**TCP Client Server for Coding Challenge: Karen West**

**Github Link:** <https://github.com/KarenWest/tcpClientServerCodeChallenge/tree/main>

--The document describing the code challenge is in this repo: **Code Challenge (1).docx.**

--This document : TCPClientSErverCodeChallenge.docx describes how to run the code, how the code works, and what code was modified for the coding challenge requirement.

**Adapted for Coding Challenge from This Github Link**: <https://github.com/elhayra/tcp_server_client>

--I adapted this github repo to make it work for the coding challenge.

--I will highlight the code description that I either added or modified to make it meet the coding challenge requirements.

--I included descriptions of code not modified, and also included notes on systems calls and build tools at the end which can be scrolled past but are there for reference.

**Running the Code:**

To run the code to demonstrate the coding requirement: please see what I did in my Ubuntu VM running on Oracle Virtual Box in the following snips.

**In this coding challenge scenario:**

N = 3 sender/clients here (I only tested for N=3 client senders):

I ran on 1 Ubuntu Linux Virtual Machine (VM)  in Virtual Box (on a Windows 10 machine running Oracle's Virtual Box).

So all client senders have the same IP address: (localhost 127.0.0.1) for this one Linux VM.

In the snip below, they are distinguished by their FD (file descriptor) number 4,5,6.

**Server/Receiver Output: This is one terminal window of my Ubuntu Linux VM:**

**--You see the 3 client senders connect to the receiver/server at localhost IP 127.0.0.1 at FD's: 4,5,6.**

**--In the output,  you can see that Client 2 sent its message 2absd which was received by the server/receiver before Client 1 sends its message (in next snip). Then client 3 sends its message.**

**--Menu Item 4 in the server/receiver prints the received client messages in ID order (client 1’s message is printed first, then 2 and 3).**

A screenshot of a computer

Description automatically generated

**Client/Sender 2 Output in another terminal window in my Ubuntu Linux VM: Sends msg 2absd to server/receiver first**

A computer screen shot of a program

Description automatically generated

**Client/Sender 1 Output in another terminal window in my Ubuntu Linux VM : sends 1tuyuw6 to server/receiver next**

A computer screen shot of a program

Description automatically generated

**Client / Sender 3 Output: sends 3qwet to server/receiver last**

A screenshot of a computer program

Description automatically generated

**--After the 3 clients sent their messages (Client 2 then Client 1 then Client 3):**

--You can scroll back above to see when I selected the server/receiver menu option to print in ascending ID order the messages it received from the clients.

**Check on the Message Size Requirement of <= 10 characters:**

--Here client 2 sends a message > 10 characters and is asked to re-enter it:

A screenshot of a computer program

Description automatically generated

**Server/Receiver then receives the correct length message from Client 2:**

A screenshot of a computer

Description automatically generated

**Code Description:**

**examples/server\_example.cpp: main()**

* A server is started on port 65123.
* An observer (server\_observer\_t) is initialized with the fields:
  + incomingPacketHandler: onIncomingMsg
  + disconnectionHandler: onClientDisconnected
  + wantedIP: localhost: 127.0.0.1
* The server then subscribes to this observer handler.
* There are 3 client senders for this server example. There is a loop to accept these 3 clients calling the function: acceptClient().
* A Boolean called: shouldTerminate is initialized to FALSE, and the while loop looks at that and loops this server\_example with the following until this value becomes TRUE.
  + printMenu() is called – there are 5 options.
  + getMenuSelection() is called to determine what to do next from the menu selection.
  + shouldTerminate is the return from: handleMenuSelection().

**examples/client\_example.cpp: main()**

* signal() is called with parameters: SIGINT Linux signal, and sig\_exit as the handler of that. If the user does CTRL-C, this closes the client.
* A client observer (client\_observer\_t) is initialized with the following fields:
  + wantedIP: localhost: 127.0.0.1
  + incomingPacketHandler: onIncomingMsg
  + disconnectionHandler: onDisconnection
* The client the subscribes to this observer.
* A Boolean called connected is initialized to FALSE and the while loop looks at that and loops this client\_example with the following until this value becomes TRUE:
  + The client calls connectTo with these parameters:
    - Localhost IP address: 127.0.0.1
    - Port number: 65123
  + The loops stops when the connection succeeds.
* A Boolean called shouldTerminate is initialized to FALSE, and the while loop looks at that and loops this client\_example with the following until this value becomes TRUE.
  + printMenu() – there are 2 options.
  + getMenuSelection() is called to determine what to do next from the menu selection.
  + shouldTerminate is the return from: handleMenuSelection().

**--The first line in examples/server\_example.cpp: main() calls the server (TcpServer class in src/tcp\_server.cpp) code’s start method** on port 65123 and allows the default maxNumberOfClients to be 5 and the removeDeadClientsAutomatically to be TRUE.

**--The next part of examples/server\_example.cpp: main() configures the observer:**

**include/server\_observer.h: struct server\_observer\_t defines these fields that are used in the observer:**

* **wantedIP** (std::string) initialized to the empty string
* **incomingPacketHander** (std::function<void (const std::string &clientIP, const char \* msg, size\_t size)>)
* **disconnectionHandler** (std::function<void (const std::string &ip, const std::string &msg)>

--**So the examples/server\_example.cpp:** main() configures the observer (server\_oibserver\_t) to be:

* observer.incomingPacketHandler = onIncomingMsg
* observer.disconnectionHandler = onClientDisconnected
* observer.wantedIP = “127.0.0.1” //localhost

--**The examples/server\_example.cpp: then subscribes to this observer:** server.subscribe(observer).

--**The examples/server\_example.cpp: then loops for N=3 times calling acceptClient().**

**examples/server\_example.cpp: acceptClient():**

* **Sets up a try-catch block**
  + **The try block:**
    - Prints “waiting for incoming client…”
    - Sets std::string clientIP = server.acceptClient(0)
    - It calls the server.printClients() to print all currently connected clients with their IP address and file escriptor number.
  + **The catch(const std::runtime\_error &error) block:**
    - This prints to std::cout that the accepting client failed with error.what().

--**Now the examples/server\_example.cpp: main() starts it’s shouldTerminate while loop, with should terminate Boolean initialized to FALSE. These are more details on that loop that I had mentioned above:**

* **examples/server\_example.cpp: main() while loop call to: printMenu(): prints the following menu selection to** std::cout, asking the user to “select one of the following options: “
  + "1. send all clients a message"
  + "2. print list of accepted clients"
  + "3. send message to a specific client"
  + "4. Print Client Messages In Priority Order by client sender ID"
  + "5. close server and exit"
* **examples/server\_example.cpp: main() while loop call: selection is set to the return from: getMenuSelection():**
  + getMenuSelection(): Selection is set to zero. It then reads the menu selection (1-5 choices) from std::cin
  + If nothing is read from std::cin = 0, an error is thrown: std::runtime\_error() with “invalid menu input, expected a number, but got something else”.
  + It then calls: std::cin.ignore (std::numeric\_limits<std::streamsize>::max(), “\n”) amd returns selection.
* **examples/server\_example.cpp: main() while loop call: shouldTerminate is set to the return from: handleMenuSelection(selection):**
  + Static int’s: minSelection = 1 and maxSelection = 5 (currently has 5 menu choices). If not in the correct range, an error is printed and FALSE returned to shouldTerminate in main().
  + The switch case statement handles the menu selections as follows:
    - Case 1: Sends all clients a message. Msg is std::string.
      * A call to getline(std::cin, msg)
      * Pipe\_ret\_t = sendingResult = server.sendToAllClients(msg.c\_str(), msg.size()).
      * A message is printed if the send to all clients succeeds or fails and then this case breaks.
    - Case 2: print a list of accepted clients.
      * A call to the server (TcpServer class in src/tcp\_server.cpp) method printClients() is called and then this case breaks.
    - Case 3: Sends a message to a specific client
      * Std::cin reads in the IP address from the user along with the message to send to this client.
      * Pipe\_ret\_t result = server.sendToClient( clientIP, message.c\_str(), message.size()) and prints whether it was a success or failure before breaking from this case.
    - Case 4: prints all client messages based on the priority of the client ID in ascending ID order:
      * It calls: server.printClientsMsgsByPriorotyOfId() before breaking from this case.
    - Case 5: closes the server
      * Pipe\_ret\_t sendingResult = server.close() is called, and prints whether this was a success or failure, and returns true to main()’s shouldTerminate variable, which ends the main() while loop, now that the server has been closed. terminateDeadClientsRemover() is called. The server and all clients are closed.
    - Default case: prints an error message that the menu selection was out of range.
    - FALSE is returned to main’s shouldTerminte variable at the end, and TRUE is only returned in case 5 when the server is closed.

**examples/server\_example.cpp: onIncomingMsg handler that is initialized to be the callback function for the observer.incomingPacketHandler (see structure definition in : include/server\_observer.h, also described above). This handler will be called for every new message received by clients with the requested IP address.**

* This handler is shared by all the clients in this example, since it was used in a single Ubuntu Linux VM running on Virtual box. So it’s IP is localhost 127.0.0.1. The client’s are distinguished by their file descriptor numbers, in the example I ran in the output above, as 4,5 and 6.
* To demonstrate the priority scheme requested by the coding challenge, this handler first looked at the message’s front character, which was required to be the client ID number 1,2, or 3. So in my example output provided above, you will see this handler first received client 2’s message, followed by client 1’s message, and then client 3’s message.
* However, the coding challenge required as well that we print out the messages sent by client’s in ascending order of the client ID, so that is done with server/receiver menu option 4 after all 3 clients have sent their messages. It is then that you see the coding challenge requirement to print the messages in order of ascending client ID order is met. In this handler, you observe that the messages were not received that way (client 2 sent first, then client 1, then client 3) in the output provided above.

**examples/server\_example.cpp: onClientDisconnected handler that is initialized to be the callback function for the observer.disconnectionHandler (see structure definition in : include/server\_observer.h, also described above). This handler will be called when a client disconnects.**

* This handler prints a message that the client with it’s IP address disconnected with a reason as well.

**--After the first line in examples/client\_example.cpp: main() that calls signal(SIGINT, sig\_exit) to set up the signal handler for SIGINT to be sig\_exit(), the client then sets up its observer.**

**include/client\_observer.h: struct client\_observer\_t defines these fields that are used in the observer:**

* **wantedIP** (std::string) initialized to the empty string
* **incomingPacketHander** = nullptr: (std::function<void (const char \* msg, size\_t size)>)
* **disconnectionHandler** = nullptr (std::function<void (const pipe\_ret\_t &ret)>)

--**So the examples/client\_example.cpp:** main() configures the client (client\_observer\_t) to be:

* observer.incomingPacketHandler = onIncomingMsg
* observer.disconnectionHandler = onDisconnection
* observer.wantedIP = “127.0.0.1” //localhost

--**The examples/client\_example.cpp: then subscribes to this observer:** client.subscribe(observer).

-- **The examples/client\_example.cpp: then connects the client to an open server: a while loop that loops on the Boolean connected is started and exits when the client is connected to the server:**

* pipe\_ret\_t connectedRet = client.connectTo(“127.0.0.1” //localhost, 65123 //port)
* A message is printed saying whether the connection was a success or failure, and if a failure, it sleeps(2), and then prints a message that it is trying to reconnect the client to the server. IF the connection was a success, the Boolean connected is TRUE and the while loop exits.

--**Now the examples/client\_example.cpp: main() starts it’s shouldTerminate while loop, with shouldTerminate Boolean initialized to FALSE. These are more details on that loop that I had mentioned above:**

* **examples/client\_example.cpp: main() while loop call to: printMenu(): prints the following menu selection to** std::cout, asking the user to “select one of the following options: “
  + "1. send message to server"
  + "2. close client and exit"
* **examples/client\_example.cpp: main() while loop call: selection is set to the return from: getMenuSelection():**
  + getMenuSelection(): Selection is set to zero. It then reads the menu selection (1-2 choices) from std::cin
  + If nothing is read from std::cin = 0, an error is thrown: std::runtime\_error() with “invalid menu input, expected a number, but got something else”.
  + It then calls: std::cin.ignore (std::numeric\_limits<std::streamsize>::max(), “\n”) and returns selection.
* **examples/client\_example.cpp: main() while loop call: shouldTerminate is set to the return from: handleMenuSelection(selection):**
  + Static int’s: minSelection = 1 and maxSelection = 2 (currently has 2 menu choices). If not in the correct range, an error is printed and FALSE returned to shouldTerminate in main().
  + The switch case statement handles the menu selections as follows:
    - Case 1: Sends message to the server.
      * A call to std::cin is made to get the std::string message. A check is made to ensure the message length is <= 10 characters or otherwise it asks the user to reenter a message size in this range.
      * Pipe\_ret\_t = sendRet = client.sendMSg(message.c\_str(), message.size()).
      * A message is printed if the send to the server succeeds or fails and then this case breaks.
    - Case 2: closes the client
      * Pipe\_ret\_t closeResult = client.close() is called, and prints whether this was a success or failure, and returns true to main()’s shouldTerminate variable, which ends the main() while loop, now that the client has been closed. The client is closed if it’s not been closed already. **src/tcp\_server.cpp:TcpServer::terminateReceiveThread**() is called to **delete the \_receiveTask**. The socket is closed and the \_isClosed member variable is set to TRUE.
    - Default case: prints an error message that the menu selection was out of range.
    - FALSE is returned to main’s shouldTerminate variable at the end, and TRUE is only returned in case 2 when the client is closed.

**examples/client\_example.cpp: onIncomingMsg handler that is initialized to be the callback function for the observer.incomingPacketHandler (see structure definition in : include/client\_observer.h, also described above). This handler will be called for every new message received by the server.**

* A message is printed that says “Got msg from server:“ , along with the msg received from the server.

**examples/client\_example.cpp: onDisconnection handler that is initialized to be the callback function for the observer.disconnectionHandler (see structure definition in : include/client\_observer.h, also described above). This handler will be called when a client disconnects.**

* This handler prints a message that the server disconnected with a message sent as a parameter as well.

**examples/client\_example.cpp: sig\_exit(int s):** when the SIGINT signal is sent while running the client, it means the user did a CTRL-C to stop the client from running, and sig\_exit() handles that:

* A message is printed that the client is closing.
* Pipe\_ret\_t finishRet = client.close() is called, and a message is printed to std::cout that the client is closed if successful, or that it failed to close the client if the client.close() fails. **Src/client.cpp:Client::terminateReceiveThread()** is called to **delete the \_receiveThread**. If closing the socket fails, a run time error is thrown.
* Exit(0) is called to end the client process from running on Linux.

**--The src directory object files are made into a *static shared library* (libtcp\_client\_server.a) for the client and server example code to use. It is a compiled archive (.a) but not linked with the other code until the program build’s final step. It is *not* a dynamically linked library that is linked only at run time of the code.**

* src/tcp\_server.cpp
* src/tcp\_client.cpp:
* src/common.cpp
* src/client.cpp:
* src/pipe\_ret\_t.cpp

**src/Tcp\_server.cpp: Constructor : TcpServer::TcpServer():**

* Initializes TcpServer class member variable **\_subscribers.reserve(10),** so at least 10 server\_observer\_t’s are allocated for the \_subscribers vector, and the vector space is automatically expanded as needed (similarly for the \_clients vector space below).
* Initializes TcpServer class member variable **\_clients.reserve(10)**
* Initializes TcpServer class member variable \_**stopRemoveClientsTask = FALSE**

**src/Tcp\_server.cpp: Destructor : TcpServer::~TcpServer():**

* Calls close() on TcpServer(), resources released (see TcpServer::close()).

