PA02 - Analysis of Network I/O primitives using perf (Report Template)

Title Page

- Assignment: Graduate Systems (CSE638) - PA02
- Title: Analysis of Network I/O primitives using perf
- Student name: Rahul Pardasani
- Roll number: MT25035
- GitHub repository URL: https://github.com/rahul25035/GRS_PA02

Abstract

This assignment studies the cost of data transfer in network I/O by comparing two-copy, one-copy, and zero-copy TCP socket codes. Multithreaded client–server programs were written in C and tested with different message sizes and thread counts. Throughput and latency were measured at the **application level**, and CPU cycles, cache misses, and context switches were measured using **perf**. The results show that reducing data copies can improve performance, but zero-copy does not always give the best throughput due to extra kernel overhead. The experiments highlight the trade-offs between reducing copies and overall system performance.

1. List of Submitted Files

- **MT25035_Part_A_A1_client.c**
  Client code for the **two-copy** socket communication using send() and recv().

- **MT25035_Part_A_A1_server.c**
  Server code for the **two-copy** socket communication using send() and recv().

- **MT25035_Part_A_A2_client.c**
  Client code for the **one-copy** socket communication using sendmsg().

- **MT25035_Part_A_A2_server.c**
  Server code for the **one-copy** socket communication with reduced data copying.

- **MT25035_Part_A_A3_client.c**
  Client code for the **zero-copy** socket communication using sendmsg() with MSG_ZEROCOPY.

- **MT25035_Part_A_A3_server.c**
  Server implementation for the **zero-copy** socket communication using MSG_ZEROCOPY.

- **MT25035_Part_C_Results.csv**
  Raw CSV file containing experimental results for all implementations, message sizes, and thread counts.

- **MT25035_Part_C_shell.sh**
  Bash script to compile all programs, run experiments across different configurations, and collect profiling data.

- **MT25035_Part_D_Plots.py**
  Python script using **matplotlib** to generate all required plots with hardcoded values.

- **README.md**
  Documentation explaining the project structure, build steps, execution instructions, and usage details.

- **Makefile**
  Makefile containing compilation rules for all implementations

- **MT25035_PA02_Report.pdf**
  Final report containing implementation details, experimental results, plots, analysis.

2. System Configuration (Environment)

- Machine Type: 64-bit architecture
- OS and kernel version: Ubuntu 24.04.03
- CPU core count: 8
- RAM size: 8GB
- Compiler and flags used : gcc --version, -O2 -pthread
- perf
- Python & Matplotlib (for plots)

3. AI Usage Declaration

AI tools used: ChatGPT, Google Gemini

Components where assistance was taken

The following parts of the assignment involved guidance or refinement assistance from AI tools:

1. Client–Server networking programs (A1, A2, A3 implementations)
2. Linux namespaces setup and isolation
3. Multi-threaded server design and concurrency handling
4. Shell scripting and experiment automation
5. awk-based metric calculations and result processing
6. Plot generation and visualization in Python
7. Report formatting, paraphrasing, and consistency checks

Nature of assistance

- Refactoring and organizing existing code
- Debugging logical and runtime errors
- Adding missing functionality required by the assignment
- Improving threading and namespace configuration
- Fixing script bugs and loop issues
- Helping derive awk expressions for metrics
- Improving plot scripts to match required outputs
- Formatting and polishing the report

All final code, experiments, and analysis were written, executed, and understood by me. AI tools were used only for guidance, debugging help, and refinement.

Prompts used and file affected:

1. **MT25035_Part_A_A1_client.c, MT25035_Part_A_A1_server.c, MT25035_Part_A_A2_client.c, MT25035_Part_A_A2_server.c, MT25035_Part_A_A3_client.c, MT25035_Part_A_A3_server.c**
    - Refine this code such that separate namespaces are made for the client and server programs.
    - Fix the part where it sends 8 bytes message for the fixed duration.
    - I cant do the part where each client has its own thread in server. Fix that please.
2. **MT25035_Part_C_main.sh**
    - Add the different namespace functionality in this shell script.
    - There is a bug in this script and my program does not work after the first loop iteration. Fix that.

- Help me with the awk calculations for the calculation of the metrics given in the assignment.

