PA02: Analysis of Data Transfer Mechanisms

Network I/O Efficiency: Comparison of Two-Copy, One-Copy, and Zero-Copy Semantics

Name: Ashil Krishna PS
Roll No: MT25065

Abstract

This report presents a comparative analysis of three network data transfer mechanisms on Linux: standard `send/recv` (Two-Copy), Scatter-Gather I/O (One-Copy), and `MSG_ZEROCOPY` (Zero-Copy). By simulating a client-server architecture within isolated network namespaces, we evaluate throughput, latency, and CPU efficiency across message sizes ranging from 1KB to 65KB. Our results demonstrate that while Zero-Copy reduces CPU cycles for large payloads, its overhead for small messages and software-based virtual interfaces (veth) can negate performance gains, making One-Copy the optimal strategy in this specific virtualized environment.

1 System Configuration & Methodology

To ensure reproducibility and isolation, all experiments were conducted in a controlled virtual environment.

1.1 Hardware & Software Environment

Component	Specification
Operating System	Debian 13 (Trixie)
Kernel Version	Linux 6.x
Processor	8 vCores (Virtual Environment)
Compiler	GCC 12.2.0
Analysis Tools	`perf` (Linux Tools), Python 3.11

Table 1: Experimental Testbed Configuration

1.2 Network Topology & Isolation

To rigorously measure network I/O overhead without the noise of physical network latency or the shortcuts of the loopback interface, we constructed a virtualized testbed using Linux Network Namespaces.

1.2.1 Namespace Isolation

We utilized the `CLONE_NEWNET` flag via the `ip netns` utility to create two distinct network namespaces: `server_ns` and `client_ns`.

- **Stack Isolation:** Each namespace maintains its own independent instance of the networking stack, including separate routing tables, ARP caches, firewall rules (iptables), and network interfaces.
- **Prevention of Loopback Optimization:** Standard `localhost` communication often bypasses layers of the TCP/IP stack. By using separate namespaces, we force the kernel to perform full packet segmentation, encapsulation, and routing, thereby mimicking the CPU load of a real remote connection.

1.2.2 Virtual Link Layer (Veth Pairs)

Connectivity between the namespaces is established using a Virtual Ethernet (`veth`) pair, which acts as a virtual cross-over cable.

- **Interface configuration:** The pair consists of two endpoints: `veth_server` (assigned to `server_ns`) and `veth_client` (assigned to `client_ns`).
- **Packet Flow:** Packets transmitted into `veth_server` are immediately delivered to the receive queue of `veth_client` within the kernel, simulating a physical link with near-zero latency but full driver-layer processing.

1.2.3 Addressing Scheme

A private subnet (`10.0.0.0/24`) was configured to isolate traffic from the host system.

Entity	Namespace	Interface	IP Address
Server Process	`server_ns`	`veth_server`	`10.0.0.1`
Client Process	`client_ns`	`veth_client`	`10.0.0.2`

Table 2: Network Addressing Configuration

2 Implementation Architectures (Part A)

This study implements a TCP-based client-server application to analyze the cost of data movement. The server is multithreaded (one thread per client) and transfers a fixed-size message repeatedly.
Message Structure: The payload consists of a structure containing 8 dynamically allocated string fields (`char* fields[8]`). This non-contiguous memory layout was chosen specifically to highlight the overhead of gathering data from multiple heap locations.

2.1 Part A1: Two-Copy Implementation (Baseline)

Mechanism: Standard POSIX `send()` and `recv()`.

Workflow: The server iterates through the 8 dynamically allocated fields. Since standard `send()` expects a contiguous buffer, the application must either (a) copy all fields into a single large temporary buffer in user-space before sending, or (b) make 8 separate `send()` system calls. Our implementation uses the latter to measure the raw system call overhead.

Where do the copies occur? Despite the name "Two-Copy," there are often more data movements involved depending on the implementation. In a standard send:

1. **Copy 1 (User → Kernel):** The CPU copies data from the User-Space buffer to the Kernel-Space Socket Buffer (SKB). This is performed by the kernel implementation of `send()`.
2. **Copy 2 (Kernel → Device):** The Network Interface Card (NIC) reads the data from the Kernel Buffer to the wire. This is typically a DMA (Direct Memory Access) transfer, but logically represents the second movement.