**src/Tcp\_server.cpp: TcpServer::subscribe(const server\_observer\_t &observer):**

* Locks the server member variable: \_subscribersMtx with:
  + std::lock\_guard<std::mutex> lock(\_subscriberMtx)
* The server adds this subscriber observer to its list:
  + \_subscribers.push\_back(observer)

**src/Tcp\_server.cpp : TcpServer::printClients():**

* Locks the server member variable: \_clientsMtx with:
  + std::lock\_guard<std::mutex> lock(\_clientsMtx)
* If the member variable vector \_clients is empty, the “no connected clients” message is printed.
* A for loop is started on the client vector, and the src/clients.cpp class print() is called on each Client in the server’s member variable vector \_clients.

**src/Tcp\_server.cpp: TcpServer::removeDeadClients(): (dead client = disconnected client):**

* An iterator is created for the client vector list:
  + Std::vector<Client\*>::const\_iterator clientToRemove;
* A while loop is started looping on the member Boolean variable \_stopRemoveDeadClientsTask
  + Locks the server member variable: \_clientsMtx with:
    - std::lock\_guard<std::mutex> lock(\_clientsMtx)
  + A do-while loop is started:
    - clientToRemove = std::find\_if(\_clients.begin(), \_clients.end(), [] (Client \*client) { return !client->isConnected(); } )
    - If the clientToRemove is not at the end of the vector list:
      * (\*clientToRemove)->close() is called to release that client’s resources and then the client is deleted.
    - The do-while loop ends if: clientToRemove is the \_clients.end(), the last on the clients vector list.
  + There is a sleep(2) and then back to the top of the while loop to evaluate the member Boolean again: \_stopRemoveClientsTask. When this Boolean is true, TcpServer::removeDeadClients is done (which is set to true in: TcpServer::terminateDeadClientsRemover()).

**src/Tcp\_server.cpp: TcpServer::terminateDeadClientsRemover():**

* If the TcpServer member variable: \_clientsRemoverThread is not zero:
  + the TcpServer member variable: \_stopRemoveClientsTask is set to TRUE.
  + The TcpServer member variable \*: **\_clientsRemoverThread->join()** is called. The TcpServer thread \*: **\_clientsRemoverThread blocks until it finishes executing**.
  + TcpServer \* \_clientsRemoverThread is deleted and then set to nullptr.

**src/Tcp\_server.cpp: TcpServer::clientEventHandler(const Client &client, ClientEvent event, const std::string &msg)): handle different client events. Subscriber callbacks should be short and fast and not call other server functions to avoid deadlock.**

* A switch case statement on the parameter: event:
  + Case ClientEvent::DISCONNECTED:
    - A call is made to TcpServer: publishClientDisconnected(client.getIp(), msg) before breaking from this case.
  + Case ClientEvent::INCOMING\_MSG:
    - A call is made to TcpServer: publishClientMsg(client, msg.c\_str(), msg.size()) before breaking from this case.

**src/Tcp\_server.cpp: TcpServer::publishClientMsg(const Client & client, const char \* msg, size\_t msgSize):**

* Locks the server member variable: \_subscribersMtx with:
  + std::lock\_guard<std::mutex> lock(\_subscriberMtx)
* A std::string msgStr is made from the parameter const char \* msg
* A for loop is started to loop on the server observer subscriber vector list member variable: \_subscribers:
  + If the subscriber.wantedIP == client.getIp() (observer gets notification about client from specific client IP here) OR subscriber.wantedIp is empty:
    - If the subscriber.incomingPacketHandler is defined:
      * The first letter in the message string parameter is checked to find out which client sent this message, since the first character is always the ID of the client (1,2,3).
      * The member variable \_observerMsgs[] array of std::string’s is indexed by the client ID and the message placed here so that later it can be printed in the ascending priority order as specified in the coding challenge.
      * Subscriber.incomingPacketHandler(client.getIp(), msg, msgSize) is called.

**src/Tcp\_server.cpp: TcpServer::printClientMsgsByPriorityOfId():** assumes for the coding challenge example that there are N=3 clients: This is the server’s menu option to print clients by client ID in ascending order, and not necessarily in the order that the messages were received by the server from the clients. These are in the TcpServer’s \_observerMsgs array, which has the messages it receives there from the clients in the TcpServer publishClientMsg().

**src/Tcp\_server.cpp: TcpServer::publishClientDisconnected(const std::string &clientIP, const std::string &clientMsg):**

* Locks the server member variable: \_subscribersMtx with:
  + std::lock\_guard<std::mutex> lock(\_subscriberMtx)
* A for loop is started to loop on the server observer subscriber vector list member variable: \_subscribers:
* If the subscriber.wantedIP == clientIP (observer gets notification about client from specific client IP here)
  + If the subscriber.disconnectionHandler is defined:
    - A call is made to: subscriber.disconnectionHandler(clientIP, clientMsg)

**src/Tcp\_server.cpp: TcpServer::start(int port, int maxNumOfClients, bool removeDeadClientsAutomatically): binds the port and and starts listening**

* If the removeDeadClientsAutomatically Boolean parameter to start() is TRUE, the **server’s member variable \_clientsRemoverThread is instantiated with a call to : new std::Thread ()** passing it the **parameter &TcpServer::removeDeadClients** and it’s TcpServer class object (this) as well.
* **It then sets up a try-catch block:**
  + **The try block:**
    - Calls initializeSocket()
    - bindAddress()
    - listenToClients(maxNumberOfClients)
  + **The catch ( const std::runtime\_error &error) block:**
    - Returns to the caller: pipe\_ret\_t::failure(error.what())
  + **Otherwise this is returned:** pipe\_ret\_t::success()

**src/Tcp\_server.cpp: TcpServer::initializeSocket():**

* The member variable \_sockfd is initialized using **socket() with protocol family AF\_INET, socket type SOCK\_STREAM, and protocol zero.** This socket is the uniform interface between the user process and the network protocol stacks in the kernel. The socket layer functions are used by the user process (tcp\_server here) to send or receive packets and to do other socket operations. If this socket creation fails a runtime error is thrown.
* **Setsockopt()** is used **to set the socket for reuse**. This is done because if the socket is closed, you may have to wait 4 minutes. It is given level **SOL\_SOCKET** so that the socket can be manipulated at the API level. Optname **SO\_REUSEADDR** is passed uninterpreted to the appropriate protocol module for interpretation (socket reuse). Most **socket-level options** utilize an int argument for optval, and it should be non-zero (1) to enable a boolean option, or zero if the option is to be disabled.

**src/Tcp\_server.cpp: TcpServer::bindAddress(int port):**

* The tcp\_server member variable \_serverAddress (type struct sockaddr\_in) is first zeroed and then initialized with these fields:
  + \_serverAddress.sin\_family = AF\_INET
  + \_serverAddress.sin\_addr.s\_addr = htonl(INADDR\_ANY)
  + \_serverAddress.sin\_port = htons(port)
  + Note: uint32\_t htonl(uint32\_t hostlong) : The **htonl**() function converts the unsigned integer *hostlong* from host byte order to network byte order.
  + Note: uint16\_t htons(uint16\_t hostshort): The **htons**() function converts the unsigned short integer *hostshort* from host byte order to network byte order.
  + Note: On the i386 the host byte order is Least Significant Byte first, whereas the network byte order, as used on the Internet, is Most Significant Byte first.
  + Note: Some systems require the inclusion of *<[netinet/in.h](https://linux.die.net/include/netinet/in.h)>* instead of *<[arpa/inet.h](https://linux.die.net/include/arpa/inet.h)>*.
* **Bind**() is done next: When a socket is created with socket(), it exists in a name space (address family) but has no address assigned to it. **bind**() assigns the address specified by *\_serverAddress* to the socket referred to by the file descriptor \_*sockfd*. *addrlen* specifies the size, in bytes, of the address structure pointed to by *\_serverAddress.* Traditionally, this operation is called “assigning a name to a socket”. If bind fails, a run time error is thrown.

**src/Tcp\_server.cpp: TcpServer::** **listenToClients(int maxNumOfClients):**

* **Listen(int sockfd, int backlog) is called** passing in the \_sockfd and the maxNumOfClients.
* Listen() marks the socket referred to by *sockfd* as a passive socket, that is, as a socket that will be used to accept incoming connection requests using accept(). The *sockfd* argument is a file descriptor that refers to a socket of type SOCK\_STREAM in this code, but could also be SOCK\_SEQPACKET. The *backlog* argument defines the maximum length to which the queue of pending connections for *sockfd* may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of **ECONNREFUSED** or, if the underlying protocol supports retransmission, the request may be ignored so that a later reattempt at connection succeeds. If listening to the maxNumOfClients fails, a run time error is thrown.

--still working on the document—the following are not completed yet:

**src/tcp\_server.c** – some of this is above, but any code that I added or changed is already described and highlighted above.

**src/tcp\_server.cpp: TcpServer::acceptClient(uint timeout):**

* **Accept and handle new client socket.**
* To handle multiple clients, client\_example.cpp : main() calls this 3 times to enable the acceptance of more than one client.
* If the timeout argument is zero, this function is executed in blocking mode.
* If the timeout argument is greater than zero, this function is executed in non-blocking mode (asynchronously) and will quit after timeout seconds if no client tried to connect.
* It returns the accept client IP (std::string).
* src/tcp\_server.cpp: **TcpServer:: waitForClient(timeout) is called**, blocking if timeout=0, and within a timeout if timeout > 0. If not successful in connecting to a client, a run time error is thrown.
* **fileDescriptor = accept(\_sockfd.get(), (struct sockaddr\*)&\_clientAddress, &socketSize):** accept() accepts a connection on socket. This system call is used with connection-based socket types (SOCK\_STREAM, SOCK\_SEQPACKET—SOCKSTREAM in this project example). It extracts the first connection request on the queue of pending connections for the listening socket, \_sockfd.get(), creates a new connected socket, and returns a fileDescriptor referring to that socket. The newly created socket is not in the listening state. The original socket in \_sockfd is unaffected by this call. The argument \_sockfd is a socket that has been created with socket(), bound to a local IP address with bind(), and is listening for connections with listen(). The argument \_clientAddress is a pointer to a sockaddr structure. This structure is filled in with the address of the peer socket, as known to the communications layer. The exact format of the address returned to \_clientAddress is determined by the socket’s address family (referenced in socket() and respective protocol man pages). When \_clientAddress is NULL, nothing is filled in, and in this case, &socketSize is NULL since its not used. The socketSize argument is a value-result argument. The caller must initialize it to contain the size in bytes of the structure pointed to by \_clientAddress, and on return, it will contain the actual size of the peer address. The returned address is truncated if the buffer provided is too small. In this case, socketSize will return a value greater than was supplied to the call. If no pending connections are waiting on the queue, and the socket is NOT marked as non-blocking, accept() blocks the caller until a connection is present. If the socket is marked as non-blocking, and no pending connections are present in the queue, accept() fails with the error EAGAIN or EWOULDBLOCK.
* **In order to be notified of incoming connections on a socket, you can use select(), poll() or epoll(). A readable event will be delivered when a new connection is attempted and you may then call accept() to get a socket for that connection.** Alternatively, you can set the socket to deliver SIGIO when activity occurs on the socket.
* **There is another version of accept() that has a flags parameter.** When this is zero, it’s the same as the other accept() system call, and when non-zero, the following values can be bitwise Ored in flags to obtain different behavior:
  + SOCK\_NONBLOCK: Set the O\_NONBLOCK file status flag on the open file description (see open()) referred to by the new file descriptor. Using this flag saves extra calls to fcntl() to achieve the same result.
  + SOCK\_CLOEXEC: Set the close-on-exec (FD\_CLOEXEC) flag on the new file descriptor. See the description of the O\_CLOEXEC flag in open() for reasons why this may be useful.
* **On success, these accept() system calls return a file descriptor for the accepted socket** ( a non-negative integer). On error, -1 is returned, errno is set to indicate the error and socketSize is left unchanged. In the error case, a run time error event is thrown with the errno.
* **auto newClient = new Client(fileDescriptor);**
* The **auto** keyword in C++ **automatically detects and assigns a data type to the variable with which it is used**. The compiler analyses the variable's data type by looking at its initialization. It is necessary to initialize the variable when declaring it using the auto keyword.
* The IP address is assigned: **newClient->setIp(inet\_ntoa(\_clientAddress.sin\_addr));**
* NOTE: The **inet\_ntoa**() function converts the Internet host address *in*, given in network byte order, to a string in IPv4 dotted-decimal notation. The string is returned in a statically allocated buffer, which subsequent calls will overwrite.
* **using namespace std::placeholders**
* **newClient->setEventsHandler(std::bind(&TcpServer::clientEventHandler, this, \_1, \_2, \_3))**
* Note: **std::bind**: Returns a function object based on fn, but with its arguments bound to args.  
  Each argument may either be bound to a *value* or be a [*placeholder*](https://cplusplus.com/placeholders):  
  - If bound to a *value*, calling the returned function object will always use that value as argument.  
  - If a [*placeholder*](https://cplusplus.com/placeholders), calling the returned function object forwards an argument passed to the call (the one whose order number is specified by the placeholder).
* **std::placeholders: Note:**
  + **namespace placeholders** { extern /\* unspecified \*/ \_1; extern /\* unspecified \*/ \_2; extern /\* unspecified \*/ \_3; // ...}
  + **Bind argument placeholders**
  + This **namespace declares an unspecified number of objects: \_1, \_2, \_3,...,** which are used **to specify *placeholders* in calls to function** [**bind**](https://cplusplus.com/bind).
  + When the function object returned by [bind](https://cplusplus.com/bind) is called, an argument with *placeholder* \_1 is replaced by the first argument in the call, \_2 is replaced by the second argument in the call, and so on... For example:

|  |
| --- |
| * + using namespace std::placeholders;   + auto bound\_fn = std::bind (fn,100,\_1);   + bound\_fn(5); // calls fn(100,5), i.e.: replacing \_1 by the first argument: 5 |

* **newClient->startListen()**
* **std::lock\_guard<std::mutex> lock(\_clientsMtx);**
* **\_clients.push\_back(newClient); // new client is added to the servers clients vector**
* **return newClient->getIp();**

**src/tcp\_server.cpp: TcpServer::** **waitForClient(uint32\_t timeout):**

* When timeout > 0: The server waits for the client with the timeout value by calling :
  + src/common.cpp: in **namespace fd\_wait:** 
    - **const fd\_wait::Result waitResult = fd\_wait::waitFor(\_sockfd, timeout)**
  + A Boolean is initialized to**: noIncomingClient =** (!**FD\_ISSET(\_sockfd.get(), &\_fds))**
  + **FD\_ISSET(fd, &fdset):** Returns a non-zero value if the bit for the file descriptor fd is set in the file descriptor set pointed to by fdset, and 0 otherwise.
  + These are the possible error cases with waitResult: FAILURE, TIMEOUT or FILE DESCRIPTOR IS NOT SET. Otherwise, SUCCESS is returned.

