**Servers Design & Performance Report**

**Heartbeat Monitoring Design**

I declared an interface in LoadBalancer.java for heartbeat monitoring. Any heartbeat monitoring design algorithm can be plugged in at runtime using a class that implements *LoadBalancer* and the function with signature *boolean isSystemOverloaded()* that returns true or false indicating whether the system is overloaded or not.

As a particular example of heartbeat monitoring algorithm, I used simple CPU % usage monitor (see CPULoadMonitor.java). The algorithm simply gets the load average on the system and divides it by the number of cores available to the system to get the % CPU usage. I set a threshold of 90% CPU usage to indicate that system is loaded.

**AsyncServer Timeout thread Design**

To implement timeout on a client request, I used a single thread in a ScheduleExecutorService object. I added a TimeoutScheduler object to the executor service each time a request from a client was accepted. The TimeoutScheduler was configured to fire a runnable after the specified timeout interval that would in turn add the runnable object to a queue of timed out tasks. The Dispatcher, executes all these runnable objects on run which essentially deregisters the selection keys for all clients for which timeout has occurred. The runnable objects are also removed from the queue when a request from a client has fully been received.

**Comparison of Designs (x-Socket)**

Note: Line numbers are from xSocket/core/branches/branch\_V\_2\_8/src folder.

* xSocket allows for multiple dispatchers that are put in a dispatcher pool i.e. LinkedList<IoSocketDispatcher> in IoSocketDispatcherPool.java. By default, number of CPUs + 1 dispatchers are created. IoSocketDispatcher pool takes in *size* as one of its parameters that specifies how many dispatchers should be allowed. The dispatchers are selected in a round robin manner from the pool when being assigned load (see *private IoSocketDispatcher nextDispatcher(int currentTrial)* in IoSocketDispatcherPool.java, line 97).
* The dispatcher thread’s basic flow is: register handler tasks -> perform key update tasks -> handle any read/write keys -> deregister handler tasks. The basic flow of dispatcher thread is in the run() method on line 208 in IoSocketDispatcher.java.
* The onData method of EchoHandler fires when there is data available for the server to read from the connection socket. While there is no data available to read in the connection socket, the server listens in a blocking way on the connection socket (acceptor.listen() on line 603 of Server.java whose instance is opened by EchoServer.java on line 68).
* xSocket keeps an atomic reference to a HandlerAdapter that is initialized for each non blocking connection that is started (see line 722 of NonBlockingConnection.java). The server sets idle timeout on each non blocking connection that it starts with a specified time in millis (connection.setIdleTimeoutMillis(idleTimeoutMillis) on line 1194 of Server.java). The HandlerAdapter thus terminates the connection after the specified idle timeout has occurred.
* The library does testing with an EchoServer by writing a string (“hello\r\n”) to the connection after establishing a connection from the client to the server. The client then checks if it got the correct echo response back from the server i.e. (“hello”) after the delimiters have been removed. The server can be tested with idle timeout by sending a stream of bytes without a delimiter (part of a message) and then making the client thread go to sleep for some time before it sends the rest of the message with the correct delimiter at the end. If the sleep time of the client is larger than the allowed idle timeout, then the server should drop the connection with the client.

**Benchmarking Methodology**

I used the list of requests provided in gen/request-patterns and the files in gen/doc-root. First I concatenated the request patterns list with the appropriate root for all my servers (requests file) and for the Apache server at zoo (apacheRequests file). I thereby used the same list of requests for my servers and the Apache server.

I start up all my servers on different ports. My benchmarking varies the number of client threads used during each run of the client test. The set of values used for the number of threads are: (1 2 5 10 15 20 30 50 70 100 150 200 300). For each of my servers, I run my test client repeatedly with the list of requests in requests file over the varying number of threads while collecting average throughput, average number of transactions and average wait time data for each run. I repeat the same procedure with the Apache server using apacheRequests. Furthermore, I repeat this whole procedure with two different sizes of server thread pools (5 threads and 50 threads). I used a fixed test time of 60 seconds for each run of my test client.

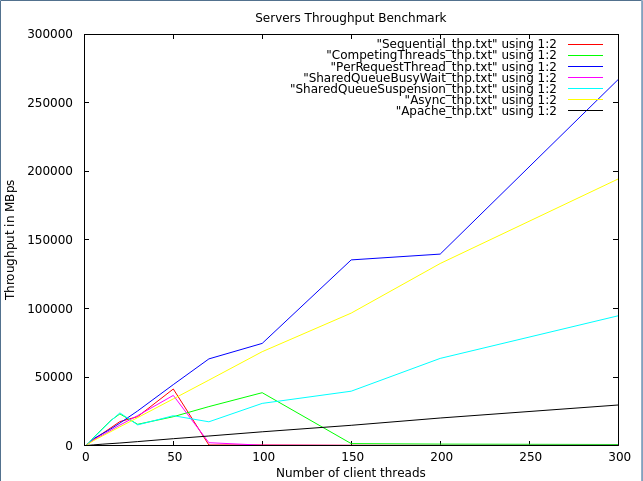
I pass in “–server node.zoo.cs.yale.edu” to my test client instead of localhost. For Apache “–server zoo.cs.yale.edu” is passed in and so I ensure equality in testing by doing so since my test client has to do DNS lookup for both cases.

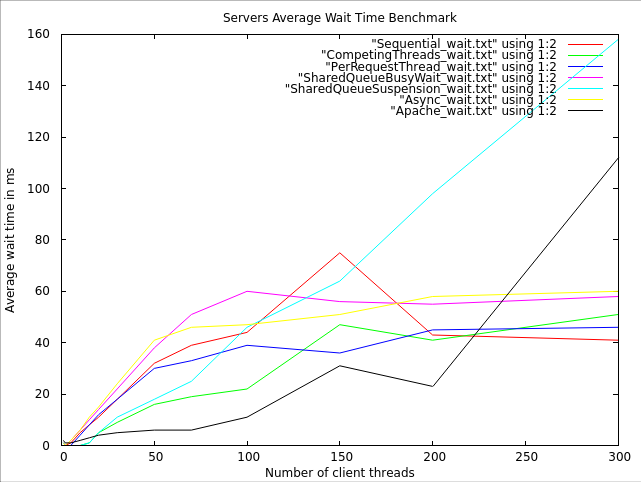
A parameter that was fine tuned for my servers in particular was the cache size. I assigned roughly 100MB of cache to each of my servers so that repeated requests could be handled efficiently.

Note: A bash script (benchmark.sh) collected all my data in the output folder. Plots were then made using gnuplot with command described in README.txt.

**Benchmarking Results**

The following plots show the throughput and average wait time comparisons between my servers and Apache when using **5 threads** for the server thread pool:





The following plots show the throughput and average wait time comparisons between my servers and Apache when using **50 threads** for the server thread pool:

