For our project, we implemented TCP. We first completed the networking lab to ensure that our e1000 driver was coded correctly and we understood the networking logic and lifecycle of a packet.

In order to add a working TCP, we added tcp_state to our sockets, and added arguments to the connect syscall and sockalloc in order to allow applications to specify what type of socket they want to open, and if it's a TCP socket the application can specify whether or not to just open up a port for connections or to try to initiate one. tcp_state contains the variables enumerated in 3.2 of the TCP RFC. Importantly, tcp_state also keeps track of the state in the state machine that the socket is in. The state machine is also described in 3.2 of the TCP RFC. If the socket is a TCP socket, sockalloc will initialize tcp_state. Socket reads, writes, and closes will check the type of the socket to see if it's UDP or TCP and call the correct net function correspondingly. We add net_tx_tcp and net_rx_tcp, which take in the usual arguments along with a tcp_state.

net_tx_tcp creates a tcp struct that represents a TCP packet header. The tcp struct contains all the headers as described in 3.1 of the TCP RFC. The header values are generated using the function arguments and the variables in tcp_state. In order to determine what packet to send (and if a packet should be sent at all), net_tx_tcp checks the state that the state machine is in. If the packet to be sent is just an ack, the mbuf passed in should be empty.

net_rx_tcp receives a TCP packet and parses the segment relevant headers and puts them in a tcp_info struct. tcp_info is passed to sockrecvtcp. sockrecvtcp finds the correct socket and determines how to handle the packet. First sockrecvtcp determines the state by pulling tcp_state from the socket, and from there determines if the packet is legal for the current state by checking the fields in tcp_info. There is logic for handling packets that are out of order. If packets come out of order, there's an ack for the last correct packet. sockrecvtcp wakes up sleeping processes if there is a new message and handles state updates as described in 3.1 of the TCP RFC. Our network stack handles unique cases where one end of the socket stops sending (as noted by a fin packet) but still wants to read - this logic is in sockclose.

To test our TCP, we have code for connecting two QEMU instances on the same virtual network, and we confirm that they can communicate with each other. We also connect a QEMU instance with an external server on localhost. The packet fidelity is also verified through wireshark.

Once we understood the networking lab, we were able to get a good grasp of what additions TCP required. However, there were a number of design choices we needed to make like where to store state and where to do updates. There were a number of confusing bugs related to piecing together and retrieving the information in the headers. One of the more confusing things we tackled in this project was getting two QEMU instances onto the same virtual network since we had never really touched the Makefile ourselves before.