**src/Tcp\_server.cpp:: TcpServer::** **sendToAllClients(const char \* msg, size\_t size):**

* **The server’s \_clientsMtx is locked for its \_clients vector**: std::lock\_guard<std::mutex> lock(\_clientsMtx);
* For each client in the server’s vector of \_clients: sendToClient() is called, passing a pointer to the client, the msg and the size, and the send result returned to the caller immediately if one client send is not successful, but if all client sends are successful, the send result of success is returned for all clients to the caller.

**src/tcp\_server.cpp::** **TcpServer::sendToClient(const Client & client, const char \* msg, size\_t size):**

* Sends a message to a specific client determined by the IP address, and returns true if the message was sent successfully.
* This is done by using a try-catch block:
  + Try: client.send(msg, size)
  + Catch(): returns the run time error if the client.send() fails.

**src/tcp\_server.cpp::** **TcpServer::sendToClient(const std::string & clientIP, const char \* msg, size\_t size):**

* Because **this version of TcpServer::sendToClient()** uses the \_clients member variable vector of clients, the mutex for it has to be locked: std::lock\_guard<std::mutex> lock(\_clientsMtx)
* An iterator is created to iterate through the \_clients vector:
* **Note**: InputIterator find\_if (InputIterator first, InputIterator last, UnaryPredicate pred);
* First iterator below is: clients.begin()
* Last iterator below is: clients.end()
* Pred below is: [&clientIP](Client \*client) { return client->getIp() == clientIP; }
* **Std::find\_if:** Returns an iterator to the first element in the range [first,last) for which pred returns true. If no such element is found, the function returns last.
  + const auto clientIter = std::find\_if(\_clients.begin(), \_clients.end(),

[&clientIP](Client \*client) { return client->getIp() == clientIP; });

* If a **client is found in the server’s \_clients vector with a matching IP address, the other TcpServer::sendToClient() described above here is called, and the return value returned to the caller:**
  + **return sendToClient(client, msg, size)**

**src/tcp\_server.cpp::** **TcpServer::close():**

* **terminateDeadClientsRemover**() is called
* In the first block: **The server’s \_clientsMtx is locked for its \_clients vector**: std::lock\_guard<std::mutex> lock(\_clientsMtx);
* A for loop on the TcpServer’s \_clients vector is started with a try-catch block:
  + **Try block: calls: client->close() //src/clients.cpp:Client::close()**
  + **Catch block: returns the run time error if the client->close() fails.**
  + The TcpServer’s \_clients vector is then cleared: **\_clients.clear()**: destroys the vector by removing all elements from the vector and sets size of vector to zero.
* **In the second block: the server is closed.** 
  + **::close( \_sockfd.get()) is called. This description goes with unistd.h version.** The *close*() function shall deallocate the file descriptor indicated by *fildes*. To deallocate means to make the file descriptor available for return by subsequent calls to *open*() or other functions that allocate file descriptors. All outstanding record locks owned by the process on the file associated with the file descriptor shall be removed (that is, unlocked).
  + If *close*() is interrupted by a signal that is to be caught, it shall return -1 with *errno* set to [EINTR] and the state of *fildes* is unspecified. If an I/O error occurred while reading from or writing to the file system during *close*(), it may return -1 with *errno* set to [EIO]; if this error is returned, the state of *fildes* is unspecified.
  + When all file descriptors associated with a pipe or FIFO special file are closed, any data remaining in the pipe or FIFO shall be discarded.
  + When all file descriptors associated with an open file description have been closed, the open file description shall be freed.
  + If the link count of the file is 0, when all file descriptors associated with the file are closed, the space occupied by the file shall be freed and the file shall no longer be accessible.
  + If a STREAMS-based *fildes* is closed and the calling process was previously registered to receive a SIGPOLL signal for events associated with that STREAM, the calling process shall be unregistered for events associated with the STREAM. The last *close*() for a STREAM shall cause the STREAM associated with *fildes* to be dismantled. If O\_NONBLOCK is not set and there have been no signals posted for the STREAM, and if there is data on the module's write queue, *close*() shall wait for an unspecified time (for each module and driver) for any output to drain before dismantling the STREAM. The time delay can be changed via an I\_SETCLTIME *ioctl*() request. If the O\_NONBLOCK flag is set, or if there are any pending signals, *close*() shall not wait for output to drain, and shall dismantle the STREAM immediately.
  + If the implementation supports STREAMS-based pipes, and *fildes* is associated with one end of a pipe, the last *close*() shall cause a hangup to occur on the other end of the pipe. In addition, if the other end of the pipe has been named by *fattach*(), then the last *close*() shall force the named end to be detached by *fdetach*(). If the named end has no open file descriptors associated with it and gets detached, the STREAM associated with that end shall also be dismantled.
  + If *fildes* refers to the master side of a pseudo-terminal, and this is the last close, a SIGHUP signal shall be sent to the controlling process, if any, for which the slave side of the pseudo-terminal is the controlling terminal. It is unspecified whether closing the master side of the pseudo-terminal flushes all queued input and output.
  + If *fildes* refers to the slave side of a STREAMS-based pseudo-terminal, a zero-length message may be sent to the master.
  + When there is an outstanding cancelable asynchronous I/O operation against *fildes* when *close*() is called, that I/O operation may be canceled. An I/O operation that is not canceled completes as if the *close*() operation had not yet occurred. All operations that are not canceled shall complete as if the *close*() blocked until the operations completed. The *close*() operation itself need not block awaiting such I/O completion. Whether any I/O operation is canceled, and which I/O operation may be canceled upon *close*(), is implementation-defined.
  + If a shared memory object or a memory mapped file remains referenced at the last close (that is, a process has it mapped), then the entire contents of the memory object shall persist until the memory object becomes unreferenced. If this is the last close of a shared memory object or a memory mapped file and the close results in the memory object becoming unreferenced, and the memory object has been unlinked, then the memory object shall be removed.
  + If *fildes* refers to a socket, *close*() shall cause the socket to be destroyed. If the socket is in connection-mode, and the SO\_LINGER option is set for the socket with non-zero linger time, and the socket has untransmitted data, then *close*() shall block for up to the current linger interval until all data is transmitted.
  + **If closing the server fails, a run time error is returned, otherwise, success is returned.**

**src/tcp\_client.cpp:**

**src/tcp\_client.cpp:TcpClient::TcpClient() constructor:**

* The TcpClient constructor initializes \_isConnected to FALSE and \_isClosed to TRUE.

**src/tcp\_client.cpp:TcpClient::~TcpClient() destructor:**

* TcpClient::close() is called, terminating the receive thread, calling the unistd.h ::close() on the socket file descriptor and returns fail if this fails, sets \_isClosed to TRUE, and returns success.

**src/tcp\_client.cpp:TcpClient::connectTo(const std::string & address, int port):**

* **A try-catch block starts the connectTo() function.**
  + **Try block:** 
    - * **Calls initializeSocket()**
      * **Calls setAddress(address, port)**
  + **Catch block: returns a run time error if the try block fails.**
* **connectResult = connect(\_sockfd.get() , (struct sockaddr \*)&\_server , sizeof(\_server))**
  + The **connect**() system call connects the socket referred to by the file descriptor *sockfd* to the address specified by *addr*. The *addrlen* argument specifies the size of *addr*. The format of the address in *addr* is determined by the address space of the socket *sockfd*; see [socket(2)](https://man7.org/linux/man-pages/man2/socket.2.html) for further details.
  + If the socket *sockfd* is of type **SOCK\_DGRAM**, then *addr* is the address to which datagrams are sent by default, and the only address from which datagrams are received. If the socket is of type **SOCK\_STREAM** or **SOCK\_SEQPACKET**, this call attempts to make a connection to the socket that is bound to the address specified by *addr*.
  + Some protocol sockets (e.g., UNIX domain stream sockets) may successfully **connect**() only once.
  + Some protocol sockets (e.g., datagram sockets in the UNIX and Internet domains) may use **connect**() multiple times to change their association.
  + Some protocol sockets (e.g., TCP sockets as well as datagram sockets in the UNIX and Internet domains) may dissolve the association by connecting to an address with the *sa\_family* member of *sockaddr* set to **AF\_UNSPEC**; thereafter, the socket can be connected to another address. (**AF\_UNSPEC** is supported since Linux 2.2.)
* If the connect() fails, failure is returned.
* TcpClient::startReceivingMessages() is called.
* The TcpClient member variable \_isConnected is set to TRUE and \_isClosed is set to FALSE, and success returned.

**src/tcp\_client.cpp:TcpClient::** **initializeSocket():**

* The member variable \_sockfd is initialized using **socket() with protocol family AF\_INET, socket type SOCK\_STREAM, and protocol zero.** This socket is the uniform interface between the user process and the network protocol stacks in the kernel. The socket layer functions are used by the user process (tcp\_client here) to send or receive packets and to do other socket operations. If this socket creation fails , a run time error is thrown.

**src/tcp\_client.cpp:TcpClient::** **setAddress(const std::string& address, int port):**

* const int inetSuccess = **inet\_aton(address.c\_str(), &\_server.sin\_addr)**
  + **Note: int inet\_aton(const char \****cp***, struct in\_addr \****inp***);**
  + **inet\_aton**() converts the Internet host address *cp* from the IPv4 numbers-and-dots notation into binary form (in network byte order) and stores it in the structure that *inp* points to. **inet\_aton**() returns nonzero if the address is valid, zero if not. The address supplied in *cp* can have one of the following forms:
  + *a.b.c.d*
  + Each of the four numeric parts specifies a byte of the address; the bytes are assigned in left-to-right order to produce the binary address.
  + *a.b.c*
  + Parts *a* and *b* specify the first two bytes of the binary address. Part *c* is interpreted as a 16-bit value that defines the rightmost two bytes of the binary address. This notation is suitable for specifying (outmoded) Class B network addresses.
  + *a.b*
  + Part *a* specifies the first byte of the binary address. Part *b* is interpreted as a 24-bit value that defines the rightmost three bytes of the binary address. This notation is suitable for specifying (outmoded) Class C network addresses.
  + *a*
  + The value *a* is interpreted as a 32-bit value that is stored directly into the binary address without any byte rearrangement.
  + In all of the above forms, components of the dotted address can be specified in decimal, octal (with a leading *0*), or hexadecimal, with a leading *0X*). Addresses in any of these forms are collectively termed *IPV4 numbers-and-dots notation*. The form that uses exactly four decimal numbers is referred to as *IPv4 dotted-decimal notation* (or sometimes: *IPv4 dotted-quad notation*
* **If inet\_aton() does not put the IP address into IP strings and dots formet, the next step is to try to resolve it:** 
  + - struct hostent \*host
    - struct in\_addr \*\*addrList
    - if ( (host = **gethostbyname( address.c\_str() )** ) == nullptr){
    - throw std::runtime\_error("Failed to resolve hostname")
    - }
    - addrList = (struct in\_addr \*\*) host->h\_addr\_list
    - \_server.sin\_addr = \*addrList[0]
* If inet\_aton() succeeded in putting the IP address into IP strings and dots format:
  + \_server.sin\_family = AF\_INET //**protocol family**
  + \_server.sin\_port = **htons**(port)
  + Note: uint16\_t htons(uint16\_t hostshort): The **htons**() function converts the unsigned short integer *hostshort* from host byte order to network byte order.
  + Note: On the i386 the host byte order is Least Significant Byte first, whereas the network byte order, as used on the Internet, is Most Significant Byte first.
  + Note: Some systems require the inclusion of *<[netinet/in.h](https://linux.die.net/include/netinet/in.h)>* instead of *<[arpa/inet.h](https://linux.die.net/include/arpa/inet.h)>*.

**src/tcp\_client.cpp:TcpClient::** **startReceivingMessages():**

* The client member variable **\_receiveTask** is initialized with:
  + - **new std::thread(&TcpClient::receiveTask, this);**
    - **Note: std::thread:**
    - class thread;
    - Thread
    - Class to represent individual *threads of execution*.
    - A thread of execution is a sequence of instructions that can be executed concurrently with other such sequences in *multithreading* environments, while sharing a same address space.
    - An initialized thread object represents an active thread of execution; Such a thread object is *joinable*, and has a unique *thread id*.
    - A default-constructed (non-initialized) thread object is *not joinable*, and its *thread id* is common for all *non-joinable* threads.
    - A *joinable* thread becomes *not joinable* if *moved from*, or if either join or detach are called on them

