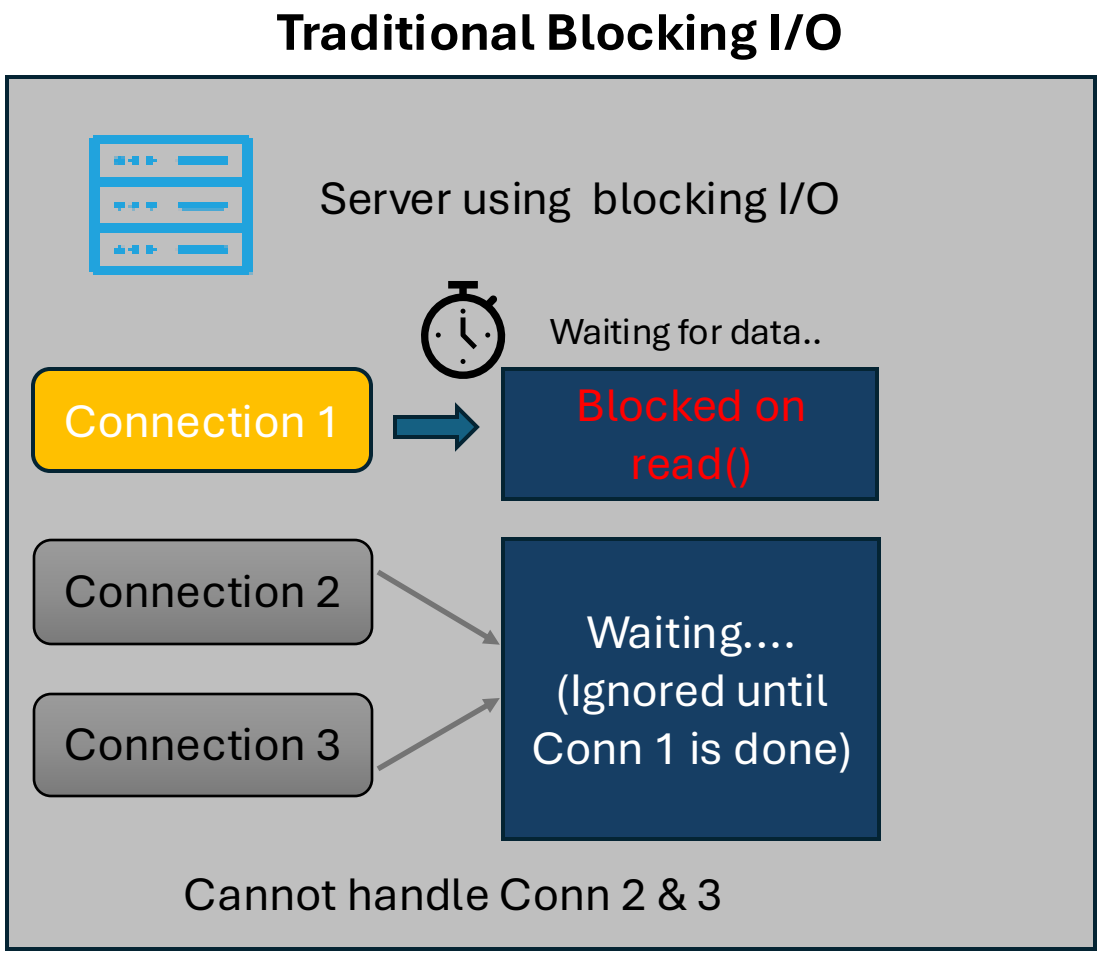
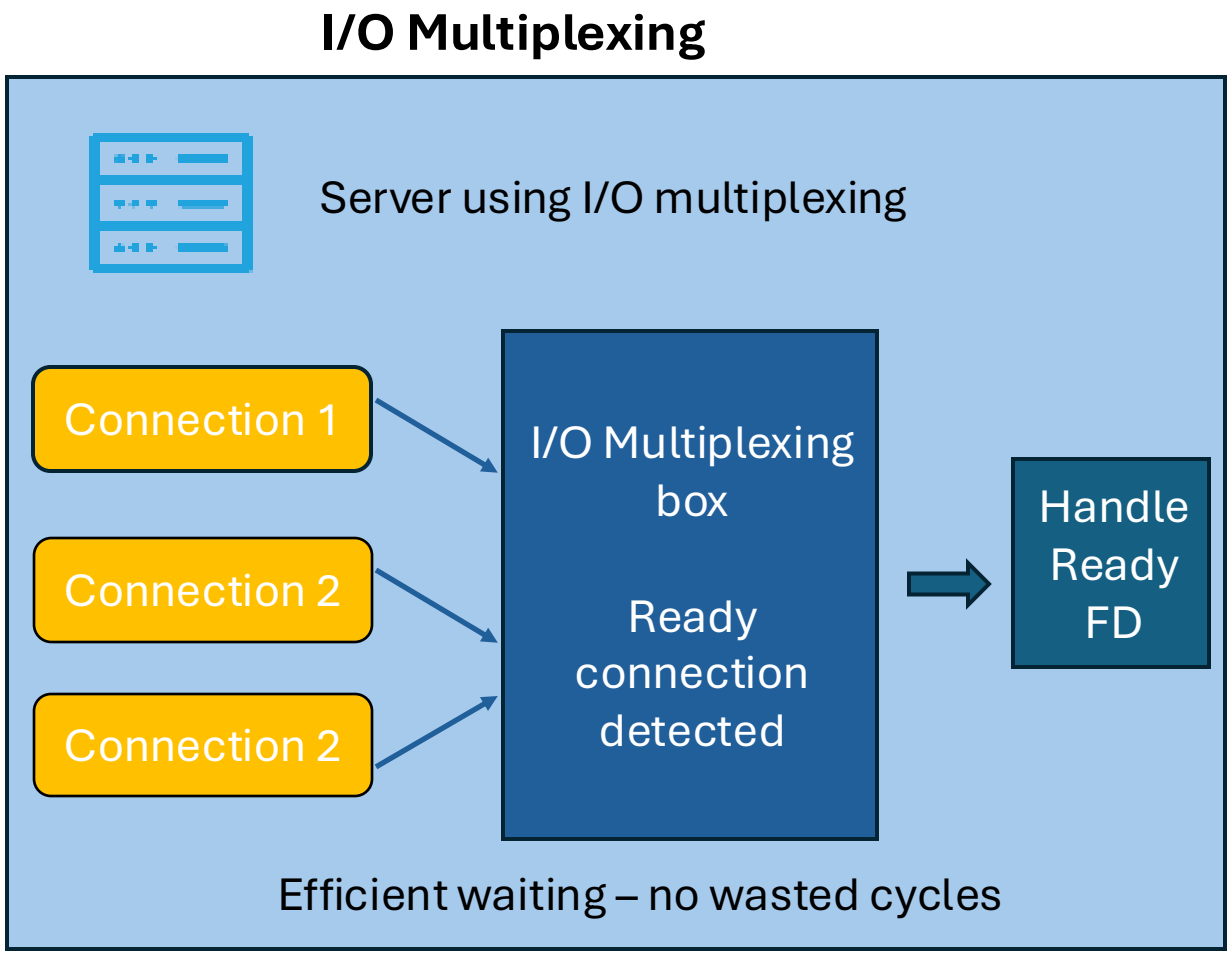
Analyzing the Fairness of the Scheduling of Epoll

Presenter: Akanksha Dadhich
Guide: Prof. Ashutosh Gupta and
Prof. Mythili Vutukuru
R & D

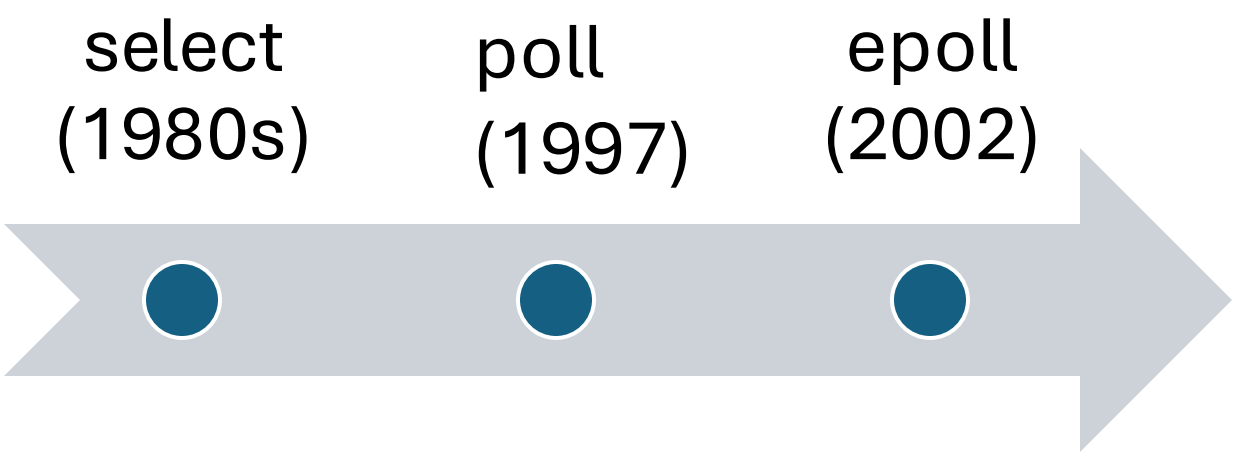
I/O Multiplexing vs Blocking I/O in Servers



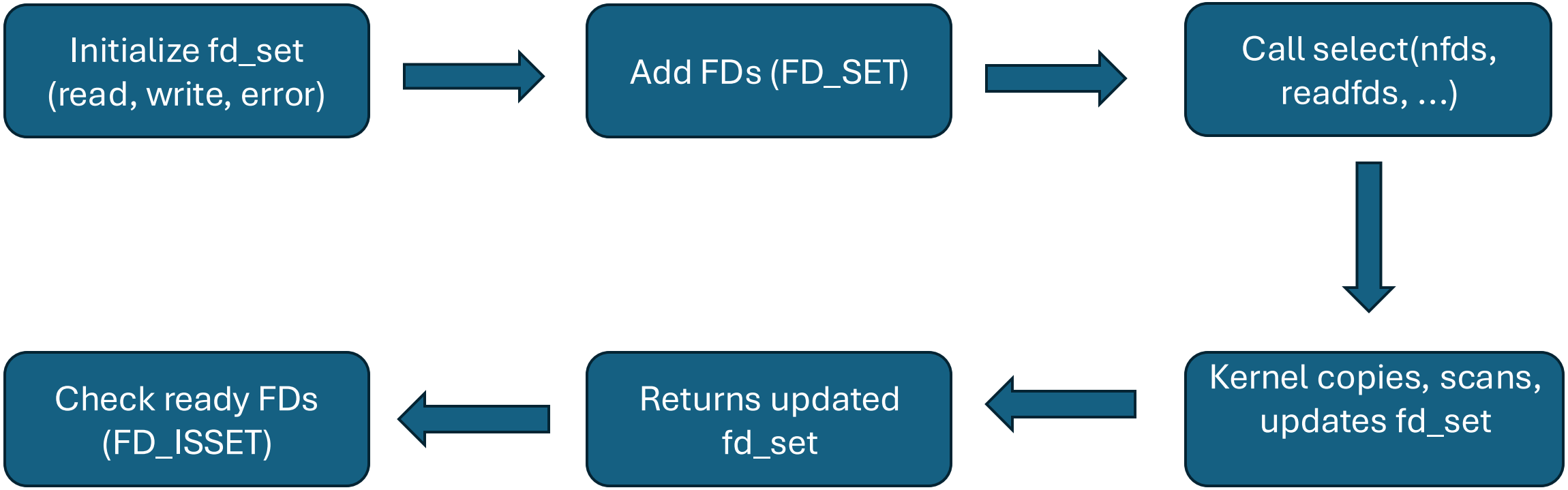


Introduction to I/O Multiplexing

- What is I/O Multiplexing?
  - •Monitors multiple FDs for events
  - •Used in servers for multiple connections
- System Calls: select, poll, epoll
- In server applications (e.g., handling thousands of clients), blocking on one connection would prevent handling others.
- I/O multiplexing solves this by allowing the program to:
  - Wait for multiple file descriptors (FDs) to become ready.
  - Avoid wasting CPU cycles polling each one.



