A web server with an event-driven architecture and thread pooling

David Moon

1 Introduction

For my web server project, I implemented a web server with an event-driven architecture and thread-pooling. This approach combines the multithreaded and event-driven approaches to get the best of both worlds: (1) high performance via concurrency, and (2) graceful degradation via queueing of tasks in memory.

The server supports both HTTP/1.0 and HTTP/1.1. As the focus of this project was server performance rather than general usability, however, the server only supports enough of each protocol to server the static webpage at `http://sysnet.cs.williams.edu`.

This paper describes my implementation and performance evaluation. Load tests using Apache's `ab` benchmarking tool and 20 Amazon EC2 instances indicate that my server successfully handles bursts of at least 1200 concurrent requests at a time. On these particular tests, my server performs comparably to a commercial-grade Apache server (version 2.4).

2 Architectural Overview

Greeters and workers At a high level, my server architecture takes the form of a multi-producer-multi-consumer queue. Greeter (producer) threads listen for connections and push them onto the queue as they arrive; worker (consumer) threads pop connections off the queue and handle the incoming HTTP requests. Figure 1 shows the flow of connections.

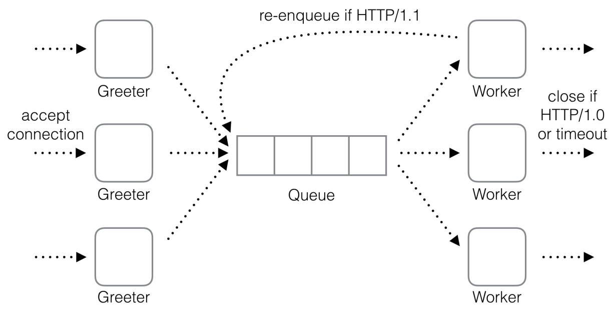


Figure 1: Connections are accepted by the greeter; enqueued; dequeued; parsed and handled by workers; and possibly re-enqueued for further handling.

The total thread pool consists of 25 threads per core—a number which various voices on the Internet seem to agree is roughly optimal. These threads are distributed evenly between the greeters and the workers.

Along with accepting and enqueueing connections, the greeter threads set the connection timeouts. For each accepted connection, the responsible greeter sets 10 seconds as a lower bound and adds a number of seconds roughly equal to the number of enqueued connections divided by 50 at the time of acceptance. Intuitively, the size of the queue seems like a good heuristic 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.

Upon dequeueing 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.

Concurrent queue The queue implementation is based on 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` in 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 Performance Evaluation

Experimental Setup To evaluate my server's performance, I used Apache's benchmarking tool `ab` to run 30-second bursts of HTTP requests at varying concurrency levels and recorded response times on the client side. Every request was a GET request for the directory index file (417 bytes) of the static webpage `http://sysnet.cs.williams.edu`.

The `ab` tests were distributed over 20 Amazon EC2 micro-instances. `ab` allows the user to vary the concurrency level at which requests are sent, such that a concurrency level of 5 roughly simulates 5 independent clients; as the concurrency level goes up, however, the accuracy of this simulation is naturally limited by the hardware of the machine on which `ab` is run. Distributing `ab` tests over many machines mitigates this inaccuracy and better simulates heavy loads. Using 20 EC2 instances, I could simulate 1000 simultaneous clients with only a concurrency level of 50 on each instance. In this set of tests, I simulated up to 1200 concurrent connections. Running `ab` on EC2 instances also allowed me to take into account the effect of long-distance connections: the EC2 machines were located in Oregon, while my server ran on a machine in Massachusetts.

For comparison, I ran the same set of tests on an Apache server (version 2.4). Both my server and the Apache server were run on a Tempest T2H Workstation featuring an Intel Core i7 3.1GHz quad-core processor and 32GB RAM. Both sets of tests were run between 9 and 11 pm on Saturday, July 25 2016.

Results Figure 2 shows the servers' mean response times per request. Figure 3 shows the servers' 50th, 75th, and 90th percentile response times per request. My server's measurements are shown in red, the Apache server's measurements in blue. In every plot, each point marks the relevant statistic taken over all requests sent from a single EC2 instance at a fixed concurrency level over a 30-second burst. All the points for a given EC2 instance are connected by a line. The concurrency level refers to the "total" concurrency level, i.e., 20 times the `ab` concurrency level on each instance.

My server successfully responded to every request in these tests. As shown in Figure 2, it appears to perform comparably to the Apache server. However, it does not scale gracefully as the Apache server does on increased concurrency levels. On additional concurrent connections, my server begins to hit system limits with respect to open file descriptors.