**src/tcp\_client.cpp:TcpClient::** **sendMsg(const char \* msg, size\_t size):**

* **const size\_t numBytesSent = send(\_sockfd.get(), msg, size, 0)**
* **If the number of bytes sent is less than zero or if less than the number of bytes requested in the size parameter, an error message is printed, and failure returned. Otherwise, success is returned if the send() succeeds.**
  + **Note:** ssize\_t send(int *sockfd*, const void *buf*[.*len*], size\_t *len*, int *flags*)
  + The system calls **send**(), **sendto**(), and **sendmsg**() are used to transmit a message to another socket.
  + The **send**() call may be used only when the socket is in a *connected* state (so that the intended recipient is known). The only difference between **send**() and [write(2)](https://man7.org/linux/man-pages/man2/write.2.html) is the presence of *flags*. With a zero *flags* argument, **send**() is equivalent to [write(2)](https://man7.org/linux/man-pages/man2/write.2.html). Also, the following call:
    - send(sockfd, buf, len, flags)
    - is equivalent to
    - sendto(sockfd, buf, len, flags, NULL, 0)
    - The argument *sockfd* is the file descriptor of the sending socket.
    - If **sendto**() is used on a connection-mode (**SOCK\_STREAM**, **SOCK\_SEQPACKET**) socket, the arguments *dest\_addr* and *addrlen* are ignored (and the error **EISCONN** may be returned when they are not NULL and 0), and the error **ENOTCONN** is returned when the socket was not actually connected. Otherwise, the address of the target is given by *dest\_addr* with *addrlen* specifying its size.
    - For **sendmsg**(), the address of the target is given by *msg.msg\_name*, with *msg.msg\_namelen* specifying its size. For **send**() and **sendto**(), the message is found in *buf* and has length *len*. For **sendmsg**(), the message is pointed to by the elements of the array *msg.msg\_iov*. The **sendmsg**() call also allows sending ancillary data (also known as control information). If the message is too long to pass atomically through the underlying protocol, the error **EMSGSIZE** is returned, and the message is not transmitted.
    - No indication of failure to deliver is implicit in a **send**().
    - Locally detected errors are indicated by a return value of -1.
    - When the message does not fit into the send buffer of the socket, **send**() normally blocks, unless the socket has been placed in nonblocking I/O mode. In nonblocking mode it would fail with the error **EAGAIN** or **EWOULDBLOCK** in this case.
    - The select() call may be used to determine when it is possible to send more data.
    - **The flags argument:**
    - The *flags* argument is the bitwise OR of zero or more of the following flags.
    - **MSG\_CONFIRM:** Tell the link layer that forward progress happened: you got a successful reply from the other side. If the link layer doesn't get this it will regularly reprobe the neighbor (e.g., via a unicast ARP). Valid only on **SOCK\_DGRAM** and **SOCK\_RAW** sockets and currently implemented only for IPv4 and IPv6. See [arp(7)](https://man7.org/linux/man-pages/man7/arp.7.html) for details.
    - **MSG\_DONTROUTE:** Don't use a gateway to send out the packet, send to hosts only on directly connected networks. This is usually use only by diagnostic or routing programs. This is defined only for protocol families that route; packet sockets don't.
    - **MSG\_DONTWAIT** : Enables nonblocking operation; if the operation would block, **EAGAIN** or **EWOULDBLOCK** is returned. This provides similar behavior to setting the **O\_NONBLOCK** flag (via the [fcntl(2)](https://man7.org/linux/man-pages/man2/fcntl.2.html) **F\_SETFL** operation), but differs in that **MSG\_DONTWAIT** is a per-call option, whereas **O\_NONBLOCK** is a setting on the open file description (see [open(2)](https://man7.org/linux/man-pages/man2/open.2.html)), which will affect all threads in the calling process and as well as other processes that hold file descriptors referring to the same open file description.
    - **MSG\_EOR** : Terminates a record (when this notion is supported, as for sockets of type **SOCK\_SEQPACKET**).
    - **MSG\_MORE** : The caller has more data to send. This flag is used with TCP sockets to obtain the same effect as the **TCP\_CORK** socket option (see [tcp(7)](https://man7.org/linux/man-pages/man7/tcp.7.html)), with the difference that this flag can be set on a per-call basis Since Linux 2.6, this flag is also supported for UDP sockets, and informs the kernel to package all of the data sent in calls with this flag set into a single datagram which is transmitted only when a call is performed that does not specify this flag. (See also the **UDP\_CORK** socket option described in [udp(7)](https://man7.org/linux/man-pages/man7/udp.7.html).
    - **MSG\_NOSIGNAL** (since Linux 2.2) Don't generate a **SIGPIPE** signal if the peer on a stream-oriented socket has closed the connection. The **EPIPE** error is still returned. This provides similar behavior to using [sigaction(2)](https://man7.org/linux/man-pages/man2/sigaction.2.html) to ignore **SIGPIPE**, but, whereas **MSG\_NOSIGNAL** is a per-call feature, ignoring **SIGPIPE** setts a process attribute that affects all threads in the process.
    - **MSG\_OOB:** Sends *out-of-band* data on sockets that support this notion (e.g., of type **SOCK\_STREAM**); the underlying protocol must also support *out-of-band* data.
    - **MSG\_FASTOPEN** : Attempts TCP Fast Open (RFC7413) and sends data in the SYN like a combination of [connect(2)](https://man7.org/linux/man-pages/man2/connect.2.html) and [write(2)](https://man7.org/linux/man-pages/man2/write.2.html), by performing an implicit [connect(2)](https://man7.org/linux/man-pages/man2/connect.2.html) operation. It blocks until the data is buffered and the handshake has completed. For a non-blocking socket, it returns the number of bytes buffered and sent in the SYN packet. If the cookie is not available locally, it returns **EINPROGRESS**, and sends a SYN with a Fast Open cookie request automatically. The caller needs to write the data again when the socket is connected. On errors, it set the same [*errno*](https://man7.org/linux/man-pages/man3/errno.3.html) as [connect(2)](https://man7.org/linux/man-pages/man2/connect.2.html) if the handshake fails. The flag requires enabling TCP Fast Open client support on sysctl *net.ipv4.tcp\_fastopen*. Refer to **TCP\_FASTOPEN\_CONNECT** socket option in [tcp(7)](https://man7.org/linux/man-pages/man7/tcp.7.html) for an alternative approach.
    - **sendmsg()**: The definition of the *msghdr* structure employed by **sendmsg**() is as follows:
      * struct msghdr {
        + void \*msg\_name; /\* Optional address \*/
        + socklen\_t msg\_namelen; /\* Size of address \*/
        + struct iovec \*msg\_iov; /\* Scatter/gather array \*/
        + size\_t msg\_iovlen; /\* # elements in msg\_iov \*/
        + void \*msg\_control; /\* Ancillary data, see below \*/
        + size\_t msg\_controllen; /\* Ancillary data buffer len \*/
        + int msg\_flags; /\* Flags (unused) \*/
      * };
      * The *msg\_name* field is used on an unconnected socket to specify the target address for a datagram. It points to a buffer containing the address; the *msg\_namelen* field should be set to the size of the address. For a connected socket, these fields should be specified as NULL and 0, respectively.
      * The *msg\_iov* and *msg\_iovlen* fields specify scatter-gather locations, as for [writev(2)](https://man7.org/linux/man-pages/man2/writev.2.html).
      * You may send control information (ancillary data) using the *msg\_control* and *msg\_controllen* members. The maximum control buffer length the kernel can process is limited per socket by the value in */proc/sys/net/core/optmem\_max*; see [socket(7)](https://man7.org/linux/man-pages/man7/socket.7.html). For further information on the use of ancillary data in various socket domains, see [unix(7)](https://man7.org/linux/man-pages/man7/unix.7.html) and [ip(7)](https://man7.org/linux/man-pages/man7/ip.7.html).
      * The *msg\_flags* field is ignored.

**src/tcp\_client.cpp:TcpClient::** **subscribe(const client\_observer\_t & observer):**

* Locks the server member variable: \_subscribersMtx with:
  + std::lock\_guard<std::mutex> lock(\_subscriberMtx)
* **The \_subscribers TcpClient member variable vector adds the observer to it.**

**src/tcp\_client.cpp:TcpClient::** **publishServerMsg(const char \* msg, size\_t msgSize):**

* This publishes the incomingPacketHandler client message to the observer. Observers get only messages that originated from clients with IP addresses identical to the specific observer requested IP.
* Locks the server member variable: \_subscribersMtx with:
  + std::lock\_guard<std::mutex> lock(\_subscriberMtx)
* A for loop is started to loop on the client subscriber vector list member variable: \_subscribers:
  + If the subscriber.incomingPacketHandler is defined:
    - A call is made to: subscriber.incomingPacketHandler(msg. msgSize)

**src/tcp\_client.cpp:TcpClient::** **publishServerDisconnected(const pipe\_ret\_t & ret):**

* Publish Client disconnection to observer. Observers only get notified about clients with an IP address identical to the specific observer requested IP.
* Locks the server member variable: \_subscribersMtx with:
  + std::lock\_guard<std::mutex> lock(\_subscriberMtx)
* A for loop is started to loop on the server observer subscriber vector list member variable: \_subscribers:
  + If the subscriber.disconnectionHandler is defined:
    - A call is made to: subscriber.disconnectionHandler(ret)

**src/tcp\_client.cpp:TcpClient::** **receiveTask():**

* **The TcpClient receives server packets and notifies the user.**
* **A while loop is started on TcpClient::isConnected() being true:** std::atomic<bool> \_isConnected: The main characteristic of atomic objects is that access to this contained value from different threads cannot cause data races (i.e., doing that is *well-defined behavior*, with accesses properly sequenced). Generally, for all other objects, the possibility of causing a data race for accessing the same object concurrently qualifies the operation as *undefined behavior*.
* src/common.cpp: in namespace fd\_wait:Result waitResult = fd\_wait:waitFor(\_sockfd)
  + If waitResult == FAILURE, a run time error is thrown.
  + If waitResult == TIMEOUT, the while loop continues from the top.
* char msg[MAX\_PACKET\_SIZE]
* **const size\_t numOfBytesReceived = recv(\_sockfd.get(), msg, MAX\_PACKET\_SIZE, 0)**
* **If the numBytesReceived is less than 1 byte, a std::string errorMsg is created.**
  + - If the numBytesReceived == 0, the server closed the connection goes into the errorMsg. Otherwise, the errorMsg gets the errno string error message.
    - The TcpClient member variable \_isConnected is set to FALSE.
    - **TcpClient::publishServerDisconnected()** is called with the failure error message (described just above) and return called.
* **If the numBytesReceived is greater than or equal to 1 byte:**
  + - **TcpClient::publishServerMsg( msg, numBytesReceived)** is called, described just above.
* **Note:**  The **recv**(), **recvfrom**(), and **recvmsg**() calls are used to receive

messages from a socket. They may be used to receive data on both

connectionless and connection-oriented sockets. This page first

describes common features of all three system calls, and then

describes the differences between the calls.

The only difference between **recv**() and [read(2)](https://man7.org/linux/man-pages/man2/read.2.html) is the presence of

*flags*. With a zero *flags* argument, **recv**() is generally

equivalent to [read(2)](https://man7.org/linux/man-pages/man2/read.2.html) (but see NOTES). Also, the following call

recv(sockfd, buf, len, flags);

is equivalent to

recvfrom(sockfd, buf, len, flags, NULL, NULL);

All three calls return the length of the message on successful

completion. If a message is too long to fit in the supplied

buffer, excess bytes may be discarded depending on the type of

socket the message is received from.

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)](https://man7.org/linux/man-pages/man2/fcntl.2.html)), in which case the value -1 is returned and [*errno*](https://man7.org/linux/man-pages/man3/errno.3.html)

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.

An application can use [select(2)](https://man7.org/linux/man-pages/man2/select.2.html), [poll(2)](https://man7.org/linux/man-pages/man2/poll.2.html), or [epoll(7)](https://man7.org/linux/man-pages/man7/epoll.7.html) to

determine when more data arrives on a socket.

**The flags argument**

The *flags* argument is formed by ORing one or more of the

following values:

**MSG\_CMSG\_CLOEXEC** (**recvmsg**() only; since Linux 2.6.23)

Set the close-on-exec flag for the file descriptor

received via a UNIX domain file descriptor using the

**SCM\_RIGHTS** operation (described in [unix(7)](https://man7.org/linux/man-pages/man7/unix.7.html)). This flag is

useful for the same reasons as the **O\_CLOEXEC** flag of

[open(2)](https://man7.org/linux/man-pages/man2/open.2.html).

**MSG\_DONTWAIT** (since Linux 2.2)

Enables nonblocking operation; if the operation would

block, the call fails with the error **EAGAIN** or

**EWOULDBLOCK**. This provides similar behavior to setting

the **O\_NONBLOCK** flag (via the [fcntl(2)](https://man7.org/linux/man-pages/man2/fcntl.2.html) **F\_SETFL** operation),

but differs in that **MSG\_DONTWAIT** is a per-call option,

whereas **O\_NONBLOCK** is a setting on the open file

description (see [open(2)](https://man7.org/linux/man-pages/man2/open.2.html)), which will affect all threads

in the calling process and as well as other processes that

hold file descriptors referring to the same open file

description.

**MSG\_ERRQUEUE** (since Linux 2.2)

This flag specifies that queued errors should be received

from the socket error queue. The error is passed in an

ancillary message with a type dependent on the protocol

(for IPv4 **IP\_RECVERR**). The user should supply a buffer of

sufficient size. See [cmsg(3)](https://man7.org/linux/man-pages/man3/cmsg.3.html) and [ip(7)](https://man7.org/linux/man-pages/man7/ip.7.html) for more

information. The payload of the original packet that

caused the error is passed as normal data via *msg\_iovec*.

The original destination address of the datagram that

caused the error is supplied via *msg\_name*.

The error is supplied in a *sock\_extended\_err* structure:

#define SO\_EE\_ORIGIN\_NONE 0

#define SO\_EE\_ORIGIN\_LOCAL 1

#define SO\_EE\_ORIGIN\_ICMP 2

#define SO\_EE\_ORIGIN\_ICMP6 3

struct sock\_extended\_err

{

uint32\_t ee\_errno; /\* Error number \*/

uint8\_t ee\_origin; /\* Where the error originated \*/

uint8\_t ee\_type; /\* Type \*/

uint8\_t ee\_code; /\* Code \*/

uint8\_t ee\_pad; /\* Padding \*/

uint32\_t ee\_info; /\* Additional information \*/

uint32\_t ee\_data; /\* Other data \*/

/\* More data may follow \*/

};

struct sockaddr \*SO\_EE\_OFFENDER(struct sock\_extended\_err \*);

*ee\_errno* contains the [*errno*](https://man7.org/linux/man-pages/man3/errno.3.html) number of the queued error.

*ee\_origin* is the origin code of where the error

originated. The other fields are protocol-specific. The

macro **SO\_EE\_OFFENDER** returns a pointer to the address of

the network object where the error originated from given a

pointer to the ancillary message. If this address is not

known, the *sa\_family* member of the *sockaddr* contains

**AF\_UNSPEC** and the other fields of the *sockaddr* are

undefined. The payload of the packet that caused the

error is passed as normal data.

For local errors, no address is passed (this can be

checked with the *cmsg\_len* member of the *cmsghdr*). For

error receives, the **MSG\_ERRQUEUE** flag is set in the

*msghdr*. After an error has been passed, the pending

socket error is regenerated based on the next queued error

and will be passed on the next socket operation.

**MSG\_OOB**

This flag requests receipt of out-of-band data that would

not be received in the normal data stream. Some protocols

place expedited data at the head of the normal data queue,

and thus this flag cannot be used with such protocols.

**MSG\_PEEK**

This flag causes the receive operation to return data from

the beginning of the receive queue without removing that

data from the queue. Thus, a subsequent receive call will

return the same data.

**MSG\_TRUNC** (since Linux 2.2)

For raw (**AF\_PACKET**), Internet datagram (since Linux

2.4.27/2.6.8), netlink (since Linux 2.6.22), and UNIX

datagram as well as sequenced-packet (since Linux 3.4)

sockets: return the real length of the packet or datagram,

even when it was longer than the passed buffer.

For use with Internet stream sockets, see [tcp(7)](https://man7.org/linux/man-pages/man7/tcp.7.html).

**MSG\_WAITALL** (since Linux 2.2)

This flag requests that the operation block until the full

request is satisfied. However, the call may still return

less data than requested if a signal is caught, an error

or disconnect occurs, or the next data to be received is

of a different type than that returned. This flag has no

effect for datagram sockets.

**recvfrom()**

**recvfrom**() places the received message into the buffer *buf*. The

caller must specify the size of the buffer in *len*.

If *src\_addr* is not NULL, and the underlying protocol provides the

source address of the message, that source address is placed in

the buffer pointed to by *src\_addr*. In this case, *addrlen* is a

value-result argument. Before the call, it should be initialized

to the size of the buffer associated with *src\_addr*. Upon return,

*addrlen* is updated to contain the actual size of the source

address. The returned address is truncated if the buffer

provided is too small; in this case, *addrlen* will return a value

greater than was supplied to the call.

If the caller is not interested in the source address, *src\_addr*

and *addrlen* should be specified as NULL.

**recv()**

The **recv**() call is normally used only on a *connected* socket (see

[connect(2)](https://man7.org/linux/man-pages/man2/connect.2.html)). It is equivalent to the call:

recvfrom(fd, buf, len, flags, NULL, 0);

**recvmsg()**

The **recvmsg**() call uses a *msghdr* structure to minimize the number

of directly supplied arguments. This structure is defined as

follows in *<sys/socket.h>*:

struct msghdr {

void \*msg\_name; /\* Optional address \*/

socklen\_t msg\_namelen; /\* Size of address \*/

struct iovec \*msg\_iov; /\* Scatter/gather array \*/

size\_t msg\_iovlen; /\* # elements in msg\_iov \*/

void \*msg\_control; /\* Ancillary data, see below \*/

size\_t msg\_controllen; /\* Ancillary data buffer len \*/

int msg\_flags; /\* Flags on received message \*/

};

The *msg\_name* field points to a caller-allocated buffer that is

used to return the source address if the socket is unconnected.

