CPE 464 Lab
Socket Programming – TCP and poll()

See canvas for due date.

Note – For this lab program, you can work with others to write this code. You must type and turn in your own lab program – I am ok with seeing someone else's lab code for this lab, but you must type in and debug your own code.

Also, I provide some code for this lab, you can/should use any code I provide! For any code I provide, you can use it as is or modify it to meet your needs. You can use this code for both the labs and programs. Please leave my name in the code but use it as you wish.

Steps (details below):

A. Getting started - download and compile my code and verify it works for you.
   a. In the Makefile change the names to cclient and server
   b. I would also change the code file names to cclient.c and server.c
B. Read (section II below) about the application PDU you are going to create and use in program #2.
C. Write (section III) functions to sendPDU() and recvPDU() which send/recv application PDUs.
D. Modify my server code to continually receive PDUs until you ^c the server.
E. Modify the server code to handle multiple clients using poll().
F. Modify the client to handle the server terminating.
G. Modify the client to use poll() to handle both the socket to the server and stdin.

I. Getting started.

If you have not already done so, download the tar file from Canvas. This tar file should in both the client and server code, networking code, poll code and Makefile.

1. Compile the myclient/myserver code provided and verify that the client talks with the server.
   a. Modify the Makefile to produce executables called **cclient and server** (these are the executable names needed for program #2)
   b. Modify the .c files to be cclient.c and server.c to match the executable names

II. Application Level PDU Discussion

In the next part of the lab, you are going to implement the process for sending and receiving an application PDU in the following format. This is an application level PDU which includes a two byte header and then the payload.

Discussion of this application PDU:

- PDU: Two-byte PDU length in **network order**, then the payload. For this lab the payload will be a null terminated text message (in program #2 there are different types of payloads).

PDU Length (2 bytes) (header) (in network order)	Payload (Null terminated text message for this lab)

- The PDU length is for the entire PDU (length of header+payload). The PDU length in the PDU must be in network order.
- You are creating a simple application level PDU (2 byte length header + payload)
- As discussed below, the entire PDU must be sent in one send() call
- As discussed below, all messages (both on the client and server) must be recv()ed in **two recv() calls** using the **MSG_WAITALL**

III. Implementing (in a new .c and .h file you create sendPDU() and recvPDU().

Now you are going to implement two functions in new .c/.h files that handle sending and receiving an application level PDU with the format. These functions prototypes, are:

- int sendPDU(int clientSocket, uint8_t * dataBuffer, int lengthOfData);
- int recvPDU(int socketNumber, uint8_t * dataBuffer, int bufferSize);

Note – the dataBuffer passed in/out of these functions does NOT include the length field of the PDU. The only place the length field is available is in the sendPDU()/recvPDU() functions.

- Sending Side: Create the PDU and send the PDU. Return value is **data bytes sent**.
- Receiving Side: recv() the PDU and pass back the dataBuffer. Return value is **data bytes received**.

These functions must be in files (.c and .h) separate from my code and your main code.

- **sendPDU**: Implement a sending function that send()s an application level PDU in one send(). This function should create the application level PDU, send() the PDU in one send() call, and check for errors on the send() (<0). This function must be in a separate .c file from your client and server code (and must be declared in a .h file). You MUST use this function for all send().[1]

[1] Note – if you use my safeSend() and safeRecv() calls, these already check for < 0 errors on sending and receiving. So these functions will never return a -1 value to you.

Function prototype:

int sendPDU(int socketNumber, uint8_t * dataBuffer, int lengthOfData) ;

- NOTE – DO NOT assume the dataBuffer is a C null terminated string – it might be for this lab but assume it is a data buffer and use memcpy() in the sendPDU() function. You **cannot** use strcpy() or any string (e.g strncpy()) function in the sendPDU() function. You MUST use this function for all sends in both the lab and program #2.
- The return value is the number of **data** bytes sent (so not including the 2-byte length field).

