Analyzing the Fairness of the Scheduling of Epoll

CS692: Research and Development Report

Computer Science and Engineering

Master of Technology

by

AKANKSHA DADHICH

(Roll No. 23M0830)

Under the guidance of

Prof. Ashutosh Gupta and Prof. Mythili Vutukuru

Department of Computer Science and Engineering
Indian Institute of Technology Bombay

Acknowledgement

I express my sincere gratitude to my guides, Prof. Ashutosh Gupta and Prof. Mythili Vutukuru, for their invaluable guidance, support, and the opportunity to work on this pertinent research topic. Their insights and encouragement were instrumental throughout the course of this study.
Akanksha Dadhich
Department of Computer Science and Engineering
IIT Bombay

Abstract

Input/Output (I/O) multiplexing is a fundamental technique for building scalable network servers that can handle thousands of concurrent client connections efficiently. Among various I/O multiplexing mechanisms like `select()` and `poll()`, `epoll()` has become the de-facto standard on Linux systems due to its superior performance and scalability. Given its widespread use in latency-sensitive and high-throughput applications such as financial trading systems, web servers, and real-time communication platforms, understanding the fairness of `epoll()` in scheduling events from multiple file descriptors (FDs) is crucial.

This report investigates the scheduling fairness of the `epoll` mechanism. We begin by contrasting traditional blocking I/O with I/O multiplexing, highlighting the need for efficient event handling. We then briefly discuss `select()` and `poll()`, leading into a detailed examination of `epoll`, including its system calls (`epoll_create()`, `epoll_ctl()`, `epoll_wait()`) and the internal mechanisms involved in packet processing from Network Interface Card (NIC) arrival through NAPI (New API) and `epoll`'s ready list management.

Two primary experiments were conducted. The first experiment involved a server using `epoll` to manage two TCP connections from a single client, with the client sending packets in a round-robin fashion. Various scenarios were tested, including different total data volumes, different per-packet sizes, and varying time intervals. The second experiment introduced heterogeneity by having one client maintain a persistent connection while the other established non-persistent connections (new connection per request), both sending data to the server.

The experimental results, presented through graphical analysis of connection processing ratios and data throughput, consistently indicate that `epoll` provides fair scheduling. In both homogeneous and heterogeneous connection scenarios, the processing of events and data transfer for concurrent connections was observed to be almost equally distributed (approximately 50-50 ratio). This suggests that `epoll` does not inherently bias one connection over another under the tested conditions. The report concludes with these findings and suggests future work, such as kernel-level tracing of FD processing order, to further validate these observations under more complex scenarios.

Contents

List of Figures

Chapter 1

Introduction

Efficiently handling concurrent client connections is a cornerstone of modern server architecture. Network applications, ranging from web servers and databases to real-time communication platforms and financial trading systems, often need to manage thousands, if not millions, of simultaneous connections. The manner in which a server handles Input/Output (I/O) operations for these connections significantly impacts its performance, scalability, and responsiveness.

1.1 Traditional Blocking I/O

In traditional server models, blocking I/O is often the simplest approach. When a server performs an I/O operation, such as reading data from a client socket (`read()`), the calling process or thread blocks until the operation is complete (i.e., data is available or an error occurs). While simple for a single connection, this model becomes highly inefficient for concurrent connections. If a server uses a single thread/process to handle multiple connections sequentially, it will be stuck waiting for one client if that client is slow or sends no data, thereby starving all other clients.

A common workaround is to use a multi-process or multi-threaded model, where each connection is handled by a separate process or thread. However, this approach has its own limitations:

- **Resource Overhead:** Creating and managing a large number of processes or threads consumes significant system resources (memory, CPU for context switching).
- **Scalability Limits:** There's a practical limit to the number of concurrent threads/processes a system can efficiently handle, often far less than the desired number of client connections (e.g., C10k problem).

Figure 1.1 illustrates the problem with a single-threaded server using blocking I/O. When Connection 1 is blocked on a `read()` call, Connections 2 and 3 cannot be processed.

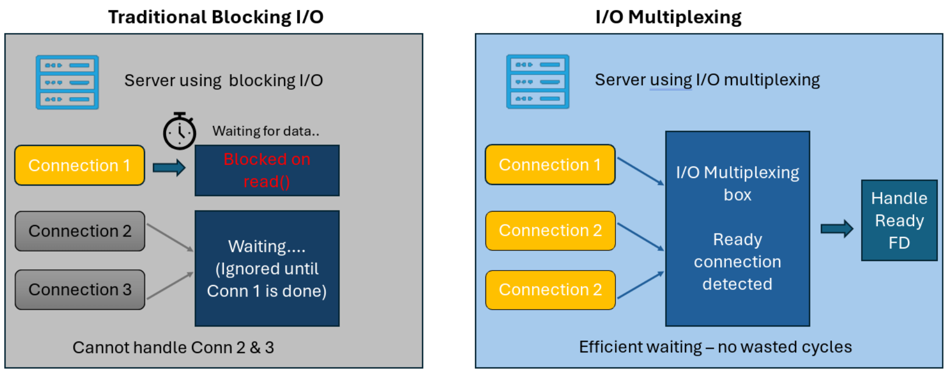


Figure 1.1: Traditional Blocking I/O vs. I/O Multiplexing in Servers

1.2 I/O Multiplexing: The Need for Efficiency

To overcome the limitations of blocking I/O and the resource-intensive nature of one-thread-per-connection models, I/O multiplexing was introduced. I/O multiplexing allows a single process/thread to monitor multiple file descriptors (FDs) – representing network sockets, pipes, files, etc. – to see if any of them are ready for an I/O operation (e.g., data available for reading, buffer ready for writing).

The core idea is to wait for events on a set of FDs rather than blocking on a single FD. When the I/O multiplexing system call returns, it indicates which FDs are ready, allowing the server to process them without blocking. This approach, depicted on the right side of Figure 1.1, enables a server to handle many connections concurrently within a single thread (or a small pool of threads), leading to:

- **Reduced Resource Consumption:** Fewer threads/processes are needed.
- **Improved Scalability:** Servers can handle a much larger number of concurrent connections.
- **Efficient Waiting:** CPU cycles are not wasted polling each FD individually or blocking idly.

I/O multiplexing is crucial for server applications, such as those handling thousands of clients, as blocking on one connection would prevent others from being served. It allows the program to wait for multiple FDs to become ready and avoids wasting CPU cycles by polling each one, as overviewed in Figure 1.2.

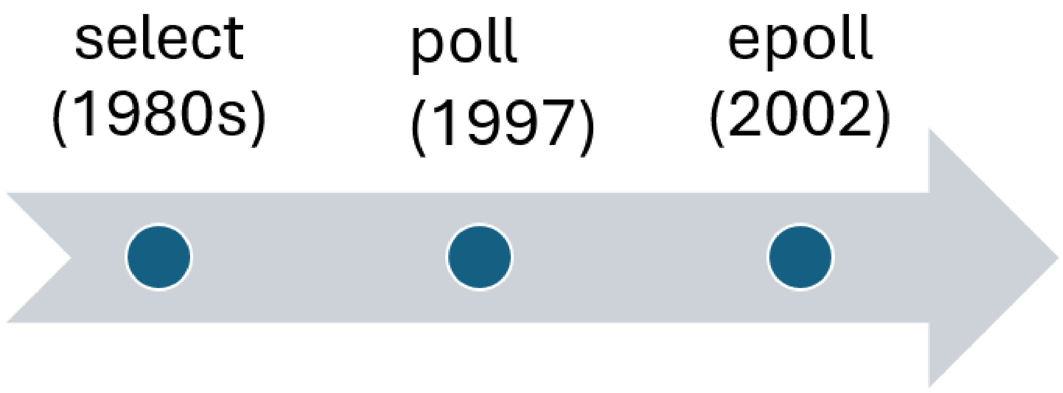


Figure 1.2: Overview of I/O Multiplexing Calls

This report focuses on `epoll`, a Linux-specific I/O multiplexing mechanism, and investigates its fairness in scheduling events from multiple connections, a critical aspect for ensuring quality of service in high-performance systems.

Chapter 2

I/O Multiplexing Mechanisms

Over the years, several system calls have been developed to provide I/O multiplexing capabilities. The most common ones are `select()`, `poll()`, and the Linux-specific `epoll()`. Their timeline of introduction is also indicated in Figure 1.2.

2.1 `select()`

The `select()` system call was one of the earliest I/O multiplexing mechanisms, introduced in the 1980s. It allows a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some class of I/O operation.

2.1.1 How `select()` Works

The typical workflow for `select()` is shown in Figure 2.1:

1. **Initialize `fd_sets`:** Three sets of file descriptors (`fd_set`) are prepared: one for read readiness, one for write readiness, and one for exceptional conditions. These are typically fixed-size bitmasks.
2. **Add FDs (`FD_SET`):** The program adds the FDs it wants to monitor to the appropriate `fd_set`(s) using macros like `FD_ZERO` (to clear a set) and `FD_SET` (to add an FD to a set).
3. **Call `select()`:** The `select(nfds, readfds, writefds, exceptfds, timeout)` system call is invoked. `nfds` is the highest-numbered file descriptor in any of the three sets plus 1. The `fd_sets` are passed by value (copied into the kernel).
4. **Kernel Processing:** The kernel waits until at least one FD in the specified sets becomes ready or the timeout expires. Upon return, the kernel *modifies* the `fd_sets` in place to indicate which FDs are ready.
5. **Check Ready FDs (`FD_ISSET`):** The program must iterate through all FDs it originally added to the sets and use the `FD_ISSET` macro to check if each FD is still set, meaning it's ready.

6. **Process Ready FDs:** Perform non-blocking I/O operations on the ready FDs.
7. **Repeat:** The `fd_sets` must be re-initialized and re-populated before calling `select()` again, as the kernel modifies them.

