James Maciej Krywin

Lab 3

Discussion

**General Discussion**

Coming in from Lab 2, my implementation of a multi-threaded server was weak and inefficient. It operated by spawning a thread for each new connection that would be responsible for delegating all GET, POST, and DELETE requests to the thread pool. This was inefficient, because the program thread creation and destruction incurs a large overhead when a server has a lot of connections coming in. Thus, I decided to fully utilize the thread pool by assigning a thread from the pool for each new connection. This optimized the server by eliminating the time dedicated for instantiating new threads as described earlier. However, this modification made the server extremely vulnerable to being overloaded. If a server was instantiated with a thread pool of 4 threads and 4 clients connected to this server. The 4 threads would be occupied with listening to the connections leaving no threads behind to handle the requests.

Therefore, I modified the threads responsible for listening to the socket and handling the requests as a single action. Additionally, I modified the approach to handling the sockets. Instead of having each thread assigned to a specific connection, the threads would listen to a queue of sockets. After each request would be processed, the socket would be pushed onto the queue and the thread would pop the next socket. This strengthened the server by preventing connections from being starved and hanging.

**Lab 3 Concepts Addressed**

The first implementation of the server was fragile and crashed in situations where the number of sessions was greater than the number threads. Thus, my initial tests were limited. However, as seen in the following graph, thread times dropped when they matched the number of cores. The reason behind this occurrence is that a thread is allocated to a core. Thus, it never wasted time on waiting to be scheduled.

After testing the server with the optimizations as well as a version with the Global Lock, the yielded results demonstrated the importance of optimizing critical areas. When testing both versions of the server on NYU’s brawler - which as 4 cores. The average thread times were comparable for the first 3 threads. However,

the advantage of using read/write locks over simple mutecies began to show once the servers started using threads pools of 4 or more threads. At 5 threads, the thread times of the globally locked server were 3-4x of its faster counterpart. As seen in the followong graph, this trend continued as the number of threads increased (at 100 threads, the (g) server was 2-3 times slower).

Graph B. – Zoom in of Graph C.

Graph C.

As seen in Graph C, the general trend for a multithreaded server is that the average thread time for a request to complete increases along with the thread pool size. The culprit of this bottleneck is the limited number of cores on a machine. It can only handle 4 threads at a time. Thus as threads receive a request to handle, they need to wait for a core to become available. So, an increase in the number of threads seems to make the server inefficient. However, this is not the case, because the throughput increases. Having a healthy excess of threads helps to manage requests as they come in to the server. Instead of having a socket wait unattended in the queue of sockets, the thread can start working on a request while a working thread becomes blocked for an I/O or system call.

Graph D.

In Graph F, we address the client side statistics reported by httperf. The time per request along with thread pool size as well. However, the time starts to drop off after passing 100 threads. The reason for this trend change is that the the cpu will have the additonal threads to work on while a thread goes through I/O operations as well as any other protntail system calls and interrupts. Here, this trend is most visible for the server without read/write locks for the KV stores as it enables reader parallelism for GET requests.