UNIVERSITY OF WISCONSIN-MADISON
Computer Sciences Department

CS 736
Fall 2014

Barton Miller

Paper Assignment 1: Benchmarking Interprocess Communications

(Assigned: Wednesday, September 3)
(Due: Monday, September 22, noon (new date))

Description

The goal of this assignment is to get some experience doing benchmarking by measuring the performance of various operating system and interprocess communication mechanisms. You will design various experiments, build simple tools, and carry out a methodical experiment, summarize the results, and draw conclusions.

Be careful! Benchmarking is a subtle and tricky business; things that look simple on first glance will often turn out to be quite intricate.

The Communication Mechanisms

A. *Unix Pipe:* The most basic of IPC mechanisms on UNIX is the pipe; it has been around since the earliest versions of UNIX. The `pipe` system call is executed by a process to create both ends of a uni-directional communication channel. This channel is a stream of bytes that insures ordering and correct delivery. The pipe, when combined with a `fork` operation, allows two processes to pass messages. The `socketpair` operation will create a bi-directional channel that is equivalent to two pipes.

B. *Internet (INET) Stream Sockets:* The stream socket, based on TCP/IP, is the backbone of the Internet. These sockets can be used for remote (inter-host) and local communication, providing much the same abstraction as a pipe: a reliable, ordered byte stream. Almost every operating system supports communication over these sockets.

C. *Internet (INET) Datagram Sockets:* The datagram socket is based on the UDP protocol. As with the stream socket, it can be used for remote (inter-host) and local communication. It provides a message abstraction instead of a stream. In addition, it does not provide reliability or message ordering guarantees. However it does provide checksums so that if a message is delivered, it's contents are intact. As a simpler protocol, should have less overhead.

   NB: Datagram sockets can (and will) experience packet loss, so your experimental set-up must be designed to tolerate such loss. The tricky part is to figure out how to handle this so that you get meaningful results.

The Measurements

You will perform the measurements on one of the Linux systems available in one of the Computer Sciences Department's instructional labs. On this platform, you will measure the follow features:

1. *Clock precision:* The operating system and hardware provide various ways to measure time. Identify two ways of measuring elapsed time and determine the resolution (precision) of the clock.

   One way to do this is to read the clock value at the start and end of a simple loop. Start with a single loop iteration, then increase the iteration count of the loop until the difference between the before and after samples is greater than zero. Try to get the smallest non-zero positive difference. If a single iteration of a loop takes too much time, try putting simple statements between the two timer calls.

   Repeat this test for each of the two way that you measure time. Use the more precise way in the rest of your experiments.

2. *Trivial kernel call:* Choose a simple kernel call such `getpid` to measure and compare the elapsed time to perform the calls. Choose one or two other kernel calls that you suspect perform trivial operations, and measure the time to perform these.

   NB: Some Linux distributions have had their simple kernel calls modified to run faster just to look good on benchmarks, so it is important to measure more than one simple call.

3. *Inter-Process Communication Time:* For each of the communication mechanisms listed above, you will measure the following characteristics:

   a. *Message latency:* Latency is the time for some 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 hosts 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).

      NB: Beware of nagling on the internet stream experiments. This mechanism can cause unexpected delays. You can disable nagling with the `TCP_NODELAY` socket option.

      Measure latency for a variety of message sizes: 4, 16, 64, 256, 1K, 4K, 16K, 64K, 256K, and 512K bytes.

      NB: Watch out for message size limits on UDP. How will you handle experiments where you send a large enough UDB packet?

      NB 2: Transmission times can be affecting in TCP by the MTU (maximum transmission unit) size (typically 1500 bytes) and read and write buffer sizes (typically 128KB. If you have root access to a Linux system, you can experiment with increasing these sizes. Of course, you will not be allowed to do that on the instructional Linux systems.

   b. *Throughput:* Throughput is the amount data that is sent per unit time. In this case, a round trip measure is not necessary; you can sent a return message when the entire transfer amount has been sent. Send a large enough total quantity of data such that the single "ack" response contributes a small amount of time compared to the whole transfer.

      Measure throughput for a variety of message sizes, the same as 3a above (and same warning).

The Experimental Method

Computer Scientists are notably sloppy experimentalists. While we do a lot of experimental work, we typically do not follow good experimental practice. The experimental method is a well-established regimen, used in all areas of science. The use of the experimental method keeps us honest and gives form to the work that we do.

