X-IO: A High-performance Unified I/O Interface using Lock-free Shared Memory Processing

Shixiong Qi[*] Han-Sing Tsai[†] Yu-Sheng Liu[†],
K. K. Ramakrishnan[*] Jyh-Cheng Chen[†]

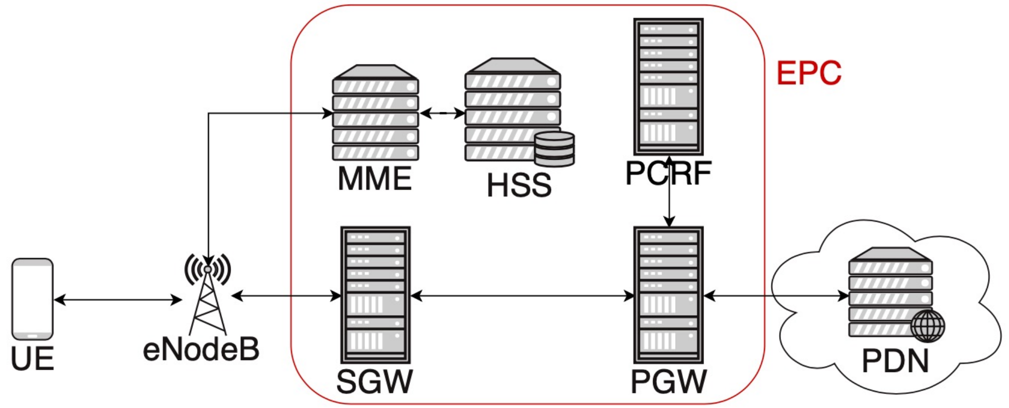
[*]*University of California, Riverside* [†]*National Yang Ming Chiao Tung University*

Visit us at:
https://kknetsyslab.cs.ucr.edu/
https://cslab.cs.nycu.edu.tw/

Cloud-native Applications

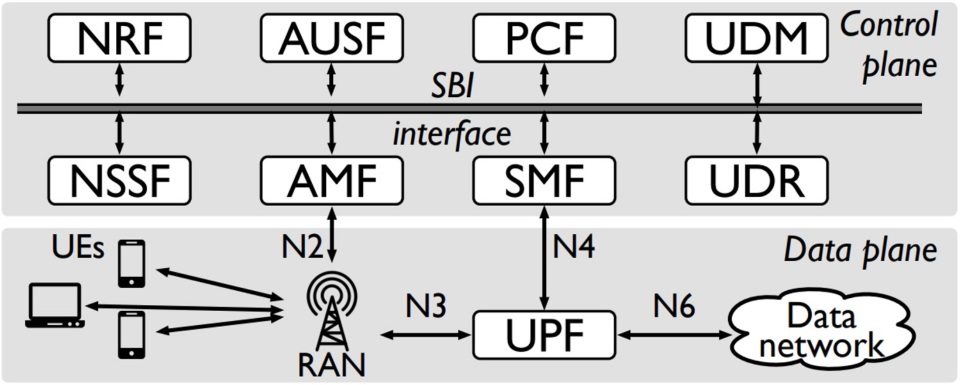
Moving from monolithic services to microservices: ***e.g., the evolution of cellular core***

Monolith LTE EPC



- All-in-one
- ***Hard*** to scale out
- ***Poor*** modularity

Microservices 5GC



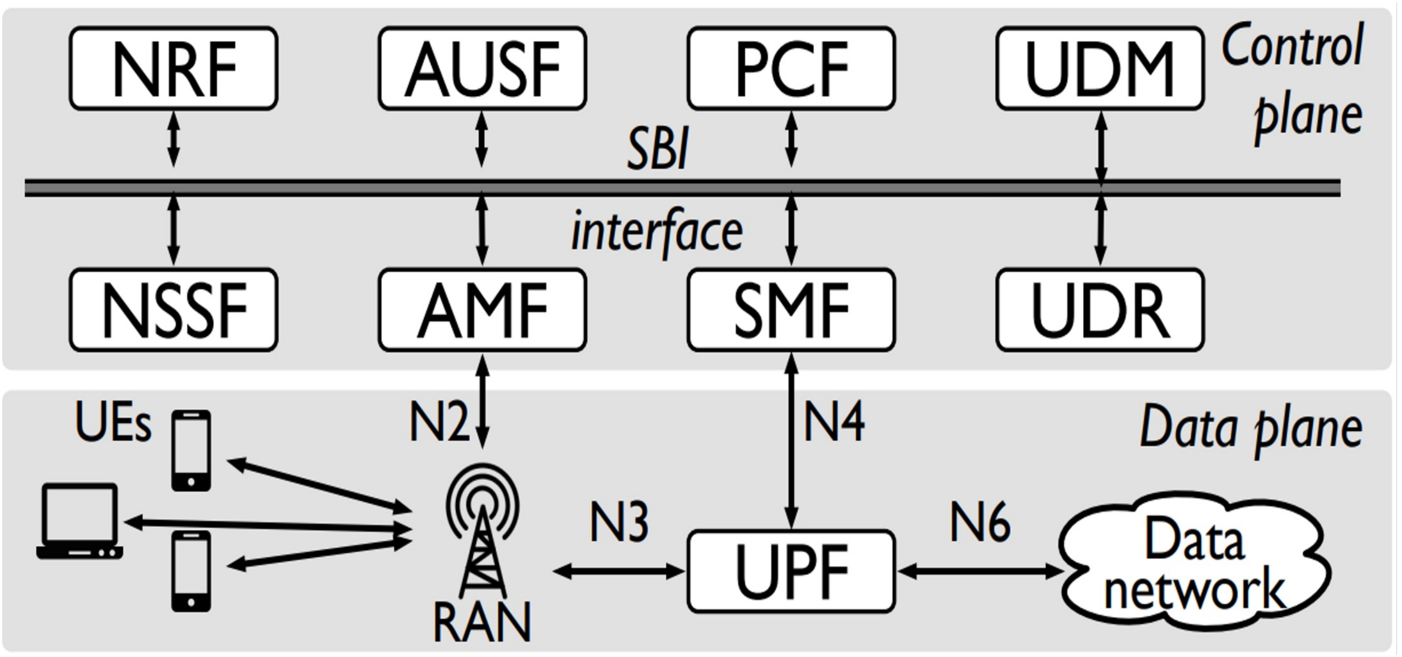
- ***Independently*** deployable
- ***Loosely*** coupled
- ***Easy*** to scale out
- ***Good*** modularity