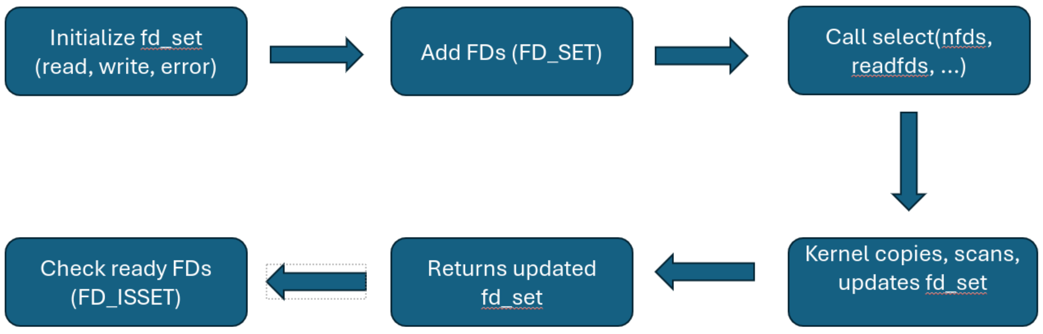

Initialize fd_set (read, write, error)
Add FDs (FD_SET)
Call select(nfds, readfds, ...)
Kernel copies, scans, updates fd_set
Returns updated fd_set
Check ready FDs (FD_ISSET)

Figure 2.1: Workflow of the `select()` System Call

2.1.2 Defects of `select()`

Despite its utility, `select()` has several significant drawbacks, especially for applications with a large number of connections, as listed in Figure 2.2:

- **FD Limit:** `select()` uses a fixed-size bit array (`fd_set`), typically limited by `FD_SETSIZE` (often 1024). This makes it non-scalable for thousands of connections.
- **Copy Overhead:** On each call, the entire `fd_sets` are copied between user space and kernel space, and then back. This is inefficient if the sets are large but only a few FDs are active.
- **Linear Scan ($O(n)$ for event detection):** After `select()` returns, the application must iterate through all FDs in the set (up to `nfds`) to find out which ones are ready. This is $O(n)$ complexity, where n is the maximum FD number.
- **Inefficient for Idle Connections:** The kernel also has to scan all FDs passed to it. If many connections are idle, this is wasteful.
- **Stateful Modification:** Since `fd_sets` are modified by the kernel, they must be re-initialized before each call.

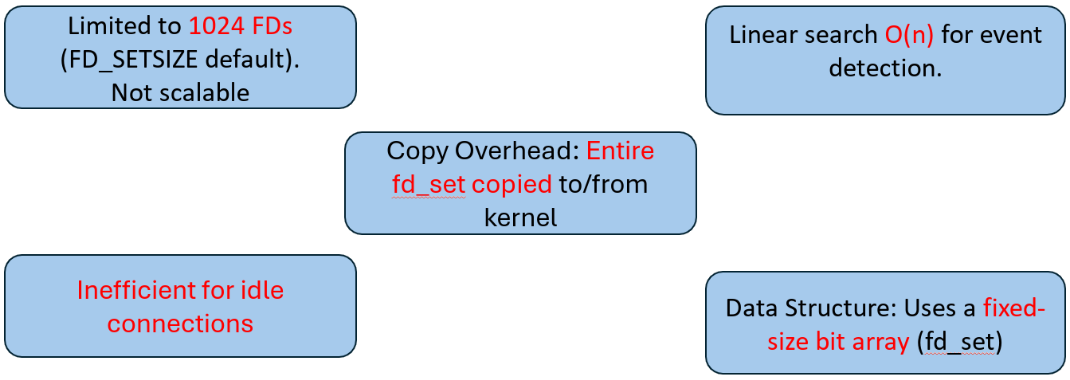

Limited to 1024 FDs (FD_SETSIZE default). Not scalable
Linear search O(n) for event detection.
Copy Overhead: Entire fd_set copied to/from kernel
Inefficient for idle connections
Data Structure: Uses a fixed-size bit array (fd_set)

Figure 2.2: Limitations of the `select()` System Call

2.2 `poll()`

The `poll()` system call, introduced around 1997, was designed to address some of the limitations of `select()`, particularly the `FD_SETSIZE` limit.

2.2.1 How `poll()` Works

Figure 2.3 illustrates the operation of `poll()`:

1. **Initialize `struct pollfd` array:** Instead of `fd_sets`, `poll()` uses an array of `struct pollfd`. Each structure contains:
    - `fd`: The file descriptor to monitor.
    - `events`: A bitmask of events the application is interested in for this FD (e.g., `POLLIN` for readable, `POLLOUT` for writable).
    - `revents`: A bitmask filled by the kernel with the events that actually occurred.
2. **Set Events:** The application populates the `fd` and `events` fields for each FD it wants to monitor.
3. **Call `poll()`:** The `poll(struct pollfd *fds, nfds_t nfds, int timeout)` system call is invoked. `fds` is a pointer to the array, `nfds` is the number of items in the array.
4. **Kernel Processing:** The kernel waits for events or timeout. It then fills the `revents` field in each `struct pollfd` for which an event occurred.
5. **Check `revents`:** The application iterates through the array and checks the `revents` field of each structure to determine which FDs are ready and for what events.
6. **Process Ready FDs:** Perform I/O on ready FDs.

7. **Repeat:** The `events` fields usually don't need to be reset unless the application changes what it's interested in, but the `revents` field is cleared by the application or kernel on the next call.

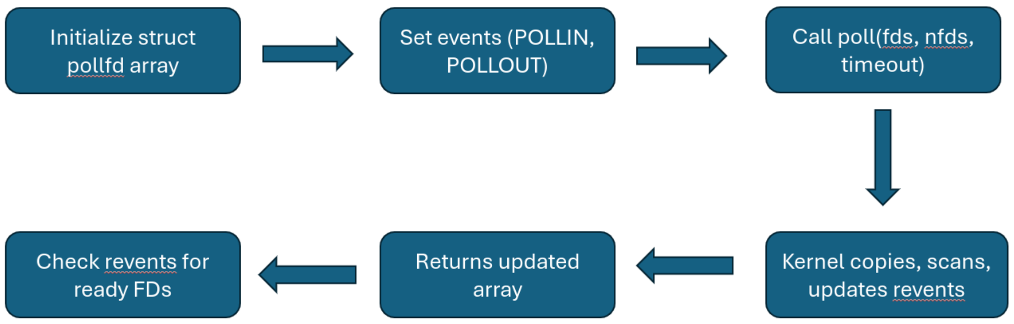

Initialize struct pollfd array
Set events (POLLIN, POLLOUT)
Call poll(fds, nfds, timeout)
Kernel copies, scans, updates revents
Returns updated array
Check revents for ready FDs

Figure 2.3: Workflow of the `poll()` System Call

2.2.2 Advantages and Disadvantages of `poll()`

`poll()` offers some improvements over `select()`, as shown in Figure 2.4:

- **No FD Limit:** It uses a dynamic array of `struct pollfd`, so it's not limited by `FD_SETSIZE` and is more scalable in terms of the number of FDs it can monitor.

However, it still suffers from some of the same performance issues:

- **Copy Overhead:** The entire `struct pollfd` array is copied between user space and kernel space on each call.
- **Linear Scan ($O(n)$):** Both the application (after `poll()` returns) and the kernel typically need to scan the entire array of FDs. This is $O(n)$ complexity, where n is the number of FDs being monitored.
- **Inefficient for Idle Connections:** Similar to `select()`, performance degrades as the number of monitored FDs increases, especially if many are idle.

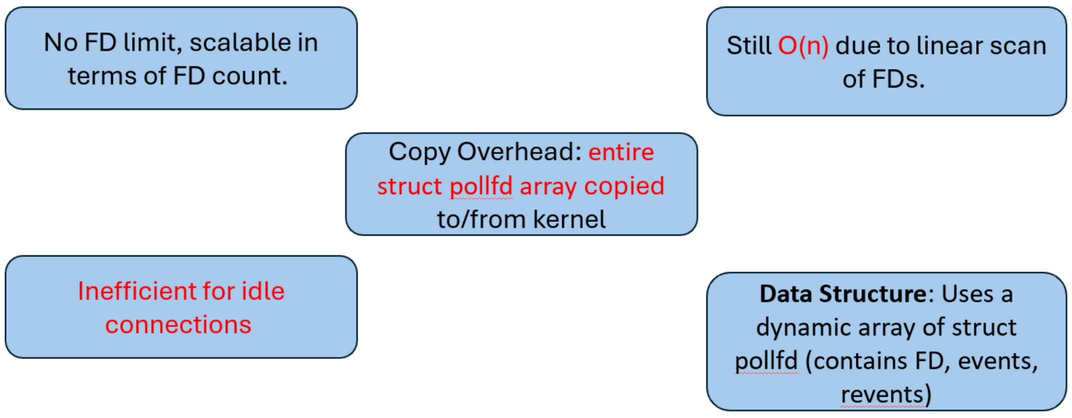

No FD limit, scalable in terms of FD count.
Still O(n) due to linear scan of FDs.
Copy Overhead: entire struct pollfd array copied to/from kernel
Inefficient for idle connections
Data Structure: Uses a dynamic array of struct pollfd (contains FD, events, revents)

Figure 2.4: Characteristics of the `poll()` System Call

2.3 epoll()

To address the scalability and performance issues of `select()` and `poll()`, Linux introduced `epoll()` (event poll) around 2002. It is specifically designed for applications that need to handle a very large number of concurrent connections efficiently.

`epoll` offers a more sophisticated and performant way to manage I/O events:

- **Kernel-based FD List:** Instead of the application passing the entire list of FDs to monitor on each call, `epoll` maintains a persistent list of FDs in the kernel space associated with an `epoll` instance. FDs are added, modified, or removed using a separate system call (`epoll_ctl`).
- **Event-driven Notification:** When `epoll_wait()` is called, it only returns the FDs that are actually ready, rather than requiring the application to scan all monitored FDs.
- **Efficient Data Structures:** Internally, `epoll` often uses efficient data structures like red-black trees to store monitored FDs and a linked list (ready list) for FDs that have events pending. This can lead to $O(1)$ performance for adding/removing FDs and for retrieving ready FDs when the number of active FDs is small compared to the total number of monitored FDs (illustrated conceptually in Figure 2.5).
- **Edge-Triggered (ET) and Level-Triggered (LT) Modes:** `epoll` supports both modes.
  - **Level-Triggered (LT):** (Default) `epoll_wait` will report an event as long as the condition holds (e.g., data available to read). If you don't read all data, the next `epoll_wait` will report it again. Similar to `select`/`poll`.
  - **Edge-Triggered (ET):** `epoll_wait` reports an event only when a change occurs on the FD (e.g., new data arrives). If you get an event and don't process all available data, `epoll_wait` will not report it again until another new event occurs. This mode requires careful non-blocking I/O handling to avoid starvation.

