**­­­NETWORK**

When measuring the network's characteristics, we implemented benchmarks to measure both the loopback and remote devices for every aspect we were asked to measure. To write network related benchmarks, we used sockets to create client-server connections.

**Round Trip Time**

**Loopback Interface:**

To measure the network's round trip time for the loopback interface, we implemented a simple client-server benchmark. To keep things simple, we implemented both the client and the server in the same source file. With this approach, we don't need to run two programs on the same system and as such, our results are not subject to error because of context switching and other aspects. To write both the client and server codes in one file, we had to make the server waiting for connections asynchronously, so the program can continue into the client's code. This optimization does not alter the behavior of our benchmark, since we start measuring time long after the connection is achieved.

When a connection is established, the client sends a message to the server. Upon receiving the message, the server simply writes it back to the socket so the client can receive it back. We measure the time between the client's first send and after receiving the message back. To add more confidence to our measurement, we repeated the same procedure 1000 times, with each time measurement stored in an array. Then the array of measurements was used to compute the average time for the Round-Trip time on the loopback device. Since we are measuring the RTT, creation and initialization of the required buffers are not included in the measurements.

We also compared our benchmark's results with the results of the ping command. For a fair comparison, we set the packet size of our benchmark to 56 bytes in order to match ping's packets. According to ping's manual page, "The default [size] is 56, which translates into 64 ICMP data bytes when combined with the 8 bytes of ICMP header data".

The following table shows our benchmark's measurement for the round-trip time in cycles and in microseconds, and a comparison against ping results.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Prediction (µs) | RTT (cycles) | RTT (µs) | STDV (cycles) | Ping (µs) |
|  | 14191 | 4.05 | 526 | 23 |

**Remote Interface:**

To measure the RTT of remote interfaces, we implemented the same benchmark, however this time we created separate files for the server and client to allow us to run each one on its own machine. Again, the client connects, sends a message and received the message back. Like before, we set the message size to 56 Bytes for a fair comparison between our benchmark and ping.

To eliminate the risk of external factors altering our results, we run both the client and server programs on machines with identical specs (presented in the first chapter), both hardware-wise and software-wise.

The following table shows our benchmark's measurement for the round-trip time in cycles and in microseconds, and a comparison against ping results.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Prediction (µs) | RTT (cycles) | RTT (µs) | STDV (cycles) | Ping (µs) |
|  | 1627965 | 132.9 | 118923 | 213 |

**Peak Bandwidth**

The second part of the network evaluation was to measure our system's peak bandwidth. To achieve this we implemented a client/server benchmark similar to the one we used before, with some modifications. Since we are using identical machines, it does not make a difference on which machine the server runs or the client.

First, to measure the peak bandwidth, all we have to do is to get the time needed for the server to perform a read call on a socket. The rest is simple math. Again, we have the client sending a message to the server. The server simply measures the time it needs to read the socket. We repeat this procedure 10K times for each experiment and report the average results.

The size of data being sent should affect the measured bandwidth. Under-utilizing the available resources could lead to pessimistic measurements. To overcome this issue, we needed a methodology that allows us to saturate the connection to its limits. We increment the size of the message in steps in order to get a clearer view of our system.

**Loopback Interface**

For the loopback interface, the system's NIC is bypassed and because of that we expect the limit to be our memory's bandwidth (reported in early sections). Again for simplicity, we implemented both the client and server in the same source file to prevent context switching from taking place and altering our results.

The following table presents the measurements and the figure visualizes the measured peak bandwidth versus the size of the message. We can see that indeed the results show the peak bandwidth to be very close to the memory's bandwidth, as expected.

|  |  |  |
| --- | --- | --- |
| Prediction (GB/s) | Peak Bandwidth (GB/s) | STDV (MB/s) |
|  | 15.2 | 53.1 |

**Remote Interface**

For this case, the server and client are running on two identical machines, performing the exact same steps described earlier. The two machines are part of the same LAN, in which the routers and CAT cables support 1Gbps (128 MBps). This is much lower than the memory's bandwidth, and we expect this number to be the bottleneck of the communication.

We present our measurements in a similar way as the Loopback Interface part of this experiment.

|  |  |  |
| --- | --- | --- |
| Prediction (MB/s) | Peak Bandwidth (MB/s) | STDV (MB/s) |
| 128 | 117.8 | 2.95 |

**Connection Overhead**

To measure the time required to setup and tear down a connection, we implemented very similar benchmarks as those used so far for our network analysis. This time, we measure the time that the server was waiting for a connection (Setup time) and the time required to close the connection socket (Tear-down time).

**Loopback Interface**

We used the client/server benchmark described in the previous sections, with both the server and client in the same source file. For the setup time we measure the time difference between the non-blocking connect instruction (client) and right after the accept instruction (Server). To ensure that the server is always waiting for a connection and it doesn't skew our measurements, we repeated the connection procedure 10K times and ignored the first hundred measurements as a warm-up period.