3. **MT25035_Part_D_Plots.py**
    - Refine the code of the plot such that these plots are generated which are given in the question.
4. **MT25035_PA02_Report**
    - Fix the formatting of the document and give me in canvas so that i can copy paste.
    - Paraphrase this section.
    - Check if my analysis that i have written is consistent with the results of the experiment or not.

4. Implementation Details

This section describes the final implementation used in the experiments. The description below matches the **current codebase**, including the use of **network namespaces** and the exact behavior of A1, A2, and A3.

4.1 Common Requirements (All Versions)

Concurrency
The server handles multiple clients concurrently. Each client connection is served by a **separate pthread**, created after accept().

Message Structure
Each logical message consists of **8 fixed-size fields**. On the server side, these fields are stored in a struct message, where each field is a dynamically allocated heap buffer.

Memory Management
For every client connection:

- The server allocates 8 heap buffers using malloc() inside create_message().
- These buffers remain allocated for the lifetime of the client connection.
- When the client disconnects, all buffers are freed using free_message().

The client uses **stack-allocated buffers** for sending requests and receiving responses.

Runtime Parameters

- Server: ./server <field_size> <max_threads>
- Client: ./client <field_size> <num_threads> <duration>

Network Isolation
The client and server run in **separate Linux network namespaces**. A virtual Ethernet (veth) pair connects the namespaces, and communication happens using a private IP address. All sockets and threads inherit the namespace of the process.

System Calls and APIs Used

- Networking: socket(), bind(), listen(), accept(), connect(), send(), recv(), sendmsg()
- Threading: POSIX pthread APIs
- Namespace management (external to application logic): ip netns

4.2 A1 – Basic TCP Communication (send / recv)

Files

- MT25035_Part_A_A1_server.c – Multithreaded TCP server using send()
- MT25035_Part_A_A1_client.c – Multithreaded TCP client using recv()

Server Flow

1. Create a TCP socket using socket().
2. Bind the socket to port 8080 and start listening using listen().
3. Accept incoming client connections using accept().
4. For each client, create a new thread.
5. Allocate a message structure containing 8 heap-allocated fields.
6. Receive request data from the client using recv().
7. Send the response by calling send() **once per field** (8 send calls).
8. On client disconnect, free all allocated memory and close the socket.

Client Flow

1. Create multiple threads based on num_threads.
2. Each thread creates a socket and connects to the server.
3. The client sends a single request message containing all 8 fields using send().
4. The response is received using **8 separate recv() calls**, one per field.
5. This loop continues for the specified duration, after which the socket is closed.

Copy Behavior

- Data is copied from user space to kernel space during send().
- Data is copied back from kernel space to user space during recv().
- This represents a **standard TCP data path with multiple user-to-kernel copies**, one per field.

4.3 A2 – Vectorized I/O Using sendmsg()

Files

- MT25035_Part_A_A2_server.c – Multithreaded TCP server using sendmsg()
- MT25035_Part_A_A2_client.c – Multithreaded TCP client using recv()

Server Flow

1. Create a TCP socket, bind to port 8080, and listen for connections.
2. Accept each client and handle it in a separate detached thread.
3. Allocate a message structure with 8 heap-allocated fields.
4. Initialize an array of iovec structures, each pointing to one field buffer.
5. Send all 8 fields together using a **single sendmsg() call**.
6. Repeat this process for each client request.

7. On client disconnect, free all allocated memory and close the socket.

Client Flow

1. Spawn multiple client threads.
2. Each thread connects to the server and sends a combined request message using send().
3. The response is received using repeated recv() calls, one per field.
4. The loop runs for the specified duration.

Copy Behavior

- sendmsg() reduces the overhead of multiple system calls compared to A1.
- Data is still copied from user space to kernel space before transmission.
- Kernel-to-NIC copies still occur as part of the TCP/IP stack.
- This implementation reduces overhead compared to A1 but **does not eliminate data copies**.

