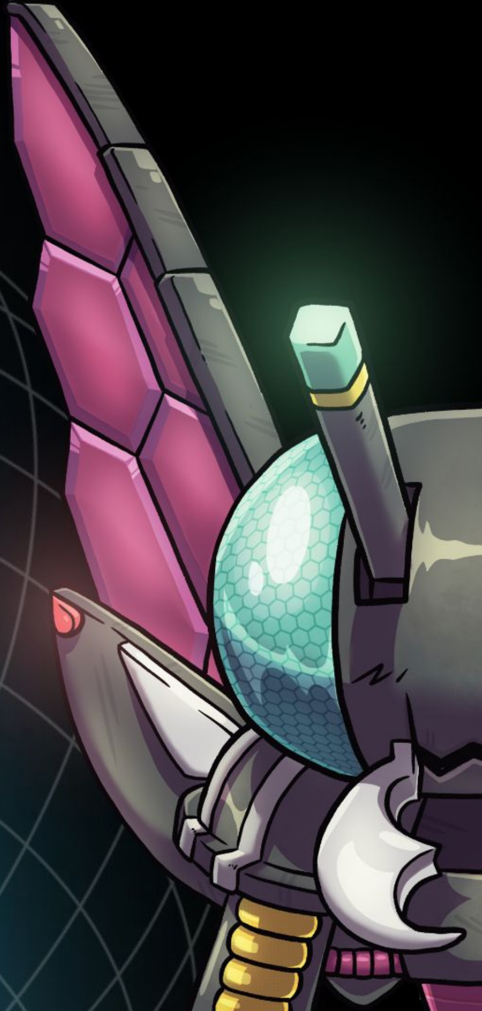
TigerBeetle

1000X WORLD TOUR

13 CITIES | 6 DAYS



Mama Said...
"I ♥ Coffee"

ROLEX
Emirates Air

BOOKS NOT BOMBS
Marshall

TigerBeetle

How I Downgraded a Safety Bug to an Availability bug… using Assertions!

December 4th, 2025

Consensus Protocols

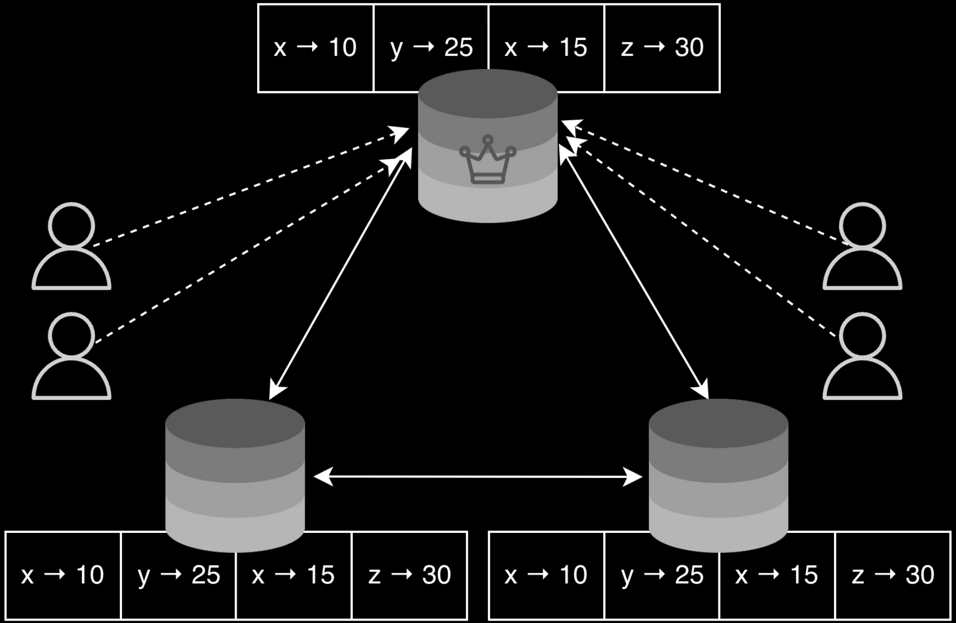
Agreement on the state of the system, even under faults

What do these protocols guarantee?