- **recvPDU**: Implement a receiving function that recv()s an application level PDU. This function includes checking for recv() errors (return value <0), checking for closed connections (return value==0) and does the two step recv() process (e.g. using MSG_WAITALL). This function must be in a separate .c file from your client and server code (and must be declared in a .h file). You MUST use this function for all receives in both the lab and program #2.

Function prototype:

int recvPDU(int clientSocket, uint8_t * dataBuffer, int bufferSize);[2]

- NOTE – In the recvPDU() function DO NOT assume the received data is a C null terminated string – it might be for this lab but assume it is a data buffer and use memcpy() in this function. You **cannot** use strcpy() or any form of str functions in your function.
- You need to use the MSG_WAITALL on both recv() calls. The first call to receive 2 bytes, the second call to receive the rest of the PDU.
- The return value is the number of **data** bytes received (so this is just the number of bytes received in the second recv()), and 0 if the connection was closed by the other end (this means you received 0 bytes). Note – is possible the first recv() receives 0 bytes. If that is the case you should NOT do the second recv(). If you recv() 0-bytes, then the connection was closed by the other side.
- The return value from this function does not include the two bytes of PDU length. (So if all goes well it returns the number of bytes recv()ed in the 2nd recv(). Or 0 if the connection is closed.
- You need to check that buffer (using bufferSize) is large enough to receive the PDU. If not, you should print out an error and exit the program (this means you have a bug since you are writing both the client and server and this should not happen).

[2] The bufferSize should be used for error checking. If the length of the PDU as defined by the first two bytes of the PDU is greater than bufferSize then you wrote a bug in your program (so printf an error message and exit(-1)).

a) Testing your new recvPDU/sendPDU functions

i. Starting with my server code, modify the server code to use your new recvPDU() function. When the server receives data from a client print out the message (e.g. Message received on socket 5, length: 6 Data: hello

ii. Starting with my client code, modify the client code to use your new sendPDU() function.

iii. Test/Make sure your new recvPDU/sendPDU functions work before continuing.

IV. Modify the server to loop until ^c kills the server

i. If you didn't do this in your testing above, modify the server to loop infinitely. The only way the server terminates is with a ^c. No cleanup is needed when ^c happens, so do not handle SIGINT!!

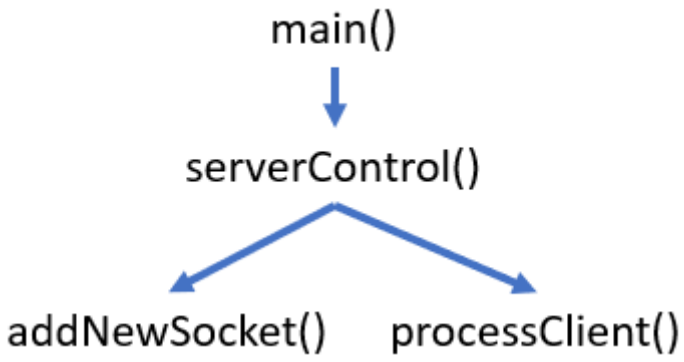
V. Modify the server to use poll() to handle multiple clients

i. Make a copy of your code from the previous part of the lab.

ii. This step builds off the code you finished in the steps above

iii. Modify the server so that it loops and accept()s new clients while at the same time recv()ing packets from currently active clients. To do this implement a function called serverControl() that implements the following pseudo code:

- At the beginning of the program, add main server socket to poll set.
- While (1)
  - Call poll() (e.g. use my poll library)
  - If poll() returns the main server socket, call addNewSocket().
  - If poll() returns a client socket, call processClient().



iv. Write your functions addNewSocket() that processes a new connection (e.g. accept(), add to poll set()).

v. Write your processClient() which calls recvPDU(), and then outputs the message.

vi. When the server receives data from a client print out the message

(e.g. Message received on socket 5, length: 6 Data: hello)

vii. The server processing must be done in a single Unix process (so no threads, no fork()). You must use poll() on the server to process the multiple sockets. Feel free (I recommend) to use my poll() library. My poll() library is part of the tar file you downloaded for this program.