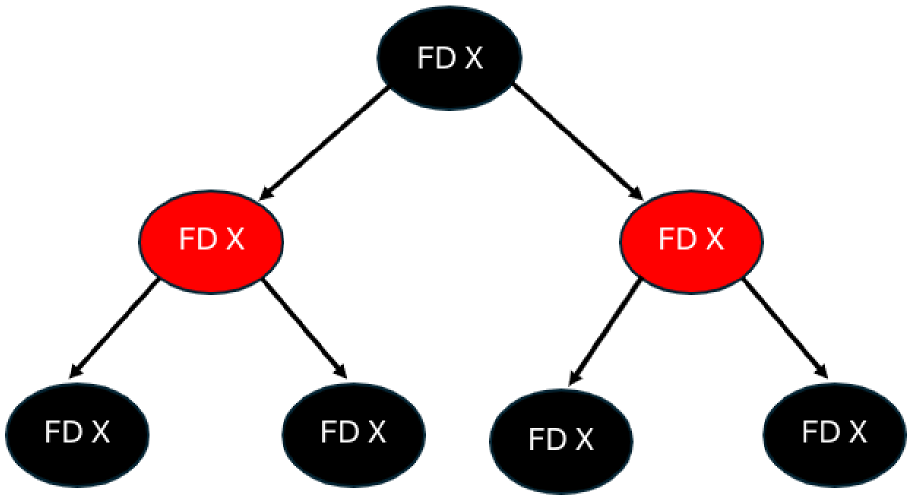


Figure 2.5: `epoll` Internal Structure Concept

The benefits of `epoll` are summarized in Figure 2.6: scalability, O(1) event detection (for ready list), low copy overhead (only ready FDs are copied back), no FD limit, and efficiency for idle connections.

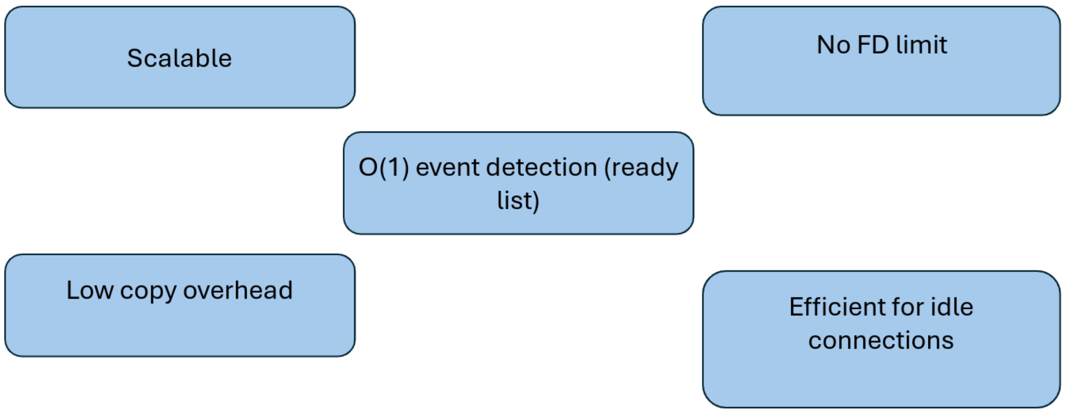

Scalable
No FD limit
O(1) event detection (ready list)
Low copy overhead
Efficient for idle connections

Figure 2.6: Advantages of the `epoll()` System Call

Due to these advantages, `epoll` has become the preferred I/O multiplexing mechanism on Linux for high-performance network applications. The remainder of this report will delve deeper into `epoll`'s workings and analyze its scheduling fairness.

Chapter 3

Deep Dive into epoll

The epoll API consists of three main system calls: epoll_create(), epoll_ctl(), and epoll_wait(). These calls work together to provide an efficient event notification mechanism. A conceptual overview of their usage is shown in the code snippet in Figure 3.1.

```
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
```

Figure 3.1: Overview of epoll System Call Usage

3.1 epoll System Calls

3.1.1 epoll_create() / epoll_create1()

int epoll_create(int size); (deprecated)
int epoll_create1(int flags);

This system call creates an epoll instance in the kernel and returns a file descriptor that refers to this instance. This FD is then used in subsequent epoll_ctl() and epoll_wait() calls.

- epoll_create(size): The size argument was a hint to the kernel about the number of FDs expected to be monitored. Since Linux 2.6.8, this argument is ignored (the kernel dynamically sizes its internal structures), but must be greater than zero. This call is now deprecated in favor of epoll_create1().

- `epoll_create1(flags)`: This is the modern version. If `flags` is 0, it behaves like `epoll_create()`. The most common flag is `EPOLL_CLOEXEC`, which causes the `epoll` file descriptor to be automatically closed when any `exec`-family function is called.

Figure 3.2 visualizes the creation of an epoll instance associated with a file descriptor (`epfd`) within a process.

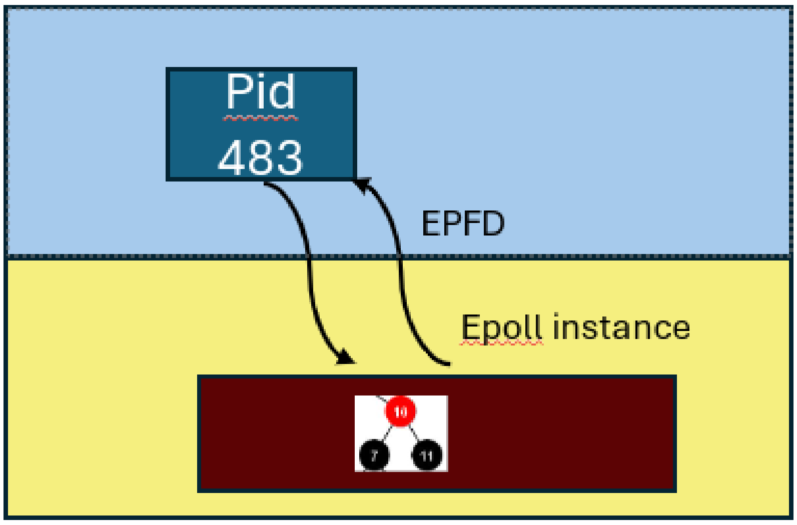


Figure 3.2: Creating an `epoll` Instance with `epoll_create1()`

3.1.2 `epoll_ctl()`

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

This system call is used to add, modify, or remove file descriptors from the interest list (or "epoll set") of the `epoll` instance `epfd`.

- `epfd`: The file descriptor representing the `epoll` instance, returned by `epoll_create1()`.
- `op`: The operation to be performed:
  - `EPOLL_CTL_ADD`: Add `fd` to the interest list and associate the settings specified in `event` with it.
  - `EPOLL_CTL_MOD`: Modify the event settings associated with `fd`.
  - `EPOLL_CTL_DEL`: Remove `fd` from the interest list. The `event` argument is ignored but must be non-NULL for older kernels (nowadays can be NULL).
- `fd`: The file descriptor to be targeted by the operation.
- `event`: A pointer to a `struct epoll_event`:

```
struct epoll_event {
    uint32_t     events;      /* Epoll events */
    epoll_data_t data;        /* User data variable */
};
```

  - `events`: A bitmask of events the application is interested in for `fd`. Examples include `EPOLLIN` (data available for read), `EPOLLOUT` (buffer ready for write), `EPOLLPRI` (urgent data), `EPOLLERR` (error), `EPOLLHUP` (hang-up), `EPOLLET` (edge-triggered mode).
  - `data`: A union (`epoll_data_t`) that allows the application to store arbitrary data (e.g., a pointer, an integer, the FD itself) associated with the `fd`. This data is returned when `epoll_wait()` reports an event for this `fd`, allowing quick context retrieval.

    ```
    typedef union epoll_data {
        void        *ptr;
        int          fd;
        uint32_t     u32;
        uint64_t     u64;
    } epoll_data_t;
    ```

Internally, each monitored FD within the epoll instance is represented by a `struct epitem`. This structure typically contains the elements shown in Figure **??**:

- **File Descriptor (FD)**: The actual FD being monitored (e.g., `epi->ffd.fd`).
- **Event Types**: The events the user is interested in (e.g., `epi->event.events`).
- **User Data**: The user-defined data from `epoll_event.data` (e.g., `epi->event.data`).
- **Ready List Link**: A pointer (e.g., `epi->rdllink`) to link this `epitem` into the kernel's "ready list" when an event occurs.
- **Red-Black Tree Metadata**: Nodes for linking the `epitem` into a red-black tree for efficient lookup of all monitored FDs.

The "ready list" is a subset of monitored FDs that are currently ready for I/O. Figure 3.3 illustrates adding client sockets to the epoll instance.

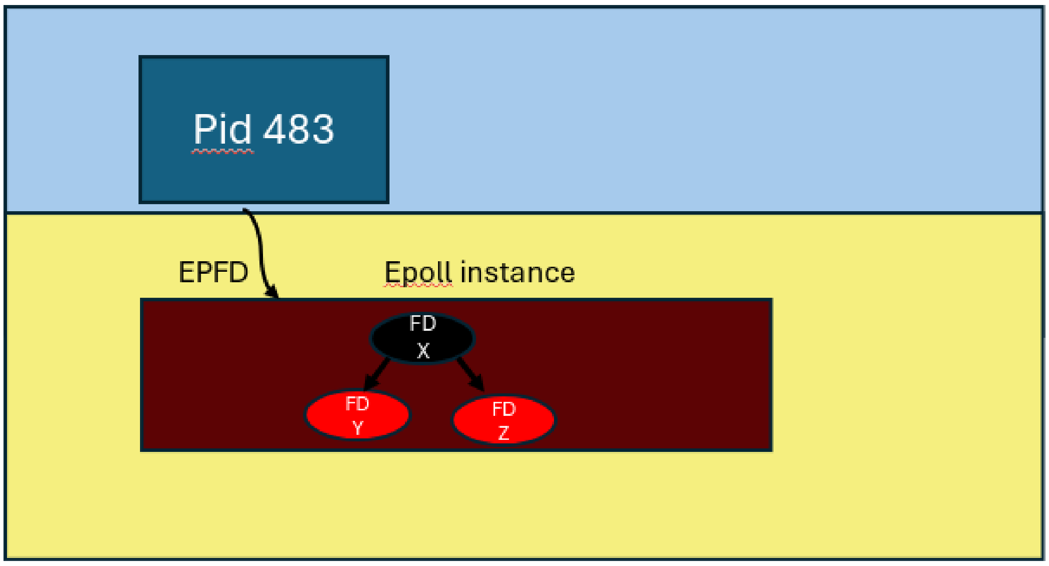


Figure 3.3: `epoll_ctl`: Adding Client Sockets to an `epoll` Instance

3.1.3 `epoll_wait()`

```
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int
timeout);
```

This system call waits for I/O events on the file descriptors associated with the `epoll` instance `epfd`.

