Q1:

The biggest challenge I encountered in this assignment was tracking connections. Creating and maintaining references to socket endpoints, how they correlated to a node's IP address and port number, and selecting the correct endpoint for a message were all pieces of this challenge. Part of this challenge was a socket switching its client port number when a serverSocket called its accept() method. I spent some time struggling to produce a mapping of relationship between the newly assigned port number and the original port number then how to communicate this information between nodes. Initially I tried to maintain a design where a messaging node either referenced a server's IP address and port number or it referenced a socket endpoint to decide where to send a message, but finally settled on referencing socket endpoints then mapping this information to a server's IP address and port number, so that a node could always be contacted just by sending a message to its server's IP address and port number then translating this endpoint to the socket's endpoint.

Q2:

The biggest problem with my final program was its inability to send large numbers of messages between nodes. I think this was due to over-synchronization in my onEvent() methods. The tradeoff was that without synchronization the jvm would run out of heap space. Instead of synchronization, I would implement some sort of lock that would only allow around 50 threads to access the onEvent method. If this wasn't feasible I would probably test different methods of limiting the amount of time a reference is held to the byte array created in TCPReceiverThread's run method.

The other thing I would improve is the amount of memory I used. For example, I passed a new adjacency matrix to every node to represent the link weights between nodes when I probably could have created just one adjacency matrix. I would not have had to synchronize access to it because only read operations would have occurred, so there would be no foreseeable tradeoffs to this approach.

Q3:

I would have to constantly recompute Dijkstra's algorithm, once every time a message is sent and again when it is relayed. This would obsolete a routing cache class because routes could no longer be cached. In addition, there would probably have to be some safe guard against a packet getting routed to the same nodes over and over because this could introduce a scenario where a packet moves from node A to node B, node B calculates its shortest path goes through node A, node A calculates its shortest path goes through node B and so on and so forth.

In addition, in order to improve efficiency, I would store the new link weights in an adjacency matrix shared between threads, so that the registry could acquire a lock, assign link weights, release its lock and allow the other nodes to read from the matrix. Doing this would be considerably more efficient than sending a message containing updated link weights to every messaging node every time the link weights were updated. I would also have to synchronize access to the adjacency matrix for this implementation which could slow down the program because every thread would have to access the adjacency matrix one at a time.

Q4:

It wasn't necessary to send a routing plan for every message in homework one. We could have just included the next stop for a message every time it reached a new messaging node. This approach increased time-efficiency, however, because a message would contain all of the information it needed to arrive at its destination rather than putting the work on the messaging nodes. This contrasts with the most efficient solution to the proposed problem in question three; building a routing plan for every message would be inefficient because the routing plan would just have to be recomputed every time the message reached a new node. It would be more efficient to find the next stop, send the message to the next stop, find the next stop, send the message, etc. until a message reached its destination.

Q5:

The minimum spanning tree would usually find a more inefficient route than Dijkstra's shortest path algorithm. This is because the minimum spanning tree is seeking to solve a different problem than Dijkstra's shortest path algorithm. A minimum spanning tree gives information about the shortest path that connects all nodes to one another rather than the shortest path between any two nodes given by Dijkstra's shortest path algorithm. For example, consider the case where node A is connected to B with a link weight of 5, node B is connected to node C with a link weight of 5, and node A is connected to node C with a link weight of 9; an implementation using Dijkstra's algorithm will send a message directly from node A to node C while an implementation using a minimum spanning tree will route the message from node A to node B to node C although this path's total weight of 10 is longer than the shortest weight of 9.