# Bootcamp Week 5: Performance testing

This week, we will perform a load test of your HTTP server to measure its capacity. Before you begin, please review the lectures pertaining to performance testing (31-33) from Design and Engineering of Computer Systems.

## Checking for Memory leaks

Memory leak occurs when programmers create a memory in heap and forget to delete it. The consequences of memory leak is that it reduces the performance of the computer by reducing the amount of available memory. Eventually, in the worst case, too much of the available memory may become allocated and all or part of the system stops working correctly, the application fails, or the system slows down vastly. Memory leaks are particularly serious issues for programs like servers which by definition never terminate. Since performance testing requires running the server under high load, a server with memory leak(s) would likely run out of memory.

Valgrind is a tool used to check for memory leaks. Refer to the Memory Leaks using Valgrind to learn about Valgrind and do some practice exercises. Then, use Valgrind to remove memory leaks from servers written by you in the last three weeks.

## Closed-loop load generator client

To measure the capacity of your web server, you will first build a load generator client to rapidly fire requests at the server. Begin with writing a simple HTTP client that sends a HTTP request to the web server. You can add HTTP request generation logic to your simple echo client, for example. Next, modify this client to act as a closed-loop load generator, i.e., load is generated from a certain number of concurrent emulated users. You are not expected to modify your server in any way, except removing memory leaks or fixing any bugs that may arise during the testing.

Your load generator will be a closed-loop multi threaded program, with the number of concurrent users/threads, think time between requests, and the duration of the load test specified as command line arguments. A simple template is provided here, which you may choose to use. Each thread of the load generator will emulate a HTTP user/client, by sending a HTTP request to the server, waiting for a response from the server, and firing the next request after the think time. You can use a pre-defined set of URLs to request from each emulated user, and ensure that the corresponding resources are available at the server as well. The load generator threads also need not display the received HTTP responses. After all the load generator threads run for the specified duration, the load generator must compute (across all its threads) and display the following performance metrics before terminating.

1. Average throughput of the server, defined as the average number of HTTP requests per second successfully processed by the server for the duration of the load test.

2. Average response time of the server, defined as the average amount of time taken to get a response from the server for any request, as measured at the load generator.

After writing the code for your load generator to generate load and compute throughput/response time statistics, you must run a load test on your server. Run multiple experiments by varying the load level (i.e., number of concurrent load generating threads) at the load generator. You can experiment with different think times as well to see what works well. In the end, you must generate plots of the average throughput and response time of the server as a function of the load level. You must then use these performance graphs to estimate the capacity of your server, and identify the resource causing the performance bottleneck.

## Load testing guidelines

A few things to keep in mind when running this load test:

1. Each experiment at a given load level must run for at least a minute to ensure that the throughput and response time have converged to steady state values. Your plots for throughput and response time should include at least 5 throughput/latency measurements at 5 different load levels, and must show the server reaching saturation capacity.

2. If all goes well, you will notice that the average server throughput initially increases with increasing load, but eventually flattens out at the server's capacity. The response time of the server starts small, but rapidly grows as the server approaches its maximum capacity. At the load when the server hits capacity, you will also notice that some hardware resource (e.g., CPU or network or disk) has hit close to 100% utilization.

3. You may use a client and server running on one or more CPU cores for your capacity measurement. It is highly recommended that you perform your load test with multiple cores assigned to the server, in order to fully test that your server can saturate multiple cores with its threading model. You may use as many worker threads as needed to saturate all the cores of the server. Of course, assign resources based on the availability of machines at your end.

4. It is recommended that you use two separate (physical or virtual) machines to host the server and the load generator, to easily separate their resource usages. If you cannot manage two separate machines, you must at least pin the server and the load generator processes to separate CPU cores, e.g., using the `taskset` command. Without a clear separation of client and server resources, your results of the load test will not make any sense.

For example, you may want to pin the server and load generator processes to (virtual) CPU cores 0 and 1, respectively. (`taskset -c 0 ./server 8000` and `taskset -c 1 ./load_gen localhost 8000 100 0.5 60`).

5. When you find that the server's throughput has flattened out, you must verify that the server's capacity is limited by some hardware bottleneck (e.g., CPU or network or disk). If you find that the throughput of your server is flattening out even with no apparent hardware bottleneck, you must investigate why your server is not able to handle more requests. Perhaps your load generator is not generating enough load, or your server does not have enough worker threads to handle all the requests coming in. Or perhaps you are printing out too much debug output to the screen, causing the server to wait for I/O most of the time. You must carefully debug your experiments until you are convinced that you have really saturated some hardware resource at your server. You can use tools like `top` or `iostat` to measure the utilization of various hardware resources at the server.
   For example, you may want to use `top` to measure the CPU utilization of the server. (`top -p <pid of server>`).

6. Note that the throughput and response time you measure will be a function of how many files your server is serving, what your think time is, and so on. For example, if your server is serving only a small set of files, the files may be served mainly from the disk buffer cache. But if the server is serving a large set of files, it may have to go to disk more often, resulting in lower throughput and higher response time. Therefore, you must carefully interpret your throughput and response time measurements to convince yourself that they make sense.

7. You may assign as many hardware resources (CPU cores, memory etc.) as required to your load generator, in order to ensure that it is capable of generating enough load to saturate the server. You must ensure that the system whose capacity is being measured (the HTTP server) is saturated by some hardware resource, while the system that is generating load (the load generator) is not saturated and is able to generate enough load.

8. You may find that your server cannot process all client requests, especially at high loads. You must carefully write your code to gracefully handle all possible failure scenarios that may occur under high loads. For example, if a load generator thread fails to connect to the server, it must retry again before proceeding to send requests.

## Submission

Push your load generator code as well as your load test results to your GitHub repository.