- `epfd`: The `epoll` instance file descriptor.
- `events`: A pointer to an array of `struct epoll_event` allocated by the application. The kernel will fill this array with information about FDs that have become ready.
- `maxevents`: The maximum number of events that can be returned in the `events` array. It must be greater than zero.
- `timeout`: Specifies the maximum time `epoll_wait()` will block, in milliseconds.
  - `-1`: Block indefinitely until an event occurs.
  - `0`: Return immediately, even if no events are ready (non-blocking check).
  - `>0`: Wait for the specified number of milliseconds.

`epoll_wait()` returns the number of file descriptors ready for I/O, or 0 if no FD became ready before the timeout expired, or -1 on error (with `errno` set). The `events` array will contain one `struct epoll_event` for each ready FD, up to `maxevents`. The `data` field in each returned structure will be the same data that was set with `epoll_ctl()`. Figure 3.4 shows `epoll_wait()` returning a list of ready FDs from the epoll instance.

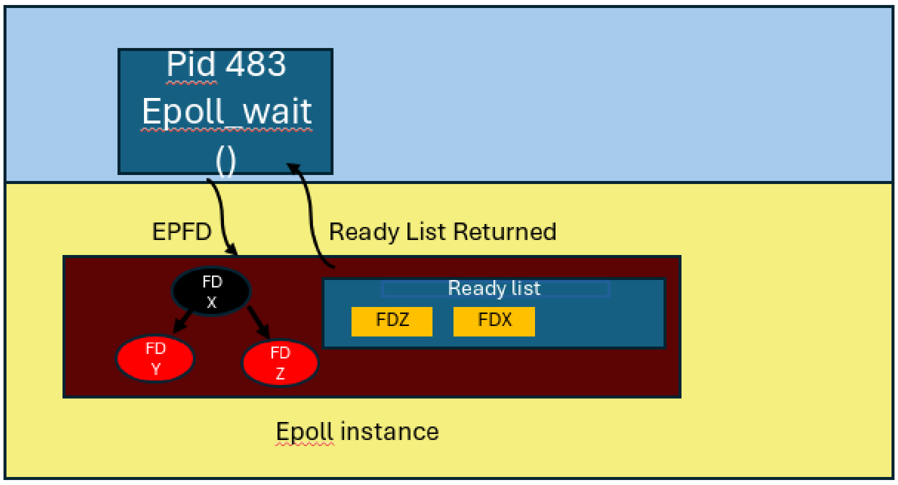


Figure 3.4: `epoll_wait`: Retrieving Ready Events from an `epoll` Instance

3.2 Packet Path and `epoll` Processing

Understanding how a packet's arrival triggers an `epoll` event involves several kernel mechanisms, including the Network Interface Card (NIC), NAPI (New API), socket buffers, and `epoll`'s internal data structures.

3.2.1 Packet Arrival and NAPI

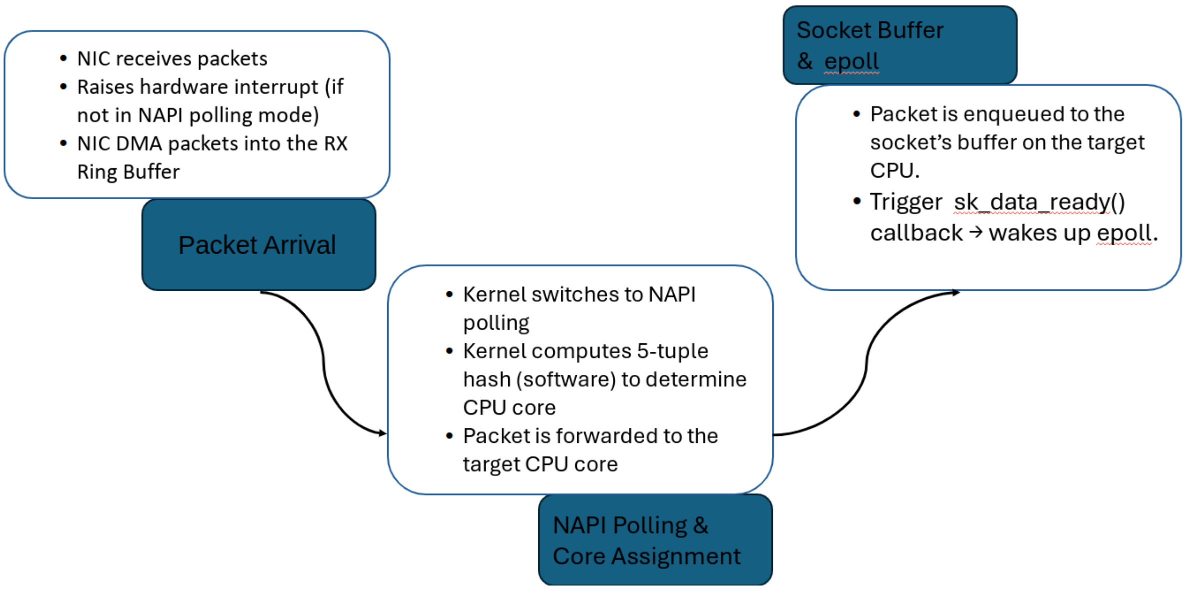
The general flow from packet arrival at the NIC to being ready for `epoll` is summarized in Figure 3.5:

1. **Packet Arrival at NIC:** The NIC receives a packet from the network.
2. **DMA to RX Ring Buffer:** The NIC uses Direct Memory Access (DMA) to transfer the packet data into a pre-allocated kernel memory region called an RX (receive) ring buffer.
3. **Hardware Interrupt (or NAPI):**
   - **Traditional Interrupt Mode:** If not in NAPI polling mode, the NIC raises a hardware interrupt.
   - **NAPI (New API):** To mitigate interrupt overhead, Linux uses NAPI (Figure ??). NAPI is a hybrid interrupt-polling mechanism.
     - **Initial Interrupt:** The first packet arrival might trigger an interrupt.
     - **Switch to Polling:** The interrupt handler disables further interrupts for that RX queue and schedules a NAPI polling routine.
     - **NAPI Poll Loop:** The NAPI poll loop processes packets in bulk from the RX ring buffer.

    - **Exit Polling:** If the poll loop empties the queue or its budget is exhausted, it might re-enable interrupts.

4. **Core Assignment and Packet Forwarding:** Packets are processed and routed to the target socket.
5. **Enqueue to Socket Buffer:** The packet (now typically a `sk_buff`) is enqueued to the socket's receive queue.
6. **Wake up `epoll`:** A callback function, often `sk_data_ready()`, is triggered, eventually adding the socket's FD to `epoll`'s ready list and unblocking `epoll_wait()`.

Figure ?? highlights the benefit of NAPI in reducing interrupt overhead.



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

Figure 3.5: Packet Flow from NIC to Socket Buffer and `epoll` Notification

3.2.2 NAPI: New API for High-Speed Packet Processing

NAPI (New API) is a Linux kernel mechanism designed to efficiently handle high-speed packet reception in network drivers. Traditional interrupt-driven approaches, where the CPU is interrupted for every incoming packet, become inefficient under high traffic loads. NAPI addresses this by introducing a hybrid model that combines both interrupts and polling to reduce CPU overhead while maintaining responsiveness.

Hybrid Interrupt + Polling Model

- **Initial Interrupt:** When the first packet of a burst arrives, the Network Interface Card (NIC) triggers a hardware interrupt to alert the CPU of incoming data.
- **Switch to Polling:** After the initial interrupt, the kernel disables further interrupts for the corresponding RX queue and switches to polling mode.

- **Polling Loop:** The NAPI poll function is invoked by the kernel to fetch packets in bulk from the RX ring buffer. This allows efficient processing of multiple packets in one go, reducing per-packet overhead.
- **Exit Polling:** Once the RX buffer is empty, the driver re-enables interrupts for that RX queue. This ensures that the system remains responsive even when traffic becomes sporadic.

Reducing Interrupt Overhead

Under high traffic conditions, per-packet interrupts can overwhelm the CPU, leading to performance degradation. NAPI helps mitigate this by converting interrupt storms into efficient polling loops. For example:

- **Without NAPI:** Receiving 10,000 packets results in 10,000 interrupts.
- **With NAPI:** The same 10,000 packets may trigger only one interrupt followed by a single polling loop to process the bulk of the packets.

This batching effect significantly improves throughput and reduces CPU load, especially in high-bandwidth environments.

Packet Flow Summary

The following summarizes the packet flow in a system using NAPI:

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

Here, RSS (Receive Side Scaling) helps distribute packets across multiple RX queues. Each queue is associated with a specific NAPI context (ID) and is processed by an assigned CPU core, improving parallelism and efficiency.

3.2.3 `epoll` Callback and Ready List Management

When data arrives for a socket monitored by `epoll`, `sk_data_ready()` invokes `ep_poll_callback()`. This kernel function adds the ready FD to `epoll`'s internal lists:

- `rdllist` **(Ready Doubly Linked List):** Primary list for newly ready FDs.
- `ovflist` **(Overflow List):** Temporary list used when `ep_send_events()` is processing `rdllist`.

The logic within `ep_poll_callback()` involves checking if `ovflist` is active. If so, the new event is added to `ovflist` using `chain_epi_lockless()`. Otherwise, it's added to `rdllist` using `list_add_tail_lockless()`. Both use atomic operations. Finally, it wakes up any userspace process blocked in `epoll_wait()`. This decision logic is illustrated in Figure 3.6. A timeline example of list usage is in Figure 3.10.