Cloud-native microservice networking

Coupling microservice together

- Microservices are loosely coupled
  - Microservices ***networked*** together
  - Account for task dependencies

- Networked Microservices
  - **Synchronous** I/O
    - gRPC, 3GPP Service Bus Interface
  - **Asynchronous** I/O
    - Organize communication among a set of interdependent microservices as a Directed Acyclic Graph (DAG)

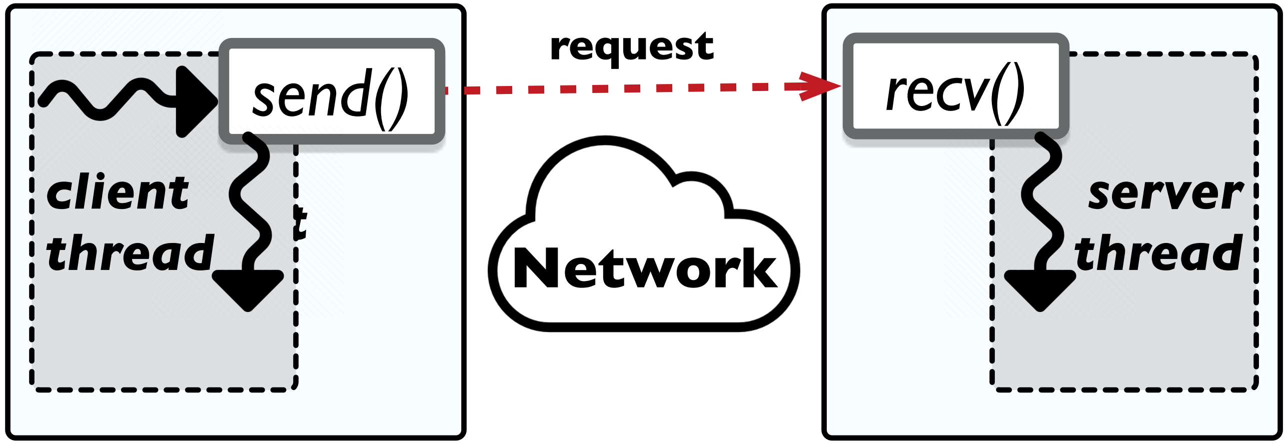


Cloud-native microservice networking

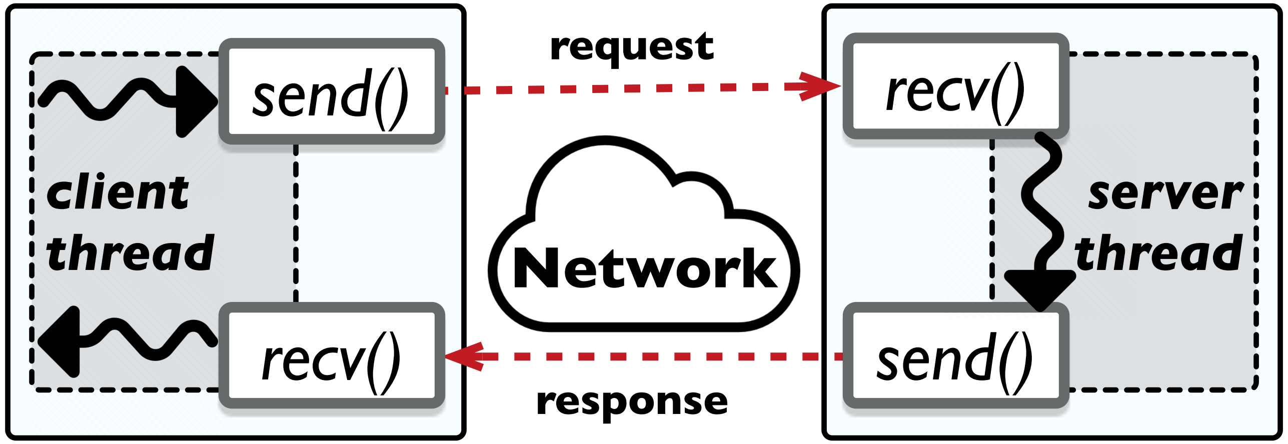
Problems

- Asynchronous I/O and Synchronous I/O:
    - distinct; ***mismatched*** *for a common implementation*

- **Asynchronous I/O**
    - Caller **<u>does not</u>** wait for a response from callee
        - The caller is **not blocked** waiting for the request
    - ***Unidirectional*** data exchange: a send/recv pair



- **Synchronous I/O**
    - Caller **<u>expects</u>** a response from callee
        - Caller **waits** (blocked) until response is returned
    - ***Bidirectional*** data exchange: two send/recv pairs