Fault Tolerance

Ordering

Strict Serializability



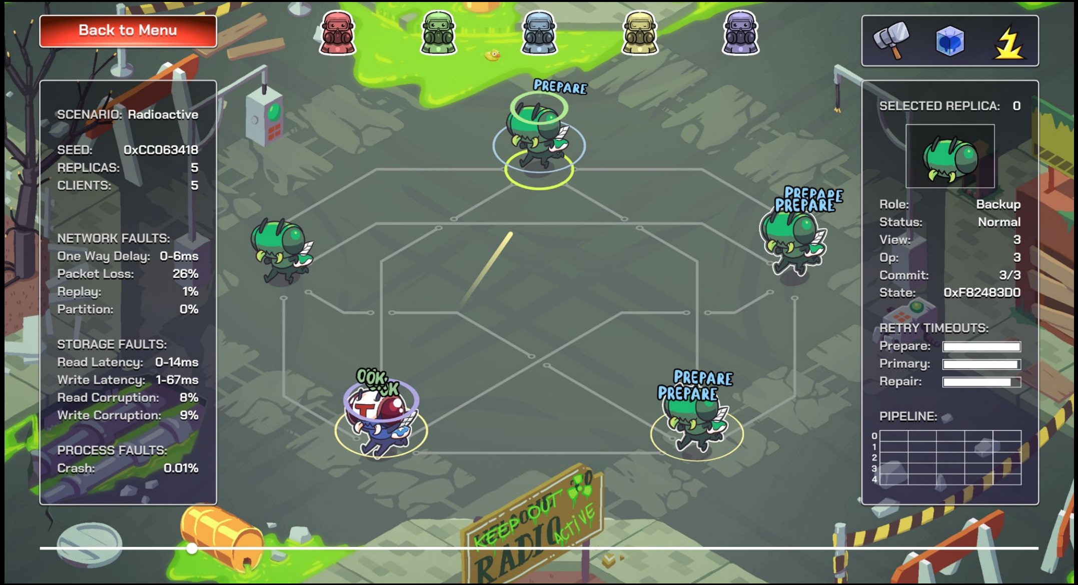
Building Consensus Protocols

Notoriously hard to get right

How we test consensus at

Deterministic Simulation Testing

- Nondeterministic, physical interactions stubbed
- Time sped up
- Finds bugs using *assertions*



SCENARIO: Radioactive

SEED:	0xCC063418
REPLICAS:	5
CLIENTS:	5

NETWORK FAULTS:

One Way Delay:	0-6ms
Packet Loss:	26%
Replay:	1%
Partition:	0%

STORAGE FAULTS:

Read Latency:	0-14ms
Write Latency:	1-67ms
Read Corruption:	8%
Write Corruption:	9%

PROCESS FAULTS:

Crash:	0.01%

SELECTED REPLICA: 0

Role:	Backup
Status:	Normal
View:	3
Op:	3
Commit:	3/3
State:	0xF82483D0

RETRY TIMEOUTS:
Prepare:
Primary:
Repair:

PIPELINE:
0
1
2
3
4

Assertions

Validate system invariants at runtime

Assertions across layers

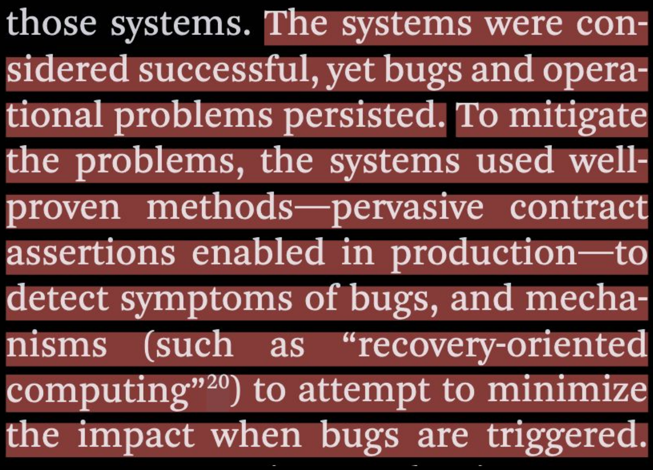
Assertions in Prod

- Enforce *replica*-level invariants

Assertions in DST

- Check *cluster*-level invariants

```zig
fn on_request_start_view(
    self: *Replica,
    message: *const Message.RequestStartView,
) void {
    assert(message.header.command == .request_start_view);
    if (self.ignore_repair_message(message.base_const())) return;

    assert(self.status == .normal);
    assert(self.view == self.log_view);
    assert(message.header.view == self.view);
    assert(message.header.replica != self.replica);
    assert(self.primary());

    const start_view_message = self.create_start_view_message(message.header.nonce);
    defer self.message_bus.unref(start_view_message);

    assert(start_view_message.header.command == .start_view);
    assert(start_view_message.references == 1);
    assert(start_view_message.header.view == self.view);
    assert(start_view_message.header.op == self.op);
    assert(start_view_message.header.commit_max == self.commit_max);
    assert(start_view_message.header.nonce == message.header.nonce);
    self.send_message_to_replica(message.header.replica, start_view_message);
}
```



those systems. The systems were considered successful, yet bugs and operational problems persisted. To mitigate the problems, the systems used well-proven methods—pervasive contract assertions enabled in production—to detect symptoms of bugs, and mechanisms (such as "recovery-oriented computing"[20]) to attempt to minimize the impact when bugs are triggered.

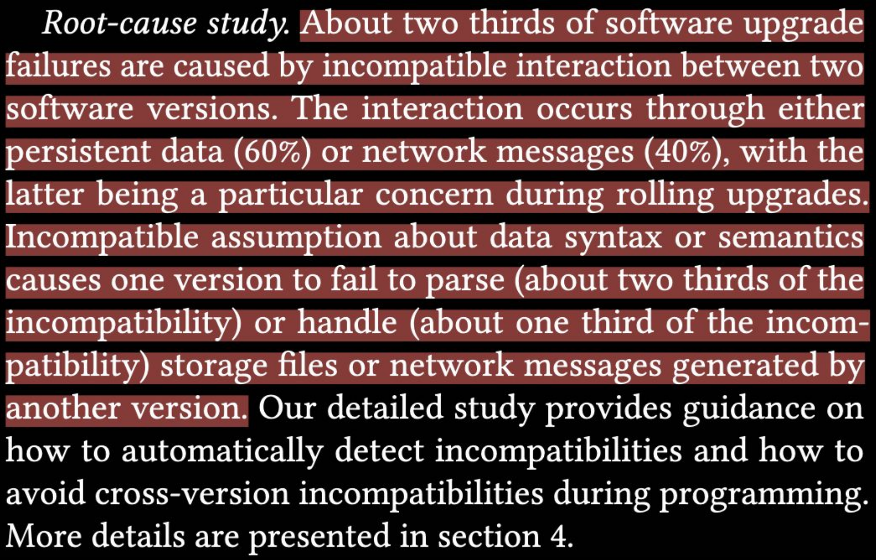
How Amazon Web Services Uses Formal Methods" – Newcombe et al., CACM '15

