**Exercise I.a.1** The listening socket is bound to a specific address. What address is this? (Give

both the symbolic name used in the code, and the corresponding IPv4 address in numeric or

dotted notation).

**Solution I.a.1:** The IP address of the server executing the code. INADDR\_ANY is the symbolic name while 127.0.0.1 is the IPv4 in dotted notation

**Exercise I.a.2** In the code, there is a call to **recv** () as follows:

ret = **recv** (cd.sock, cd.buffer, kTransferBufferSize, 0 );

The return value ret will be one of the following:

(a) ret = -1

(b) ret = 0

(c) 0 < ret < kTransferBufferSize

(d) ret = kTransferBufferSize

Describe the implications of each case! Additionally, why is cd.buffer (see ConnectionData

declaration) defined to be of size kTransferBufferSize+1 rather than just plain

kTransferBufferSize?

kTransferBufferSize+1 keeps track of the buffer size to determine when it’s full or empty.

Solution **I.a.2:** Option d will the return value.

1. ret = -1 implies that an error occurred [the socket is nonblocking]. it sets errno to indicate the type of error [EAGAIN or EWOULDBLOCK].
2. ret = 0 implies orderly shutdown [end of file return].
3. 0 < ret < kTransferBufferSize implies that the number of bytes received from stream socket is between zero and 64.
4. ret = kTransferBufferSize implies that recv function received 64 bytes from stream socket.

**Exercise I.a.3** Sending is performed using the **send** () method as follows:

ret = **send** (cd.sock,

cd.buffer+cd.bufferOffset,

cd.bufferSize-cd.bufferOffset,

**MSG\_NOSIGNAL**

);

How does the **send** () method indicate that the connection in question has been closed/reset?

How does **MSG\_NOSIGNAL** relate to this (on Linux machines)?

**Solution** **I.a.3**

A. The **send** () indicates a closed/reset connection by clearing/zeroizing the buffer and changing the connection state to receiving.

B. **MSG\_NOSIGNAL** prevents generation of **SIGPIPE** signals if an attempt to send is made on a closed or reset connection [EPIPE is returned].

**Exercise I.a.4** Two different strategies are used to handle errors. During setup, the server program will exit with an error code after encountering and reporting an error. In the loop that handles incoming

connections, errors cause the server to stop processing the active client (and close its connection if necessary).

Discuss the reasons for this behaviour with your partner. Why are these two

strategies used?

Also, quickly look through the error codes (values of **errno**) possible after **accept** (), **send** (),

and **recv** () (check the man-pages!). Under which conditions attempting to continue execution

might be unreasonable?

**Solution** **I.a.4**

A.

// set up listening socket - see setup\_server\_socket() for details.

int listenfd = setup\_server\_socket( serverPort );

if( -1 == listenfd )

return 1;

int clientfd = accept( listenfd, (sockaddr\*)&clientAddr, &addrSize );

if( -1 == clientfd )

{

perror( "accept() failed" );

continue; // attempt to accept a different client.

}

The exit strategy during set up—listen ()—ensures that the server program terminates if it’s unable to accept incoming connections. On the other hand, the exit strategy used in the incoming connections—accept ()—allows the server to accept pending incoming connections on the queue.

B.

It will be unreasonable to continue execution under the following error conditions:

1. EAGAIN or EWOULDBLOCK – Socket is nonblocking and there’s no connection to accept.
2. ECONNABORTED – terminated connections.
3. EINVAL - No listening socket connections, or address size is invalid/negative.
4. EMFILE - The per-process limit on the number of open file descriptors has been reached.
5. ENFILE - The system-wide limit on the total number of open files has been reached.
6. ENOBUFS, ENOMEM - Not enough free memory. This often means that the memory allocation is limited by the socket buffer limits, not by the system memory.

**Discussion with Partner**

In the client code, the input buffer is defined to be 256 bytes:

**const size\_t** kInputBufferSize = 256; Thus, the largest query a client can send is around 256 bytes.

On the server, the size of the transfer buffer is much smaller:

**const size\_t** kTransferBufferSize = 64; What happens if the clients sends a message larger than 64 bytes? Try it! (You may temporarily reduce the buffer size on the server if 64+ bytes is too long of message for your taste!) **if a client sends a message lager than 64, the client keeps the message in its buffer and then breaks the message into smaller bytes before sending the message to the server.**

