Rakaia: Scalable In-Kernel Scheduling for TCP-Based RPCs

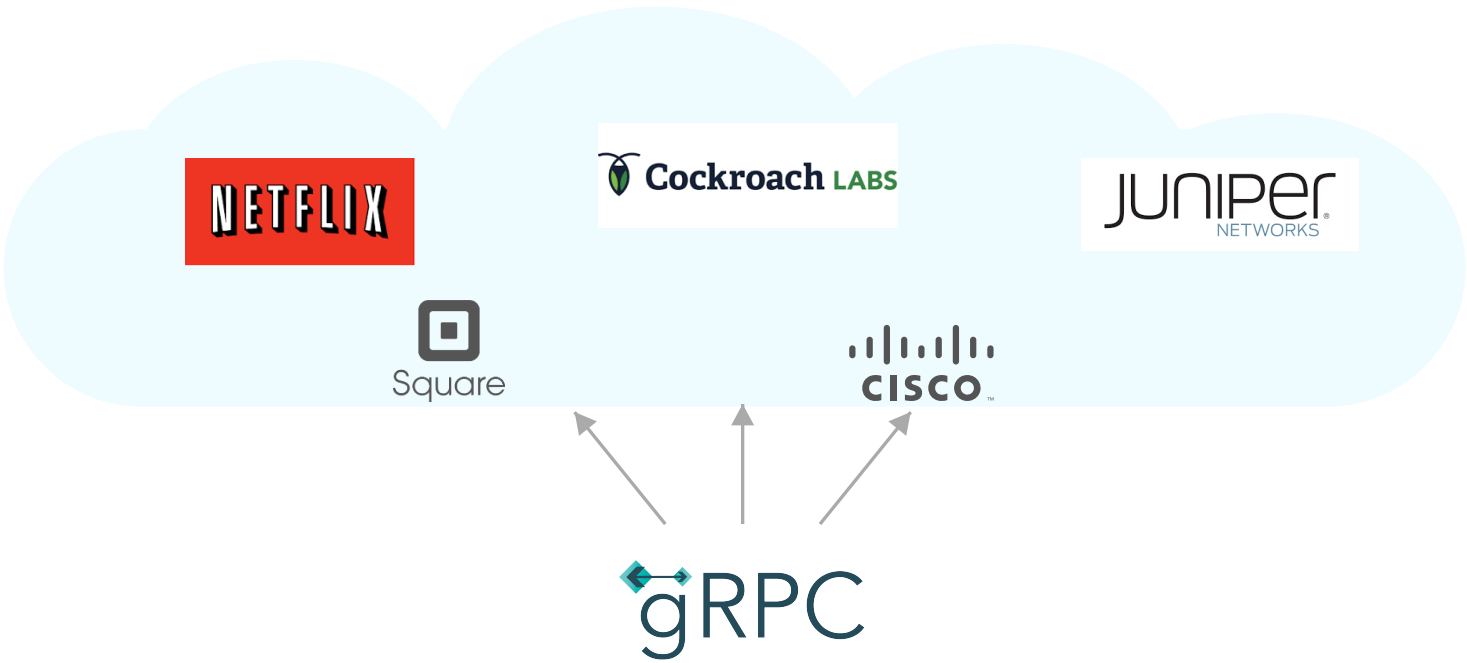


Rui Yang Konstantinos Prasopoulos Edouard Bugnion

EPFL

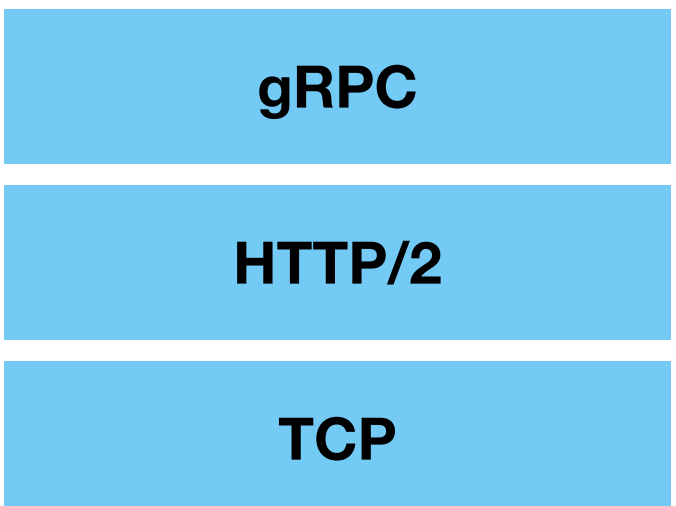
OSDI 2026

gRPC is the cornerstone for modern infrastructure



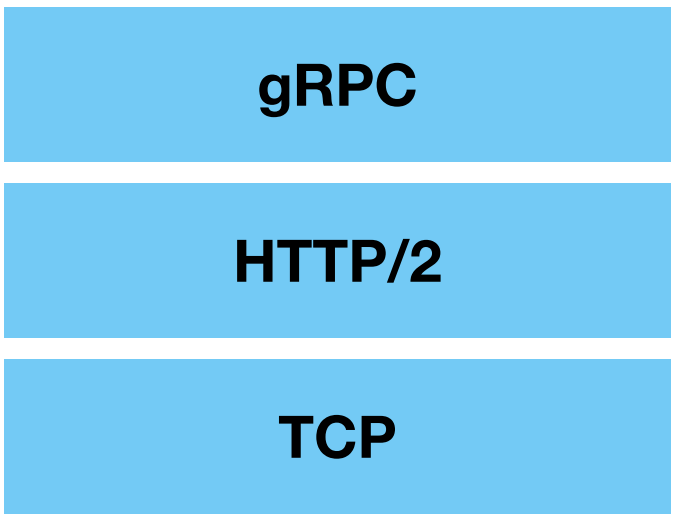
gRPC is the cornerstone for modern infrastructure

- Layered atop TCP and HTTP/2
  - compatibility and deployability
- Multi-languages support
