

My Project

Generated by Doxygen 1.10.0

| | |
|--|-----------|
| 1 Functionality | 1 |
| 1.1 Limitations | 1 |
| 2 IPK - first project - Client-chat | 3 |
| 2.1 Theory | 3 |
| 2.2 Implementation | 3 |
| 3 Hierarchical Index | 5 |
| 3.1 Class Hierarchy | 5 |
| 4 Class Index | 7 |
| 4.1 Class List | 7 |
| 5 File Index | 9 |
| 5.1 File List | 9 |
| 6 Class Documentation | 11 |
| 6.1 ArgumentsHandler Class Reference | 11 |
| 6.1.1 Detailed Description | 12 |
| 6.1.2 Member Function Documentation | 12 |
| 6.1.2.1 get_address() | 12 |
| 6.1.2.2 get_args() | 12 |
| 6.1.2.3 get_port() | 13 |
| 6.1.2.4 get_retransmissions() | 13 |
| 6.1.2.5 get_threads() | 13 |
| 6.1.2.6 get_timeout() | 13 |
| 6.1.2.7 print_help() | 13 |
| 6.1.3 Member Data Documentation | 14 |
| 6.1.3.1 address | 14 |
| 6.1.3.2 number_of_threads | 14 |
| 6.1.3.3 port | 14 |
| 6.1.3.4 retransmissions | 14 |
| 6.1.3.5 timeout | 14 |
| 6.2 AuthPackets Struct Reference | 14 |
| 6.2.1 Detailed Description | 16 |
| 6.2.2 Constructor & Destructor Documentation | 16 |
| 6.2.2.1 AuthPackets() | 16 |
| 6.2.3 Member Function Documentation | 16 |
| 6.2.3.1 construct_message() | 16 |
| 6.2.4 Member Data Documentation | 17 |
| 6.2.4.1 DisplayName | 17 |
| 6.2.4.2 Secret | 17 |
| 6.2.4.3 Username | 17 |
| 6.3 ConfirmPackets Struct Reference | 17 |

| | |
|--|----|
| 6.3.1 Detailed Description | 18 |
| 6.3.2 Constructor & Destructor Documentation | 18 |
| 6.3.2.1 ConfirmPackets() | 18 |
| 6.3.3 Member Function Documentation | 19 |
| 6.3.3.1 construct_message() | 19 |
| 6.3.4 Member Data Documentation | 19 |
| 6.3.4.1 Ref_MessageID | 19 |
| 6.4 JoinPackets Struct Reference | 19 |
| 6.4.1 Detailed Description | 21 |
| 6.4.2 Constructor & Destructor Documentation | 21 |
| 6.4.2.1 JoinPackets() | 21 |
| 6.4.3 Member Function Documentation | 21 |
| 6.4.3.1 construct_message() | 21 |
| 6.4.4 Member Data Documentation | 21 |
| 6.4.4.1 ChannelID | 21 |
| 6.4.4.2 DisplayName | 22 |
| 6.5 MsgPackets Struct Reference | 22 |
| 6.5.1 Detailed Description | 23 |
| 6.5.2 Constructor & Destructor Documentation | 23 |
| 6.5.2.1 MsgPackets() | 23 |
| 6.5.3 Member Function Documentation | 24 |
| 6.5.3.1 construct_message() | 24 |
| 6.5.4 Member Data Documentation | 24 |
| 6.5.4.1 DisplayName | 24 |
| 6.5.4.2 MessageContents | 24 |
| 6.6 Packets Struct Reference | 24 |
| 6.6.1 Detailed Description | 25 |
| 6.6.2 Constructor & Destructor Documentation | 25 |
| 6.6.2.1 Packets() | 25 |
| 6.6.3 Member Function Documentation | 26 |
| 6.6.3.1 construct_message() | 26 |
| 6.6.4 Member Data Documentation | 26 |
| 6.6.4.1 MessageID | 26 |
| 6.6.4.2 MessageType | 26 |
| 6.7 ReplyPackets Struct Reference | 26 |
| 6.7.1 Detailed Description | 28 |
| 6.7.2 Constructor & Destructor Documentation | 28 |
| 6.7.2.1 ReplyPackets() | 28 |
| 6.7.3 Member Function Documentation | 28 |
| 6.7.3.1 construct_message() | 28 |
| 6.7.4 Member Data Documentation | 29 |
| 6.7.4.1 Message | 29 |

| | |
|---|----|
| 6.7.4.2 ref_id | 29 |
| 6.7.4.3 result | 29 |
| 6.8 Server Class Reference | 29 |
| 6.8.1 Detailed Description | 30 |
| 6.8.2 Member Function Documentation | 30 |
| 6.8.2.1 Initialize() | 30 |
| 6.8.2.2 Listen() | 30 |
| 6.9 synch Struct Reference | 30 |
| 6.9.1 Detailed Description | 31 |
| 6.9.2 Constructor & Destructor Documentation | 31 |
| 6.9.2.1 synch() | 31 |
| 6.9.3 Member Data Documentation | 31 |
| 6.9.3.1 cv | 31 |
| 6.9.3.2 cv2 | 32 |
| 6.9.3.3 finished | 32 |
| 6.9.3.4 mtx | 32 |
| 6.9.3.5 ready | 32 |
| 6.9.3.6 un | 32 |
| 6.9.3.7 usernames | 32 |
| 6.9.3.8 waiting | 32 |
| 6.10 TCPhandler Class Reference | 32 |
| 6.10.1 Detailed Description | 34 |
| 6.10.2 Constructor & Destructor Documentation | 34 |
| 6.10.2.1 TCPhandler() | 34 |
| 6.10.3 Member Function Documentation | 34 |
| 6.10.3.1 convert_from_udp() | 34 |
| 6.10.3.2 create_bye() | 35 |
| 6.10.3.3 create_message() | 36 |
| 6.10.3.4 create_reply() | 36 |
| 6.10.3.5 decipher_the_message() | 37 |
| 6.10.3.6 handleTCP() | 39 |
| 6.10.3.7 listening_for_incoming_connection() | 40 |
| 6.10.3.8 message() | 41 |
| 6.10.3.9 send_buf() | 41 |
| 6.10.3.10 send_string() | 42 |
| 6.10.3.11 user_changed_channel() | 42 |
| 6.10.3.12 username_already_exists() | 43 |
| 6.10.4 Member Data Documentation | 43 |
| 6.10.4.1 auth | 43 |
| 6.10.4.2 channel_name | 43 |
| 6.10.4.3 client_addr | 43 |
| 6.10.4.4 client_socket | 44 |

| | |
|---|----|
| 6.10.4.5 display_name | 44 |
| 6.10.4.6 epoll_fd | 44 |
| 6.10.4.7 events | 44 |
| 6.10.4.8 user_n | 44 |
| 6.11 TCPserver Class Reference | 44 |
| 6.11.1 Detailed Description | 45 |
| 6.11.2 Constructor & Destructor Documentation | 45 |
| 6.11.2.1 TCPserver() | 45 |
| 6.11.3 Member Function Documentation | 45 |
| 6.11.3.1 Destroy() | 45 |
| 6.11.3.2 Initialize() | 46 |
| 6.11.3.3 Listen() | 46 |
| 6.11.4 Member Data Documentation | 47 |
| 6.11.4.1 sockFD | 47 |
| 6.12 ThreadPool Class Reference | 47 |
| 6.12.1 Detailed Description | 49 |
| 6.12.2 Constructor & Destructor Documentation | 49 |
| 6.12.2.1 ThreadPool() [1/3] | 49 |
| 6.12.2.2 ~ThreadPool() | 49 |
| 6.12.2.3 ThreadPool() [2/3] | 49 |
| 6.12.2.4 ThreadPool() [3/3] | 49 |
| 6.12.3 Member Function Documentation | 49 |
| 6.12.3.1 AddTask() | 49 |
| 6.12.3.2 operator=() [1/2] | 50 |
| 6.12.3.3 operator=() [2/2] | 50 |
| 6.12.3.4 QueueSize() | 50 |
| 6.12.3.5 Shutdown() | 50 |
| 6.12.4 Member Data Documentation | 51 |
| 6.12.4.1 busy_threads | 51 |
| 6.12.4.2 condition_variable | 51 |
| 6.12.4.3 mutex | 51 |
| 6.12.4.4 queue | 51 |
| 6.12.4.5 shutdown_requested | 51 |
| 6.12.4.6 threads | 51 |
| 6.13 ThreadPool::ThreadWorker Class Reference | 52 |
| 6.13.1 Detailed Description | 52 |
| 6.13.2 Constructor & Destructor Documentation | 53 |
| 6.13.2.1 ThreadWorker() | 53 |
| 6.13.3 Member Function Documentation | 53 |
| 6.13.3.1 operator>() | 53 |
| 6.13.4 Member Data Documentation | 53 |
| 6.13.4.1 thread_pool | 53 |

| | |
|---|----|
| 6.14 UDPhandler Class Reference | 53 |
| 6.14.1 Detailed Description | 55 |
| 6.14.2 Constructor & Destructor Documentation | 55 |
| 6.14.2.1 UDPhandler() | 55 |
| 6.14.3 Member Function Documentation | 56 |
| 6.14.3.1 buffer_validation() | 56 |
| 6.14.3.2 change_display_name() | 57 |
| 6.14.3.3 client_leaving() | 57 |
| 6.14.3.4 convert_from_tcp() | 58 |
| 6.14.3.5 create_bye() | 59 |
| 6.14.3.6 create_message() | 59 |
| 6.14.3.7 decipher_the_message() | 60 |
| 6.14.3.8 handleUDP() | 62 |
| 6.14.3.9 message() | 63 |
| 6.14.3.10 read_channel_name() | 64 |
| 6.14.3.11 read_packet_id() | 64 |
| 6.14.3.12 respond_to_auth() | 65 |
| 6.14.3.13 respond_to_join() | 66 |
| 6.14.3.14 send_confirm() | 67 |
| 6.14.3.15 send_message() | 68 |
| 6.14.3.16 send_reply() | 69 |
| 6.14.3.17 username_exists() | 70 |
| 6.14.3.18 wait_for_the_incoming_connection() | 70 |
| 6.14.3.19 waiting_for_confirm() | 71 |
| 6.14.4 Member Data Documentation | 72 |
| 6.14.4.1 auth | 72 |
| 6.14.4.2 channel_name | 72 |
| 6.14.4.3 client_addr | 72 |
| 6.14.4.4 client_socket | 72 |
| 6.14.4.5 display_name | 72 |
| 6.14.4.6 epoll_fd | 72 |
| 6.14.4.7 events | 72 |
| 6.14.4.8 global_counter | 72 |
| 6.14.4.9 retransmissions | 72 |
| 6.14.4.10 timeout_chat | 72 |
| 6.14.4.11 user_n | 72 |
| 6.14.4.12 vec | 73 |
| 6.15 UDPserver Class Reference | 73 |
| 6.15.1 Detailed Description | 74 |
| 6.15.2 Constructor & Destructor Documentation | 74 |
| 6.15.2.1 UDPserver() | 74 |
| 6.15.3 Member Function Documentation | 75 |

| | |
|---|-----------|
| 6.15.3.1 Destroy() | 75 |
| 6.15.3.2 Initialize() | 75 |
| 6.15.3.3 Listen() | 75 |
| 6.15.4 Member Data Documentation | 76 |
| 6.15.4.1 retransmissions | 76 |
| 6.15.4.2 sockFD | 77 |
| 6.15.4.3 timeout | 77 |
| 6.16 UserInfo Struct Reference | 77 |
| 6.16.1 Detailed Description | 77 |
| 6.16.2 Constructor & Destructor Documentation | 78 |
| 6.16.2.1 UserInfo() | 78 |
| 6.16.3 Member Data Documentation | 78 |
| 6.16.3.1 buf | 78 |
| 6.16.3.2 channel | 78 |
| 6.16.3.3 client | 78 |
| 6.16.3.4 length | 78 |
| 6.16.3.5 tcp | 78 |
| 6.16.3.6 tcp_socket | 78 |
| 7 File Documentation | 79 |
| 7.1 ArgumentsHandler.cpp File Reference | 79 |
| 7.2 ArgumentsHandler.cpp | 79 |
| 7.3 ArgumentsHandler.h File Reference | 81 |
| 7.4 ArgumentsHandler.h | 81 |
| 7.5 CHAGELOG.md File Reference | 82 |
| 7.6 main.cpp File Reference | 82 |
| 7.6.1 Function Documentation | 82 |
| 7.6.1.1 handle_sigint() | 82 |
| 7.6.1.2 init() | 83 |
| 7.6.1.3 main() | 83 |
| 7.6.2 Variable Documentation | 84 |
| 7.6.2.1 pipefd | 84 |
| 7.7 main.cpp | 84 |
| 7.8 packets.h File Reference | 85 |
| 7.8.1 Typedef Documentation | 86 |
| 7.8.1.1 AuthPacket | 86 |
| 7.8.1.2 ConfirmPacket | 86 |
| 7.8.1.3 JoinPacket | 86 |
| 7.8.1.4 MsgPacket | 87 |
| 7.8.1.5 Packet | 87 |
| 7.8.1.6 ReplyPacket | 87 |
| 7.9 packets.h | 87 |

| | |
|--|------------|
| 7.10 README.md File Reference | 89 |
| 7.11 server_classes.cpp File Reference | 89 |
| 7.12 server_classes.cpp | 89 |
| 7.13 server_classes.h File Reference | 91 |
| 7.13.1 Macro Definition Documentation | 92 |
| 7.13.1.1 PORT | 92 |
| 7.14 server_classes.h | 92 |
| 7.15 synch.h File Reference | 93 |
| 7.16 synch.h | 94 |
| 7.17 TCPhandler.cpp File Reference | 95 |
| 7.17.1 Function Documentation | 95 |
| 7.17.1.1 read_queue() | 95 |
| 7.17.1.2 tcp_logger() | 96 |
| 7.18 TCPhandler.cpp | 97 |
| 7.19 TCPhandler.h File Reference | 101 |
| 7.19.1 Function Documentation | 101 |
| 7.19.1.1 read_queue() | 101 |
| 7.19.1.2 tcp_logger() | 102 |
| 7.20 TCPhandler.h | 103 |
| 7.21 thread_pool.h File Reference | 104 |
| 7.22 thread_pool.h | 105 |
| 7.23 UDPhandler.cpp File Reference | 107 |
| 7.23.1 Function Documentation | 107 |
| 7.23.1.1 logger() | 107 |
| 7.23.1.2 read_queue() | 107 |
| 7.24 UDPhandler.cpp | 108 |
| 7.25 UDPhandler.h File Reference | 114 |
| 7.25.1 Function Documentation | 115 |
| 7.25.1.1 logger() | 115 |
| 7.25.1.2 read_queue() | 116 |
| 7.26 UDPhandler.h | 117 |
| Index | 119 |

Chapter 1

Functionality

The program provides basic server functionality, working with TCP and UDP protocols simultaneously.

1.1 Limitations

There are currently no limitations known to me.

Chapter 2

IPK - first project - Client-chat

This is a simple chat application written in C++.

Table of contents:

- IPK - first project - Client-chat
 - Theory
 - Implementation
 - * UDP communication
 - Message confirmation tracking
 - * TCP communication
 - Testing
 - * UDP tests
 - * TCP tests

2.1 Theory

- ### TCP

Transmission Control Protocol is a transport protocol that is used on top of IP (Internet Protocol) to ensure reliable transmission of packets over the internet or other networks. TCP is a connection-oriented protocol, which means that it establishes and maintains a connection between the two parties until the data transfer is complete. TCP provides mechanisms to solve problems that arise from packet-based messaging, e.g. lost packets or out-of-order packets, duplicate packets, and corrupted packets. TCP achieves this by using sequence and acknowledgement numbers, checksums, flow control, error control, and congestion control.