The basic parts of an experiment are:

- Identify your variables: Variables are things that you can observe and quantify. You need to identify which variables might be related and whether a variable is a cause (i.e., the message size of a send operation) or the effect (e.g., the time to complete the send). Even though this sounds obvious, you should consciously identify the variables in each experiment that you perform.
- Hypothesis: The hypothesis is a guess (we hope, an educated guess) about the outcome of the experiment. The hypothesis needs to be worded in a way that can be tested in an experiment, so it should be stated in terms of the experimental variables.
- Experimental apparatus: You need to obtain the necessary equipment for your experiment. In this case, it will be the needed computer and software.
- Performance of experiment and record the results: This part is the one that we typically think of as the real work. Note that several important steps come before it.
- Summarize the results: Summarization means putting the data in a form that you can understand. You might put the data in tables, graphs, or use statistical techniques to understand the raw data. If you are using averages, make sure to read Jim Smith's paper in the October 1988 issue of *CACM* (there are many types of means, and you need to use the right one)! However, as a warning, you probably do **not** want to use averages; taking the minimum makes much more sense in this case.
- Draw conclusions: Note that performing the experiment and summarizing the results are separate steps and both come *before* you draw conclusions. To present honest and understandable results, we must present the basic data first (so that the reader can draw their own conclusions) before we insert our bias.

The experimental method has more subtleties than this (such as trying to account for experimenter and subject biases), but the above description is sufficient for basic computer measurement experiments.

Learning about Sockets

If you need help with using the various ocket calls, here are some resources suggested by my members of my research group:

- http://www.wiziq.com/tutorial/7498-Socket-Programming-Introduction
- http://www.tenouk.com/cnlinuxsockettutorials.html

Constraints

The paper should be **at most 6 pages** (all inclusive), 10 point font, 18 point spacing, single-sided, one column, and 1 inch margins. If you do not understand any of these constraints, make sure to come talk with me.

The paper must contain the following parts:

Title:

The title should be descriptive and fit in one line across the page. Interesting titles are acceptable, but avoid overly cute ones.

Abstract:

This is the paper in brief; it is not a description of what is in the paper. It should state the basic ideas, techniques, results, and conclusions of the paper. The abstract is **not** the introduction, but a summary of everything. It is an advertisement that will draw the reader to your paper, without being misleading. It should be complete enough to understand what will be covered in the paper. Avoid phrases such as "The paper describes...." This is a technical paper and not a mystery novel; do not be afraid of giving away the ending.

Body:

This is the main part of the paper. It should include an introduction that prepares the reader for the remainder of the paper. Assume that the reader is knowledgeable about operating systems. The introduction should motivate the rest of the discussion and outline the approach. The main part of the paper should be split into reasonable sections that follow the basics of the experimental method. This is a discussion of what the reader should have learned from the paper. You can repeat things stated earlier in the paper, but only to the extent that they contribute to the final discussion.

References:

You must cite each paper that you have referenced. This section appears at the end of the paper.

Figures:

A paper without figures, graphs, or diagrams is boring. This paper will certainly need several performance tables and graphs. Your paper **must** have figures.

Do not re-describe the assignment; address the issues described above. The paper must be written using correct English grammar. There should be no spelling mistakes.

Note that your paper will be evaluated on both the technical content and the presentation (as would any paper submitted to a journal or conference).

Note that I have a list of writing suggestions available to help you avoid common mistakes. It is essential that you take a look at these suggestions as you prepare your paper.

Last modified: Wed Sep 17 08:02:33 CDT 2014 by bart

Kai Zhao
kzhao32@cs.wisc.edu
CS 736 Adv Operating Systems
University of Wisconsin-Madison
Department of Computer Sciences
2014 October 8

Title: TCP, UDP, and Pipes Communication Benchmarking

Abstract:

Computer Scientists are notoriously inconsistent with benchmarking performance of various message passing protocols. This paper will present protocols to measure the performance of Internet protocol suite (TCP and UDP) and interprocess communication (pipes). Latency and throughput should be measured by the same process because the clocks on different processes are difficult to align. Message latency is measured by half of the time to complete a round trip. On the other hand, message throughput is measured by the time it takes to send the message and receive a byte back as acknowledgment. This paper will discuss the methodology to obtain a fair benchmarking of messaging passing. The results show 1) UDP is faster TCP, 2) local communication is faster than remote communication, and 3) pipes are slower than local communication but faster than remote communication in terms of latency and throughput.