4.4 A3 – Zero-Copy TCP Transmission

Files

- MT25035_Part_A_A3_server.c – TCP server using Linux zero-copy support
- MT25035_Part_A_A3_client.c – Multithreaded TCP client using recv()

Server Flow

1. Create a TCP socket, bind to port 8080, and listen for connections.
2. For each client, spawn a detached thread.
3. Enable zero-copy transmission on the connected socket using setsockopt() with SO_ZEROCOPY.
4. Allocate a message structure with 8 heap-allocated fields.
5. Build an array of iovec structures pointing to the field buffers.
6. Transmit all fields using sendmsg() with the MSG_ZEROCOPY flag.
7. Repeat for each request until the client disconnects.
8. Free allocated memory and close the socket.

Client Flow

1. Create multiple client threads.
2. Each thread connects to the server and sends a combined request using send().
3. The response is received using repeated recv() calls, one per field.
4. Execution continues for the specified duration.

Zero-Copy Behavior

- With SO_ZEROCOPY enabled, the kernel can transmit data without copying it into kernel buffers.
- User-space buffers are referenced directly until transmission completes.
- This reduces CPU overhead caused by memory copying compared to A2.
- The TCP/IP stack is still used; only the data copy path is optimized.

Limitations

- Completion notifications from the socket error queue are not explicitly processed.
- Buffers may be reused without waiting for zero-copy completion events.
- The implementation demonstrates the zero-copy data path but does not fully manage asynchronous completion handling.

5. Build & Run Instructions

This section describes how the final experiments were built and executed.

Compilation

All implementations are compiled using gcc with debugging enabled and optimizations disabled.

You can compile everything using:

```
Shell
make all
```

This generates the following binaries:

- a1_server, a1_client
- a2_server, a2_client
- a3_server, a3_client

Running the Experiments

The final experiments are executed using an automated script. The client and server run in **separate network namespaces** to ensure isolation.

To run the complete benchmark:

```shell
sudo make partC
```

This command:

- Creates network namespaces and a veth connection
- Starts the server in the server namespace
- Runs the client and perf stat in the client namespace
- Collects performance counters and saves results to a CSV file
- Generates plots automatically

Notes

- Root privileges are required for network namespaces and perf.
- Manual execution in separate terminals is not used for the final results.
- The client prints per-thread message counts, while perf stat reports hardware counters for analysis.

Performance measurement

Performance statistics are collected using perf stat by running it on the **client process** from the terminal.

The reported counters include CPU cycles, cache misses, last-level cache misses, and context switches. These values are recorded for analysis after each run.

Experiment repetition

The above steps are repeated for different field sizes and thread counts for all three implementations (A1, A2, and A3). Results from multiple runs are later aggregated and analyzed offline.

6. Profiling Methodology (Part B)

This section describes how performance metrics are measured in a consistent manner for all implementations.

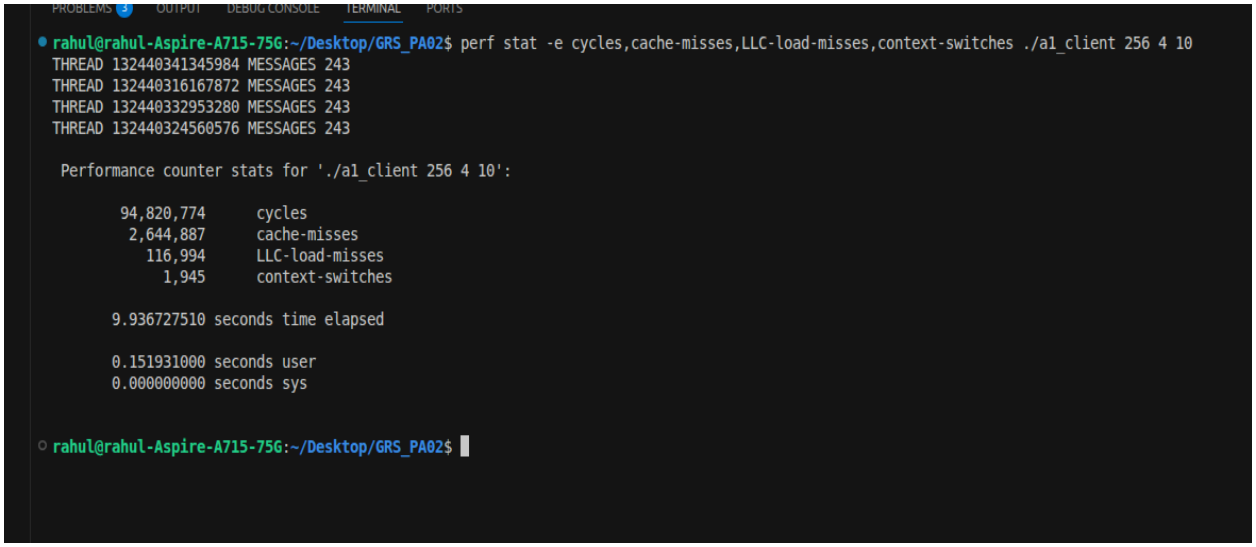
Throughput (Gbps): Throughput is computed at the application level using the total number of bytes processed during the duration. It is calculated as:
(total_bytes × 8) / (duration × 10^9).
The total number of messages is printed by the client, from which total bytes are calculated.