- ### UDP

User Datagram Protocol is a connectionless and unreliable protocol that provides a simple and efficient way to send and receive datagrams over an IP network. UDP does not guarantee delivery, order, or integrity of the data, but it minimizes the overhead and latency involved in transmitting data when compared to TCP. UDP is suitable for applications that require speed, simplicity, or real-time communication, such as streaming media, online gaming, voice over IP, or DNS queries.

- ### Thread Pool

Thread pool is a software design pattern for achieving concurrency of execution in a computer program. Essentially, a thread pool is a group of pre-instantiated, idle threads which stand ready to be given work. These are preferred over instantiating new threads for each task when there is a large number of short tasks to be done rather than a small number of long ones. This prevents having to incur the overhead of creating a thread a large number of times.

2.2 Implementation

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

| | |
|------------------------------------|----|
| ArgumentsHandler | 11 |
| Packets | 24 |
| AuthPackets | 14 |
| ConfirmPackets | 17 |
| JoinPackets | 19 |
| MsgPackets | 22 |
| ReplyPackets | 26 |
| Server | 29 |
| TCPserver | 44 |
| UDPserver | 73 |
| synch | 30 |
| TCPHandler | 32 |
| ThreadPool | 47 |
| ThreadPool::ThreadWorker | 52 |
| UDPHandler | 53 |
| UserInfo | 77 |

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

| | |
|--|----|
| ArgumentsHandler | 11 |
| AuthPackets | 14 |
| ConfirmPackets | 17 |
| JoinPackets | 19 |
| MsgPackets | 22 |
| Packets | 24 |
| ReplyPackets | 26 |
| Server | 29 |
| synch | 30 |
| TCPHandler | 32 |
| TCPserver | 44 |
| ThreadPool | 47 |
| ThreadPool::ThreadWorker | 52 |
| UDPHandler | 53 |
| UDPserver | 73 |
| UserInfo | 77 |

Chapter 5

File Index

5.1 File List

Here is a list of all files with brief descriptions:

| | |
|--------------------------------------|-----|
| ArgumentsHandler.cpp | 79 |
| ArgumentsHandler.h | 81 |
| main.cpp | 82 |
| packets.h | 85 |
| server_classes.cpp | 89 |
| server_classes.h | 91 |
| synch.h | 93 |
| TCPhandler.cpp | 95 |
| TCPhandler.h | 101 |
| thread_pool.h | 104 |
| UDPhandler.cpp | 107 |
| UDPhandler.h | 114 |

Chapter 6

Class Documentation

6.1 ArgumentsHandler Class Reference

```
#include <ArgumentsHandler.h>
```

Collaboration diagram for ArgumentsHandler:

| ArgumentsHandler |
|--|
| <ul style="list-style-type: none">- timeout- port- address- retransmissions- number_of_threads |
| <ul style="list-style-type: none">+ get_args()+ get_timeout()+ get_retransmissions()+ get_port()+ get_address()+ get_threads()- print_help() |

Public Member Functions

- void [get_args](#) (int argc, char *argv[])
- int [get_timeout](#) ()
- int [get_retransmissions](#) ()
- int [get_port](#) () const
- char * [get_address](#) ()
- int [get_threads](#) ()

Static Private Member Functions

- static void [print_help](#) ()

Private Attributes

- int [timeout](#)
- int [port](#)
- char * [address](#)
- int [retransmissions](#)
- int [number_of_threads](#)

6.1.1 Detailed Description

Definition at line 11 of file [ArgumentsHandler.h](#).

6.1.2 Member Function Documentation

6.1.2.1 [get_address\(\)](#)

char * [ArgumentsHandler::get_address](#) ()

Definition at line 104 of file [ArgumentsHandler.cpp](#).

```
00104 {
00105     return this->retransmissions;
00106 }
```

6.1.2.2 [get_args\(\)](#)

void [ArgumentsHandler::get_args](#) (
 int argc,
 char * argv[])

Definition at line 12 of file [ArgumentsHandler.cpp](#).

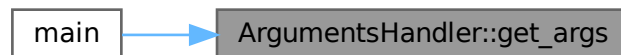
```
00020 {
00021     this->timeout = 500;
00022     this->retransmissions = 3;
00023     this->port = 47356;
00024     this->address = new char[13];
00025     this->number_of_threads = 20;
00026     strcpy(address, "127.0.0.1");
00027
00028     for (int i = 0; i < argc; i++) {
00029         std::string arg = argv[i];
00030
00031         if (arg == "-h") {
00032             print_help();
00033             exit(0);
00034         } else if (arg == "-l") {
00035             i++;
00036             if (i < argc) {
00037                 address = argv[i];
00038             } else {
00039                 std::cout << "Nothing passed to address" << std::endl;
00040                 exit(1);
00041             }
00042         } else if (arg == "-p") {
00043             i++;
00044             if (i < argc) {
00045                 try {
00046                     port = std::stoi(argv[i]);
00047                 } catch (std::invalid_argument &) {
00048                     std::cout << "Passed non-int value to port" << std::endl;
00049                     exit(1);
00050                 }
00051             } else {
00052                 std::cout << "Nothing passed to port" << std::endl;
00053                 exit(1);
00054             }
00055         } else if (arg == "-d") {
00056             i++;
00057             if (i < argc) {
00058                 try {
00059                     this->timeout = std::stoi(argv[i]);
00060                 } catch (std::invalid_argument &) {
00061                     std::cout << "Passed non-int value to timeout" << std::endl;
00062                     exit(1);
00063                 }
00064             } else {
00065                 std::cout << "Nothing passed to timeout" << std::endl;
00066                 exit(1);
00067             }
00068         }
00069     }
00070 }
```

```

00068         } else if (arg == "-r") {
00069             i++;
00070             if (i < argc) {
00071                 try {
00072                     this->retransmissions = std::stoi(argv[i]);
00073                 } catch (std::invalid_argument &) {
00074                     std::cout << "Passed non-int value to retransmissions" << std::endl;
00075                     exit(1);
00076                 }
00077             } else {
00078                 std::cout << "Nothing passed to retransmissions" << std::endl;
00079                 exit(1);
00080             }
00081         } else if (arg == "-n") {
00082             i++;
00083             if (i < argc) {
00084                 try {
00085                     this->number_of_threads = std::stoi(argv[i]);
00086                 } catch (std::invalid_argument &) {
00087                     std::cout << "Passed non-int value to number of threads" << std::endl;
00088                     exit(1);
00089                 }
00090             } else {

```

Here is the caller graph for this function:



6.1.2.3 get_port()

int ArgumentsHandler::get_port () const
Definition at line 92 of file [ArgumentsHandler.cpp](#).

6.1.2.4 get_retransmissions()

int ArgumentsHandler::get_retransmissions ()
Definition at line 96 of file [ArgumentsHandler.cpp](#).

6.1.2.5 get_threads()

int ArgumentsHandler::get_threads ()
Definition at line 108 of file [ArgumentsHandler.cpp](#).

```

00108         {
00109             return this->timeout;
00110         }

```

6.1.2.6 get_timeout()

int ArgumentsHandler::get_timeout ()
Definition at line 100 of file [ArgumentsHandler.cpp](#).

```

00100         {
00101             return this->port;
00102         }

```

6.1.2.7 print_help()

void ArgumentsHandler::print_help () [static], [private]
Definition at line 8 of file [ArgumentsHandler.cpp](#).

```

00008         {
00009             std::cout << R"(|Argument | Default values | Type
00010

```

6.1.3 Member Data Documentation

6.1.3.1 address

```
char* ArgumentsHandler::address [private]
```

Definition at line 28 of file [ArgumentsHandler.h](#).

6.1.3.2 number_of_threads

```
int ArgumentsHandler::number_of_threads [private]
```

Definition at line 30 of file [ArgumentsHandler.h](#).

6.1.3.3 port

```
int ArgumentsHandler::port [private]
```

Definition at line 27 of file [ArgumentsHandler.h](#).

6.1.3.4 retransmissions

```
int ArgumentsHandler::retransmissions [private]
```

Definition at line 29 of file [ArgumentsHandler.h](#).

6.1.3.5 timeout

```
int ArgumentsHandler::timeout [private]
```

Definition at line 26 of file [ArgumentsHandler.h](#).

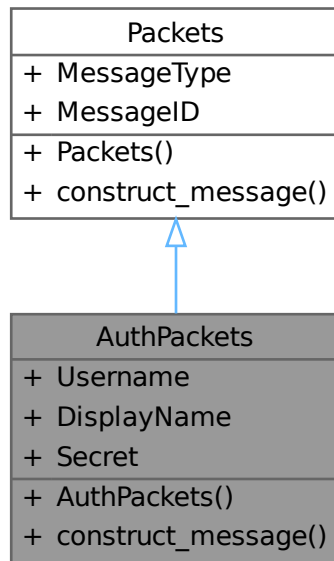
The documentation for this class was generated from the following files:

- [ArgumentsHandler.h](#)
- [ArgumentsHandler.cpp](#)

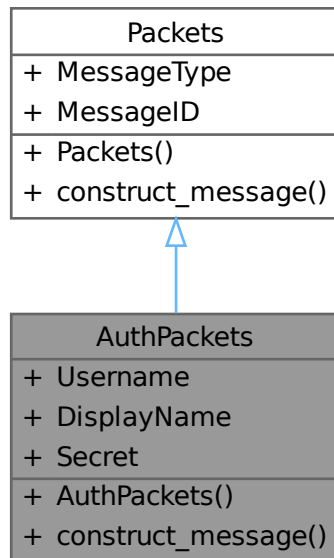
6.2 AuthPackets Struct Reference

```
#include <packets.h>
```


Inheritance diagram for AuthPackets:



Collaboration diagram for AuthPackets:



Public Member Functions

- [AuthPackets](#) (uint8_t type, uint16_t id, std::string u_n, std::string disp_name, std::string sec)

- int [construct_message](#) (uint8_t *b) override

Public Member Functions inherited from [Packets](#)

- [Packets](#) (uint8_t type, uint16_t id)

Public Attributes

- std::string [Username](#)
- std::string [DisplayName](#)
- std::string [Secret](#)

Public Attributes inherited from [Packets](#)

- uint8_t [MessageType](#)
- uint16_t [MessageID](#)

6.2.1 Detailed Description

Definition at line 122 of file [packets.h](#).

6.2.2 Constructor & Destructor Documentation

6.2.2.1 AuthPackets()

```
AuthPackets::AuthPackets (
    uint8_t type,
    uint16_t id,
    std::string u_n,
    std::string disp_name,
    std::string sec ) [inline]
```

Definition at line 128 of file [packets.h](#).

```
00128                                     :
00129     Packets(type,
00130             id) {
00131         Username = std::move(u_n);
00132         DisplayName = std::move(disp_name);
00133         Secret = std::move(sec);
00134     }
```

6.2.3 Member Function Documentation

6.2.3.1 construct_message()

```
int AuthPackets::construct_message (
    uint8_t * b ) [inline], [override], [virtual]
```

Reimplemented from [Packets](#).

Definition at line 135 of file [packets.h](#).

```
00135     {
00136         memcpy(b, &this->MessageType, sizeof(this->MessageType));
00137         b += sizeof(this->MessageType);
00138         //std::cout<<this->MessageType<<std::endl;
00139
00140         //uint16_t ID = htons(this->MessageID);
00141         uint16_t ID = this->MessageID;
00142         memcpy(b, &ID, sizeof(ID));
00143         b += sizeof(ID);
00144
00145         memcpy(b, Username.c_str(), Username.length());
00146         b[Username.length()] = '\0';
00147         b += Username.length() + 1;
00148
00149         memcpy(b, DisplayName.c_str(), DisplayName.length());
00150         b[DisplayName.length()] = '\0';
00151         b += DisplayName.length() + 1;
00152
00153         memcpy(b, Secret.c_str(), Secret.length());
```

```

00154         b[Secret.length()] = '\0';
00155         b += Secret.length() + 1;
00156         return sizeof(this->MessageType) + sizeof(ID) + Username.length() + 1 + DisplayName.length() +
1 +
00157             Secret.length() + 1;
00158     }

```

6.2.4 Member Data Documentation

6.2.4.1 DisplayName

std::string AuthPackets::DisplayName
Definition at line 125 of file [packets.h](#).

6.2.4.2 Secret

std::string AuthPackets::Secret
Definition at line 126 of file [packets.h](#).

6.2.4.3 Username

std::string AuthPackets::Username
Definition at line 124 of file [packets.h](#).

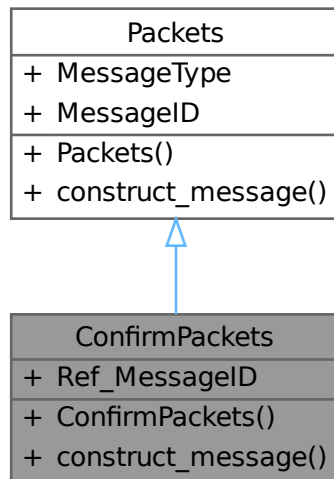
The documentation for this struct was generated from the following file:

- [packets.h](#)

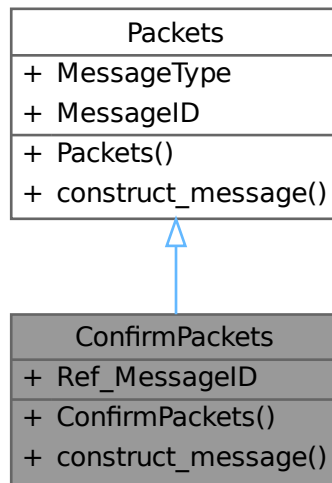
6.3 ConfirmPackets Struct Reference

```
#include <packets.h>
```

Inheritance diagram for ConfirmPackets:



Collaboration diagram for ConfirmPackets:



Public Member Functions

- [ConfirmPackets](#) (uint8_t type, uint16_t id, uint16_t ref_id)
- int [construct_message](#) (uint8_t *b) override

Public Member Functions inherited from [Packets](#)

- [Packets](#) (uint8_t type, uint16_t id)

Public Attributes

- uint16_t [Ref_MessageID](#)

Public Attributes inherited from [Packets](#)

- uint8_t [MessageType](#)
- uint16_t [MessageID](#)

6.3.1 Detailed Description

Definition at line 39 of file [packets.h](#).

6.3.2 Constructor & Destructor Documentation

6.3.2.1 ConfirmPackets()

```

ConfirmPackets::ConfirmPackets (
    uint8_t type,
    uint16_t id,
    uint16_t ref_id ) [inline]
  
```

Definition at line 43 of file [packets.h](#).

```

00043                                     : Packets(type, id) {
00044     Ref\_MessageID = ref_id;
00045 }
  
```

6.3.3 Member Function Documentation

6.3.3.1 construct_message()

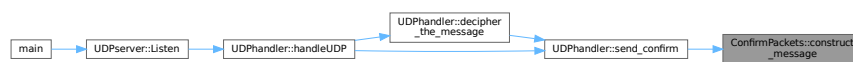
```
int ConfirmPackets::construct_message (
    uint8_t * b ) [inline], [override], [virtual]
```

Reimplemented from [Packets](#).