The reason why large buffers are split in the server is fairly simple. When you recv you provide a preallocated buffer and since TCP is a character oriented reliable streaming you only read whatever you can fit on it. Afterwards you read the rest of the buffer on another recv. Depending on the OS it may even return data from the next TCP packet if it is already there!

|  |  |  |  |
| --- | --- | --- | --- |
| **Message Sent By Client from remote12.chalmers.se** | **Number of Bytes Sent** | **Response from Server on remote11.chalmers** | **Number of Bytes Received & RRT** |
| suggestion each processor reads the secret of every other processor and puts it in its own r-register. after that, each processor reads the r-register of every other processor and assembles a frequency table for the values it found. the frequency of that processor's own secret will be the only one with a frequency not divisible by (n-2)problem: size of r depends on n (linearly) If no messages are available at the socket, the receive calls wait for a message to arrive, unless the socket is nonblocking (see fcntl(2)), in which case the value -1 is returned and the external variable errno is set to EAGAIN or EWOULDBLOCK. The receive calls normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount requested. The socket is nonblocking and the connection cannot be completed immediately. (UNIX domain sockets failed with EAGAIN instead. It is possible to select(2) or poll(2) for completion by selecting the socket for writing. After select(2) indicates writability, use getsockopt(2) to rean the SO\_ERROR op‐tion at level SOL\_SOCKET to determine whether connect() completed successfully (SO\_ERROR is If no messages are available at the socket, the receive calls wait for a message to arrive, unless the socket is nonblocking (see fcntl(2)), in which case the value -1 is returned and the external variable errno is set to EAGAIN or EWOULDBLOCK. The receive calls normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount requested. Test it and see what happens to the server when the request is | 264 | suggestion each processor reads the secret of every other processor and puts it in its own r-register. after that, each processor reads the r-register of every other processor and assembles a frequency table for the values it found. the frequency of that '  processor's own secret will be the only one with a frequency not divisible by (n-2)problem: size of r depends on n (linearly) If no messages are available at the socket, the receive calls wait for a message to arrive, unless the socket is nonblocking (  see fcntl(2)), in which case the value -1 is returned and the external variable errno is set to EAGAIN or EWOULDBLOCK. The receive calls normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount  requested. The socket is nonblocking and the connection cannot be completed immediately. (UNIX domain sockets failed with EAGAIN instead. It is possible to select(2) or poll(2) for completion by selecting the socket for writing. After select(2) i  ndicates writability, use getsockopt(2) to rean the SO\_ERROR op‐tion at level SOL\_SOCKET to determine whether connect() completed successfully (SO\_ERROR is If no messages are available at the socket, the receive calls wait for a message to arrive, u  nless the socket is nonblocking (see fcntl(2)), in which case the value -1 is returned and the external variable errno is set to EAGAIN or EWOULDBLOCK. The receive calls normally return any data available, up to the requested amount, rather than wai  ting for receipt of the full amount requested. Test it and see what happens to the server when the request is | 42  round trip time is 0.483663 ms  45  round trip time is 41.588267 ms  43  round trip time is 39.871551 ms  37 bytes  round trip time is 40.103334 ms  38  round trip time is 39.914059 ms  42  round trip time is 39.937993 ms  21 [109]  round trip time is 39.938151 ms |
|  |  |  |  |
|  |  |  |  |

**Exercise I.b.1** The multi-client emulator can attempt to establish many TCP connections concurrently. For this, it performs a non-blocking connect.

Discuss with your partner: How is the program notified that a connection

attempt has failed or succeeded?

Hint: the process is described in the course book!

**Solution I.b.1**

**if( connErrors > 0 )**

**{**

**printf( " %zu errors while establishing connections\n", connErrors );**

**if( connErrors == numClients )**

**{**

**printf( "All clients errored. Bye\n" );**

**return 1;**

**}**

**}**

**printf( " successfully initiated %zu connection attempts!\n",**

**numClients-connErrors );**

When there are no connection errors [connection is zero] it notifies the program that connection is successful otherwise it returns it compares the numbers of errors with the number of clients and if equal it returns all connections as failed. However, if one client fails it returns an error but will proceed with other clients established.

**Exercise I.c.1** Try to send messages with each of the clients. Describe the results – do you

receive a response immediately?

Check with netstat and document the status of the connection from each client.

**Solution** **I.c.1**

I sent a string of 9 bytes from the first client plus another 44 bytes of data and received a response immediately. I also received a message” response **does match original query**” after each receipt. On the other hand, I sent a 43 bytes’ message from the second client but did not receive a message.

