Benchmarking of Inter-Process Communication Mechanisms

By Moldovan Anita-Györgyi

Introduction

Inter-process communication (IPC) is crucial in computer science because it allows different processes running on a system to share data with each other. Independent processes have their own memory space and execution context, but often need to share data or coordinate actions. IPC gives a way for this communication to happen, while also ensuring that the processes can remain independent. Without IPC, different processes running on the system would be isolated and unable to work together, severely limiting the system's capabilities and functionality.

This document compares different IPC mechanisms (TCP, FIFO, Unix sockets, message queues), and their performance with varying data sizes.

Test Scope definition

IPC Mechanisms Considered

- **TCP (Transmission Control Protocol)** is one of the main protocols of the Internet protocol suite. It originated in the initial network implementation in which it complemented the Internet Protocol (IP). TCP provides reliable, ordered, and error-checked delivery of a stream of octets (bytes) between applications running on hosts communicating via an IP network. For the current benchmark, both the server and client processes are running on the same host.
- **FIFO**, also known as named pipe, is an extension to the traditional pipe concept on Unix and Unix-like systems and is one of the methods of inter-process communication (IPC). A traditional pipe is "unnamed" and lasts only as long as the process. A named pipe, however, can last as long as the system is up, beyond the life of the process. It can be deleted if no longer used. Usually, a named pipe appears as a file, and generally processes attach to it for IPC.
- **Message queues** are typically used for inter-process communication, enabling communication between various system components. By separating message producers (senders) and consumers (receivers), message queues provide scalability, fault tolerance, and load balancing.
- **Unix domain sockets** allow data to be exchanged between two processes executing on the same Unix or Unix-like host computer. They are similar to an Internet domain socket that allows data to be exchanged between two processes executing on different host computers. For a Unix domain socket, the socket's address is a `/path/filename` identifier.

Tested payloads

The tests have been conducted using the Round-Trip Time method, where the client sends data of a given size to the server, then the server sends it back to the client. The setup times of

the IPC mechanisms are excluded from the measurements. This whole process happens a given number of times, which is set by one of the parameters of the client code.

The data sizes used for the measurements were 1KB, 8KB, 16KB and 32KB. Testing different data sizes is important to ensure we get a rounder picture at the end. Smaller messages typically result in lower latency since less data is processed, transmitted, and received. Larger messages, such as 16 KB and 32 KB, usually introduce higher latencies due to the increased data volume. This helps assess how each IPC method handles varying message sizes, with potential performance differences between small and large payloads.

These sizes represent common use cases in real-world applications, helping identify system constraints and throughput limits. Testing across these sizes provides insights into the scalability, efficiency, and overall performance of each IPC mechanism under different conditions.

Test Code, Toolset and Environment

Software Components

- For each IPC mechanism there is a corresponding server-client program written in C. The server does the setup for the given IPC, and also makes sure of a proper cleanup, while the client times the execution and returns it at the end. The client-server programs are vaguely similar, they mostly differ in the IPC used. It deserves to be mentioned that in the case of TCP, FIFO and Unix sockets safe read is implemented, to make sure that no data is left in the channel, but which is not necessary in case of message queues.
- **run_benchmark.sh** is a bash script used to execute all of the programs in various scenarios, and to extract the performance results (execution times) from the output of the client program. The script does 1000 rounds of test runs, which are eventually aggregated and analyzed for deviation. If the file containing the test results is not empty, the new test results are aggregated with the ones found in the file. The output of the script is a JSON file: results. Json.
- **chart_generator.py** is responsible for creating bar charts (as shown in this document) from the results. Json files. The generated charts are „candlestick bar charts”, where the max and minimum results of each testcase are also displayed as thin gray lines. They also do automatic scaling, both horizontally (dependent on the number of data points) and vertically (dependent on the values displayed).
- **generator.py:** generates pseudo-random data of a given size, used in the tests

Benchmarking environment

The current testing environment is a laptop with the following characteristics:

- 12th Gen Intel(R) Core (TM) i7-12650H, 2.30 GHz
- 16GB of RAM
- 10 cores, 16 logical processors

Test Executions and Results

Comparing the Round-Trip Time (RTT) Across Various Data Sizes

The test compares the performance of the IPC mechanisms over 100 RTT, sending data back-and-forth between the client and the server. The time taken for the setup and cleanup of the IPC mechanisms is excluded from the measurements.

Observations:

- **FIFO** consistently delivers the lowest average latencies (AVG), indicating its efficiency for small-scale data transfer. However, its maximum latencies (MAX) are higher than those of Unix sockets, which demonstrates a balance between stability and speed.
- **Message queues** exhibit the highest variability, as evidenced by their larger standard deviations, especially for larger message sizes, suggesting potential instability. **TCP**, while slower on average, maintains steady performance across sizes, but its high MAX values make it less predictable for real-time systems.