Definition at line 47 of file [packets.h](#).

```
00047     {
00048         memcpy(b, &this->MessageType, sizeof(this->MessageType));
00049         b += sizeof(this->MessageType);
00050
00051         uint16_t ID = this->Ref_MessageID;
00052         memcpy(b, &ID, sizeof(ID));
00053         b += sizeof(ID);
00054         return sizeof(this->MessageType) + sizeof(ID);
00055     }
```

Here is the caller graph for this function:



6.3.4 Member Data Documentation

6.3.4.1 Ref_MessageID

```
uint16_t ConfirmPackets::Ref_MessageID
```

Definition at line 41 of file [packets.h](#).

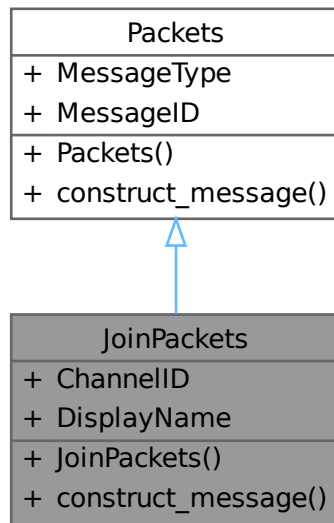
The documentation for this struct was generated from the following file:

- [packets.h](#)

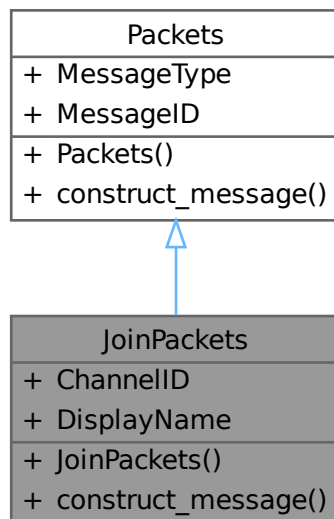
6.4 JoinPackets Struct Reference

```
#include <packets.h>
```

Inheritance diagram for JoinPackets:



Collaboration diagram for JoinPackets:



Public Member Functions

- [JoinPackets](#) (uint8_t type, uint16_t id, std::string ch_id, std::string disp_name)
- int [construct_message](#) (uint8_t *b) override

Public Member Functions inherited from [Packets](#)

- [Packets](#) (uint8_t type, uint16_t id)

Public Attributes

- std::string [ChannelID](#)
- std::string [DisplayName](#)

Public Attributes inherited from [Packets](#)

- uint8_t [MessageType](#)
- uint16_t [MessageID](#)

6.4.1 Detailed Description

Definition at line 59 of file [packets.h](#).

6.4.2 Constructor & Destructor Documentation

6.4.2.1 JoinPackets()

```
JoinPackets::JoinPackets (
    uint8_t type,
    uint16_t id,
    std::string ch_id,
    std::string disp_name ) [inline]
```

Definition at line 64 of file [packets.h](#).

```
00064                                     : Packets(type, id) {
00065     ChannelID = std::move(ch_id);
00066     DisplayName = std::move(disp_name);
00067 }
```

6.4.3 Member Function Documentation

6.4.3.1 construct_message()

```
int JoinPackets::construct_message (
    uint8_t * b ) [inline], [override], [virtual]
```

Reimplemented from [Packets](#).

Definition at line 69 of file [packets.h](#).

```
00069     {
00070         memcpy(b, &this->MessageType, sizeof(this->MessageType));
00071         b += sizeof(this->MessageType);
00072
00073         //uint16_t ID = htons(this->MessageID);
00074         uint16_t ID = this->MessageID;
00075         memcpy(b, &ID, sizeof(ID));
00076         b += sizeof(ID);
00077
00078         memcpy(b, ChannelID.c_str(), ChannelID.length());
00079         b[ChannelID.length()] = '\0';
00080         b += ChannelID.length() + 1;
00081
00082         memcpy(b, DisplayName.c_str(), DisplayName.length());
00083         b[DisplayName.length()] = '\0';
00084         b += DisplayName.length() + 1;
00085         return sizeof(this->MessageType) + sizeof(ID) + ChannelID.length() + 1 + DisplayName.length()
00086             + 1;
00087     }
```

6.4.4 Member Data Documentation

6.4.4.1 ChannelID

std::string JoinPackets::ChannelID

Definition at line 61 of file [packets.h](#).

6.4.4.2 DisplayName

```
std::string JoinPackets::DisplayName
```

Definition at line 62 of file [packets.h](#).

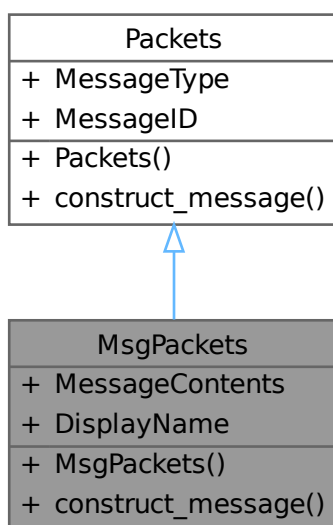
The documentation for this struct was generated from the following file:

- [packets.h](#)

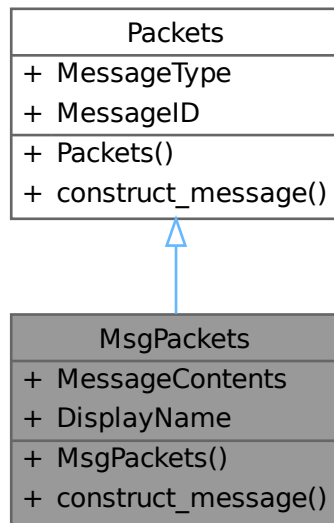
6.5 MsgPackets Struct Reference

```
#include <packets.h>
```

Inheritance diagram for MsgPackets:



Collaboration diagram for MsgPackets:



Public Member Functions

- [MsgPackets](#) (uint8_t type, uint16_t id, std::string content, std::string disp_name)
- int [construct_message](#) (uint8_t *b) override

Public Member Functions inherited from [Packets](#)

- [Packets](#) (uint8_t type, uint16_t id)

Public Attributes

- std::string [MessageContents](#)
- std::string [DisplayName](#)

Public Attributes inherited from [Packets](#)

- uint8_t [MessageType](#)
- uint16_t [MessageID](#)

6.5.1 Detailed Description

Definition at line 91 of file [packets.h](#).

6.5.2 Constructor & Destructor Documentation

6.5.2.1 MsgPackets()

```

MsgPackets::MsgPackets (
    uint8_t type,
    uint16_t id,
    std::string content,
    std::string disp_name ) [inline]
  
```

Definition at line 96 of file [packets.h](#).

```
00096                                     : Packets(type, id)
00097 {
00097     MessageContents = std::move(content);
00098     DisplayName = std::move(displ_name);
00099 }
```

6.5.3 Member Function Documentation

6.5.3.1 construct_message()

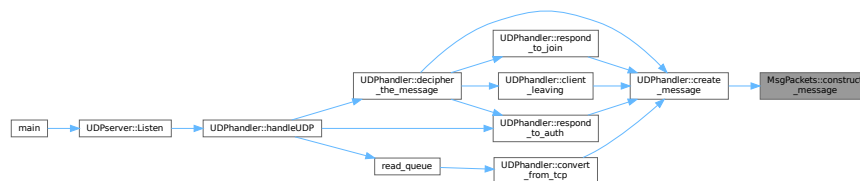
```
int MsgPackets::construct_message (
    uint8_t * b ) [inline], [override], [virtual]
```

Reimplemented from [Packets](#).

Definition at line 101 of file [packets.h](#).

```
00101 {
00102     memcpy(b, &this->MessageType, sizeof(this->MessageType));
00103     b += sizeof(this->MessageType);
00104
00105     //uint16_t ID = htons(this->MessageID);
00106     uint16_t ID = this->MessageID;
00107     memcpy(b, &ID, sizeof(ID));
00108     b += sizeof(ID);
00109
00110     memcpy(b, DisplayName.c_str(), DisplayName.length());
00111     b[DisplayName.length()] = '\0';
00112     b += DisplayName.length() + 1;
00113
00114     memcpy(b, MessageContents.c_str(), MessageContents.length());
00115     b[MessageContents.length()] = '\0';
00116     b += MessageContents.length() + 1;
00117     return sizeof(this->MessageType) + sizeof(ID) + DisplayName.length() + 1 +
00118         MessageContents.length() + 1;
00118 }
```

Here is the caller graph for this function:



6.5.4 Member Data Documentation

6.5.4.1 DisplayName

std::string MsgPackets::DisplayName

Definition at line 94 of file [packets.h](#).

6.5.4.2 MessageContents

std::string MsgPackets::MessageContents

Definition at line 93 of file [packets.h](#).

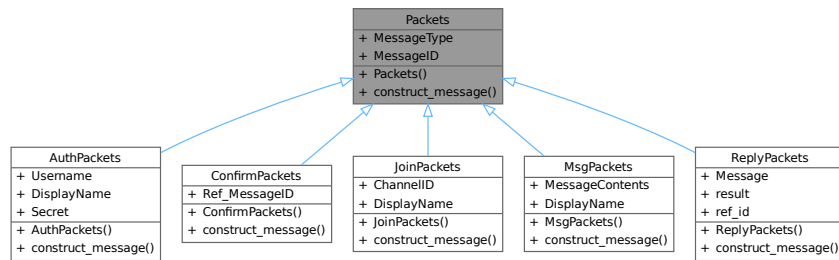
The documentation for this struct was generated from the following file:

- [packets.h](#)

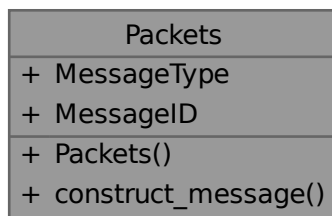
6.6 Packets Struct Reference

```
#include <packets.h>
```

Inheritance diagram for Packets:



Collaboration diagram for Packets:



Public Member Functions

- [Packets](#) (uint8_t type, uint16_t id)
- virtual int [construct_message](#) (uint8_t *b)

Public Attributes

- uint8_t [MessageType](#)
- uint16_t [MessageID](#)

6.6.1 Detailed Description

Definition at line 16 of file [packets.h](#).

6.6.2 Constructor & Destructor Documentation

6.6.2.1 Packets()

```

Packets::Packets (
    uint8_t type,
    uint16_t id ) [inline]
  
```

Definition at line 20 of file [packets.h](#).

```

00020 {
00021     MessageType = type;
00022     MessageID = id;
00023 }
  
```

6.6.3 Member Function Documentation

6.6.3.1 construct_message()

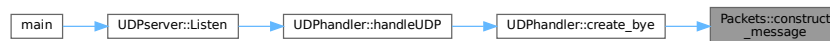
```
virtual int Packets::construct_message (
    uint8_t * b ) [inline], [virtual]
```

Reimplemented in [ConfirmPackets](#), [JoinPackets](#), [MsgPackets](#), [AuthPackets](#), and [ReplyPackets](#).

Definition at line 25 of file [packets.h](#).

```
00025     {
00026         memcpy(b, &this->MessageType, sizeof(this->MessageType));
00027         b += sizeof(this->MessageType);
00028
00029
00030         //uint16_t ID = htons(this->MessageID);
00031         uint16_t ID = this->MessageID;
00032         memcpy(b, &ID, sizeof(ID));
00033         b += sizeof(ID);
00034         return 3;
00035     }
```

Here is the caller graph for this function:



6.6.4 Member Data Documentation

6.6.4.1 MessageID

```
uint16_t Packets::MessageID
```

Definition at line 18 of file [packets.h](#).

6.6.4.2 MessageType

```
uint8_t Packets::MessageType
```

Definition at line 17 of file [packets.h](#).

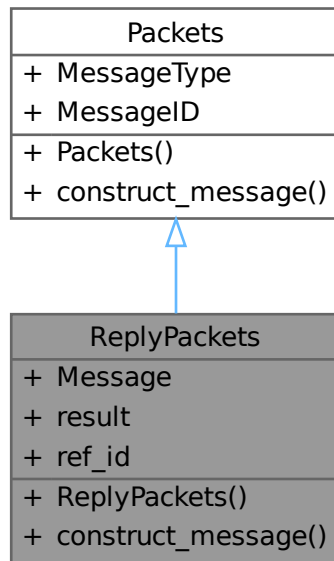
The documentation for this struct was generated from the following file:

- [packets.h](#)

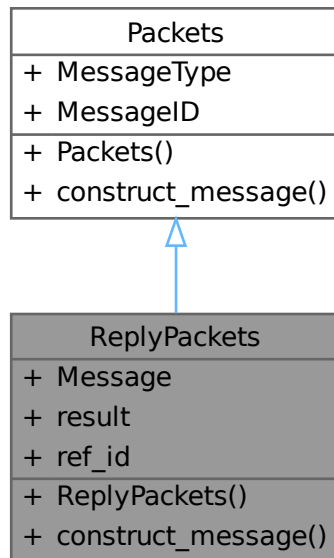
6.7 ReplyPackets Struct Reference

```
#include <packets.h>
```

Inheritance diagram for ReplyPackets:



Collaboration diagram for ReplyPackets:



Public Member Functions

- [ReplyPackets](#) (uint8_t type, uint16_t id, std::string mes, uint8_t res, uint16_t ref)

- `int construct_message (uint8_t *b)` override

Public Member Functions inherited from `Packets`

- `Packets (uint8_t type, uint16_t id)`

Public Attributes

- `std::string Message`
- `uint8_t result`
- `uint16_t ref_id`

Public Attributes inherited from `Packets`

- `uint8_t MessageType`
- `uint16_t MessageID`

6.7.1 Detailed Description

Definition at line 162 of file `packets.h`.

6.7.2 Constructor & Destructor Documentation

6.7.2.1 ReplyPackets()

```
ReplyPackets::ReplyPackets (
    uint8_t type,
    uint16_t id,
    std::string mes,
    uint8_t res,
    uint16_t ref ) [inline]
```

Definition at line 167 of file `packets.h`.

```
00167 id) {
00168     Message = std::move(mes);
00169     result = res;
00170     ref_id = ref;
00171 }
```

: `Packets (type,`

6.7.3 Member Function Documentation

6.7.3.1 construct_message()

```
int ReplyPackets::construct_message (
    uint8_t * b ) [inline], [override], [virtual]
```

Reimplemented from `Packets`.

Definition at line 173 of file `packets.h`.

```
00173 {
00174     memcpy(b, &this->MessageType, sizeof(this->MessageType));
00175     b += sizeof(this->MessageType);
00176     uint16_t ID = this->MessageID;
00177     memcpy(b, &ID, sizeof(ID));
00178     b += sizeof(ID);
00179
00180     memcpy(b, &result, sizeof(result));
00181     b += sizeof(result);
00182
00183     memcpy(b, &ref_id, sizeof(ref_id));
00184     b += sizeof(ref_id);
00185
00186     memcpy(b, Message.c_str(), Message.length());
00187     b[Message.length()] = '\0';
00188     b += Message.length() + 1;
00189
00190     return sizeof(this->MessageType) + sizeof(ID) + sizeof(result) + sizeof(ref_id) +
00191     Message.length() + 1;
00191 }
```

Here is the caller graph for this function:



6.7.4 Member Data Documentation

6.7.4.1 Message

`std::string ReplyPackets::Message`
 Definition at line 163 of file [packets.h](#).

6.7.4.2 ref_id

`uint16_t ReplyPackets::ref_id`
 Definition at line 165 of file [packets.h](#).