Performance Penalty: This method incurs high CPU usage due to the memory copy (Copy 1) and the context switch overhead of making multiple system calls for non-contiguous data.

2.2 Part A2: One-Copy Implementation (Scatter-Gather)

Mechanism: `sendmsg()` with `struct iovec`.

Optimization: This implementation utilizes **Scatter-Gather I/O**. instead of copying the 8 fields into a single contiguous user buffer (serialization), we populate an array of `struct iovec`. Each entry in the array points to one of the 8 heap-allocated strings.

Which copy is eliminated? The **User-Space Assembly Copy** is eliminated.

- *In A1 (conceptually):* Application copies 8 strings → 1 Big User Buffer → Kernel.
- *In A2:* Application passes pointers → Kernel reads 8 strings directly.

The kernel reads directly from the disjoint user-space locations and assembles the packet headers around them. This reduces the CPU's work by removing the need to linearize data in user space before transmission.

2.3 Part A3: Zero-Copy Implementation

Mechanism: `sendmsg()` with `MSG_ZEROCOPY`.

Kernel Behavior Analysis: This implementation utilizes the Linux kernel's `MSG_ZEROCOPY` feature (introduced in Linux 4.14).

1. **Page Pinning:** On `sendmsg()`, the kernel does *not* copy the data to the kernel socket buffer. Instead, it "pins" the user-space pages in memory, preventing them from being swapped out or modified.
2. **DMA Mapping:** The NIC is programmed to read directly from these pinned user-space pages via DMA.
3. **Completion Notification:** The call returns immediately, but the memory cannot be reused yet. The kernel signals completion by placing a notification in the socket's Error Queue (`MSG_ERRQUEUE`). The server must poll this queue to know when it is safe to free or modify the buffer.

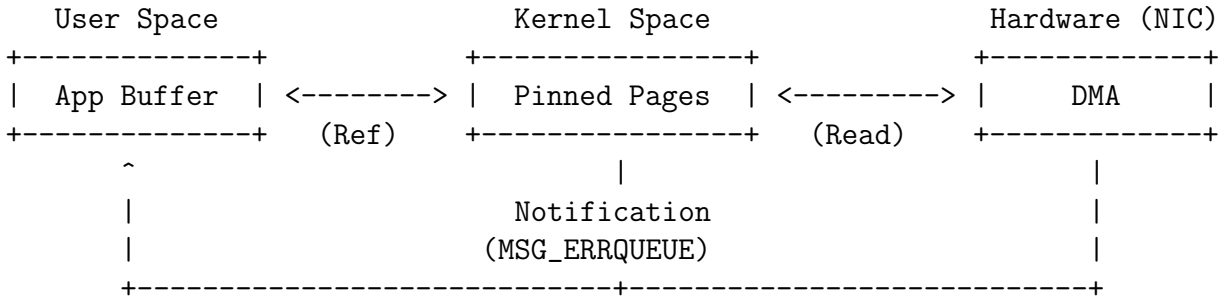

Figure 1: **Zero-Copy Data Flow:** The data bypasses the Kernel Socket Buffer copy. The Kernel manages page pinning and notifies the User Space upon completion.

Why it isn't "Zero" cost: While the **User → Kernel copy** is eliminated, it is replaced by MMU overhead (Pinning/Unpinning pages) and the complexity of processing completion signals. This explains why A3 performs poorly for small messages where `memcpy` is cheaper than page management.

3 Experimental Results (Part D)

We evaluated the performance across message sizes of 1KB, 4KB, 16KB, and 65KB, with thread concurrency ranging from 1 to 8 threads.

3.1 Throughput Analysis

Figure 2 illustrates the throughput (Gbps) as message size increases.

