Implementing a Web Server Using a Thread Pool

David Moon

1 Introduction

For my web server project, I implemented a web server using a fixed-size thread pool. This approach combines the multi-threaded and event-driven architectures to get the best of both worlds: (1) high performance via concurrency, and (2) graceful degradation via queueing of tasks in memory.

2 Architectural Overview

At a high level, my server consists of a shared, thread-safe queue; a greeter thread that listens for connections and pushes them onto the queue as they arrive; and a large set of worker threads that pull connections off of the queue and handle the incoming HTTP requests. Figure 1 shows the flow of connections.

The greeter thread sets the timeout of each connection. For my timeout heuristic, I set 5 second as a lower bound and add a number of seconds roughly equal to the number of enqueued connections divided by 50. Intuitively, the size of the queue seems like a good measure of server load, in which case we would want to grant each request more leniency so that they may make it through the queue.

There are about 25 worker threads per core—a number on which various voices on the Internet seem to agree is roughly optimal. Upon `dequeue`-ing a connection, a worker thread checks for a timeout, receives and parses the incoming HTTP request, then sends the appropriate response. Finally, if the request used HTTP/1.0, the worker closes the connection; otherwise, it re-enqueues the connection. For simplicity, I assume my server only receives HTTP requests without any content body.

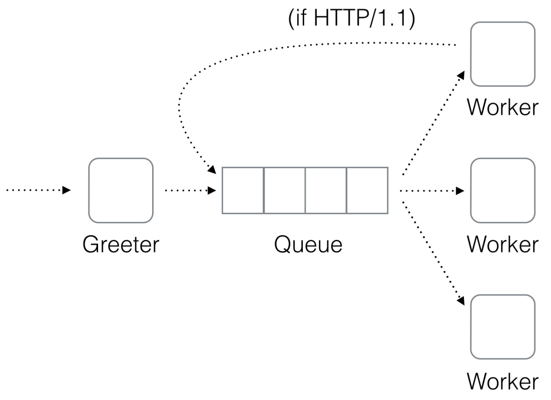

Figure 1: Connections are accepted by the greeter; enqueued; dequeued; parsed and handled by workers; and possibly re-enqueued for further handling.

The queue implementation is based off of a simple, yet fast queue algorithm presented by Scott et al. in [1]. The queue uses two locks, a head lock and a tail lock, to enforce mutual exclusion among concurrently `dequeue`-ing threads and among concurrently `enqueue`-ing threads, respectively. Unlike most other thread-safe queue implementations, which tend to use a single lock to protect all accesses of the queue, this design supports concurrent `dequeue` and `enqueue` operations.

This is made possible by having the queue maintain a dummy node that points to the actual head of the queue and acts as a buffer between a concurrent pair of `dequeue` and `enqueue` operations. The enqueue blithely allocates a new node with the appropriate data, then adds it to the tail of the queue. The dequeue is more cautious: first, it peeks at the next node behind the dummy node. If there is none, then the queue is empty and so it returns empty-handed; if there is a node, the dequeue may safely read its value because no enqueue touches a node in the queue once it has been added. Then the old dummy is freed, and the node whose value was read becomes the new dummy.

Where a dequeue on Scott et al.'s queue immediately returns in the case of an empty queue, I modify their implementation so that the dequeue-ing thread (non-busily) waits until an enqueue-ing thread signals it. This keeps the worker threads out of the way of the thread scheduler during low load, but immediately available when needed.

3 Evaluation

As hoped, my server appears to scale relatively well and degrades gracefully. Consistently, it was able to complete 100,000 requests perfectly at a rate of up to 800 simultaneous requests sent at a time. The plot in Figure 2 shows in blue its performance in milliseconds per request on those 100,000 requests as the rate of concurrency goes up, as measured by the ab benchmarking tool. The sysnet server's performance on the same benchmarks is shown in green. My server was able to handle greater rates of concurrency as well at the cost of some (∼100) incomplete requests, but for whatever reason, ab does not display performance metrics on imperfect completions.

As shown in the plot, my server scales competitively until it hits about 600 simultaneous requests, at which point its performance degrades and becomes more erratic. My guess is that the bottleneck lies in the single greeter thread.

Most producer-consumer queues such as this also specify an explicit maximum size limit; if the limit is reached, then any enqueue-ing threads must symmetrically wait until signaled by a dequeue-ing thread. I decided not to impose such a limit beause: (1) it was simpler given my implementation, and (2) I thought it would be more efficient to keep the greeter thread's routine streamlined for

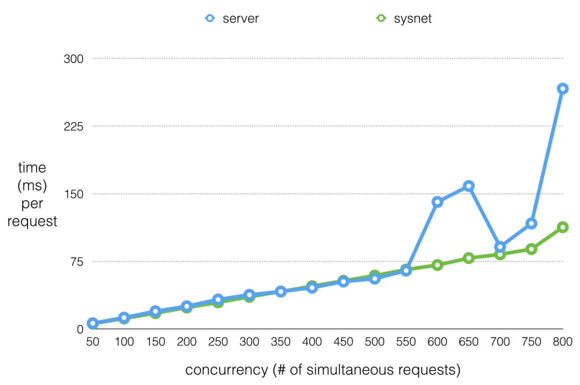

Figure 2: My server compared to sysnet on time per request versus number of simultaneous requests made at a time. (Sorry for the size.)

times of high load. In a future version, I would add such a limit and support for multiple greeter threads; multiple greeter threads widen the bottleneck, while having such a size limit would allows the greeter threads to take a role in sending error messages in case of overloading.

4 Discussion

1. Currently, my single greeter does no parsing of the incoming message and immediately pushes it onto the queue, which seems important in the case of many incoming connections. In order to implement a filtering layer using .htaccess files, I could add another queue to my server. The greeter threads would push incoming connections into the new queue; new filter threads would do a quick parsing of relevant headers, close any unwanted connections, and push the rest onto the existing queue; and the worker threads would continue to operate as usual.

2. The serialism of HTTP/1.1 is not so bad if the pipelines are kept relatively full, and other connections with other clients can be processed in parallel. However, this points to when HTTP/1.1 may be sub-optimal: when the pipelines are sparse, and there are relatively few of them. In the case of my server, for example, this would dilute the flow of incoming data at the cost of extra space in the

queue. That said, these issues could be handled by being more careful with timeouts. Furthermore, it seems to me that relatively worse performance under a light load is a perfectly acceptable cost for better performance under a heavy load.

References

[1] M. Scott and M. Michael. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. *PODC*, 267-275, 1996. `http://www.cs.rochester.edu/~scott/papers/1996_PODC_queues.pdf`