Another feature worth noting is the strange oscillation in my server's performance as the concur-

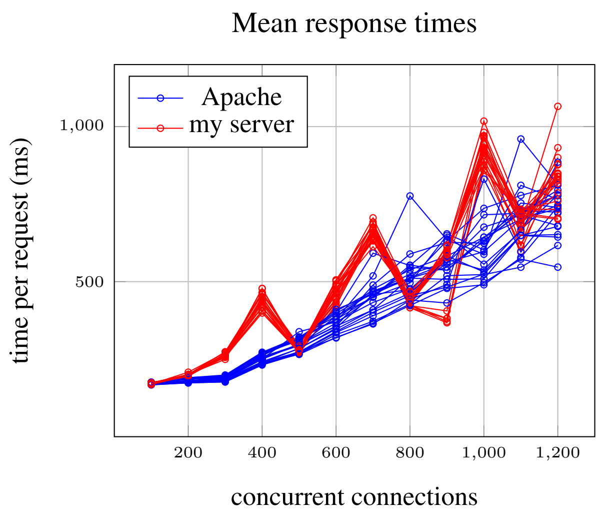
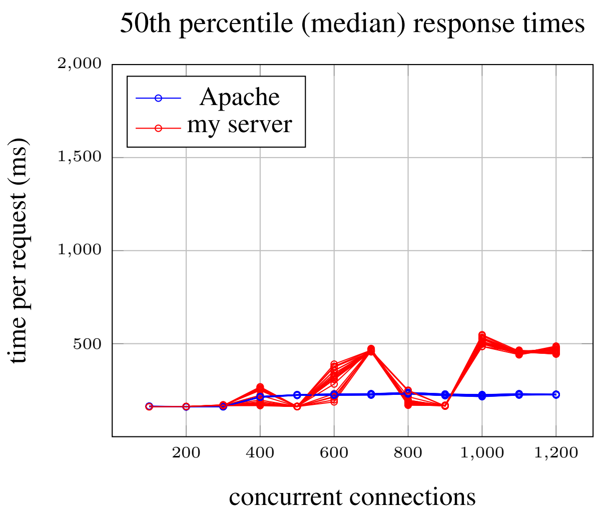


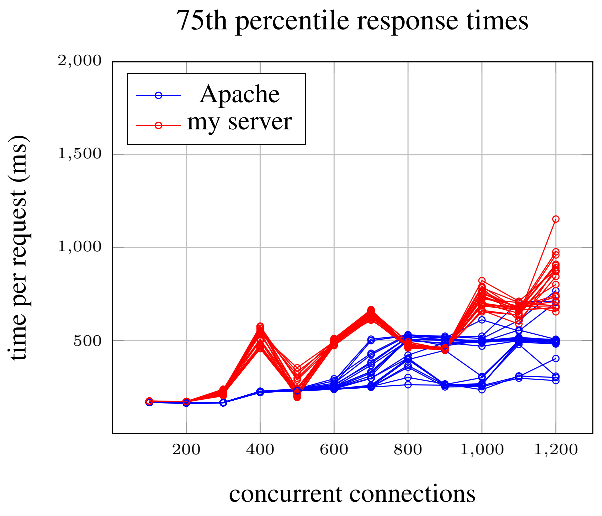
Figure 2: Mean response times per request.

rency level goes up. My suspicion is that the upward spikes are due to network issues on individual requests in combination with my use of blocking sockets. If a `recv` call on a particular request blocks for a long time, then responses to requests that arrive directly after it are significantly delayed. In a future iteration, I hope to explore the use of nonblocking sockets.

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`





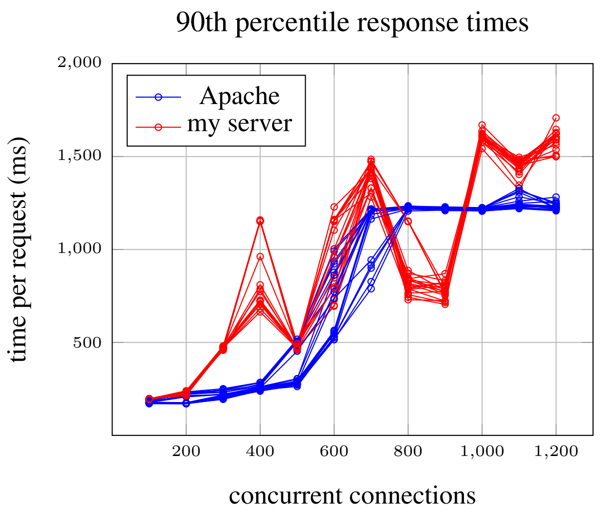


Figure 3: Percentile response times per request.