

The localhost Tax: Transparent TCP Splicing for Co-located Services

Cong Wang

Founder and CEO, Multikernel Technologies, Inc.

P99 CONF

Your Services Share a Host

- Microservices, sidecars, and caches increasingly land on the same machine
- App to sidecar, service to Redis, service to local database hop
- Two processes, microseconds apart, talking through the full TCP/IP stack

"The bytes travel micrometers. The code path travels the whole stack."

The Same-Host TCP Tax

Every local request still pays for:

- skb allocation and management for every segment
- softirq scheduling and the per-CPU backlog
- The loopback device, or worse: veth + bridge, twice per hop
- The entire TCP receive engine, just to re-queue local bytes

For thin, network-bound services (Redis, Memcached, gRPC), this tax **dominates** each request.

tcp_splice in One Slide

An out-of-tree Linux kernel module:

- **Detects** a co-located TCP connection via a `sock_ops` BPF program
- **Pairs** both endpoints through a module kfunc, no sockmap
- **Moves** `sendmsg` / `recvmsg` through a small in-kernel ring
- The packet path is bypassed entirely; the TCP connection stays fully alive

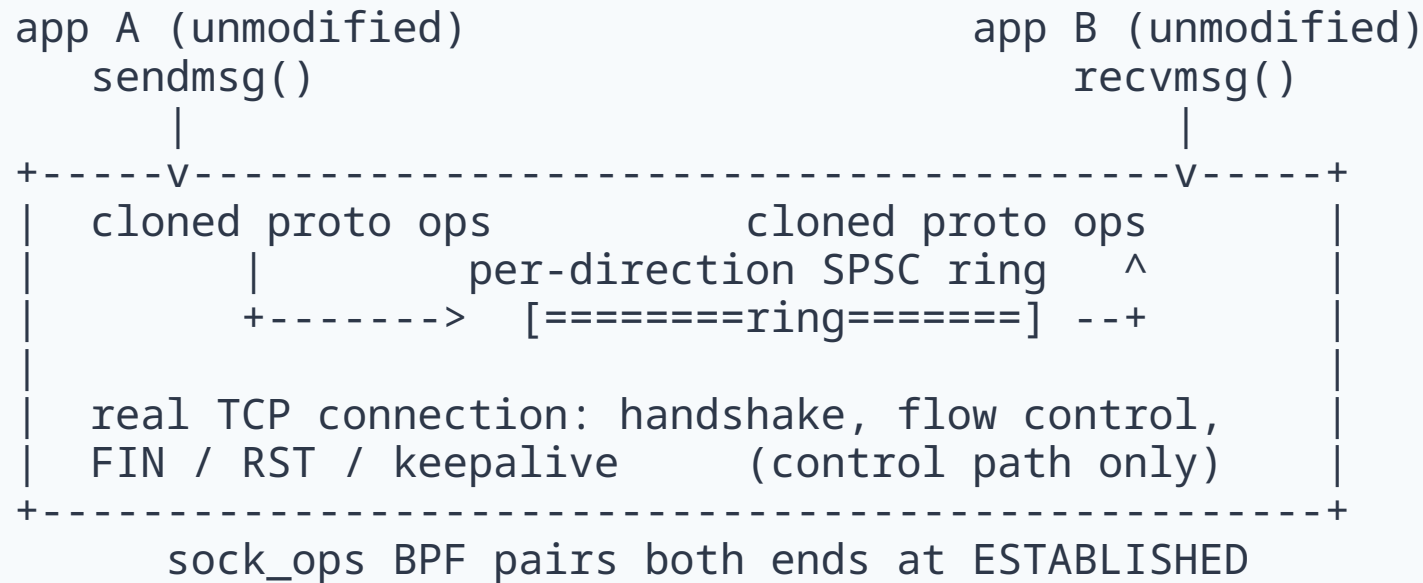
"Load a module, attach a policy to a cgroup, done."