I. Introduction:

Computer scientists are notoriously inconsistent with benchmarking performance of various message passing protocols. The first common mistake is using a coarse clock precision, which will result in poor accuracy. An analogy is measuring with inches gives errors of up to half an inch while measuring with centimeters gives errors of up to half a centimeter. Therefore, measuring with smaller lengths or smaller clock precisions will give better accuracy. The next common mistake is using different clocks from different processes. This mistake is significant when clocks are misaligned by significant portion of the execution time. A better strategy is to use the same clock to measure the round trip time. The third common mistake is computing the message throughput as the message length divided by the message latency. Sending the same message back is the same amount of communication as sending a single byte acknowledgment. Therefore, the server to client message should count towards latency, but not throughput. The technical contributions of this paper are how to accurately benchmark TCP, UDP, and pipes communication to avoid common mistakes.

This paper is organized as followed: section II discusses TCP, UDP, and pipes; section III explains how to benchmark the communication to avoid the common mistakes; section IV is the latency and throughput results of TCP, UDP, and pipes; section V is the discussion of the results; and section VI is the conclusion and future works.

II. Protocols:

There has been a lot of benchmarking work on sockets. A socket is an endpoint used for inter-processor communication for message passing. The message passing helps pass data for function calls and helps with program synchronization for parallel processing [1].

The TCP protocol creates a physical connection between processes for sending data packets. The TCP protocol requires acknowledgment of each packet, or else it will block to resend the packet. Therefore, TCP guarantees reliable and ordered communication [2].

The UDP protocol creates a wireless connection between processes for sending data packets. The UDP protocol labels data packets and sends them in any order. The receiver will inform the sender of the packets it has and has not received yet so that the sender can resend packets [2].

Pipes are used for Inter-Processor Communication (IPC) to communicate address space that is protected from other outside access. Pipes are unidirectional and optimal for producer-consumer interactions among processes [3].

III. Experimental Setup:

Computing elapsed time between processes is difficult due to possible misaligned clocks. Therefore, the entire start and stop of the stopwatch will be done by only one process. Furthermore, the stopwatch should start right before the communication and stop right after the communication. This is because we want to only benchmark the communication time and no additional legwork (incrementing counters, writing results).