Latency (µs): Average latency is estimated at the application level as the total experiment duration divided by the number of messages processed. The value is converted to microseconds.

Hardware performance counters: Hardware-level metrics are collected using perf stat on the client process. The following counters are recorded:

- cycles to measure CPU execution cost
- cache-misses and LLC-load-misses to analyze cache behavior
- context-switches to capture scheduling overhead

Each configuration is executed once per parameter set. The server is started first and allowed to initialize before the client begins, ensuring that connection setup overhead is excluded from measurements.



```
rahul@rahul-Aspire-A715-75G:~/Desktop/GRS_PA02$ perf stat -e cycles,cache-misses,LLC-load-misses,context-switches \
./a3_client 2048 8 10
THREAD 134910362769088 MESSAGES 166619
THREAD 134910354376384 MESSAGES 173577
THREAD 134910345983680 MESSAGES 172735
THREAD 134910320805568 MESSAGES 174868
THREAD 134910337590976 MESSAGES 174368
THREAD 134910329198272 MESSAGES 162158
THREAD 134910379554496 MESSAGES 172847
THREAD 134910371161792 MESSAGES 172377

 Performance counter stats for './a3_client 2048 8 10':

    91,477,607,076      cycles
         3,919,338      cache-misses
           267,882      LLC-load-misses
         1,371,106      context-switches

       9.005440176 seconds time elapsed

       1.703392000 seconds user
      33.647632000 seconds sys
```

```
PROBLEMS 3  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

rahul@rahul-Aspire-A715-75G:~/Desktop/GRS_PA02$ perf stat -e cycles,cache-misses,LLC-load-misses,context-switches ./a2_client 1024 8 10
THREAD 127525996959424 MESSAGES 224157
THREAD 127526038922944 MESSAGES 220720
THREAD 127526047315648 MESSAGES 225336
THREAD 127526030530240 MESSAGES 222907
THREAD 127526005352128 MESSAGES 221055
THREAD 127525988566720 MESSAGES 228810
THREAD 127526013744832 MESSAGES 227957
THREAD 127526022137536 MESSAGES 225895

 Performance counter stats for './a2_client 1024 8 10':

   112,753,860,062      cycles
         3,036,732      cache-misses
           198,052      LLC-load-misses
         1,798,500      context-switches

       9.884319708 seconds time elapsed

       2.171388000 seconds user
      41.816338000 seconds sys


rahul@rahul-Aspire-A715-75G:~/Desktop/GRS_PA02$
```

7. Experiment Automation (Part C)

All experiments are automated using a single Bash script. The script compiles all client and server binaries for implementations A1, A2, and A3.

After compilation, the script runs experiments for each implementation across multiple field sizes and client thread counts. For every configuration, the server is started in the background inside a server network namespace, and the client is executed inside a separate client network namespace for a fixed duration.

The client is run under perf stat to collect hardware performance counters, including CPU cycles, cache misses, last-level cache misses, and context switches. Application-level statistics such as total messages processed are printed by the client and captured from its output.

Using these values, throughput (in Gbps) and average latency (in microseconds) are computed automatically by the script. All metrics are written as rows into a single CSV file for later analysis.

After each experiment, the server process is terminated, and once all experiments complete, temporary binaries are removed and plots are generated from the collected CSV results. This automation ensures consistent and repeatable measurements across all configurations without manual intervention.

