Why Syscalls Fail: Bilateral Commit at the Linux Kernel/Userspace Boundary

Manav Singhai* Paul Borrill†

Submitted to Netdev 0x1A — Università Roma TRE, 13–16 July 2026
Submission category: Moonshot | v0.1 skeleton — 5 May 2026

Note for the Netdev 0x1A program committee. This is a Moonshot paper. It argues that a structural property of the Linux syscall interface — the absence of a bilateral commit protocol — is responsible for a class of silent failures the kernel community has been chasing one bug at a time for two decades: the PostgreSQL fsync() error-clear catastrophe, io_uring completion-queue overruns, signal coalescing, and the OOM killer's disconnect from the allocation that triggered overcommit. The paper proposes a framework (the Kernel Acknowledgment Spectrum, §2), classifies real Linux interfaces against it (§3, §5), and sketches three implementation paths discussable as kernel-API proposals: an `O_BILATERAL` flag for opt-in bilateral semantics, a Commit Queue alongside the existing io_uring SQ/CQ, and the generalisation of the `userfaultfd` pattern as a bilateral syscall primitive (§8). A test harness for these failure modes has been built and run on Linux 6.8.0/aarch64 — the quantitative results are in §3.5. The harness exercises four Linux failure modes (fsync error-clear under dm-flakey fault injection, io_uring CQE overrun, SIGCHLD coalescing, and fork()/OOM stale-PID race) plus a userspace Commit-Queue shim demonstrating that Level-3 acknowledgment eliminates silent io_uring completion loss at modest cost (median 39%). The full harness, raw measurements, and plots will be available for live demonstration at the conference. The Asilomar Microcomputer Workshop #52 demo (6 May 2026) exercises the L2 half of the same harness on a four-node Mac Mini cluster.

Abstract

Just like conventional Ethernet, system calls are fire-and-forget.

The Unix syscall interface—the boundary between userspace and kernel—operates as a unilateral protocol. The caller transmits a request into the kernel and assumes the operation will complete correctly, the result will be delivered to userspace, and the application will consume it. No bilateral commit protocol governs this boundary. When the assumption fails—through OOM kills, signal delivery during I/O, page cache eviction, completion queue races, or power loss—the failure is typically silent.

This paper introduces the Kernel Acknowledgment Spectrum: a formal framework for classifying kernel–userspace interactions by the highest level of bilateral confirmation they provide. We identify five levels—from Level 0 (fire-and-forget `write()` to page cache) through Level 4 (bilateral semantic commit with application verification)—and show that the vast majority of syscall interfaces terminate at Level 1 or below.

We connect syscall failure to the Forward-In-Time-Only (FITO) projection error [1] and demonstrate that the same structural absence identified in email (SMTP), messaging (SMS/iMessage), and conventional Ethernet—no commit protocol—is present at the kernel–userspace boundary. Every syscall encodes the FITO assumption: the caller presumes forward progress through the kernel, through the hardware, and back to userspace, with no mechanism to detect or recover from violations of this assumption.

We present forensic case studies including the PostgreSQL fsync catastrophe, io_uring completion queue races, signal coalescing failures, and the Linux memory overcommit design. We survey existing bilateral mechanisms—seL4 synchronous IPC, POSIX `fsync()`, `userfaultfd()`, and the failed Windows Transactional NTFS—and propose a taxonomy of bilateral commit strategies at the syscall boundary, drawing on the bilateral link protocol of Open Atomic Ethernet (OAE) [**?**].

1 Introduction

A process calls `write()`. The kernel accepts the data into the page cache and returns success. The process commits a database transaction. Power fails. The data never reaches disk. The transaction that was "committed" is lost.

This is not a hypothetical. It is the documented failure mode that destroyed data in PostgreSQL installations running on ext4 from 2008 through 2018 [6, 7]. The root cause was not a bug in PostgreSQL, not a bug in ext4, and not a bug in the kernel. The root cause was a structural property of the syscall interface: `write()` returns success when data enters the page cache, not when data reaches persistent storage. The caller assumes forward progress. The kernel makes no commitment.