JEPSEN

TigerBeetle 0.16.11

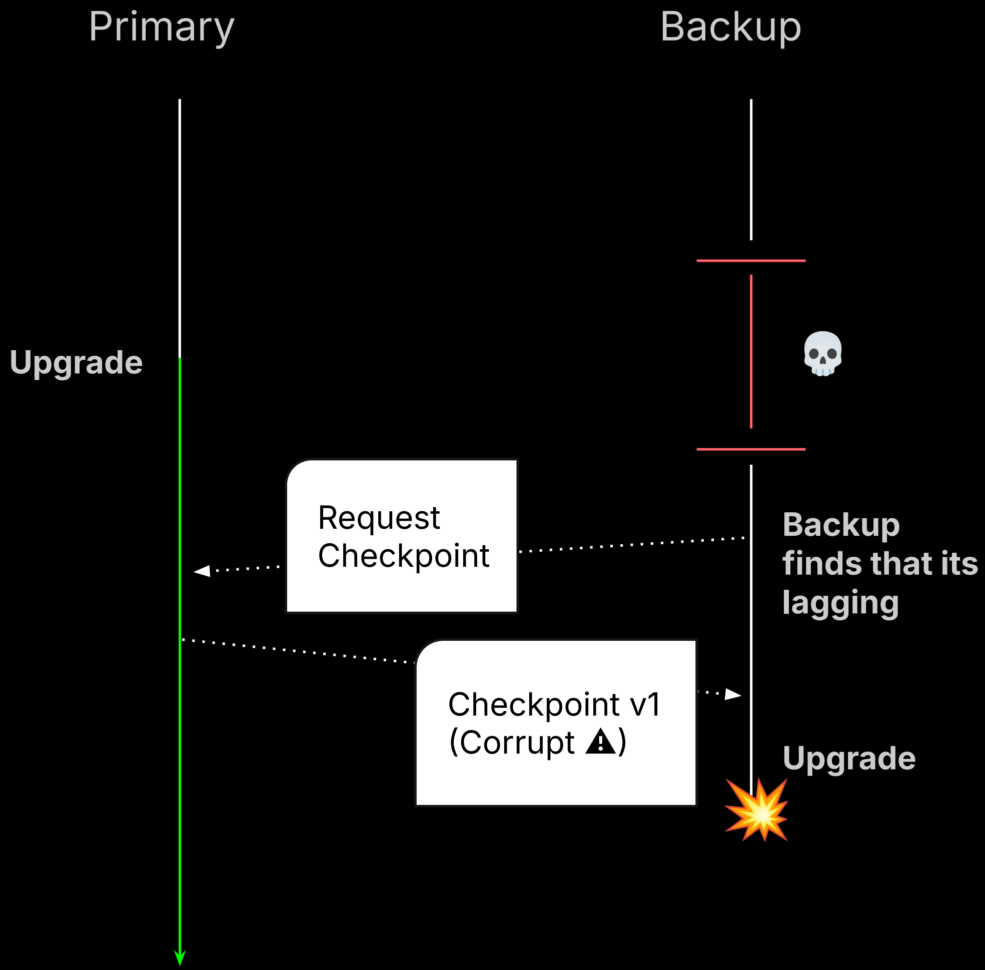
Kyle Kingsbury
2025-06-06

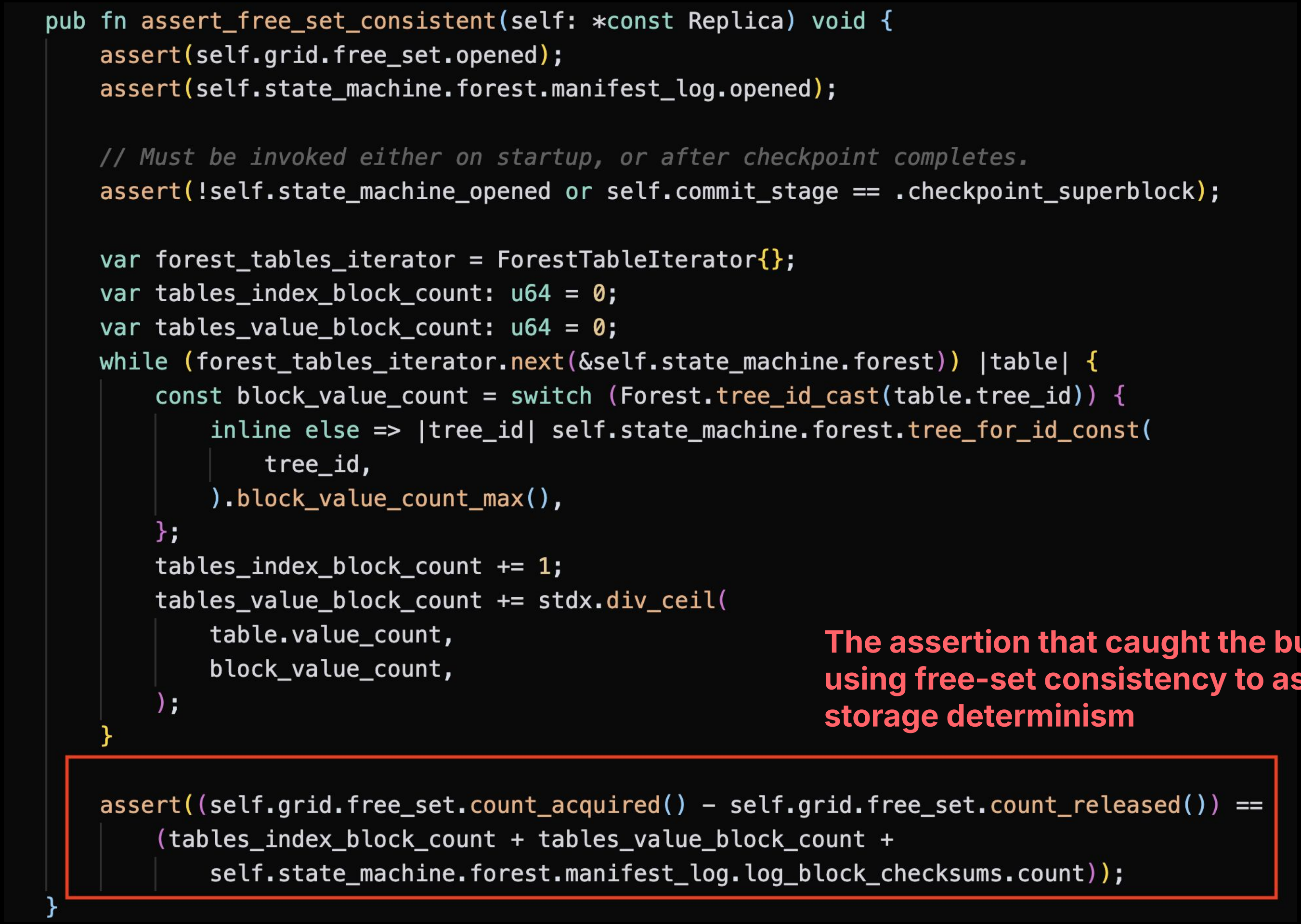
TigerBeetle is a distributed OLTP database oriented towards financial transactions. We tested TigerBeetle 0.16.11 through 0.16.30. We discovered seven client and server crashes, including a segfault on client close and several panics during server upgrades. Single-node failures could cause significantly 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.



Root-cause study. About two thirds of software upgrade failures are caused by incompatible interaction between two software versions. The interaction occurs through either persistent data (60%) or network messages (40%), with the latter being a particular concern during rolling upgrades. Incompatible assumption about data syntax or semantics causes one version to fail to parse (about two thirds of the incompatibility) or handle (about one third of the incompatibility) storage files or network messages generated by another version. Our detailed study provides guidance on how to automatically detect incompatibilities and how to avoid cross-version incompatibilities during programming. More details are presented in section 4.