How select() Works?



Copies fd_set, scans all FDs

Defects of select()

Limited to 1024 FDs (FD_SETSIZE default). Not scalable

Linear search O(n) for event detection.

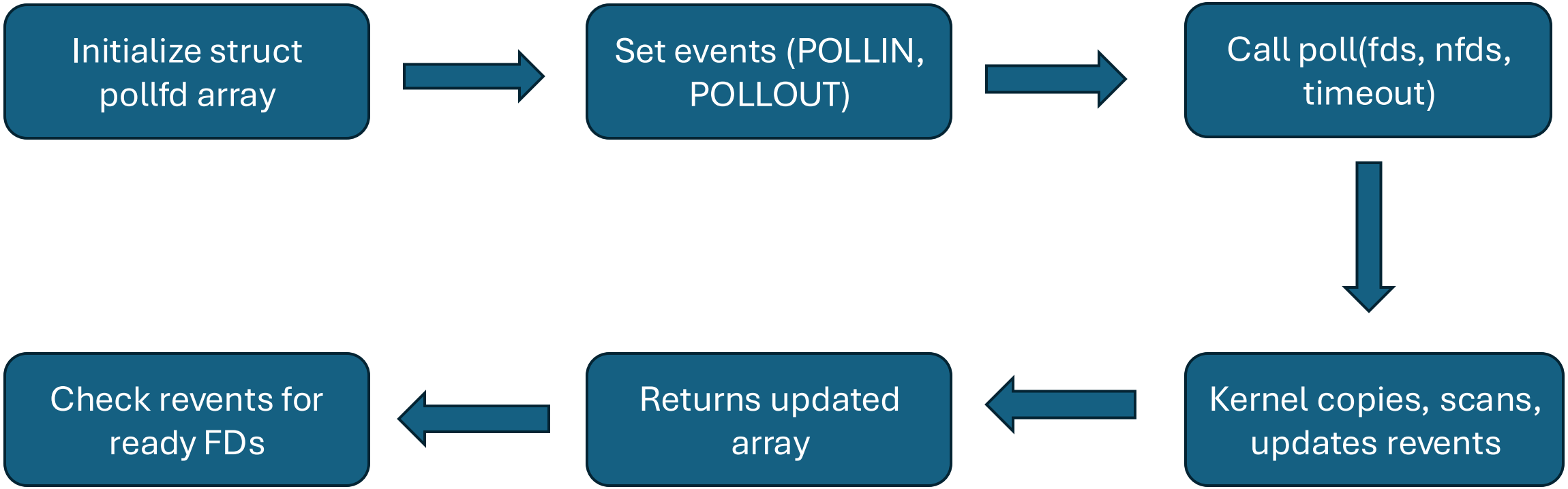
Copy Overhead: Entire fd_set copied to/from kernel

Inefficient for idle connections

Data Structure: Uses a fixed-size bit array (fd_set)

How poll() Works?

Monitors FDs using struct pollfd array: events, revents, FDs



poll()

No FD limit, scalable in terms of FD count.

Still O(n) due to linear scan of FDs.

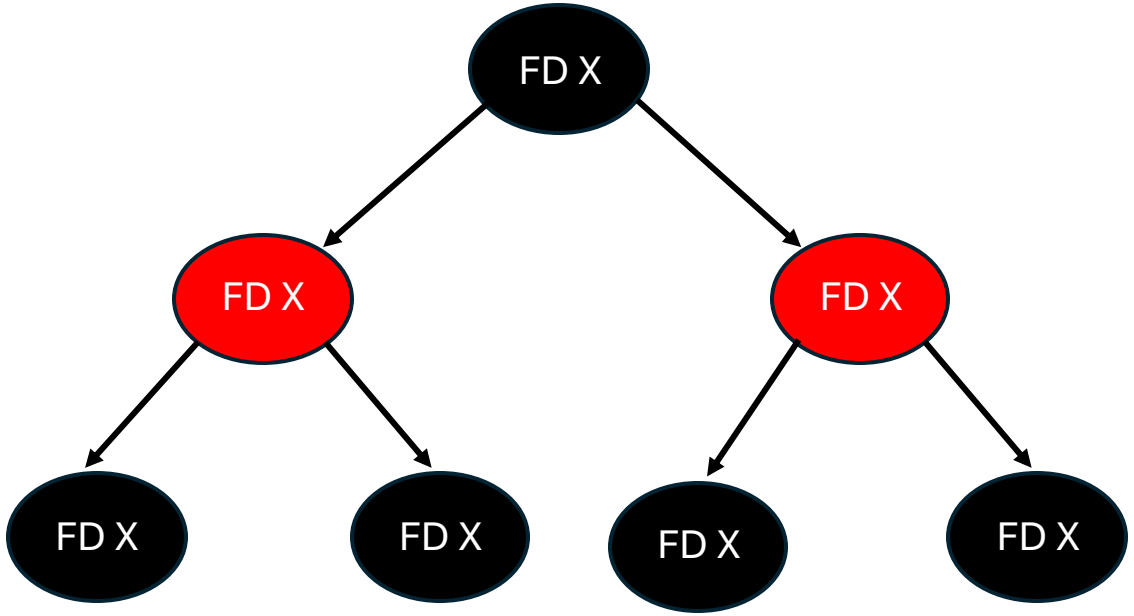
Copy Overhead: entire struct pollfd array copied to/from kernel

Inefficient for idle connections

Data Structure: Uses a dynamic array of struct pollfd (contains FD, events, revents)

What is epoll?

- epoll stands for event poll. Unlike poll, epoll itself is not a system call.
- It’s a kernel data structure Handles thousands of FDs easily
- Efficient, scalable I/O event notification
- Edge-triggered and Level-triggered modes
- Used in socket programming for non-blocking, asynchronous I/O.
- Replaces select/poll with O(1) performance using red-black tree and ready list.



epoll()

Scalable

No FD limit

O(1) event detection
(ready list)

Low copy overhead

Efficient for idle
connections

epoll System Calls

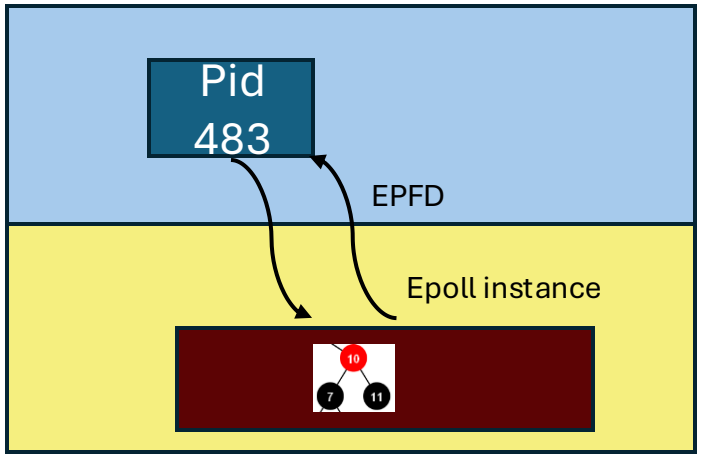
- epoll_create()
- epoll_ctl()
- epoll_wait()

```
int epfd = epoll_create1(0); // Create epoll instance
struct epoll_event event;
event.events = EPOLLIN;
event.data.fd = sockfd;
epoll_ctl(epfd, EPOLL_CTL_ADD, sockfd, &event); // Add socket
struct epoll_event events[10];
int nfds = epoll_wait(epfd, events, 10, -1); // Wait for events
```

epoll_create: Creating the epoll Instance

- **epoll_create(int size):** Creates an epoll instance, returning a file descriptor (deprecated since Linux 2.6.8).
  - size: Hint for kernel on expected FD count (ignored, dynamically resized).
- **epoll_create1(int flags)**: Modern version; flags = 0 (same as epoll_create) or EPOLL_CLOEXEC (closes FD on exec).

