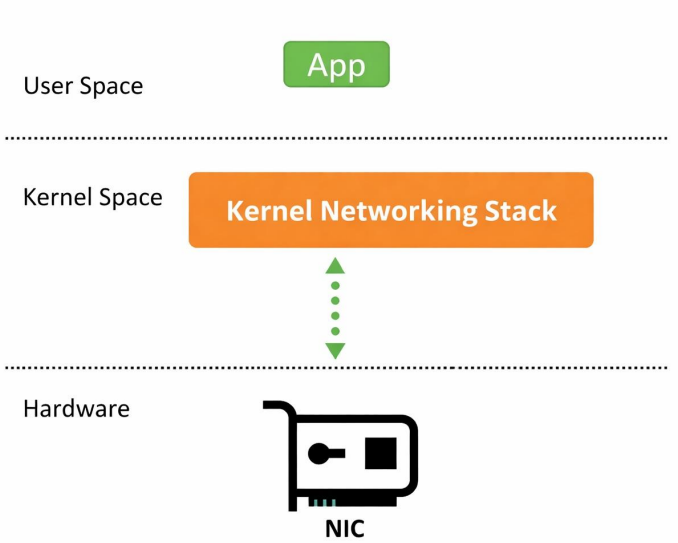
Networking in Kernel Programming: Basics, Performance, and Modern Alternatives

Why do we care about networking in kernel programming

- Most applications use the network through the kernel
- Performance bottlenecks often come from the kernel data path
- Bugs in networking code often involve buffers, queues, interrupts, and concurrency
- Understanding the kernel path helps explain latency, throughput, drops, and CPU overhead

Overall data path

- The application runs in user space
- The networking stack runs mostly in kernel space
- The network card is hardware



So the packet journey is:

App → Kernel networking stack → NIC hardware → wire → NIC hardware → Kernel networking stack → App

How does kernel networking stack work

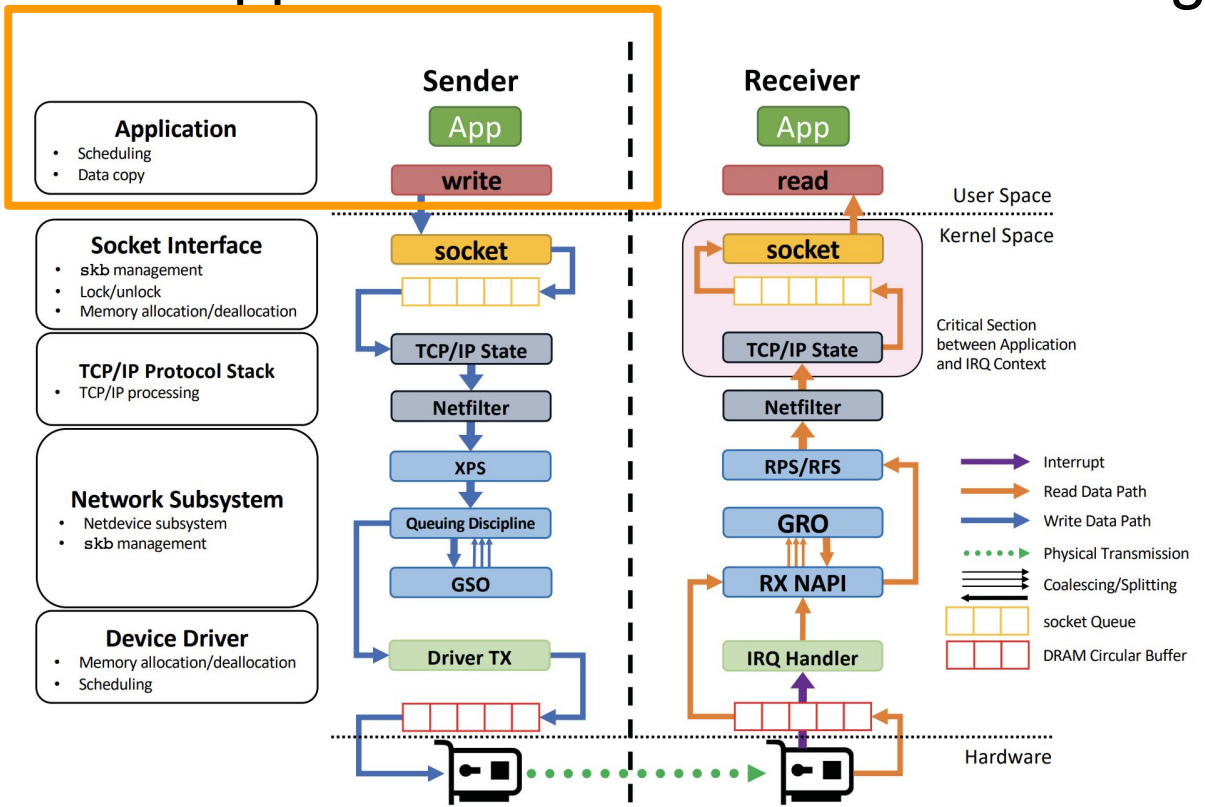
The app only sees:

- `socket()`
- `connect()`
- `write()` / `send()`
- `read()` / `recv()`

```c
int fd = socket(AF_INET, SOCK_STREAM, 0);
connect(fd, ...);
write(fd, buf, len);
read(fd, buf, len);
```

From the application’s point of view, networking looks simple. But the kernel has to do a lot of work behind that simple interface.

How application access network through sockets

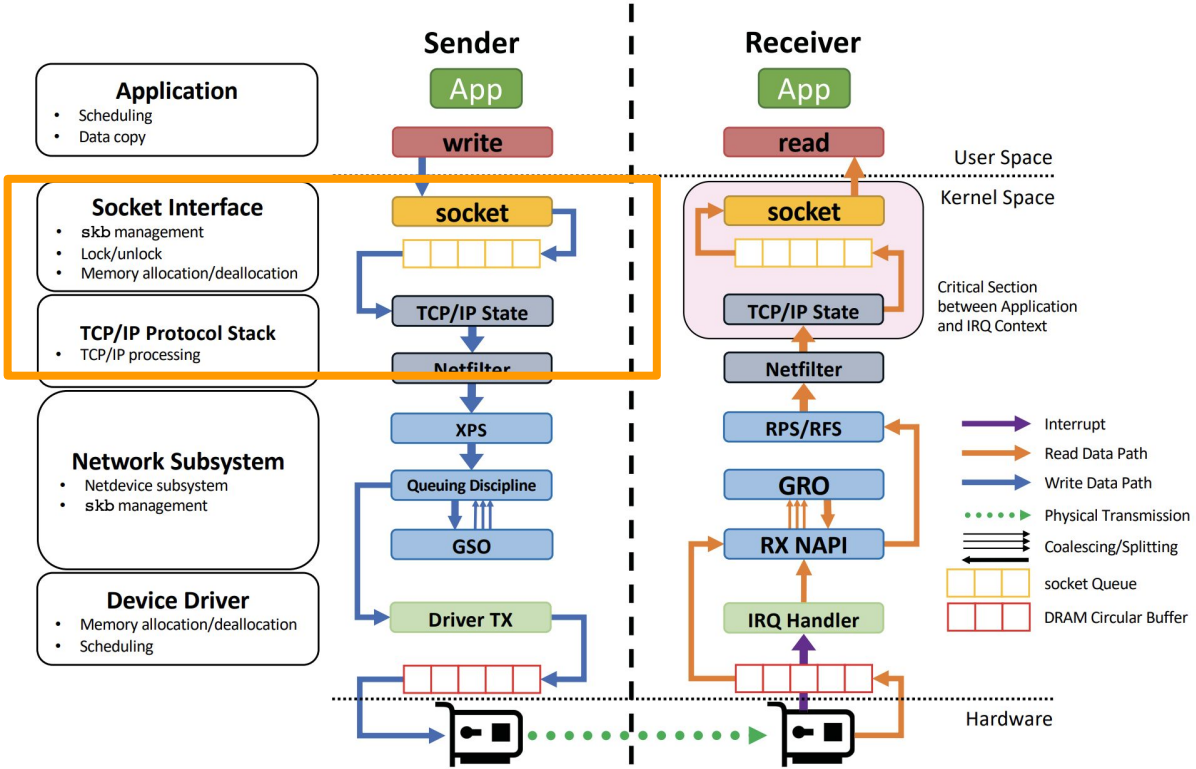