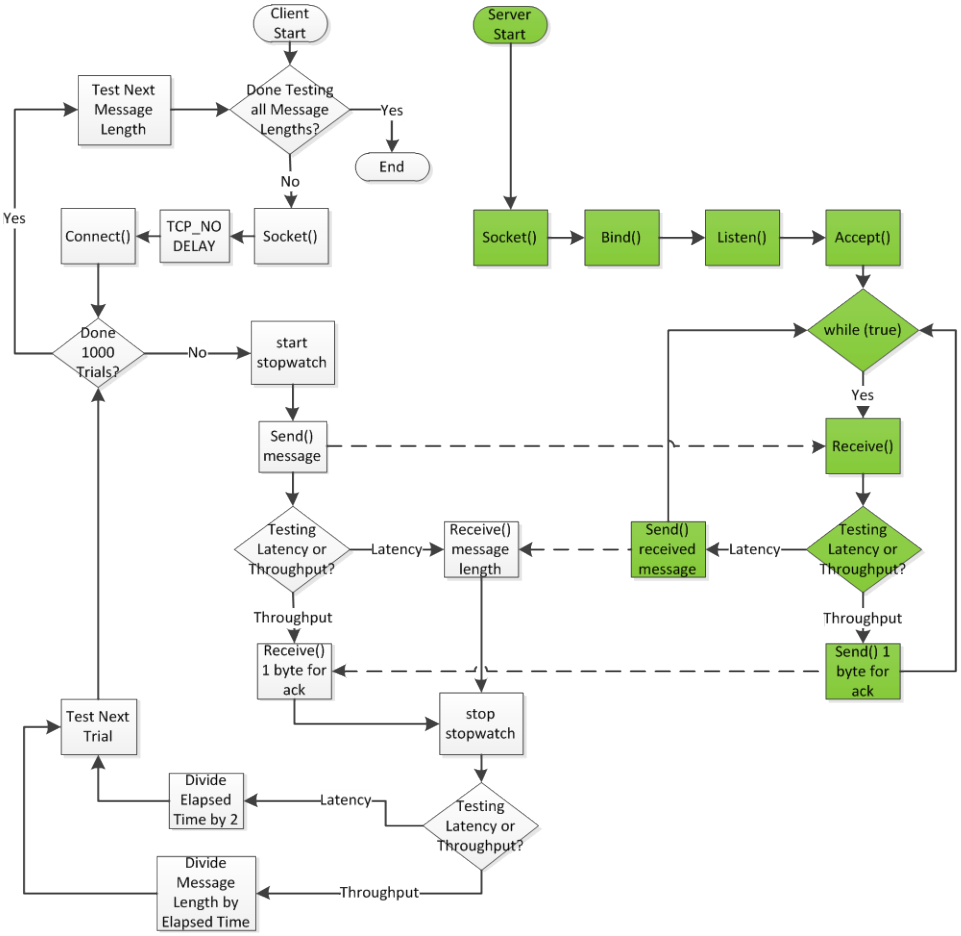

Figure 1: Flow Chart for Benchmarking Local and Remote TPC Latency and Throughput. The client processes are white while server processes are green.

The benchmarking protocol for the Local and Remote TCP Latency and Throughput protocol is shown in Figure 1. TCP tests every message length by sending and receiving messages to and from the client and server. The client and the server process executes concurrently. The TCP protocol uses TCP_NODELAY to avoid nagling to send messages as soon as possible at the cost of higher network bandwidth. Since the limiting factor should be the network protocol rather than the network bandwidth, using NODELAY gives a better benchmark of the TCP protocol [4]. A single socket is used to intentionally cause blocking. This allows latency to be measured by halving the time of the round trip message passing.

The UDP protocol is similar, except 1) UDP does not connect to the socket because UDP is connectionless and 2) UDP does not need to use NODELAY because UDP does not suffer from nagling [5]. Lastly, the UDP protocol may have multiple rounds of reads and writes if data packets were lost.

The pipe protocol was benchmarked by forking a process to pass messages from the child to the parent. To make the pipe unidirectional, the child process can write to the pipe by closing a file descriptor while the parent processor can read from the pipe by closing the other file descriptor [6].

All communication are assumed to reliable, or else the protocol will handle the retransmission of the data packers. Therefore, no assertions were used in code.

Each trial will run 1,000 times. More trials will require a longer runtime while fewer trials may lead to inconsistent data. The average is easily askew with outliers. Outliers are common in network benchmarking because the processor can context switch during the stop timer. In fact, context switches occur often because the instructional lab computers service many users via Secure Shell (SSH). Therefore, showing the minimum, first quartile, median, third quartile, and the maximum is more meaningful than showing the average.

There will be five trials that are executed but not recorded due to performance differences from compulsory/cold cache misses and branch predictor warm-up phase. In general, code usually runs much slower the first time through a loop. This is because of cold cache misses, where the variable that the code uses is likely not in cache [7]. Using data from disk is slower than using data from cache by magnitudes. Therefore, the cache should be in a consistent state for benchmarking purposes. Cache hits are a better model because 1) clients tend to use spatial and temporal data and 2) servers tend to serve the same client through multiple messages. Another reason for the slower execution the first time is the branch predictor warm-up phase. After the loop is executed repeatedly, the branch predictor will have history to make better branch predictions, which leads to more speculative execution and faster runtimes [8].

Latency measures the time to send and receive messages. However, since align clocks on different hosts is difficult, latency was measured by sending messages between processes and then sending the same message back. Therefore, the latency for each message size is half of the total time to send and receive the message.

Throughput measures the amount of data per a unit time. However, sending the same message back adds nothing more than a simple acknowledgment. Therefore, throughput was measured by sending the message through the communication channel and having the receiver send one byte back as acknowledgment. Then, the throughput is computed as the message length divided by the time to send the message and receive acknowledgment.

The trials were conducted on computers in University of Wisconsin-Madison Computer Science Department's instructional labs, macaroni-01 (local client and server) and macaroni-02 (remote server). Both of these machines have i5-4570 quad-core processors @ 3.20GHz with 6 MB cache and 16 GB of main memory.