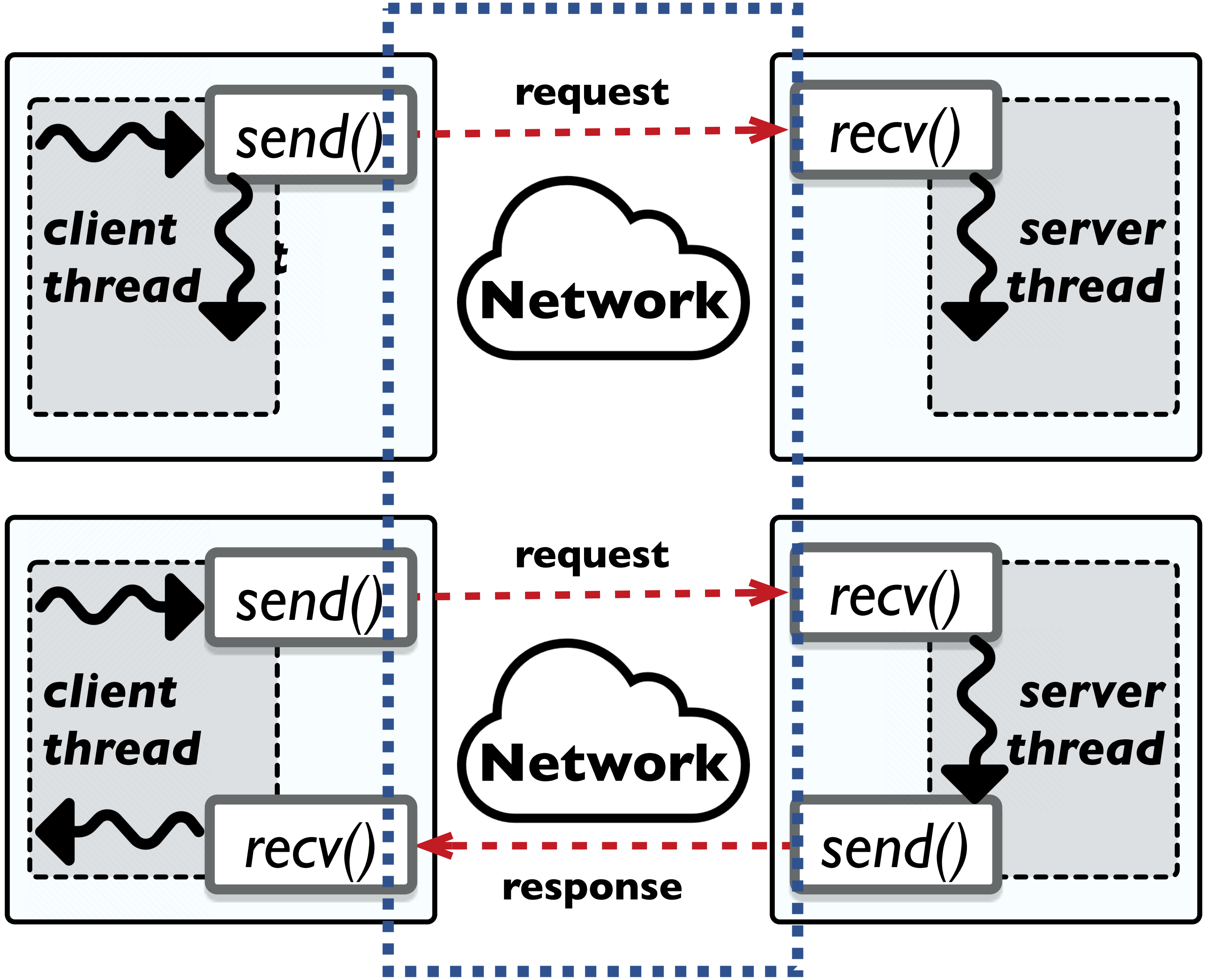


Cloud-native microservice networking

Problems

- Utilize Kernel-based networking stack
- High communication overhead for server/client
  - **Data copies**
  - **Protocol processing**
  - **Context switches**
  - **Interrupts**
  - **Serialization/Deserialization**

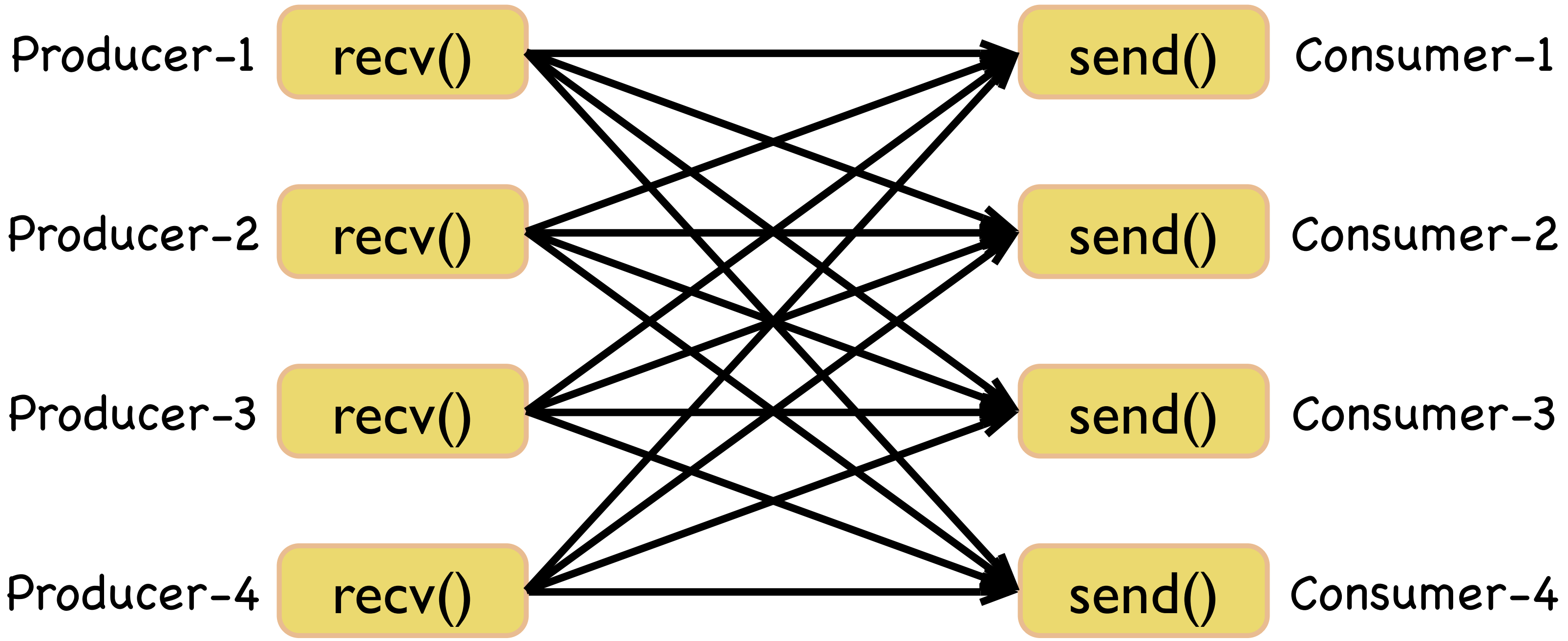
- ***CPU cycles and memory bandwidth are consumed!***



Cloud-native microservice networking

Problems

- **<u>Multiple</u>** producers, **<u>Multiple</u>** consumers communication pattern
  - **Contention** and **lock**: ***performance loss***



Cloud-native microservice networking

Summary

- **Mismatched** communication models between Asynchronous I/O and Synchronous I/O
- **Heavyweight** kernel-based networking stack
- **Multiple**-producer, **multiple**-consumer communication pattern
- Different user sessions or flows need to be handled **in parallel**
- Programming language **incompatibility**