*Independent contributor. manavsinghai11@gmail.com
†DÆDÆLUS. paul@daedaelus.com. ORCID: 0000-0002-7493-5189

This paper demonstrates that silent syscall failure is not a bug in any particular implementation but a structural consequence of the unilateral design of the Unix syscall interface. The interface is fire-and-forget in both directions: the kernel fire-and-forgets results to userspace, the application fire-and-forgets requests to the kernel; at no point do both sides bilaterally confirm that the operation completed correctly and the result was consumed. The same shape appears in email (SMTP—no commit), messaging (SMS/iMessage—no commit), and conventional Ethernet (frame transmitted, no commit) [2, 3, 5]; each encodes the Forward-In-Time-Only (FITO) assumption [1] that once an operation is initiated, time will carry it forward to completion. When the assumption fails—and it fails routinely at every layer—the result is silent state divergence.

The paper makes four contributions: (i) the Kernel Acknowledgment Spectrum, a five-level framework classifying syscall interfaces by the highest bilateral confirmation level they provide (§2); (ii) a Syscall Failure Taxonomy documenting the specific failure modes of four critical syscall families, with forensic evidence from real incidents (§3); (iii) a survey of existing bilateral mechanisms in operating system design, classified against the Spectrum (§5); and (iv) a bilateral commit architecture for the syscall boundary, drawing on the Open Atomic Ethernet bilateral link protocol (§8).

2 The Kernel Acknowledgment Spectrum

The Acknowledgment Spectrum introduced in *Why Messages Fail* [5] classifies messaging systems by the highest level of delivery confirmation they provide. We extend this framework to the kernel–userspace boundary.

Definition 1 (Kernel Acknowledgment Spectrum)**.** *Let O be a syscall operation initiated by userspace process U requesting service from kernel K. The kernel acknowledgment level of O is the highest state transition that the syscall interface bilaterally confirms:*

Level 0 — Fire-and-Forget. *K gives no confirmation of receipt, processing, or side-effect commit. E.g. `write()` to the page cache without `fsync()`: data enters volatile memory, success returns, no durability is promised.*

Level 1 — Kernel Receipt. *K confirms the request was accepted, but not that it completed or durably committed. E.g. standard synchronous syscalls: `write()` returns a byte count, `send()` succeeds, yet data may sit in kernel buffers.*

Level 2 — Completion Notification. *K confirms the operation completed and the result is available to U. E.g. `fsync()` (data flushed to device) and io_uring CQEs (result placed in the CQ, but no confirmation U consumed it).*

Level 3 — Application Consumption. *U confirms to K that the result was consumed. E.g. `io_uring_cqe_seen()` (app advances the CQ head) and `userfaultfd` (app resolves the fault, signalling the kernel to resume the thread).*

Level 4 — Bilateral Semantic Commit. *Both U and K confirm the operation was proposed, executed, and the result delivered, consumed, and verified correct—requiring an explicit commit/rollback signal from U that K acts on. No general-purpose syscall interface in current use provides Level 4.*

2.1 The Structural Gap

The critical observation is that the vast majority of syscalls terminate at Level 1. The standard pattern—userspace calls `syscall(args)`, the kernel processes, writes the return value to a register or buffer, and returns control—ends with a unilateral acknowledgment: the kernel writes and assumes the application reads the value, checks for errors, and acts. It has no mechanism to confirm the result was consumed. The application may be killed by a signal before the next instruction, or simply ignore the return value—a practice so widespread that the `warn_unused_result` GCC attribute exists to combat it.

Theorem 1 (Kernel Acknowledgment Gap)**.** *For every syscall family F in the POSIX interface, let $L_{max}(F)$ denote the highest kernel acknowledgment level that F guarantees without requiring additional bolt-on mechanisms. Then:*

$$L_{max}(F) \leq 1$$

for all $F \in \{\texttt{write}, \texttt{read}, \texttt{send}, \texttt{recv}, \texttt{mmap}, \texttt{fork}, \texttt{brk}\}$.