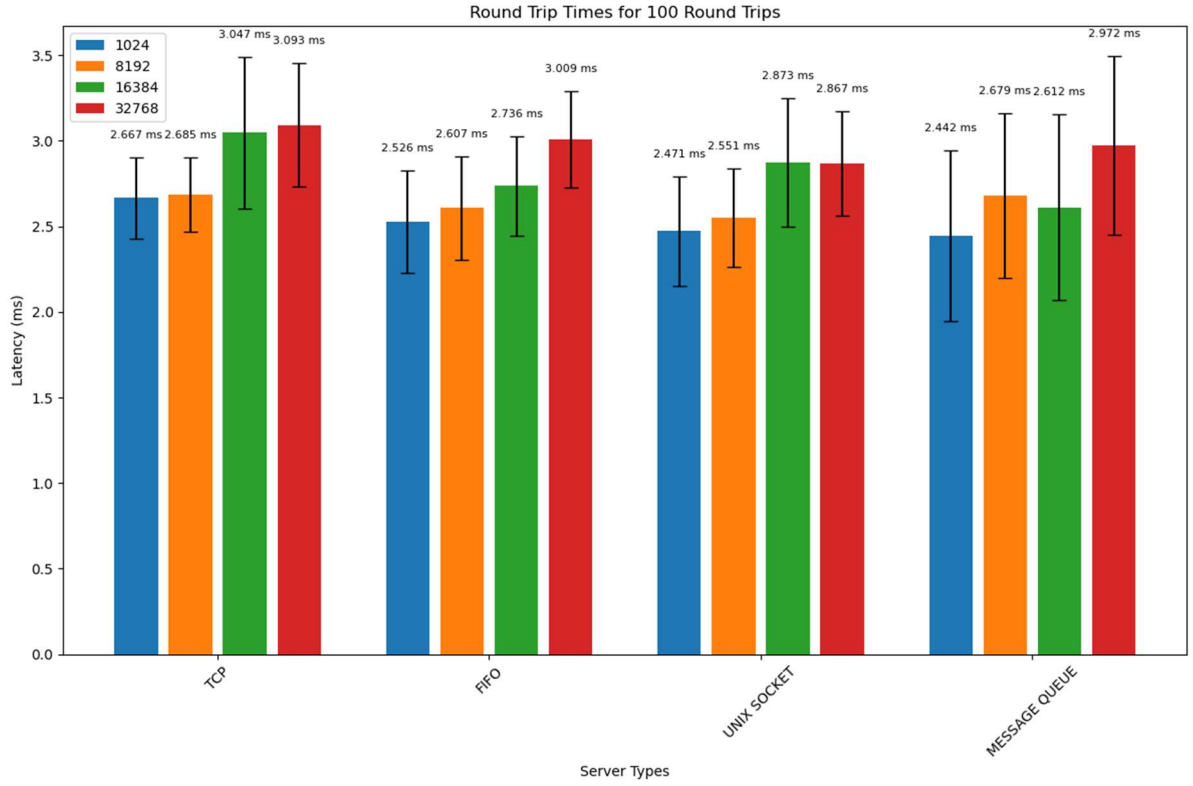

Figure: The Round Trip Times (RTT) Across Various Data Sizes

Conclusion

The analysis of the four IPC mechanisms—TCP, FIFO, Unix sockets, and message queues—shows clear distinctions in their performance and use cases.

FIFO consistently offers low average latencies across all message sizes, making it a strong choice for applications requiring efficient, sequential data transfer. However, its slightly higher maximum latencies suggest that while it is fast, it may not always deliver uniform performance under varying conditions.

Unix sockets strike a good balance between performance and stability. While their average latency is slightly higher than FIFO, their maximum latencies are lower, and their standard deviation is smaller. This makes them a solid option for systems prioritizing both speed and predictability, particularly for inter-process communication within the same machine.

Message queues stand out for their high variability. Their average latencies are competitive, but larger standard deviations indicate that they may introduce delays in high-pressure scenarios or with larger messages. Despite this, message queues could be advantageous in systems that rely on decoupled communication patterns or need built-in priority mechanisms.

TCP, while not the fastest, provides steady and reliable performance, particularly at larger message sizes. Its suitability extends to networked environments where reliability and compatibility across systems are more critical than raw speed.

In conclusion, the choice of IPC mechanism depends on the application's specific requirements. **FIFO** is ideal for simplicity and speed, **Unix sockets** for predictability, **message queues** for flexibility, and **TCP** for networked reliability. Understanding these trade-offs allows developers to select the best mechanism for their system’s goals.

All that said, the benchmark would benefit from extending the measurements to other scenarios, e.g.:

1. Expanding the benchmark to include multiple clients would simulate real-world conditions where concurrent processes communicate simultaneously.
2. Implement these mechanisms using other languages like Python or Java.

References

https://en.wikipedia.org

https://www.geeksforgeeks.org