Q1.

The biggest challenge I encountered was setting interest ops for the server. Ultimately, the solution proved to be creating multiple hashmaps of SelectionKeys and ready values, e.g. (SelectionKey,  true) for when a SelectionKey's channel was already in a read state. The solution to this evaded me for some time, however, and so I kept encountering a bug where the server would create a read task and continue creating read tasks until the write interest op was set, then continue creating write tasks until the read interest op was set. This was a problem because it generated too many read and write tasks for a particular SelectionKey, so multiple threads could end up hanging, waiting for a client that was never going to write to its socket channel.

Q2.

I feel pretty good about my implementation, so there's not much I would change. I think instead of creating separate hashmaps for read and write ready states for a SelectionKey, I would have just created one hashmap with a string to indicate for which operation the SelectionKey is ready.

The other thing that I might change is the ability for a client to send messages while waiting for a message that was already sent to return to it, rather than my current implementation which has a client send a message, wait for a response, then wait for its specified message rate before sending another message. This would allow for greater throughput, I believe, and create fewer dependencies in the network.

Q3.

The program was very scalable. I hardly noticed a difference in performance of the program as the number of clients was increased. The only point at which there was an effective difference between the number of messages sent and processed was when I scaled up to around 1000 clients. The reason I think I started to see a discrepancy here was because of the relatively small number of processors existing in one machine which needed to support two threads per client, two threads for the server, and 10 threads for the thread-pool. The number of clients the system could support was relative, however, to the number of machines that were used for testing.

The reason, I think, this number is ten times higher than the number of clients requested in the assignment is because of the non-blocking-I/O implementation in the server. This allowed multiple clients to connect almost instantaneously to the server and then handle the read and write tasks with a thread-pool, so that there was almost no delay between when a client sent a message to when it started being processed.

Q4.

An alternative implementation for this problem would be to have two threadpools: one for read tasks and another for iterating through a subset of SelectionKeys. If the amount of time since the last write task on a particular SelectionKey is 3 seconds, the thread examining the SelectionKey would then initiate a write task across the channel corresponding to the SelectionKey. This would allow the server to maintain a high degree of concurrent message processing while still allowing for polling across several channels. The downside of this implementation would only arise with a relatively small number of threads because if SelectionKey's channel was ready to be written to and there were not enough threads to detect this, the channel may be written to after more than three seconds have elapsed.

Q5.

If each messaging node in the overlay was limited to 10 threads in its thread-pool and 100 concurrent connections, we would need 100 messaging nodes to handle 10,000 concurrent connections to the overlay. The messaging nodes would need to first form a chain, so that each messaging node is at most 50 nodes away from any other node. Picturing the topology as a circular chain of messaging nodes, connecting equidistant messaging nodes across the circle from each other will continually reduce the maximum number of hops between any two messaging nodes, i.e. connecting the first messaging node in the chain to the 50th one, and the 25th one to the 75th one will reduce the maximum number of hops between any two nodes to 25. Therefore, every fourth node in the circle should be connected to the node directly across the circle from itself, so that each node is at most four hops away from any other node in the overlay.