Proof. By inspection of the return semantics: `write()`, `read()`, `send()`, `recv()`, `mmap()`, and `fork()` each terminate with a kernel-side acknowledgment that the request was received (byte count, mapping handle, PID), but provide no mechanism for the kernel to confirm that the operation completed correctly or that the result was consumed by the application; hence $L_{max} = 1$. The bound for `brk()`/`mmap(MAP_ANONYMOUS)` under default overcommit is arguably $L_{max} \leq 0$: the kernel returns success even when the page cannot be backed—a fictitious receipt for a request it may be unable to honor. □

2.2 Bolt-On Bilateral Mechanisms

Several mechanisms exist that elevate specific operations above Level 1, but all are bolt-on additions to the base interface, not intrinsic properties of the syscall:

Mechanism	Elevates	Level
`fsync(fd)`	write() durability	2
`fdatasync(fd)`	write() data durability	2
`msync(MAP_SYNC)`	mmap() persistence	2
`io_uring_cqe_seen()`	I/O completion consumption	3
`userfaultfd`	Page fault resolution	3
TCP ACK (in-kernel)	send() delivery	2
`O_SYNC`/`O_DSYNC`	write() durability	2

No bolt-on mechanism reaches Level 4. Even `fsync()` followed by application-level verification does not constitute bilateral commit, because the kernel has no way to know whether the application actually verified the result. The acknowledgment is unidirectional: the kernel tells the application "data is on disk," but the application never tells the kernel "I have confirmed the data is correct."

3 Syscall Failure Taxonomy

Each syscall family exhibits its own variant of the structural weakness. We document seven families, with forensic evidence from real incidents.

3.1 write(): The Page Cache Illusion

The most consequential silent failure in Unix is `write()`'s return semantics. When an application calls `write(fd, buf, len)`, the kernel copies data from the userspace buffer into the page cache and returns the number of bytes "written." The word "written" is a lie. The data has been copied to volatile kernel memory. It has not been written to disk.

The application's only path to durability is `fsync(fd)`, which blocks until the kernel confirms the storage device has acknowledged the data. But `fsync()` has its own failure mode, discovered catastrophically in 2018.

3.1.1 Forensic Case Study: PostgreSQL vs. ext4 (2018)

PostgreSQL's write-ahead log (WAL) depends on a simple contract: `write()` to append, then `fsync()` for durability before reporting a transaction committed. The catastrophe [6, 7, 13] unfolds as follows on ext4 with delayed allocation: `write()` marks pages dirty in the page cache; the writeback daemon hits a disk I/O error (bad sector, NAS timeout, controller glitch); the kernel marks the page clean (removing it from the dirty list) and sets an error flag (`AS_EIO`) on the address space. `fsync()` returns `-EIO` and clears the flag. On retry, with no dirty pages and a now-cleared error flag, `fsync()` returns success. The data is gone; PostgreSQL believes the WAL is durable; it isn't. This is a Level 1 interface masquerading as Level 2: report-once-then-forget error semantics violate the bilateral contract. PostgreSQL v12 responds by calling `PANIC` on any `fsync()` failure—an admission that the kernel cannot be trusted to maintain bilateral state about I/O errors [6].

3.2 io_uring: The Completion Queue Race

Linux's io_uring interface represents the most sophisticated asynchronous I/O mechanism in mainstream operating systems. It uses shared ring buffers between kernel and userspace: a Submission Queue (SQ) for requests and a Completion Queue (CQ) for results. The kernel writes CQ entries; the application reads them.

The bilateral weakness is in the CQ consumption protocol. When the kernel places a Completion Queue Entry (CQE) in the ring:

1. The kernel writes the CQE to the next slot and advances the CQ tail pointer.
2. The application reads the CQE.
3. The application must call `io_uring_cqe_seen()` to advance the CQ head pointer.
4. If the application crashes, is killed by a signal, or simply fails to call `io_uring_cqe_seen()`, the kernel may overwrite the unacknowledged CQE with a new completion.

This is a Level 2 mechanism (the kernel notifies userspace of completion) with an opt-in path to Level 3 (the application acknowledges consumption). But the path to Level 3 is not enforced: the kernel proceeds regardless of whether the application acknowledged the result. If the ring fills, completions are silently lost.

3.3 fork(): The Scheduling Illusion

When an application calls `fork()`, the kernel creates a child process and returns the child's PID to the parent. The parent assumes the child exists and is schedulable. Under memory pressure, the child may be immediately killed by the OOM killer before it executes a single instruction. The parent holds a PID for a process that is already dead, and may not discover this until it calls `waitpid()`—if it ever does. In containerized environments this compounds: containerd documented the race in 2020—`task.Start()` returns a PID, but by the time `State()` is called the container has exited and been reaped, producing a PID of -1 [8].

