Josh Hellauer Project 3 Write-Up

CS 1550

Dr. Jon Misurda

Threaded vs Process-driven HTTP server

To compare the speeds of both the two approaches to a concurrent server, a simple experiment can be conducted. The experiment will involve three Linux machines on a peer-to-peer Ethernet network with no central router or gateway, including one HTTP server and two HTTP clients.

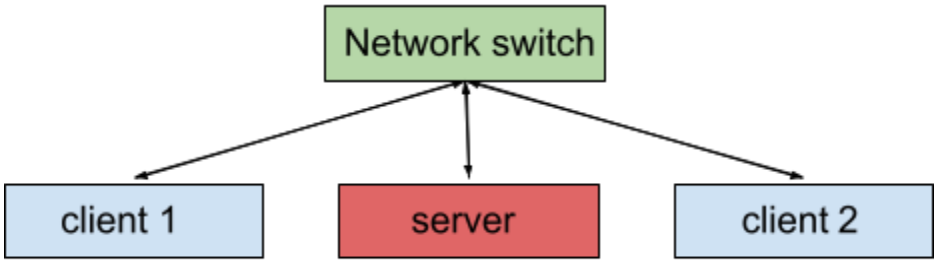


Figure 1.1 Network Topology

For each test run, the server machine will run the respective server program. Then, both clients will simultaneously run a script which uses wget to make 20 sequential groups of 12 concurrent GET requests to our server for a web page (240 total requests). The client scripts will be started manually, so they won't begin at exactly the same time.

The results for three test runs of varying sizes (small: 1KB, medium: 484 KB, large: 2MB) are shown in figure 1.2.

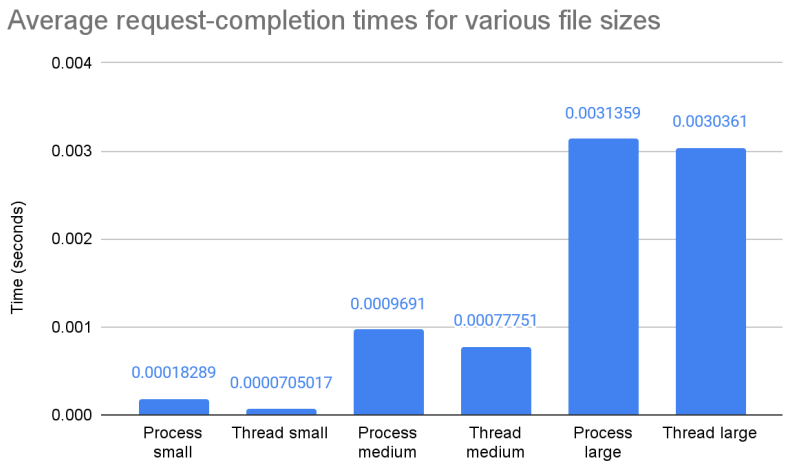
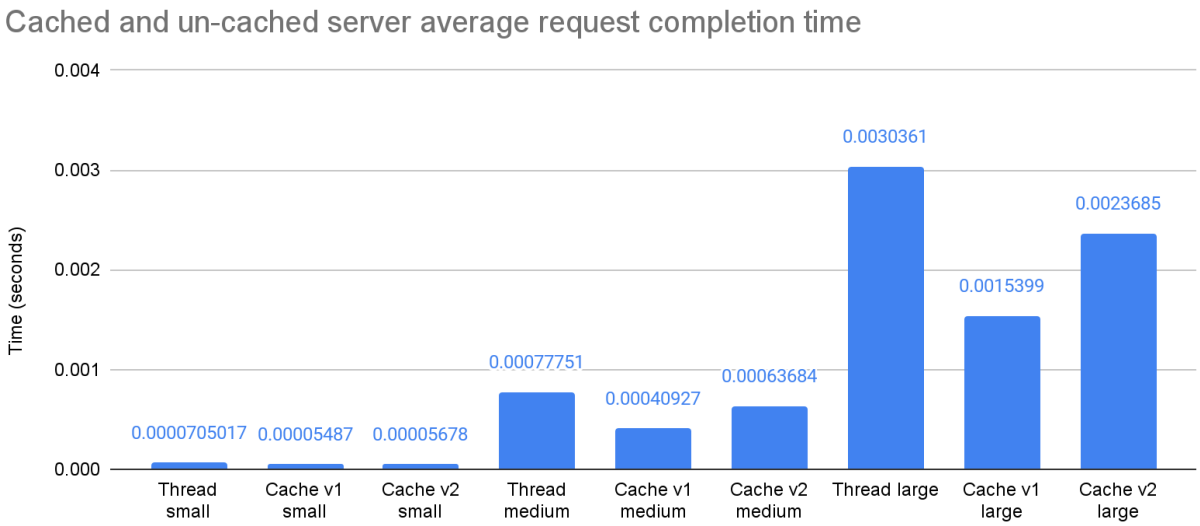


Figure 1.2

The threaded server was slightly faster. Most interestingly, the small-file size test showed the greatest disparity, which best illustrates that the amount of overhead caused by thread creation is clearly lower than the overhead caused by forking off new processes.

Cached versus Un-cached Threaded Server

We can use the same testing paradigm to compare the cached and un-cached versions of the threaded server. The results are shown in figure 2.1.



Cache v1 program uses a PriorityQueue based on last-request time to store recent HTTP responses. Cache v2 uses a linked list whose nodes have fields which indicate how many reference counts there are (how many threads are currently sending this cached response) and a valid field (indicating whether the response should be freed after use).

I would have expected Cache v2 to be faster than v1, since the reference-counting allows for responses to be sent truly concurrently. Due to naive implementation, the mutual exclusion in the PriorityQueue version only allows for 1 thread to have access to the cached responses at the same time. Therefore, only one thread can be sending data at once.

There are a few potential explanations for the naive version being faster. For one, the memory-access patterns of the naive implementation are likely simpler than the second version. Moreover, perhaps the context-switching that must occur when multiple threads are able to send data concurrently caused this increase in observed overhead. Finally, there are extra mutex interactions in the second version; there is an extra mutex lock & unlock in order to ‘put down’ the response that a worker thread was using. Since the clock isn’t started until after the cache has been searched, this additional mutex lock could have caused a slight increase in reported

time. Especially since this experiment isn’t exactly comparable to the web traffic a major web server might receive.