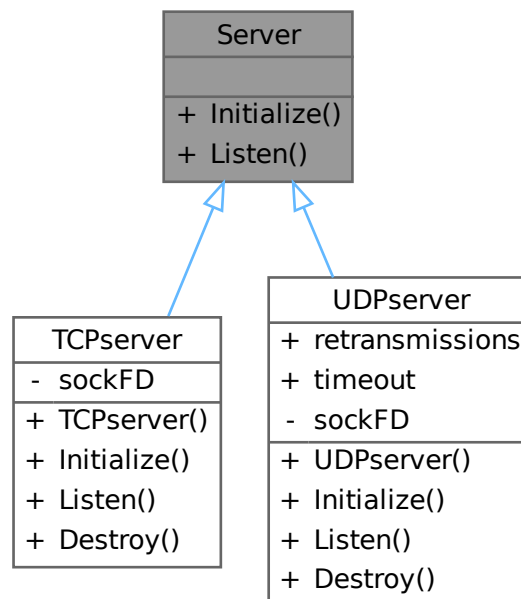
6.7.4.3 result

`uint8_t ReplyPackets::result`
 Definition at line 164 of file [packets.h](#).
 The documentation for this struct was generated from the following file:

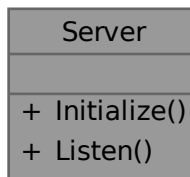
- [packets.h](#)

6.8 Server Class Reference

`#include <server_classes.h>`
 Inheritance diagram for Server:



Collaboration diagram for Server:



Public Member Functions

- virtual void [Initialize](#) (struct sockaddr_in *server_address)=0
- virtual void [Listen](#) (ThreadPool *tp, std::stack< [UserInfo](#) > *s, [synch](#) *synch_variables, int signal_listener)=0

6.8.1 Detailed Description

Definition at line 24 of file [server_classes.h](#).

6.8.2 Member Function Documentation

6.8.2.1 Initialize()

```
virtual void Server::Initialize (
    struct sockaddr_in * server_address ) [pure virtual]
```

Implemented in [TCPserver](#), and [UDPserver](#).

6.8.2.2 Listen()

```
virtual void Server::Listen (
    ThreadPool * tp,
    std::stack< UserInfo > * s,
    synch * synch_variables,
    int signal_listener ) [pure virtual]
```

Implemented in [TCPserver](#), and [UDPserver](#).

The documentation for this class was generated from the following file:

- [server_classes.h](#)

6.9 synch Struct Reference

```
#include <synch.h>
```


Collaboration diagram for synch:

| synch |
|-------------|
| + mtx |
| + waiting |
| + un |
| + ready |
| + cv |
| + cv2 |
| + finished |
| + usernames |
| + synch() |

Public Member Functions

- [synch](#) (int b)

Public Attributes

- std::mutex [mtx](#)
- std::mutex [waiting](#)
- std::mutex [un](#)
- bool [ready](#)
- std::condition_variable [cv](#)
- std::condition_variable [cv2](#)
- int [finished](#)
- std::unordered_set< std::string > [usernames](#)

6.9.1 Detailed Description

Definition at line 26 of file [synch.h](#).

6.9.2 Constructor & Destructor Documentation

6.9.2.1 synch()

```
synch::synch (  
    int b ) [inline], [explicit]
```

Definition at line 36 of file [synch.h](#).

```
00036 : finished(b), ready(false){};
```

6.9.3 Member Data Documentation

6.9.3.1 cv

```
std::condition_variable synch::cv
```

Definition at line 31 of file [synch.h](#).

6.9.3.2 cv2

`std::condition_variable synch::cv2`
Definition at line 32 of file [synch.h](#).

6.9.3.3 finished

`int synch::finished`
Definition at line 33 of file [synch.h](#).

6.9.3.4 mtx

`std::mutex synch::mtx`
Definition at line 27 of file [synch.h](#).

6.9.3.5 ready

`bool synch::ready`
Definition at line 30 of file [synch.h](#).

6.9.3.6 un

`std::mutex synch::un`
Definition at line 29 of file [synch.h](#).

6.9.3.7 usernames

`std::unordered_set<std::string> synch::usernames`
Definition at line 34 of file [synch.h](#).

6.9.3.8 waiting

`std::mutex synch::waiting`
Definition at line 28 of file [synch.h](#).

The documentation for this struct was generated from the following file:

- [synch.h](#)

6.10 TCPhandler Class Reference

```
#include <TCPhandler.h>
```

Collaboration diagram for TCPhandler:

| TCPhandler |
|---|
| <ul style="list-style-type: none"> + channel_name + display_name + client_socket + epoll_fd + events + client_addr + auth + user_n |
| <ul style="list-style-type: none"> + TCPhandler() + send_buf() + create_message() + convert_from_udp() + handleTCP() - listening_for_incoming_connection() - decipher_the_message() - send_string() - create_reply() - create_bye() - message() - user_changed_channel() - username_already_exists() |

Public Member Functions

- [TCPhandler](#) (int s, sockaddr_in c, int kill)
- void [send_buf](#) (uint8_t *buf, int length) const
- void [create_message](#) (bool error, const char *msg)
- int [convert_from_udp](#) (uint8_t *buf, uint8_t *tcp_buf)

Static Public Member Functions

- static void [handleTCP](#) (int [client_socket](#), int *busy, std::stack< [UserInfo](#) > *s, [synch](#) *synch_var, sockaddr_in client, int signal_listener)

Public Attributes

- std::string [channel_name](#)
- std::string [display_name](#)
- int [client_socket](#)
- int [epoll_fd](#)
- epoll_event [events](#) [2]

- sockaddr_in [client_addr](#)
- bool [auth](#)
- std::string [user_n](#)

Private Member Functions

- int [listening_for_incoming_connection](#) (uint8_t *buf, int len)
- bool [decipher_the_message](#) (uint8_t *buf, int length, std::stack< [UserInfo](#) > *s, [synch](#) *synch_var)
- void [send_string](#) (std::string &msg) const
- void [create_reply](#) (const char *status, const char *msg)
- void [create_bye](#) ()
- void [message](#) (uint8_t *buf, int message_length, std::stack< [UserInfo](#) > *s, [synch](#) *synch_var, std::string &channel)
- void [user_changed_channel](#) (std::stack< [UserInfo](#) > *s, [synch](#) *synch_var, const char *action)
- bool [username_already_exists](#) (std::string &username, [synch](#) *synch_vars)

6.10.1 Detailed Description

Definition at line 10 of file [TCPHandler.h](#).

6.10.2 Constructor & Destructor Documentation

6.10.2.1 TCPHandler()

```
TCPHandler::TCPHandler (
    int s,
    sockaddr_in c,
    int kill ) [inline]
```

Definition at line 22 of file [TCPHandler.h](#).

```
00022         {
00023             this->channel_name = "general";
00024             this->client_socket = s;
00025
00026             epoll_fd = epoll_create1(0);
00027             if (epoll_fd == -1) {
00028                 std::cerr << "Failed to create epoll file descriptor\n";
00029                 exit(EXIT_FAILURE);
00030             }
00031
00032             // setup epoll event
00033             struct epoll_event ev;
00034             ev.events = EPOLLIN;
00035             ev.data.fd = this->client_socket;
00036
00037             // add socket file descriptor to epoll
00038             if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, this->client_socket, &ev) == -1) {
00039                 std::cerr << "Failed to add file descriptor to epoll\n";
00040                 close(epoll_fd);
00041                 exit(EXIT_FAILURE);
00042             }
00043
00044             ev.data.fd = kill;
00045             if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, kill, &ev) < 0) {
00046                 std::cerr << "Unable to add socket to epoll\n";
00047                 exit(EXIT_FAILURE);
00048             }
00049
00050             client_addr = c;
00051             auth = false;
00052         }
00053     }
```

6.10.3 Member Function Documentation

6.10.3.1 convert_from_udp()

```
int TCPHandler::convert_from_udp (
    uint8_t * buf,
    uint8_t * tcp_buf )
```

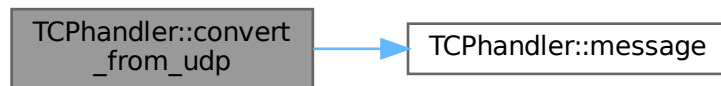
Definition at line 285 of file [TCPhandler.cpp](#).

```

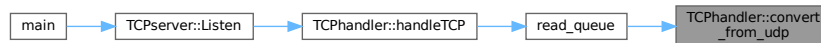
00285                                     {
00286     int i = 3;
00287     std::string display_n;
00288     std::string contents;
00289
00290     while (udp_buf[i] != 0x00) {
00291         display_n.push_back(static_cast<char>(udp_buf[i]));
00292         i++;
00293     }
00294
00295     i++;
00296
00297     while (udp_buf[i] != 0x00) {
00298         contents.push_back(static_cast<char>(udp_buf[i]));
00299         i++;
00300     }
00301
00302     std::string message = "MSG FROM " + display_n + " IS " + contents + "\r\n";
00303
00304     memcpy(buf, message.c_str(), message.length());
00305
00306     return message.length();
00307 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



6.10.3.2 create_bye()

```
void TCPhandler::create_bye ( ) [private]
```

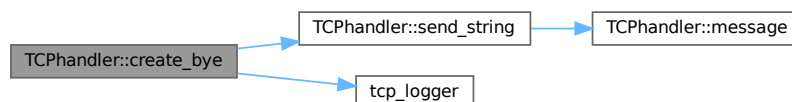
Definition at line 245 of file [TCPhandler.cpp](#).

```

00245     {
00246         std::string bye = "BYE\r\n";
00247         tcp_logger(this->client_addr, "BYE", "SENT");
00248         this->send_string(bye);
00249     }

```

Here is the call graph for this function:



Here is the caller graph for this function:



6.10.3.3 create_message()

```

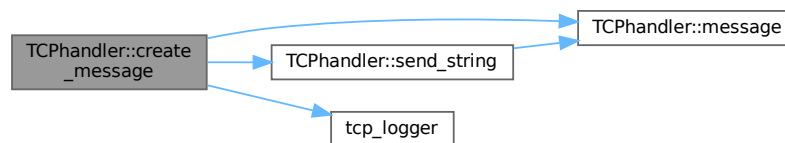
void TCPHandler::create_message (
    bool error,
    const char * msg )
  
```

Definition at line 237 of file [TCPHandler.cpp](#).

```

00237                                     {
00238     std::string message;
00239     error ? message = "ERR FROM SERVER IS " + std::string(msg) + "\r\n" : message = "MSG FROM SERVER
IS " +
00240                                     std::string(msg) +
"\r\n";
00241     tcp_logger(this->client_addr, "MSG", "SENT");
00242     this->send_string(message);
00243 }
  
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.10.3.4 create_reply()

```

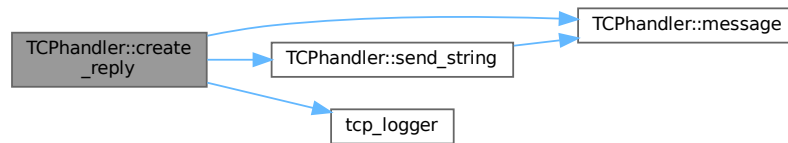
void TCPHandler::create_reply (
    const char * status,
    const char * msg ) [private]
  
```

Definition at line 230 of file [TCPHandler.cpp](#).

```

00230                                     {
00231     std::string message;
00232     message = "REPLY " + std::string(status) + " IS " + std::string(msg) + "\r\n";
00233     tcp_logger(this->client_addr, "REPLY", "SENT");
00234     this->send_string(message);
00235 }
  
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.10.3.5 decipher_the_message()

```

bool TCPhandler::decipher_the_message (
    uint8_t * buf,
    int length,
    std::stack< UserInfo > * s,
    synch * synch_var ) [private]
  
```

Definition at line 115 of file [TCPhandler.cpp](#).

```

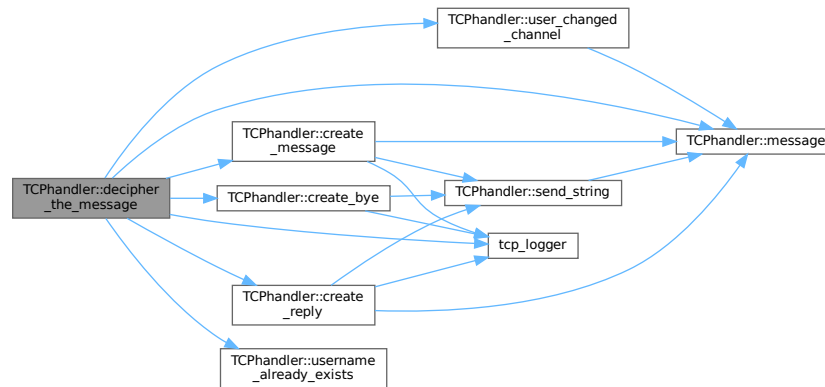
00115 {
00116     std::string out_str;
00117     for (int i = 0; i < length - 2; ++i) {
00118         out_str += static_cast<char>(buf[i]);
00119     }
00120
00121     std::istringstream iss(out_str);
00122     std::vector<std::string> result;
00123     for (std::string element; std::getline(iss, element, ' ');) {
00124         result.push_back(element);
00125     }
00126
00127     if (!this->auth) {
00128         if (result[0] != "AUTH") {
00129             if (result[0] != "BYE") {
00130                 if (result[0] != "ERR") {
00131                     this->create_message(true, "You should log-in before doing anything else");
00132                     return true;
00133                 }
00134             }
00135         }
00136     }
00137
00138     if (result[0] == "AUTH") {
00139         std::regex e("^AUTH.{1,20} AS.{1,20} USING.{1,128}$");
00140         if (!std::regex_match(out_str, e)) {
00141             std::string mes = "Wrong AUTH format";
00142             std::cout << mes << std::endl;
00143             create_message(true, "Wrong AUTH format");
00144             std::this_thread::sleep_for(std::chrono::milliseconds(10));
00145             this->create_bye();
00146             return false;
00147         }
00148         synch_var->un.lock();
00149         bool exists = username_already_exists(result[1], synch_var);
00150         synch_var->un.unlock();
00151         tcp_logger(this->client_addr, "AUTH", "RECV");
00152         if (exists) {
00153             this->create_reply("NOK", "Username already exists");
00154         } else {
00155             this->create_reply("OK", "Authentication is successful");
00156             this->display_name = result[3];
00157         }
00158     }
  
```

```

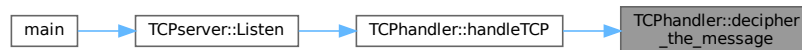
00157         this->user_changed_channel(s, synch_var, "joined");
00158         this->auth = true;
00159     }
00160
00161     } else if (result[0] == "MSG") {
00162         std::regex e("^MSG FROM .{1,20} IS .{1,1400}$");
00163         if (!std::regex_match(out_str, e)) {
00164             std::string mes = "Wrong MSG format";
00165             std::cout << mes << std::endl;
00166             create_message(true, "Wrong MSG format");
00167             std::this_thread::sleep_for(std::chrono::milliseconds(10));
00168             this->create_bye();
00169             return false;
00170         }
00171         tcp_logger(this->client_addr, "MSG", "RCV");
00172         this->display_name = result[2];
00173         this->message(buf, length, s, synch_var, this->channel_name);
00174     } else if (result[0] == "JOIN") {
00175         std::regex e("^JOIN .{1,20} AS .{1,20}$");
00176         if (!std::regex_match(out_str, e)) {
00177             std::string mes = "Wrong JOIN format";
00178             create_message(true, mes.c_str());
00179             create_message(true, "Wrong JOIN format");
00180             std::this_thread::sleep_for(std::chrono::milliseconds(10));
00181             this->create_bye();
00182             return false;
00183         }
00184         if (result[1] != this->channel_name) {
00185             tcp_logger(this->client_addr, "JOIN", "RCV");
00186             this->user_changed_channel(s, synch_var, "left");
00187             std::this_thread::sleep_for(std::chrono::milliseconds(30));
00188             this->channel_name = result[1];
00189             this->display_name = result[3];
00190             this->user_changed_channel(s, synch_var, "joined");
00191             this->create_reply("OK", "Join was successful");
00192         } else {
00193             this->create_reply("NOK", "Tried to join to the current channel");
00194         }
00195     } else if (result[0] == "BYE") {
00196         tcp_logger(this->client_addr, "BYE", "RCV");
00197         if (this->auth) {
00198             user_changed_channel(s, synch_var, "left");
00199         }
00200         return false;
00201     } else if (result[0] == "ERR") {
00202         tcp_logger(this->client_addr, "ERR", "RCV");
00203         if (this->auth) {
00204             user_changed_channel(s, synch_var, "left");
00205         }
00206         this->create_bye();
00207         return false;
00208     } else {
00209         tcp_logger(this->client_addr, "UNDEFINED", "RCV");
00210         this->create_message(true, "Unknown command");
00211         std::this_thread::sleep_for(std::chrono::milliseconds(10));
00212         this->create_bye();
00213         return false;
00214     }
00215
00216     return true;
00217 }

```


Here is the call graph for this function:



Here is the caller graph for this function:



6.10.3.6 handleTCP()

```

void TCPhandler::handleTCP (
    int client_socket,
    int * busy,
    std::stack< UserInfo > * s,
    synch * synch_var,
    sockaddr_in client,
    int signal_listener ) [static]

```

Definition at line 7 of file `TCPhandler.cpp`.