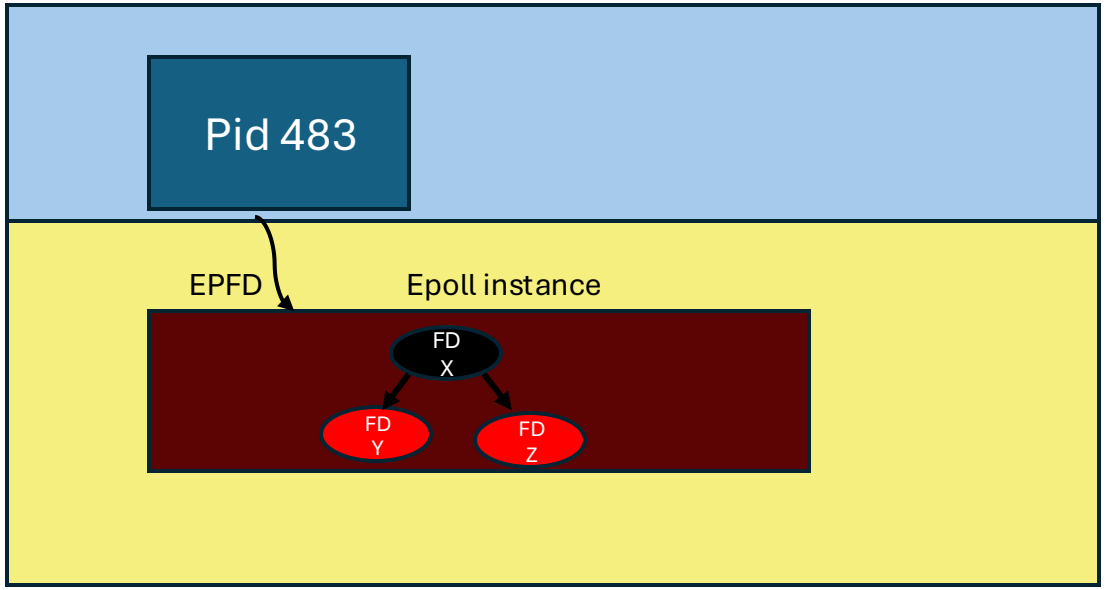
```
int epfd = epoll_create1(0);
```



epoll_ctl: Adding Client Sockets to epoll

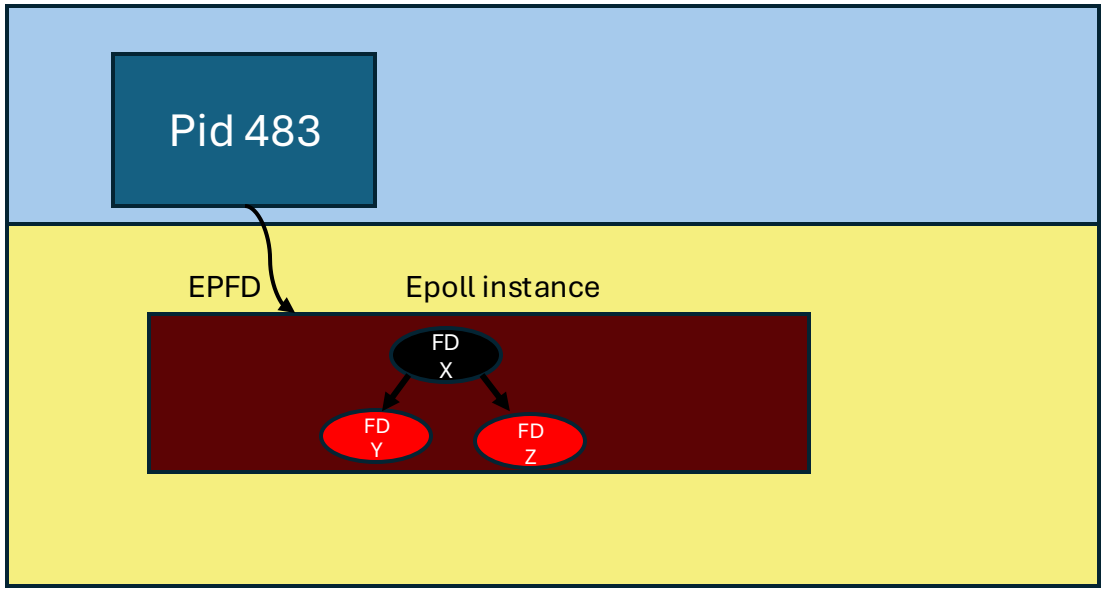
- **epoll_ctl**(int epfd, int op, int fd, struct epoll_event *event): Manages FDs in epoll set.
  - epfd: File descriptor of epoll instance.
  - op: EPOLL_CTL_ADD (register FD), EPOLL_CTL_DEL (deregister), EPOLL_CTL_MOD (modify events).
  - event: epoll_event struct with events (bitmask, e.g., EPOLLIN, EPOLLET) and union data.
- **ready list:** Subset of FDs ready for I/O.

```
struct epoll_event event;
event.events = EPOLLIN;
event.data.fd = client_fd;
epoll_ctl(epfd, EPOLL_CTL_ADD, client_fd,
&event);
```



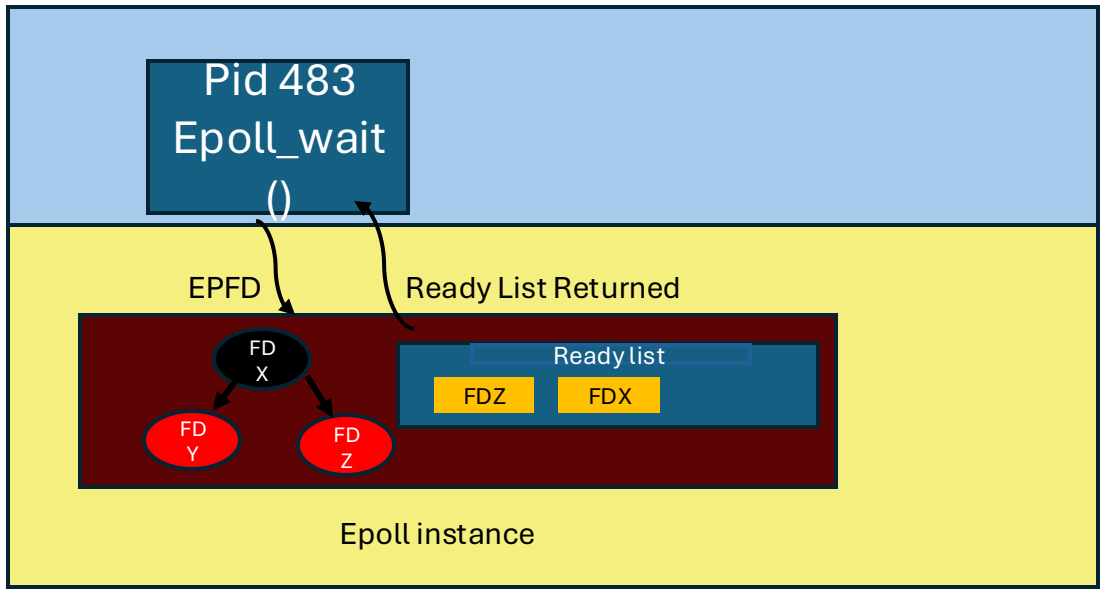
epoll_ctl: Adding Client Sockets to epoll

- Each node in the red-black tree is a struct epitem, representing one monitored FD. The structure contains:
  - **File Descriptor (FD)**: Stored as epi->ffd.fd
  - **Event Types**: Events the user is interested in (e.g., EPOLLIN, EPOLLOUT) Stored in epi->event.events
  - **User Data**: User-defined data (epoll_event.data, typically a pointer). Stored in epi->event.data
  - **Ready List Link**: A list_head structure (epi->rdlink) to link this epitem into the ready list when events occur.
  - **Red-Black Tree Metadata**: struct rb_node for tree links (left, right, parent)



epoll_wait: Waiting for Events

- **epoll_wait**(int epfd, struct epoll_event *evlist, int maxevents, int timeout): Blocks until FDs are ready.
  - epfd: Identifies epoll instance.
  - evlist: Array of epoll_event structs, filled with ready FDs.
  - maxevents: Size of evlist array.
  - timeout: 0 (non-blocking), -1 (block forever), or positive (ms timeout).
- Returns: -1 (error), 0 (timeout), or positive (number of ready FDs).



But is epoll fair?

Financial Trading Systems

Web Servers & Proxies

Real-Time Chat

What is Fairness in epoll?

- Fairness refers to the equal treatment of two connections (Persistent and Non-Persistent connections) during scheduling by the epoll mechanism.
- Fair means no consistent preference is shown to either connection type.
- The order of scheduling should be random or balanced over time.
- Both connections (Persistent and Non-Persistent connections) should have equal chances to be processed first.

Flow

- NIC receives packets
- Raises hardware interrupt (if not in NAPI polling mode)
- NIC DMA packets into the RX Ring Buffer

Packet Arrival

- Kernel switches to NAPI polling
- Kernel computes 5-tuple hash (software) to determine CPU core
- Packet is forwarded to the target CPU core

NAPI Polling & Core Assignment

- Packet is enqueued to the socket's buffer on the target CPU.
- Trigger sk_data_ready() callback → wakes up epoll.

Socket Buffer & epoll

What Does NAPI Do?

- NAPI (New API) is a Linux kernel mechanism to handle high-speed packet processing efficiently.
- Hybrid Interrupt + Polling Model
  - Initial Interrupt:
  - When the first packet arrives, the NIC raises a hardware interrupt to notify the CPU.
  - Switch to Polling:
  - The kernel disables further interrupts for that RX queue and switches to polling mode.
  - The NAPI poll loop processes packets in bulk from the RX ring buffer.
  - Exit Polling:
  - When the buffer is empty, re-enable interrupts.

Why NAPI Polling?

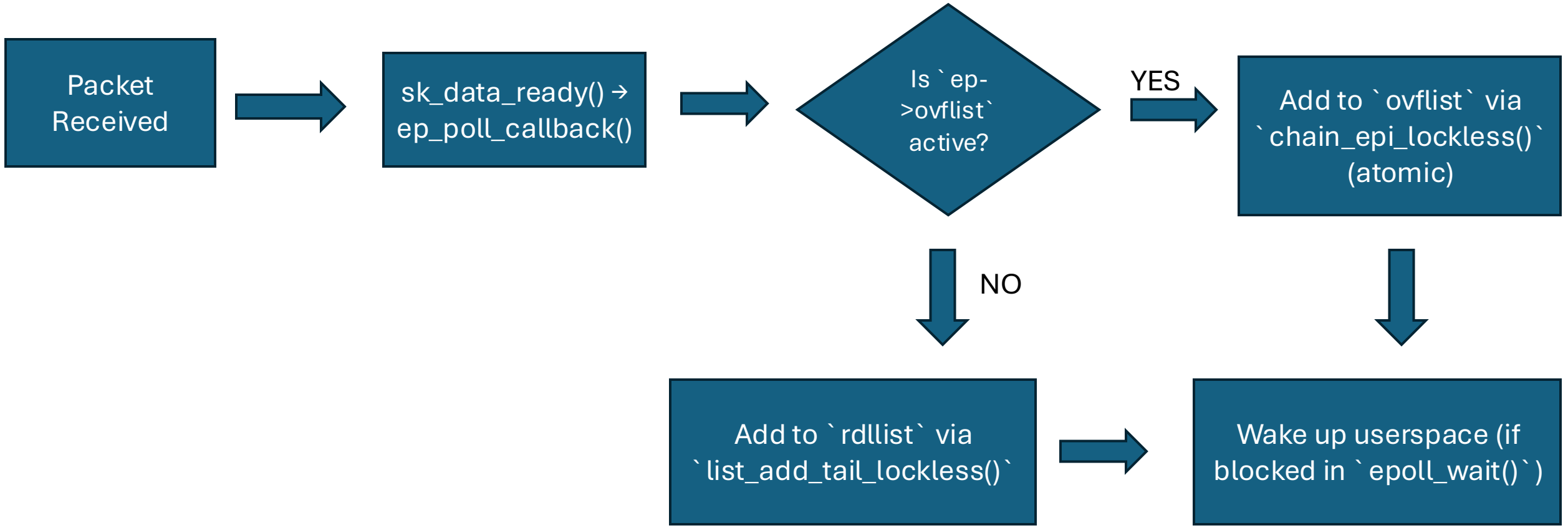
- Reduce Overhead:
- Under high traffic, interrupts (per-packet notifications) overload the CPU.
- Polling batches packets, improving throughput.

- Without NAPI: 10,000 packets → 10,000 interrupts.
- With NAPI: 10,000 packets → 1 interrupt + 1 polling loop.

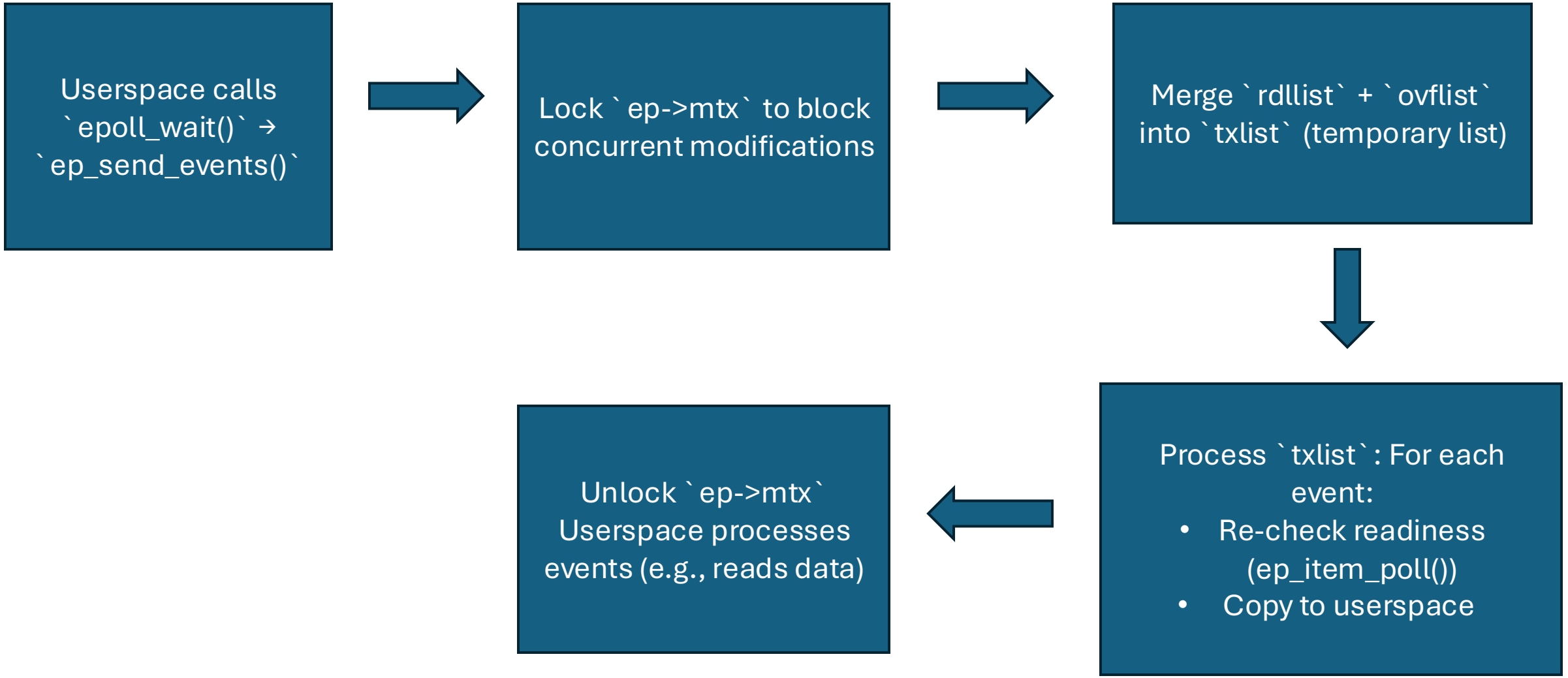
- Flow Summary

Packet → NIC → RX Ring Buffer (Queue X) → NAPI ID X → CPU Core X → Socket Buffer (Core X)