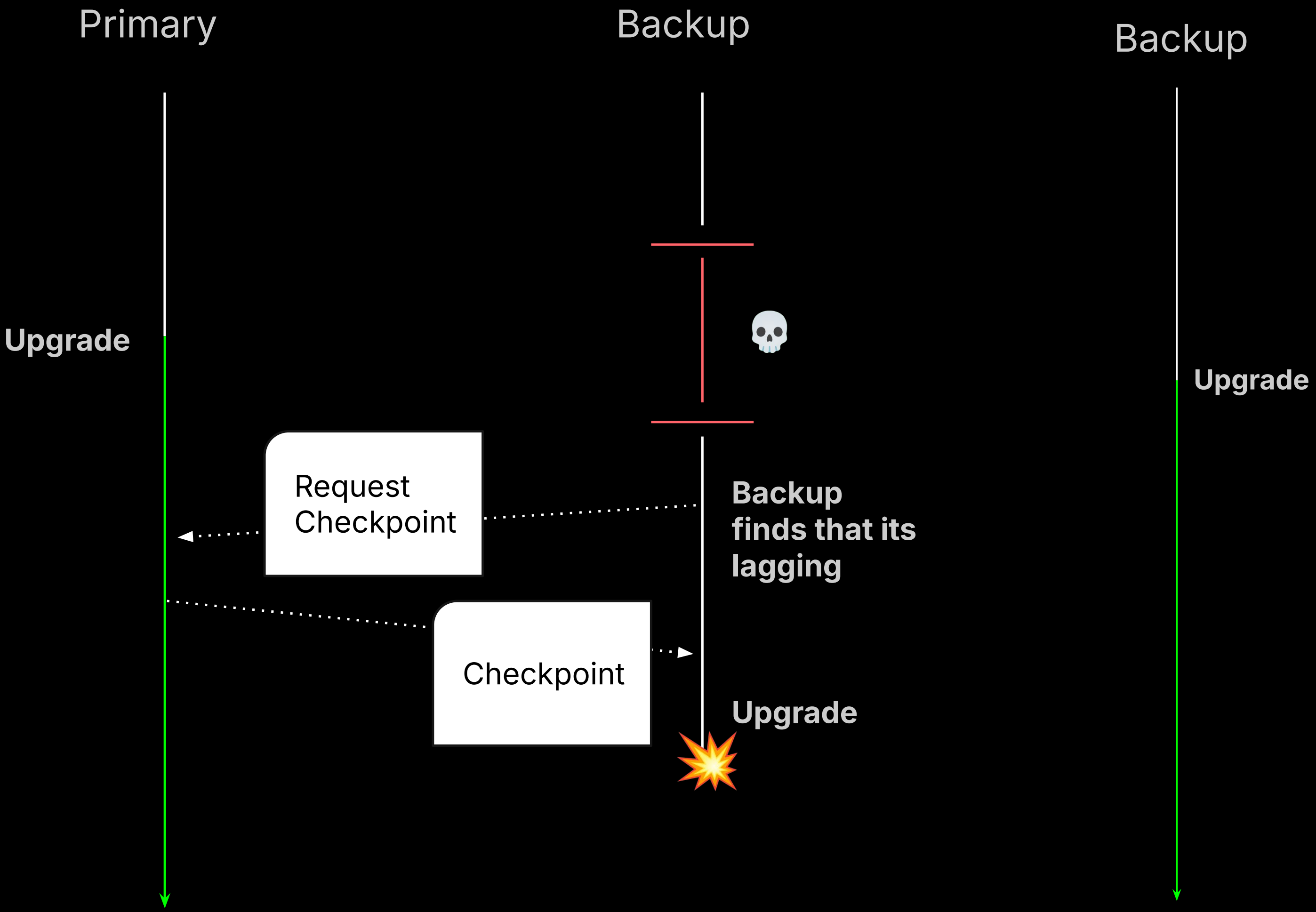
Understanding and Detecting Software Upgrade Failures in Distributed Systems - Zhang et al., SOSP 2021