The caller should set *msg\_namelen* to the size of this buffer

before this call; upon return from a successful call, *msg\_namelen*

will contain the length of the returned address. If the

application does not need to know the source address, *msg\_name*

can be specified as NULL.

The fields *msg\_iov* and *msg\_iovlen* describe scatter-gather

locations, as discussed in [readv(2)](https://man7.org/linux/man-pages/man2/readv.2.html).

The field *msg\_control*, which has length *msg\_controllen*, points to

a buffer for other protocol control-related messages or

miscellaneous ancillary data. When **recvmsg**() is called,

*msg\_controllen* should contain the length of the available buffer

in *msg\_control*; upon return from a successful call it will

contain the length of the control message sequence.

The messages are of the form:

struct cmsghdr {

size\_t cmsg\_len; /\* Data byte count, including header

(type is socklen\_t in POSIX) \*/

int cmsg\_level; /\* Originating protocol \*/

int cmsg\_type; /\* Protocol-specific type \*/

/\* followed by

unsigned char cmsg\_data[]; \*/

};

Ancillary data should be accessed only by the macros defined in

[cmsg(3)](https://man7.org/linux/man-pages/man3/cmsg.3.html).

As an example, Linux uses this ancillary data mechanism to pass

extended errors, IP options, or file descriptors over UNIX domain

sockets. For further information on the use of ancillary data in

various socket domains, see [unix(7)](https://man7.org/linux/man-pages/man7/unix.7.html) and [ip(7)](https://man7.org/linux/man-pages/man7/ip.7.html).

The *msg\_flags* field in the *msghdr* is set on return of **recvmsg**().

It can contain several flags:

**MSG\_EOR**

indicates end-of-record; the data returned completed a

record (generally used with sockets of type

**SOCK\_SEQPACKET**).

**MSG\_TRUNC**

indicates that the trailing portion of a datagram was

discarded because the datagram was larger than the buffer

supplied.

**MSG\_CTRUNC**

indicates that some control data was discarded due to lack

of space in the buffer for ancillary data.

**MSG\_OOB**

is returned to indicate that expedited or out-of-band data

was received.

**MSG\_ERRQUEUE**

indicates that no data was received but an extended error

from the socket error queue.

**src/tcp\_client.cpp:TcpClient::** **terminateReceiveThread():**

* The TcpClient member variable \_isConnected is set to FALSE.
* If the TcpClient member variable \_receiveTask is defined:
  + - \_receiveTask->join() is called to wait for the \_receiveTask to finish running its thread.
    - \_receiveTask is then deleted and set to the nullptr.

**src/tcp\_client.cpp:TcpClient::close():**

* If the TcpClient member variable \_isClosed is not zero, a failure message of “client is already closed” is returned.
* TcpClient::terminateReceiveThread() is called (described above).
* A boolean is created closedFailed = (::close(\_sockfd.get()) == -1) :
  + See a detailed description of: ::close( \_sockfd.get()) above in TcpServer::close(). The client’s socket is closed. If it fails, failure is returned but if it passes, the member variable \_isClosed is set to TRUE, and success returned.

**src/common.cpp:**

**src/common.cpp: in namespace fd\_wait: waitFor(const FileDescriptor &fileDescriptor, uint32\_t timeoutSeconds):**

* **Monitors file descriptor and waits for an I/O operation**
* **It uses the struct timeval tv to set:**
  + - tv.tv\_sec = timeoutSeconds;
    - tv.tv\_usec = 0;
    - fd\_set fds;
    - FD\_ZERO(&fds);
    - FD\_SET(fileDescriptor.get(), &fds);
* **const int selectRet = select(fileDescriptor.get () + 1, &fds, nullptr, nullptr, &tv)**
  + - **One of these is returned to the caller: FAILURE, TIMEOUT or SUCCESS.**
* **Note: select**() and **pselect**() allow a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some class of I/O operation (e.g., input possible). A file descriptor is considered ready if it is possible to perform the corresponding I/O operation (e.g., [**read**](https://linux.die.net/man/2/read)(2)) without blocking.
* The operation of **select**() and **pselect**() is identical, other than these three differences:
* (i)
* **select**() uses a timeout that is a *struct timeval* (with seconds and microseconds), while **pselect**() uses a *struct timespec* (with seconds and nanoseconds).
* (ii)
* **select**() may update the *timeout* argument to indicate how much time was left. **pselect**() does not change this argument.
* (iii)
* **select**() has no *sigmask* argument, and behaves as **pselect**() called with NULL *sigmask*.
* Three independent sets of file descriptors are watched. Those listed in *readfds* will be watched to see if characters become available for reading (more precisely, to see if a read will not block; in particular, a file descriptor is also ready on end-of-file), those in *writefds* will be watched to see if a write will not block, and those in *exceptfds* will be watched for exceptions. On exit, the sets are modified in place to indicate which file descriptors actually changed status. Each of the three file descriptor sets may be specified as NULL if no file descriptors are to be watched for the corresponding class of events.
* Four macros are provided to manipulate the sets. **FD\_ZERO**() clears a set. **FD\_SET**() and **FD\_CLR**() respectively add and remove a given file descriptor from a set. **FD\_ISSET**() tests to see if a file descriptor is part of the set; this is useful after **select**() returns.
* *nfds* is the highest-numbered file descriptor in any of the three sets, plus 1.
* The *timeout* argument specifies the minimum interval that **select**() should block waiting for a file descriptor to become ready. (This interval will be rounded up to the system clock granularity, and kernel scheduling delays mean that the blocking interval may overrun by a small amount.) If both fields of the *timeval* structure are zero, then **select**() returns immediately. (This is useful for polling.) If *timeout* is NULL (no timeout), **select**() can block indefinitely.
* *sigmask* is a pointer to a signal mask (see [**sigprocmask**](https://linux.die.net/man/2/sigprocmask)(2)); if it is not NULL, then **pselect**() first replaces the current signal mask by the one pointed to by *sigmask*, then does the "select" function, and then restores the original signal mask.
* Other than the difference in the precision of the *timeout* argument, the following **pselect**() call:
* ready = pselect(nfds, &readfds, &writefds, &exceptfds,

timeout, &sigmask);

* is equivalent to *atomically* executing the following calls:
* sigset\_t origmask;
* pthread\_sigmask(SIG\_SETMASK, &sigmask, &origmask);
* ready = select(nfds, &readfds, &writefds, &exceptfds, timeout);
* pthread\_sigmask(SIG\_SETMASK, &origmask, NULL);
* The reason that **pselect**() is needed is that if one wants to wait for either a signal or for a file descriptor to become ready, then an atomic test is needed to prevent race conditions. (Suppose the signal handler sets a global flag and returns. Then a test of this global flag followed by a call of **select**() could hang indefinitely if the signal arrived just after the test but just before the call. By contrast, **pselect**() allows one to first block signals, handle the signals that have come in, then call **pselect**() with the desired *sigmask*, avoiding the race.)
* **The timeout**
* The time structures involved are defined in *<*[*sys/time.h*](https://linux.die.net/include/sys/time.h)*>* and look like

struct timeval {

long tv\_sec; /\* seconds \*/

long tv\_usec; /\* microseconds \*/

};

* and

struct timespec {

long tv\_sec; /\* seconds \*/

long tv\_nsec; /\* nanoseconds \*/

};

* (However, see below on the POSIX.1-2001 versions.)
* Some code calls **select**() with all three sets empty, *nfds* zero, and a non-NULL *timeout* as a fairly portable way to sleep with subsecond precision.
* On Linux, **select**() modifies *timeout* to reflect the amount of time not slept; most other implementations do not do this. (POSIX.1-2001 permits either behavior.) This causes problems both when Linux code which reads *timeout* is ported to other operating systems, and when code is ported to Linux that reuses a *struct timeval* for multiple **select**()s in a loop without reinitializing it. Consider *timeout* to be undefined after **select**() returns.

**src/client.cpp:**

**src/client.cpp:Client::Client(int fileDescriptor): constructor:**

* The Client member variable \_sockfd.set(fileDescriptor) is called.
* Client::setConnected(false) is called.

**src/client.cpp:Client::** **operator==(const Client & other) const:**

* If the other Client’s \_sockfd.get() equals this client’s \_sockfd.get() and both the other.\_ip address is equal to this clients this.\_ip address, then TRUE is returned, otherwise FALSE is returned.

**src/client.cpp:Client::** **startListen():**

* **Client::setConnected(TRUE) is called.**
* **\_receiveThread = new std::thread(&Client::receiveTask, this) is called:**
* **See src/tcp\_client.cpp:TcpClient::** **startReceivingMessages() above for a detailed description of std::thread.**

**src/client.cpp:Client::** **send(const char \*msg, size\_t msgSize) const:**

* **const size\_t numBytesSent = ::send(\_sockfd.get(), (char \*)msg, msgSize, 0)**
* See src/tcp\_client.cpp:TcpClient:: sendMsg(const char \* msg, size\_t size)—for a detailed description of ::send() in the line above.
* If the number of bytes sent is less than zero or if less than the number of bytes requested in the size parameter, an error message is printed.

**src/client.cpp:Client::** **receiveTask(): Note that receiveTask() is privately defined by Client (as is the TcpClient’s receiveTask()).**

* **The Client receives packets and notifies the user.**
* **A while loop is started on Client::isConnected() being true.**
* std::atomic<bool> \_isConnected: The main characteristic of atomic objects is that access to this contained value from different threads cannot cause data races (i.e., doing that is *well-defined behavior*, with accesses properly sequenced). Generally, for all other objects, the possibility of causing a data race for accessing the same object concurrently qualifies the operation as *undefined behavior*.
* src/common.cpp: in namespace fd\_wait:Result waitResult = fd\_wait:waitFor(\_sockfd)
  + If waitResult == FAILURE, a run time error is thrown.
  + If waitResult == TIMEOUT, the while loop continues from the top.
* char receivedMessage[MAX\_PACKET\_SIZE]
* **const size\_t numOfBytesReceived = recv(\_sockfd.get(), receivedMessage, MAX\_PACKET\_SIZE, 0)**
* **If the numBytesReceived is less than 1 byte, a std::string disconnectionMsg is created.**
  + - If the numBytesReceived == 0, the client closed the connection goes into the disconnectionMsg. Otherwise, disconnection Msg gets the errno string disconnection message.
    - Client::setConnected(FALSE): The Client member variable \_isConnected is set to FALSE.
    - **Client::publishEvent**(ClientEvent::DISCONNECTED, disconnectionMessage) is called with the disconnection message (described just below) and return called.
* **If the numBytesReceived is greater than or equal to 1 byte:**
  + - **Client::publishEvent(ClientEvent::INCOMING\_MSG, receivedMessage)** is called, described just below.

**src/client.cpp:Client::** **publishEvent(ClientEvent clientEvent, const std::string &msg):**

* **The Client member variable handler is called with:** 
  + **\_eventHandlerCallback(\*this, clientEvent, msg)**

**src/client.cpp:Client::** **print() const:**

* **The following information is printed:**
  + **IP address: by calling: Client::getIp()**
  + **Connected: by calling: Client::isConnected()**
  + **Socket File Descriptor (FD) by calling: Client::\_sockfd.get()**

**src/client.cpp:Client::** **terminateReceiveThread():**

* Client::setConnected(FALSE) is called.
* If the Client member variable \_receiveThread is defined:
  + - \_receiveThread->join() is called to wait for the \_receiveThread to finish running its thread.
    - \_receiveThread is then deleted and set to the nullptr.

**src/client.cpp:Client::** **close():**

* If the Client member variable \_isClosed is not zero, a failure message of “client is already closed” is returned.
* Client::terminateReceiveThread() is called (described above).
* A boolean is created closedFailed = (::close(\_sockfd.get()) == -1) :
  + See a detailed description of: ::close( \_sockfd.get()) above in TcpServer::close(). The client’s socket is closed. If it fails, failure is returned but if it passes, the member variable \_isClosed is set to TRUE, and success returned.

**src/pipe\_ret\_t.cpp:**

* **Note:** C++ allows defining static data members within a class using the static keyword.  
  When a data member is declared as static, then we must keep the following note in mind:
* Irrespective of the number of objects created, ***only a single copy of the static member is created in memory.***
* ***All objects of a class share the static member.***
* ***All static data members are initiated to zero when the first object of that class is created.***
* ***Static data members are visible only within the class but their lifetime is the entire program.***
* **Relevance:** Static data members are usually used to maintain values that are common for the entire class. **, For Example,** to keep a track of how many objects of a particular class have been created.
* **Place of Storage:** Although static data members are declared inside a class, they are not considered to be a part of the objects. Consequently, their declaration in the class is not considered as their definition. A static data member is defined outside the class. This means that even though the static data member is declared in class scope, their definition persists in the entire file. A static member has file scope. However, since a static data member is declared inside the class, they can be accessed only by using the class name and the scope resolution operator.

**src/pipe\_ret\_t.cpp:** **pipe\_ret\_t::failure(const std::string &msg):**

* This returns the static pipe\_ret\_t value failure with the \_successFlag set to FALSE and the msg set to what is passed into this call.

**src/pipe\_ret\_t.cpp:** **pipe\_ret\_t::** **success(const std::string &msg):**

* This returns the static pipe\_ret\_t value success with the \_successFlag set to TRUE and the msg set to what is passed into this call.