  - via generated Protobuf stubs and language libraries



Yet, we all know gRPC is “slow”[1][2]

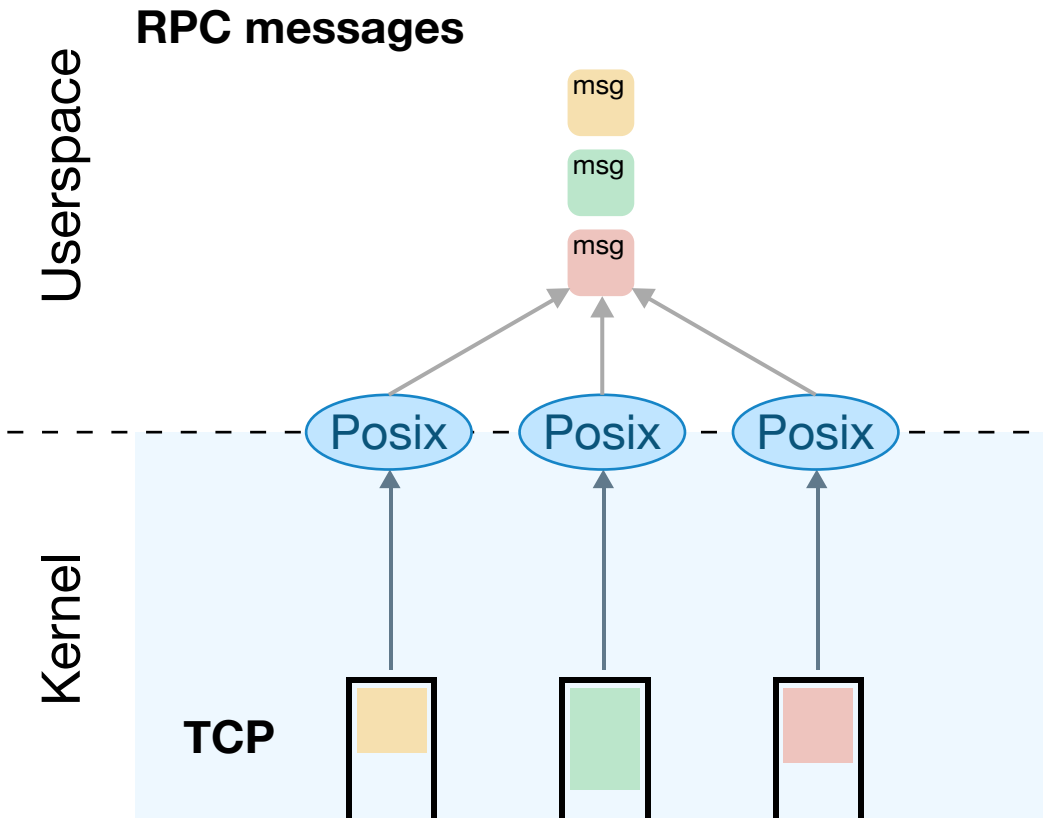
- Serialization overhead
  - protobuf marshalling and unmarshalling
- Transport overhead
  - kernel TCP/IP and TLS
  - HTTP/2 framing and flow control



[1] Kalia, Kaminsky, and Andersen, “Datacenter RPCs can be General and Fast,” NSDI 2019.
[2] Xue et al., “Fast Distributed Deep Learning over RDMA,” EuroSys 2019.