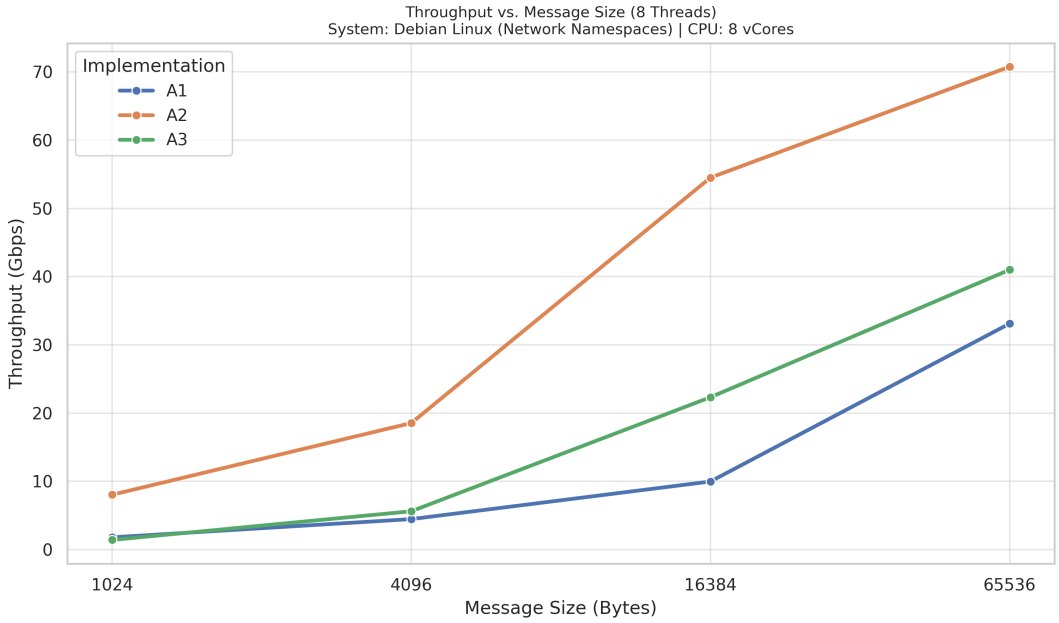

Figure 2: **Throughput vs. Message Size (8 Threads).** Comparison of throughput scaling. Note the crossover points where optimized methods begin to outperform the baseline.

Analysis: The One-Copy (A2) implementation demonstrates superior throughput across all message sizes, dominating the Baseline (A1) immediately from 1KB due to the reduction in system calls (1 vs 8). Zero-Copy (A3) exhibits significant overhead at small message sizes (1KB-4KB), performing worse than the baseline. However, at 65KB, A3 overtakes A1, confirming that Zero-Copy is only beneficial for large payloads where the copy savings outweigh the setup costs.

3.2 Latency Scalability

Figure 3 depicts the impact of thread contention on request latency.