The Payoff First

workload	baseline	splice	
Redis GET, latency-bound (<code>-c 1</code>)	29 μ s RTT	12 μ s RTT	2.5x
Redis GET, saturated (<code>-c 50</code>)	131k ops/s	214k ops/s	1.6x
Container TCP_STREAM, 4 KB	8.1 Gbit/s	48.8 Gbit/s	6.0x
TCP_RR 1 B + busy-poll	110k tps	1114k tps	10.0x

- Real Redis, real containers over veth + bridge, no application changes

Architecture



- Data takes the ring; control stays on the untouched TCP connection

Pairing with sock_ops BPF

- A `sock_ops` program attached to a cgroup calls the module's kfunc from each endpoint's ESTABLISHED callback
- Both ends are matched in a private registry by a **canonical, direction-independent 4-tuple**, no sockmap
- Loopback keys fold in the netns: unrelated `127.0.0.1` flows in different namespaces never mis-pair
- Veth container IPs match across namespaces, so the two ends still find each other
- Only one end pairs? That socket just keeps using plain TCP

Interposing the Data Path

- Each paired socket's `sk_prot` is replaced with a **clone** that overrides `sendmsg` / `recvmsg` / `poll` / `close`
- The swap happens per side, under that socket's own lock
- Everything else (FIN, RST, keepalive, timers) falls through to the original TCP ops
- No core kernel patch, no sockmap, no userspace proxy, no recompile

"There is no protocol to re-implement, so there is nothing to break."

The Lock-Free Ring

- One SPSC byte ring **per direction**: one producer socket, one consumer socket
- Data path is lock-free: head/tail advance with release/acquire stores
- Producer and consumer cursors on **separate cache lines**; producer keeps a private `cached_tail` (standard SPSC cursor caching)
- One copy in (`sendmsg` iov to ring), one copy out (ring to `recvmsg` iov)
- Compare baseline: skb alloc, segmentation, softirq, backlog, receive engine, then the same two copies anyway

Why the Stream Never Reorders

The one dangerous window: data sent over TCP *before* the second end pairs.

- The sender records `ring_seq`: the exact sequence number where it switched to the ring
- The receiver drains all startup TCP **below** `ring_seq` **first**, then switches to the ring
- After that, the ring is the **sole** data path: no per-write TCP fallback, so ring bytes and TCP bytes can never interleave
- A full ring backpressures the sender exactly like a full send buffer

TCP Stays in Charge

- Sequence numbers stay frozen at post-handshake values
- FIN, RST, and keepalive travel over the real connection, untouched
- Flow control still exists: the ring provides it (full ring blocks the sender)
- Close tears down the pair and the connection finishes as ordinary TCP
- `ss` still shows the connection; your tooling keeps working

Deployment: One Command

```
sudo insmod tcp_splice.ko

sudo tcp-splice-ctl enable --cgroup /sys/fs/cgroup \
    --loopback-only --ports 6379

sudo tcp-splice-ctl status
# tcp_splice: enabled (loopback_only=1, ports=6379)
```

- Policy is a **pinned BPF link**: survives the tool exiting, `disable` removes it
- Scope by cgroup, restrict by port list or loopback-only
- Both endpoints must fall under the attached cgroup

Benchmark: Request/Response

netperf TCP_RR, transactions/sec, loopback, adjacent CPUs:

msg size	baseline	splice	splice + busy-poll
1 B	110.2k	267.0k (2.4x)	1113.8k (10.0x)
1 KB	105.8k	235.1k (2.2x)	713.0k (6.7x)
16 KB	40.5k	89.6k (2.2x)	123.7k (3.1x)
64 KB	17.8k	20.9k (1.2x)	40.5k (2.3x)

- Container TCP_RR tracks loopback closely (1 KB: 100.4k, 233.9k, 704.9k)

Benchmark: Bulk Throughput

netperf TCP_STREAM, container to container (veth + bridge), Mbit/s:

msg size	baseline	splice
1 KB	3710	21326 (5.8x)
4 KB	8084	48834 (6.0x)
16 KB	26083	27788 (1.1x)

