now documented.

We found some surprising performance and availability issues in TigerBeetle. Server latencies jumped dramatically when even a single node was unavailable. This behavior is unusual—most consensus systems are relatively insensitive to single-node failures—and stemmed from a design choice to replicate data in a ring, rather than broadcasting from the primary to all backups directly. This behavior was somewhat improved in 0.16.30, but still quite noticeable. After our collaboration, TigerBeetle extended their simulation tests to measure performance under various faults, and used those tests to drive extensive improvements in 0.16.43. The ring topology now continually adapts to observed latencies, and messages are broadcast in both directions around the ring. We believe these improvements should significantly mitigate the latency impact of failures.

TigerBeetle also lacked a safe path to recover a node which had suffered catastrophic disk failure. After our collaboration, TigerBeetle built a new recovery command, which is available as of 0.16.43.

Only one issue remains unresolved. By design, client requests are retried forever, which complicates error handling. TigerBeetle plans to address this, but the work will take some time.

[24] Issue 2745 is in some sense both a safety and liveness issue. Replicas disagree on which blocks are free, violating a key safety property in TigerBeetle's design: replicas should have identical on-disk state. However, a defensive assertion converts this safety violation to a crash, which prevents users from observing the divergence. In this sense it is a liveness issue, and we report it as such.

[25] For the formal verification enthusiasts in the crowd: yes, recoverable crashes, transient availability issues, and high latency are all technically safety issues, in that they involve finite counterexamples.

We recommend users upgrade to 0.16.43, which addresses all but one of the issues reported here. Users should exercise particular caution during the upgrade to (or past) 0.16.26; consult the release notes. We also suggest that users simulate single-node failures in a testing environment, and measure how their application responds to elevated latencies.

TigerBeetle exhibits a refreshing dedication to correctness. The architecture appears sound: Viewstamped Replication is a well-established consensus protocol, and TigerBeetle's integration of flexible quorums and protocol-aware recovery seem to have allowed improved availability and extreme resilience to data file corruption. Integrating these protocols does not appear to have compromised the key invariant of Strong Serializability. Most of our findings involved crashes or performance degradation, rather than safety errors. Moreover, several of those crashes were due to overly cautious assertions.

We attribute this robustness in large part to TigerBeetle's extensive simulation, integration, and property-based tests, which caught a broad range of safety bugs both before and during our engagement. As we brought new issues to the TigerBeetle team, they quickly expanded their internal test suite to reproduce them. We are confident that TigerBeetle's investment in careful engineering and rigorous testing will continue to pay off, and we're excited to see these techniques adopted by more databases in the future.

As always, we caution that Jepsen takes an experimental approach to safety verification: we can prove the presence of bugs, but not their absence. While we make extensive efforts to find problems, we cannot prove correctness.

4.1 Disk Faults

TigerBeetle offers exceptional resilience to disk faults. In our tests, it recovered from bitflips and other kinds of file corruption in almost every part of a node's data file, so long as corruption was limited to a minority of nodes. In some file zones, like the grid, TigerBeetle tolerated the loss or corruption of all but one copy. In the superblock, client replies, and grid zones of the data file, TigerBeetle could recover our "helical" faults, in which every node experienced data corruption spread across disjoint regions of the file.

As previously noted, bitflips in the superblock copy number, or in various zero-padding regions, could cause TigerBeetle to crash. These issues have been resolved as of 0.16.26.

Rolling back all four copies of a node's superblock to an earlier version can permanently disable a TigerBeetle node; it will crash upon detecting WAL entries newer than the superblock. TigerBeetle considers this outside their fault model, and we concur. Given that TigerBeetle performs four separate, sequential writes of the superblock and reads them back to confirm their presence, it seems remarkably unlikely to encounter this by accident.

Helical faults in the WAL can permanently disable a TigerBeetle cluster. The most recent "head" entry of the WAL is critical, and since some nodes may lag behind others, they may have heads at different file offsets. In our tests, helical faults often corrupted the head of the WAL on a majority of nodes, rendering the entire cluster unusable.

When a node's disk file is lost or corrupted beyond repair, TigerBeetle currently has no safe path for recovery. We recommend users exercise caution when reformatting a failed node. Avoid upgrades when a node is down, and try to reformat a node only when the remainder of the cluster appears healthy. If possible, pause clients and check node logs before upgrading: none should be logging sync-related messages.

4.2 Retries

Users should think carefully about the official TigerBeetle clients' retry behavior. By default, clients retry operations forever. Synchronous operations will never time out; you may need to implement your own timeouts. The futures returned by asynchronous calls do offer APIs with timeouts, but the client will continue retrying those operations forever. Long-lasting unavailability could cause TigerBeetle clients to consume unbounded memory as they attempt to buffer and retry an ever-growing set of requests.