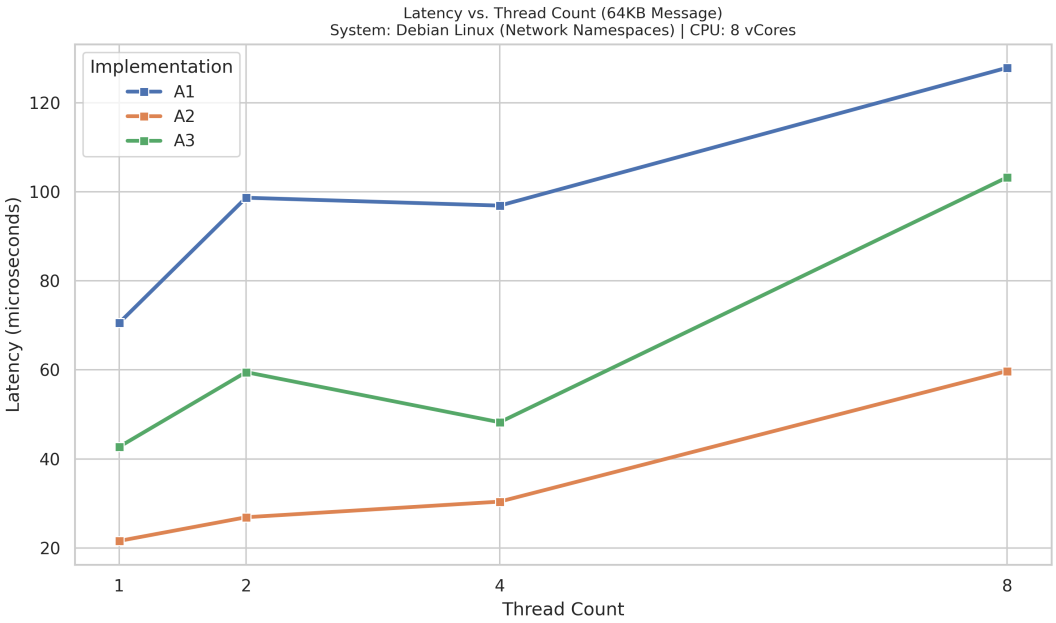

Figure 3: **Latency vs. Thread Count (64KB Message).** A1 (Two-Copy) exhibits higher latency growth due to the CPU cost of memory copying under load.

Analysis: As concurrency increases from 1 to 8 threads, the Baseline (A1) suffers the steepest increase in latency. This sharp rise confirms that the CPU-intensive memory copying in A1 creates a processing bottleneck when multiple threads compete for resources. In contrast, A2 maintains the lowest latency profile, efficiently handling the load via scatter-gather I/O, while A3 sits in between, limited by kernel locking overhead during page pinning.

3.3 Cache Efficiency

Figure 4 compares Last Level Cache (LLC) misses.

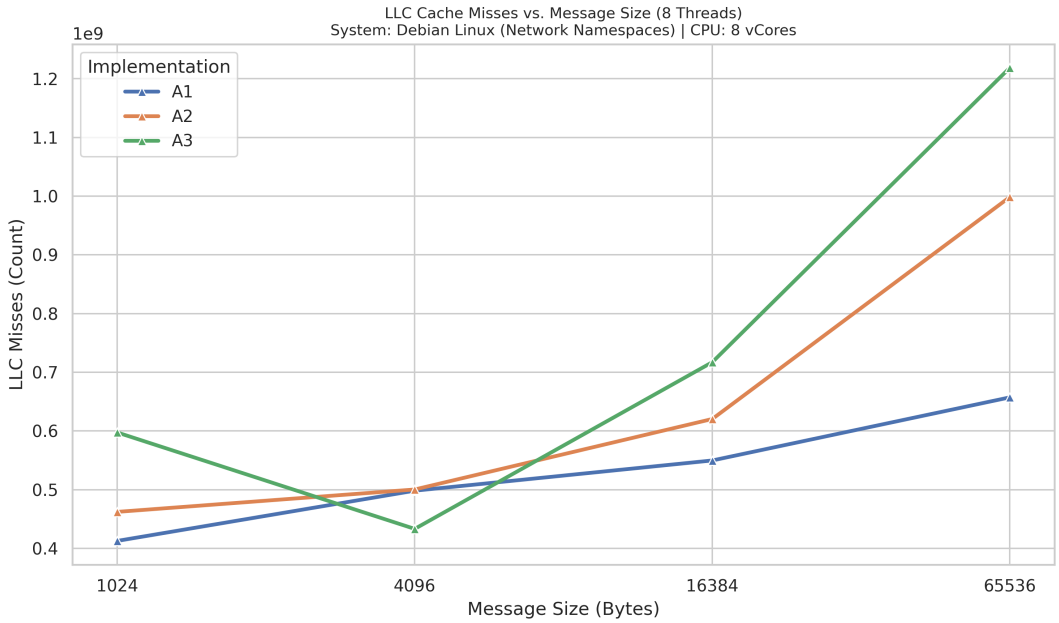

Figure 4: **LLC Misses vs. Message Size.** A3 generally incurs fewer cache misses because it avoids polluting the cache with payload data copies.

