Bandwidth and Latency Measurement on Inter-Process Communications

Kedar Bastakoti, Ruohui Wang

Abstract

There are multiple ways to do inter-process communications(IPC). For example, processes other use pipes to communicate with each other, whereas TCP and UDP can also choices to communicate both locally and remotely. Due to their differences in design, they tend to have different latencies and througputs. Our experiments showed such difference, and that the packet size and system calls are important factors to the latency and the througput of IPC.

1 Clock Resolution

To begin with, how can we measure the time? We need clocks. The Linux operating system, along with the x86-64 CPU, provides multiple methods for such purpose. But how can we choose one? One of the criteria is how precise they can be. And the resolution is an important standard for precision.

The resolution is the smallest possible increase of the clock. In order to measure the resolution, we tried to create minimal possible differences between two-time measurements. We create such difference by inserting a line of assembly `nop` into the code.

As per the POSIX manual[1], the function `gettimeofday` is obsolete, and the switch to `clock_gettime` is recommended, so we used `clock_gettime` instead, along with `clock_gettime`, function `clock_getres` is provided for user to query the resolution of time. As we will demonstrated, it produces the same result as ours.

Also, the x86 CPU provides `rdtscp` instruction to give the CPU timestamp, in terms of TSC(Time Stamp Counter) cycles. As Sergiu and Terry pointed out, to use TSC as a reliable clock source, it must be stable and have a constant rate, which can be examined using `\proc\cpuinfo`[2].

2 System Calls

Since we are measuring I/O performance, we cannot avoid doing a lot of system calls. System calls usually involve context switching, and trapping into the kernel. To get a better idea of how these system calls take up time. We measured some trivial system calls to see if the influence is actually trivial.

To measure the impact, we repeatedly make system calls that do least extra work. Here, we chose `getpid` and `getuid` as our syscalls. In addition, to avoid function returns that may take up time, we directly used the inline assembly to call the `syscall` instruction. And we return the minimum across all measurements to eliminate the effects of random events including page faults and preemptions.

3 Pipe

We measured the performance of pipes in terms of latency and throughput. For a pipe to be used, we first create a pair of pipes and then fork the process to allow inter-process communication.

We tried to measure the impact of how the number of bytes transmitted each time impacts the latency and throughput. Since the pipe does not transmit messages in the form of a packet, we simulate it by initiating `read` and `write` each time with a fixed `count` parameter.

We measure the latency in terms of RTT. And repeat the experiment 100,000 times to get the minimum latency as the final result. For latency, we make one process send 1GiB data, and the other process, on receiving all data, return a message to the sender, and use the time of the whole process to calculate the throughput. We apply the same method to the measurements on TCP/IP and UDP.

4 TCP

TCP uses sockets to communicate between the hosts and is used across local and remote communication. TCP uses byte-ordered reliable communication mechanism which is widely supported across today's operating systems. To measure the latency and throughput of the TCP stream sockets, packet sizes of 4, 16, 64, 256, 1K, 4K, 16K, 64K, 256K, and 512K were considered.

One of the issues encountered while measuring TCP latency was that when using the read and write function calls of the Rust language on TCP stream, the functions were reading and writing random bytes of data which were different than the expected values. This caused discrepancies in the read and write a sequence of the server and the client model. To remedy this problem, we switched to different function calls, `read_to_end` and `write_all`, which helped synchronize the flow of data between the client and the server.

5 UDP

Similar to TCP, UDP also uses sockets for communication. However, unlike TCP, UDP doesn't provide reliability mechanism or any guarantees on the transmission of the data. We hypothesized that similar to UDP, the TCP latency should increase as packet sizes increase, and the throughput will also increase as the packet sizes are increased.

As UDP didn't have the builtin mechanism to guarantee whether the data were received at the other end or not, it caused issues while performing the latency and throughput tests in various stages. Unlike TCP, there is no concept of server and client. Instead, we have a port number associated with each process and both of the processes send and receive using the port number of the other processes. When the sequences of read and write function calls were inconsistent the program would hang or crash after a while.

Another error we encountered during reading and writing was regarding how Rust's arrays work. A stack overflow error was received while running the throughput test. So, it was natural to check the function calls and the call stack to debug the issue.