The hidden cost is its architecture design

- The POSIX API provides a stream abstraction
- RPC frameworks require a message abstraction



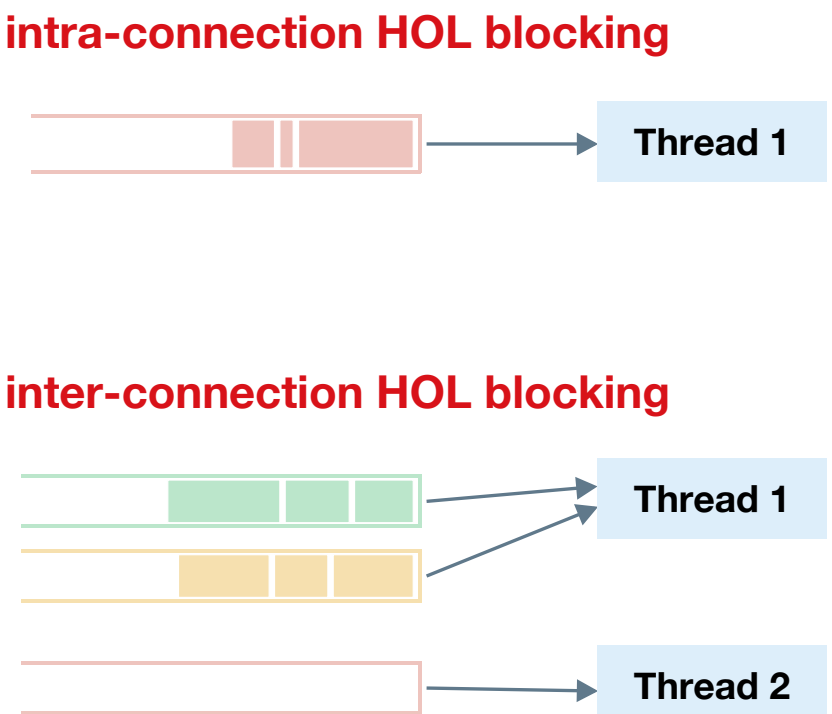
Streams create head-of-line (HOL) blocking

- Intra-connection HOL blocking
  - a slow RPC delays later RPCs on the same TCP stream



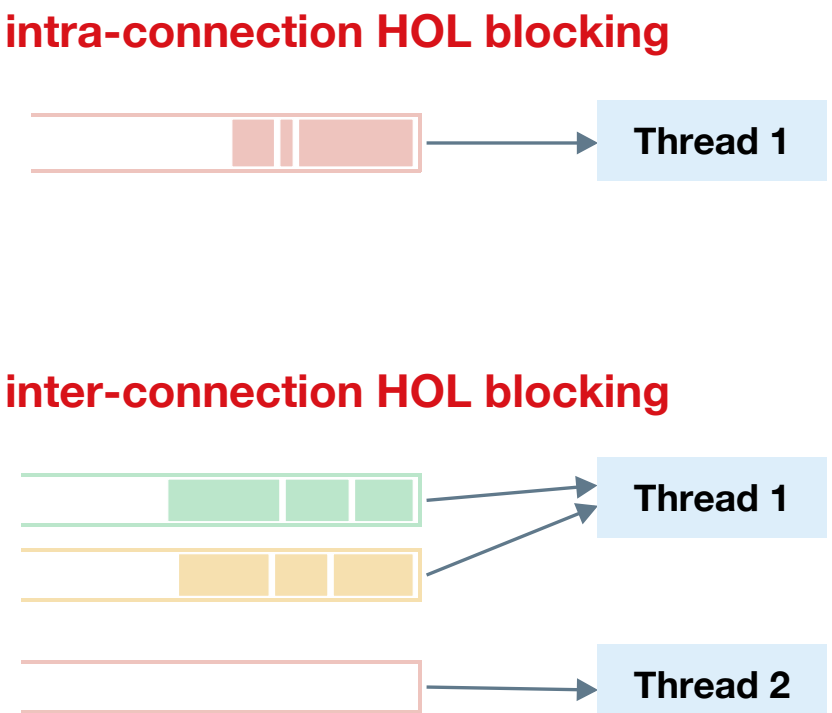
Streams create head-of-line (HOL) blocking

- Intra-connection HOL blocking
  - a slow RPC delays later RPCs on the same TCP stream
- Inter-connection HOL blocking
  - busy connections accumulate work while other cores sit idle