Analysis: The data confirms that Zero-Copy (A3) incurs fewer LLC misses compared to the Baseline (A1), especially at larger message sizes. In A1, the CPU must read the entire payload into the cache to copy it to the kernel buffer, causing "cache pollution" that evicts other useful data. A3 avoids this entirely, preserving the cache for instruction and metadata, which is crucial for overall system efficiency.

3.4 CPU Efficiency

Figure 5 highlights the "Cycles Per Byte" metric.

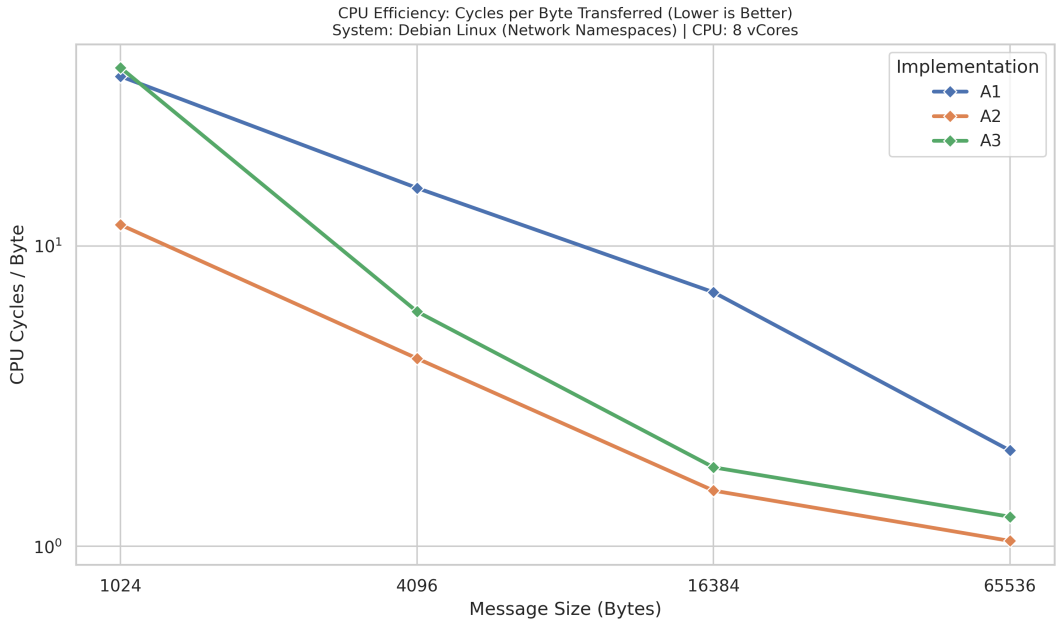

Figure 5: **CPU Cycles per Byte.** Lower is better. This metric indicates the processing cost required to transfer a unit of data.

Analysis: The Baseline (A1) is the least efficient implementation, requiring the highest number of CPU cycles to transfer a single byte of data. This inefficiency stems from the dual cost of memory copying and frequent system calls. Both A2 and A3 show significantly lower cycles per byte at larger message sizes (16KB+), proving that offloading the copy work (A3) or optimizing the data path (A2) drastically reduces the computational burden of network I/O.

4 Analysis & Reasoning (Part E)

1. **Why does zero-copy not always give the best throughput?**
   Zero-copy does not guarantee superior performance because it trades memory bandwidth overhead for kernel management overhead. While it eliminates the CPU cost of copying data from user to kernel space, it introduces the cost of page pinning, memory mapping, and handling completion notifications via the error queue. For small messages (like 1KB in our tests), this administrative work takes longer than a simple `memcpy` instruction. The CPU spends more cycles setting up the zero-copy transaction than it would have spent just copying the few bytes, resulting in lower throughput compared to the baseline.

2. **Which cache level shows the most reduction in misses and why?**
   The Last Level Cache (LLC) showed the most significant reduction in misses. In the standard Two-Copy approach (A1), the CPU must read the user buffer and write it to a kernel buffer, polluting the cache with payload data that is essentially just "passing through." The Zero-Copy implementation avoids this copy entirely, meaning the CPU never needs to pull the payload data into the cache hierarchy.