→

- A **Unified** I/O interface
- **Zero-copy** packet delivery
- **Lock-free** producer/consumer rings
- **Concurrent connection** support
- **Cross-language** support

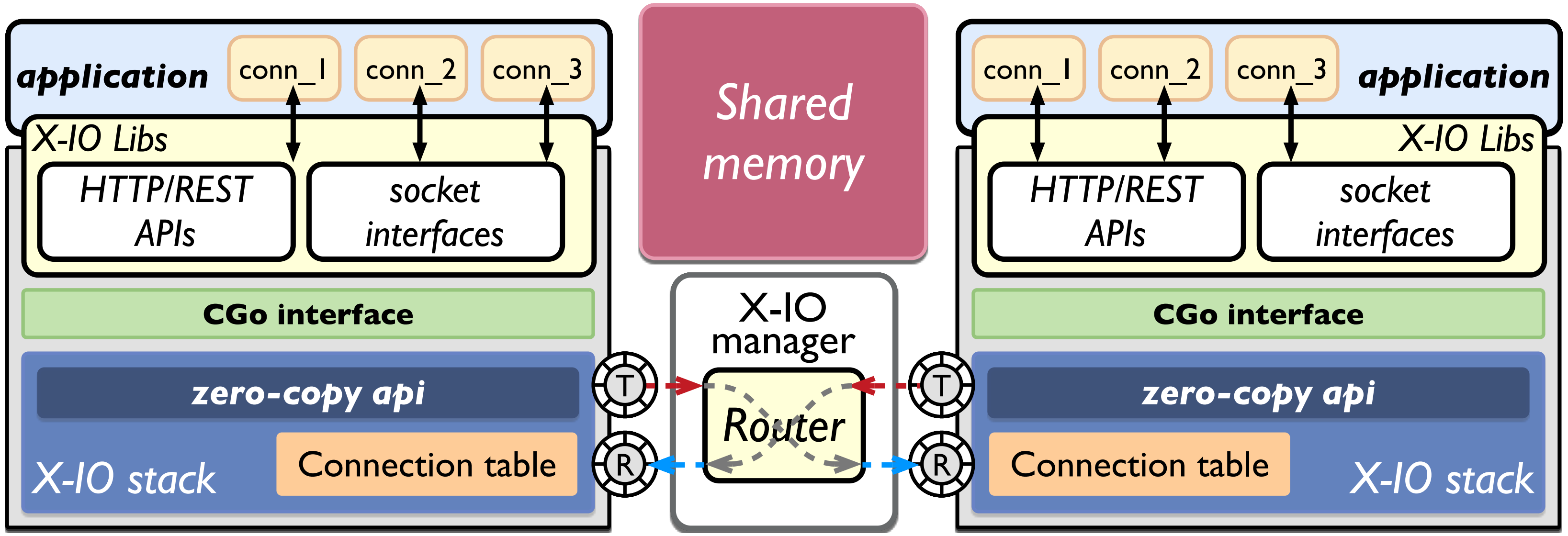
X-IO: A High-performance Unified I/O Interface

X-IO:A High-performance Unified I/O Interface

Networked Systems Group
UC RIVERSIDE

Overview: X-IO in a box

- ***Shared memory processing with lock-free producer/consumer rings***
  - Built in X-IO stack
  - We consider **DPDK** for the implementation
    - Other shared memory processing design, e.g., SPRIGHT [1], is also applicable
- ***Raw I/O primitives: zero-copy interface***
  - Exposed via X-IO stack
- ***POSIX-like I/O primitives: socket interface & HTTP/REST APIs***
  - Exposed via **X-IO libs**
- ***Concurrent connection management***
  - via **X-IO stack**
  - using **"Connection Table"**
- ***Cross-language support***
  - **CGo interface** in Golang



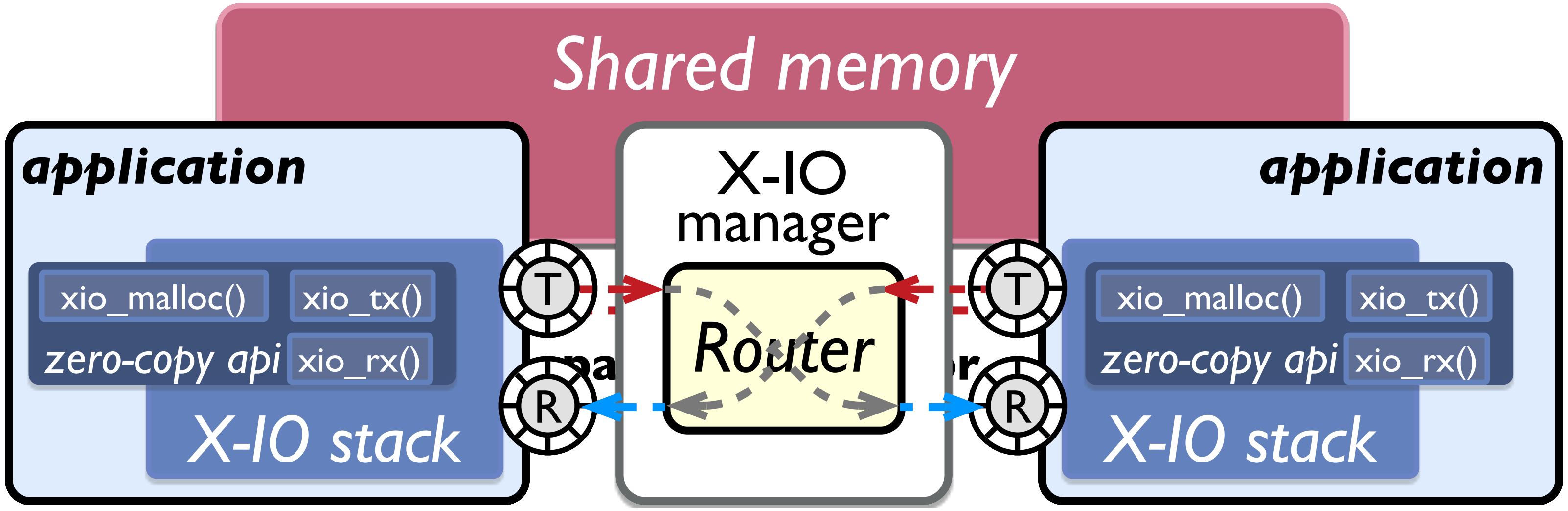
[1] Qi, Shixiong, et al. "SPRIGHT: extracting the server from serverless computing! high-performance eBPF-based event-driven, shared-memory processing." *Proceedings of the ACM SIGCOMM 2022 Conference*. 2022.