IV. Results:

The clock precision of a median of 25 ns.

As shown in Figure 2, kernel calls of getpid() and getuid() took 25 ns and 59 ns respectively. Latency and Throughput of remote TCP, remote UDP, and pipes are shown in Figures 3-8.

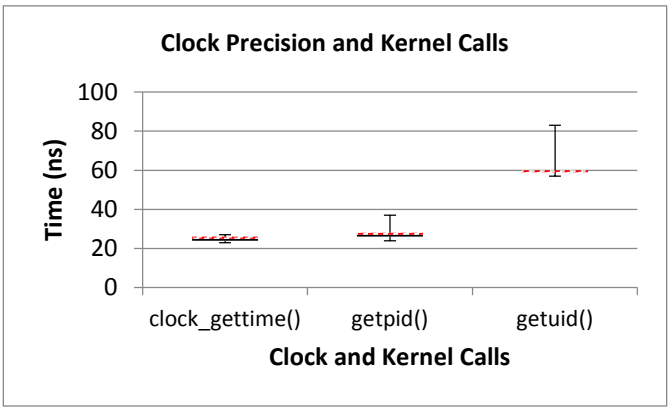

Figure 2: Clock Precision (25 ns) and Kernel Calls (23 ns to 80 ns).

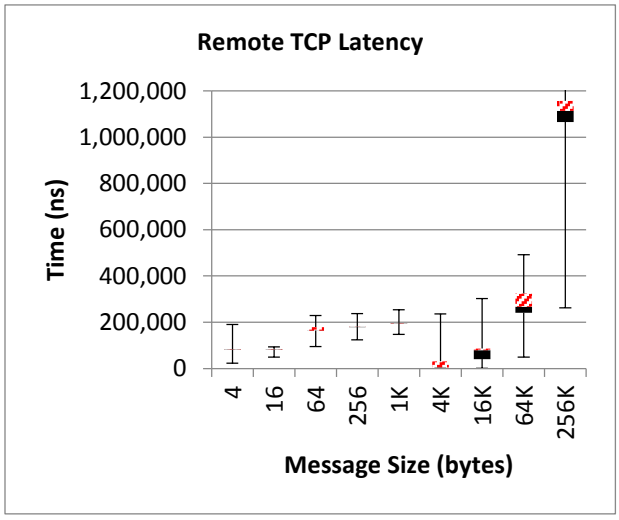

Figure 3: Remote TCP Latency, which varies from 80,000 ns to 1,100,000 ns. TCP_NODELAY helps larger message sizes.

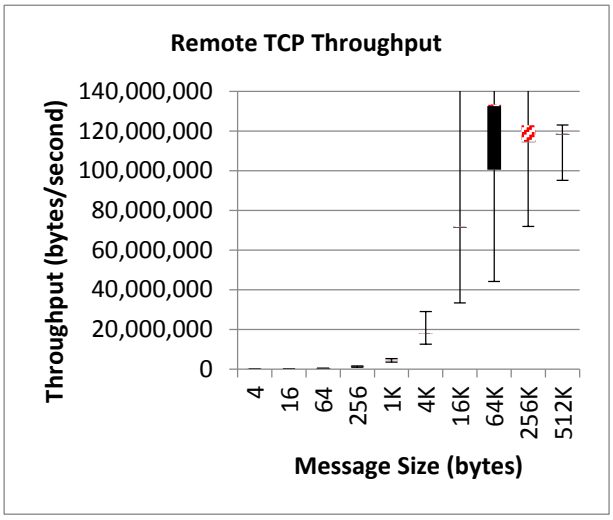

Figure 4: TCP Throughput, which varies from 25,000 bytes/second to 130,000,000 bytes/second.