A socket is the application's handle to a communication endpoint.

1. Application creates a socket
2. kernel allocates an internal socket object and associated state
3. Socket object is connected to the rest of the kernel network stack

*SIGCOMM2021 understanding host network stack overheads

How does kernel handle application data

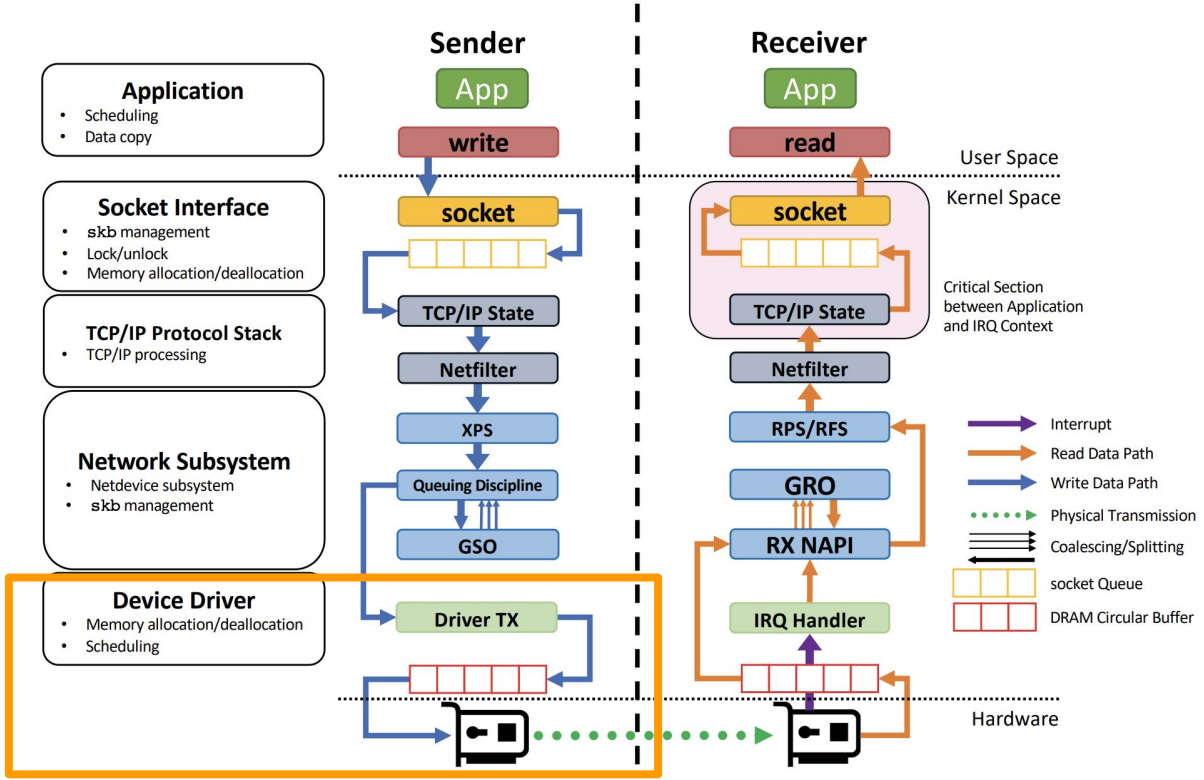


Specifically, when application sends data:

- Control moves from user space to kernel space
- Kernel verifies the socket and permissions
- Data is copied into kernel socket buffers
- TCP/IP stack packages and manages the data for sending

*SIGCOMM2021 understanding host network stack overheads

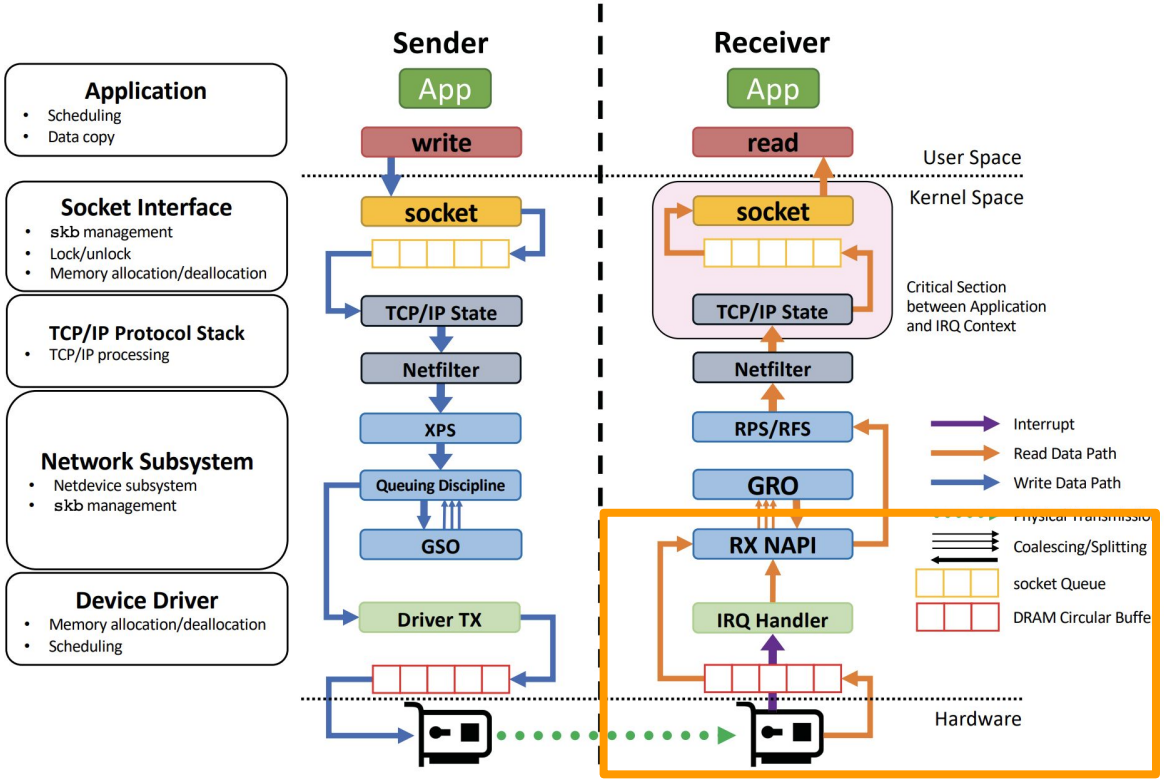
How data leaves kernel for transmission



- Data sit in send buffer
  - NIC is busy?
  - TCP congestion window is full?
  - Not the right transmission time yet?
- Kernel sends the packet down the network stack
- Device driver queues it for the NIC
- NIC transmits it onto the wire