X-IO:A High-performance Unified I/O Interface

Shared memory processing with lock-free producer/consumer rings

- ***Building blocks of shared memory processing***
  - Shared memory pool
  - Packet descriptor delivery mechanism
    - Deliver packet descriptor instead of packet: **NO** memory-memory copy
- Zero-copy I/O primitives from the **X-IO stack**
  - ***xio_malloc***(), ***xio_tx***(), and ***xio_rx***()
  - construct a **truly** zero-copy communication channel between microservices

- ***Lock-free packet descriptor delivery***
  - each X-IO stack has a pair of receive (RX) and transmit (TX) RINGs
  - X-IO stack to share its RING pair with the X-IO manager
    - Single-producer, single-consumer ring access
    - thereby avoiding having to acquire a lock
    - We use the X-IO manager to forward descriptors between different X-IO stacks



X-IO: A High-performance Unified I/O Interface

Networked Systems Group
UC RIVERSIDE

POSIX-like I/O primitives in X-IO: socket interface

- Supporting **seamless** porting of applications that depend on the POSIX socket API
- Exposed via an abstraction layer, namely X-IO lib
  - equivalent Golang-style socket interfaces.
  - Read(), Write(), Listen(), Accept(), Dial()

```
import "net"

/* Golang-style socket server */
listener, _ := net.Listen(server_address)
conn, _ := listener.Accept()

receive_buffer := make([]byte, RECV_MSG_SIZE)
n, err := conn.Read(receive_buffer)

conn.Close()

/* Golang-style socket client */
conn, err := net.Dial(server_address)

send_buffer := make([]byte, SEND_MSG_SIZE)
n, err := conn.Write(send_buffer)

conn.Close()
```

→

```
import "xio"

/* X-IO-based socket server */
listener, _ := xio.Listen(server_address)
conn, _ := listener.Accept()

receive_buffer := make([]byte, RECV_MSG_SIZE)
n, err := conn.Read(receive_buffer)

conn.Close()

/* X-IO-based socket client */
conn, err := xio.Dial(server_address)

send_buffer := make([]byte, SEND_MSG_SIZE)
n, err := conn.Write(send_buffer)

conn.Close()
```

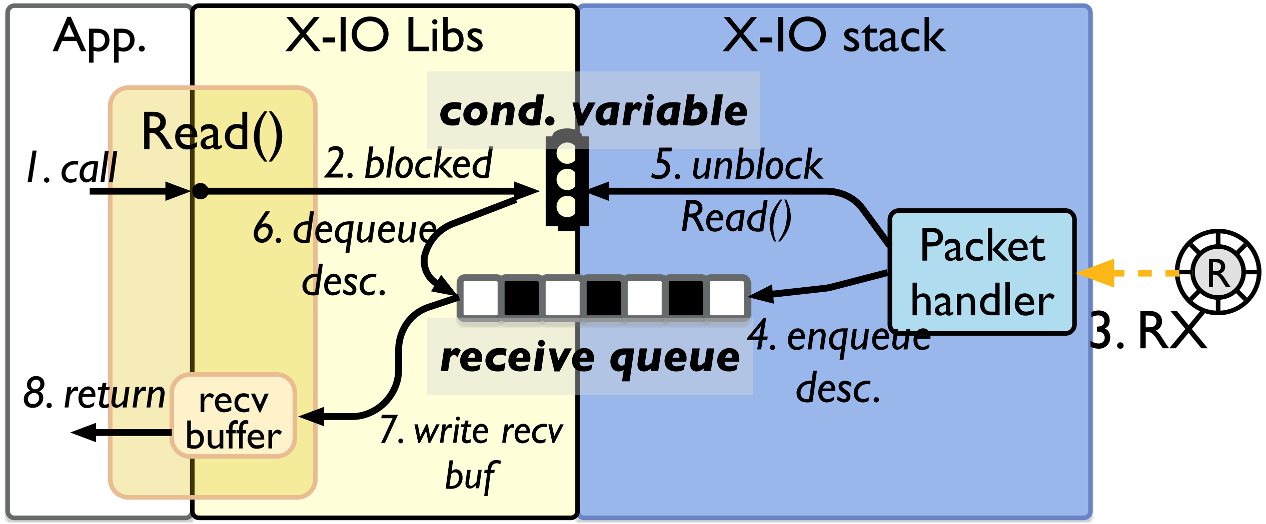
X-IO:A High-performance Unified I/O Interface

POSIX-like I/O primitives in X-IO: Read() interface

- **Read()**: basic read socket interface in X-IO
  - supports both "***blocking***" and "***non-blocking***" modes

- "***blocking***" mode:
  - The caller of Read() is **blocked** until it **receives** the request from the X-IO stack
  - Blocking primitive:
    - **wait** until it is **signaled to wake up**
  - Batch wake-up mechanism
    - a **receive queue** to buffer the requests
    - ***Reduce wake-up overhead***