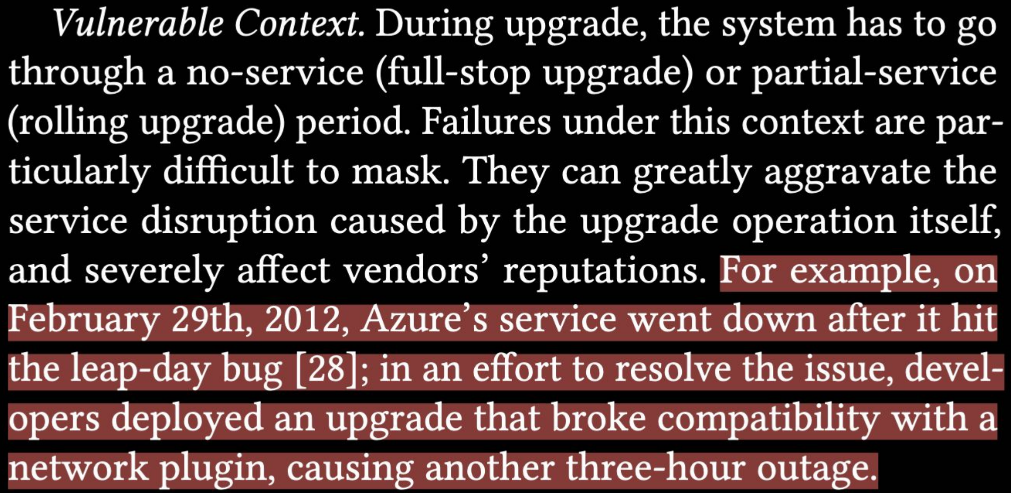

Primary
Backup
Upgrade
💀
Request Checkpoint
Backup finds that its lagging
Checkpoint v1 (Corrupt ⚠)
Upgrade
💥



```
pub fn assert_free_set_consistent(self: *const Replica) void {
    assert(self.grid.free_set.opened);
    assert(self.state_machine.forest.manifest_log.opened);

    // Must be invoked either on startup, or after checkpoint completes.
    assert(!self.state_machine_opened or self.commit_stage == .checkpoint_superblock);

    var forest_tables_iterator = ForestTableIterator{};
    var tables_index_block_count: u64 = 0;
    var tables_value_block_count: u64 = 0;
    while (forest_tables_iterator.next(&self.state_machine.forest)) |table| {
        const block_value_count = switch (Forest.tree_id_cast(table.tree_id)) {
            inline else => |tree_id| self.state_machine.forest.tree_for_id_const(
                tree_id,
            ).block_value_count_max(),
        };
        tables_index_block_count += 1;
        tables_value_block_count += stdx.div_ceil(
            table.value_count,
            block_value_count,
        );
    }

    assert((self.grid.free_set.count_acquired() - self.grid.free_set.count_released()) ==
        (tables_index_block_count + tables_value_block_count +
            self.state_machine.forest.manifest_log.log_block_checksums.count));
}
```

