KV511 Project 1 Performance Evaluation

Richard Heidorn, Michael Wheatman

November 2, 2016

1 Overview of the Program

The client program for KV511 begins as a single, "master" thread which reads in a configuration file and spools up a variable number of "worker" client threads that each send a number of sessions to the server via an API. The client program can currently work in a single mode that can be treated as two types. The first type generates one worker thread and executes a set of 10 sessions, each consisting of 10 random GET and POST requests to the server. The second type generates N worker threads, where N is defined by the configuration file, and each worker executes 10 sets of 10 sessions simultaneously to the server.

Like the client program, the server program also begins as a single master thread, and as incoming connections arrive from the client, the server handles each request and returns the results to the client. The server can work in two ways: multithreaded, by spooling up a new server worker thread to handle each incoming connection (which is a single session of 10 requests from the client), or asynchronously, by handling each connection in tandem using a single-threaded, non-blocking asynchronous model.

2 Performance Evaluations

The first round of evaluations were carried out by examining the performance of the single-threaded, asynchronous server versus the multithreaded server. We ran four key performance experiments:

1. Compare the average time to service a single client session between asynchronous and multithreaded servers for both Type 1 (single client thread) and Type 2 (multiple client threads)
2. Compare the average time to service all of a client's sessions between asynchronous and multithreaded servers for both Type 1 and Type 2. This is the similar to the prior, but includes inter-session waiting time.

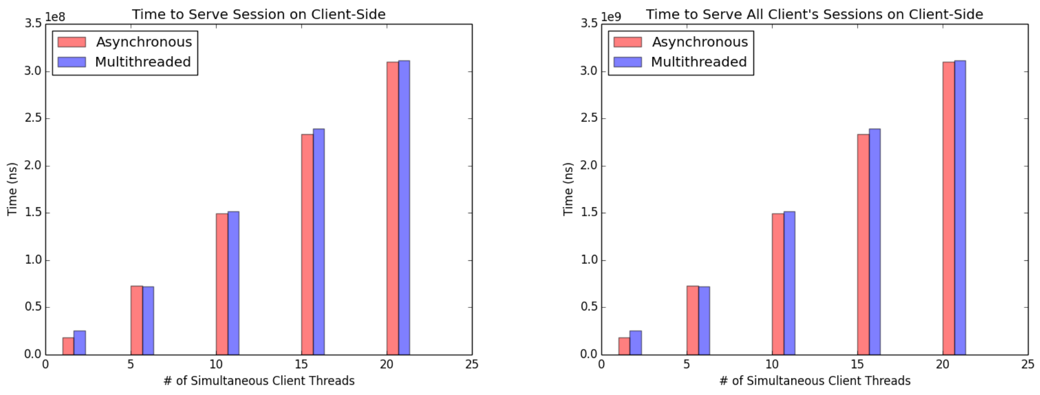

Figure 2.1: Average times to serve a single client session of 10 requests (left) and the average times to serve all 10 of a client's sessions (right).

3. Compare the average time that the server takes to handle a single socket from open to close (with interleaving / concurrent connections in the case of multiple simultaneous clients)
4. Analyze the average time to service a single message on the server side

The above evaluations were chosen to best draw comparisons between the multithreaded and asynchronous applications in the broadest sense. The major differences between MT and AS methods include:

1. Overhead of thread creation and cleanup (multithreading)
2. Overhead of context switching (multithreading)
3. Simultaneous servicing (multithreading) v.s. sequential servicing (asynchronous)

While far more evaluations can be run, we felt that the above best encapsulated the essence of the project. We are looking forward to contributing more advanced and detailed analyses for the latter half of this project.

3 Performance Results

For the first experiment, we found that the average total time to service a single client's session increased linearly as the number of clients increased. Further, we found that the data for a single session or a full suite of sessions was nearly identical in trends (only increased in time by roughly 10, as we perform 10 sessions for each client thread).

The average time that the server experiences when managing a socket is more telling. When using the asynchronous model, we find that the time to service a socket (the time from opening to closing the socket) is nearly half of the time to service the socket in the multithreaded

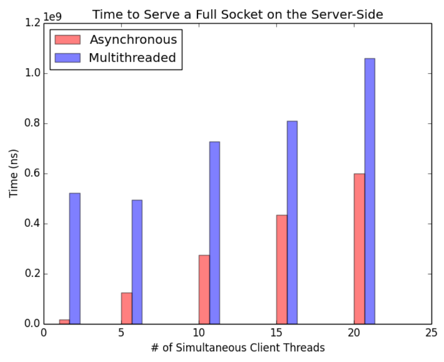

Figure 3.1: Average times for the server to handle a socket from open to close. Notice that while an asynchronous socket takes less time, the client still experiences roughly the same time for either server.

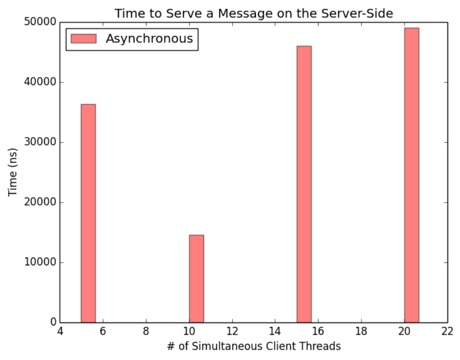

Figure 3.2: Average times for the server to handle a single message within a session. The data retrieved for the second bar is probably a fluke.

version. We suspect that this is measuring both the overhead of thread creation and some effect of context switching.

Of particular interest is the slightly less time that multithreading takes for 5 simultaneous clients from a single client - as a result, we would like to explore how the multithreaded server performs for small numbers of simultaneous clients across multiple cores. Further, we note that the total time that the client experiences for both multithreading and asynchronous models remain the same - we believe this has to do with the client's own overheads and context switching, as well as networking latencies and "waiting for a spot to open" on the server.

Finally, we've included the average time it takes the server to handle a single message using the asynchronous model. This data is rough, as the timing methods may have been too coarse-grained for the fine-grain function call to access the data store. We plan to refine this analysis, as it is a great place to analyze lock contention for the multithreaded server.