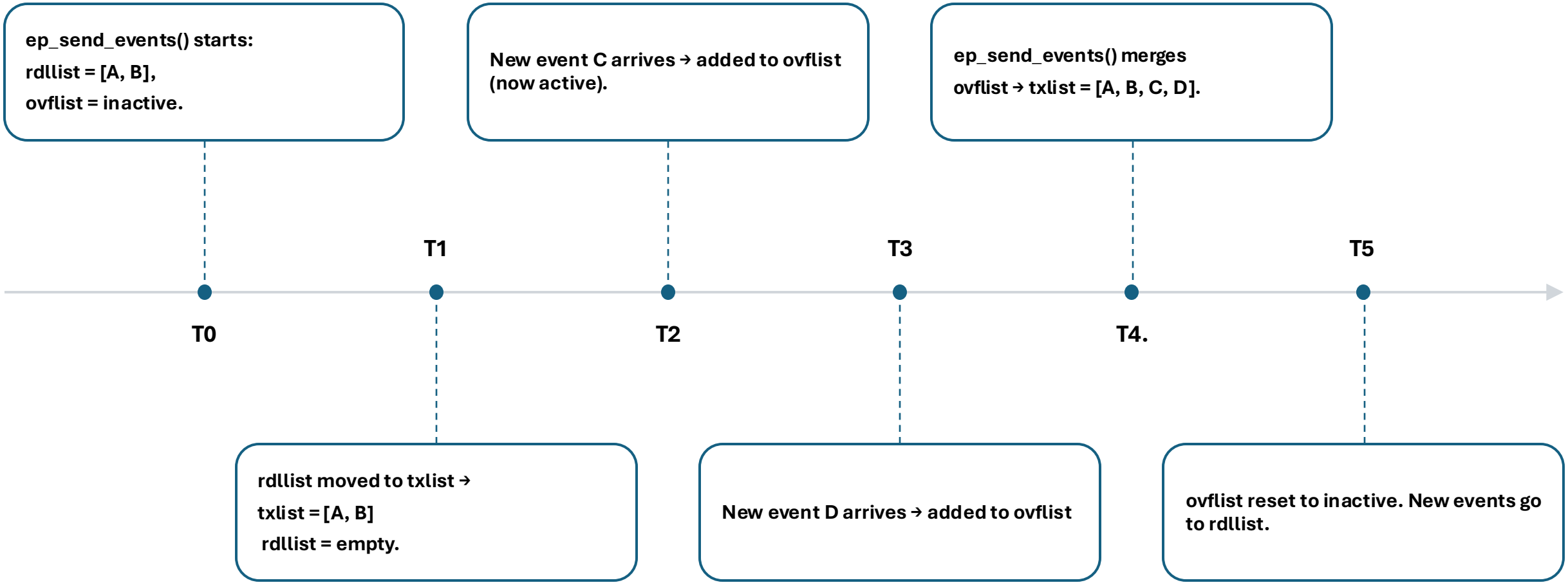
After sk_data_ready() ?



After sk_data_ready() ?



After sk_data_ready() ?



The Role of chain_epi_lockless()

When a socket becomes ready (e.g., data arrives), sk_data_ready() invokes ep_poll_callback().

```
static int ep_poll_callback(wait_queue_entry_t *wait, ...) {
  // 1. Check if events are being transferred (ovflist active)
  if (READ_ONCE(ep->ovflist) != EP_UNACTIVE_PTR) {
    // Use lockless atomic ops to add to ovflist
    chain_epi_lockless(epi);
  } else {
    // Add to rdllist (lockless)
    list_add_tail_lockless(&epi->rdllink, &ep->rdllist);
  }
  // 2. Wake up userspace blocked in epoll_wait()
  wake_up(&ep->wq);
}
```

The Role of list_add_tail_lockless()

- This function ensures atomic, lockless additions to ready_list

```
static inline bool list_add_tail_lockless(struct list_head *new, struct list_head *head) {
    struct list_head *prev;

    // Step 1: Atomic check for duplicates
    if (!try_cmpxchg(&new->next, &new, head))
        return false;

    // Step 2: Atomically update head->prev
    prev = xchg(&head->prev, new);

    // Step 3: Link new node into the list
    prev->next = new;
    new->prev = prev;

    return true;
}
```

The Role of chain_epi_lockless()

- This function ensures atomic, lockless additions to ovflist:
- cmpxchg: Ensures only one thread can mark epi->next as "in-use".
- xchg: Atomically appends the event to ovflist (no locks required).

```
static inline bool chain_epi_lockless(struct epitem *epi) {
  // 1. Check if epi is already in ovflist (fast path)
  if (epi->next != EP_UNACTIVE_PTR) return false;

  // 2. Atomic compare-and-swap: only one thread succeeds
  if (cmpxchg(&epi->next, EP_UNACTIVE_PTR, NULL) != EP_UNACTIVE_PTR)
    return false;

  // 3. Atomically update the tail of ovflist
  epi->next = xchg(&ep->ovflist, epi);
  return true;
}
```

Userspace Calls epoll_wait()?

This triggers ep_send_events(), which transfers events from the kernel to userspace.

```
static int ep_send_events(struct eventpoll *ep, ...) {
  // 1. Lock to prevent concurrent modifications
  mutex_lock(&ep->mtx);

  // 2. Atomically move rdllist + ovflist to a temporary list (txlist)
  ep_start_scan(ep, &txlist);

  // 3. Iterate over txlist and copy events to userspace
  list_for_each_entry_safe(epi, tmp, &txlist, rdllink) {
    revents = ep_item_poll(epi, ...); // Re-check if the event is still ready
    copy_to_user(events, revents);    // Send to userspace

  }

  // 4. Release lock
  mutex_unlock(&ep->mtx);
}
```

The Role of ep_start_scan()

- Merges rdllist and ovflist into txlist:

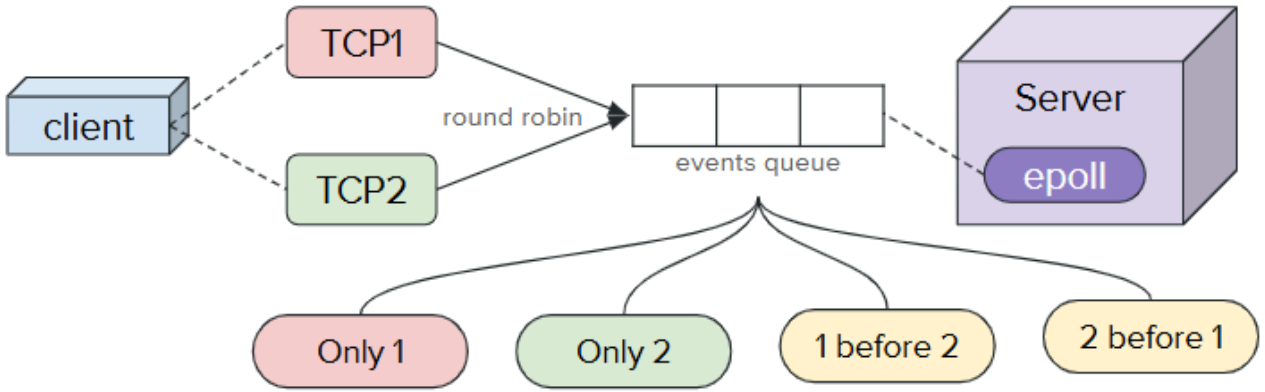
```
static void ep_start_scan(struct eventpoll *ep, struct list_head *txlist) {
  // 1. Move rdllist to txlist  rdllist → txlist (now empty)
  list_splice_init(&ep->rdllist, txlist);

  // 2. Atomically move ovflist to txlist
  WRITE_ONCE(ep->ovflist, EP_UNACTIVE_PTR);
  smp_mb(); // Memory barrier
  while (epi = xchg(&ep->ovflist, NULL)) {
    // Append ovflist events to txlist
    list_add_tail(&epi->rdllink, txlist);
  }
}
```

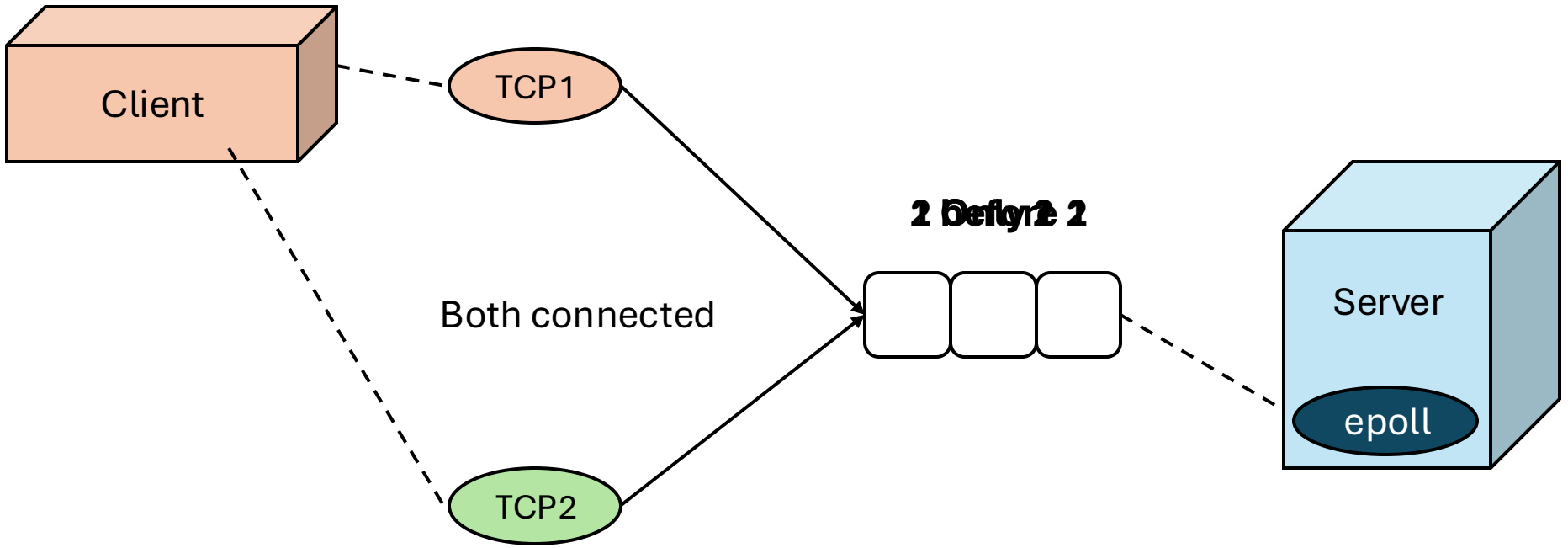
Experiments & Results

Experiment 1

- **Setup**
  - Server sets up socket
  - Registers server FD to epoll
  - Server accepts and registers client sockets and uses epoll to monitor FDs.
  - Client sends 2 TCP connections (Socket 1, Socket 2) in round-robin;
- **Event Cases**:
  - Only Socket 1: fd1 ready, fd2 not ready.
  - Only Socket 2: fd2 ready, fd1 not ready.
  - Socket 1 before Socket 2: Both ready, fd1 processed first.
  - Socket 2 before Socket 1: Both ready, fd2 processed first.