The tear down time is easy to measure, since it only involves closing the connection socket. We are measuring the time the server needs for one close (socket) instruction.

The table below presents our results.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Prediction(µs) | Overhead (cycles) | Overhead (µs) | STDV (cycles) |
| Setup |  | 33284 | 9.5 | 335 |
| Tear down |  | 26997 | 7.71 | 224 |

**Remote Interface**

With the server and client benchmarks running on two identical machines, we measured the connection setup time as the time before and after the blocking connect instruction of the client. Like in the loopback interface, we included a warm-up phase, to ensure that the server does not alter our results. In a loop, we create a connection from scratch and close it right before the end of the loop. This allows us to measure both the setup and tear down time in one run.

The setup/teardown procedure was repeated in a loop for 10K times for more accurate measurements. In our results, we present the average values. The table below presents our results.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Prediction(µs) | Overhead (cycles) | Overhead (µs) | STDV (cycles) |
| Setup |  | 2666817 | 762 | 20463 |
| Tear down |  | 23423 | 6.7 | 198 |

**FILE SYSTEM**

**File Cache Size**

File cache is used to temporarily store files accessed from the disk in order to service future requests on the same files faster. Our goal was to measure the size of this file cache. To achieve this goal, we implemented a benchmark that repeatedly reads a file (in a loop). We expect subsequent reads to be serviced from the file cache, unless the file size is larger than the amount the cache can store.

We created temporary files of sizes ranging from 0.5 GB to 14 GB (in 0.5 GB increments), which we fed to the benchmark and measured the bandwidth obtained from reading the files. The temporary files were omitted from the delivered files of this project due to their large size.

Our system has 32 GB of main memory and as such, we expect the file cache size to be in the order of Gigabytes. This is the reason we had to create so large input files. The following figure plots the measured bandwidth against the input file size.

As we can see from the plot above, files with size less than 4GB measured a very high bandwidth, while the rest of the files provided an extremely low (in comparison) bandwidth. This suggests that the size of the file cache is equal to 4GB.

**File Read Time**

To measure the bandwidth of our hard drive, we had to disable file caching. We achieved that wth the use of the O\_DIRECT flag when opening the file. This experiment is very similar to the previous one. A file is opened and read multiple times. The average bandwidth is reported in our figure below.

We should note that since we are not utilizing the file cache, we expect reading times to be very large. To make our benchmark more efficient, we are now working on much smaller file sizes (128KB - 1.5MB) and since the file size is so small, we edited our benchmark to generate the file of the desired size on-the-fly.

One of our goals was to compare the impact of sequential and random file access. For the former, after a file is opened with the O\_DIRECT flag, we sequentially read blocks of 4KB. To create random access patterns, we use a random number generator to offset the next access to a random page.

For the case of sequential access, we observe that the measured bandwidth remains roughly the same. Since we are accessing the file in 4KB segments, the file size cannot affect the performance. However, this is not the case for random accesses. The reason is that as the file size grows, a random access pattern will increase the overall seek time required. We can observe how the bandwidth degrades for larger file sizes and random accesses, while for small files the effect is small.

Even though our experiment reports a constant bandwidth for sequential accesses, in theory it is possible to observe different results. If a file is not stored sequentially due to disk fragmentation, it can have the same degradation as the random access. When moving to even larger sizes, the possibility for fragmentation increases.

**Remote File Read Time**

To measure the remote file read time, we used a NFS server and performed the same experiment as earlier for both sequential accesses and random. This time, the file sizes we used ranged from 128 KB to 3 MB. Again, file cache was disabled. The following figure demonstrates the obtained bandwidth.

We observe that the bandwidth in this case is limited by the network. The machines of this server are connected with a 100 Mbps Ethernet cable and as such it can provide a maximum bandwidth of 12.5 MBps. Our benchmark's results are near the connection's limit. We estimate that the reason the bandwidth is not closer to 12.5 MB/s is possible contention by other jobs running on the server.

**Contention**

As a final step in our evaluation, we had to measure the system's reaction under contention. Contention occurs when multiple processes compete one another for the same resource. To obtain this metric, once again we disabled the file cache and measured the time required to read a 4KB block off the disk, while multiple processes are accessing the disk simultaneously.

In our benchmark, each "contention" process creates a 64KB file and reads it indefinitely in an infinite loop. This way we ensure that each process will constantly be accessing the disk. The main process is responsible for reporting the results. In parallel with the "contention" processes, the main process accesses a 4KB block off the disk for a thousand times and reports the average time required.

In this experiment, we varied the number of "contention" threads from 0 to 9. The following figure demonstrates the results. As expected, performance degrades as more processes access the disk simultaneously.