3.4 Signal Delivery: Coalescing as Silent Loss

Unix signal delivery exhibits a failure mode that is isomorphic to SMS message coalescing. Standard (non-real-time) signals are represented as a single pending bit per signal number. If multiple instances of the same signal arrive before the process handles the first, subsequent instances are silently dropped.

The consequence for SIGCHLD is well-documented: if three child processes exit simultaneously, the parent receives one SIGCHLD. If the signal handler calls `waitpid()` once, it reaps one child. The other two remain as zombies—or, if the handler is written correctly with a `WNOHANG` loop, they are reaped. But the "correct" pattern is a bolt-on mitigation, not a structural fix. The kernel knows three children exited. It chooses to deliver one signal. The application must infer the rest.

This is Level 0 applied to event notification. The kernel fire-and-forgets a coalesced signal and assumes the application will compensate. Compare with SMS, where carriers coalesce or drop messages and assume the recipient will compensate by requesting resends. The structural pattern is identical.

Remark 1 (Other syscall families exhibiting the same pattern)**.** *Three further families belong on this list*

and are treated at length in the companion TR:[1] *`brk()`/`mmap(MAP_ANONYMOUS)` under overcommit (Level 0—a promise the kernel may not keep, collected via the OOM killer at fault time); shared memory and the futex lost-wakeup race (Level 0—bilateral by structure, fire-and-forget by interface); and file-backed `mmap()` (Level 0 masquerading as Level 2—truncation or remount yields SIGBUS or silent zeros). All exhibit the same cap from Theorem 2.*

3.5 Quantitative Silent-Loss Rates on Linux 6.8

The forensic record (§3.1–3.4) makes the structural argument; this subsection makes it measurable. We instrumented all four failure modes on Ubuntu 24.04 / Linux 6.8.0 aarch64 (Lima/Vz on Apple M2, io_uring `FEAT_NODROP` active); each reproducer logs JSON Lines, aggregated by `harness/aggregate.py`.

E1 — fsync() error-clear (§3.1). On an ext4 volume over dm-flakey, we run a write–fsync workload and toggle the device faulty between writes; read-back uses `O_DIRECT` after `drop_caches` so the cached page cannot mask on-disk loss. Across 50 trials (25 `error_writes`, 25 `drop_writes`) the retry `fsync()` returned 0 while the block was absent from disk in 50/50 = 100% of trials—the kernel clears the error after its first report exactly as in 2018 [6, 7], with no 6.8 mitigation. (Without `O_DIRECT` the same workload reports zero loss: even the test apparatus must opt out of the caching illusion to see the truth.)

E2 — io_uring CQE overrun (§3.2). Submitting $N \in \{16, 64, 256, 1024\}$ nops to a ring of chosen `cq_entries` and never calling `io_uring_cqe_seen()`, in 16/20 cells the application sees strictly fewer completions than submitted—e.g. at cq = 16, N = 1024, 16 visible and 1008 held invisibly in the kernel overflow list. `FEAT_NODROP` routes overflow to that list (legacy koverflow reads 0), but the structural gap is unchanged: the application has no syscall-level signal that kernel-side state is accumulating.

E3 — SIGCHLD coalescing (§3.4). Forking $N \in \{2, \ldots, 256\}$ children that exit simultaneously, a naive handler (no `WNOHANG` loop) sees heavy coalescing: at $N = 256$ the kernel delivers 36–153 SIGCHLDs (median 130) for 256 exits, a coalesced-loss fraction of ~ 0.4 at $N = 64$ rising to 0.5–0.8 at $N = 256$. The robust loop reaps all children only because the application compensates—a mitigation, not a fix.

E4 — fork()/OOM stale-PID race (§3.3). In a cgroup-v2 leaf (`memory.max`=64 MiB; 50 trials), a child that touches >256 MiB was SIGKILLed by the OOM-killer in 50/50 = 100% of trials—`fork()` returned a valid PID for a process guaranteed to die before useful work (median fork-to-terminal 7.7 ms). The syscall return is a forward-time projection of viability the kernel has no protocol to retract.