*SIGCOMM2021 understanding host network stack overheads

What about Receiver side

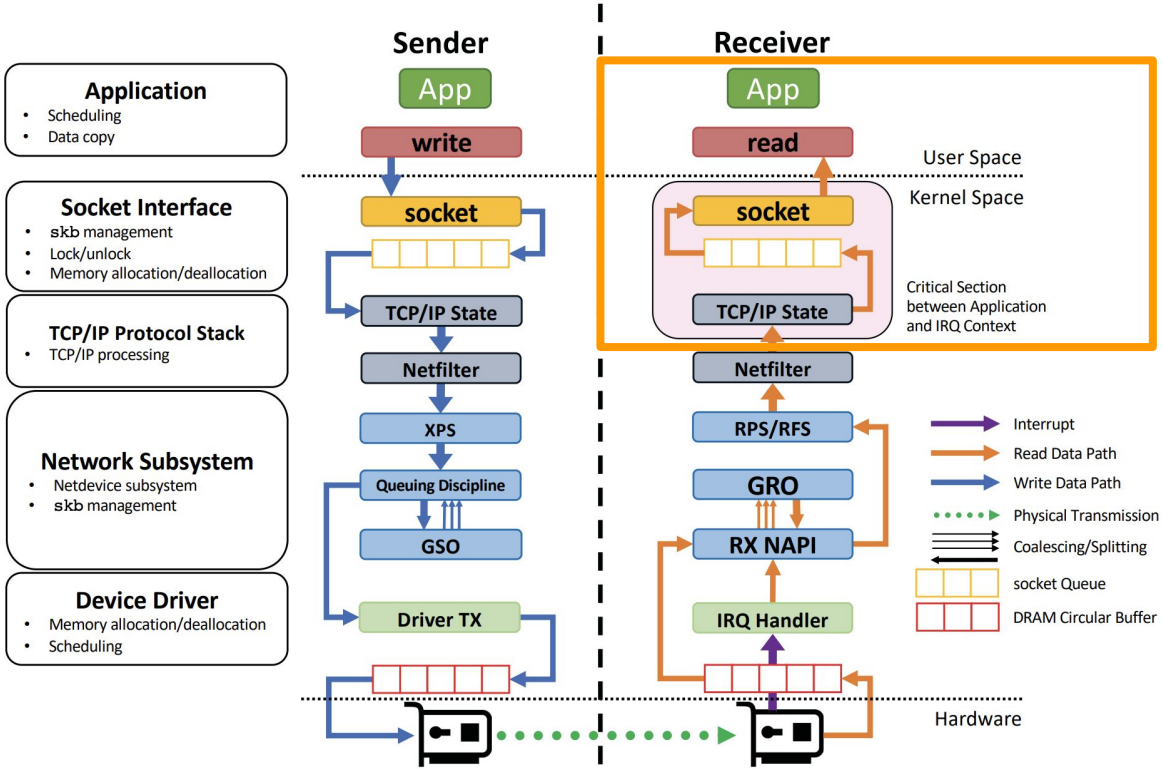


- Packets arrive.
- NIC raises an interrupt.
- Hard IRQ handler runs briefly and schedules NAPI, also disables interrupt for that queue
- NAPI polls and processes up to a budget of packets.

But if packets keep coming, can networking starve user space in practice?

*SIGCOMM2021 understanding host network stack overheads

What about Receiver side



- TCP/IP layer processes packet and appends to socket receive queue.
- If a userspace thread is blocked in `recv()`/`read()`, that thread is woken.
- Thread copies data from kernel socket buffers to userspace buffer.
- If NAPI poll drained all work, interrupts are re-enabled for the queue.

*SIGCOMM2021 understanding host network stack overheads

What can go wrong

- What happens if the machine crashes while packets are in the queue?
    - kernel starts fresh
    - NIC is reinitialized by the driver
    - all old connections are gone
    - all queues are rebuilt from scratch