- The per-skb veth + bridge cost is exactly what the ring sidesteps
- Bare-metal loopback bulk is roughly neutral: TSO already amortizes per-packet cost there

Benchmark: A Real Redis

`redis-benchmark` GET vs `redis-server`, two containers, veth + bridge:

concurrency	baseline	splice
<code>-c 1</code> (latency-bound)	35k ops/s	85k ops/s (2.5x)
<code>-c 50</code> (saturated)	131k ops/s	214k ops/s (1.6x)

- Per-GET round trip: **29 μ s down to 12 μ s**
- Below synthetic RR because even Redis's few microseconds of parse/lookup/encode per op dilute the network saving

Busy-Poll: Killing the Wakeup

- For blocking readers, the splice receiver **spins on the ring** before parking
- Budget comes from the mainline `net.core.busy_read` sysctl (or `SO_BUSY_POLL` per socket); `0` disables
- Why not `sk_busy_loop()`? Loopback and veth deliver via the per-CPU backlog and expose **no pollable** `napi_id`: generic busy-poll is a no-op there
- We borrow only the budget, never the NAPI machinery
- Epoll-driven servers (Redis) never enter the spin, so it costs them nothing

Predicting Your Win

$$\text{speedup} \approx \frac{t_{\text{net}} + t_{\text{app}}}{t_{\text{ring}} + t_{\text{app}}} \quad (t_{\text{net}} \gg t_{\text{ring}})$$

- The splice removes most of `t_net` (the co-located hop); `t_app` (your per-op userspace work) is untouched
- Thin services (Redis, Memcached, gRPC hops): `t_app` is a few μs , win is 2x or more
- Heavy servers (an L7 proxy doing real work): `t_net` is already a small fraction, expect less
- Quick test: profile one request; the co-located network share is your ceiling

Bugs Only Real Traffic Found (1/2)

Found running pgjdbc/BenchBase over a co-located Citus cluster:

Spurious EOF on a not-yet-paired socket

- The proto is installed on the passive child at `TCP_SYN_RECV`, *before* the pair forms
- The receive path treated "no peer" as "peer gone", so an early read returned EOF
- The pgjdbc client saw the server hang up before it even sent its startup packet
- Fix: an explicit `unpaired` flag set only on real teardown; a parked channel **waits**

Bugs Only Real Traffic Found (2/2)

Final message lost on write-then-close

- On peer teardown the receiver discarded unread ring data
- A sender that wrote a reply and immediately closed lost it: the data was still valid
- Fix: drain your own ring **before** reporting EOF

Module ref leak on aborted passive sockets

- A child that is reset or never accepted dies via `->destroy`, which the proto clone did not override, so teardown never ran
- Every leaked ref pinned the module; unload needed a reboot

- Fix: hook `->destroy`; make teardown idempotent via `xchg`



Lessons for Kernel Data Paths

- Synthetic benchmarks validate performance; only **production-shaped traffic** validates correctness
- Connection churn, early reads, abrupt closes, and aborted accepts are where splice-like designs die
- Every bug became a regression test: churn, poll, bidirectional-write deadlock
- Keeping real TCP as the control path meant every failure degraded to normal TCP semantics, never to silent corruption

Takeaways

- Co-located TCP pays a real tax: skbs, softirqs, veth + bridge, the receive engine
- tcp_splice removes it transparently: module + `sock_ops` BPF + in-kernel ring
- The TCP connection stays intact, so nothing is re-implemented and nothing breaks
- Redis RTT 29 μ s to 12 μ s; container throughput up to 6x; busy-poll RR up to 10x
- Aim it with the model: the thinner the service, the bigger the win

"If your services share a host, you are leaving microseconds on the table on every hop."

Questions and Feedback

Contact: cwang@multikernel.io

Project: github.com/multikernel/tcp_splice