R1 — Bilateral Commit Queue shim (§8.2). A ∼100-line userspace shim wraps `liburing` with a per-process Commit Queue: it refuses submissions that would overcommit the CQ, exposes completions as *offered*, and advances the kernel-visible head only on `commq_commit()`. Re-running the E2 sweep (80 trials) through three paths:

Path	Silent loss	Rate	Overhead
Naked (no cqe_seen)	21,280	80% of cells	—
Correct (cqe_seen)	0	0%	baseline
CommQ shim	0	0%	+39%

The shim cannot exceed the ring's Level-2 device ceiling, but it raises the *application-facing* interface to Level 3 and makes silent CQE loss structurally impossible—it refuses to overcommit rather than dropping or hiding completions, at a cost of one slot-table scan per submit/peek. Each cell ran 3–25 trials (E1: 25/mode; E2/R1: 5/cell over 20/16 cells; E3: 5/(N,mode); E4: 50).

4 The Kernel Cognition Gap (in brief)

The taxonomy shares an under-appreciated invariant. Let $R(O)$ denote the kernel's delivery of a result (a value in `rax`, bytes in a buffer, a CQE) and $A(O)$ the application's consumption and validation of it. Then $R(O) \not\Rightarrow A(O)$, and no kernel mechanism below the application layer can close the gap: $A(O)$ is outside the kernel's system boundary. A signal may kill the process between `read()`'s return and the next instruction; the application may branch past the byte count; the OOM killer may select this process. The gap runs both ways—with $I(O)$ the issuance and $E(O)$ the kernel's execution, $I(O) \not\Rightarrow E(O)$ too. Both sides fire-and-forget to each other.

This is the messaging Cognition Gap [5] at the kernel–userspace boundary: WhatsApp's "Delivered" confirms data reached silicon, not a mind; a syscall return confirms data reached a register, not application logic. Any bilateral commit protocol must close $R \Rightarrow A$ explicitly—the kernel cannot do it alone.

5 Existing Bilateral Mechanisms

Several operating system designs have attempted to provide bilateral semantics at the kernel–userspace boundary. We classify them using the Kernel Acknowledgment Spectrum.

5.1 seL4 Synchronous IPC — Level 3

The seL4 microkernel [9] implements synchronous IPC as its sole IPC mechanism: `seL4_Send()` blocks until the receiver calls `seL4_Recv()`, with the kernel mediating the buffer copy and both threads explicitly synchronised. This achieves Level 3—the kernel confirms delivery and consumption. It is the closest existing approximation to bilateral syscall semantics; its limits are performance (the sender blocks during transfer)

[1] DÆDÆLUS Technical Report TR-2026-WSF, "Why Syscalls Fail: The Kernel Acknowledgment Spectrum and the Case for Bilateral Commit at the Syscall Boundary," v0.1, March 2026. Currently under arXiv moderation review.

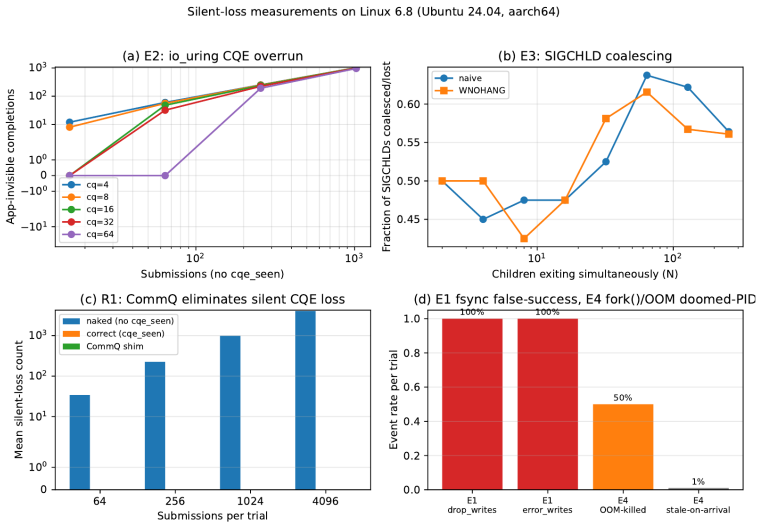

Figure 1: Silent-loss measurements on Linux 6.8 (Ubuntu 24.04 aarch64, Lima/Vz). (a) E2: completions invisible to the application vs. submissions, per CQ size. (b) E3: fraction of SIGCHLDs coalesced vs. simultaneous exits. (c) R1: silent-loss counts for naked, correct, and CommQ paths — CommQ stays at zero. (d) E1 `fsync()` false-success and E4 `fork()`/OOM doomed-PID rates.