**include files:**

* **client.h: defines the class “Client”:**

#pragma once

#include <string>

#include <thread>

#include <functional>

#include <mutex>

#include <atomic>

#include "pipe\_ret\_t.h"

#include "client\_event.h"

#include "file\_descriptor.h"

**class** **Client** {

**using client\_event\_handler\_t** = std::function<void(const Client&, ClientEvent, const std::string&)>;

**private**:

FileDescriptor \_sockfd;

std::string **\_ip** = "";

std::atomic<bool> **\_isConnected**;

std::thread \* **\_receiveThread** = nullptr;

client\_event\_handler\_t **\_eventHandlerCallback**;

void **setConnected**(bool flag) { \_isConnected = flag; }

void **receiveTask**();

void **terminateReceiveThread**();

**public**:

**Client**(int);

bool **operator ==**(const Client & other) const ;

void **setIp**(const std::string & ip) { \_ip = ip; }

std::string **getIp**() const { return \_ip; }

void **setEventsHandler**(const client\_event\_handler\_t & eventHandler) { \_eventHandlerCallback = eventHandler; }

void **publishEvent**(ClientEvent clientEvent, const std::string &msg = "");

bool **isConnected**() const { return \_isConnected; }

void **startListen**();

void **send**(const char \* msg, size\_t msgSize) const;

void **close**();

void **print**() const;

};

* **client\_event.h: defines the enum ClientEvent:**

#pragma once

**enum ClientEvent** {

DISCONNECTED,

INCOMING\_MSG

};

* **client\_observer.h: defines the structure of a client observer: the wantedIP address, function template definitions for the incomingPacketHandler and the disconnectionHandler:**

#pragma once

#include <string>

#include <functional>

#include "pipe\_ret\_t.h"

**struct client\_observer\_t** {

std::string wantedIP = "";

std::function<void(const char \* msg, size\_t size)> incomingPacketHandler = nullptr;

std::function<void(const pipe\_ret\_t & ret)> disconnectionHandler = nullptr;

};

* **common.h: defines the MAX\_PACKET\_SIZE, the enum Result for the fd\_wait(), and the definition header of the waitFor() function.**

#pragma once

#include <cstdio>

#define MAX\_PACKET\_SIZE 4096

**namespace fd\_wait** {

**enum Result** {

FAILURE,

TIMEOUT,

SUCCESS

};

Result **waitFor**(const FileDescriptor &fileDescriptor, uint32\_t timeoutSeconds = 1);

};

* **file\_descriptior.h: defines the FileDescriptor class with a private \_sockfd, set and get public functions for it:**

#pragma once

**class** **FileDescriptor** {

**private:**

int \_sockfd = 0;

**public**:

void set(int fd) { \_sockfd = fd; }

int get() const { return \_sockfd; }

};

* **pipe\_ret\_t.h: defines the class pipe\_ret\_t that contains private variables for \_successFlag and \_msg. It has in its public space: multiple constructors (default and one that initializes the member variables), message() to return the message() private variable, isSuccessful() to return the private \_successFlag. It has static functions for returning failure() and success() with the msg string.**

#pragma once

**class pipe\_ret\_t** {

**private**:

bool **\_successFlag** = false;

std::string **\_msg** = "";

**public:**

**pipe\_ret\_t()** = default;

**pipe\_ret\_t(bool successFlag, const std::string &msg) :**

**\_successFlag{successFlag},**

**\_msg{msg}**

{}

std::string **message()** const { return \_msg; }

bool **isSuccessful()** const { return \_successFlag; }

**static pipe\_ret\_t failure**(const std::string & msg);

**static pipe\_ret\_t success**(const std::string &msg = "");

};

* **server\_observer.h**: **defines the structure of a server observer: the wantedIP address, function template definitions for the incomingPacketHandler and the disconnectionHandler: Note that it is different than the client\_observer.h definitions of this.**

#pragma once

#include <string>

#include <functional>

#include "client.h"

**struct server\_observer\_t** {

std::string **wantedIP** = "";

std::function<void(const std::string &clientIP, const char \* msg, size\_t size)> **incomingPacketHandler;**

std::function<void(const std::string &ip, const std::string &msg)> **disconnectionHandler;**

};

* **tcp\_client.h: defines the TcpClient class:**

#pragma once

#include <iostream>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <netdb.h>

#include <netdb.h>

#include <vector>

#include <errno.h>

#include <thread>

#include <mutex>

#include <atomic>

#include "client\_observer.h"

#include "pipe\_ret\_t.h"

#include "file\_descriptor.h"

**class TcpClient**

{

**private**:

FileDescriptor \_sockfd;

std::atomic<bool> \_isConnected;

std::atomic<bool> \_isClosed;

struct sockaddr\_in \_server;

std::vector<client\_observer\_t> \_subscibers;

std::thread \* \_receiveTask = nullptr;

std::mutex \_subscribersMtx;

void initializeSocket();

void startReceivingMessages();

void setAddress(const std::string& address, int port);

void publishServerMsg(const char \* msg, size\_t msgSize);

void publishServerDisconnected(const pipe\_ret\_t & ret);

void receiveTask();

void terminateReceiveThread();

**public**:

TcpClient();

~TcpClient();

pipe\_ret\_t connectTo(const std::string & address, int port);

pipe\_ret\_t sendMsg(const char \* msg, size\_t size);

void subscribe(const client\_observer\_t & observer);

bool isConnected() const { return \_isConnected; }

pipe\_ret\_t close();

};

* **tcp\_server.h: defines the TcpServer class**
  + added this private variable: std::string \_observerMsgs[3];

#pragma once

#include <vector>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <thread>

#include <functional>

#include <cstring>

#include <errno.h>

#include <iostream>

#include <mutex>

#include "client.h"

#include "server\_observer.h"

#include "pipe\_ret\_t.h"

#include "file\_descriptor.h"

**class TcpServer** {

**private**:

FileDescriptor \_sockfd;

struct sockaddr\_in \_serverAddress;

struct sockaddr\_in \_clientAddress;

fd\_set \_fds;

std::vector<Client\*> \_clients;

std::vector<server\_observer\_t> \_subscribers;

std::mutex \_subscribersMtx;

std::mutex \_clientsMtx;

std::thread \* \_clientsRemoverThread = nullptr;

std::atomic<bool> \_stopRemoveClientsTask;

std::string \_observerMsgs[3];

void publishClientMsg(const Client & client, const char \* msg, size\_t msgSize);

void publishClientDisconnected(const std::string&, const std::string&);

pipe\_ret\_t waitForClient(uint32\_t timeout);

void clientEventHandler(const Client&, ClientEvent, const std::string &msg);

void removeDeadClients();

void terminateDeadClientsRemover();

static pipe\_ret\_t sendToClient(const Client & client, const char \* msg, size\_t size);

**public**:

TcpServer();

~TcpServer();

pipe\_ret\_t start(int port, int maxNumOfClients = 5, bool removeDeadClientsAutomatically = true);

void initializeSocket();

void bindAddress(int port);

void listenToClients(int maxNumOfClients);

std::string acceptClient(uint timeout);

void subscribe(const server\_observer\_t & observer);

pipe\_ret\_t sendToAllClients(const char \* msg, size\_t size);

pipe\_ret\_t sendToClient(const std::string & clientIP, const char \* msg, size\_t size);

pipe\_ret\_t close();

void printClients();

void printClientMsgsByPriorityOfId();

};

**build directory: Cmake tutorial notes follow below**

* CMakefiles directory: This contains the following:
  + 3.22.1 (version in my Ubuntu Linux for Cmake) directory
  + CMakeTemp directory
  + Tcp\_client.dir directory
  + Tcp\_client\_server.dir directory
  + Tcp\_server.dir directory
  + Cmake.check\_cache file
  + CMakeDirectoryInformation file
  + CMakeOutput file
  + Makefile file
  + Makefile2 file
  + progess.marks file
  + TargetDirectories file
* cmake\_install file
* CMakeCache file
* libtcp\_client\_server.a **static library file**:
* --in this file toward the bottom, you see the objects that make the static library in the build: build/CMakeFiles/tcp\_client\_server.dir/build:

* + libtcp\_client\_server.a: CMakeFiles/tcp\_client\_server.dir/src/tcp\_client.cpp.o
  + libtcp\_client\_server.a: CMakeFiles/tcp\_client\_server.dir/src/tcp\_server.cpp.o
  + libtcp\_client\_server.a: CMakeFiles/tcp\_client\_server.dir/src/client.cpp.o
  + libtcp\_client\_server.a: CMakeFiles/tcp\_client\_server.dir/src/pipe\_ret\_t.cpp.o
  + libtcp\_client\_server.a: CMakeFiles/tcp\_client\_server.dir/src/common.cpp.o
* Makefile file
* tcp\_client and tcp\_server executable binary files—these are shown in the snips when I ran them for this coding challenge.
* **Cmake Info: tutorial below taken from here:** 
  + <https://medium.com/@onur.dundar1/cmake-tutorial-585dd180109b>

Introduction

CMake is an extensible, open-source system that manages the build process in an operating system and in a compiler-independent manner.

Unlike many cross-platform systems, CMake is designed to be used in conjunction with the native build environment. Simple configuration files placed in each source directory (called CMakeLists.txt files) are used to generate standard build files (e.g., Makefiles on Unix and projects/workspaces in Windows MSVC) which are used in the usual way.

CMake can generate a native build environment that will compile source code, create libraries, generate wrappers and build executable binaries in arbitrary combinations. CMake supports in-place and out-of-place builds, and can therefore support multiple builds from a single source tree.

CMake has supports for static and dynamic library builds. Another nice feature of CMake is that it can generates a cache file that is designed to be used with a graphical editor. For example, while CMake is running, it locates include files, libraries, and executables, and may encounter optional build directives. This information is gathered into the cache, which may be changed by the user prior to the generation of the native build files.

CMake scripts also make source management easier because it simplifies build script into one file and more organized, readable format.

**Popular Open Source Project with CMake**

Here is a list of popular open source projects using CMake for build purposes:

* OpenCV: <https://github.com/opencv/opencv>
* Caffe2: <https://github.com/caffe2/caffe2>
* MySql Server: <https://github.com/mysql/mysql-server>

See Wikipedia for a longer list <https://en.wikipedia.org/wiki/CMake#Applications_that_use_CMake>

It is always a good practice to read open source projects, which are widely used to learn about the best practices

# CMake as a Scripting Language

CMake intended to be a cross-platform build process manager so it defines it is own scripting language with certain syntax and built-in features. CMake itself is a software program, so it should be invoked with the script file to interpret and generate actual build file.

A developer can write either simple or complex building scripts using CMake language for the projects.

Build logic and definitions with CMake language is written either in CMakeLists.txt or a file ends with <project\_name>.cmake. But as a best practice, main script is named as CMakeLists.txt instead of cmake.

* CMakeLists.txt file is placed at the source of the project you want to build.
* CMakeLists.txt is placed at the root of the source tree of any application, library it will work for.
* If there are multiple modules, and each module can be compiled and built separately, CMakeLists.txt can be inserted into the sub folder.
* .cmake files can be used as scripts, which runs cmake command to prepare environment pre-processing or split tasks which can be written outside of CMakeLists.txt.
* .cmake files can also define modules for projects. These projects can be separated build processes for libraries or extra methods for complex, multi-module projects.
* Writing Makefiles might be harder than writing CMake scripts. CMake scripts by syntax and logic have similarity to high level languages so it makes easier for developers to create their cmake scripts with less effort and without getting lost in Makefiles.

# CMake Commands

CMake commands are similar to C++/Java methods or functions, which take parameters as a list and perform certain tasks accordingly.

CMake commands are case insensitive. There are built-in commands, which can be found from cmake documentation: <https://cmake.org/cmake/help/latest/manual/cmake-commands.7.html>

Some commonly used commands

* message: prints given message
* [cmake\_minimum\_required](https://cmake.org/cmake/help/latest/command/cmake_minimum_required.html): sets minimum version of cmake to be used
* add\_executable: adds executable target with given name
* add\_library: adds a library target to be build from listed source files
* add\_subdirectory: adds a subdirectory to build

There are also commands to enable developers write out conditional statements, loops, iterate on list, assignments:

* if, endif
* elif, endif
* while, endwhile
* foreach, endforeach
* list
* return
* set\_property (assign value to variable.)

Indentation is not mandatory but suggested while writing CMake scripts. CMake doesn’t use ‘;’ to understand end of statement.

All conditional statements should be ended with its corresponding end command (endif, endwhile, endforeach, etc)

All these properties of CMake help developers to program complex build processes including multiple modules, libraries and platforms.

For example, KDE has its own CMake style guideline as in following URL:

* <https://community.kde.org/Policies/CMake_Coding_Style>

# CMake Environment Variables

Environment variables are used to configure compiler flags, linker flags, test configurations for a regular build process. Compiler have to be guided to search for given directories for libraries.

A detailed list of environment variables can be seen from following URL:

* <https://cmake.org/cmake/help/latest/manual/cmake-env-variables.7.html>

Some of the environment variables are overriden by predefined CMake Variables. e.g. CXXFLAGS is overriden when CMAKE\_CXX\_FLAGS is defined.

Below is an example use case, when you want to enable all warnings during compile process, you may write -Wall to build command. If you are building your code with CMake, you can add -Wall flag with using set command.

set(CMAKE\_CXX\_FLAGS "-Wall")  
# append flag, best practice, suggested, don't lose previously defined flags  
set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -Wall")

# CMake Variables

CMake includes predefined variables which are set by default as location of source tree and system components.

Variables are case-sensitive, not like commands. You can only use alpha numeric chars and underscore, dash (\_, -) in definition of variable.

You can find more details about CMake variables in the following URLs

* <https://cmake.org/cmake/help/v3.0/manual/cmake-language.7.html#variables>
* <https://cmake.org/cmake/help/v3.0/manual/cmake-variables.7.html#manual:cmake-variables(7)>

Some of the variables can be seen as below, these are predefined according to root folder:

* CMAKE\_BINARY\_DIR: Full path to top level of build tree and binary output folder, by default it is defined as top level of build tree.
* CMAKE\_HOME\_DIRECTORY: Path to top of source tree
* CMAKE\_SOURCE\_DIR: Full path to top level of source tree.
* CMAKE\_INCLUDE\_PATH: Path used to find file, path
* Variable values can be accessed with ${<variable\_name>}.
  + - message("CXX Standard: ${CMAKE\_CXX\_STANDARD}")
    - set(CMAKE\_CXX\_STANDARD 14)
* Just like above variables, you can define your own variables. You can call set command to set a value to a new variable or change value of existing variable like below:
  + set(TRIAL\_VARIABLE "VALUE")
  + message("${TRIAL\_VARIABLE}")

# CMake Lists

All values in CMake are stored as string but a string can be treated as list in certain context.

A list of elements represented as a string by concatenating elements separated by semi-column ‘;’.

set(files a.txt b.txt c.txt)  
# sets files to "a.txt;b.txt;c.txt"

In order to access the list of values you can use foreach command of CMake as following:

foreach(file ${files})  
 message("Filename: ${file}")  
endforeach()

# CMake Generator Expressions

Generator expressions are evaluated during build system generation to produce information specific to each build configuration.

Generator expressions are allowed in the context of many target properties, such as [LINK\_LIBRARIES](https://cmake.org/cmake/help/v3.3/prop_tgt/LINK_LIBRARIES.html#prop_tgt:LINK_LIBRARIES), [INCLUDE\_DIRECTORIES](https://cmake.org/cmake/help/v3.3/prop_tgt/INCLUDE_DIRECTORIES.html#prop_tgt:INCLUDE_DIRECTORIES), [COMPILE\_DEFINITIONS](https://cmake.org/cmake/help/v3.3/prop_tgt/COMPILE_DEFINITIONS.html#prop_tgt:COMPILE_DEFINITIONS) and others. They may also be used when using commands to populate those properties, such as [target\_link\_libraries()](https://cmake.org/cmake/help/v3.3/command/target_link_libraries.html#command:target_link_libraries), [target\_include\_directories()](https://cmake.org/cmake/help/v3.3/command/target_include_directories.html#command:target_include_directories), [target\_compile\_definitions()](https://cmake.org/cmake/help/v3.3/command/target_compile_definitions.html#command:target_compile_definitions) and others.

<https://cmake.org/cmake/help/v3.3/manual/cmake-generator-expressions.7.html>

# Start Building C++ Code with CMake

In previous sections, we have covered the core principles of writing CMake scripts. Now, we can continue to write actual scripts to start building C++ code.

We can just start with a basic “Hello World!” example with CMake so we wrote the following “Hello CMake!” the main.cpp file as following:

#include <iostream>  
int main() {  
 std::cout<<"Hello CMake!"<<std::endl;  
}

Our purpose is to generate a binary to print “Hello CMake!”.

If there is no CMake we can run compiler to generate us a target basically with only following commands.

$ g++ main.cpp -o cmake\_hello

CMake helps to generate bash commands with the instructions you gave, for this simple project we can just use a simple CMakeLists.txt which creates Makefile for you to build binary.