The netstat command shows that both clients and the server connections where established with different process id’s [12851 for the first client, 9054 for the second client and 3041 for the server].

**Exercise I.c.2** When you disconnected the first client, what happened? Explain why.

**Solution** **I.c.2**

On disconnection of the first client [using CTRL+D key], the server performed an orderly shutdown of the socket connection and established a connection to the second client on the same socket—socket 4. The netstat command did not show the process id of the first client. Further, after disconnection of client 1, the 43 bytes sent from the second client, sent 0 bytes to the server and the server returned zero bytes as well.

Subsequent messages sent to the server after disconnecting the first client returned same message immediately.

socket 4 - orderly shutdown

Connection from 127.0.0.1:48396 -> socket 4

**Exercise I.c.3** Measure the round trip time when the client and server are running on the

same machine. Also measure the round trip time when they are on different machines.

Can you observe any differences? Write down the times. (Note: take the average of a few

(> 5) attempts.)

**Solution I.c.3**

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Sent** | **Round Trip Time (milliseconds – ms)**  **Client & Server on Same Machine** | **Round Trip Time (milliseconds – ms)**  **Client & Server on Different Machine** | **Remarks** |
| This is a test sent to the server – 33 bytes | 0.084463 | 5.936388 | It was faster when client and server are on the same machine |
| I am enjoying this lab work – 27 bytes | 0.100951 | 9.161370 |  |
| It is really nice connecting to chalmers remote server – 54 bytes | 0.083655 | 4.706442 |  |
| This is a test sent to the server – 33 bytes | 0.092891 | 11.162449 |  |
| Test Data – 9 bytes | 0.108161 | 10.770205 |  |
| I am enjoying this lab work – 27 bytes | 0.061767 | 9.626789 |  |
| It will be a nice experience studying in Chalmers Sweden. Looking forward to the experiencing the culture – 105 bytes | 0.071855 | 29.402143 |  |

**Exercise I.c.4** Measure the round trip times for two concurrently connected simple clients

(similar to exercise I.c.1 ).

Discuss with your partner: What is the largest factor in the measured round trip time of the

second client?

**Solution** **I.c.4**

To handle concurrent connections the iterative server program has to be modified to accept simultaneous connections from clients.

Two concurrently connected clients on the same machine with server table

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Sent** | **Round Trip Time (milliseconds – ms)**  **Client 1** | **Round Trip Time (milliseconds – ms)**  **Client 2** | **Remarks** |
| This is a test sent to the server – 33 bytes | 0.192737 | 279481.412377 |  |
| I am enjoying this lab work – 27 bytes | 0.077390 | 0.132229 |  |
| It is really nice connecting to chalmers remote server – 54 bytes |  |  |  |
| This is a test sent to the server – 33 bytes |  |  |  |
| Test Data – 9 bytes |  |  |  |
| I am enjoying this lab work – 27 bytes |  |  |  |
| It will be a nice experience studying in Chalmers Sweden. Looking forward to the experiencing the culture – 105 bytes |  |  |  |

**Exercise I.d.1** Run the above command (make sure that the server is still running), and note

the results. Increase the number of clients a few times (try, for example, using 10, 15, 30, 50 and 100 clients).

What happens to the minimal/maximal times required to establish a connection? What happens

to the round-trip times? Did any errors occur?

Solution **I.d.1**

Both Minimal, maximal and RTT times increases as the number of clients increases. Yes we noted the below errors.

Establishing 100 connections...

setsockopt(TCP\_NODELAY) failed: Invalid argument

setsockopt(TCP\_NODELAY) failed: Invalid argument

2 errors while establishing connections

Connection reset by peers

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Client | Successful | Min | Max | Average |
| 10 | 9 | 1.418700 | 2.571400 | 1.949156 |
| RTT | 9 | 114.292300 | 1023.320700 | 581.152667 |
| 15 | 15 | 1.075200 | 3.655600 | 2.101660 |
| RTT | 15 | 134.901100 | 1861.569600 | 1041.374573 |
| 30 | 30 | 3.236800 | 7.000700 | 4.888853 |
| RTT | 30 | 175.116600 | 175.116600 | 2268.378723 |
| 50 | 50 | 1.765800 | 7.075200 | 4.057298 |
| RTT | 50 | 218.468300 | 8594.946600 | 4930.216040 |
| 100 | 98 | 4.984300 | 20.691400 | 10.377661 |
| RTT | 98 | 391.958000 | 25383.484900 | 15996.084690 |

**Exercise I.d.2** Take note of the timing results. You will want to compare them to results in

the next Lab/Exercise.