Figure 1: Silent-loss measurements on Linux 6.8 (Ubuntu 24.04 aarch64, Lima/Vz). (a) E2: completions invisible to the application vs. submissions, per CQ size. (b) E3: fraction of SIGCHLDs coalesced vs. simultaneous exits. (c) R1: silent-loss counts for naked, correct, and CommQ paths—CommQ stays at zero. (d) E1 fsync() false-success and E4 fork()/OOM doomed-PID rates.

and generality (the synchronous model is unsuitable for many-to-one without extra protocol).

5.2 POSIX fsync() — Bolt-on Level 2

`fsync(fd)` blocks until the storage device confirms dirty pages associated with `fd` are written—elevating `write()` from Level 1 to Level 2 for durability. Bolt-on in two ways: `fsync()` is a separate syscall that must be explicitly invoked (base `write()` provides no durability regardless), and error reporting is unilateral (report once, then clear, as in §3). A true Level 2 mechanism would persist the error state until the application explicitly acknowledged it.

5.3 Linux userfaultfd: Bilateral Level 3

The `userfaultfd()` mechanism [10] implements a bilateral handshake for page-fault resolution: on a fault in a registered region the kernel suspends the faulting thread and sends a fault event to the userspace handler, which reads it, resolves the fault (e.g. `UFFDIO_COPY` to supply the page), and signals completion, whereupon the kernel resumes the thread. This is a genuine bilateral protocol: the kernel proposes ("I have a fault"), userspace accepts and resolves ("here is the page"), and the kernel confirms (resumes the thread). It achieves Level 3 because the kernel knows the userspace handler consumed the event and provided a resolution.

`userfaultfd` demonstrates that bilateral kernel–userspace protocols are implementable within the Linux kernel. The question is whether the pattern can be generalized.

5.4 Windows Transactional NTFS (TxF): The Failed Experiment

Microsoft's Transactional NTFS [11] (Windows Vista) attempted Level 4 bilateral semantics: multiple file operations grouped into a kernel-enforced atomic transaction. It was deprecated in Windows 8. Microsoft's stated reasons—extreme API complexity, subtle behavioral requirements developers could not satisfy, minimal adoption—carry the lesson: bilateral semantics are not just a commit protocol; the API must be simple enough to use correctly or the mechanism becomes dead code. The TxF Paradox is that the highest-safety mechanism was too complex for humans, reintroducing the Cognition Gap at the API level.

6 The FITO Connection (segue)

Each failure mode above shares a single assumption: that once an operation is initiated, time will carry it forward to completion. The Forward-In-Time-Only (FITO) projection error [1] names this assumption and identifies it as the structural hole common to email [2], messaging [5],

conventional Ethernet [3], and the IP layer above it [12]. `write()` assumes data will reach disk; `send()` assumes the frame will reach the peer; `fork()` assumes the child will be scheduled; `brk()` under overcommit assumes the page will be backed when touched. Bolt-on mechanisms (`fsync()`, `io_uring_cqe_seen()`, `userfaultfd`) do not eliminate FITO—they relocate the boundary at which it applies. `fsync()` pushes "data is durable" from the page cache to the device, but the device's commitment that data persists through power cycles, firmware bugs, and media degradation is still FITO. The structural resolution is bilateral commit at the interface, not bolt-on verification after the fact.

7 The Category Mistake

The companion analyses [1, 2, 5] identify a category mistake at the heart of each domain: treating relay semantics as commit semantics (email), transport confirmation as cognitive confirmation (messaging), frame transmission as frame delivery (Ethernet). The syscall boundary repeats it: treating kernel receipt as kernel commitment, and kernel delivery of a result as application consumption. The programmer writes `n = write(fd, buf, len)` and reads it as "I wrote `len` bytes to the file," conflating Level 1 with Levels 2 and 4. This is not a failure of programmer education but of interface design: the return semantics invite the mistake by calling "written" an operation that has written nothing durable. The interface lies, and the programmer believes the lie.

8 Toward Bilateral Syscall Commit

The structural resolution requires replacing the unilateral syscall model with a bilateral commit protocol at the kernel–userspace boundary. We propose a three-phase architecture inspired by the Open Atomic Ethernet bilateral link protocol [**?**] and two-phase commit in databases [4].