```

00008                                     {
00009
00010     TCPhandler tcp(client_socket, client, signal_listener);
00011
00012     bool end = false;
00013
00014     std::thread sender(read_queue, s, &end, synch_var, busy, &tcp);
00015
00016     uint8_t internal_buf[2048];
00017
00018     while (true) {
00019         int length = tcp.listening_for_incoming_connection(internal_buf, 1024);
00020         if (length == 0)
00021             break;
00022         if (length == -1) {
00023             tcp.create_bye();
00024             break;
00025         }
00026         if (!tcp.decipher_the_message(internal_buf, length, s, synch_var))
00027             break;
00028     }
00029
00030     if (synch_var->usernames.find(tcp.user_n) != synch_var->usernames.end())
00031         synch_var->usernames.erase(tcp.user_n);
00032
00033     end = true;
00034     {
00035         std::lock_guard<std::mutex> lock(synch_var->mtx);

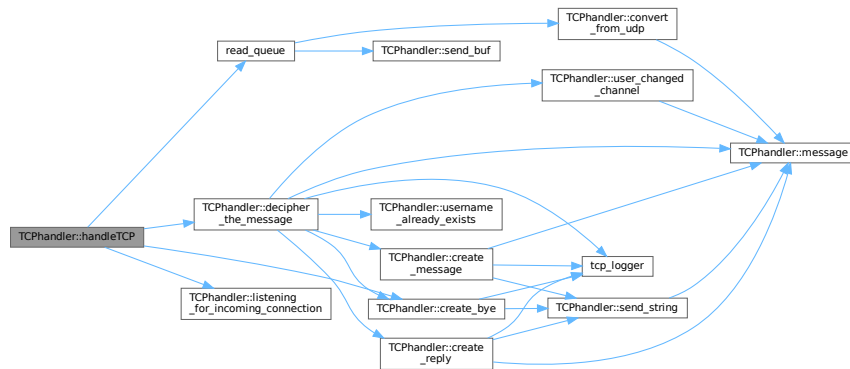
```

```

00036     synch_var->ready = true;
00037 }
00038 synch_var->cv.notify_all();
00039
00040 sender.join();
00041 shutdown(tcp.client_socket, SHUT_RDWR);
00042 close(tcp.client_socket);
00043 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



6.10.3.7 listening_for_incoming_connection()

```

int TCPHandler::listening_for_incoming_connection (
    uint8_t * buf,
    int len ) [private]

```

Definition at line 87 of file [TCPHandler.cpp](#).

```

00087 {
00088
00089     int event_count = epoll_wait(this->epoll_fd, this->events, 2, -1);
00090
00091     if (event_count == -1) {
00092         perror("epoll_wait");
00093         close(this->epoll_fd);
00094         exit(EXIT_FAILURE);
00095     } else if (event_count > 0) {
00096         for (int j = 0; j < event_count; j++) {
00097             if (events[j].data.fd == this->client_socket) { // check if EPOLLIN event has occurred
00098                 int n = recv(this->client_socket, buf, len, 0);
00099                 if (n == -1) {
00100                     std::cerr << "recvfrom failed. errno: " << errno << '\n';
00101                     continue;
00102                 } else if (n == 0) {
00103                     return 0;
00104                 } else if (n > 0) {
00105                     return n;
00106                 }
00107             } else {
00108                 return -1;
00109             }
00110         }
00111     }
00112     return 0;

```

```
00113 }
```

Here is the caller graph for this function:



6.10.3.8 message()

```

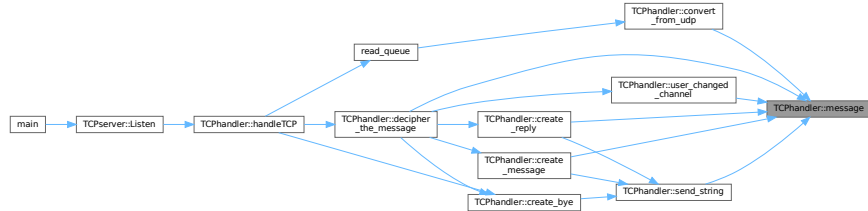
void TCPHandler::message (
    uint8_t * buf,
    int message_length,
    std::stack< UserInfo > * s,
    synch * synch_var,
    std::string & channel ) [private]
  
```

Definition at line 219 of file [TCPHandler.cpp](#).

```

00220                                     {
00221     struct sockaddr_in blank;
00222     {
00223         std::lock_guard<std::mutex> lock(synch_var->mtx);
00224         s->emplace(blank, buf, message_length, channel, true, this->client_socket);
00225         synch_var->ready = true;
00226     }
00227     synch_var->cv.notify_all();
00228 }
  
```

Here is the caller graph for this function:



6.10.3.9 send_buf()

```

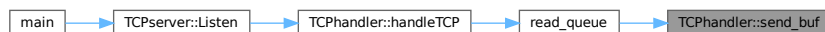
void TCPHandler::send_buf (
    uint8_t * buf,
    int length ) const
  
```

Definition at line 262 of file [TCPHandler.cpp](#).

```

00262                                     {
00263     ssize_t tx = send(this->client_socket, buf, length, 0);
00264
00265     if (tx < 0) {
00266         perror("Error sending message");
00267     }
00268 }
  
```

Here is the caller graph for this function:



6.10.3.10 send_string()

```
void TCPhandler::send_string (
    std::string & msg ) const [private]
```

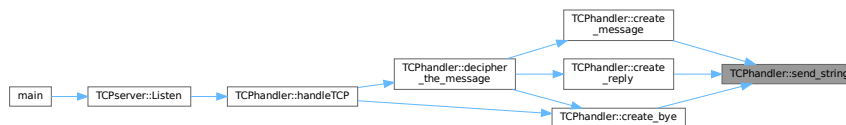
Definition at line 251 of file [TCPhandler.cpp](#).

```
00251 {
00252     const char *message = msg.c_str();
00253     size_t bytes_left = msg.size();
00254
00255     ssize_t tx = send(this->client_socket, message, bytes_left, 0);
00256
00257     if (tx < 0) {
00258         perror("Error sending message");
00259     }
00260 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



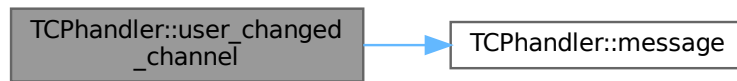
6.10.3.11 user_changed_channel()

```
void TCPhandler::user_changed_channel (
    std::stack< UserInfo > * s,
    synch * synch_var,
    const char * action ) [private]
```

Definition at line 270 of file [TCPhandler.cpp](#).

```
00270 {
00271
00272     std::stringstream ss;
00273     ss << this->display_name << " has " << std::string(action) << " " << this->channel_name << ".";
00274     std::string content = ss.str();
00275
00276     std::string message = "MSG FROM Server IS " + content + "\r\n";
00277
00278     uint8_t buffer[1024];
00279
00280     memcpy(buffer, message.c_str(), message.length());
00281
00282     this->message(buffer, message.length(), s, synch_var, this->channel_name);
00283 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.10.3.12 username_already_exists()

```

bool TCPHandler::username_already_exists (
    std::string & username,
    synch * synch_vars ) [private]
  
```

Definition at line 314 of file [TCPHandler.cpp](#).

```

00314                                     {
00315     if (!synch_vars->usernames.empty()) {
00316         if (synch_vars->usernames.find(username) != synch_vars->usernames.end())
00317             return true;
00318     }
00319     synch_vars->usernames.insert(username);
00320     this->user_n = username;
00321     return false;
00322 }
  
```

Here is the caller graph for this function:



6.10.4 Member Data Documentation

6.10.4.1 auth

```
bool TCPHandler::auth
```

Definition at line 19 of file [TCPHandler.h](#).

6.10.4.2 channel_name

```
std::string TCPHandler::channel_name
```

Definition at line 13 of file [TCPHandler.h](#).

6.10.4.3 client_addr

```
sockaddr_in TCPHandler::client_addr
```

Definition at line 18 of file [TCPHandler.h](#).

6.10.4.4 client_socket

```
int TCPHandler::client_socket
```

Definition at line 15 of file [TCPHandler.h](#).

6.10.4.5 display_name

```
std::string TCPHandler::display_name
```

Definition at line 14 of file [TCPHandler.h](#).

6.10.4.6 epoll_fd

```
int TCPHandler::epoll_fd
```

Definition at line 16 of file [TCPHandler.h](#).

6.10.4.7 events

```
epoll_event TCPHandler::events[2]
```

Definition at line 17 of file [TCPHandler.h](#).

6.10.4.8 user_n

```
std::string TCPHandler::user_n
```

Definition at line 20 of file [TCPHandler.h](#).

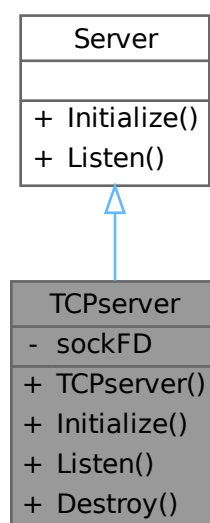
The documentation for this class was generated from the following files:

- [TCPHandler.h](#)
- [TCPHandler.cpp](#)

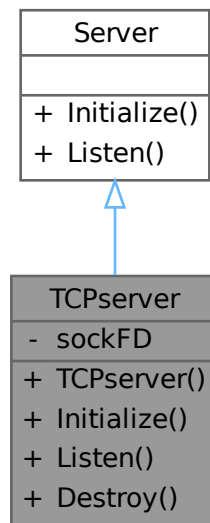
6.11 TCPserver Class Reference

```
#include <server_classes.h>
```

Inheritance diagram for TCPserver:



Collaboration diagram for TCPserver:



Public Member Functions

- [TCPserver](#) ()
- void [Initialize](#) (struct sockaddr_in *server_address) override
- void [Listen](#) (ThreadPool *tp, std::stack< [UserInfo](#) > *s, [synch](#) *synch_variables, int signal_listener) override
- void [Destroy](#) ()

Private Attributes

- int [sockFD](#)

6.11.1 Detailed Description

Definition at line 31 of file [server_classes.h](#).

6.11.2 Constructor & Destructor Documentation

6.11.2.1 TCPserver()

```
TCPserver::TCPserver ( ) [inline]
```

Definition at line 34 of file [server_classes.h](#).

```
00034         {
00035
00036     }
```

6.11.3 Member Function Documentation

6.11.3.1 Destroy()

```
void TCPserver::Destroy ( )
```

Definition at line 148 of file [server_classes.cpp](#).

```
00148     {
00149         shutdown(this->sockFD, SHUT_RDWR);
00150         close(this->sockFD);
00151     }
```

6.11.3.2 Initialize()

```
void TCPserver::Initialize (
    struct sockaddr_in * server_address ) [override], [virtual]
```

Implements [Server](#).

Definition at line 82 of file [server_classes.cpp](#).

```
00082
00083     if ((this->sockFD = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
00084         perror("socket creation failed");
00085         exit(EXIT_FAILURE);
00086     }
00087
00088     if (bind(this->sockFD, (const struct sockaddr *) server_address, sizeof(*server_address)) < 0) {
00089         perror("binding failed tcp");
00090         exit(EXIT_FAILURE);
00091     }
00092 }
```

Here is the caller graph for this function:



6.11.3.3 Listen()

```
void TCPserver::Listen (
    ThreadPool * tp,
    std::stack< UserInfo > * s,
    synch * synch_variables,
    int signal_listener ) [override], [virtual]
```

Implements [Server](#).