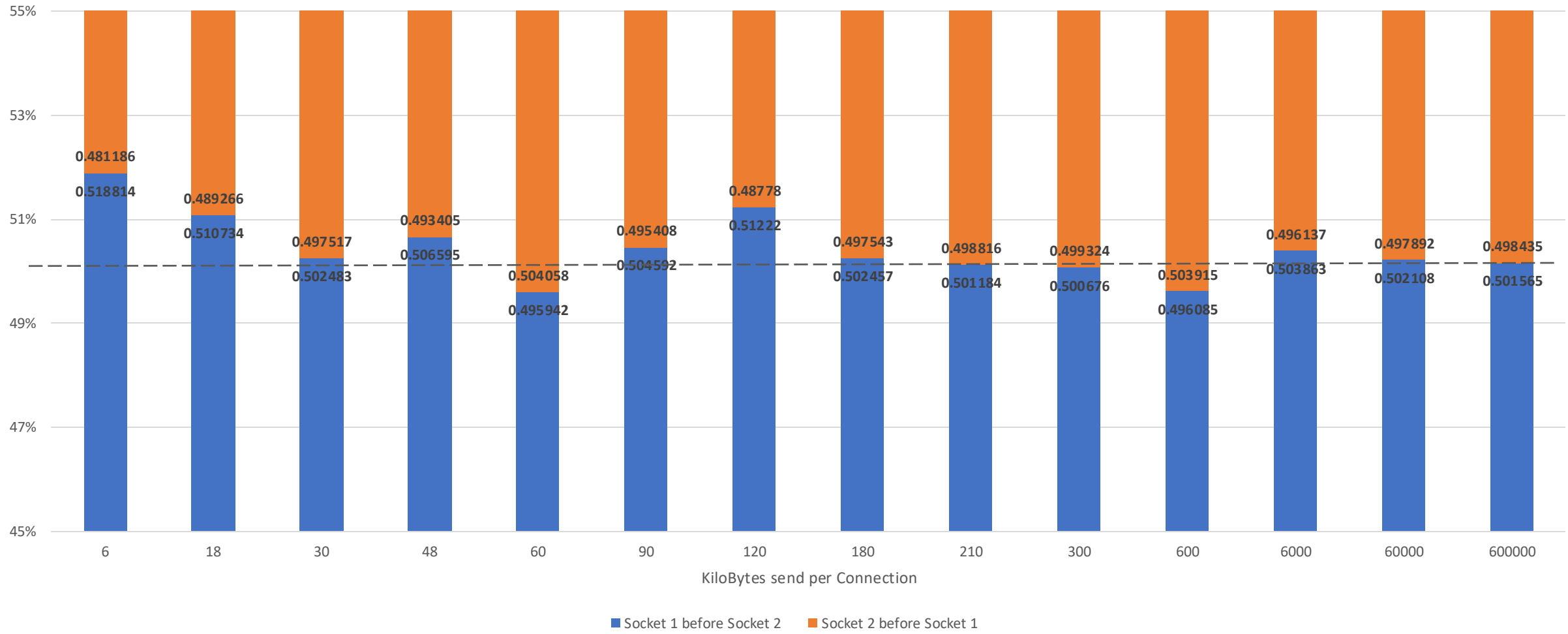


Experiment 1



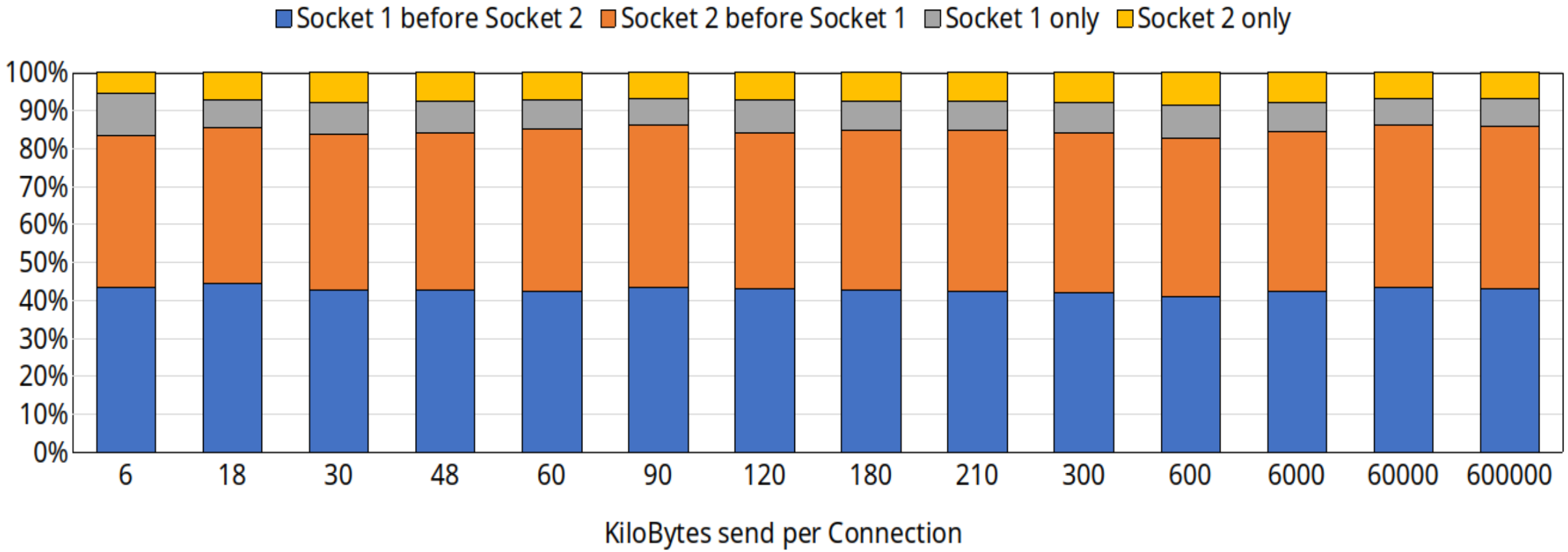
Sending different KB of 60B Packets

Each value is averaged over 20 runs.