This retry behavior flattens definite and indefinite failures into indefinite ones: *everything* becomes a timeout. Contrary to TigerBeetle's documentation, indefinite network errors are very much possible. Indeed, they are more likely in TigerBeetle than in systems which return distinct network errors! Moreover, TigerBeetle users may find it more difficult to implement (e.g.) exponential backoff or load-shedding circuit breakers: in order to abandon a single request, the entire client must be torn down.

Jepsen recommends that users carefully consider and test their timeout behavior during faults. We also suggest TigerBeetle introduce at least two kinds of error, so users can distinguish between definite and indefinite faults. Finally, clients should take configurable timeouts, so users can bound their time and memory consumption.

4.3 Crashing as a Way of Life

TigerBeetle prizes safety, and employs defensive programming techniques to ensure it. In addition to carefully designed algorithms and extensive testing, both client and server code are full of assertions which double-check that intended invariants have been preserved. Assertion failures crash the entire program to preserve safety. When this happens, clients or servers may be partially or totally unavailable, sometimes for minutes, sometimes permanently. Many of our findings involved an assertion which turned what *would* have been a safety hazard into a simple crash: a welcome tradeoff for safety-critical systems.

This is a sensible approach. Complex systems ensure safety through an interlocking set of guards: each guard screens out errors the others might have missed. Abandoning possibly-incorrect execution is also a core tenet of Erlang/OTP's "let it crash" ethos. However, TigerBeetle's approach is not without drawbacks.

First, several of the crashes we found in this report were due to overly conservative assertions. For instance, prior to 0.16.26, TigerBeetle crashed on encountering non-zero bytes in unused padding regions on disk. Safety was never endangered by these errors, but the crashes compromised availability—and could push users into the dangerous recovery path of reformatting.

Second, in 0.16.11, TigerBeetle's client library forcibly crashed the entire application process when a client used a newer version than the server, or when the server simply had too many connections. These are errors that a well-designed application can and should recover from—for instance, through an exponential backoff and retry system, or by coordinating with other clients. Crashing the process, instead of returning an error code or throwing an exception, denies the application the ability to make these mitigations.

In Erlang, "let it crash" means more than simply abandoning computation early. It is integrally linked with Erlang's actor model, which allows actors to crash independently of one another. It also relies on Erlang's *supervisor trees*: every actor has a supervisor which is notified of a crash and can restart the failed computation. In TigerBeetle, the failure domain is the entire POSIX process, and the supervisor, where one exists, is something like the init system or Kubernetes. These supervisors generally lack visibility into *why* the crash happened or how to recover, and they are often not equipped to adapt to changing circumstances. They may restart the process over and over again, crashing every time. On repeated crashes, they may give up on the process forever.

Despite these limitations, we feel TigerBeetle makes a reasonable compromise. TigerBeetle is intended for financial systems of record where integrity is key, and overly-cautious assertions can be fixed easily as they arise. Those assertions also help to experimentally validate and guide the mental models of engineers working on TigerBeetle. TigerBeetle's clients have shifted more towards returning error codes, rather than crashing outright.

More generally, we encourage engineers to think about fault domains when designing error paths. Ask, "If we must crash, how can we keep a part of the system running?" And after a crash, "How will that part recover?" This is especially important for client libraries, which are guests in another system's home.

4.4 Future Work

TigerBeetle includes a timeout mechanism for pending transfers. We do not know how to robustly test this system, since timeouts may, by design, not void a transfer until some time after their deadline has passed. We would like to revisit timeout semantics with an eye towards establishing quantitative bounds.

During the course of this research, Jepsen, TigerBeetle, and Antithesis collaborated to run Jepsen's TigerBeetle test suite within Antithesis's environment—taking advantage of Antithesis' deterministic simulation, fault injection, and time-travel debugging capabilities. These experiments are still in the early stages, but could lay the groundwork for a powerful, complementary analysis of distributed systems.

Multi-version systems are also devilishly hard to pull off. While TigerBeetle already had excellent test coverage for single versions, they lacked fuzz tests for cross-version upgrades. Our tests found several issues in the upgrade process, and TigerBeetle plans to expand their testing of upgrades in the future. Similarly, membership changes in distributed systems are notoriously challenging, and currently unimplemented in TigerBeetle. As TigerBeetle builds support for adding and removing nodes, we anticipate a rich opportunity for further testing.

Finally, TigerBeetle's approach to retries has been the subject of ongoing discussion, and redesigning their approch will take time. We anticipate further work towards a robust client representation of errors.

This work would not have been possible without the invaluable assistance of the TigerBeetle team, including Fabio Arnold, Rafael Batiati, Chaitanya Bhandari, Lewis Daly, Joran Dirk Greef, djg, Alex Kladov, Federico Lorenzi, and Tobias Ziegler. Our thanks to Ellen Marie Dash for helping write the new file-corruption nemesis used in this research. We are grateful to Irene Kannyo for her editorial support. This report was funded by TigerBeetle, Inc. and conducted in accordance with the Jepsen ethics policy.