Programming Assignment 4

The Server Moved Out!

Reminder:

Please, take a few minutes of your time to evaluate the course anonymously on **https://tamu.aefis.net/ - Your feedback is valuable**.
Note: the deadline to submit your course evaluation is December 6th, 2023.

Note: ⚠️ This PA requires manual testing. Once local tests pass, bring it to the lab for final grading.

Due date

Nov 28, 2023 11:59 PM

Academic Integrity

The following actions are strictly prohibited and violate the honor code. **The minimum penalty for plagiarism is a grade of zero and a report to the Aggie honor system office.**

- Uploading assignments to external websites or tutoring websites such as Chegg, Coursehero, Stackoverflow, Bartleby, etc.
- Copying code from external websites or tutoring websites such as Chegg, Coursehero, Stackoverflow, Bartleby, etc.
- Copying code from a classmate or unauthorized sources and submitting it as your work.

Keywords

TCP/IP, sockets, Multithreading, producer-consumer relationship, race conditions, overflow, underflow, mutual exclusion.

Introduction

In this programming assignment, you add a class called **TCPRequestChannel** to extend the IPC capabilities of the client-server implementation in PA3 using the TCP/IP protocol. The client-side and server-side ends of a TCPRequestChannel will reside on different computers.

Since the communication API (not just the underlying functionality and features) through TCP/IP is different from FIFO, you will also need to restructure the server.cpp and client.cpp as part of this programming assignment:

- **The server program must be modified** to handle incoming requests across the network request channels using the TCP/IP protocol. The server must be able to handle multiple request channels from the client residing on a different computer.
- You must also modify the client to send requests across the network request channels using the TCP/IP protocol.

Requirements

In this assignment, you will restructure your PA3 client.cpp in order to accomplish data point and file transfers using TCP/IP request channels. You will also rewrite your server.cpp so that multiple instances of the client program can connect to the server simultaneously.

The client will take in two additional mandatory arguments compared to PA3.

- “-a” which denotes the IP address the server is running on.
- “-r” which denotes the port number the server is deploying its services to.

The following command line options are valid for PA4:

For data point transfers: a, r [with optional n, w, b, p, h, m arguments]

```
./client -n <# data items> -w <# workers> -b <BoundedBuffer size> -p <# patients> -h <# histogram threads> -m <buffer capacity> -a <IP address> -r <port no>
```

For file transfers: a, r, f [with optional w, b, m arguments]

```
./client -w <# workers> -b <BoundedBuffer size> -m <buffer capacity> -f <filename> -a <IP address> -r <port no>
```

The server must be called in the following form:

```
./server -r <port no> -m <buffer capacity>
```

- "-m" is an optional argument whereas "-r" is a mandatory argument for the server.
- The server no longer runs using the exec() function from the client. It is run separately from a terminal on a different computer.
- The server executable does not need the ip address argument because it runs the service in the current host. It just needs to deploy its services on the user-assigned port.
- The client, on the other hand, needs to know where the server is running (i.e., IP address) and at which port number.

Implementation Note

A TCPRequestChannel class was added to replace the FIFORequestChannel class from PA3. The following matches the class declaration (comments explaining functions in starter code):

```
class TCPRequestChannel {
private:
      int sockfd;

public:
      TCPRequestChannel (const std::string _ip_address,
                         const std::string _port_no);
      TCPRequestChannel (int _sockfd);

      ~TCPRequestChannel ();

      int accept_conn ();
```

```
        int cread (void* msgbuf, int msgsize);
        int cwrite (void* msgbuf, int msgsize);
};
```

You should provide definitions for the functions declared above in a TCPRequestChannel.cpp file. Also, you will need to make changes in the server.cpp and the client.cpp programs to implement the desired functionality for realizing TCP/IP connection. For instance, the server should run an infinite loop to accept() incoming connections from the client and create a new thread with the accepted socket to process subsequent communication using the handle_process_loop function (which should stay mostly unchanged except for the input argument type - it should now be TCPRequestChannel* instead of FIFORequestChannel*).

The client on the other hand, can simply call the respective constructor w times to create all the channels. There is no need for sending a NEWCHANNEL_MSG to the server because calling the constructor results in a separate dedicated channel. Due to this reason, the client does not need a separate control channel either.

- Create TCPRequestChannel.h/.cpp from the template code above.
- Fulfill requirements of network socket connections in TCPRequestChannel
- Modify client.cpp/server.cpp

1. Server creates TCPRequestChannel(ip_address="", port_no=r)
2. Server enters an infinite loop to establish connections with TCPRequestChannel::accept_conn() and TCPRequestChannel::TCPRequestChannel(int sockfd)
3. Client creates channels with TCPRequestChannel(ip_address=a, port_no=r)

Getting started

1. **You must have a functioning PA3 as a baseline for this project.**
2. Go to the assignment's GitHub classroom: https://classroom.github.com/a/fd4GDL2J
3. Create a repository for your project.
4. Starter video: https://www.youtube.com/watch?v=3OBf3bJUrrY
5. Additional video (audio does not work well, sorry about that): https://www.youtube.com/watch?v=R7FdkkEFI4A

Rubric Instructions:

Code Requirements [100 points]

- Remote network communication over sockets. (Not tested by autograder, manually graded, with partial credit)
- [30 points] Datapoint requests work locally but not to a server (either LAN or WAN or

both)

   a. Test by using 127.0.0.1 as server's IP
- [30 points] File transfer requests work locally but not over a network (either LAN or WAN or both)

   a. Test by using 127.0.0.1 as server's IP
- [20 points] Datapoint requests work both locally and over either LAN or WAN or both

   a. Test with TA
- [20 points] File transfer requests work locally and over either LAN or WAN or both

   a. Test with TA
- You can test yourself by having two separate VMs, computers, or GCP instances. If you don't, bring it to lab for manual grading.

- If you are testing with two computers, you can use "ipconfig"/"ifconfig" to get the ip address first and then try "telnet" to see whether you can connect to other computers. If you can't connect to other computers, it might be the problem of the firewall and you might need to configure your firewall rules.
- A reminder for testing with GCP instances: because by default google cloud vm only allows http(port 80)/https(port 443) on external ip, if you are binding other ports, you would need to set up a separate firewall rules. Or you can create another vm, and let two machines communicate through **internal** IP address.

- Basically, you can test with "make test" first. If you pass all the local tests, you can do the server-client test with two machines. If your code functions well with two machines, then it's highly possible that you can pass the manual test with TA.

Q&A

1. Since we were given default values for the flags used in PA3, and this 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.

2. Once I run the server, then terminate it, 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

3. 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.

4. For the 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.

5. 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).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.

6. 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.

7. 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.

8. 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.

9. Should we be using read/write or send/recv?

   Either one should work, but send/recv is the traditional choice for network programming.