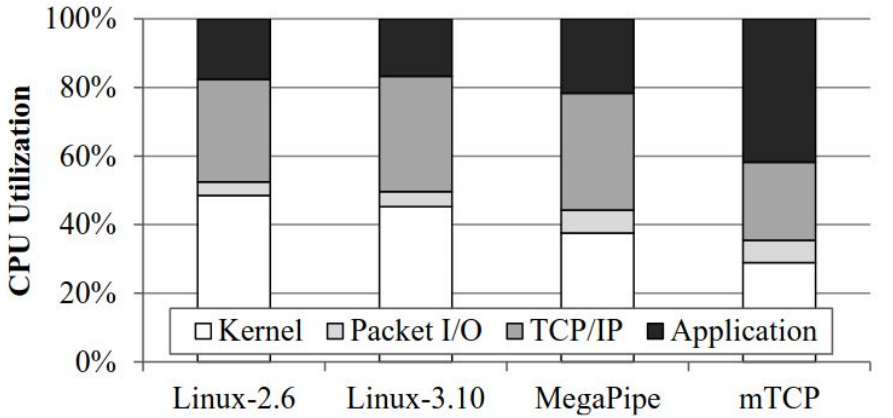
loss of all in-memory network state and abrupt termination of all communication!

- Solution?
    - kernel networking handles this by recovery, not preservation
    - reconnects, and retries
    - E.g. TCP retransmission

Why kernel networking can be slower

- Poor connection locality
- Shared file descriptor space
- Inefficient per-packet processing
- System call overhead

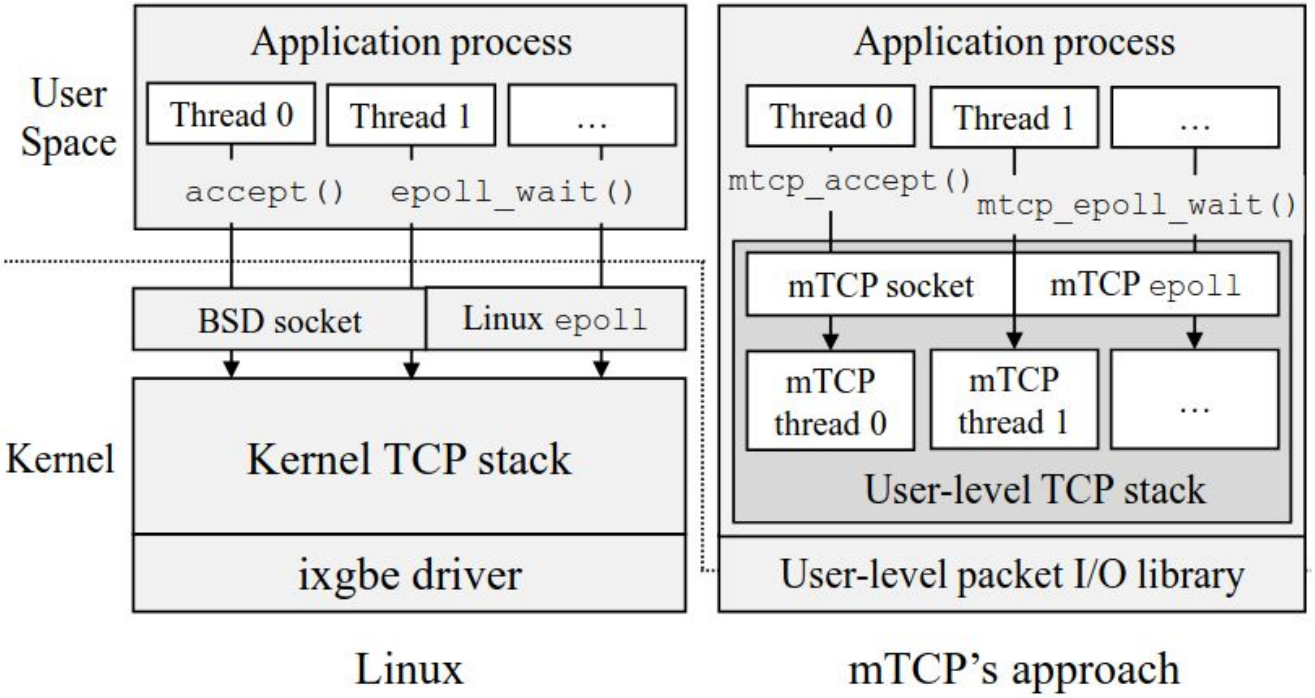
Why do we care about percentage of CPU cycles used by kernel?