It is obvious that, for such a small project it is redundant to use CMake but when things get complicated it will help a lot.

In order to build main.cpp, using add\_executable would be enough, however, let’s keep things in order and write it with proper project name and cmake version requirement as below:

* cmake\_minimum\_required(VERSION 3.9.1)
* project(CMakeHello)
* add\_executable(cmake\_hello main.cpp)

When the script ready, you can run cmake command to generate Makefile/s for the project.

You will notice that, cmake is identifying compiler versions and configurations with default information.

$ cmake CMakeLists.txt  
-- The C compiler identification is AppleClang 9.0.0.9000039  
-- The CXX compiler identification is AppleClang 9.0.0.9000039  
-- Check for working C compiler: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/cc  
-- Check for working C compiler: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/cc -- works  
-- Detecting C compiler ABI info  
-- Detecting C compiler ABI info - done  
-- Detecting C compile features  
-- Detecting C compile features - done  
-- Check for working CXX compiler: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/c++  
-- Check for working CXX compiler: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/c++ -- works  
-- Detecting CXX compiler ABI info  
-- Detecting CXX compiler ABI info - done  
-- Detecting CXX compile features  
-- Detecting CXX compile features - done  
-- Configuring done  
-- Generating done  
-- Build files have been written to: /Users/User/Projects

/CMakeTutorial

When cmake finishes its job, Makefile will be generated together with CMakeCache.txt and some other artefects about build configuration. You can run make command to build project.

$ make all  
#or  
$ make cmake\_hello

Let’s make things little more complicated.

What if your code is depended on C++14 or later. If you look for C++14 specifications, you see that, return type deduction has been introduced with C++14, lets add a new method to our main.cpp with auto return type as below.

#include <iostream>auto sum(int a, int b){  
 return a + b;  
}int main() {  
 std::cout<<"Hello CMake!"<<std::endl;  
 std::cout<<"Sum of 3 + 4 :"<<sum(3, 4)<<std::endl;  
 return 0;

#include <iostream>auto sum(int a, int b){  
 return a + b;  
}int main() {  
 std::cout<<"Hello CMake!"<<std::endl;  
 std::cout<<"Sum of 3 + 4 :"<<sum(3, 4)<<std::endl;  
 return 0;  
}

If you try to build above code with default settings, you could get an error about auto return type because most hosts configured to work C++99 config by default. Therefore you should point your compiler to build with C++14 with setting CMAKE\_CXX\_STANDARD variable to 14. If you want to add C++14 on command line, you can set -std=c++14.

CMake generates Makefile with defaults settings which is mostly lower than C++14, so you should add 14 flag as seen below.

cmake\_minimum\_required(VERSION 3.9.1)project(CMakeHello)set(CMAKE\_CXX\_STANDARD 14)add\_executable(cmake\_hello main.cpp)

Now, you should be able to build it correctly. (cmake command would work even though you didn’t add 14 standard but make command would return error.)

CMake eases building process for you; especially when you do cross compile to make sure you are working with correct version of compiler with correct configurations, this will enable multiple developers to use same building configurations across all machines including build server and developer PC.

What if, you want to build for multiple platforms:

* generate executables for Windows, Mac and Linux separetely.
* add different macros for Linux Kernel version later than X.

Following variables can be used to check for system related information: Pasted from: <https://cmake.org/Wiki/CMake_Checking_Platform>

* CMAKE\_SYSTEM   
  the complete system name, e.g. "Linux-2.4.22", "FreeBSD-5.4-RELEASE" or "Windows 5.1"
* CMAKE\_SYSTEM\_NAME   
  The name of the system targeted by the build. The three common values are Windows, Darwin, and Linux, though several others exist, such as Android, FreeBSD, and CrayLinuxEnvironment. Platforms without an operating system, such as embedded devices, are given Generic as a system name.
* CMAKE\_SYSTEM\_VERSION   
  Version of the operating system. Generally the kernel version.
* CMAKE\_SYSTEM\_PROCESSOR   
  the processor name (e.g. "Intel(R) Pentium(R) M processor 2.00GHz")
* CMAKE\_HOST\_SYSTEM\_NAME   
  The name of the system hosting the build. Has the same possible values as CMAKE\_SYSTEM\_NAME.

Let’s check if build system is Unix or Windows.

cmake\_minimum\_required(VERSION 3.9.1)project(CMakeHello)set(CMAKE\_CXX\_STANDARD 14)# UNIX, WIN32, WINRT, CYGWIN, APPLE are environment variables as flags set by default system  
if(UNIX)  
 message("This is a ${CMAKE\_SYSTEM\_NAME} system")  
elseif(WIN32)  
 message("This is a Windows System")  
endif()# or use MATCHES to see if actual system name   
# Darwin is Apple's system name  
if(${CMAKE\_SYSTEM\_NAME} MATCHES Darwin)  
 message("This is a ${CMAKE\_SYSTEM\_NAME} system")  
elseif(${CMAKE\_SYSTEM\_NAME} MATCHES Windows)  
 message("This is a Windows System")  
endif()add\_executable(cmake\_hello main.cpp)

System information checks your build system other than building correct binary, like using macros. You can define compiler macros to send to code during build process to change behavior.

Large code bases are implemented to be system agnostic with macros to use certain methods only for the correct system. That also prevents errors, next section shows how to define a macro and use in code.

# Defining Macros using CMake

Macros help engineers to build code conditionally to discard or include certain methods according to running system configurations.

You can define macros in CMake with add\_definitions command, using **-D** flag before the macro name.

Lets define macro named CMAKEMACROSAMPLE and print it in the code.

cmake\_minimum\_required(VERSION 3.9.1)project(CMakeHello)set(CMAKE\_CXX\_STANDARD 14)# or use MATCHES to see if actual system name   
# Darwin is Apple's system name  
if(${CMAKE\_SYSTEM\_NAME} MATCHES Darwin)  
 add\_definitions(-DCMAKEMACROSAMPLE="Apple MacOS")  
elseif(${CMAKE\_SYSTEM\_NAME} MATCHES Windows)  
 add\_definitions(-DCMAKEMACROSAMPLE="Windows PC")  
endif()add\_executable(cmake\_hello main.cpp)

Below is the new main.cpp with printing macro.

#include <iostream>#ifndef CMAKEMACROSAMPLE  
 #define CMAKEMACROSAMPLE "NO SYSTEM NAME"  
#endifauto sum(int a, int b){  
 return a + b;  
}int main() {  
 std::cout<<"Hello CMake!"<<std::endl;  
 std::cout<<CMAKEMACROSAMPLE<<std::endl;  
 std::cout<<"Sum of 3 + 4 :"<<sum(3, 4)<<std::endl;  
 return 0;  
}

# CMake Folder Organization

While building applications, we try to keep source tree clean and separate auto-generated files and binaries.

For CMake, many developers prefer to create a build folder under root tree and start CMake command inside, as below. Make sure you did cleaned all previously generate files (CMakeCache.txt), otherwise it doesn’t creates files inside build.

$ mkdir build  
$ cmake ..  
$ ls -all  
-rw-r--r-- 1 onur staff 13010 Jan 25 18:40 CMakeCache.txt  
drwxr-xr-x 15 onur staff 480 Jan 25 18:40 CMakeFiles  
-rw-r--r-- 1 onur staff 4964 Jan 25 18:40 Makefile  
-rw-r--r-- 1 onur staff 1256 Jan 25 18:40 cmake\_install.cmake  
$ make all

Above commands creates build files inside build directory. It is called out-source build.

You can add **build/** folder to your **.gitignore** to disable tracking.

At this time, you can not force CMake with variables to force creating artefacts into another folder with setting variables so if you don’t want to create a directory and cd into it, you can also use below command to create folder and generate files inside it. **-H** and **-B** flags will help it.

* -H points the source tree root.
* -B points the build directory.

$ cmake -H. -Bbuild  
# H indicates source directory  
# B indicates build directory  
# For CLion, you can navigate to CLion -> Preferences -> Build, Execution and Deployment -> CMake -> Generation Path

However, you can write script in CMake to manipulate where the libraries will be, executables will be.

Let’s edit our CMakeLists.txt to generate binary file inside bin folder with setting CMAKE\_RUNTIME\_OUTPUT\_DIRECTORY or EXECUTABLE\_OUTPUT\_PATH.

cmake\_minimum\_required(VERSION 3.9.1)project(CMakeHello)set(CMAKE\_CXX\_STANDARD 14)  
set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -Wall")set(CMAKE\_RUNTIME\_OUTPUT\_DIRECTORY ${CMAKE\_BINARY\_DIR}/bin)add\_executable(cmake\_hello main.cpp)

After building the project, cmake\_hello binary will be generated inside build/bin folder.

Same procedure can be done for library paths as well for shared libraries (.dll or .so) with following variables.

* LIBRARY\_OUTPUT\_PATH
* CMAKE\_LIBRARY\_OUTPUT\_DIRECTORY

Or you can use archive output paths for static libraries (.a or .lib)

* CMAKE\_ARCHIVE\_OUTPUT\_DIRECTORY
* ARCHIVE\_OUTPUT\_PATH

In some cases, setting single path can be enough, but for larger projects with multiple modules, you may need to write cmake scripts to manage folders to easily organize build structure. This will help you to pack and deploy, install the generated executables, libraries, documents with certain folders.

It is a common practice to disable in-source building. Following script can be used to block in-source building.

* Source of script OpenCV Project: <https://github.com/opencv/opencv/blob/master/CMakeLists.txt>

# Disable in-source builds to prevent source tree corruption.  
if(" ${CMAKE\_SOURCE\_DIR}" STREQUAL " ${CMAKE\_BINARY\_DIR}")  
 message(FATAL\_ERROR "  
FATAL: In-source builds are not allowed.  
 You should create a separate directory for build files.  
")  
endif()

# Building a Library with CMake

Let’s expand the CMakeHello project with additional sub folder and a class.

To keep things simple, we added a template class which is able to do 4 math operations, sum, subtraction, division, multiplication only on integers.

* First, we created lib/math folder inside source tree.
* Then, added class files, operations.cpp, operations.hpp as following

#ifndef CMAKEHELLO\_OPERATIONS\_HPP  
#define CMAKEHELLO\_OPERATIONS\_HPPnamespace math {  
 class operations{  
 public:  
 int sum(const int &a, const int &b);  
 int mult(const int &a, const int &b);  
 int div(const int &a, const int &b);  
 int sub(const int &a, const int &b);  
 };  
}#endif //CMAKEHELLO\_OPERATIONS\_HPP#include <exception>  
#include <stdexcept>  
#include <iostream>#include "operations.hpp"int math::operations::sum(const int &a, const int &b){  
 return a + b;  
}int math::operations::mult(const int &a, const int &b){  
 return a \* b;  
}int math::operations::div(const int &a, const int &b){  
 if(b == 0){  
 throw std::overflow\_error("Divide by zero exception");  
 }  
 return a/b;  
}int math::operations::sub(const int &a, const int &b){  
 return a - b;  
}

* Then, we modified main.cpp to include operations.hpp and use its sum method instead of previously written sum.

#include <iostream>#include "lib/math/operations.hpp"int main() {  
 std::cout<<"Hello CMake!"<<std::endl; math::operations op; int sum = op.sum(3, 4); std::cout<<"Sum of 3 + 4 :"<<sum<<std::endl; return 0;  
}

# Quick Note: Shared vs Static Library

Before going more with library building with C++, lets get a quick brief about shared and static libraries.

Shared Library File Extensions:

* Windows: **.dll**
* Mac OS X: **.dylib**
* Linux: **.so**

Static Library File Extensions:

* Windows: **.lib**
* Mac OS X: **.a**
* Linux: **.a**

Shared libraries are mainly placed in a shared resource of host to make sure multiple applications can access them. Compiler’s build systems with assumption that, shared libraries will be in a shared folder during the execution time so application binary size would reduce, it will handle some of the resources from shared libraries during executions. That requirement also decrease the performance because at each execution it tries to load instructions from shared objects.

Static libraries are used to fetch instructions directly into application binary by compiler, so all the code required from library are already injected into final application binary. That increase the size of object but increase the size of binary but performance get increased. Applications build with static library will also don’t need dependencies on the running platform.

# Building Library with Target

If you just want to build these files together with main.cpp, you can just add source files next to add\_executable command. This will compile all together and create a single binary file. Leading to 15.076 bytes of exe file.

cmake\_minimum\_required(VERSION 3.9.1)project(CMakeHello)set(CMAKE\_CXX\_STANDARD 14)  
set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -Wall")set(EXECUTABLE\_OUTPUT\_PATH ${CMAKE\_BINARY\_DIR}/bin)add\_executable(cmake\_hello main.cpp lib/math/operations.cpp lib/math/operations.hpp)

As an alternate, you can create a variable named ${SOURCES} as a list to include target sources. It can be done in a lot of different ways, depending on your methodology.

cmake\_minimum\_required(VERSION 3.9.1)project(CMakeHello)set(CMAKE\_CXX\_STANDARD 14)  
set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -Wall")set(EXECUTABLE\_OUTPUT\_PATH ${CMAKE\_BINARY\_DIR}/bin)  
set(SOURCES main.cpp   
 lib/math/operations.cpp   
 lib/math/operations.hpp)add\_executable(cmake\_hello ${SOURCES})

# Building Library Separate than Target

We can build library separately either as **shared** or **static**.

If we do that, we also need to link library to executable in order to enable executable file to make calls from operations library. We also want to generate library binaries inside **lib** directory of build folder.

* set LIBRARY\_OUTPUT\_PATH
* add\_library command as SHARED or STATIC
* target\_link\_libraries to target (cmake\_hello)

cmake\_minimum\_required(VERSION 3.9.1)project(CMakeHello)set(CMAKE\_CXX\_STANDARD 14)  
set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -Wall")set(EXECUTABLE\_OUTPUT\_PATH ${CMAKE\_BINARY\_DIR}/bin)  
set(LIBRARY\_OUTPUT\_PATH ${CMAKE\_BINARY\_DIR}/lib)message(${CMAKE\_BINARY\_DIR})add\_library(math SHARED lib/math/operations.cpp)  
#add\_library(math STATIC lib/math/operations.cpp)add\_executable(cmake\_hello main.cpp)target\_link\_libraries(cmake\_hello math)

You will see that, in mac OS, libmath.dylib is generated and binary size decreased to 14.876 bytes. (No significant change because code is already very small).

# Build Library as Sub-Module CMake

Another library building process is that, you can write a new CMakeLists.txt file inside lib/operations folder, to build it independently just before building exe file.

This kind of situations are mostly needed when there is an optional build required to generate module. In our case, above solution is better since they all dependend. However, if you want to add a case to only build libraries and skipping executable build process, below example can work as well.

Generate a new CMakeLists.txt inside lib/math folder as shown in below code snippet to build library.

cmake\_minimum\_required(VERSION 3.9.1)set(LIBRARY\_OUTPUT\_PATH ${CMAKE\_BINARY\_DIR}/lib)add\_library(math SHARED operations.cpp)

* You should delete add\_library command and setting LIBRARY\_OUTPUT\_PATH from main CMakeLists.txt.
* Now, you should add new build path with add\_subdirectoy. This command makes cmake to go for the folder and build it as well with the CMakeLists.txt inside it.

cmake\_minimum\_required(VERSION 3.9.1)project(CMakeHello)set(CMAKE\_CXX\_STANDARD 14)  
set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -Wall")set(EXECUTABLE\_OUTPUT\_PATH ${CMAKE\_BINARY\_DIR}/bin)message(${CMAKE\_BINARY\_DIR})add\_subdirectory(lib/math)add\_executable(cmake\_hello main.cpp)target\_link\_libraries(cmake\_hello math)

* Now, you will see the libmath.diylib inside build/lib folder again.