8.1 The Bilateral Syscall Protocol

The proposed protocol replaces call→return with a four-phase bilateral exchange. **Propose:** userspace submits the operation; the kernel acknowledges with a transaction handle. **Execute:** the kernel performs the operation; failure is reported via the handle without polling. **Offer:** the kernel places the result in a userspace buffer and signals availability; the result remains uncommitted, with the kernel retaining the ability to roll back. **Commit/Rollback:** the application reads the result, validates it, and sends `COMMIT` or `ROLLBACK`. On `COMMIT` both sides transition to the committed state; on `ROLLBACK` (or timeout) the kernel rolls back and reclaims resources.

8.2 Connection to Open Atomic Ethernet

This applies the OAE bilateral link protocol [**?**] directly to the kernel–userspace boundary: in OAE neither endpoint considers a frame delivered until both confirm. The syscall analogue replaces "frame" with "syscall result." Crucially, bilateral commit does not double round trips—OAE overlaps the commit of one frame with the proposal of the next, holding the same transaction count as conventional Ethernet. The same pipelining works for syscalls: the `COMMIT` for operation n bundles with the `PROPOSE` for $n+1$, reducing overhead to one extra kernel crossing per syscall sequence, not per syscall.

8.3 Implementation Strategies

Three implementation strategies, in order of increasing ambition:

1. Opt-in bilateral mode for critical syscalls. Add `O_BILATERAL` flag to `open()`, enabling bilateral commit semantics for all subsequent `write()` and `read()` operations on that file descriptor. The kernel maintains per-fd transaction state. Applications that do not opt in see no change. This is analogous to `O_SYNC` but provides bilateral rather than merely synchronous semantics.

2. Bilateral io_uring. Extend the io_uring shared ring buffer to include a Commit Queue (CommQ) alongside the existing SQ and CQ. After the application reads a CQE, it writes a commit or rollback entry to CommQ. The kernel does not reuse CQE slots until the corresponding CommQ entry is received. This is a natural extension of the existing io_uring design.

3. Bilateral microkernel IPC. Adopt the seL4 synchronous IPC model for critical kernel–userspace interactions, with the pipelined commit optimization to amortize latency. This requires a microkernel or hybrid-kernel architecture.

8.4 Performance and design principles

The cost is bounded. seL4's synchronous IPC achieves $\approx 0.2\,\mu$s per round trip on ARM64 [9]; for I/O-bound operations this is negligible against device latency, and pipelined commit reduces per-syscall overhead to a single memory write—far less than a context switch. Compute-bound operations can opt out, as applications that do not need durability omit `fsync()`. The TxF lesson [11] holds: the API must be simple. The design principle is bilateral by default, opt-out for performance. Today's model places the burden on the developer to know when bilateral commit is needed; most do not—hence the PostgreSQL fsync() catastrophe.

8.5 Future work — the link layer below

A bilateral commit protocol at the syscall boundary terminates only when the layer below also commits bilaterally. If the kernel issues a bilateral commit for `send()` and the underlying NIC accepts the bytes onto a wire whose endpoints disagree about link state, the guarantee re-introduces the FITO assumption one layer down. Open Atomic Ethernet [**?**] provides one candidate mechanism for closing the gap at L2 via a Reliable Failure Detector primitive—a sub-millisecond, jointly-observable signal of link failure between

two NIC endpoints. We do not pursue that direction in this paper; the syscall-layer argument stands on its own. But the symmetry is worth flagging for the Netdev community: bilateral commit at the kernel boundary is a partial answer if the link layer below it remains unilateral, and conversely the L2 work needs a bilateral syscall API to surface link-failure events to the application without silent loss. Both layers are missing the same primitive, for the same structural reason.

