1. **How do I test my program to see if it works over a remote connection?**

Assuming you choose VirtualBox VM as your environment for this class and already have one VM working.

If you are planning to use two VMs to demo your PA-5, do the following steps to enable networking between two VMs-

1. **Change your existing VM network Adapter:**
   - right click on your VM -> Setting -> Network -> Adapter 1
   - choose "Bridged Adapter" for '*Attached to*' option
   - under MAC Address, check '*Cable Connected*' *option*
   - save by clicking OK
2. **Clone your existing VM:**
   - right click on your VM and select *Clone*
   - must choose "Generate New MAC Addresses For All Network Adapters" for MAC Address Policy
   - proceed to Next and let the clone finish
3. **Get a new MAC address for the cloned VM (ensuring #2):**
   - right click on your cloned VM -> Setting -> Network -> Adapter 1 -> Advanced
   - click on the reset icon next to '*MAC Address*' field to get a new MAC address for this VM
   - save by clicking OK

Now if you open your VMs, open terminals in each VMs, and execute command ***ifconfig*** you should see different IP addresses for different VMs. Ping one VM from another using ping command (***ping <ip>***) and you should see ping is successful.

2. **What system calls should I be using in this PA? The lecture slides use gethostbyname() and gethostbyaddr(), but the lab slides mention getaddrinfo()?**
   gethostbyname() and gethostbyaddr() are deprecated on most platforms. getaddrinfo() (used in place of `gethostbyname`) converts text-strings which represent IP addresses to a linked list of addrinfo structures which can be further used in calls to bind and connect. Note that your implementation does not have to use getaddrinfo, it is merely one approach. getnameinfo() (used in place of `gethostbyaddr`) is the inverse of getaddrinfo(); it converts an internal binary representation of IP addresses in the form of sockaddr structures to a string containing the hostname or IP address (if it cannot be resolved to the hostname), as well as the service port number. You do not need to use getnameinfo() for this particular PA.

3. **I'm a little confused, regarding the socket, bind, listen, accept, etc. commands, are those called within client.cpp or server.cpp, or are those called within the TCPRequestChannel constructor? I'm also confused why a control channel between server and client isn't needed anymore.**
   There's still a notion of a control channel, but it only belongs to the server. It's the way the server sets up the port. Since we no longer depend on a name, but instead IP+port, we don't have to worry about pre-agreement. So the server-side will create a channel with ip_address="" and port_no=r by calling the two-parameter TCPRC constructor. Whenever the client wants to create a new channel, it calls the same constructor with ip_address=a and port_no=r. Inside that constructor, you'll differentiate between server and client, and if it's the server, you'll go through the set-up process for the server to listen; if it's the client, you'll go

through the set-up process for allowing the client to connect to the server. Once the server has the initial channel created, it then enters a loop where it continually accepts connection requests, calls the int constructor of TCPRC with the client's socket number, then dispatches a thread to handle communication through an independent channel inside of handle_process_loop.

4. **When would the server finish terminating considering `accept(...)` is blocking?**
   The server never dies. Since all client request channels are handled on threads, there's no way to tell the server to stop accepting. So you would just terminate it from the terminal.

5. **I'm slightly confused on what the *TCPRequestChannel(int sockfd)* constructor does, and how it helps to build the overall program.**
   When you accept a connection, what is returned is a file descriptor with all the information about the socket. Any future send/recv calls need to use this socket file descriptor. It already contains all the information about the connection, there's nothing more you need to do with it. The server only opens one socket (its call to TCPRequestChannel(ip_address="". port_no=r)). Then the client connects to the IP+port using TCPRequestChannel(ip_address=a, port_no=r); inside that call, you'll call connect which will transfer information about the socket to accept (assuming the server is listen-ing) and accept will store that information and return the file descriptor pointing to that information. No new port or socket is created by the server. The server just creates a new TCPRequestChannel with the socket file descriptor returned by accept by calling TCPRequestChannel(sockfd) for future calls to send/recv when it calls cwrite/cread.

6. **Should we be using read/write or send/recv?**
   Either one should work, but send/recv is the traditional choice for network programming.

7. **Since we were given default values for the flags used in PA3 and PA5 builds on PA3 code, are there/should there be default values for -a and -r?**
   You can use the *loopback address 127.0.0.1* for -a and any big enough integer (i.e., 10k to 60k) for -r as default values.
   Then again, sometimes you may see a "port already in use" error message, then you'll have to change the default value for -r into something else.

8. **Once I run the server, then terminate, then run it again, the server can no longer bind to the same port because of error 98, EADDRINUSE. How can I get around this?**
   This happens because after a server-side TCP socket is closed by the process, the operating system keeps the socket around in the TIMED_WAIT state to ensure that any remaining clients get the chance to close cleanly. This state can last up to a few minutes, but eventually the system will fully close the socket. There are two ways to get over this problem:
   - Modify your program to use a different port number whenever you get this error
   - Use setsockopt and SO_REUSEADDR to forcibly re-bind the port in the server

9. **Has anyone run into an issue where creating a raw thread(func, params) results in an abort after the thread finishes running? I solved it by assigning the thread to a temp variable but I was curious as to why it would happen.**

Make sure you are detaching the thread if you aren't planning on keeping track of it.

10. **For the report/demo, I am confused as to what the instructions mean when it talks about the ulimit parameter for Part B and the diminishing returns for Part C.**
    This refers to a system resource limit; you may have encountered this issue if you ever got a “Bad file descriptor” error when running otherwise-functional code. Basically, this means the system ran out of room when opening connections. As for diminishing returns, that's just a type of relationship - in general, just define/discuss the relationships in your graph.

11. **I understand that we do not call `bind(...)` on the client side, but does `connect(...)` first implicitly connect a port to the client's socket and then send data from that port to the server's port?**
    Yes; if you call connect() on an unbound socket, it will first attempt to bind the socket (it will choose an arbitrary free port in the upper portion of the range – an *ephemeral port*). This is because all sockets must be bound locally before they can be used to send/receive data. For a TCP socket, its entry in the system can be thought of as four numbers: (local address, local port, remote address, remote port).