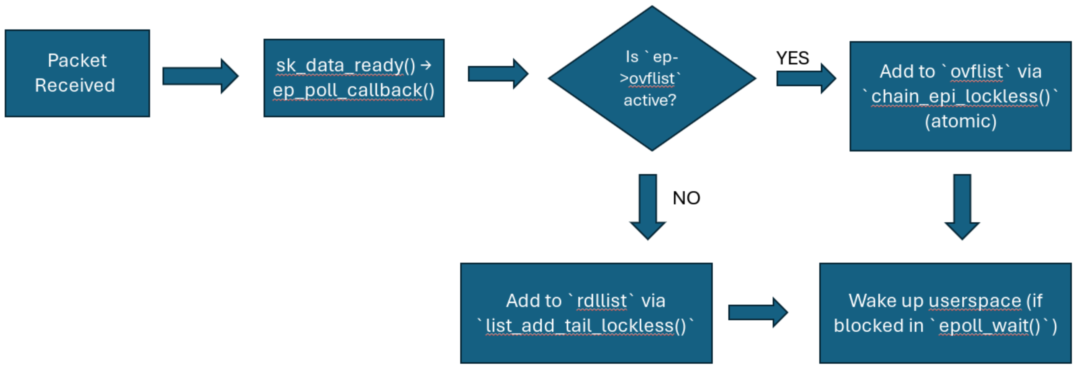

Packet Received
sk_data_ready() → ep_poll_callback()
Is `ep->ovflist` active?
YES
Add to `ovflist` via `chain_epi_lockless()` (atomic)
NO
Add to `rdllist` via `list_add_tail_lockless()`
Wake up userspace (if blocked in `epoll_wait()`)

Figure 3.6: Flow for Adding Events to `epoll`'s Ready Lists after `sk_data_ready()`

```c
static int ep_poll_callback(wait_queue_entry_t *wait, ...) {
    struct epitem *epi = container_of(wait, struct epitem,
  wqe);
    struct eventpoll *ep = epi->ep;

    if (READ_ONCE(ep->ovflist) != EP_UNACTIVE_PTR) { //
  EP_UNACTIVE_PTR is conceptual
        // Use lockless atomic ops to add to ovflist
        if (chain_epi_lockless(epi, ep)) // Pass ep for
  ovflist
            return 1; // Already on a list or added to
  ovflist
    } else {
        // Add to rdllist (lockless)
        list_add_tail_lockless(&epi->rdllink, &ep->rdllist);
            // simplified for brevity, actual kernel code
  handles if already on list
    }

    // Wake up userspace blocked in epoll_wait()
    if (waitqueue_active(&ep->wq))
        wake_up_locked(&ep->wq); // Or wake_up() depending on
   context
    return 1;
}
```

Listing 3.1: Conceptual `ep_poll_callback()` logic

Figure 3.7: Conceptual Code: `ep_poll_callback()`

```c
static inline bool list_add_tail_lockless(struct list_head *
   new_entry,
                                              struct list_head *
   head) {
    struct list_head *prev;
    // Simplified conceptual logic
    // Check if already linked (new_entry should point to
   itself if not linked)
    if (new_entry->next != new_entry)
        return false; // Already on a list or invalid state

    // Atomically add to the tail
    do {
        prev = READ_ONCE(head->prev); // Get current tail
        new_entry->prev = prev;
        new_entry->next = head; // Point to head for circular
    list
    } while (cmpxchg(&head->prev, prev, new_entry) != prev);
   // Atomically set head->prev = new_entry

   // if head->prev is still 'prev'
    // Atomically set old tail's next pointer
    WRITE_ONCE(prev->next, new_entry); // Old tail now points
    to new_entry

    return true;
}
```

Listing 3.2: Conceptual `list_add_tail_lockless()`

Figure 3.8: Conceptual Code: `list_add_tail_lockless()`

```c
static inline bool chain_epi_lockless(struct epitem *epi,
   struct eventpoll *ep) {
    struct epitem *old_head;
    // Simplified conceptual logic
    // Check if epi is already in ovflist or some other
   active state
    if (READ_ONCE(epi->next) != EP_UNACTIVE_PTR) //
   EP_UNACTIVE_PTR is conceptual
        return false;

    // Atomic compare-and-swap: Try to mark epi->next from
   unactive to "linking" (e.g., NULL)
    if (cmpxchg(&epi->next, EP_UNACTIVE_PTR, NULL) !=
   EP_UNACTIVE_PTR)
        return false; // Another thread is handling this epi,
    or it's not unactive

    // Atomically prepend to ovflist (which is a LIFO stack)
    do {
        old_head = READ_ONCE(ep->ovflist);
        epi->next = old_head; // epi will point to the
    current head of ovflist
    } while (cmpxchg(&ep->ovflist, old_head, epi) != old_head
    ); // Atomically set ep->ovflist to epi

    return true;
}
```
Listing 3.3: Conceptual `chain_epi_lockless()`

Figure 3.9: Conceptual Code: `chain_epi_lockless()`

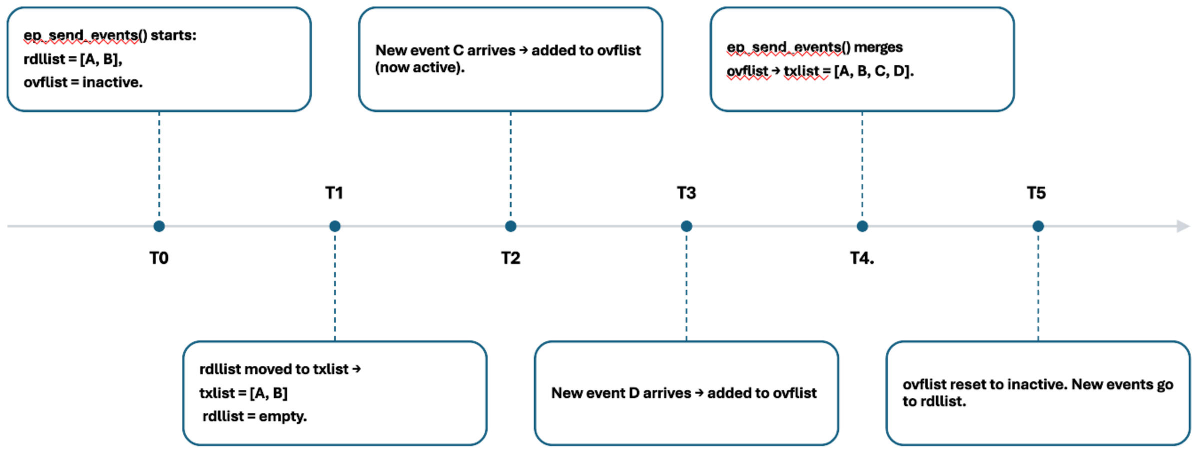

ep_send_events() starts:
rdllist = [A, B],
ovflist = inactive.
New event C arrives → added to ovflist (now active).
ep_send_events() merges ovflist → txlist = [A, B, C, D].
T0
T1
T2
T3
T4.
T5
rdllist moved to txlist → txlist = [A, B] rdllist = empty.
New event D arrives → added to ovflist
ovflist reset to inactive. New events go to rdllist.

Figure 3.10: Timeline of `rdllist` and `ovflist` usage during `ep_send_events()`

3.2.4 Userspace Call: `epoll_wait()` and `ep_send_events()`

When userspace calls `epoll_wait()`, the kernel's `ep_send_events()` function transfers ready events. Its logic is (conceptually):

1. **Lock:** Acquires a mutex (`ep->mtx`).
2. **Atomically Move Lists (`ep_start_scan`):** Calls `ep_start_scan()` to merge `rdllist` and `ovflist` into a temporary `txlist`.
3. **Iterate and Copy:** Iterates `txlist`. For each event:
    - **Re-check Readiness (`ep_item_poll`):** Verifies the event is still active.
    - **Copy to Userspace (`copy_to_user`):** Copies event data to the user's buffer.
4. **Release Lock:** Releases `ep->mtx`.

Conceptual code for these operations is shown in Figure 3.11 and Figure 3.12.

```c
static int ep_send_events(struct eventpoll *ep, struct epoll_event __user *events, int maxevents) {
    struct list_head txlist; // Temporary list
    int num_events_sent = 0;
    struct epitem *epi, *tmp;

    mutex_lock(&ep->mtx);
    ep_start_scan(ep, &txlist); // txlist now contains all ready events

    list_for_each_entry_safe(epi, tmp, &txlist, rdllink) {
        if (num_events_sent >= maxevents)
            break; // User buffer full

        // Re-check if the event is still ready
        if (ep_item_poll(epi, &epi->event.events) /* returns true if ready, simplified */) {
            // Copy epi->event to userspace
            if (copy_to_user(&events[num_events_sent], &epi->event, sizeof(struct epoll_event))) {
                mutex_unlock(&ep->mtx);
                return -EFAULT; // Error copying
            }
            num_events_sent++;
            // If Edge Triggered (EPOLLET), specific handling is needed here
            // if (epi->event.events & EPOLLET) { ... }
        }
    }
    // Unprocessed events in txlist might be re-added to ep->rdllist if LT & still active
    mutex_unlock(&ep->mtx);
    return num_events_sent;
}
```

Listing 3.4: Conceptual ep_send_events()

Figure 3.11: Conceptual Code: ep_send_events()

```c
static void ep_start_scan(struct eventpoll *ep, struct list_head *txlist) {
    struct epitem *epi, *current_ovflist_head;
    INIT_LIST_HEAD(txlist);

    // 1. Move rdllist to txlist. rdllist -> txlist (now empty)
    list_splice_init(&ep->rdllist, txlist);

    // 2. Atomically move ovflist to txlist
    // Mark ovflist as inactive so ep_poll_callback starts using rdllist again.
    current_ovflist_head = xchg(&ep->ovflist, (struct epitem *)EP_UNACTIVE_PTR); // EP_UNACTIVE_PTR is conceptual
    smp_mb(); // Memory barrier

    // Drain the captured ovflist (current_ovflist_head) into txlist
    epi = current_ovflist_head;
    while (epi && epi != (struct epitem *)EP_UNACTIVE_PTR) { // Check against marker too
        struct epitem *next_epi = epi->next; // Assuming epi->next was used for ovflist linking
        // Add epi to the tail of txlist
        list_add_tail(&epi->rdllink, txlist); // Using rdllink for txlist as well
        epi = next_epi;
    }
}
```

Listing 3.5: Conceptual `ep_start_scan()`

Figure 3.12: Conceptual Code: `ep_start_scan()`

Chapter 4

Experiment 1: Fairness with Homogeneous TCP Connections

The first experiment aimed to evaluate the fairness of `epoll` when handling multiple, identical TCP connections sending data in a balanced manner.

4.1 Setup

The setup involved:

- **Server:** Listened for connections, registered listening FD and accepted client FDs to an `epoll` instance, monitoring for incoming data (`EPOLLIN`).
- **Client:** Established two TCP connections (Socket 1, Socket 2) and sent data packets in a strict round-robin fashion.

The core question was whether `epoll` shows bias when both connections are equally active. Results were averaged over 20 runs.

4.2 Results and Analysis

4.2.1 Varying Total Data (60B Packets)

Figure 4.1 shows the ratio of times Socket 1's events were processed before Socket 2's, and vice-versa, for different total data volumes using 60-byte packets. Ratios consistently hovered near 0.5 (50%), suggesting fairness. Figure 4.2 provides a stacked bar chart of event scenarios, where "Socket 1 before Socket 2" and "Socket 2 before Socket 1" are dominant and roughly equal.