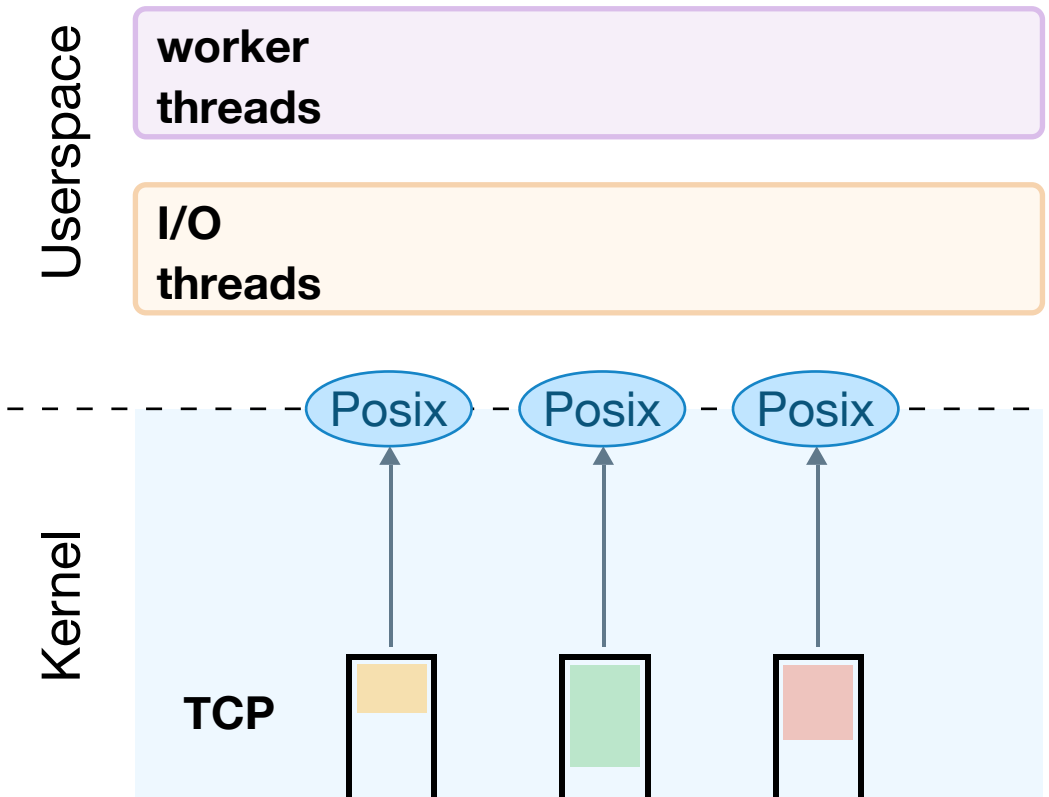
Streams create head-of-line (HOL) blocking

- Intra-connection HOL blocking
  - a slow RPC delays later RPCs on the same TCP stream
- Inter-connection HOL blocking
  - busy connections accumulate work while other cores sit idle
- RPC servers need message-level scheduling across connections and cores



The hidden cost is its architecture design

- The POSIX API provides a stream abstraction
- RPC frameworks requires a message abstraction
- **gRPC Solution**: dedicated IO threads reconstruct messages and hand them to worker threads



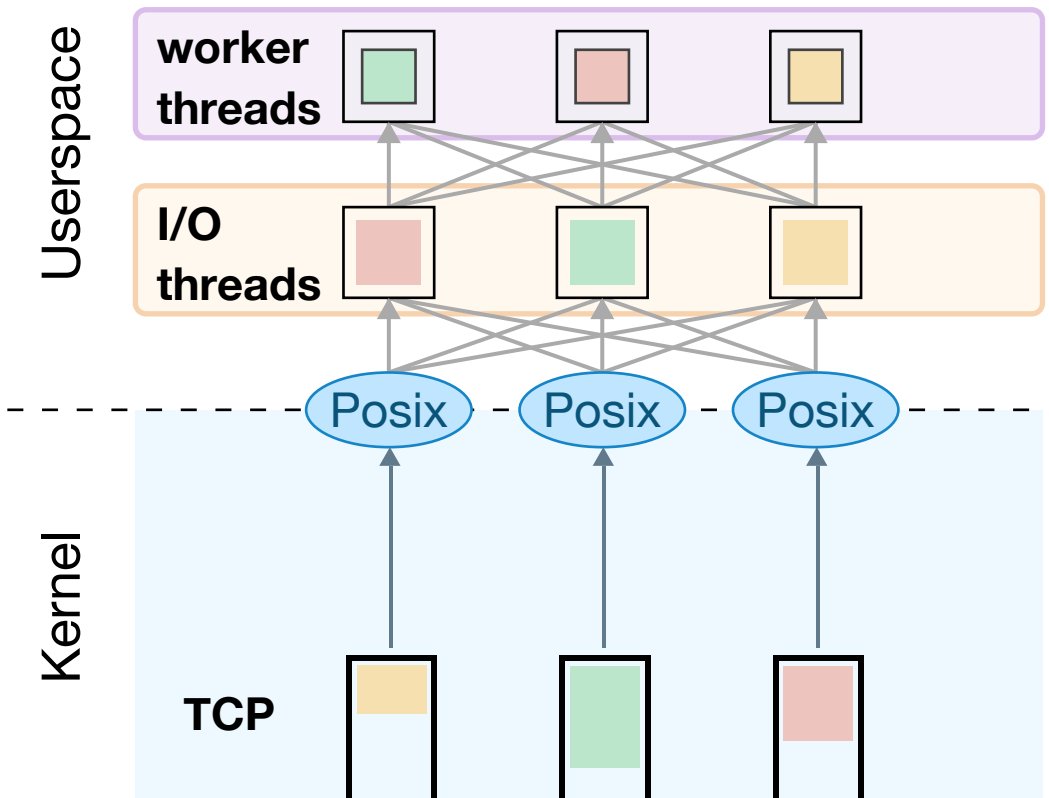
The hidden cost is its architecture design