80% of CPU cycles are spent in kernel!

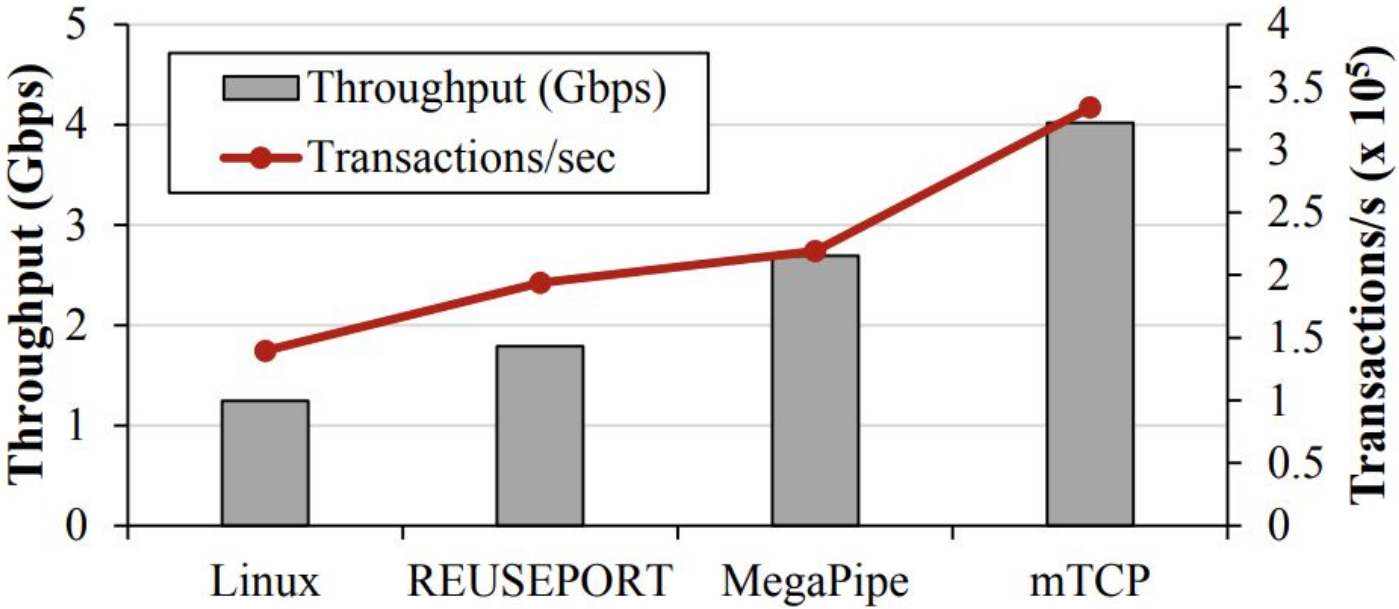
*NSDI 14' mTCP

Kernel networking VS. user-space networking



*NSDI 14' mTCP

Performance Improvement



What do we lose if we move networking to user space

- The use of shared memory space offers limited protection between the TCP stack and the application.
- Moving the TCP stack will also bypass all existing kernel services, such as the firewall and packet scheduling.
- Need to rebuilt the kernel’s natural multi-application sharing model otherwise limit one application per NIC port.

What if I move all kernel functionalities in user space?

Can we do better?

Depends on what performance metrics you are looking for.

- Compatibility and less engineering pain?
  - a faster kernel-based path with zero-copy and batching
- Maximum packet-rate performance?
  - a user-space fast path like DPDK
- the lowest CPU overhead / latency?
  - RDMA or hardware offload (e.g., smartNIC)