8. Results and Analysis

This section explains what we observed from the experiments and why the results look the way they do. The discussion is based directly on the measured values and plots.

Overall Observation

The results clearly show that **A1 performs the worst** in all cases. Both A2 and A3 perform much better than A1 across all message sizes and thread counts. Between A2 and A3, the behavior is similar, but they are optimized for slightly different goals. **A2 usually gives the highest throughput**, while **A3 is more CPU efficient**. In simple terms, A2 is faster, and A3 is lighter on the CPU.

CPU Cycles per Byte

For all implementations, CPU cycles per byte decrease as message size increases. This happens because the fixed overhead of system calls and protocol handling is spread over more data.

- **A1** has very high cycles per byte for small messages. This is because each message is broken into many send() and recv() calls, which adds a lot of overhead.
- **A2** reduces this cost by sending all fields together using sendmsg(). Fewer system calls means less CPU work.
- **A3** usually has the lowest cycles per byte because it avoids extra data copying between user space and kernel space.

For large messages, the difference between A2 and A3 becomes smaller, since copying overhead matters less when more data is sent at once.

Cache Misses

Cache behavior also shows clear differences between the implementations.

- **A1** has the highest cache misses, especially as message size grows. Frequent copying and repeated system calls cause more cache activity.

- **A2** shows the most stable and generally low cache misses. Sending data in one call results in more predictable memory access.
- **A3** reduces copying but does not always have the lowest cache misses. In some cases, cache misses increase due to buffer reuse and the way zero-copy works internally.

Overall, fewer system calls and fewer copies usually lead to better cache behavior.

Latency vs Thread Count

Latency drops as the number of client threads increases for all implementations.

- **A1** has very high latency when only a few threads are used. This is mainly due to heavy system call overhead.
- **A2** and **A3** have much lower latency even at low thread counts.
- Increasing the number of threads helps slightly, but it does not change the trend much for A2 and A3.

This shows that the main reason for high latency in A1 is system call and copy overhead, not poor threading.

Throughput

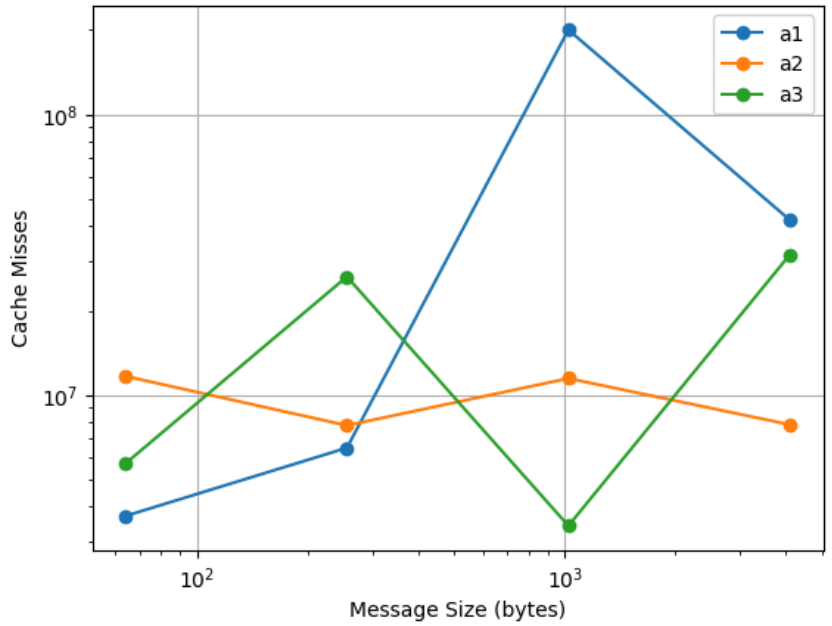
Throughput increases with message size for all three implementations.

