Katherine Schneider
4/3/2015
ECEN 4003
Project Checkpoint

Introduction

For this course's final project, I chose to implement a concurrent web server, written in Java. The server should be able to serve multiple client requests at one time, using the HEAD and GET methods. The web server will serve static file resources, and will record the IP address of each client that accesses it, as well as the files requested, in a shared data structure that must be accessed concurrently by the various threads. Files served will be of varying length, to test for noticeable differences between the serial and concurrent server versions (since differences may not be substantial with smaller file sizes). Additionally, the server will be tested with varying numbers of client requests, to see how it performs when there is a spike of client activity. Both individual request processing time and throughput will be tested, as these are both important metrics in different settings.

The goal of this project is to complete a multithreaded web server that uses a fine-grain or lock-free implementation to control each thread's access to a shared data structure. This implementation of the server should provide significant speedup when compared to a sequential server, especially in extreme cases, such as when there are a large amount of client requests at one time, or when files requested are of larger file size. In working toward a fine-grain locking implementation, I am building several versions of the server: first, a sequential server which can only process a single client request at a time, then a coarse-grain locking implementation, which uses coarse locks to control access to the shared data structure (a list) which holds each client's IP address and requested file, and finally, a fine-grain locking implementation, which will hopefully provide significant speedup over the coarse-grain version.

Coding Completed

The sequential web server, which handles a limited HTTP protocol, has been coded, and serves as a standard for testing various concurrent versions' performance. The sequential server works by accepting a single request and handling that request through a requestParse method. The requestParse method parses and processes the client's request, then returns the appropriate HTTP headers and requested file (if the request was a GET). In the sequential version, this method is located within the server class itself. This method also stores the client's IP address and the name of the requested file in a list to keep a log of server activity. In the sequential server code, each request must be processed before a new request can be handled, so it is necessary that some clients must wait, especially if the current request takes a long time to process, or gets stuck somewhere.

The first concurrent version of the server has also been coded, using coarse-grain locks to control thread access to shared data structures. In this improved version, the server hands each new request off to a worker thread, which handles the HTTP request on its own (thus, the requestParse method is now in the workerThread class, rather than in the server class itself). Each worker thread also stores the client's IP address and the file requested in a shared list, access to which is controlled by coarse locking (the entire structure is locked while a single thread accesses it.) This implementation causes a sequential bottleneck when multiple threads attempt to access the list, an issue which I hope to resolve in the next server implementation, which uses fine-grain locks. Both the sequential and concurrent versions will serve three static files of different sizes to test performance in regard to size of the files served. Both versions have been tested for correctness, though full-scale performance testing has not yet begun.

Future Directions

As mentioned in the project proposal, in addition to implementing a fine-grain locking server, I would also like to investigate lock-free methods, and possibly using thread pools to increase the speedup of the concurrent server. While a fine-grain locking algorithm for shared data structures will hopefully provide significant performance improvement, I suspect that using Java's native utilities to improve concurrency will serve me better, and provide a greater speedup than an algorithm I write myself. In addition to improving concurrency, I would also like to handle a larger subset of HTTP protocol than is currently implemented, including PUT and POST requests, and test the various servers' performances when handling these new methods. This is more of a stretch goal, as increasing concurrency and investigating different algorithms is more in line with the goals of this course than is parsing HTTP.