However, it was detected that the stack overflow occurred due to the allocation of the arrays on the stack. The array which was used to test the throughput was larger than 1 GiB. Even though it was statically declared, it passed the compilation and only returned as a runtime error. To remedy this problem we allocated the array on the heap using the Rust language's Vector type.

6 Evaluation

We ran the experiments on the instructional machines, with Intel(R) Core(TM) i5-4570 CPU @ 3.20GHz, with 8GB RAM.

6.1 Clock Resolution

We experimented on a single machine on the resolution of both `clock_gettime` and `rdtscp`.

We used the real time clock for `clock_gettime`, and obtained 1ns resolution, which agrees what we get with `clock_getres`. For `rdtscp`, experiments give 1 cycle difference. To transform it into nanoseconds, we subsequently measured the TSC frequency, which is constant on the machine(nonstop_tsc and constant_tsc flag are on), by sleeping for a period of time(3 seconds in the test) and counting how many ticks have passed. It revealed that the TSC frequency roughly agreed with CPU base frequency, 3.2GHz. So we used `rdtscp` for subsequent measurements.

6.2 System Calls

We measured trivial syscalls `getpid` and `getuid`, `getpid` took 366.6ns, and `getuid` took 364.7ns.

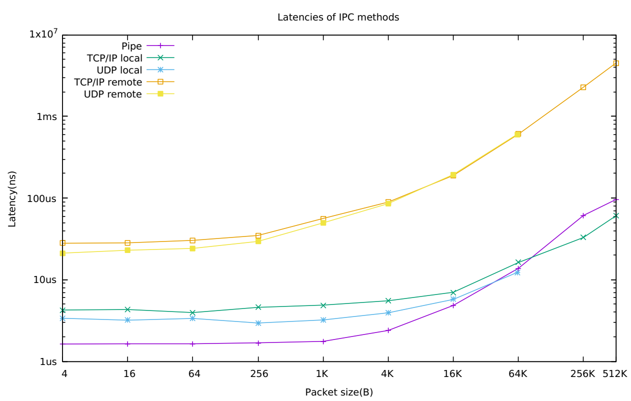
6.3 Latency and Throughput

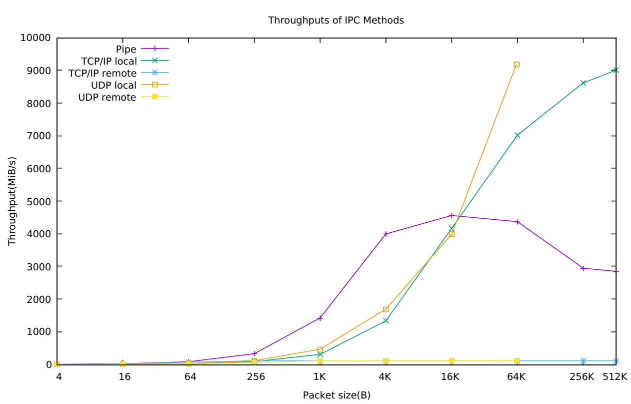
We measured IPC performance on a single machine, and between two machines in the same LAN.

The latencies of all IPCs are stable at first, and grow up after a threshold, which means the bottleneck are changed during the process – when packets are small a fixed time cost in the same order as system calls is dominant, but as they become larger, the transmission cost starts to take dominance.

On the other side, the throughput becomes larger as the packet size increases, but usually stops increasing at some point.

The measurement with pipe shows something interesting: the bandwidth first increases with the packet size, but after reaching around 16KiB packet size, the performance starts to deteriorate as the packet size grows up. We hypothesized the default 64KiB pipe buffer in Linux is the reason, but due to the limitation of the instructional machine, we cannot modify the pipe buffer size to verify such hypothesis.





7 Conclusion

We measured IPC performance, both locally and remotedly. We found out that the system call has an non-negligible impact on the latency of small packets, and we discovered that pipe could actually perform lower throughput than local TCP/IP and UDP, which hints that buffer size might also be the contributing factor.

References

[1] IEEE, O. The open group base specifications issue 7, 2018 edition, 2017.

[2] Sergiu Iordache, T. L. TSC resynchronization, 2019.