Benchmarking Inter-process Communications

Abbinaya Kalyanaraman,Akshaya Kalyanaraman

1 Abstract

The operating system provides general mechanisms for flexible interprocess communication (IPC). We have studied and evaluated three popular IPC techniques in Linux - pipes, Internet Stream/TCP sockets and Internet Datagram/UDP sockets. We have also identified the various factors that could affect their performance such as message size, host type such as remote or local, and have studied the effects of these factors to draw some insightful conclusions. We have constructed experiments to reliably measure the latency and throughput of each IPC and identified the most reliable timer APIs available for our measurements. Therefore, we have compared three timer APIs and picked the one with most reliable results to conduct our experiments. Our observations reveal that TCP/IP sockets provide the lowest latency with the highest throughput, followed by UDP sockets and lastly, pipes. The local host environments for the internet sockets proved to have lower latency and higher throughput.

2 Introduction

Inter-process communications are a set of programming interfaces that allow multiple processes to communicate with each other. All IPCs involve synchronization and management of shared data, either by the kernel or by the application. In this paper, we will study and evaluate the commonly-used and powerful IPC mechanisms - pipes, TCP and UDP sockets. [1]

2.1 Pipes

Pipe is a one-way communication medium only i.e we can use a pipe such that one process writes to the pipe, and the other process reads from it. A pipe is created using the pipe system call, which creates file-descriptors for writing and reading from the pipe. The standard programming model is that after the pipe has been set up, two (or more) cooperative processes will be created by a fork and data will be passed using read and write.

2.2 Sockets

Sockets provide point-to-point, two-way communication between two processes. Stream sockets use TCP [4], which is a reliable, stream oriented protocol and treat communications as a continuous stream of characters. Datagram

sockets use UDP, which is unreliable and message oriented, and read entire messages at once. In this paper we will evaluate local-machine sockets and remote-machine sockets.

3 Evaluation

Latency is the time for a given activity to complete, from beginning to end. For message passing, it is the time from the start of a send to the completion of a receive. Since the clocks on two different cores (for some timer APIs) may not be sufficiently aligned, the easiest way to measure message latency is to measure the time it takes to complete a round-trip communication (and divide by two).[3] Stated below are our hypotheses of the variables that we expect the latency to depend on:

- Message Size: Intuitively, one would expect the latency to be an increasing function of message size. We perform the experiment over message sizes ranging from 4B to 512KB in order to confirm this notion.
- Local/Remote host for TCP and UDP: If the messages are sent to the local host, it would be easier for the reading process to process it and the latency would be reduced considerably. Whereas, remote servers would enforce clients to send and receive messages at a much slower rate. Our experiment with different message sizes will test this hypothesis too.

3.1 Evaluation of Timers

The goal of the study is to empirically compute the latency and throughput of popular IPC mechanisms. We evaluated the accuracy and resolution of three popular Linux timer APIs - rdtsc, clock gettime() and clock(). The accuracy was measured by putting the program to sleep for a known time-period in-between two calls to the timer API and this was looped for a high number of iterations. To identify the smallest, reliable resolution, we successively reduced the number of instructions in between two calls to the timer API until the timer could not identify the time interval. As given in Table 1, our experiments measure the clock precision of the three clock timers that are considered and also measures the time for the trivial kernel calls getpid() and getuid()for each of the timers.
The results reveal that all three APIs are quite accurate for measurements in the order of nano-seconds. However, we find that rdtsc [2] and clock gettime() have a higher resolution than clock(), as the latter cannot differentiate within a nano-second. We conclude this section of our study by choosing rdtsc as a suitable choice for measuring IPC latencies.[3]
On a multicore system, different processor cores may have different values for the timestamp. In order to handle such a situation, we used the concept of round trip time, which measures the time from sending the message to receiving the message. The following method was used to calculate precision for each timer API:

- Calculate the timer value for performing a simple operation (For example : setting a memory location to a constant value) over several iterations.

TABLE I
EVALUATION OF CLOCK TIMERS BASED ON PRECISION AND KERNEL CALLS

Clock Timers	Clock Precision (ns)	Trivial Kernel Calls	
		getpid (ns)	getuid (ns)
rdtsc	5.62	362.81	360.00
clock_gettime()	17.00	384.00	382.00
clock()	1000.00	1000.00	1000.00

- Reduce the number of iterations until the timer can't calculate time difference of the operation, i.e. it starts showing 0.

3.2 Experiments

Communication latency is defined as the time between initiation of transmission at the sender and completion of reception at the receiver. However, due to aforementioned offsets in timer readings on different cores, we compute the communication latency for a round-trip communication and then halve it. Since the latency is not expected to be exactly symmetrical along each route, we send the data packets through multiple iterations. The throughput of a communication mechanism is the amount of data that can be sent per unit time. In our experiments, we send a large message to the receiver, who then acknowledges through a single Byte message.