Definition at line 94 of file [server_classes.cpp](#).

```
00094
00095 {
00096     struct sockaddr_in client;
00097     listen(this->sockFD, 5);
00098
00099     int epoll_fd = epoll_create1(0);
00100     if (epoll_fd < 0) {
00101         std::cerr << "Unable to create epoll instance\n";
00102         exit(EXIT_FAILURE);
00103     }
00104
00105     epoll_event event;
00106     event.events = EPOLLIN | EPOLLET;
00107     event.data.fd = this->sockFD;
00108
00109     if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, this->sockFD, &event) < 0) {
00110         std::cerr << "Unable to add socket to epoll\n";
00111         exit(EXIT_FAILURE);
00112     }
00113
00114     event.data.fd = signal_listener;
00115     if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, signal_listener, &event) < 0) {
00116         std::cerr << "Unable to add socket to epoll\n";
00117         exit(EXIT_FAILURE);
00118     }
00119
00120     struct epoll_event events[2];
00121
00122     bool loop = true;
00123     while (loop) {
00124         int num_events = epoll_wait(epoll_fd, events, 2, -1); // 5 seconds timeout
00125         if (num_events < 0) {
00126             std::cerr << "Error in epoll_wait\n";
00127             exit(EXIT_FAILURE);
00128         }
00129     }
```

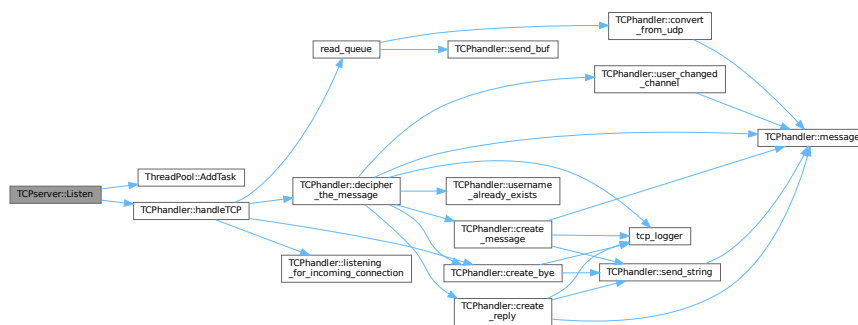


```

00128
00129     for (int i = 0; i < num_events; ++i) {
00130         if (events[i].data.fd == this->sockFD) {
00131             socklen_t len = sizeof(client);
00132             int clientSocket = accept(this->sockFD, (struct sockaddr *) &client, &len);
00133             if (clientSocket < 0) {
00134                 perror("accept failed");
00135                 continue;
00136             }
00137             tp->AddTask(
00138                 std::bind(&TCPhandler::handleTCP, clientSocket, &tp->busy_threads, s,
00139                     synch_variables, client,
00140                     signal_listener));
00141         } else {
00142             loop = false;
00143             break;
00144         }
00145     }
00146 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



6.11.4 Member Data Documentation

6.11.4.1 sockFD

```
int TCPserver::sockFD [private]
```

Definition at line 45 of file [server_classes.h](#).

The documentation for this class was generated from the following files:

- [server_classes.h](#)
- [server_classes.cpp](#)

6.12 ThreadPool Class Reference

```
#include <thread_pool.h>
```

Collaboration diagram for ThreadPool:

| ThreadPool |
|---|
| + busy_threads - mutex - condition_variable - threads - shutdown_requested - queue |
| + ThreadPool() + ~ThreadPool() + ThreadPool() + ThreadPool() + operator=() + operator=() + Shutdown() + AddTask() + QueueSize() |

Classes

- class [ThreadWorker](#)

Public Member Functions

- [ThreadPool](#) (const int size)
- [~ThreadPool](#) ()
- [ThreadPool](#) (const [ThreadPool](#) &)=delete
- [ThreadPool](#) ([ThreadPool](#) &&)=delete
- [ThreadPool](#) & [operator=](#) (const [ThreadPool](#) &)=delete
- [ThreadPool](#) & [operator=](#) ([ThreadPool](#) &&)=delete
- void [Shutdown](#) ()
- template<typename F , typename... Args>
 auto [AddTask](#) (F &&f, Args &&... args) -> std::future< decltype(f(args...))>
- int [QueueSize](#) ()

Public Attributes

- int [busy_threads](#)

Private Attributes

- std::mutex [mutex](#)
- std::condition_variable [condition_variable](#)
- std::vector< std::thread > [threads](#)
- bool [shutdown_requested](#)
- std::queue< std::function< void()> > [queue](#)

6.12.1 Detailed Description

Definition at line 11 of file [thread_pool.h](#).

6.12.2 Constructor & Destructor Documentation

6.12.2.1 ThreadPool() [1/3]

```
ThreadPool::ThreadPool (
    const int size ) [inline]
```

Definition at line 13 of file [thread_pool.h](#).

```
00013         : busy_threads(size), threads(std::vector<std::thread>(size)),
00014           shutdown_requested(false) {
00015     for (size_t i = 0; i < size; ++i) {
00016         threads[i] = std::thread(ThreadWorker(this));
00017     }
00018 }
```

6.12.2.2 ~ThreadPool()

```
ThreadPool::~~ThreadPool ( ) [inline]
```

Definition at line 20 of file [thread_pool.h](#).

```
00020     {
00021         Shutdown();
00022     }
```

Here is the call graph for this function:



6.12.2.3 ThreadPool() [2/3]

```
ThreadPool::ThreadPool (
    const ThreadPool & ) [delete]
```

6.12.2.4 ThreadPool() [3/3]

```
ThreadPool::ThreadPool (
    ThreadPool && ) [delete]
```

6.12.3 Member Function Documentation

6.12.3.1 AddTask()

```
template<typename F , typename... Args>
auto ThreadPool::AddTask (
    F && f,
    Args &&... args ) -> std::future<decltype(f(args...))> [inline]
```

Definition at line 48 of file [thread_pool.h](#).

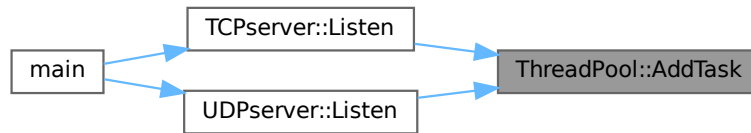
```
00048     {
00049
00050     auto task_ptr = std::make_shared<std::packaged_task<decltype(f(args...))>()>(
00051         std::bind(std::forward<F>(f), std::forward<Args>(args)...));
00052
00053     auto wrapper_func = [task_ptr]() { (*task_ptr)(); };
00054     {
00055         std::lock_guard<std::mutex> lock(mutex);
00056         queue.push(wrapper_func);
00057         // Wake up one thread if its waiting
```

```

00058         condition_variable.notify_one();
00059     }
00060
00061     // Return future from promise
00062     return task_ptr->get_future();
00063 }

```

Here is the caller graph for this function:



6.12.3.2 operator=() [1/2]

```

ThreadPool & ThreadPool::operator= (
    const ThreadPool & ) [delete]

```

6.12.3.3 operator=() [2/2]

```

ThreadPool & ThreadPool::operator= (
    ThreadPool && ) [delete]

```

6.12.3.4 QueueSize()

```

int ThreadPool::QueueSize ( ) [inline]

```

Definition at line 65 of file [thread_pool.h](#).

```

00065     {
00066         std::unique_lock<std::mutex> lock(mutex);
00067         return queue.size();
00068     }

```

6.12.3.5 Shutdown()

```

void ThreadPool::Shutdown ( ) [inline]

```

Definition at line 33 of file [thread_pool.h](#).

```

00033     {
00034     {
00035         std::lock_guard<std::mutex> lock(mutex);
00036         shutdown_requested = true;
00037         condition_variable.notify_all();
00038     }
00039
00040     for (size_t i = 0; i < threads.size(); ++i) {
00041         if (threads[i].joinable()) {
00042             threads[i].join();
00043         }
00044     }
00045 }

```

Here is the caller graph for this function:



6.12.4 Member Data Documentation

6.12.4.1 busy_threads

`int ThreadPool::busy_threads`

Definition at line 104 of file [thread_pool.h](#).

6.12.4.2 condition_variable

`std::condition_variable ThreadPool::condition_variable [private]`

Definition at line 108 of file [thread_pool.h](#).

6.12.4.3 mutex

`std::mutex ThreadPool::mutex [mutable], [private]`

Definition at line 107 of file [thread_pool.h](#).

6.12.4.4 queue

`std::queue<std::function<void()> > ThreadPool::queue [private]`

Definition at line 113 of file [thread_pool.h](#).

6.12.4.5 shutdown_requested

`bool ThreadPool::shutdown_requested [private]`

Definition at line 111 of file [thread_pool.h](#).

6.12.4.6 threads

`std::vector<std::thread> ThreadPool::threads [private]`

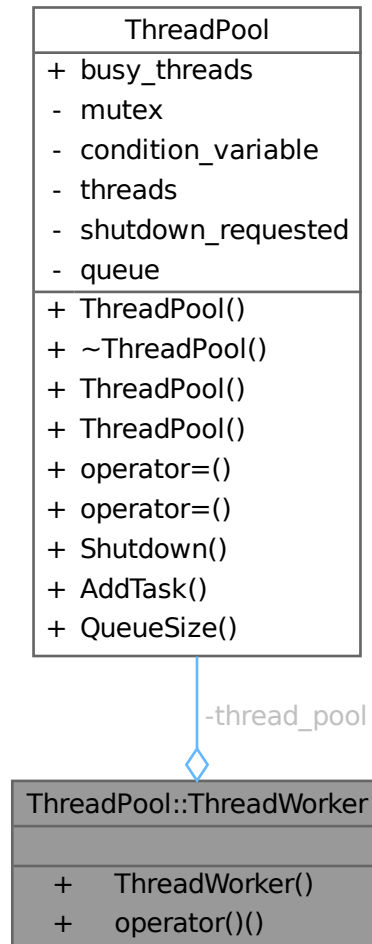
Definition at line 110 of file [thread_pool.h](#).

The documentation for this class was generated from the following file:

- [thread_pool.h](#)

6.13 ThreadPool::ThreadWorker Class Reference

Collaboration diagram for ThreadPool::ThreadWorker:



Public Member Functions

- [ThreadWorker](#) ([ThreadPool](#) *pool)
- void [operator\(\)](#) ()

Private Attributes

- [ThreadPool](#) * [thread_pool](#)

6.13.1 Detailed Description

Definition at line 71 of file [thread_pool.h](#).

6.13.2 Constructor & Destructor Documentation

6.13.2.1 ThreadWorker()

ThreadPool::ThreadWorker::ThreadWorker (
 ThreadPool * pool) [inline]

Definition at line 73 of file [thread_pool.h](#).

```
00073                                     : thread_pool(pool) {
00074                                     }
```

6.13.3 Member Function Documentation

6.13.3.1 operator()

void ThreadPool::ThreadWorker::operator() () [inline]

Definition at line 76 of file [thread_pool.h](#).

```
00076                                     {
00077         std::unique_lock<std::mutex> lock(thread_pool->mutex);
00078         while (!thread_pool->shutdown_requested ||
00079             (thread_pool->shutdown_requested && !thread_pool->queue.empty())) {
00080             thread_pool->busy_threads--;
00081             thread_pool->condition_variable.wait(lock, [this] {
00082                 return this->thread_pool->shutdown_requested || !this->thread_pool->queue.empty();
00083             });
00084             thread_pool->busy_threads++;
00085
00086             if (!this->thread_pool->queue.empty()) {
00087                 auto func = thread_pool->queue.front();
00088                 thread_pool->queue.pop();
00089
00090                 lock.unlock();
00091                 func();
00092                 lock.lock();
00093             }
00094         }
00095     }
00096 }
00097 }
```

6.13.4 Member Data Documentation

6.13.4.1 thread_pool

ThreadPool* ThreadPool::ThreadWorker::thread_pool [private]

Definition at line 100 of file [thread_pool.h](#).

The documentation for this class was generated from the following file:

- [thread_pool.h](#)

6.14 UDPhandler Class Reference

```
#include <UDPhandler.h>
```

Collaboration diagram for UDPhandler:

| UDPhandler |
|---|
| + retransmissions + timeout_chat + global_counter + client_socket + vec + events + epoll_fd + auth + client_addr + display_name + channel_name + user_n |
| + UDPhandler() + create_message() + send_message() + convert_from_tcp() + handleUDP() - decipher_the_message() - respond_to_auth() - respond_to_join() - send_confirm() - send_reply() - wait_for_the_incoming_connection() - waiting_for_confirm() - message() - buffer_validation() - change_display_name() - client_leaving() - read_channel_name() - create_bye() - username_exists() - read_packet_id() |

Public Member Functions

- [UDPhandler](#) (int ret, int t, sockaddr_in client, int kill)
- int [create_message](#) (uint8_t *buf_out, std::string &msg, bool error, std::string &name)
- void [send_message](#) (uint8_t *buf, int message_length, bool terminate)
- int [convert_from_tcp](#) (uint8_t *buf, uint8_t *tcp_buf)

Static Public Member Functions

- static void [handleUDP](#) (uint8_t *buf, sockaddr_in [client_addr](#), int length, int [retransmissions](#), int timeout, int *busy, std::stack< [UserInfo](#) > *s, [synch](#) *synch_var, int signal_listener)

Public Attributes

- int [retransmissions](#)
- int [timeout_chat](#)
- int [global_counter](#)
- int [client_socket](#)
- std::vector< int > [vec](#)
- epoll_event [events](#) [2]
- int [epoll_fd](#)
- bool [auth](#)
- sockaddr_in [client_addr](#)
- std::string [display_name](#)
- std::string [channel_name](#)
- std::string [user_n](#)

Private Member Functions

- bool [decipher_the_message](#) (uint8_t *buf, int length, std::stack< [UserInfo](#) > *s, [synch](#) *synch_var)
- int [respond_to_auth](#) (uint8_t *buf, int length, std::stack< [UserInfo](#) > *s, [synch](#) *synch_var)
- void [respond_to_join](#) (uint8_t *buf, int length, std::stack< [UserInfo](#) > *s, [synch](#) *synch_var)
- void [send_confirm](#) (uint8_t *buf)
- void [send_reply](#) (uint8_t *buf, std::string &message, bool OK)
- int [wait_for_the_incoming_connection](#) (uint8_t *buf_out, int timeout=-1)
- bool [waiting_for_confirm](#) (uint8_t *buf, int len)
- void [message](#) (uint8_t *buf, int message_length, std::stack< [UserInfo](#) > *s, [synch](#) *synch_var, std::string &channel)
- bool [buffer_validation](#) (uint8_t *buf, int message_length, int start_position, int minimal_length, int amount_↵ of_fields=2, int first_limit=20, int second_limit=20, int third_limit=5)
- void [change_display_name](#) (uint8_t *buf, bool second)
- void [client_leaving](#) (std::stack< [UserInfo](#) > *s, [synch](#) *synch_var)
- std::string [read_channel_name](#) (uint8_t *buf)
- int [create_bye](#) (uint8_t *buf)
- bool [username_exists](#) (uint8_t *buf, [synch](#) *synch_vars)

Static Private Member Functions

- static int [read_packet_id](#) (uint8_t *buf)

6.14.1 Detailed Description

Definition at line 23 of file [UDPhandler.h](#).

6.14.2 Constructor & Destructor Documentation

6.14.2.1 UDPhandler()

```
UDPhandler::UDPhandler (
    int ret,
    int t,
    sockaddr_in client,
    int kill ) [inline]
```

Definition at line 38 of file [UDPhandler.h](#).

```
00038
```

```
{
```

```

00039         this->retransmissions = ret;
00040         this->timeout_chat = t;
00041         this->global_counter = 0;
00042         this->client_socket = socket(AF_INET, SOCK_DGRAM, 0);
00043         if (this->client_socket < 0) {
00044             perror("Problem with creating response socket");
00045             exit(EXIT_FAILURE);
00046         }
00047
00048
00049         epoll_fd = epoll_create1(0);
00050         if (epoll_fd == -1) {
00051             std::cerr << "Failed to create epoll file descriptor\n";
00052             exit(EXIT_FAILURE);
00053         }
00054
00055         // setup epoll event
00056         struct epoll_event ev;
00057         ev.events = EPOLLIN | EPOLLET;
00058         ev.data.fd = this->client_socket;
00059
00060         // add socket file descriptor to epoll
00061         if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, this->client_socket, &ev) == -1) {
00062             std::cerr << "Failed to add file descriptor to epoll\n";
00063             close(epoll_fd);
00064             exit(EXIT_FAILURE);
00065         }
00066
00067         ev.data.fd = kill;
00068         if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, kill, &ev) < 0) {
00069             std::cerr << "Unable to add socket to epoll\n";
00070             exit(EXIT_FAILURE);
00071         }
00072
00073         auth = false;
00074
00075         client_addr = client;
00076
00077         channel_name = "general";
00078
00079     }

```

6.14.3 Member Function Documentation

6.14.3.1 buffer_validation()

```

bool UDHandler::buffer_validation (
    uint8_t * buf,
    int message_length,
    int start_position,
    int minimal_length,
    int amount_of_fields = 2,
    int first_limit = 20,
    int second_limit = 20,
    int third_limit = 5 ) [private]

```

Definition at line 388 of file UDHandler.cpp.