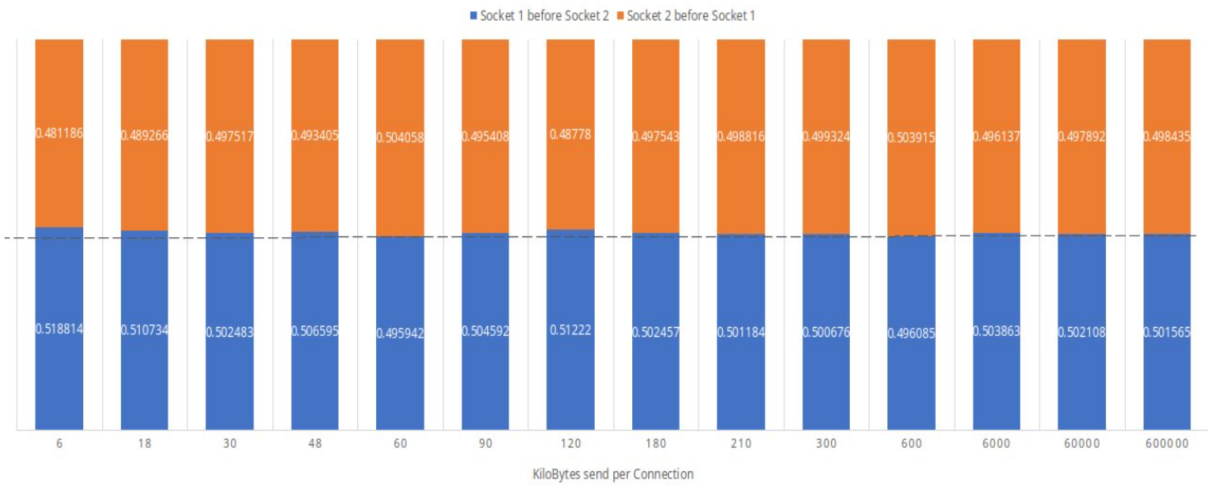


Figure 4.1: Experiment 1: Processing Order Ratio for 60B Packets (Varying Total KB)

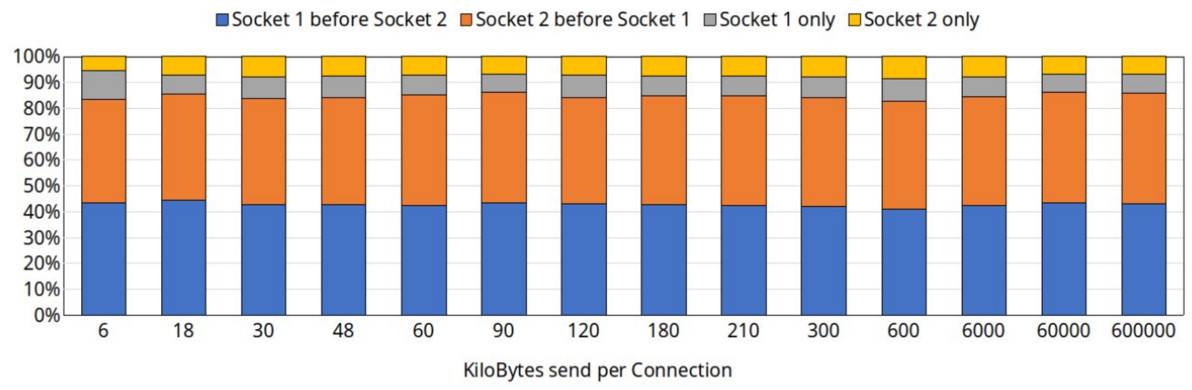


Figure 4.2: Experiment 1: Stacked Event Case Ratios for 60B Packets (Varying Total KB)

4.2.2 Varying Time Intervals (1 Million 240B Packets)

Sending 1 million 240-byte packets over different time intervals, Figure 4.3 shows processing order ratios remained close to 50/50. Figure 4.4 shows total bytes received on Connection 1 versus Connection 2 were almost identical, indicating fairness.

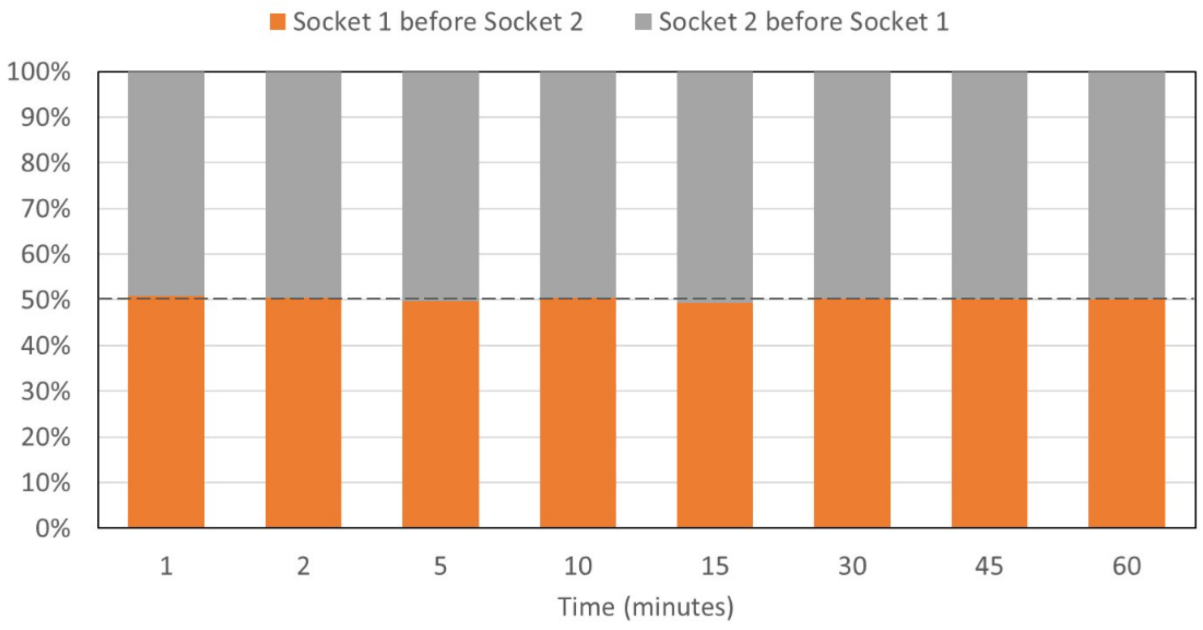


Figure 4.3: Experiment 1: Processing Order Ratio for 1M x 240B Packets (Varying Time)

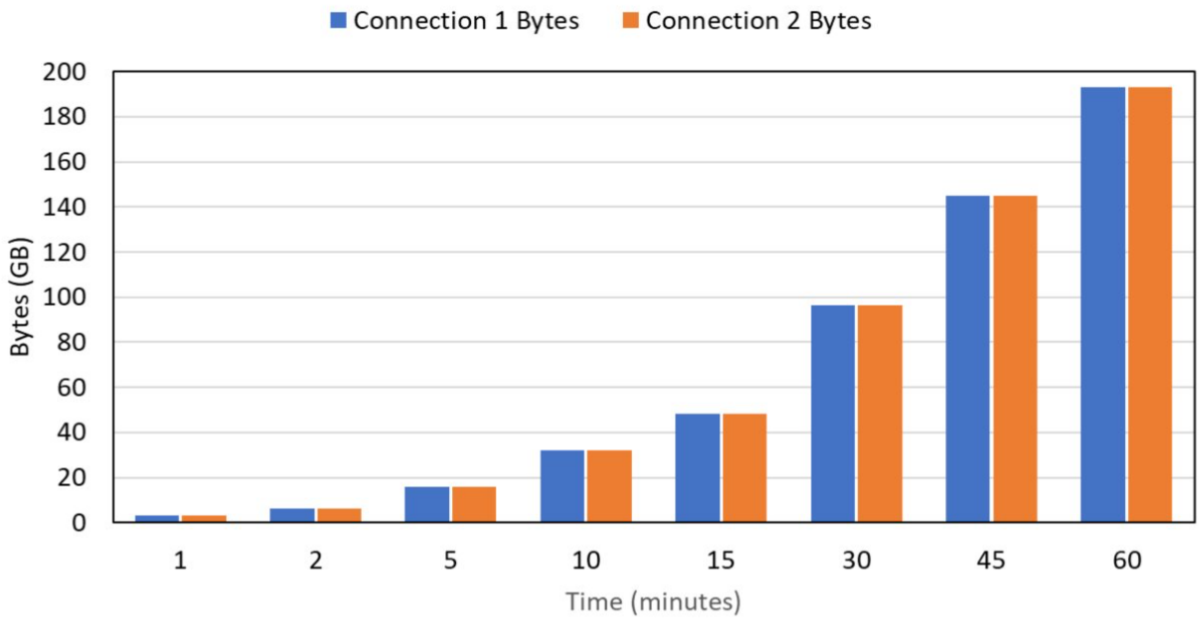


Figure 4.4: Experiment 1: Bytes Processed per Connection for 1M x 240B Packets (Varying Time)

4.2.3 Varying Per-Packet Size (1 Hour Duration)

For a 1-hour duration with varying packet sizes, Figure 4.5 shows processing order ratios near 0.5. Figure 4.6 presents stacked event ratios, showing balance when both sockets are active. Figure 4.7 shows nearly identical total bytes processed per connection.