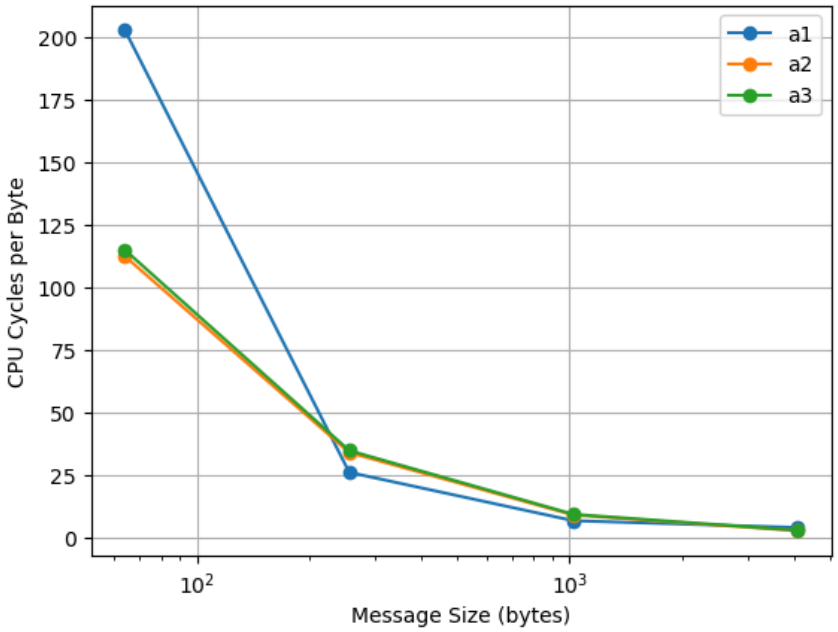
- **A1** has very low throughput because it spends most of its time in system calls.
- **A2** achieves the highest throughput in most cases by sending data efficiently using sendmsg().
- **A3** also scales well but usually has slightly lower throughput than A2. This is expected, since A3 focuses more on reducing CPU work than on maximizing raw speed.