Pros

- Eliminate both intra- and inter-connection HOL blocking

Cons

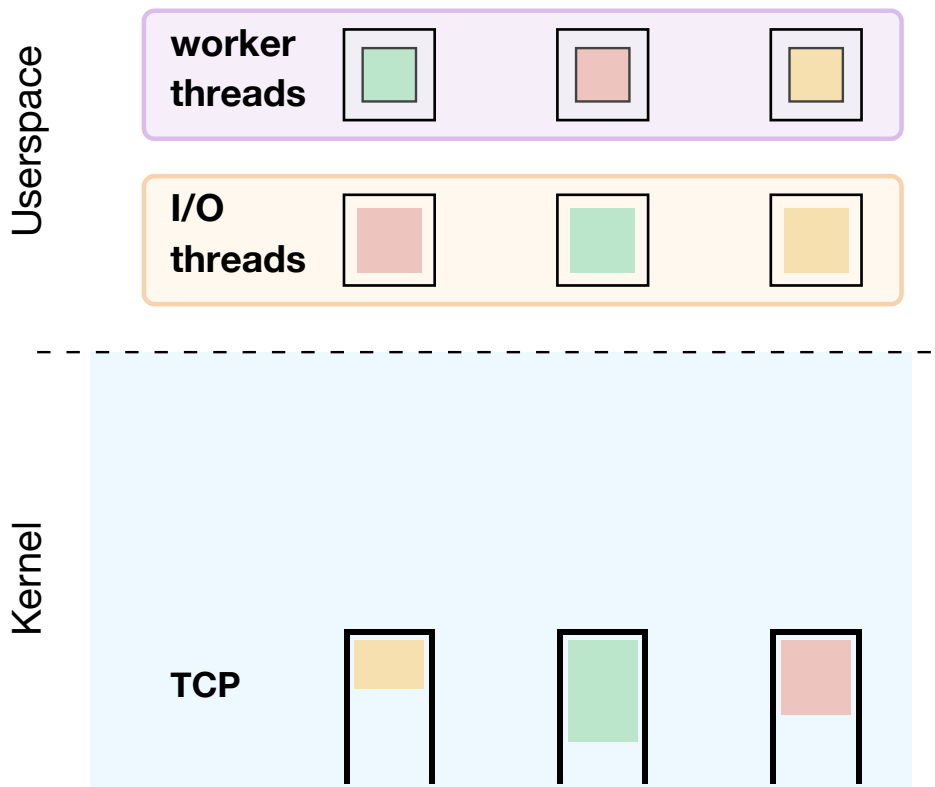
- This threading model brings overhead (e.g., context switching, locking, and scheduling)



Can we keep kernel TCP/IP compatibility with HOL-blocking-free scheduling

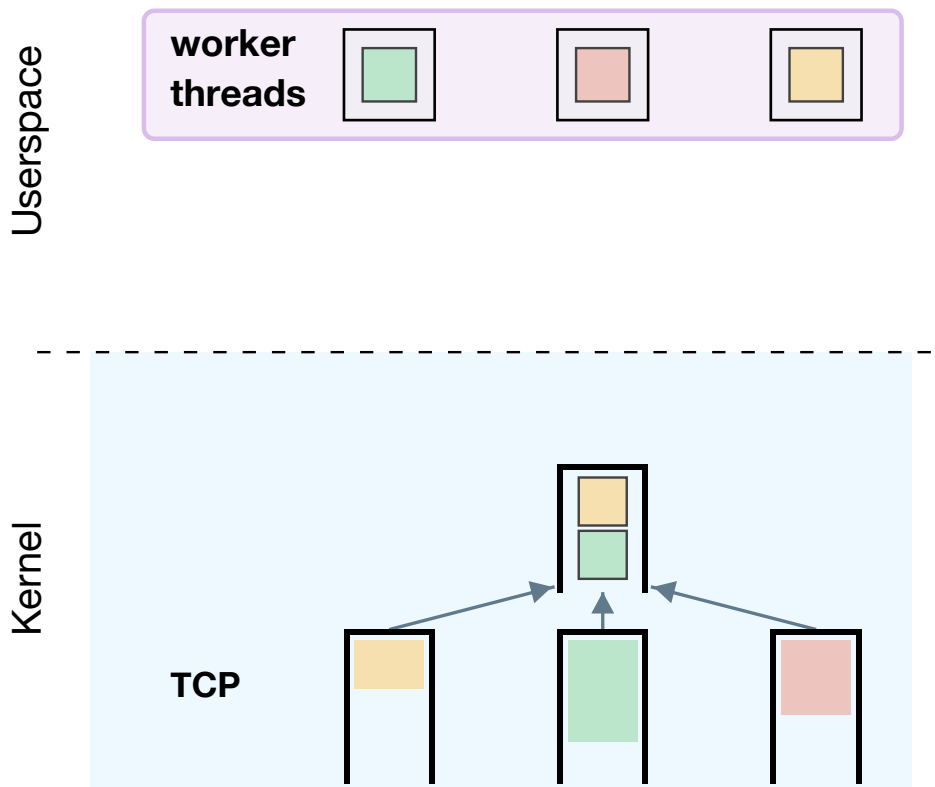
and still achieve high-performance RPCs?