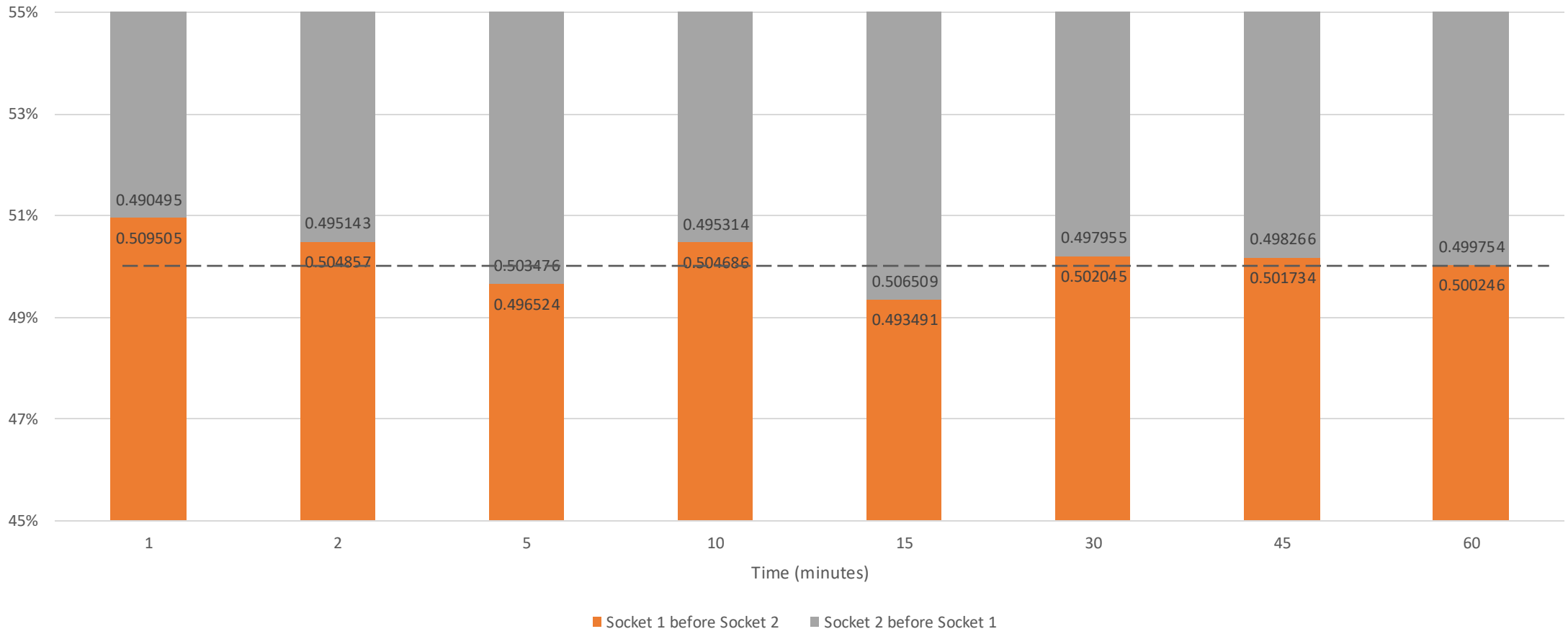


Sending different KB of 60B Packets

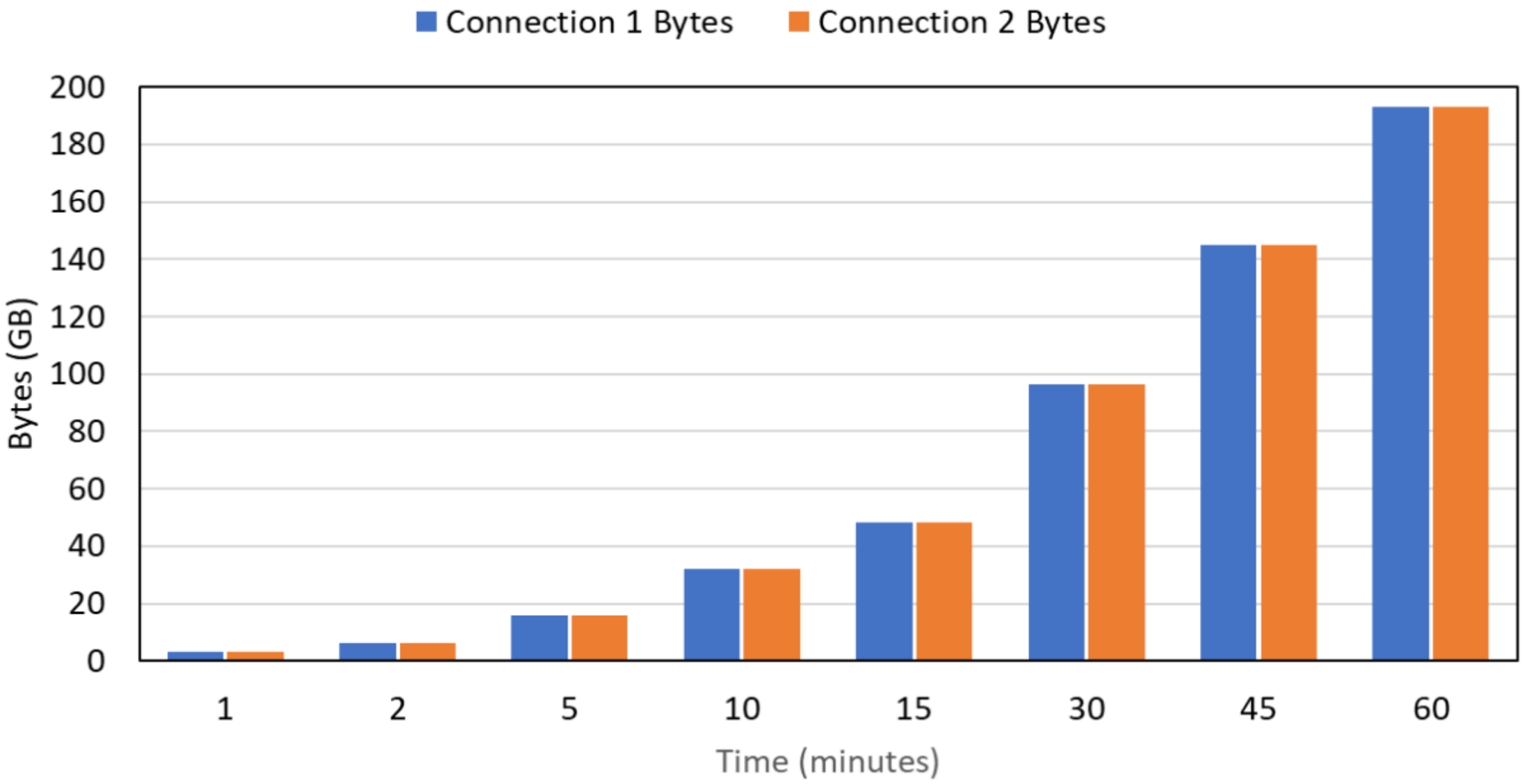
Each value is averaged over 20 runs.



Sending 1 million packets of 240B for different time intervals

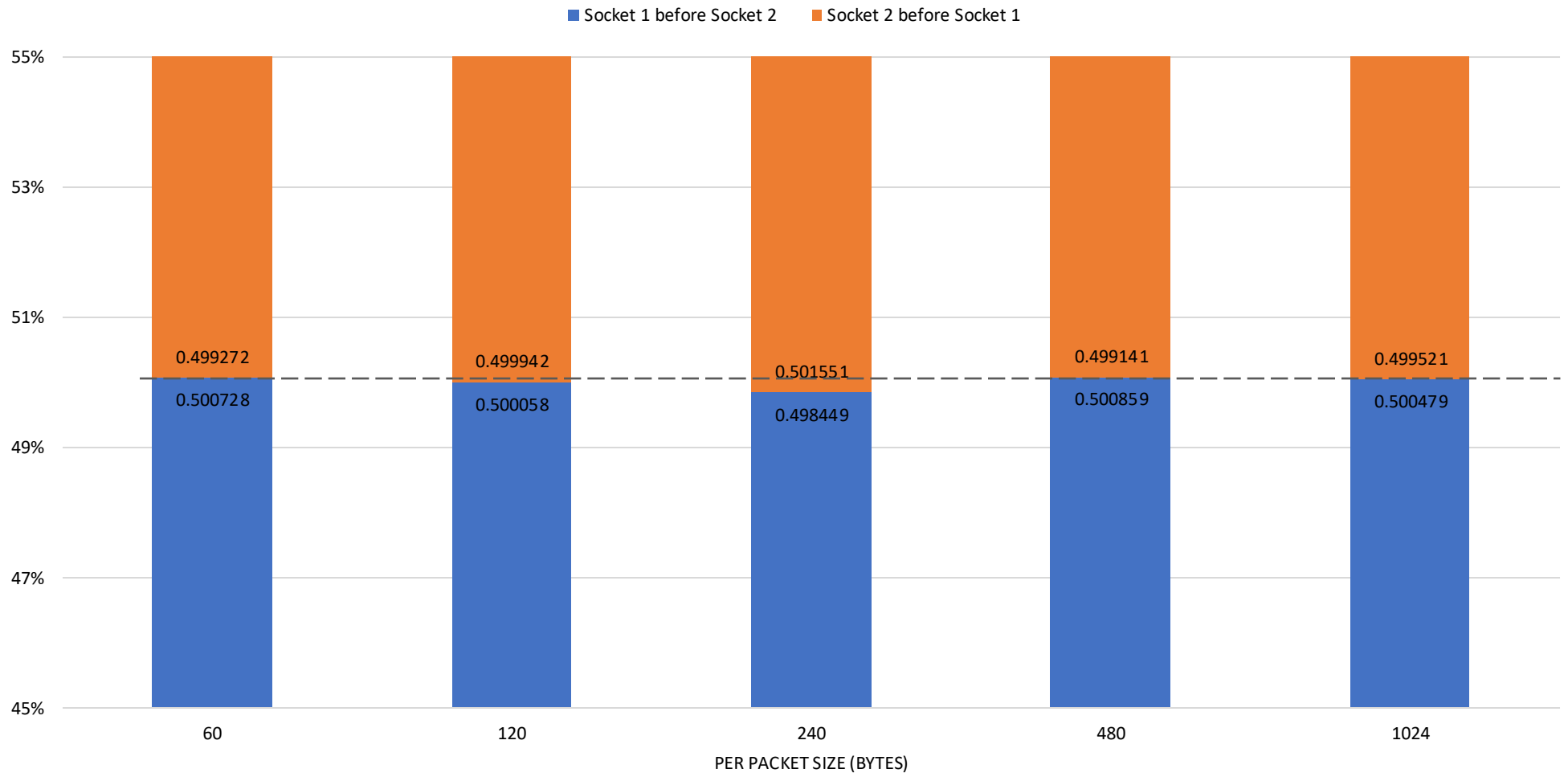


Sending 1 million packets of 240B for different time intervals



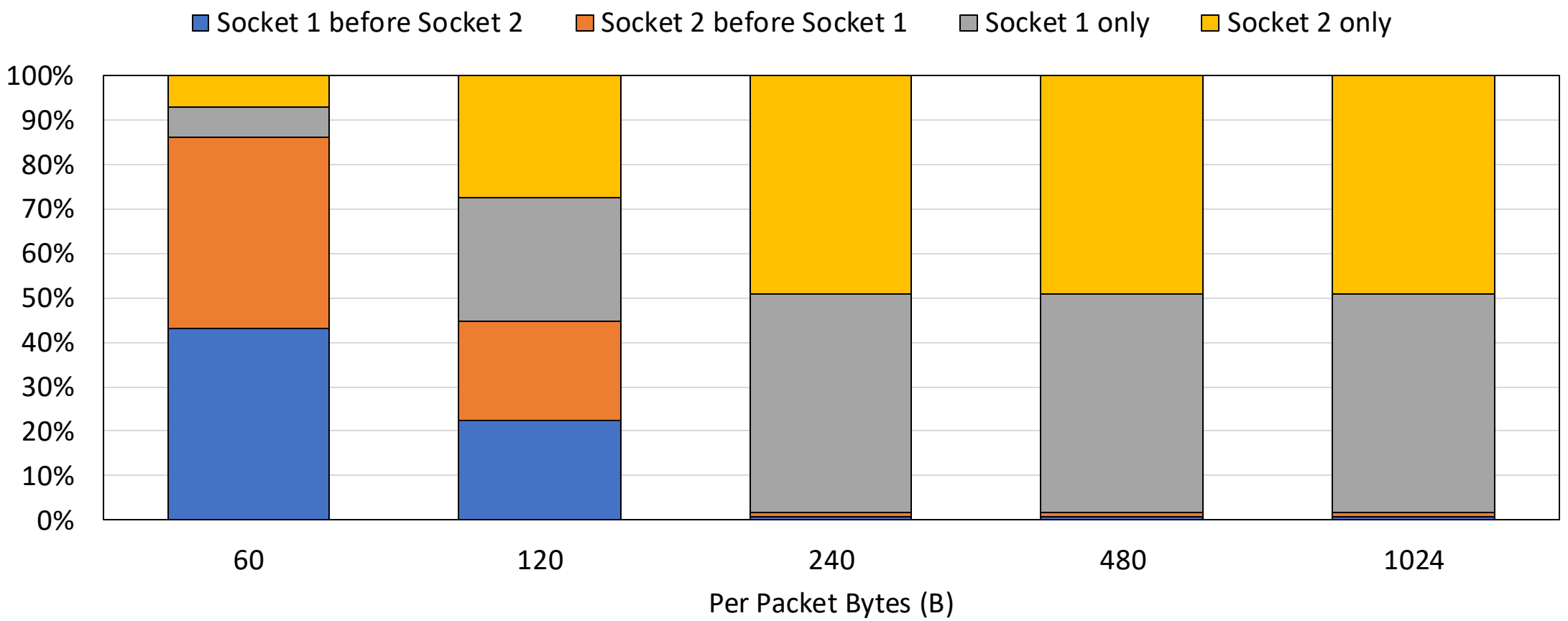
Changing per packet size and running for 1 hour

Send 16 million packets for 1 hour of different packet sizes
Each value is averaged over 5 runs.



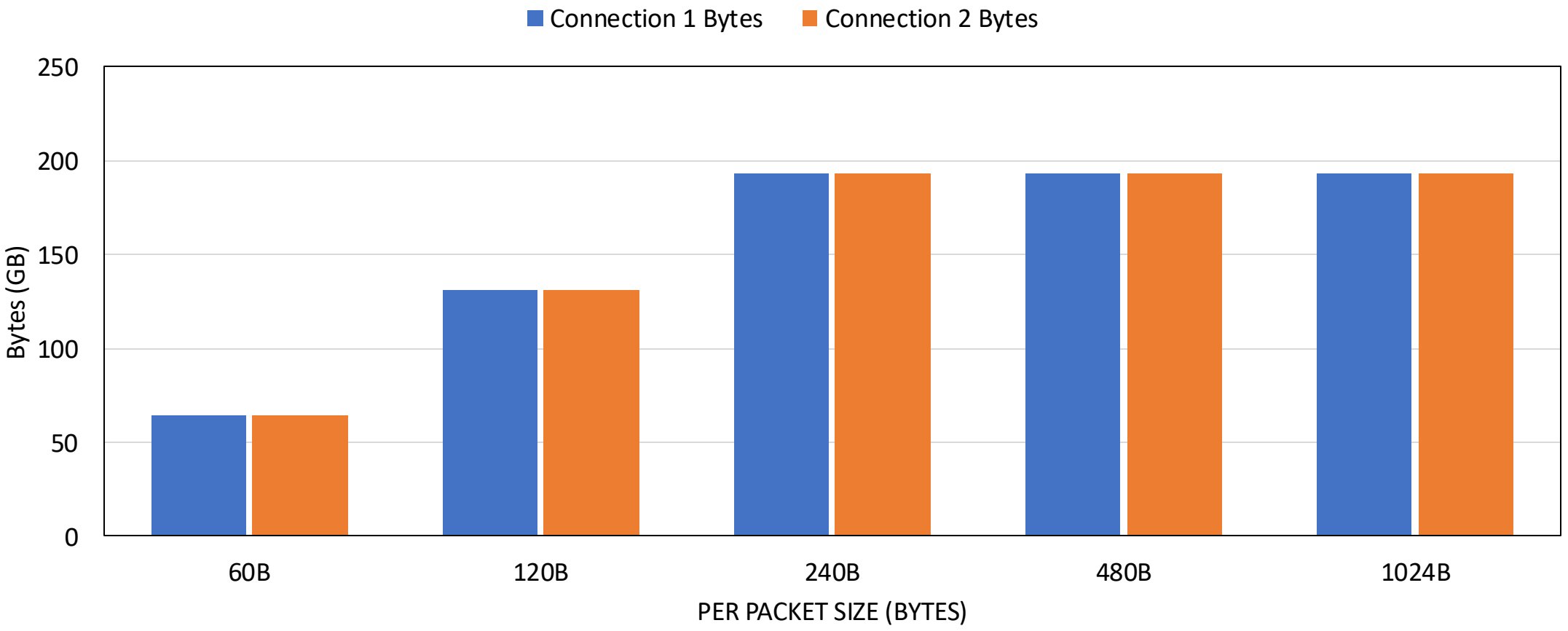
Changing per packet size and running for 1 hour

Send 16 million packets for 1 hour of different packet sizes
Each value is averaged over 5 runs.