```

00389 {
00390
00391     if (message_length < minimal_length)
00392         return false;
00393
00394     size_t i = start_position;
00395
00396     size_t count = 0;
00397     while (i < message_length && buf[i] != 0x00 && count < first_limit) {
00398         i++;
00399         count++;
00400     }
00401
00402     if (i >= message_length || buf[i] != 0x00 || count < 1) {
00403         return false;
00404     }
00405     ++i;
00406
00407     count = 0;
00408     while (i < message_length && buf[i] != 0x00 && count < second_limit) {
00409         ++i;
00410         ++count;
00411     }

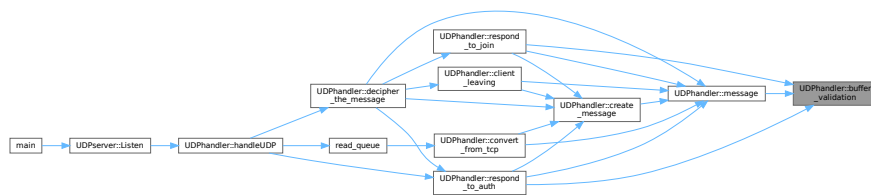
```

```

00412
00413     if (i >= message_length || buf[i] != 0x00 || count < 1) {
00414         return false;
00415     }
00416
00417     ++i;
00418
00419     if (amount_of_fields == 3) {
00420         count = 0;
00421         while (i < message_length && buf[i] != 0x00 && count < third_limit) {
00422             ++i;
00423             ++count;
00424         }
00425
00426         if (i >= message_length || buf[i] != 0x00 || count < third_limit) {
00427             return false;
00428         }
00429     }
00430
00431     return true;
00432 }

```

Here is the caller graph for this function:



6.14.3.2 change_display_name()

```

void UDPhandler::change_display_name (
    uint8_t * buf,
    bool second ) [private]

```

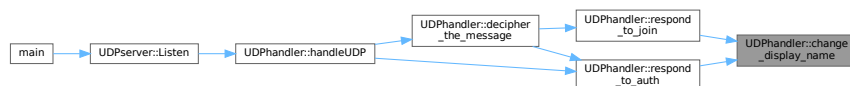
Definition at line 444 of file UDPhandler.cpp.

```

00444
00445     this->display_name.clear();
00446     int i = 3;
00447     if (second) {
00448         while (buf[i] != 0x00)
00449             i++;
00450     }
00451     i++;
00452     while (buf[i] != 0x00) {
00453         this->display_name.push_back(static_cast<char>(buf[i]));
00454         i++;
00455     }
00456 }

```

Here is the caller graph for this function:



6.14.3.3 client_leaving()

```

void UDPhandler::client_leaving (
    std::stack< UserInfo > * s,
    synch * synch_var ) [private]

```

Definition at line 258 of file UDPhandler.cpp.

```

00258

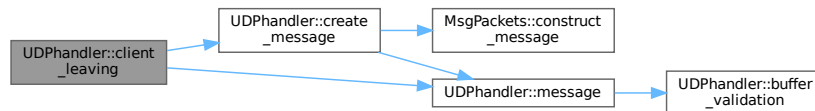
```

```

00259     std::stringstream ss;
00260     ss << this->display_name << " has left " << this->channel_name << ".";
00261     std::string message = ss.str();
00262     uint8_t buf_message[1024];
00263     std::string name = "Server";
00264     int length = this->create_message(buf_message, message, false, name);
00265     this->message(buf_message, length, s, synch_var, this->channel_name);
00266     std::this_thread::sleep_for(std::chrono::milliseconds(10));
00267 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



6.14.3.4 convert_from_tcp()

```

int UDPhandler::convert_from_tcp (
    uint8_t * buf,
    uint8_t * tcp_buf )

```

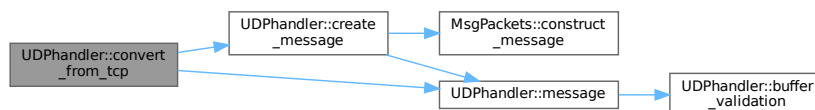
Definition at line 458 of file UDPhandler.cpp.

```

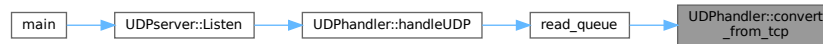
00458     {
00459
00460     std::string message;
00461     int i = 0;
00462     while (tcp_buf[i] != 0x0d) {
00463         message.push_back(static_cast<char>(tcp_buf[i]));
00464         i++;
00465     }
00466
00467     std::regex patternFromToIs(R"(FROM\s(.*)\sIS)");
00468     std::smatch matchFromToIs;
00469     std::regex_search(message, matchFromToIs, patternFromToIs);
00470     std::string name = matchFromToIs[1].str();
00471
00472     std::regex patternAfterIs(R"(IS\s(.*))");
00473     std::smatch matchAfterIs;
00474     std::regex_search(message, matchAfterIs, patternAfterIs);
00475     std::string msg = matchAfterIs[1].str();
00476
00477     return this->create_message(buf, msg, false, name);
00478 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



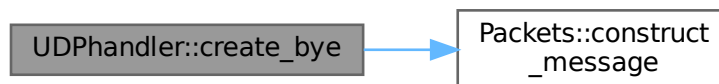
6.14.3.5 create_bye()

```
int UDPhandler::create_bye (
    uint8_t * buf ) [private]
```

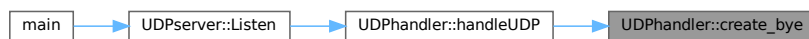
Definition at line 326 of file [UDPhandler.cpp](#).

```
00326 {
00327     Packet bye(0xFF, this->global_counter);
00328     return bye.construct_message(buf);
00329 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



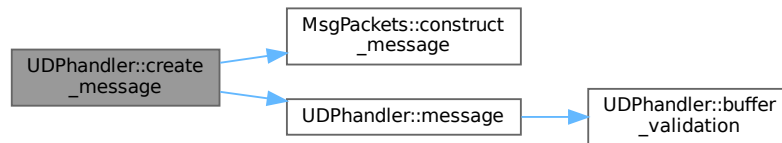
6.14.3.6 create_message()

```
int UDPhandler::create_message (
    uint8_t * buf_out,
    std::string & msg,
    bool error,
    std::string & name )
```

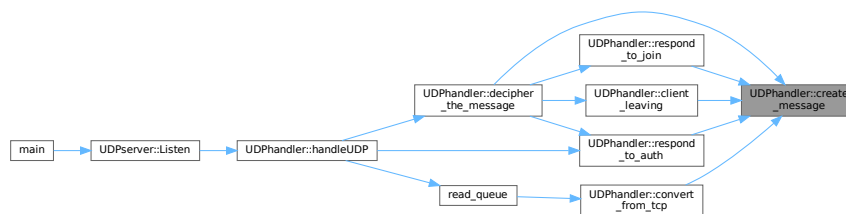
Definition at line 321 of file [UDPhandler.cpp](#).

```
00321 {
00322     MsgPacket message(error ? 0xFE : 0x04, this->global_counter, msg, name);
00323     return message.construct_message(buf_out);
00324 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.14.3.7 decipher_the_message()

```

bool UDPhandler::decipher_the_message (
    uint8_t * buf,
    int length,
    std::stack< UserInfo > * s,
    synch * synch_var ) [private]
  
```

Definition at line 89 of file UDPhandler.cpp.

```

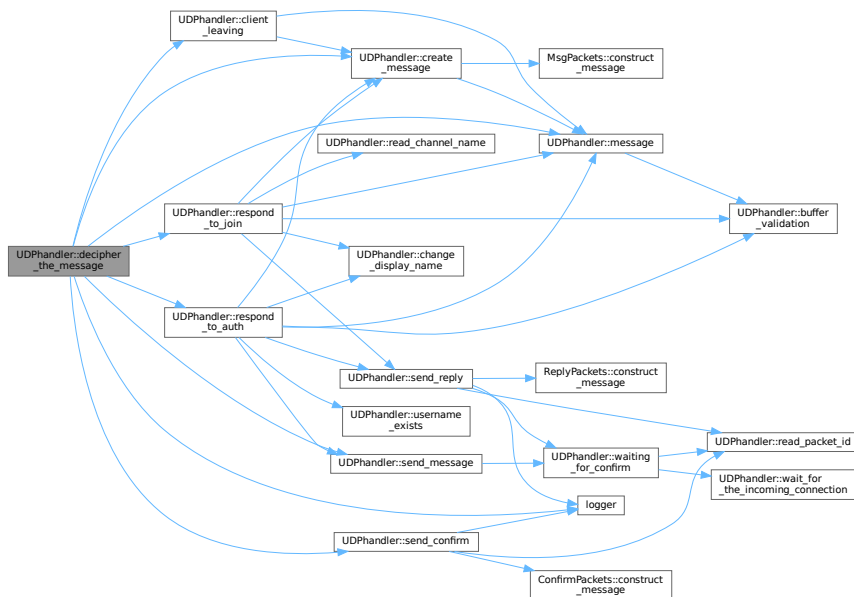
00089 {
00090     if (!this->auth) {
00091         if (buf[0] != 0x02) {
00092             if (buf[0] != 0xFF) {
00093                 if (buf[0] != 0xFE) {
00094                     uint8_t buf_int[1024];
00095                     std::string message = "You should log-in before doing anything else";
00096                     std::string name = "Server";
00097                     int length = this->create_message(buf_int, message, false, name);
00098                     this->send_message(buf_int, length, false);
00099                     return true;
00100                 }
00101             }
00102         }
00103     }
00104
00105     switch (buf[0]) {
00106     case 0x00://CONFIRM
00107         break;
00108     case 0x02://AUTH
00109         logger(this->client_addr, "AUTH", "RCV");
00110         send_confirm(buf);
00111         if (!this->auth) {
00112             respond_to_auth(buf, length, s, synch_var);
00113         } else {
00114             uint8_t buf_err[1024];
00115             std::string message = "Already authed";
00116             std::string name = "Server";
00117             int length_err = this->create_message(buf_err, message, true, name);
00118             this->send_message(buf_err, length_err, false);
00119         }
00120         break;
00121     case 0x03://JOIN
00122         logger(this->client_addr, "JOIN", "RCV");
  
```

```

00123         send_confirm(buf);
00124         respond_to_join(buf, length, s, synch_var);
00125         break;
00126     case 0x04://MSG
00127         logger(this->client_addr, "MSG", "RECV");
00128         send_confirm(buf);
00129         this->message(buf, length, s, synch_var, this->channel_name);
00130         break;
00131     case 0xFF://BYE
00132         if (this->auth) {
00133             this->client_leaving(s, synch_var);
00134         }
00135         logger(this->client_addr, "BYE", "RECV");
00136         send_confirm(buf);
00137         return false;
00138     case 0xFE://ERR
00139         if (this->auth) {
00140             this->client_leaving(s, synch_var);
00141         }
00142         logger(this->client_addr, "ERR", "RECV");
00143         send_confirm(buf);
00144         return false;
00145     default:
00146         uint8_t buf[1024];
00147         std::string message = "Unknown instruction";
00148         std::string name = "Server";
00149         int length_err = this->create_message(buf, message, true, name);
00150         this->send_message(buf, length_err, false);
00151         return false;
00152     }
00153     return true;
00154 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



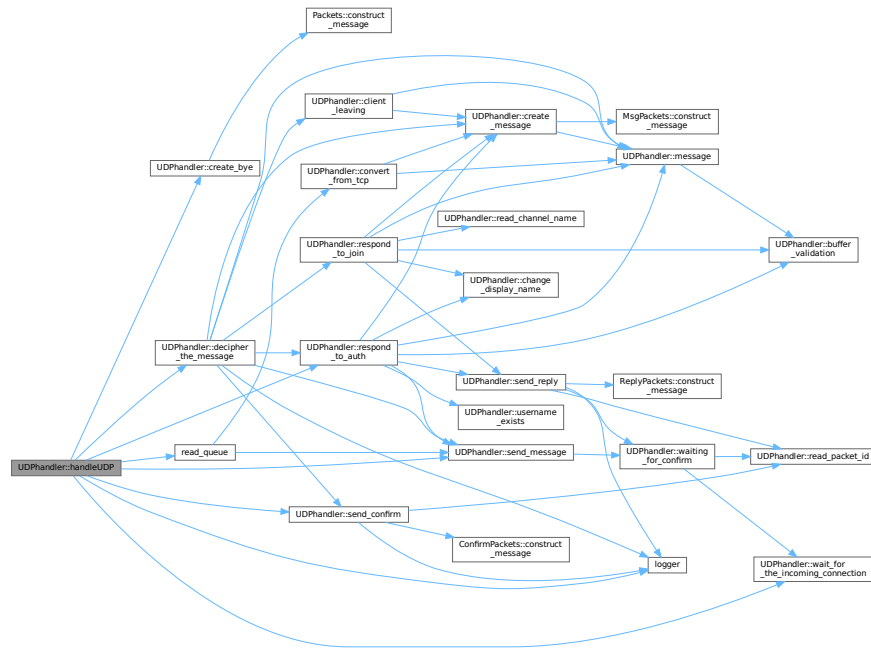
6.14.3.8 handleUDP()

```
void UDPHandler::handleUDP (
    uint8_t * buf,
    sockaddr_in client_addr,
    int length,
    int retransmissions,
    int timeout,
    int * busy,
    std::stack< UserInfo > * s,
    synch * synch_var,
    int signal_listener ) [static]
```

Definition at line 6 of file [UDPHandler.cpp](#).

```
00007 {
00008
00009     UDPHandler udp(retransmissions, timeout, client_addr, signal_listener);
00010
00011     bool end = false;
00012
00013     std::thread sender(read_queue, s, &end, synch_var, busy, &udp);
00014
00015     uint8_t internal_buf[2048];
00016     logger(udp.client_addr, "AUTH", "RCV");
00017     udp.send_confirm(buf);
00018     int result = udp.respond_to_auth(buf, length, s, synch_var);
00019
00020     if (result != -1) {
00021         while (true) {
00022             int length_internal = udp.wait_for_the_incoming_connection(internal_buf);
00023             if (length_internal == -1) {
00024                 uint8_t buf_int[256];
00025                 int length_int = udp.create_bye(buf_int);
00026                 udp.send_message(buf_int, length_int, true);
00027                 break;
00028             }
00029             if (!udp.decipher_the_message(internal_buf, length_internal, s, synch_var)) {
00030                 break;
00031             }
00032         }
00033     }
00034
00035     if (synch_var->usernames.find(udp.user_n) != synch_var->usernames.end())
00036         synch_var->usernames.erase(udp.user_n);
00037
00038     end = true;
00039     {
00040         std::lock_guard<std::mutex> lock(synch_var->mtx);
00041         synch_var->ready = true;
00042     }
00043     synch_var->cv.notify_all();
00044
00045     sender.join();
00046     close(udp.client_socket);
00047 }
```


Here is the call graph for this function:



Here is the caller graph for this function:



6.14.3.9 message()

```

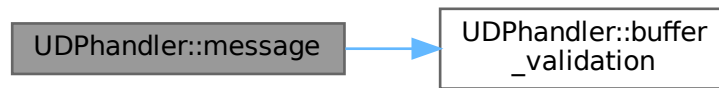
void UDPhandler::message (
    uint8_t * buf,
    int message_length,
    std::stack< UserInfo > * s,
    synch * synch_var,
    std::string & channel ) [private]
  
```

Definition at line 239 of file UDPhandler.cpp.