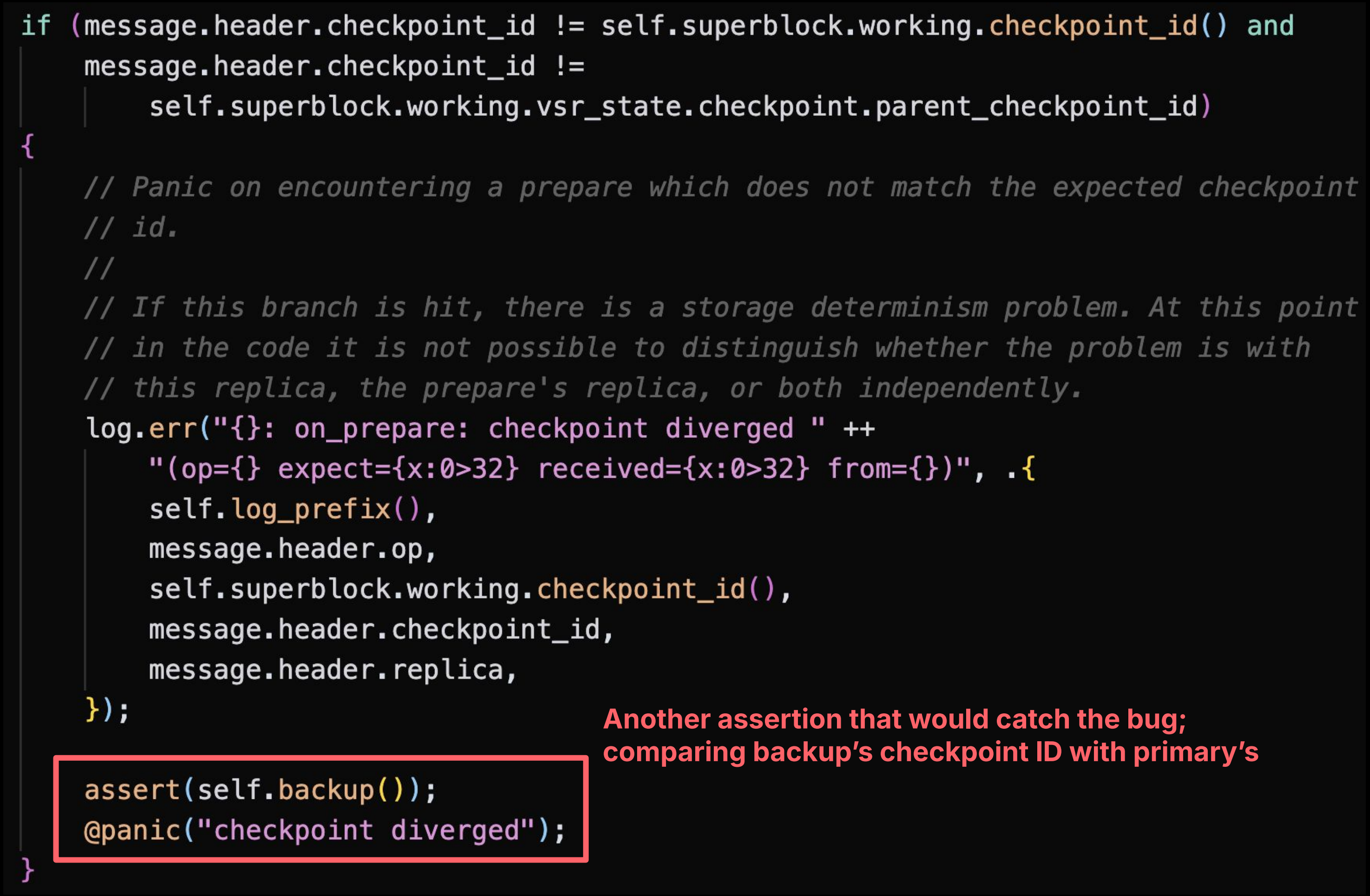
The assertion that caught the bug; using free-set consistency to assert storage determinism


Primary
Backup
Backup
Upgrade
💀
Upgrade
Request Checkpoint
Backup finds that its lagging
Checkpoint
Upgrade
💥



Vulnerable Context. During upgrade, the system has to go through a no-service (full-stop upgrade) or partial-service (rolling upgrade) period. Failures under this context are particularly difficult to mask. They can greatly aggravate the service disruption caused by the upgrade operation itself, and severely affect vendors' reputations. For example, on February 29th, 2012, Azure's service went down after it hit the leap-day bug [28]; in an effort to resolve the issue, developers deployed an upgrade that broke compatibility with a network plugin, causing another three-hour outage.

Understanding and Detecting Software Upgrade Failures in Distributed Systems - Zhang et al., SOSP 2021



```
if (message.header.checkpoint_id != self.superblock.working.checkpoint_id() and
    message.header.checkpoint_id !=
        self.superblock.working.vsr_state.checkpoint.parent_checkpoint_id)
{
    // Panic on encountering a prepare which does not match the expected checkpoint
    // id.
    //
    // If this branch is hit, there is a storage determinism problem. At this point
    // in the code it is not possible to distinguish whether the problem is with
    // this replica, the prepare's replica, or both independently.
    log.err("{}: on_prepare: checkpoint diverged " ++
        "(op={} expect={x:0>32} received={x:0>32} from={})", .{
        self.log_prefix(),
        message.header.op,
        self.superblock.working.checkpoint_id(),
        message.header.checkpoint_id,
        message.header.replica,
    });

    assert(self.backup());
    @panic("checkpoint diverged");
}
```

Another assertion that would catch the bug; comparing backup's checkpoint ID with primary's

Are we feeling assertive yet?

Tiger Style