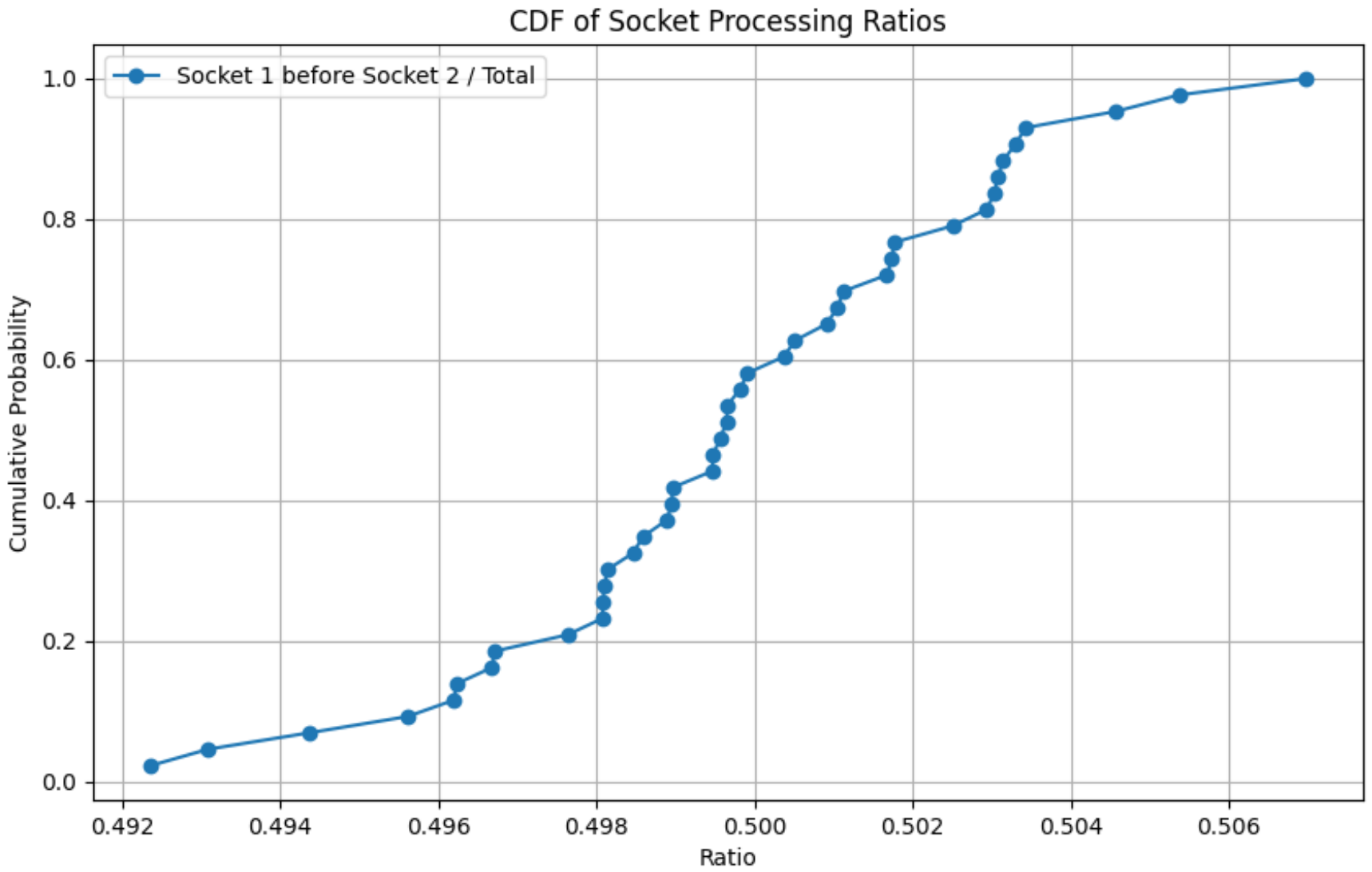


Changing per packet size and running for 1 hour

Send 16 million packets for 1 hour of different packet sizes
Each value is averaged over 5 runs.

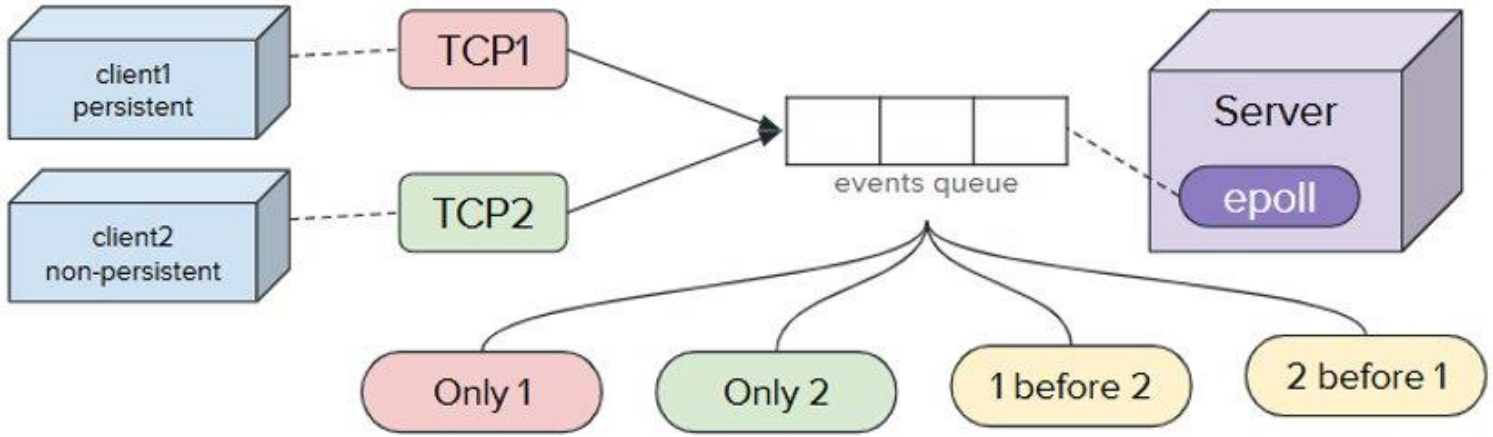


Sending 16 million 240B packet for 15 minutes (45 runs shown here)

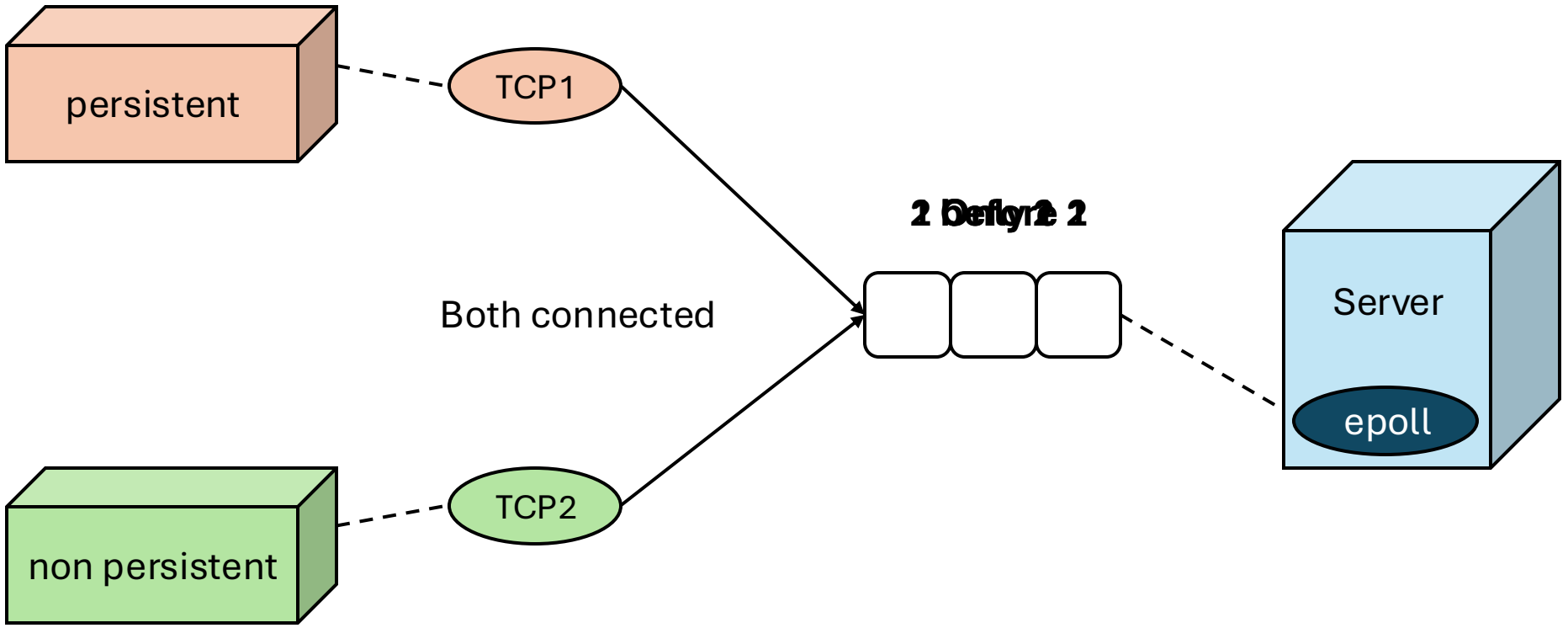


Experiment 2: Analyzing Bias in epoll for Persistent vs. Non-Persistent Connections