Rakaia design



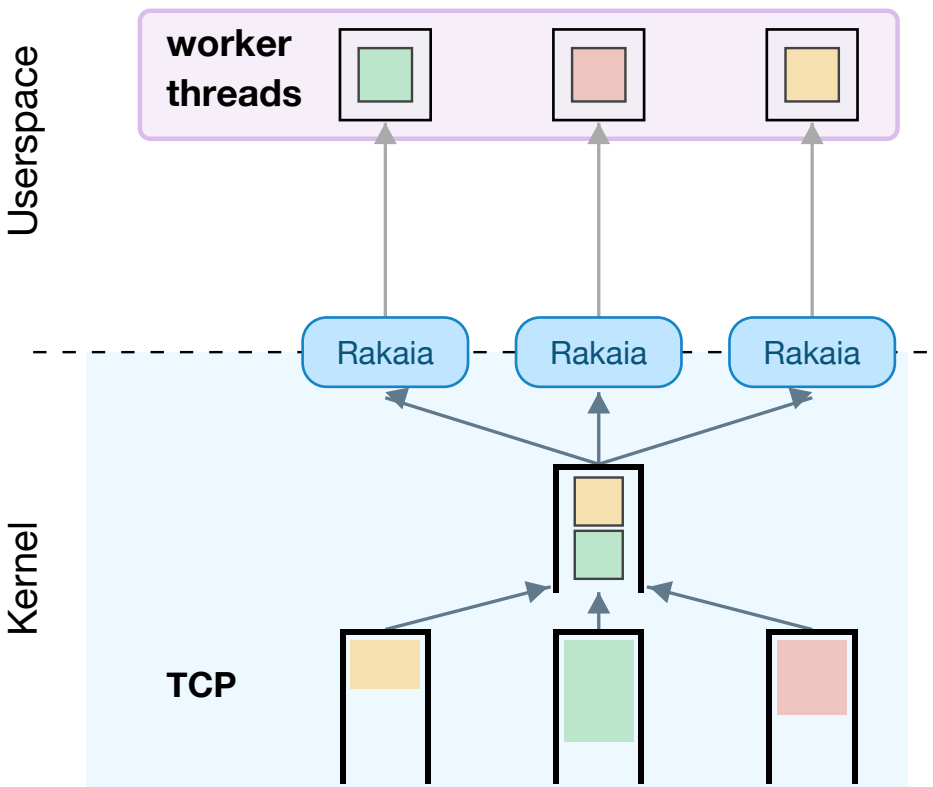
Rakaia design

- Message parsing should be done in the same context that processes TCP packets.
  - message-oriented API (e.g., Homa, KCM)
  - kick IO threads out of the picture



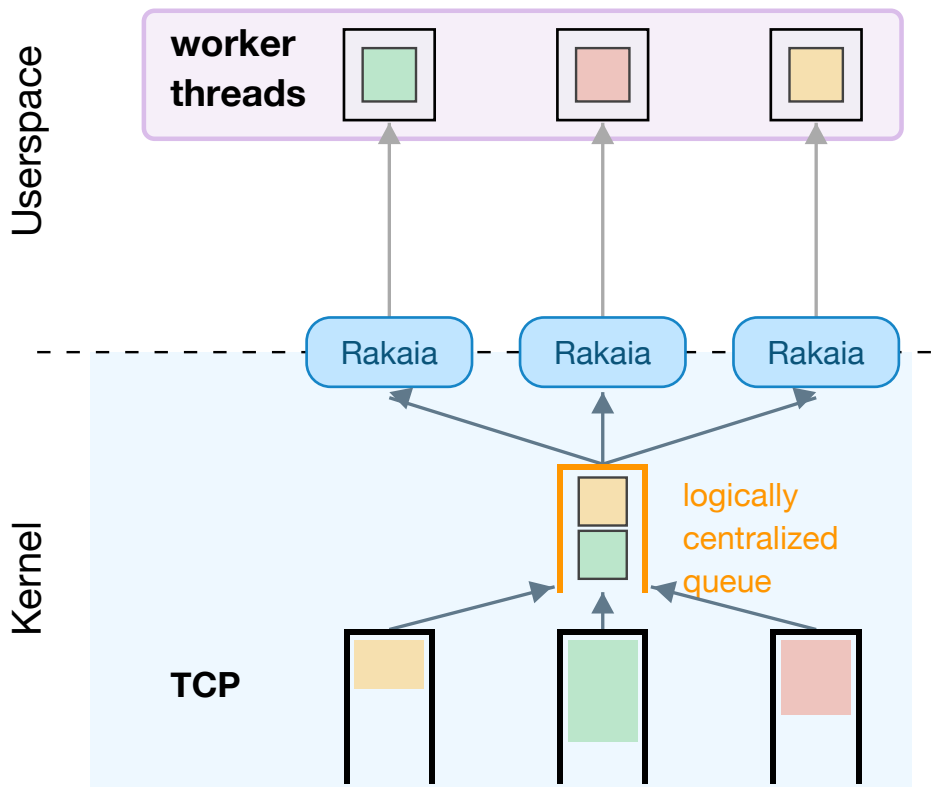
Rakaia design

- Message parsing should be done in the same context that processes TCP packets.
  - message-oriented API (e.g., Homa, KCM)
  - kick IO threads out of the picture
- Connection abstraction should be hidden from the application-facing API.
  - allow message-oriented scheduling