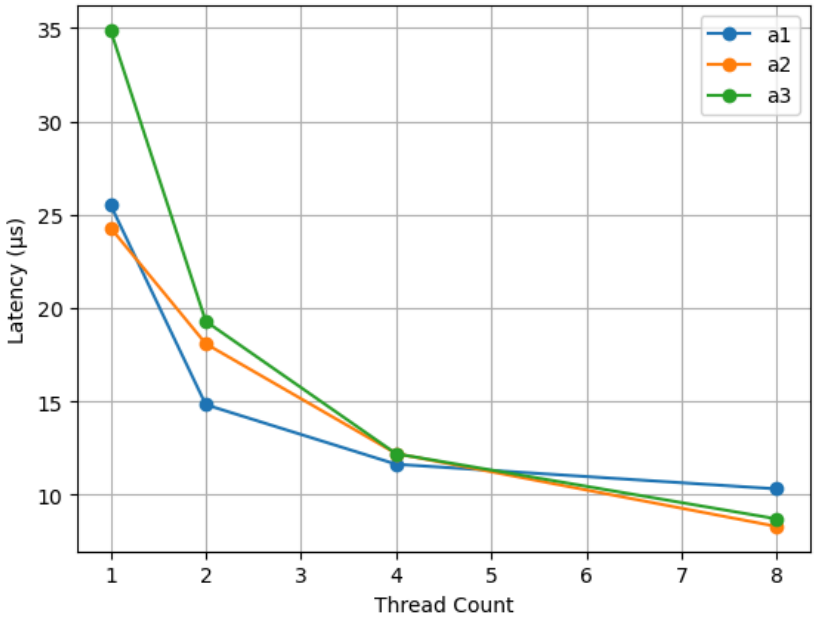
Summary

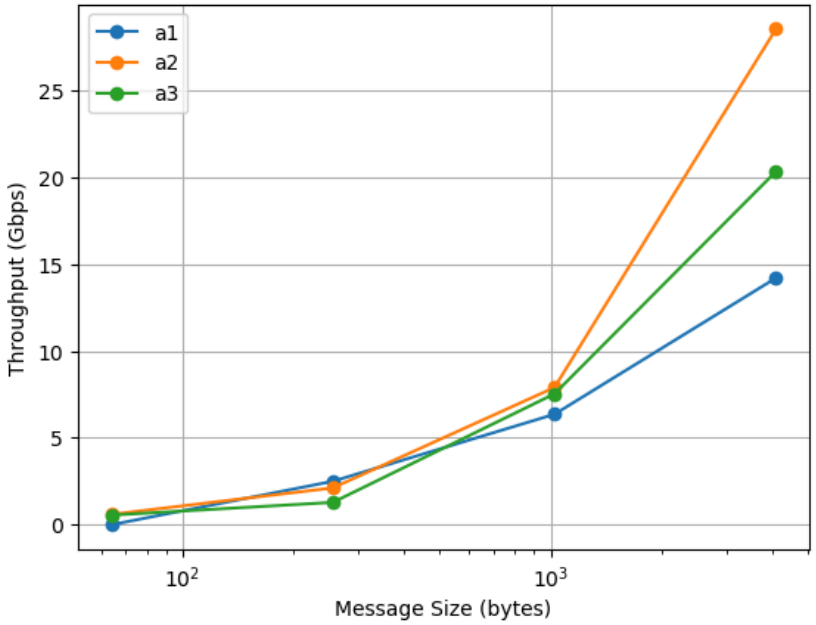
In summary, the experiments show that reducing system calls and data copying greatly improves performance. Moving from A1 to A2 gives the biggest improvement. A3 further reduces CPU overhead by avoiding extra copies, although it does not always give the highest throughput. These results highlight how important efficient I/O techniques are for high-performance client–server applications.

9. Plots (Part D)









10. Analysis & Reasoning (Part E)

1. Why does zero-copy not always give the best throughput?

From the results, zero-copy (A3) is not always faster than one-copy (A2), especially for small message sizes. For example, at 64 B, A2 achieves higher throughput than A3. This happens because, for small transfers, most of the time is spent in system call handling and TCP processing rather than copying data. The extra kernel bookkeeping needed for zero-copy reduces its benefit. As a result, zero-copy shows clearer advantages only for larger messages.

2. Which cache level shows the most reduction in misses and why?

The largest reduction in cache misses is observed when moving from A1 to A2 and A3. A1 uses multiple send() calls for each message, which repeatedly touches memory and increases cache pressure. A2 and A3 batch data into a single send operation, which improves locality and reduces frequent cache accesses. While LLC misses also decrease, the improvement is more visible in overall cache-miss behavior due to fewer repeated accesses.

3. How does thread count interact with cache contention?

As the number of threads increases, throughput improves, but cache misses and context switches also increase. More threads mean more cores competing for shared caches and CPU time. This leads to higher cache evictions and more scheduling overhead. The data shows that moving from 4 to 8 threads increases throughput, but also increases cache misses and context switches.

4. At what message size does one-copy outperform two-copy on your system?

One-copy (A2) outperforms two-copy (A1) at all tested message sizes, starting from the smallest 64 B messages. Even at small sizes and low thread counts, A2 performs much better than A1. This shows that reducing multiple send() calls and copies immediately improves performance.

5. At what message size does zero-copy outperform two-copy on your system?

Zero-copy (A3) also outperforms two-copy (A1) for all tested message sizes. By reducing data copying, A3 lowers CPU overhead and achieves much higher throughput than A1 across all configurations. This advantage becomes more noticeable as message size increases.

6. Identify one unexpected result and explain it

An unexpected result is that A2 sometimes achieves higher throughput than A3, even though A3 uses zero-copy. This is mainly seen for small message sizes. In these cases, the overhead of managing zero-copy inside the kernel is higher than the benefit of removing the copy. As a result, the simpler one-copy approach performs slightly better.

Overall, the results show that reducing system calls provides the biggest performance improvement, while zero-copy offers additional benefits mainly for larger messages.

11. Conclusion

Performance trade-offs
The experiments show a clear performance difference between the three implementations. The two-copy approach (A1) performs poorly due to repeated system calls and data copying, which leads to high latency and very low throughput. Moving to a one-copy design (A2) gives the largest improvement by batching data into a single send operation. The zero-copy design (A3) further reduces CPU and cache overhead, but its throughput advantage over A2 is smaller and mainly visible for larger message sizes.

Practical recommendations
For most real systems, the one-copy approach provides the best balance between performance and simplicity. Zero-copy is useful when CPU efficiency is important or when handling large messages under high load, but it adds complexity. The two-copy approach should be avoided for performance-critical applications and is best used only as a baseline for comparison.