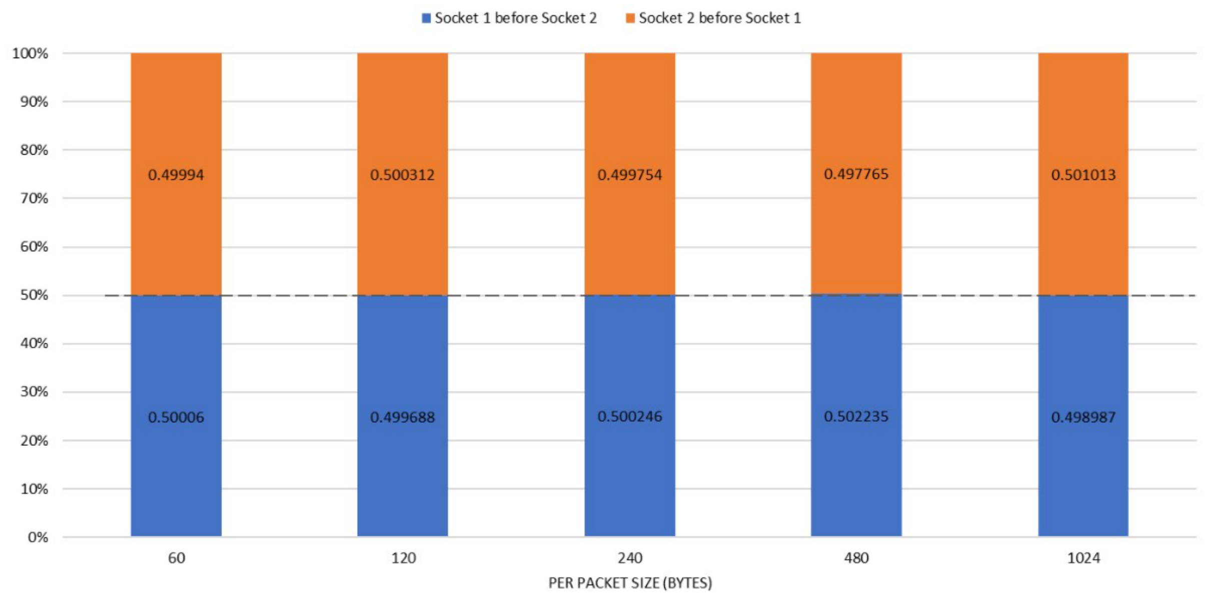


Figure 4.5: Experiment 1: Processing Order Ratio for 1 Hour (Varying Packet Sizes)

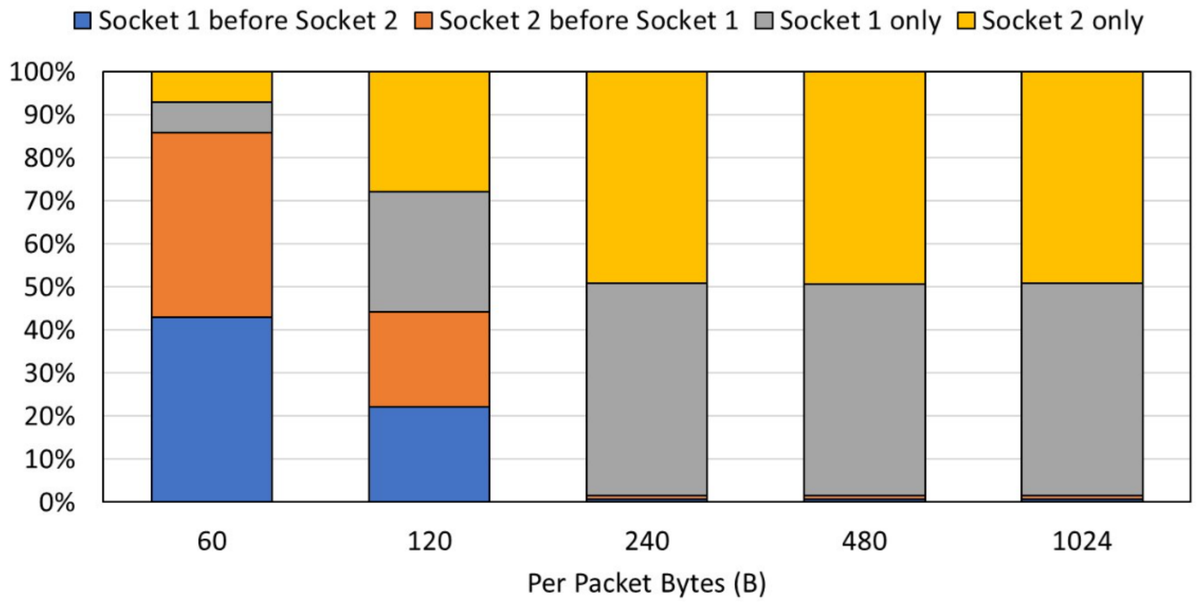


Figure 4.6: Experiment 1: Stacked Event Case Ratios for 1 Hour (Varying Packet Sizes)

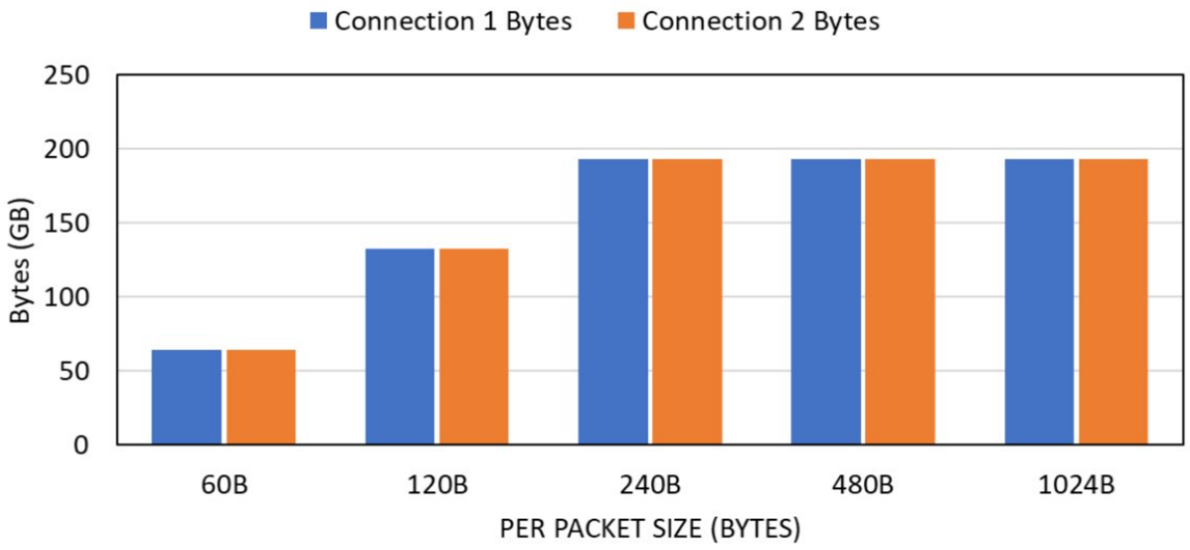


Figure 4.7: Experiment 1: Bytes Processed per Connection for 1 Hour (Varying Packet Sizes)

4.2.4 CDF of Socket Processing Ratios (240B Packets, 1 Hour, 45 Runs)

Figure 4.8 shows the Cumulative Distribution Function (CDF) of processing ratios. Both CDFs are similar and clustered around 0.5, indicating even splitting in most runs.

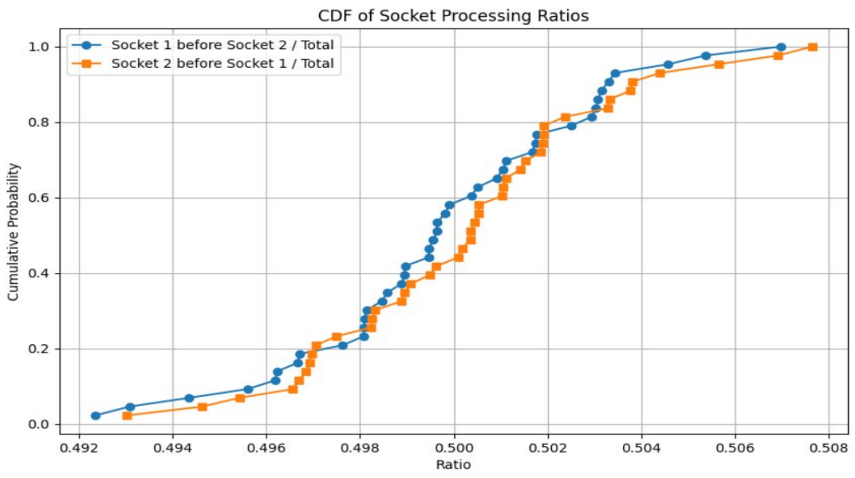


Figure 4.8: Experiment 1: CDF of Socket Processing Ratios (240B Packets, 1 Hour, 45 Runs)

4.2.5 Summary of Experiment 1

Results consistently demonstrate fair treatment by `epoll` for two identical, balanced TCP connections across various conditions.

Chapter 5

Experiment 2: Fairness with Heterogeneous Connections

This experiment investigated `epoll` fairness with persistent versus non-persistent connections.

5.1 Setup

The setup was:

- **Title:** Analyzing Bias in `epoll` for Persistent vs. Non-Persistent Connections.
- **Server:** Used `epoll` for two clients on `10.130.150.212:8080`, monitoring for `EPOLLIN`.
- **Client 1 (Persistent):** Single, long-lived TCP connection.
- **Client 2 (Non-Persistent):** New TCP connection per request (`connect()`, send, `close()`), causing frequent `epoll_ctl()` operations.
- **Goal:** Check for bias between these connection types.

5.2 Results and Analysis

5.2.1 Ratio of Packets Sent (Persistent vs. Non-Persistent)

Figure 5.1 shows the average ratio of packets processed from persistent vs. non-persistent connections. Ratios hover around 0.5, indicating roughly equal opportunity. Figure 5.2 breaks this down, showing balanced shares when both types are active.

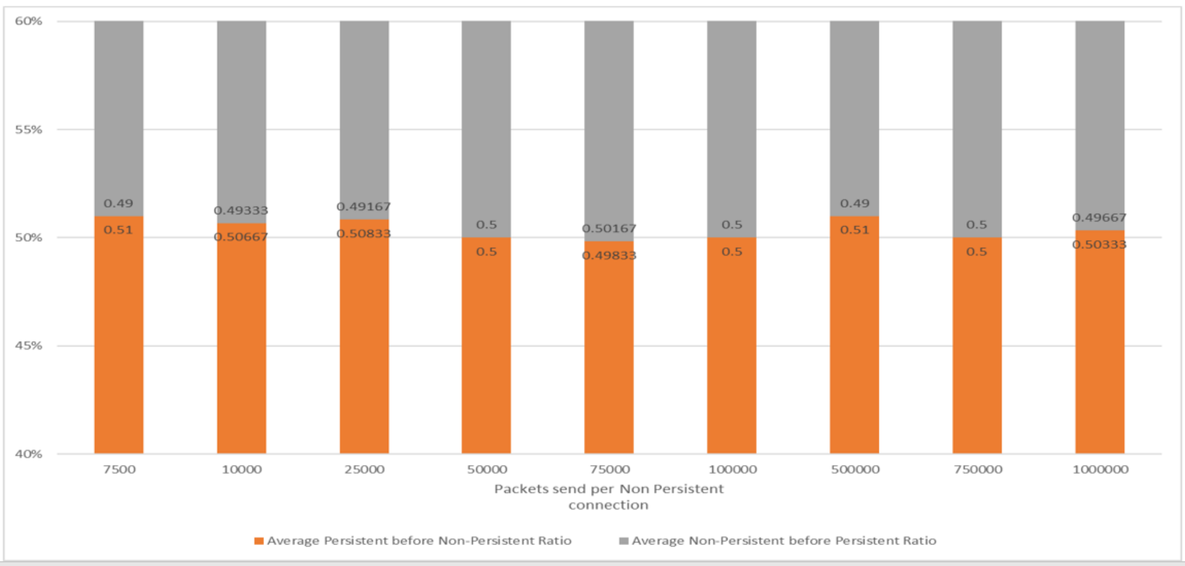


Figure 5.1: Experiment 2: Average Packet Processing Ratio (Persistent vs. Non-Persistent)

