CS 428 Programming Assignment 1
Due Thursday 2/24/2004 in class

Please use C or C++. This is an individual project.

Part I:
Measure the performance (both latency and throughput) of sockets on the Sun machines in the POD. To accomplish this, you have to write a simple client server program that you can to exchange messages and measure performance. The server and the client should each run on a different machine (although it may be interesting to evaluate the results when they are both on the same machine).

To measure round-trip latency, use UDP and TCP to send and receive messages of size 1-byte, 100-bytes, 200-bytes, 300-bytes, ..., and 1000-bytes. Report the round-trip time for each protocol. Help on measuring time is below.

Start with UDP – it is easier because it is message oriented. To measure throughput, send messages of size 1KByte, 2KBytes, 4KBytes, 8KBytes, 16KBytes, and 32KBytes in one direction, with a message of the same size echoed back in the reverse direction.

TCP presents a challenge that I will leave up to you to solve. Specifically, it is stream oriented. To the application, there are no messages – there is only one continuous stream of data. When you do a receive, whatever is there may be received (multiple messages, or part of a message). So, if you send 5 messages, they may be all received with a single receive on the other side. Alternatively, if you send a big message, only part of it may be received on the other side. So, it is a challenge to match up your sends and receives. You may assume that you know the expected message size. Remember that receive tells you how many bytes it was able to receive.

Part II:
As a second measure of TCP throughput (this does not apply to UDP), measure how long it takes to send 1MByte of data from one PC to another (and a 1-byte application-level ACK in the reverse direction), varying the number of messages and size of each message; e.g., 256 x 4KByte messages, 512 x 2KByte messages, 1024 x 1KByte messages, 2048 x 512-byte messages, and so on. Measure enough combinations to discover TCP’s performance limits. The TCP problems outlined in Part I also apply here.

Part III:
In part I above, for UDP, one of the problems is that the maximum IP packet size is limited by the frame size of the network below. So, if you send a UDP packet bigger than a certain size, it will be fragmented (which causes a performance drop). Continue the experiment in part I (increasing the size of the packet) and try to see if you can guess the maximum UDP message size that will not cause fragmentation.

What to hand in
Hand in a CD/Floppy with your code and a README file that explains how to compile/run it and any quirks. Reasonable documentation of your code is expected but we will not grade you specifically for that. You also need to hand in a short report (soft or hard copy is fine). In your report describe your experiments, present your results (this means performance tables and graphs), and draw any relevant conclusions.

Timing Help:
Your timing will most likely be below the clock granularity level (that is the reported time may be 0). To get around that problem, you should repeat your experiment for a large number of times (inside a loop) to get the overall time to be measurable. So, if your loop runs 10000 times, you can take the measured time and divide it by 10000 to get your answer.
User Time vs. System time vs. wall time. Measuring performance is pretty tricky. There are three different time measurements that are typically used; each is appropriate for different circumstances (in general, not just for this project). User time is the time that the process spends in executing the user’s code. System time is the time the process spends spending OS code (e.g., in system calls or handling interrupts). Finally, the wall time is the overall real time that passed. This includes the execution time of other processes, and also any time that a process spends blocked/waiting.

For this project, all of these quantities are of interest. The user+system time tell us the time that sockets code spends executing. However, once the message is sent, and you are waiting to receive the message from the other side, that time can only be measured by the wall clock time (there will be some noise there, because it includes the execution time of other processes, but that shouldn’t be a big problem).

There are several ways to measure time on unix, including the system call times(). times() returns a tms data structure that includes the user time and system time ticks used by the process. To measure wall clock time, use the system call gettimeofday(). Other ways to measure time include the getrusage() call (do “man getusage”) if you don’t like this one. Here is an example of how to use times and gettimeofday():

```c
#include <stdio.h> // for printf()
#include <sys(times.h) //for times()
#include<unistd.h> // for sysconf

main(){
    int i;
    int j[10000];
    struct tms start_cpu, end_cpu;
    struct timeval start_wall,end_wall;

    int clock_ticks;
    clock_ticks = sysconf(_SC_CLK_TCK); // find out ticks/second. This can
    // be different from system to system

    gettimeofday(&start_wall,NULL);
    times(&start_cpu);//time at the beginning
    //notice loop to get time granularity up
    for(i=0;i<10000;i++) { //time something in here
        j[i]=i*92;
        printf("j[%d] is %d\n",i,j[i]);
    }

    times(&end_cpu);//time at the end
    gettimeofday(&end_wall,NULL);

    printf("TICK/SEC %d, User Time (ticks) %d, System Time %d
", clock_ticks,
        (intmax_t)(end_cpu.tms_utime - start_cpu.tms_utime),
        (intmax_t)(end_cpu.tms_stime - start_cpu.tms_stime));

    //to confuse you, unlike times(), gettimeofday() returns seconds and
    //microseconds (not ticks)

    printf("Wall Time (seconds) %e\n", (end_wall.tv_sec - start_wall.tv_sec)
        +(1.0e-6)*(end_wall.tv_usec-start_wall.tv_usec));
}
```