9 Conclusion

The Unix syscall interface is fire-and-forget. The Kernel Acknowledgment Spectrum classifies syscalls by the highest bilateral confirmation they provide; the vast majority terminate at Level 1 (kernel receipt) or below. Bolt-on mechanisms (`fsync()`, `io_uring_cqe_seen()`, `userfaultfd`) elevate specific operations to Level 2 or Level 3, but no mechanism reaches Level 4 (bilateral semantic commit). PostgreSQL lost data because `fsync()` cleared its error flag after one report; io_uring completion entries can be overwritten before userspace reads them; signal coalescing silently drops events; overcommit issues promises the kernel cannot keep. In every case the failure is silent, the application is unaware, and the divergence between kernel and userspace is permanent. The Linux 6.8 measurements in §3.5 confirm the behavior is current, not historical: false `fsync()` durability in 50/50 trials, up to 1,008 silently buffered io_uring completions per ring, SIGCHLD loss up to 80% at modest concurrency, and 100% doomed-PID forks into a constrained cgroup. The structural resolution is bilateral commit at the kernel–userspace boundary—both sides confirming the operation was proposed, executed, and consumed—and need not be expensive: our userspace Commit-Queue shim (R1) achieves zero silent CQE loss at $\sim$39% overhead, a lower bound on what an in-kernel CommQ could do. The forensic record and the measured rates make bilateral syscall semantics practically necessary, not just theoretically desirable. The question is whether the kernel community recognises the category mistake—treating kernel receipt as kernel commitment—before the next PostgreSQL-scale incident, or after.

Acknowledgments

The author is grateful to Jim Gray (1944–2007), who generously volunteered his time and vision during the author's tenure at Quantum Corporation, where Gray's concept of the storage brick—a commodity building block for scalable, fault-tolerant storage—shaped the author's understanding of what transactional semantics means at the hardware boundary. Gray's two-phase commit protocol remains, four decades later, the definitive example of bilateral commitment in systems design. The structural absence identified in this paper—no commit protocol at the syscall boundary—is precisely the gap that Gray spent his career teaching us to recognize in databases. That the lesson has not yet been applied to the operating system kernel is the category mistake this paper documents.

AI Assistance Disclosure

The empirical harness in §3.5 (reproducers E1–E4 and the Commit-Queue shim R1, in C with a Python aggregation layer) was developed with the assistance of Anthropic's Claude, which scaffolded the reproducers, the Lima/Vz environment, and the plotting and LaTeX for that section under the authors' direction. All experiments ran on real Linux 6.8 hardware; the authors designed the protocol, verified every measurement against the raw logs, and are solely responsible for the claims and conclusions.

References

[1] P. Borrill, "The Forward-In-Time-Only Projection Error: A Category Mistake in Distributed Systems," DÆDÆLUS Technical Report TR-2026-FITO, 2025.

[2] P. Borrill, "Why Email Fails: The Structural Impossibility of Reliable Messaging Without Atomic Commit Semantics," DÆDÆLUS Technical Report TR-2026-WEF, 2026.

[3] P. Borrill, "The Bilateral Efficiency of Ethernet," DÆDÆLUS Technical Report TR-2026-BEE, 2026.

[4] J. Gray, "Notes on Data Base Operating Systems," in *Operating Systems: An Advanced Course*, Springer, 1978.

[5] P. Borrill, "Why Messages Fail: The Acknowledgment Spectrum and the Structural Impossibility of Reliable Messaging Without Bilateral Commit," DÆDÆLUS Technical Report TR-2026-WMF, 2026.

[6] T. Ts'o, J. Corbet, "PostgreSQL's fsync() surprise," LWN.net, April 2018. `https://lwn.net/Articles/752063/`

[7] C. Hanson, "PostgreSQL's handling of fsync() errors is unsafe and has unresolved problems," PostgreSQL Wiki, 2018. `https://wiki.postgresql.org/wiki/Fsync_Errors`

[8] containerd contributors, "Fix container pid race condition #3857," GitHub, 2020. `https://github.com/containerd/containerd/pull/3857`

[9] G. Klein, K. Elphinstone, G. Heiser, et al., "seL4: Formal Verification of an OS Kernel," in Proc. 22nd ACM SOSP, 2009.

[10] A. Arcangeli, M. Raghavendra, "Userfaultfd," Linux kernel documentation, 2015. `https://man7.org/linux/man-pages/man2/userfaultfd.2.html`

[11] Microsoft, "Alternatives to using Transactional NTFS," Win32 Documentation, 2023. https://learn.microsoft.com/en-us/windows/win32/fileio/deprecation-of-txf

[12] P. Borrill, "The Bilateral Insecurity of Internet Protocol," DÆDÆLUS Technical Report TR-2026-003, 2026.

[13] J. Corbet, "ext4 and data loss," LWN.net, March 2009. https://lwn.net/Articles/322823/