(You don’t have to hand in the results, though.)

Reduce the number of clients to 7, and instead increase the number of messages (try e.g. 1000,

5000, 10000 messages). Compare the timings as you did above.

Now, run with 100 clients and 10000 queries. This is also an excellent time to grab a cup of coffee.

Take note of the times that are reported. You’ll want to compare them to the ones you will take

in Lab 3.

Finally, you’re going to perform a (very simple) denial of service attack. Connect a simple client

to the server and make sure that the server is handling that client actively. **Note the current**

**wall-time!** Use the multi-client to open a large number (> 50) connections to the server.

At some point, connections of the multi-client should start to time out. This may take several

minutes. **Write down the time when the connections timed out!**

Solution **I.d.2**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Client | Message | Successful | Min | Max | Average |
| 7 | 1000 | 7 | 2.275400 | 3.099700 | 2.622500 |
| RTT | 1000 | 7 | 77.091100 | 489.108200 | 287.111643 |
| 7 | 5000 | 7 | 0.700000 | 1.578500 | 1.129414 |
| RTT | 5000 | 7 | 276.984800 | 1885.952900 | 1101.111057 |
| 7 | 10000 | 7 | 3.245200 | 3.715000 | 3.469457 |
| RTT | 10000 | 7 | 516.560200 | 3512.151400 | 2045.082329 |
|  |  |  |  |  |  |

**Exercise I.d.3** How long did it take for the connection attempts to time out?

Solution **I.d.3**

**Appendix**

**Recv Function**

If no messages are available at the socket, the receive calls wait for a message to arrive, unless the socket is nonblocking (see fcntl(2)), in which case the value -1 is returned and the external variable errno is set to EAGAIN or EWOULDBLOCK. The receive calls normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount requested.

**Recv Function RETURN VALUE**

These calls return the number of bytes received, or -1 if an error occurred. In the event of an error, errno is set to indicate the error. When a stream socket peer has performed an orderly shutdown, the return value will be 0 (the traditional "end-of-file" return). Datagram sockets in various domains (e.g., the UNIX and Internet domains) permit zero-length datagrams. When such a datagram is received, the return value is 0. The value 0 may also be returned if the requested number of bytes to receive from a stream socket was 0.

**Connect Function error - EINPROGRESS**

The socket is nonblocking and the connection cannot be completed immediately. (UNIX domain sockets failed with EAGAIN instead.) It is possible to select(2) or poll(2) for completion by selecting the socket for writing. After select(2) indicates writability, use getsockopt(2) to read the SO\_ERROR op‐tion at level SOL\_SOCKET to determine whether connect() completed successfully (SO\_ERROR is zero) or unsuccessfully (SO\_ERROR is one of the usual error codes listed here, explaining the reason for the failure).

suggestion each processor reads the secret of every other processor and puts it in its own r-register. after that, each processor reads the r-register of every other processor and assembles a frequency table for the values it found. the frequency of that processor's own secret will be the only one with a frequency not divisible by (n-2)problem: size of r depends on n (linearly) If no messages are available at the socket, the receive calls wait for a message to arrive, unless the socket is nonblocking (see fcntl(2)), in which case the value -1 is returned and the external variable errno is set to EAGAIN or EWOULDBLOCK. The receive calls normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount requested. The socket is nonblocking and the connection cannot be completed immediately. (UNIX domain sockets failed with EAGAIN instead. It is possible to select(2) or poll(2) for completion by selecting the socket for writing. After select(2) indicates writability, use getsockopt(2) to rean the SO\_ERROR op‐tion at level SOL\_SOCKET to determine whether connect() completed successfully (SO\_ERROR is If no messages are available at the socket, the receive calls wait for a message to arrive, unless the socket is nonblocking (see fcntl(2)), in which case the value -1 is returned and the external variable errno is set to EAGAIN or EWOULDBLOCK. The receive calls normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount requested. Test it and see what happens to the client when the request is more than the buffer size of the client is it okay to do that

[nwanu@remote11 ~]$ ./server 5703

Attempting to bind to port 5703

Socket is bound to 0.0.0.0 5703

Connection from 129.16.29.51:55538 -> socket 4

suggestion each processor reads the secret of every other processor and puts it in its own r-register. after that, each processor reads the r-register of every other processor and assembles a frequency table for the values it found. the frequency of that