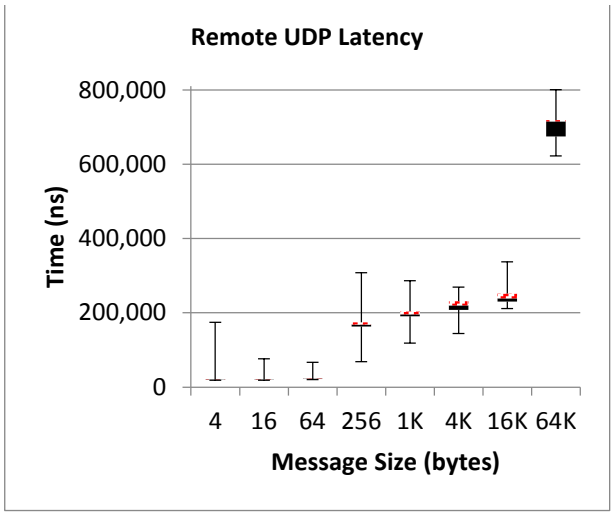

Figure 5: Remote UDP Latency, which varies from 20,000 ns to 700,000 ns. Large message sizes tend to be faulty.

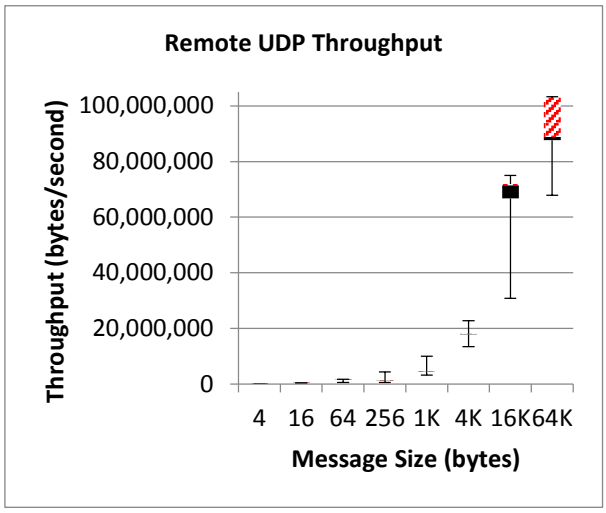

Figure 6: Remote UDP Throughput, which varies from 100,000 bytes/second to 100,000,000 bytes/second.

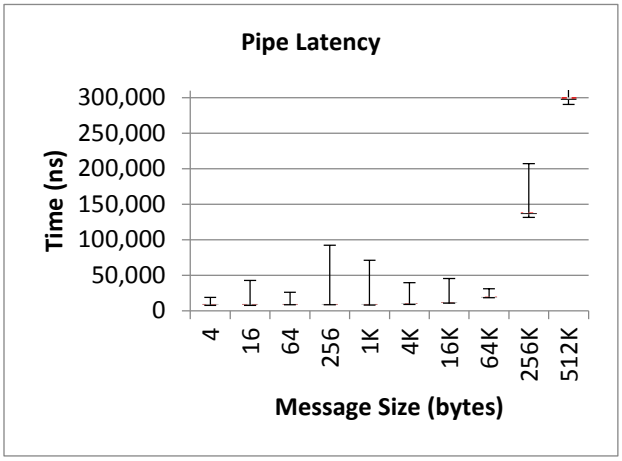

Figure 7: Pipe Latency, which varies from 8,000 ns to 300,000 ns.

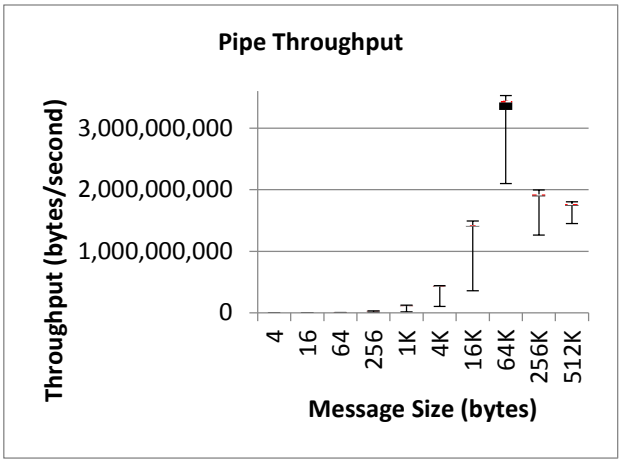

Figure 8: Pipe throughput, which varies from 200,000 bytes/second to 3,400,000,000 bytes/second.