Above, examples shows how to deal with additional sources inside source tree. Either build them all together or separately according to your release plans.

# Finding Existing Library with CMake

In most cases, you may require to use installed libraries on your build host. For example, you can install boost library with apt-get or brew package managers to your system.

CMake has conditional statements to allow you to block building process if library is not installed.

If a library installed to system with its .cmake configurations, cmake would be able to look for system default library locations to find that library. like /usr/lib;/usr/local/lib

Boost library can be installed via package managers, **brew** or **apt-get** to system. We can use, cmake’s find\_package command to check if library exists before building executable.

Let’s change our example little more. I want to use Boost normal distribution library to generate random samples. Therefore, I included boost/random.hpp, and initialized boost::random::normal\_distribution to generate numbers with variate\_generator.

#include <iostream>#include "lib/math/operations.hpp"#include <boost/random.hpp>int main() {  
 std::cout<<"Hello CMake!"<<std::endl; math::operations op; int sum = op.sum(3, 4); std::cout<<"Sum of 3 + 4 :"<<sum<<std::endl; //Boost Random Sample  
 boost::mt19937 rng;  
 double mean = 2.3;  
 double std = 0.34;  
 auto normal\_dist = boost::random::normal\_distribution<double>(mean, std); boost::variate\_generator<boost::mt19937&,  
 boost::normal\_distribution<> > random\_generator(rng, normal\_dist); for(int i = 0; i < 2; i++){  
 auto rand\_val = random\_generator();  
 std::cout<<"Random Val "<<i+1<<" :"<<rand\_val<<std::endl;  
 } return 0;  
}

How CMakeLists.txt should be written now? It should check for the Boost library, if it can find includes headers and links libraries.

**Note**: In most cases, default library and include paths are defined in cmake default configurations, so it may find these libraries without any modification, but it is not suggested to leave CMakeLists.txt with trusting to system.

cmake\_minimum\_required(VERSION 3.9.1)project(CMakeHello)set(CMAKE\_CXX\_STANDARD 14)  
set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -Wall")#set(CMAKE\_RUNTIME\_OUTPUT\_DIRECTORY ${CMAKE\_BINARY\_DIR}/bin)  
set(EXECUTABLE\_OUTPUT\_PATH ${CMAKE\_BINARY\_DIR}/bin)message(${CMAKE\_BINARY\_DIR})add\_executable(cmake\_hello main.cpp)add\_subdirectory(lib/math)find\_package(Boost 1.66)# Check for libray, if found print message, include dirs and link libraries.  
if(Boost\_FOUND)  
 message("Boost Found")  
 include\_directories(${Boost\_INCLUDE\_DIRS})  
 target\_link\_libraries(cmake\_hello ${Boost\_LIBRARIES})  
elseif(NOT Boost\_FOUND)  
 error("Boost Not Found")  
endif()target\_link\_libraries(cmake\_hello math)

include\_directories will include library headers, target\_link\_libraries will link boost library. Above code will not continue to build if package not found since we defined error, but you can also add REQUIRED flag after package to raise error.

**Note**: Boost name already recognized by system because cmake has already defined commands to check for Boost library. If it is not a common library like boost, you should write your own scripts to enable this feature.

When a package found, following variables will be initialized automatically.

<NAME>\_FOUND : Flag to show if it is found  
<NAME>\_INCLUDE\_DIRS or <NAME>\_INCLUDES : Header directories  
<NAME>\_LIBRARIES or <NAME>\_LIBRARIES or <NAME>\_LIBS : library files  
<NAME>\_DEFINITIONS

So what happens, if a library is in a custom folder and outside of source tree. If there won’t be any CMake, your command line build would look like below.

g++ main.cpp -o cmake\_hello -I/home/onur/libraries/boost/include -L/home/onur/libraries/boost -lBoost

With this logic, you should add include and library folder with following commands.

include\_directories(/Users/User/Projects/libraries/include)  
link\_directories(/Users/User/Projects/libraries/libs)# elseif case can be   
elseif(NOT Boost\_FOUND)  
message("Boost Not Found")  
 include\_directories(/Users/User/Projects/libraries/include)  
 link\_directories(/Users/User/Projects/libraries/libs)  
 target\_link\_libraries(cmake\_hello Boost)  
endif()

Many different methodologies can be followed while including custom libraries for your project. You can also write custom cmake methods to search for given folders to check libraries etc. Most important thing is to understand linkage logic of compiler. You should point the headers and library (.so, .a) folder and link library for compile process.

Above case is not very safe, because you can’t be sure about if it exists in the folders of host. Safest method is to fetch sources of library and build before going forward. In order to provide this property, dependent library should be added with its sources. If library is closed source, you should include binary to your source tree.

However, it is not always easy to distribute sources because of licensing issues, legal problems etc. It is responsibility of build engineer to figure out best practice for this situation.

# Target System Configurations

It is highly possible to build your application, library for or on multiple systems. For example, you may want to include Intel related libraries if you want to deploy your binary on a intel system or you may want to cross-compile for an embedded system, like Android. In order to organise build system, you should add all possible checks into your cmake as well as your code as macros.

# Change Compiler and Linker for Build

Let’s say you will cross-compile your project for a different target system. In such cases, your host system should include the target system compiler and linker installed. A basic example is shown in official documentation of CMake: <https://cmake.org/cmake/help/v3.6/manual/cmake-toolchains.7.html#cross-compiling-for-linux>

Official example is enough for this article’s context, in order to cover it basically. It is an example of building for Raspberry Pi with its C/C++ compiler and tools (linker etc.)

set(CMAKE\_SYSTEM\_NAME Linux)  
set(CMAKE\_SYSTEM\_PROCESSOR arm)set(CMAKE\_SYSROOT /home/devel/rasp-pi-rootfs)  
set(CMAKE\_STAGING\_PREFIX /home/devel/stage)set(tools /home/devel/gcc-4.7-linaro-rpi-gnueabihf)  
set(CMAKE\_C\_COMPILER ${tools}/bin/arm-linux-gnueabihf-gcc)  
set(CMAKE\_CXX\_COMPILER ${tools}/bin/arm-linux-gnueabihf-g++)set(CMAKE\_FIND\_ROOT\_PATH\_MODE\_PROGRAM NEVER)  
set(CMAKE\_FIND\_ROOT\_PATH\_MODE\_LIBRARY ONLY)  
set(CMAKE\_FIND\_ROOT\_PATH\_MODE\_INCLUDE ONLY)  
set(CMAKE\_FIND\_ROOT\_PATH\_MODE\_PACKAGE ONLY)

Most important part is to point compiler and tools (linker) paths correctly. Rest is configuration of build process, including optimisations.

It is also important to know your target system’s compiler properties, it would always be different than the host system. For example, intel system has different level of instructions than ARM. Even arm cpu’s would differ for instruction sets so for optimisations. Make sure you had covered those logic to apply for your cross-compile.

In the next section, we tried to cover some of the basic compiler and linker flags for GNU GCC and Clang.

# Compiler/Linker Flags with CMake

Compiler and Linker flags enables engineers to customise behavior of compiler during build process for warnings, optimisations of building process. CMake doesn’t give any new feature additional to flags used by Clang or GNU compilers. But, lets have an overview of compiler and linker flags and what they have been used for.

See, user manuals:

* <https://gcc.gnu.org/onlinedocs/gcc-2.95.2/gcc_2.html>
* <https://clang.llvm.org/docs/ClangCommandLineReference.html>

# Compiler Flags

# Setting Compiler Flags

Compiler flags are crucial when it comes to configure your application’s final properties, optimisation values, behaviour of compiler etc. In Makefile or command line it can be defined in different ways but it is rather easier to define them in CMake. Just like below:

set(CMAKE\_CXX\_FLAGS "-std=c++0x -Wall")  
# suggested way is to keep previous flags in mind and append new ones  
set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -std=c++0x -Wall")# Alternatively, you can use generator expressions, which are conditional expressions. Below says that, if compiper is c++ then set it to c++11  
add\_compile\_options("$<$<STREQUAL:$<TARGET\_PROPERTY:LINKER\_LANGUAGE>,CXX>:-std=c++11>")

It is quite important to know, how these flags works and what flags do you want to use than setting flags with CMake.

* Optimisation flags set certain compiler flags or disables them. These flags are defined to make easier to switch on/off certain optimisation flags.
* Please see manual:
* <https://clang.llvm.org/docs/ClangCommandLineReference.html#optimization-level>
* <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>

-O0, -O1, -O2, -O3, -Os, -Oz, -Ofast

* Warning flags set warning properties during build process to see any warning during build to report them. Some may be turned off to fasten compile process because each warning process would require to make analysis. See manual and tree.
* <https://gist.github.com/d0k/3608547>
* <https://clang.llvm.org/docs/DiagnosticsReference.html>

-Wstring-conversion, -Wall, -Wswitch-enum …

# Set Source File Properties

This is a complex property of CMake if there are multiple targets, it can be needed to change one target’s certain behavior. In case, you would like to build main.cpp with C++11 and if you building only library, you may want to build it with C++14. In such cases, you may want to configure certain source’s properties with using set\_source\_files\_properties command like below:

set\_source\_files\_properties(${CMAKE\_CURRENT\_SOURCE\_DIR}/\*.cpp PROPERTIES COMPILE\_FLAGS "-std=c++11")

A large set of properties can be seen from following manual.

* <https://cmake.org/cmake/help/v3.3/manual/cmake-properties.7.html#source-file-properties>

Each can be used for a specific case needed for your purpose.

# Linker Flags

Here is a list of linker flags of GNU GCC Linker.

* <https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html>

GCC linker, by default can search the directories defined in environment variables /usr/lib;/usr/local/lib , then links standard libraries by default.

Most common flag is -l to link desired library like, -lzlib, -lboost etc.

Additional flags, help you to change behavior of linking options of executable.

Below are the variables which you can add linker flags.

* CMAKE\_EXE\_LINKER\_FLAGS: Flags used to by linker during creation of executable
* CMAKE\_EXE\_LINKER\_FLAGS\_RELEASE: Flags used to by linker during creation of release executable
* CMAKE\_EXE\_LINKER\_FLAGS\_DEBUG: Flags used to by linker during creation of debug executable
* CMAKE\_STATIC\_LINKER\_FLAGS: Flags used by linker during the creation of static libraries (.a, .lib)
* CMAKE\_SHARED\_LINKER\_FLAGS: Flags used by linker during the creation of static libraries (.a, .lib)
* CMAKE\_MODULE\_LINKER\_FLAGS: Flags used by linker during the creation of static libraries (.a, .lib)

set(CMAKE\_EXE\_LINKER\_FLAGS "${CMAKE\_EXE\_LINKER\_FLAGS} -fsanitize=address")  
set(CMAKE\_EXE\_LINKER\_FLAGS "${CMAKE\_EXE\_LINKER\_FLAGS} -Wl")

# Debug and Release Configuration

It is highly recommended to create CMakeLists.txt with multiple configurations according to your needs. If you intend to deliver binaries, you should make a Release, which doesn't include debug flags in binary, which would be a way faster and ready to run. However, debug version of executable files include many of other flags which exposes the memory, method names etc for debuggers to help identify the errors. It is not a good practice and not safe to deliver debug version of an application.

CMake helps you to write script to separate final builds of both type of outputs. There are additional build types like Release with Debug Flags (RELWITHDEBINFO) or Minium Release Size (MINSIZEREL). In this example, we will show both.

* You have to create Debug and Release folders for both type of builds under your build folder. build/Debug and build/Release. cmake command line will change as below:

$ cmake -H. -Bbuild/Debug  
$ cmake -H. -Bbuild/Release

* Above configuration is not enough to create different binaries. You should also set build type with CMAKE\_BUILD\_TYPE variable on the command line. (CLion handles this process by itself.)

$ cmake -DCMAKE\_BUILD\_TYPE=Debug -H. -Bbuild/Debug  
$ cmake -DCMAKE\_BUILD\_TYPE=Release -H. -Bbuild/Release

* CMAKE\_BUILD\_TYPE is accessible inside CMakeLists.txt. You can easily check for build type in CMakeLists.txt

if(${CMAKE\_BUILD\_TYPE} MATCHES Debug)  
 message("Debug Build")  
elseif(${CMAKE\_BUILD\_TYPE} MATCHES Release)  
 message("Release Build")  
endif()

* You can also, set compiler and linker flags separately for build types using the config variables shown in the previous section.
* CMAKE\_EXE\_LINKER\_FLAGS\_RELEASE: Flags used to by linker during creation of release executable
* CMAKE\_EXE\_LINKER\_FLAGS\_DEBUG: Flags used to by linker during creation of debug executable
* CMAKE\_CXX\_FLAGS\_RELEASE
* CMAKE\_CXX\_FLAGS\_DEBUG

# CMake Installation/Deployment Configurations

CMake helps create commands for installation process as well, not only the build process.

There is a good coverage of the parameters of install command of CMake in:

* <https://cmake.org/cmake/help/v3.0/command/install.html>

Main procedure in this process is to copy the files generated by build process to a destination folder on the host. Therefore:

* First, think about destination folder. CMAKE\_INSTALL\_PREFIX is the variable to define host destination. it is set to /usr/local by default. During the build process, you should point the destination folder in command-line.

$ cmake -DCMAKE\_BUILD\_TYPE=Release -DCMAKE\_INSTALL\_PREFIX=/usr/local/test/with/cmake -H. -Bbuild/Release

* Keeping in mind that, you are getting prefix destination from terminal you should think about library and executable destinations accordingly.

# CMake Best Practices

* Always remember the previous configurations, make sure you append new flag instead of overwriting it. It is better to implement, add\_flag/remove method of yours, to achieve easier implementation.

set(VARIABLE "${VARIABLE} Flag1 Flag2")

* Always check system information carefully, raise error if a certain configuration can’t be done to prevent faulty binaries.
* Always, check required libraries to continue build process, raise error if not found.

if(Boost\_FOUND)  
 message("Boost Found")  
else()  
 error("Boost Not Found")  
endif()

# Resources

* <https://cmake.org/cmake-tutorial/>
* <https://tuannguyen68.gitbooks.io/learning-cmake-a-beginner-s-guide/content/chap1/chap1.html>
* <https://cmake.org/cmake/help/latest>
* <https://cmake.org/cmake/help/v3.10/>
* <https://www.vtk.org/Wiki/CMake/Examples>
* <https://cmake.org/cmake/help/v3.0/module/FindBoost.html>
* <https://gcc.gnu.org/onlinedocs/gcc-2.95.2/gcc_2.html>
* <https://clang.llvm.org/docs/ClangCommandLineReference.html>
* <http://www.bu.edu/tech/support/research/software-and-programming/programming/compilers/gcc-compiler-flags/>

**This document : TCPClientServerCodeChallenge.docx**