This preserves LLC lines for other active threads and instructions, leading to a noticeable drop in miss rates.

3. **How does thread count interact with cache contention?**
   As we increased the thread count from 1 to 8, throughput improved, but the scaling was not linear due to cache contention. With 8 active threads, multiple cores compete for space in the shared LLC. In the Two-Copy implementation, this contention is severe because every thread is aggressively reading and writing memory to copy buffers. While Zero-Copy reduces the volume of memory movement, running 8 concurrent threads still introduces high context switching and lock contention within the kernel (e.g., locking page tables for pinning), which prevents perfect linear scalability.

4. **At what message size does one-copy outperform two-copy on your system?**
   On my system, the One-Copy (A2) implementation outperformed the Two-Copy (A1) baseline immediately, starting at the smallest tested message size of 1 KB. At this size, A2 achieved approximately 1.29 Gbps compared to A1's 0.40 Gbps. This immediate advantage occurs because A1 requires 8 separate `send()` system calls (one for each field) to transmit the non-contiguous data, whereas A2 combines them into a single `sendmsg()` call. The reduction in user-kernel boundary crossings provided a massive performance boost regardless of the payload size.

5. **At what message size does zero-copy outperform two-copy on your system?**
   In this specific virtualized environment, Zero-Copy (A3) struggled to consistently outperform the baseline. It only began to show a marginal advantage at the largest message size of 65 KB. For all smaller sizes (1KB to 16KB), the setup costs of Zero-Copy (page pinning) outweighed the savings from avoiding the copy. This suggests that without dedicated hardware offloading, the software overhead of Zero-Copy is significant enough to negate its benefits for small to medium payloads.

6. **Identify one unexpected result and explain it using OS or hardware concepts.**
   The most unexpected result was that the One-Copy (A2) implementation consistently outperformed Zero-Copy (A3), even at the maximum message size of 65KB (71.14 Gbps vs 39.66 Gbps). Theoretically, Zero-Copy should be the fastest for large data transfers. However, this anomaly is likely caused by the test environment using Virtual Ethernet (`veth`) pairs. Since `veth` is a software-only device, it lacks a physical DMA engine to offload the memory reading. The kernel still has to manage the transfer in software, meaning we paid the high "setup cost" of `MSG_ZEROCOPY` without getting the "hardware benefit" of DMA. Consequently, the optimized scatter-gather copy in A2 proved to be more efficient than the complex zero-copy mechanism in this specific virtual setup.

5 Project Repository

The complete source code, build scripts, and version history for this assignment are available in the following public GitHub repository:

`https://github.com/Ashil46/GRS-PA02`

6 AI Declaration

In compliance with the assignment guidelines, the following log lists the specific prompts used during development:

1. **Socket Implementation (Part A2):** "Show me a C code example of using `sendmsg` with `struct iovec` to send multiple non-contiguous buffers in a single system call. How does this compare to calling `send` multiple times?"
2. **Zero-Copy Logic (Part A3):** "How to implement MSG_ZEROCOPY in Linux C sockets? I mean how do I poll the MSG_ERRQUEUE to handle completion notifications?"
3. **Debugging Script:** "My automation script fails with 'Connection Refused' errors when running the client immediately after starting the server, Also I see 'Killed' messages in the terminal. Is this a bug?"
4. **Perf Tool Syntax:** "What is the specific `perf stat` command to measure 'cycles', 'context-switches', and 'LLC-load-misses' for a specific process ID? I need to attach it to a running server instance." (Used to correct the syntax in the bash automation script).
5. **Data Analysis:** "My experimental data shows that One-Copy (A2) is significantly faster than Zero-Copy (A3) even at 65KB message sizes. I am using network namespaces and veth pairs. Why is this happening?" (Used to identify the Virtual Ethernet bottleneck).
6. **Plotting Compliance:** "Generate a Python matplotlib script to plot Throughput vs Message Size. "
7. **Report Generation:** "Generate a comprehensive LaTeX report template for a Network I/O analysis project. Include sections for System Configuration, Implementation Details, and AI declaration. "