Rakaia design

- Logically centralized queue in the kernel
  - Best of two for message placement
- TX delegation to avoid TCP lock contention
  - senders enqueue without taking TCP lock



Implementation and Evaluation

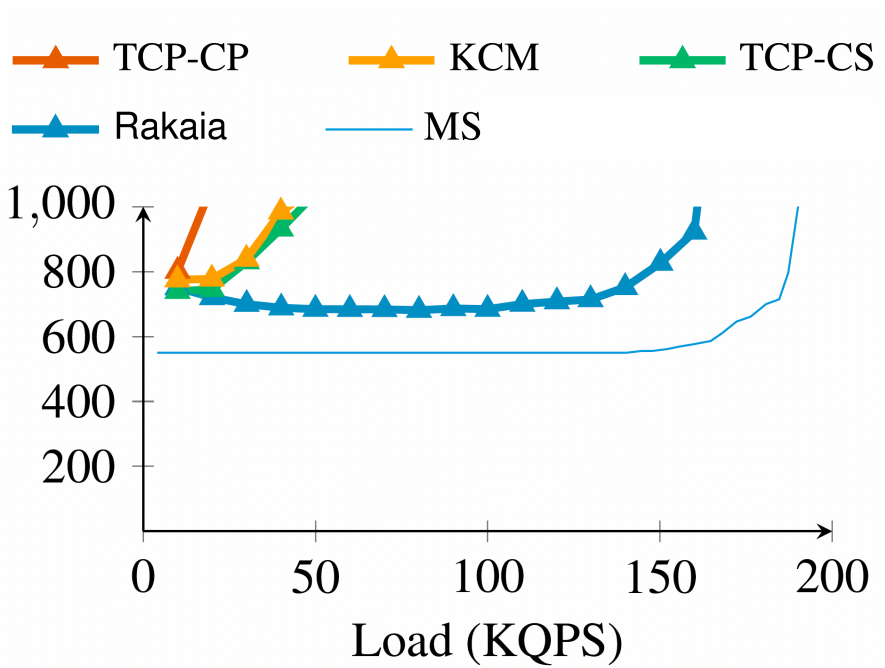
Implementation

- Linux 6.8 kernel module
- 3k-line C code and 60-line kernel patch
- compatible with default TCP/IP stack and TLS

Workloads

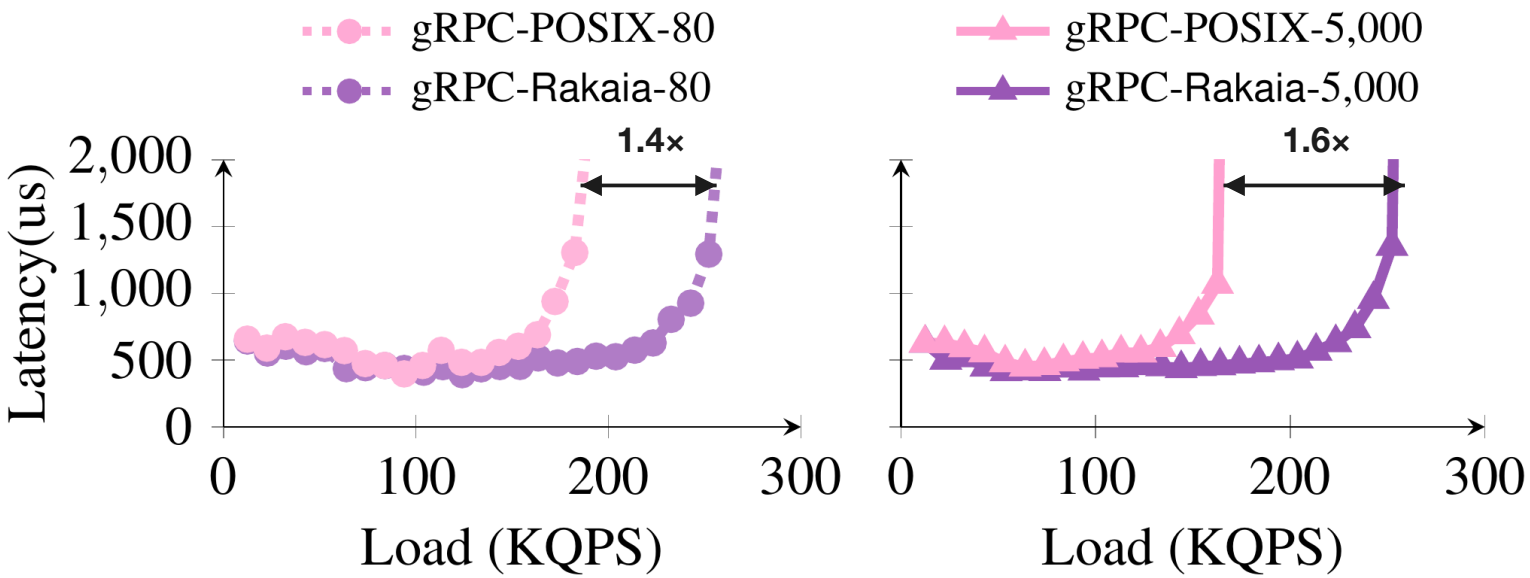
- Synthetic: plain-TCP RPCs
- Synthetic: gRPC-Go and gRPC-C++
- Applications: Silo/TPC-C and OpenTelemetry

