**Evaluating QUIC Implementations on High-rate Links**

**Motivation**

QUIC is implemented in user-space

- Multiple implementations from different developers
- Various languages and architectures

**Advantages and Disadvantages**

+ Flexibility
- More context switches

**Goal:** Identify performance bottlenecks in QUIC stacks

- Dedicated physical hosts for client and server
- Support for core pinning, profiling, DPDK, fiber tapping, ...
- Collect CPU, OS, and NIC metrics with various tools
- Experiment orchestration via POS [1]

⇒ Flexibility, Portability, Reproducibility

**Measurement Setup**

- **Workflow**
  - setup env
  - pre scripts
  - run client/server
  - post scripts
- **Collect Data:**
  - ethtool
  - tcpdump
  - perf
  - ...

**Evaluation**

- High variations between implementations
- Implementations in C & Rust show best performance
- Performance as client/server often varies
- Some implementations have interoperability issues

**Performance Bottlenecks**

- Performance is usually limited by the CPU of the sender
- Default UDP socket buffer size is too small and causes packet drops
- Offloading features that could significantly reduce CPU load are rarely used
- Acknowledgement frequency is often too high for high-rate links
- Crypto operations are CPU intensive but not a bottleneck

**Take Away Messages**

- QUIC implementations show different behavior (acknowledgement frequency, congestion control, …)
- Many QUIC implementations do not use available optimizations
- Default OS settings often not optimized for QUIC

**Outlook**

- More implementations to be evaluated
- Support for Multipath QUIC

---