VI. Modify the server to handle a client termination (e.g client ^c).

Note - There are two return values from the recv() call that we can use to figure out the other side has terminated.

First, if you recv() 0 bytes (so recv() returns 0) on a TCP socket, this means the socket has been closed by the other end. Since you cannot send 0 bytes, most kernels use recv() returning 0 as an indication that the other side closed the connection. Second, if recv() returns -1 and errno = ECONNRESET then the other side has closed the connection (I think this is only true on iOS but we can always check for it). If you look at my safeRecv() code in safeUtil.c you will see that I check for this and return 0 if this condition (recv() returns -1 and errno = ECONNRESET) happens.

On the server, when the server detects that a connection has closed have the server close the client socket and remove the client socket from the poll set. The server should continue to run after this (so call poll again). It is important to do both things (close the socket and remove the socket from the poll set). (Note – in program #2, the server would also need to remove this socket from the handle table).

VII. Modify the client to handle ^C or the server terminating

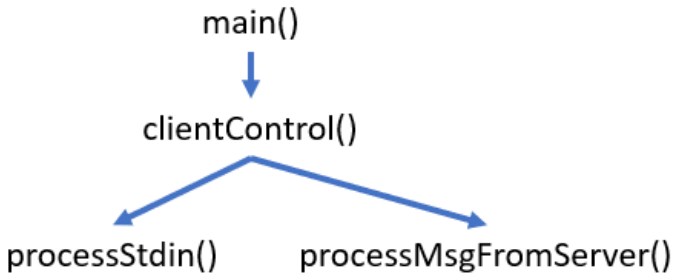
Modify your client code to continuously loop until the user hits ^C. You do not need to do any clean up on the client after the ^C. So DO NOT handle SIGINT. Just terminate your client program.

Also, modify your client to terminate if it receives 0 bytes on the socket with the server. This means that the server has terminated. Print out the error message: "Server has terminated" and terminate the client program.

VIII. Modify the client to handle both user input and the server terminating

Modify the client to handle both STDIN and incoming packets on the socket connection with the server at the same time using poll(). Remember that STDIN_FILENO is a file descriptor and works with poll().

Your client should look like:



Where the clientControl() function calls poll() with a poll set containing both the socket number to the server and STDIN_FILENO (remember that stdin is just an open file descriptor and poll() can monitor it for input, just like poll() can monitor a socket number which is also a file descriptor).

a. Modify your client to continuously loop using the control structure in the figure above and accept user input and process messages from the server until the user hits ^c on stdin or the server terminates. (You should have done this in the last step.)

b. To test your processMsgFromServer() modify the server to send back the exact PDU it received. So on the server, receive a PDU and for testing only send this PDU back to the client.

c. Have your client terminate with a message: "Server terminated" when the server terminates and then have the client exit. To test this ^c your sever and the client should then cleanly terminate. If the client infinitely loops at this point, you are not correctly handling the server termination in your client code.

This client/server/sendPDU/recvPDU code should help you start program #2.

You need to handin in your code from this lab part 2 (client code and server code that uses poll())) including the Makefile, client and server code and the network code (and any other code we need to compile these two programs, so even include my network code .c/.h).

Handin using the **lab_sockets** directory e.g.: **handin csc-cpe464 lab_sockets** *.c *.h Makefile README

Checklist before turning in your code:

A. You have written and are using sendPDU() and recvPDU() for all sending/receiving.
B. Modified the server code to continually recv() PDUs until ^c terminates the server.
C. Modify the server code to simultaneously handle multiple clients using poll().
D. Modify the client to handle the server terminating.
E. Modify the client to use poll() to handle both incoming packets from the socket with the server and user input on stdin.