3.3 Variables

We ought to be cognizant of the different variables that could affect our experiments. The variables identified are message size, and host environment. Naively, one would expect the latency to be linearly related to the size of the message; to confirm this notion, we consider a wide range of message sizes from 4B to 512KB.

3.4 Evaluation of IPCs

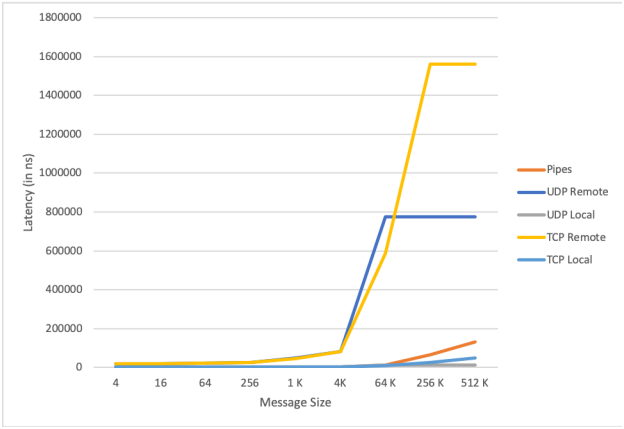
In this section, we describe the implementation and execution of the interprocess communication mechanisms discussed so far.

1) Implementation:

- Sockets: We have two processes for socket implementation - a server and a client. The server creates a socket and listens for a process to connect to it, while the client connects to server's socket using its port number. After the connection is set up, the server and client can send and receive messages through the socket.[4]
- Pipes: Pipe is created from a parent process using the pipe system call. The parent then forks into a child process which shares the pipe created by parent. This pipe can now be used to communicate between the parent and child.[4]

2) Calculation of latencies: In our experiments, we took a series of readings for each IPC and calculated the minimum of the readings after the readings seemed to have stabilized. The reason behind doing this was to eliminate any overheads associated with inconsistent cache behavior during the start of our experiments.
3) Calculation of throughput: We calculated the difference in timer values before and after writing the data for each IPC. We then obtained the number of cycles, which was divided by the frequency of the machine, thereby providing the timer value. The throughput value was obtained by dividing the message size by the timer value.

Figure 1: Latency vs. Message Size

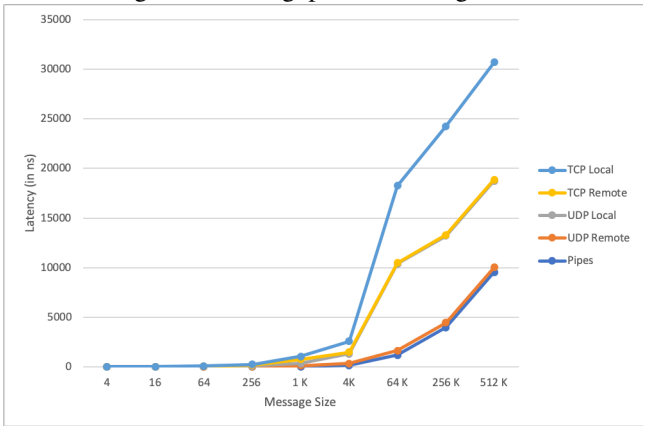


Let us consider the variation in latency as a function of message size, when the sender and receiver processes are fixed on different cores, as shown in Figure 1. We observe that in the local environment, the TCP sockets provides the minimum latency, followed by the UDP sockets and lastly pipes. While in the case of a remote environment, both TCP and UDP have similar latency, after which TCP's latency increases when the message size is 64K, while still lower than the UDP sockets. In each mechanism, the latency is roughly uniform for message sizes up to 4KB. This goes against our naive expectation of a linear relationship between message size and latency. This suggests that each mechanism uses a single-page transfer buffer for all messages up to 4KB. Beyond 4KB, the latencies start to rise as a function of message size.
Figure 2 shows the variation of throughput with message size. We see similar throughput in all three IPCs for message size less than 1KB. We observe that the throughput generally increases with message size.

4 Results

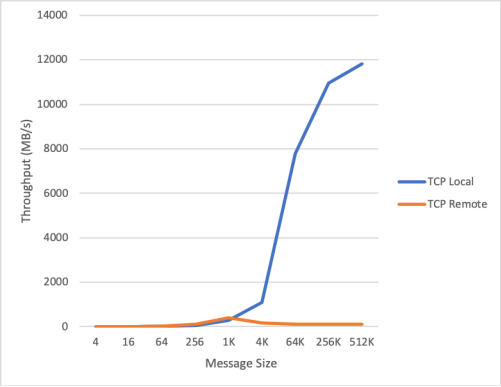
We see a similar trend for throughput in Figure 2. The pipes offer roughly similar throughput for small message sizes, but beyond 64KB, pipes experience a slight decrease in values, whereas TCP packets experience a much higher transfer rate after 4KB.
It is useful to compare the throughput of TCP local and remote connections as seen in Figure 3. We can see that TCP local has a much higher transfer rate as compared to remote host which slows down due to transmission overheads. Figure 3 illustrates the throughput usage of UDP local and remote hosts and it follows the same conclusion that the

Figure 2: Throughput vs. Message Size



local machine has a faster throughput than the remote one.