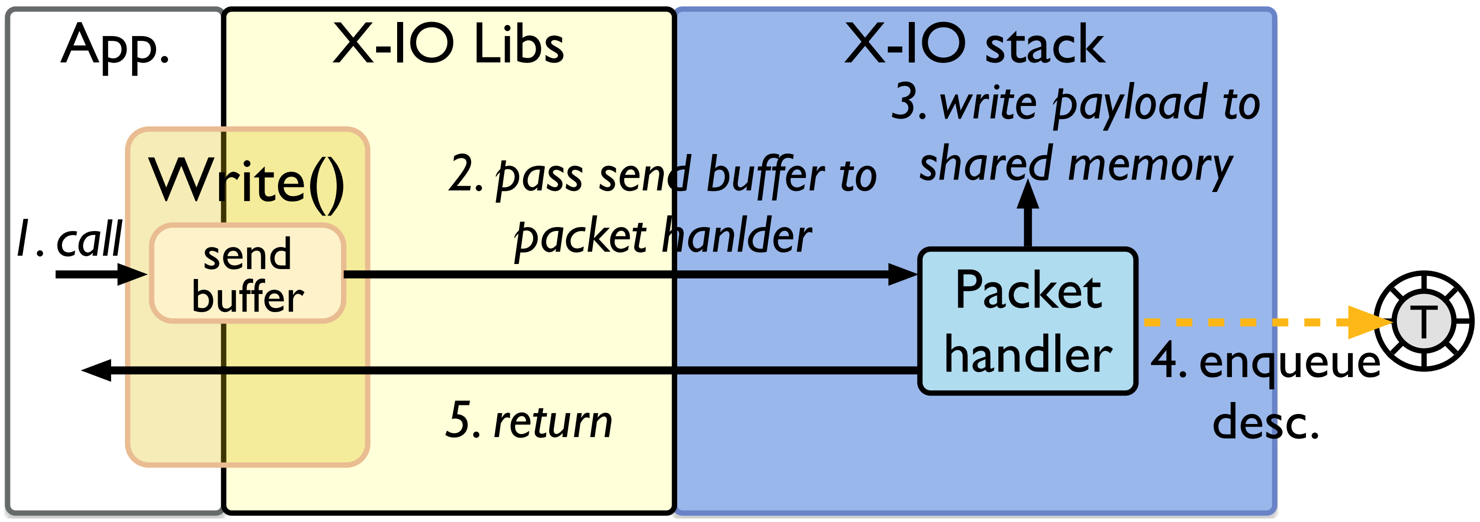
- "***non-blocking***" mode:
  - The caller of Read() is **not blocked** waiting for the request
  - Requires busy-polling



X-IO: A High-performance Unified I/O Interface

POSIX-like I/O primitives in X-IO: Write() interface

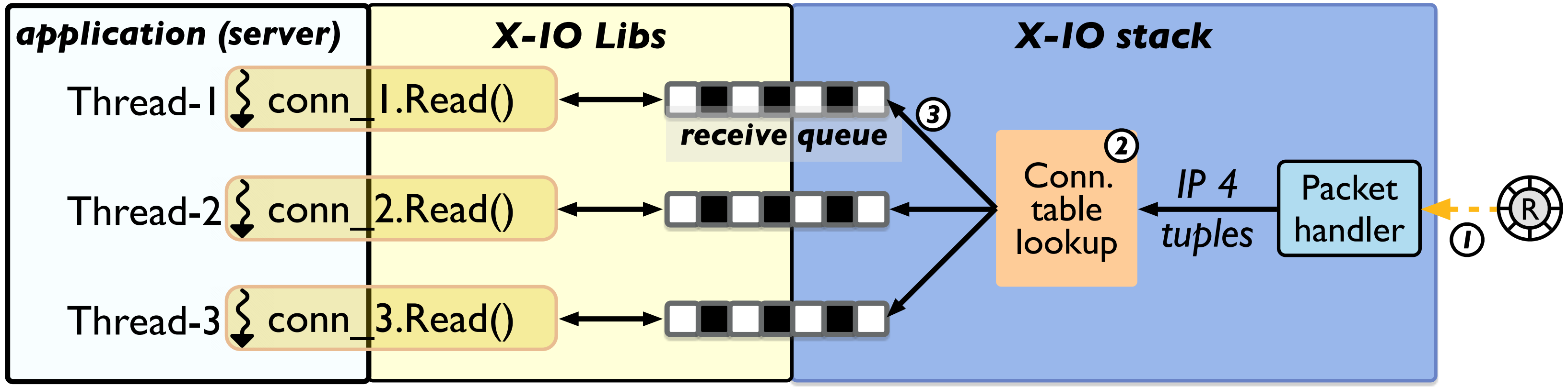
- **Write()**: basic ***write*** socket interface in X-IO
  - We only support ***blocking*** Write() in X-IO
  - Ensure all of the request payload is written into the shared memory buffer before the Write() returns



X-IO: A High-performance Unified I/O Interface

POSIX-like I/O primitives in X-IO: connection management

- Listen(), Accept(), Dial(), Close(): **Connection Establishment & Teardown**
  - Both Read() and Write() interfaces in X-IO require an *apriori established* connection for data transmission
- ***Concurrent connection support***
  - Core components: **connection table** in X-IO stack
    - Distribute requests to different connections via ***IP 4-tuples lookup***



X-IO: A High-performance Unified I/O Interface

Networked Systems Group
UC RIVERSIDE

POSIX-like I/O primitives in X-IO: socket interface

- Pros: seamless porting of existing applications
- Cons:
  - Copies introduced by "**send_buffer**" and "**receive_buffer**"
  - Price we pay to maintain alignment with POSIX-like APIs

```
import "net"

/* Golang-style socket server */
listener, _ := net.Listen(server_address)
conn, _ := listener.Accept()

receive_buffer := make([]byte, RECV_MSG_SIZE)
n, err := conn.Read(receive_buffer)

conn.Close()

/* Golang-style socket client */
conn, err := net.Dial(server_address)

send_buffer := make([]byte, SEND_MSG_SIZE)
n, err := conn.Write(send_buffer)

conn.Close()
```

→

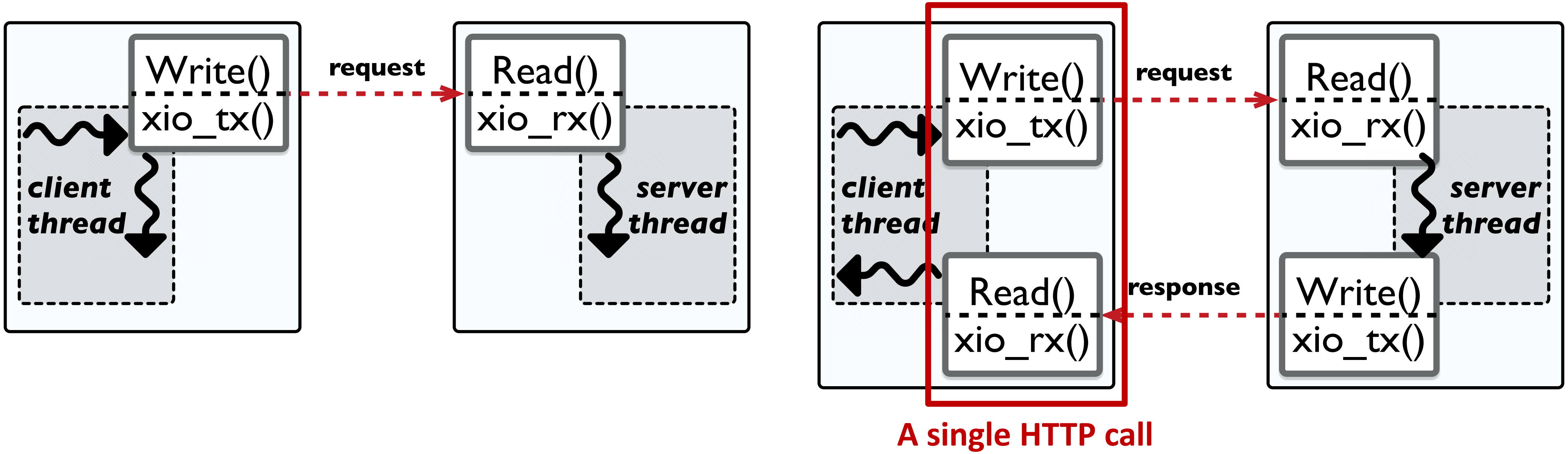
```
import "xio"

/* X-IO-based socket server */
listener, _ := xio.Listen(server_address)
conn, _ := listener.Accept()

receive_buffer := make([]byte, RECV_MSG_SIZE)
n, err := conn.Read(receive_buffer)

conn.Close()

/* X-IO-based socket client */
conn, err := xio.Dial(server_address)

send_buffer := make([]byte, SEND_MSG_SIZE)
n, err := conn.Write(send_buffer)

conn.Close()
```