Plain-TCP RPCs



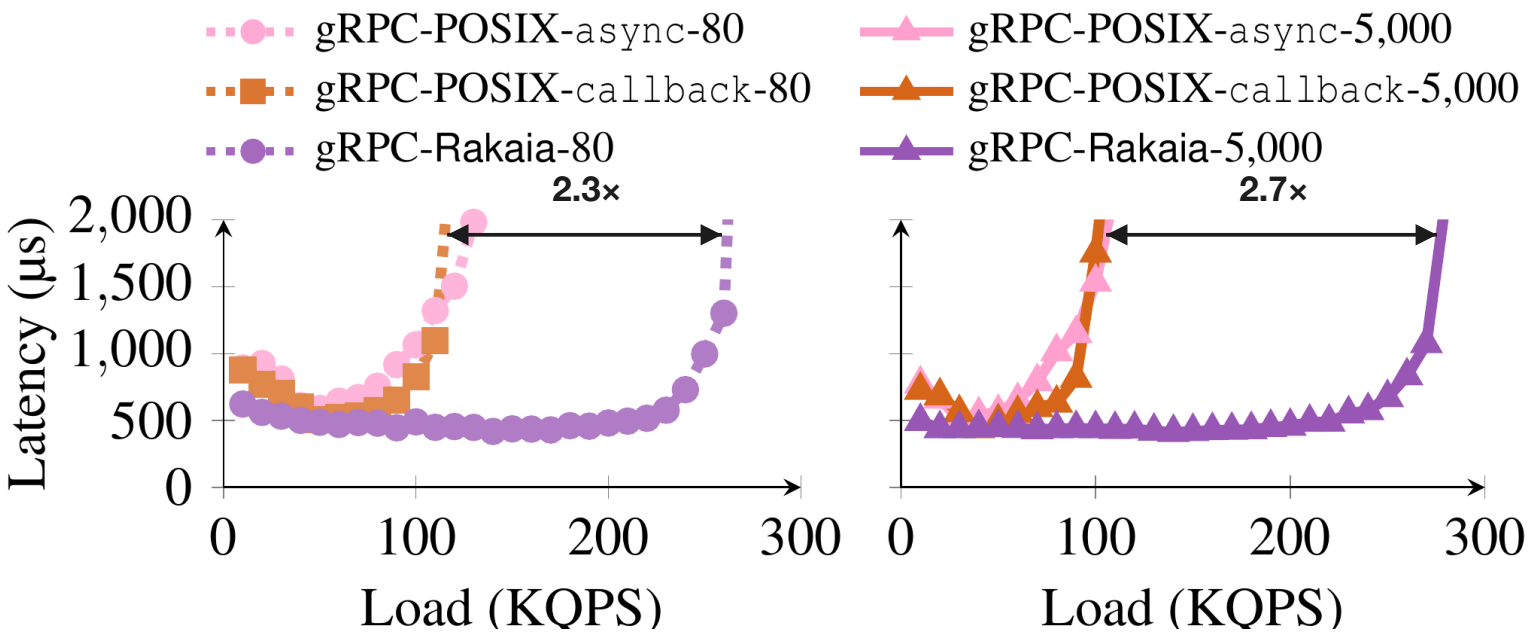
100µs service time, 20 connections, bimodal workload.

gRPC-Go comparison



With 5k connections, Rakaia only needs 33 goroutines, while gRPC-Go requires 17k goroutines.

gRPC-C++ comparison



Rakaia improves gRPC-C++ by 2.3× at 80 connections, and 2.7× at 5k connections

Conclusion

- Problem: Mismatch between stream-oriented TCP and message-oriented RPCs.
- Design: **Rakaia** parses TCP streams into messages and schedules them from the kernel.
- Result: Greatly accelerates both plain-TCP RPCs and gRPC (both C++ and Golang).



Code artifact

github.com/epfl-dcsl

/rakaia-osdi26-artifact