Figure 3: Throughput Comparison for local and remote sockets



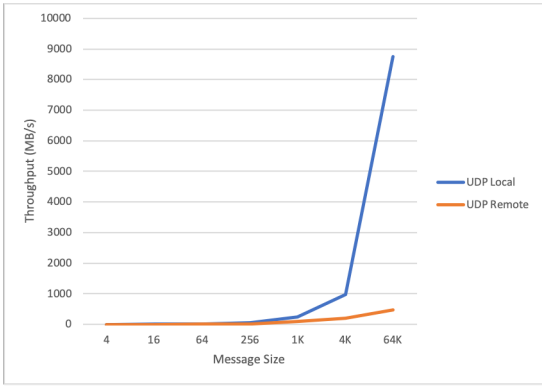
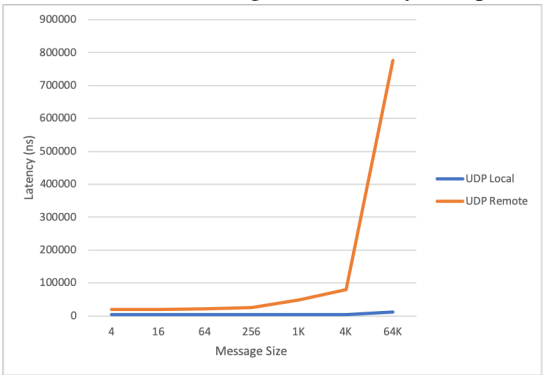
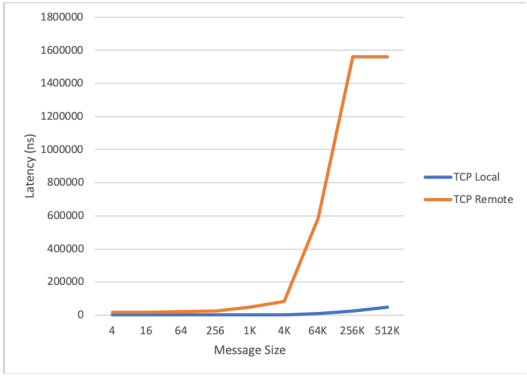


Figure 4: Latency Comparison for local and remote sockets





Comparing the latency of TCP local and remote connections as shown in Figure 4 proves that the remote host faces a high latency rate when compared to the local host which confirms the higher bandwidth rate of the local machine. Similarly, the UDP local connection faces a lower latency when compared to the remote connection as illustrated in Figure 4. This shows that host type is a factor influencing the performance of sockets.

5 Summary

In this work, we have studied and evaluated the performance of three popular inter-process communication mechanisms. We examined the accuracy and resolution of various timer APIs, settling on rdtsc and clock_gettime() as the most suitable choices for our study. We identified the variables that can influence the IPC latency and systematically constructed experiments to study their effects.[3]
We have identified TCP socket to be the fastest IPC mechanism because there is a reliable connection established and there is no question of packet loss. Being a packet-based mechanism, sockets may require breaking down large messages into multiple packets, increasing communication latency. However, sockets can be used for IPC between processes on different machines, allowing scalability for SW growth.

6 Conclusions

We did not set the CPU frequency before performing our experiments. This might have had some effect on our latency calculations since the CPU frequency keeps on fluctuating between a minimum and maximum value. Since the experiments were conducted on the same core of the same machine independently, we assume that the effect of CPU frequency fluctuations is uniform across all our experiments.
TCP Sockets are the best among the three IPCs tested. Pipes are the slowest, in terms of performance for all message size. Synchronization is simple with pipes and is built into the pipe mechanism itself - the reads (writes) will freeze and unfreeze the application based on the data (space) availability in pipe buffer. [4]

6.1 Acknowledgement

We would like to thank our instructor, Prof. Barton Miller for his helpful guidance in class for the completion of this project. We also acknowledge the contribution of our many class mates for insightful discussions and key experimentation ideas on tackling this problem.

References

Please find the code to the project in the below link:
`https://github.com/abbinayak/CS736_Benchmarking`

[1] Russell M Clapp, Trevor N Mudge, and Donald C Winsor. Cache coherence requirements for inter-process rendezvous. International Journal of Parallel Programming, 19(1):31–51, 1990.

[2] Intel Corporation. Using the rdtsc instruction for performance monitoring. Techn. Ber., tech. rep., Intel Coorporation, page 22, 1997.

[3] William Fornaciari. Inter-process communication. 2002.

[4] Jon Postel. Transmission control protocol. 1981.