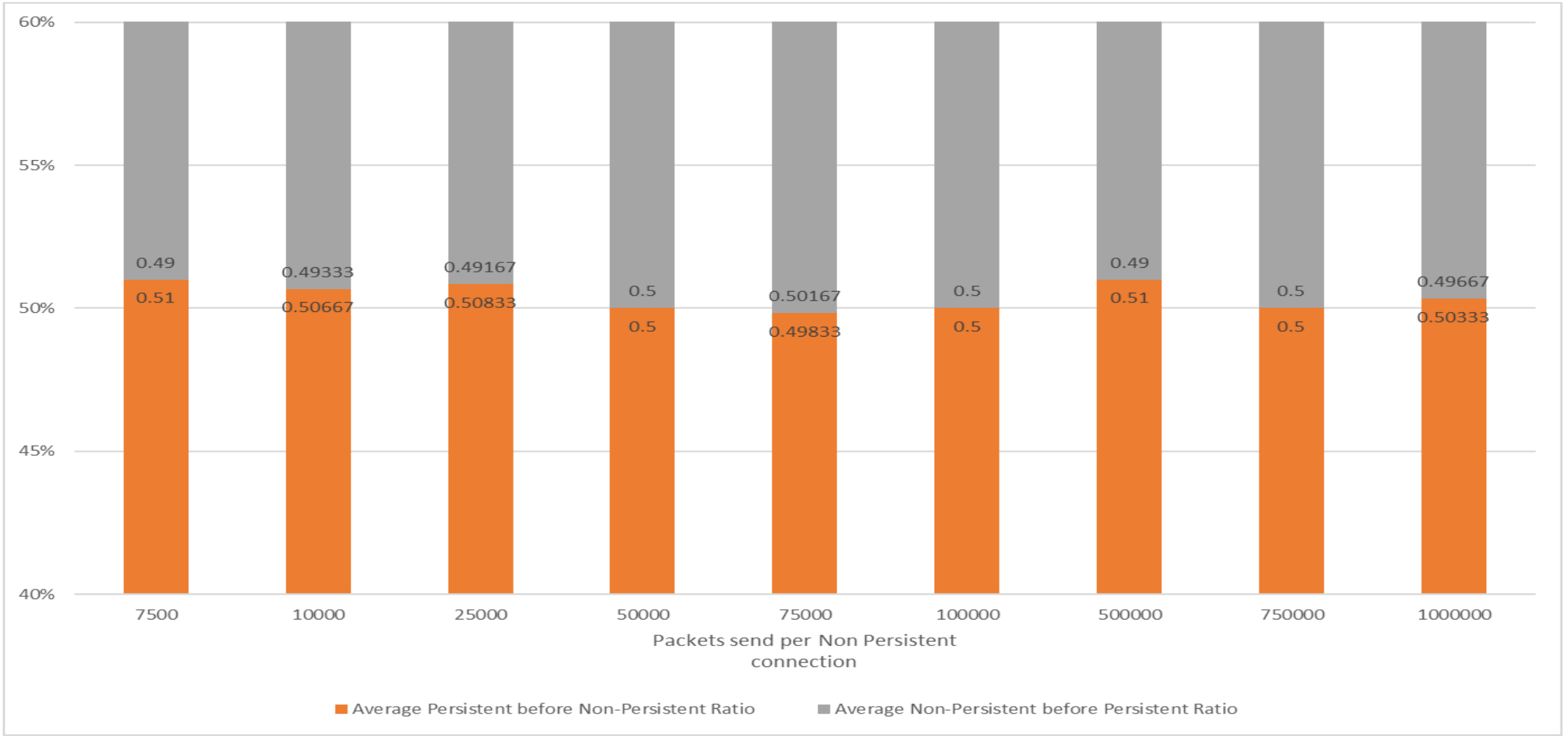
- **Setup**
  - Server uses epoll to handle two clients
  - Client 1: Persistent connection (keeps socket open).
  - Client 2: Non-persistent connection (Frequently creating new connection).
- **Goal**: Check for bias in event processing order.
- **Event Cases**:
  - Case 1: Only persistent connection active.
  - Case 2: Only non-persistent connection active.
  - Case 3: Both active, persistent chosen first.
  - Case 4: Both active, non-persistent chosen first.



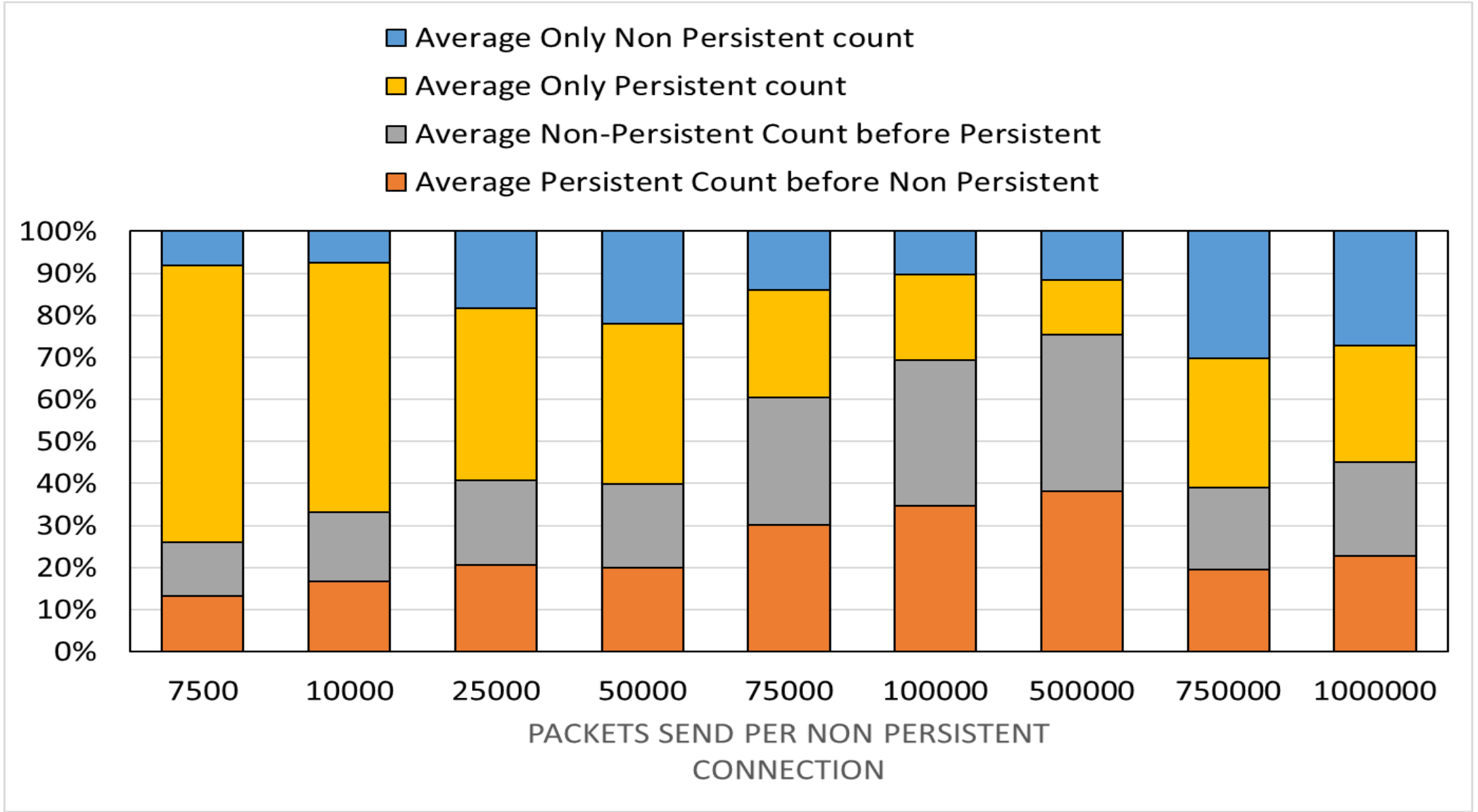
Experiment 2



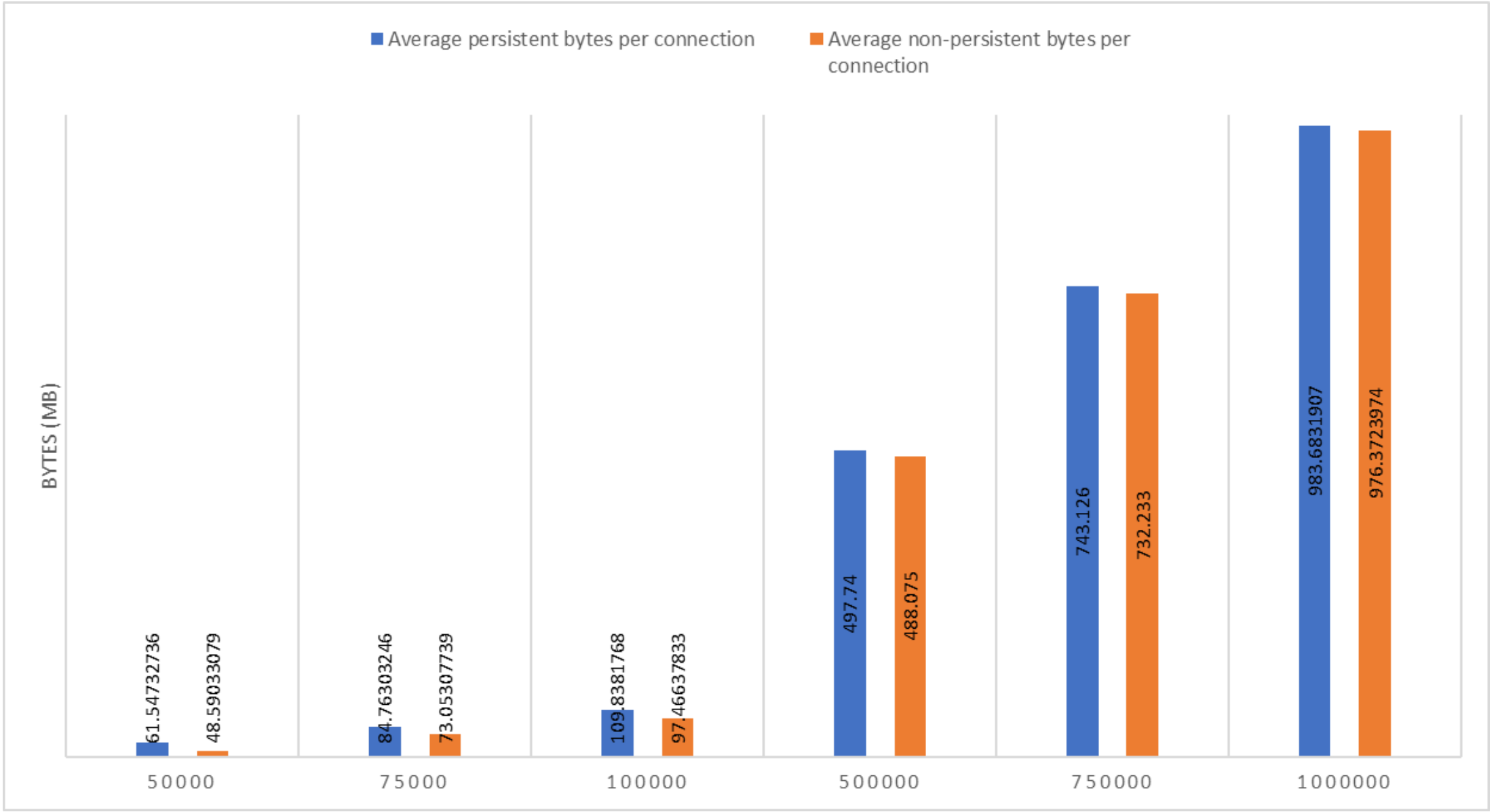
Packets send in each Non persistent connection



All four ratio vs Packets(1024B) send in each Non persistent connection



Bytes send vs Packets send in each Non persistent connection



Conclusion

- Explored I/O Multiplexing
- Three system call select(), poll(), epoll()
- Observed the nearly 50/50 split in first experiment
- Fairness was held under different test parameters for experiment 1.
- The ratio of Persistent-before-Non-Persistent is slightly above 0.5
- epoll is generally fair for both types of experiments but there may be a subtle preference towards Persistent connections.

References

- https://fasterdata.es.net/network-tuning/
- https://blog.packagecloud.io/monitoring-tuning-linux-networking-stack-receiving-data/
- https://copyconstruct.medium.com/the-method-to-epolls-madness-d9d2d6378642
- https://man7.org/linux/man-pages/man2/epoll_create.2.html
- https://man7.org/linux/man-pages/man2/epoll_ctl.2.html
- https://github.com/torvalds/linux/blob/master/fs/eventpoll.c