X-IO: A High-performance Unified I/O Interface

Networked Systems Group
UC RIVERSIDE

Asynchronous & Synchronous data exchange with X-IO

Asynchronous and synchronous I/O between microservices can be built using either **X-IO's socket interface** or **X-IO's zero-copy interface**



- ***Case study – using X-IO to support 3GPP SBI***
  - 3GPP SBI is built on top of HTTP/REST APIs
  - X-IO offers **equivalent** HTTP/REST APIs built on **X-IO's socket interface** to support seamless porting
    - Remove redundant data copies and protocol parsing

Evaluation

Experiment Setup

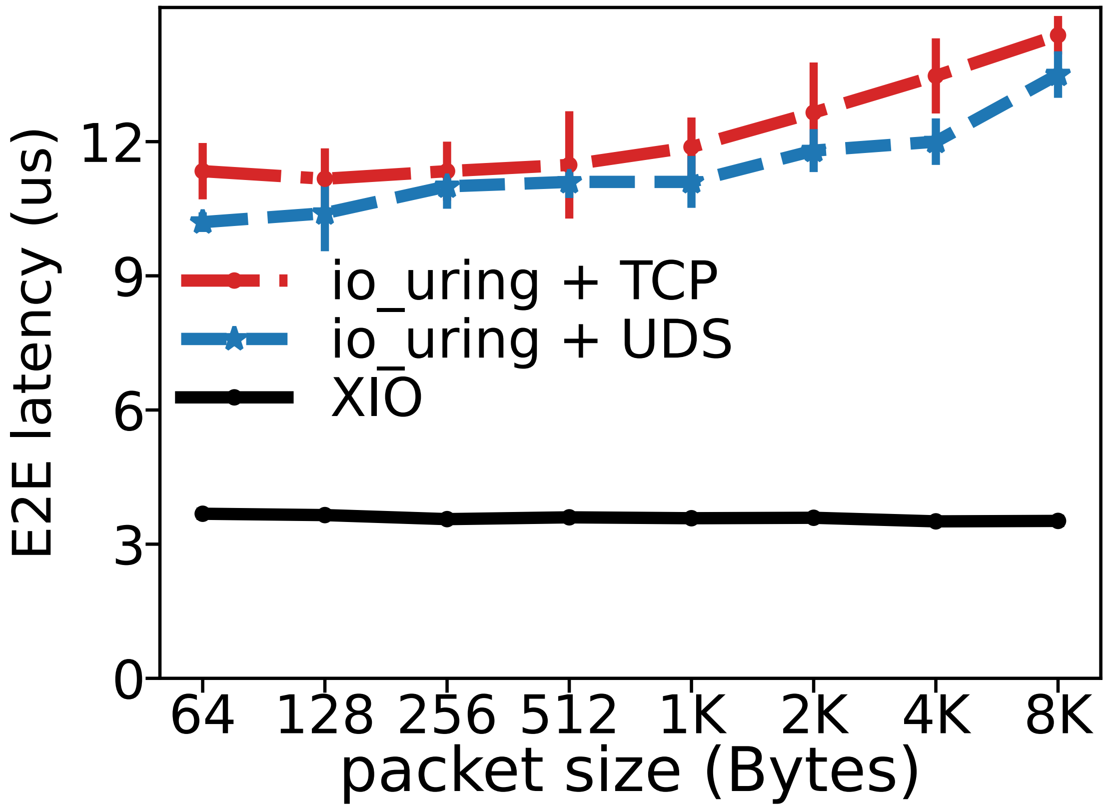
1. X-IO's zero-copy interface vs. Linux io_uring (TCP socket, UNIX-domain socket)
2. POSIX-like socket interface performance:
   - X-IO's Read()/Write() vs. Linux Read()/Write() (TCP socket, UNIX-domain socket)
3. HTTP/REST API performance
   - X-IO's "xio/http" vs. Golang's "net/http"

Evaluation

X-IO's zero-copy interface vs. Linux io_uring

- A client application and an echo server application
  - Placed on the ***same*** node
  - Both developed in ***C***

- ***Round-trip latency***
  - X-IO's zero copy interface achieves **2.8×~4.1× lower round-trip latency** than io_uring
    - *Improvement over both TCP socket or UDS*
  - X-IO's zero copy interface has ***constant*** latency across various message sizes
    - demonstrating the benefit of zero-copy shared memory communication with X-IO
    - **4** packet copies are incurred for every packet round-trip when using io_uring