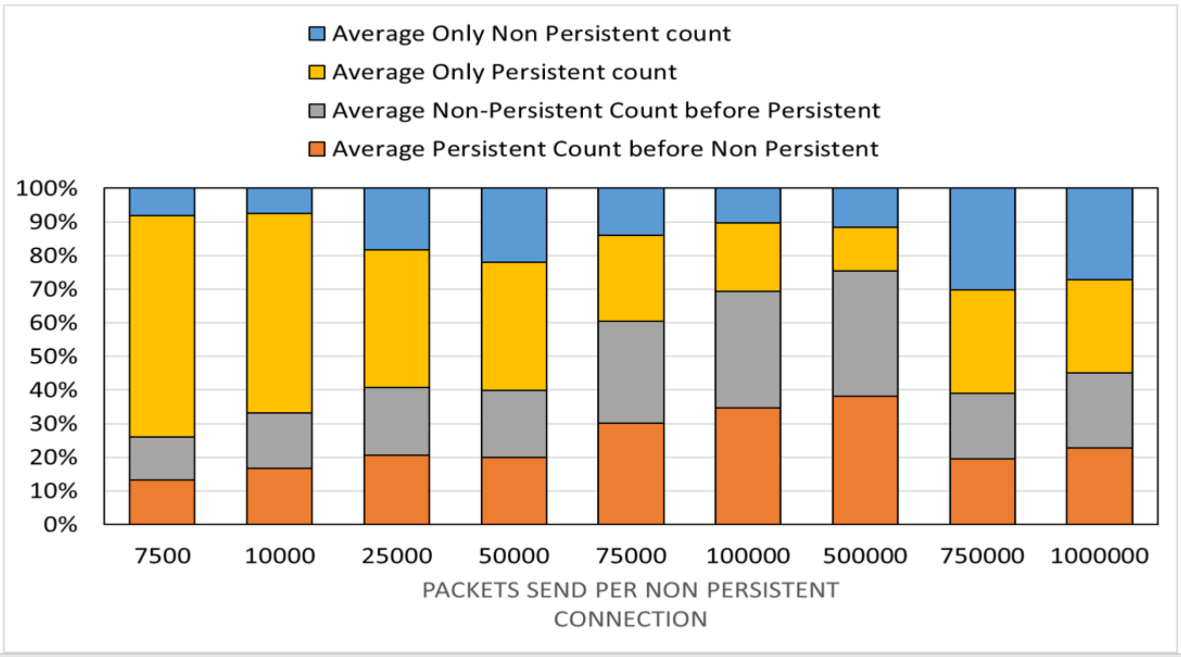


Figure 5.2: Experiment 2: Event Case Ratios vs. Packets per Non-Persistent Connection

5.2.2 Bytes Sent (Persistent vs. Non-Persistent)

Figure 5.3 plots average bytes processed for persistent vs. non-persistent connections.

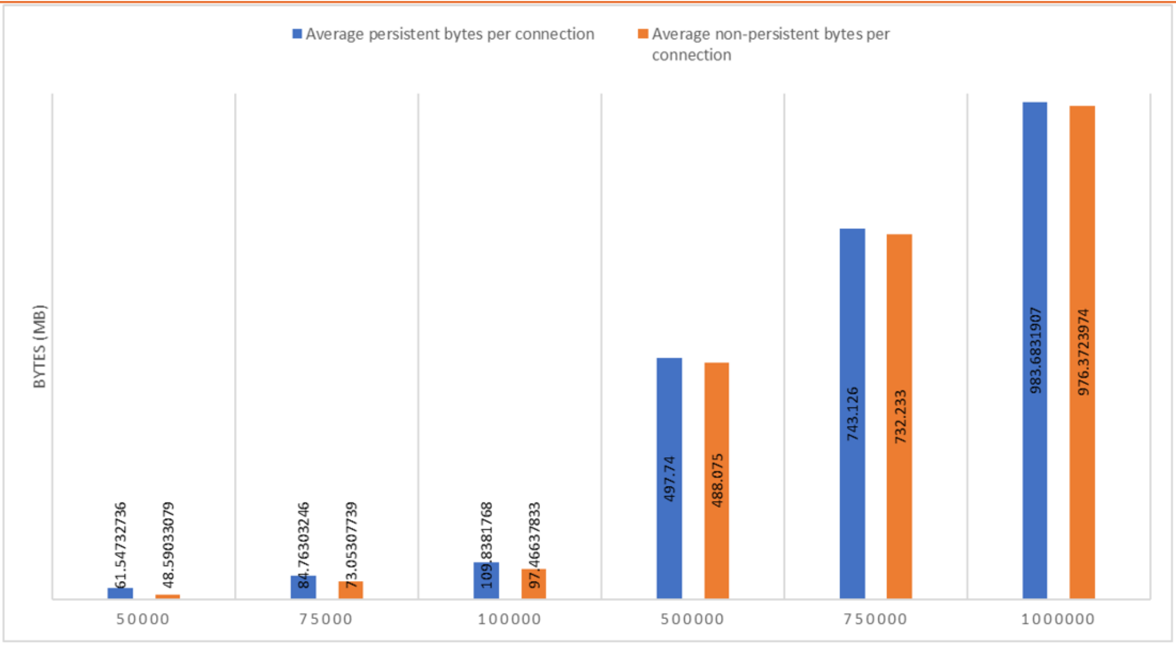


Figure 5.3: Experiment 2: Bytes Sent vs. Packets per Non-Persistent Connection

5.2.3 CDF of Persistent vs. Non-Persistent Ratios

Figure 5.4 shows the CDF for processing ratios (100k packets per connection type). CDF steps are centered near 0.5, suggesting proportional fairness.

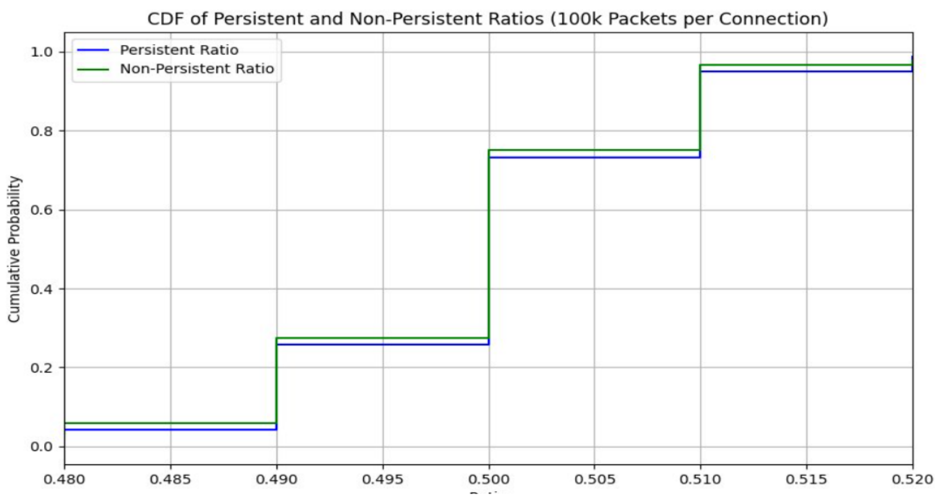


Figure 5.4: Experiment 2: CDF of Persistent and Non-Persistent Processing Ratios

5.2.4 Summary of Experiment 2

Experiment 2 demonstrates that `epoll` generally maintains fairness between stable persistent connections and frequently changing non-persistent connections.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This report investigated the scheduling fairness of the Linux `epoll` I/O multiplexing mechanism. Through detailed examination and empirical experiments, we assessed if `epoll` exhibits inherent bias.

Experiment 1 (homogeneous TCP connections) consistently showed that `epoll` divided processing opportunities and data throughput almost equally, with ratios close to an ideal 50/50 split.

Experiment 2 (persistent vs. non-persistent connections) also indicated a similar degree of fairness, even with the overhead of frequent connection setup/teardown and `epoll_ctl` operations for non-persistent clients.

Collectively, the evidence suggests that `epoll` itself is fair in making file descriptors available via `epoll_wait()` under the tested conditions. This is vital for applications requiring predictable performance and fairness.

6.2 Future Work

Further investigations could provide deeper insights:

- **Kernel-Level Tracing:** Use tools like `ftrace` or eBPF to observe the order `epitem` structures are added to `epoll`'s ready lists (`rdllist`, `ovflist`) and processed by `ep_send_events()`.
- **Larger Number of Connections:** Test with C10k or C100k connections.
- **Varying Workloads and Priorities:** Introduce connections with different data rates or investigate CPU-bound vs. I/O-bound tasks after events. Explore `SO_PRIORITY`.
- **Edge-Triggered (ET) Mode Analysis:** Analyze fairness in ET mode.
- **Impact of `epoll` Flags:** Test flags like `EPOLLONESHOT`.

These avenues can lead to a more comprehensive understanding of `epoll`'s behavior.

Bibliography

[1] ESnet. *Network Tuning*, Available at: `https://fasterdata.es.net/network-tuning/`

[2] Packagecloud Blog. *Monitoring and Tuning the Linux Networking Stack: Receiving Data*, Available at: `https://blog.packagecloud.io/monitoring-tuning-linux-networking-stack-receiving-data/`

[3] Copyconstruct. *The Method to Epoll's Madness*, Available at: `https://copyconstruct.medium.com/the-method-to-epolls-madness-d9d2d6378642`

[4] man7.org. *epoll_create(2) - Linux manual page*, Available at: `https://man7.org/linux/man-pages/man2/epoll_create.2.html`

[5] man7.org. *epoll_ctl(2) - Linux manual page*, Available at: `https://man7.org/linux/man-pages/man2/epoll_ctl.2.html`

[6] Torvalds, L. *eventpoll.c - Linux Kernel Source Code*, GitHub. Available at: `https://github.com/torvalds/linux/blob/master/fs/eventpoll.c`