Test	Message Size	Rank				
		1 (fastest)	2	3	4	5 (slowest)
Latency	16 Bytes	Local UDP	Local TCP	Pipe	Remote UDP	Remote TCP
	1K Bytes	Local UDP	Local TCP	Pipe	Remote UDP	Remote TCP
	64K Bytes	Local TCP	Local UDP	Pipe	Remote UDP	Remote TCP
Throughput	16 Bytes	Local UDP	Local TCP	Pipe	Remote UDP	Remote TCP
	1K Bytes	Local UDP	Local TCP	Pipe	Remote UDP	Remote TCP
	64K Bytes	Local TCP	Pipe	Local UDP	Remote TCP	Remote UDP

Table 1: Summary of Latency and Throughput Results for Various Message Sizes. This table shows which protocol is fastest in which situation.

As summarized in Table 1, Local UDP is the fastest, followed by Local TCP, Pipe, Remote UDP, and finally Remote TCP is the slowest in most cases.

V. Discussion:

A clock precision of 25 ns (results) running on 3.2GHz processors (macaroni computers) means there are 80 cycles (computed) between starting and stopping the stopwatch. Although 25 ns and 80 cycles are reasonable, it is not precise considering pipelines, where each instruction can execute different steps concurrently. Therefore, a more precise clock call could give a clock precision closer to 1 ns.

The kernel calls getpid() and getuid() took 27 ns and 59 ns respectively. getpid() is fast and close to the clock precision because the processor ID of the calling process is readily available. On the other hand, getuid() is slower because getting the real user ID of the calling process requires translational tables.

Local communication and pipes was always faster than remote communication in terms of both, latency and throughput. This is because local communication bypasses the switches and routers. Furthermore, there could be delay in remote communication from overloaded switches from servicing other computers in the instructional labs.

Except for large message sizes, UDP is faster than TCP. UDP is optimized for timely delivery and will send the entire message at once. Then, if there are any dropped packets, UDP will piggyback the dropped packets with the next message transmission. TCP, on the other hand, is optimized for accurate delivery and will attempt to retransmit packets as soon as a dropped packet is detected, which causes blocking [2]. TCP is faster than UDP for large message sizes because sending large messages are less unreliable, causing connectionless UDP to make multiple attempts to send and receive.

VI. Conclusion:

This paper investigated the various pitfalls in benchmarking. Obtaining accurate benchmarking data first requires obtaining a fine grain clock precision. Then, avoid any inconsistencies by showing quartiles of multiple measurements. The clock synchronization problem can be avoided by halving the time for a round trip communication. Lastly, accurately measure the latency and throughput.

The results were all as expected. Local communication is faster than remote communication. UDP is faster than TCP for small message sizes due to lack of error-checking and TCP is faster than UDP for large message sizes due to unreliable of connectionless UDP. Lastly, pipes are on the same magnitude of latency and throughput as local communication because they are essentially the same type of communication.

Future works include benchmarking the reliability of each protocol, benchmarking remote communication for distant networks with various numbers of switches and routers rather than using two computers right next to each other, and implementing Hamming or Golay error correcting codes as oppose to resending faulty data packets.

References:

[1] Hoare, C. Communicating Sequential Processes, *Communications of the ACM 21,* 8, August 1978, pp. 666-677.

[2] Fiedler, G. UDP VS. TCP, *gafferongames.com.* 22, September 2014. <http://gafferongames.com/networking-for-game-programmers/udp-vs-tcp/>.

[3] Liu, M., et al. Inter-process Communication Using Pipes in FPGA-Based Adaptive Computing, *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium*, pp.80,85, 5-7 July 2010.

[4] Miller, B. Paper Assignment 1: Benchmarking Interprocess Communications, *University of Wisconsin-Madison.* 22, September 2014. <http://pages.cs.wisc.edu/~bart/736/f2014/paper1.html>.

[5] Vasantha, Socket Programming, *WizIQ education.online.* 22, September 2014. <http://www.wiziq.com/tutorial/7498-Socket-Programming-Introduction>.

[6] Goldt, S. Creating Pipes in C, *The Linux Documentation Project*, 22, September 2014. <http://tldp.org/LDP/lpg/node11.html>.

[7] Hennessy, J., Patterson, D. Computer Architecture: A Quantitative Approach, *Morgan Kaufmann Publishers*, 30, September 2011.

[8] Mahlke, S.A; Hank, R.E.; McCormick, J.E.; August, D.I; Hwu, W.-M.W., "A comparison of full and partial predicated execution support for ILP processors," *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on* , vol., no., pp.138,149, 22-24 June 1995.