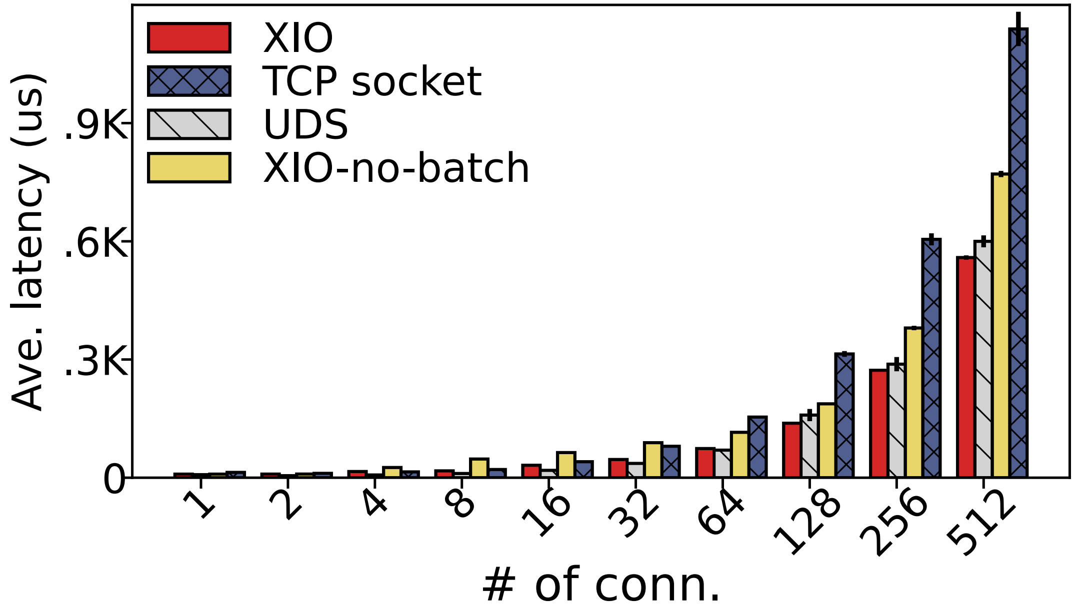
Evaluation

POSIX-like socket interface performance

- A pair of client and server application
  - Placed on the ***same*** node
  - Both developed in ***Golang***
  - Vary the number of concurrent connections (persistent)
  - Each connection is allowed to have **1** in-flight request (64B)

- ***Round-trip latency***
  - ***X-IO*** consistently has lower latency (**~1.6x**) than TCP socket

- The latency of ***X-IO-no-batch*** is always higher than default ***X-IO***
  - Performing wake-up (unblocking) of **multiple connections** in a batch that can amortize the overheads of interrupts and context switches

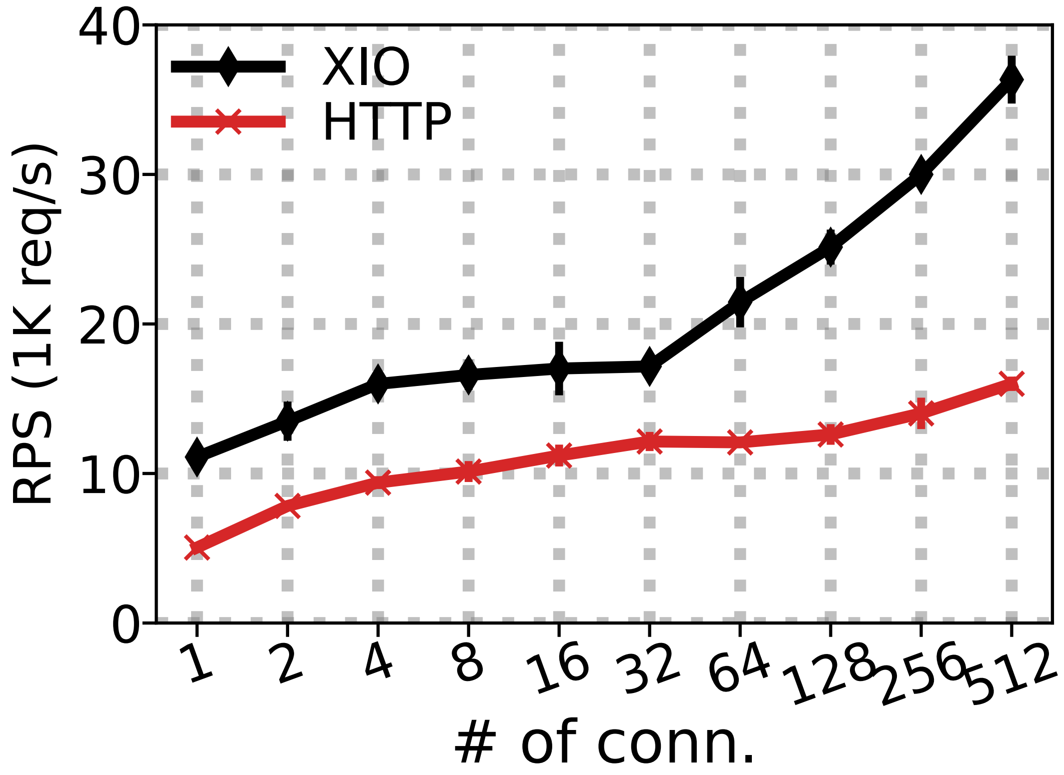


Evaluation

HTTP/REST API performance

- An HTTP echo server/client pair
  - Placed on the ***same*** node
  - Both developed in ***Golang***
  - Vary the number of concurrent HTTP connections (persistent)
  - Each connection is allowed to have **1** in-flight request (64B)

- ***HTTP Requests per second***
  - X-IO achieves **1.4×~2.3× improvement** in RPS and latency
    - X-IO avoids extra copies and protocol parsing between socket interface and HTTP interface
    - More scalable than Golang's HTTP



Conclusion

X-IO is a high-performance, unified I/O interface designed for cloud-native microservices

- **X-IO stack**
  - A shared memory based network stack with lock-free producer/consumer rings
- **Raw I/O primitives exposed by X-IO stack**
  - Zero-copy data transmission
  - Superior performance: **2.8×~4.1× lower** latency over ***both*** TCP socket or UDS
- **POSIX-like primitives abstracted by X-IO lib**
  - **Seamless** porting of applications that use POSIX-like socket interface
  - Multiple user session support
  - Outperform Linux TCP/IP socket interface: **1.6x improvement**
  - Competitive performance compared to Linux UNIX-domain socket interface
- **HTTP/REST APIs abstracted by X-IO lib**
  - **Seamless** porting of applications that use HTTP/REST APIs
  - **1.4×~2.3× improvement** in RPS and latency compared to Golang's HTTP/REST APIs

☞ **Find X-IO at:** https://github.com/nycu-ucr/xio.git

X-IO is Available

☞ **Find X-IO at:** https://github.com/nycu-ucr/xio.git