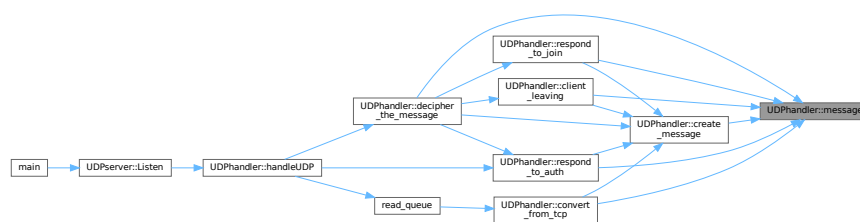
```

00240 {
00241     bool valid_message = true;
00242
00243     if (!this->buffer_validation(buf, message_length, 3, 2, 2, 20, 1400))
00244         valid_message = false;
00245
00246     if (valid_message) {
00247         {
00248             std::lock_guard<std::mutex> lock(synch_var->mtx);
00249             s->emplace(this->client_addr, buf, message_length, channel, false, 0);
00250             synch_var->ready = true;
00251         }
00252         synch_var->cv.notify_all();
00253     } else {
00254         std::cout << "Invalid message" << std::endl;
00255     }
00256 }
  
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.14.3.10 read_channel_name()

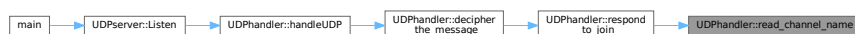
```
std::string UDPhandler::read_channel_name (
    uint8_t * buf ) [private]
```

Definition at line 434 of file [UDPhandler.cpp](#).

```

00434 {
00435     int i = 3;
00436     std::string channel;
00437     while (buf[i] != 0x00) {
00438         channel.push_back(static_cast<char>(buf[i]));
00439         i++;
00440     }
00441     return channel;
00442 }
```

Here is the caller graph for this function:



6.14.3.11 read_packet_id()

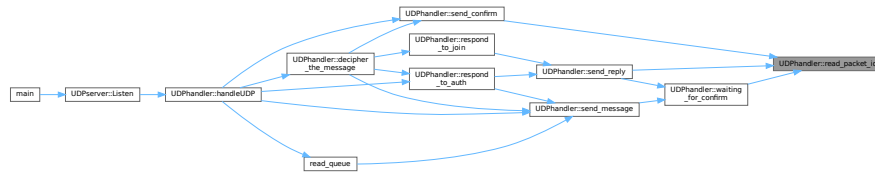
```
int UDPhandler::read_packet_id (
    uint8_t * buf ) [static], [private]
```

Definition at line 331 of file [UDPhandler.cpp](#).

```

00331 {
00332     int result = buf[1] << 8 | buf[2];
00333     return ntohs(result);
00334 }
```

Here is the caller graph for this function:



6.14.3.12 respond_to_auth()

```

int UDPhandler::respond_to_auth (
    uint8_t * buf,
    int length,
    std::stack< UserInfo > * s,
    synch * synch_var ) [private]

```

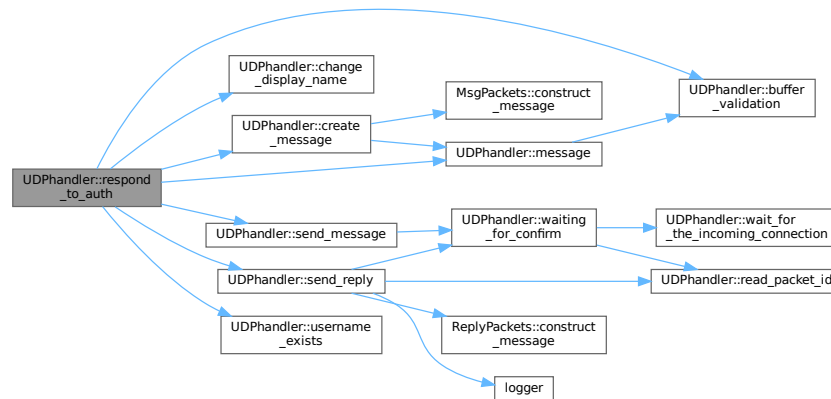
Definition at line 156 of file UDPhandler.cpp.

```

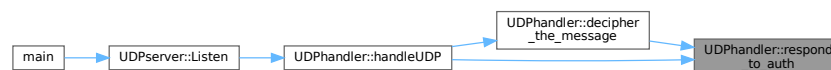
00156 {
00157     bool valid_message = true;
00158     if (buf[0] == 0xFF) {
00159         return -1;
00160     }
00161     if (buf[0] != 0x02) {
00162         uint8_t buf_int[1024];
00163         std::string message = "You should log-in before doing anything else";
00164         std::string name = "Server";
00165         int length = this->create_message(buf_int, message, false, name);
00166         this->send_message(buf_int, length, false);
00167         return 0;
00168     }
00169     if (!this->buffer_validation(buf, message_length, 3, 7, 3))
00170         valid_message = false;
00171     if (valid_message) {
00172         synch_var->un.lock();
00173         bool exists = username_exists(buf, synch_var);
00174         synch_var->un.unlock();
00175         if (!exists) {
00176             this->change_display_name(buf, true);
00177             std::string success = "Authentication is succesful";
00178             send_reply(buf, success, true);
00179             std::stringstream ss;
00180             ss << this->display_name << " has joined general.";
00181             std::string message = ss.str();
00182             uint8_t buf_message[1024];
00183             std::string name = "Server";
00184             int length = this->create_message(buf_message, message, false, name);
00185             this->message(buf_message, length, s, synch_var, this->channel_name);
00186             this->auth = true;
00187         } else {
00188             std::string failure = "Username already exists";
00189             send_reply(buf, failure, false);
00190         }
00191     } else {
00192         std::string failure = "Authentication is not succesful";
00193         send_reply(buf, failure, false);
00194     }
00195     return 0;
00196 }
00197
00201
00202

```

Here is the call graph for this function:



Here is the caller graph for this function:



6.14.3.13 respond_to_join()

```

void UDPhandler::respond_to_join (
    uint8_t * buf,
    int length,
    std::stack< UserInfo > * s,
    synch * synch_var ) [private]

```

Definition at line 204 of file UDPhandler.cpp.

```

00204 {
00205     bool valid = true;
00206
00207     if (!this->buffer_validation(buf, message_length, 3, 2))
00208         valid = false;
00209
00210     if (valid) {
00211         this->change_display_name(buf, true);
00212         std::string success = "Join is succesful";
00213         send_reply(buf, success, true);
00214
00215         std::stringstream ss;
00216         ss << this->display_name << " has left " << this->channel_name << ".";
00217         std::string message = ss.str();
00218         uint8_t buf_message[1024];
00219         std::string name = "Server";
00220         int length = this->create_message(buf_message, message, false, name);
00221         this->message(buf_message, length, s, synch_var, this->channel_name);
00222
00223         std::this_thread::sleep_for(std::chrono::milliseconds(10));
00224
00225         memset(buf_message, 0, 1024);
00226         std::stringstream joined;
00227         this->channel_name = this->read_channel_name(buf);
00228         joined << this->display_name << " has joined " << this->channel_name << ".";
00229         std::string message_new = joined.str();
00230         length = this->create_message(buf_message, message_new, false, name);
00231         this->message(buf_message, length, s, synch_var, this->channel_name);
00232     } else {
00233         std::string failure = "Join is not succesful";
00234     }

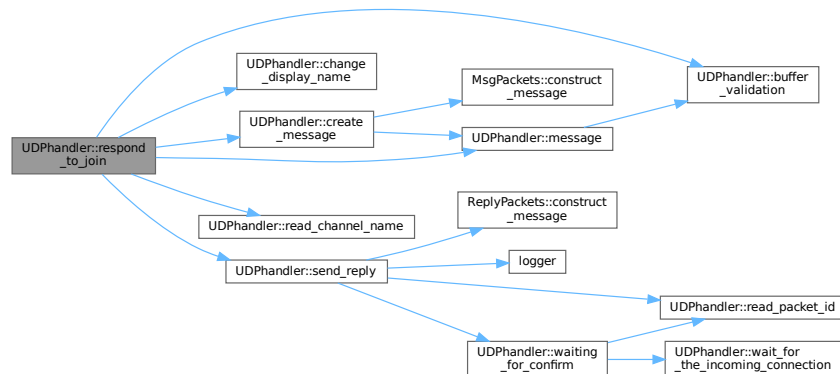
```

```

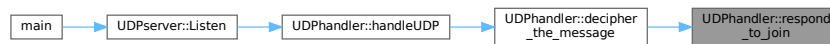
00235         send_reply(buf, failure, false);
00236     }
00237 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



6.14.3.14 send_confirm()

```

void UDPhandler::send_confirm (
    uint8_t * buf ) [private]

```

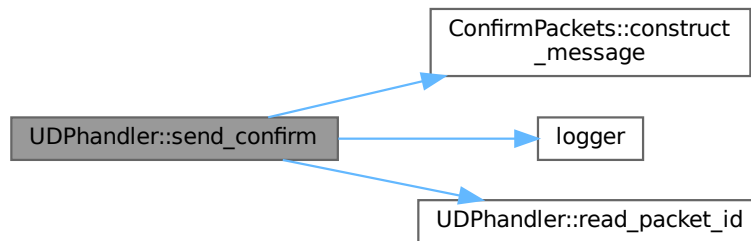
Definition at line 270 of file UDPhandler.cpp.

```

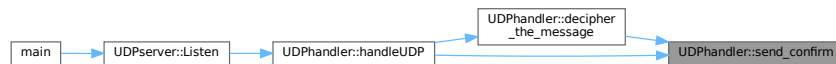
00270     {
00271         uint8_t buf_out[4];
00272         ConfirmPacket confirm(0x00, this->global_counter, read_packet_id(buf));
00273         int len = confirm.construct_message(buf_out);
00274         socklen_t address_size = sizeof(this->client_addr);
00275         long bytes_tx = sendto(this->client_socket, buf_out, len, 0, (struct sockaddr *)
00276                               &(this->client_addr), address_size);
00277         if (bytes_tx < 0) perror("ERROR: sendto");
00278         logger(this->client_addr, "CONFIRM", "SENT");
00279     }
00280
00281
00282
00283
00284
00285 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



6.14.3.15 send_message()

```

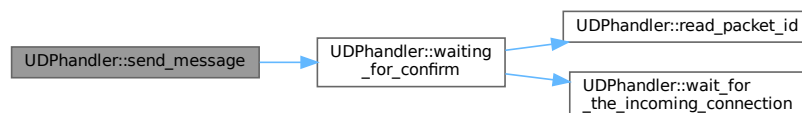
void UDPhandler::send_message (
    uint8_t * buf,
    int message_length,
    bool terminate )
  
```

Definition at line 306 of file UDPhandler.cpp.

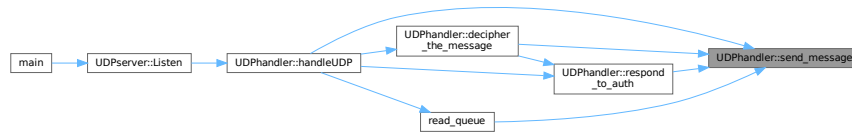
```

00306                                     {
00307     socklen_t address_size = sizeof(this->client_addr);
00308
00309     sockaddr_in backup = this->client_addr;
00310
00311     sendto(this->client_socket, buf, message_length, 0, (struct sockaddr *) &this->client_addr,
00312            address_size);
00313     this->global_counter++;
00314     if (!terminate) {
00315         if (!waiting_for_confirm(buf, message_length))
00316             std::cout << "Client didn't confirm" << std::endl;
00317     }
00318     this->client_addr = backup;
00319 }
  
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.14.3.16 send_reply()

```

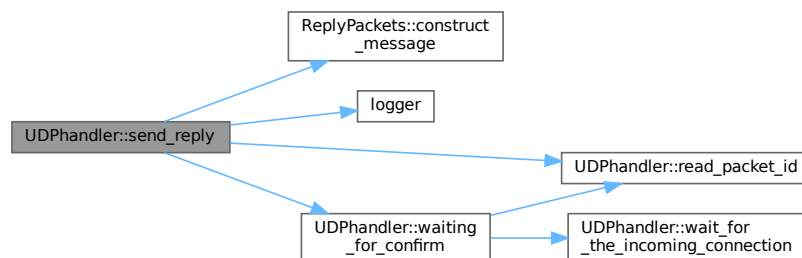
void UDPhandler::send_reply (
    uint8_t * buf,
    std::string & message,
    bool OK ) [private]
  
```

Definition at line 287 of file UDPhandler.cpp.

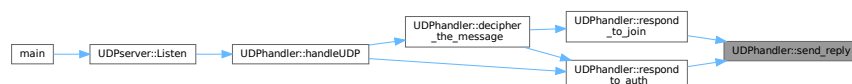
```

00287                                     {
00288     uint8_t buf_out[1024];
00289
00290     socklen_t address_size = sizeof(this->client_addr);
00291
00292     ReplyPacket reply(0x01, this->global_counter, message, OK ? 1 : 0, read_packet_id(buf));
00293     this->global_counter++;
00294
00295     int len = reply.construct_message(buf_out);
00296     long bytes_tx = sendto(this->client_socket, buf_out, len, 0, (struct sockaddr *)
00297                           &(this->client_addr),
00298                           address_size);
00299     if (bytes_tx < 0) perror("ERROR: sendto");
00300     if (!waiting_for_confirm(buf_out, len))
00301         std::cout << "Client didn't confirm" << std::endl;
00302
00303     logger(this->client_addr, "REPLY", "SENT");
00304 }
  
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.14.3.17 username_exists()

```
bool UDPhandler::username_exists (
    uint8_t * buf,
    synch * synch_vars ) [private]
```

Definition at line 485 of file UDPhandler.cpp.

```
00485 {
00486     std::string username;
00487     int i = 3;
00488     while (buf[i] != 0x00) {
00489         username.push_back(static_cast<char>(buf[i]));
00490         i++;
00491     }
00492
00493     if (!synch_vars->usernames.empty()) {
00494         if (synch_vars->usernames.find(username) != synch_vars->usernames.end())
00495             return true;
00496     }
00497     synch_vars->usernames.insert(username);
00498     this->user_n = username;
00499     return false;
00500 }
```

Here is the caller graph for this function:



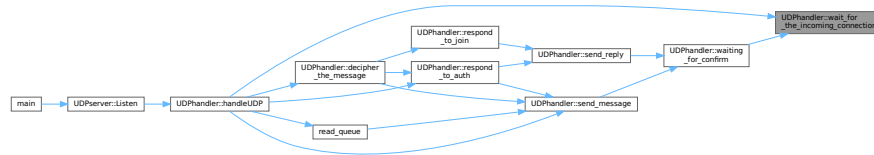
6.14.3.18 wait_for_the_incoming_connection()

```
int UDPhandler::wait_for_the_incoming_connection (
    uint8_t * buf_out,
    int timeout = -1 ) [private]
```

Definition at line 336 of file UDPhandler.cpp.

```
00336 {
00337     int event_count = epoll_wait(this->epoll_fd, this->events, 2, timeout);
00338
00339     if (event_count == -1) {
00340         perror("epoll_wait");
00341         close(this->epoll_fd);
00342         exit(EXIT_FAILURE);
00343     } else if (event_count > 0) {
00344         socklen_t len_client = sizeof(this->client_addr);
00345         for (int j = 0; j < event_count; j++) {
00346             if (events[j].data.fd == this->client_socket) { // check if EPOLLIN event has occurred
00347                 int n = recvfrom(this->client_socket, buf_out, 1024, 0, (struct sockaddr *)
&this->client_addr,
00348                                 &len_client);
00349                 if (n == -1) {
00350                     std::cerr << "recvfrom failed. errno: " << errno << '\n';
00351                     continue;
00352                 }
00353                 if (n > 0) {
00354                     return n;
00355                 }
00356             } else {
00357                 return -1;
00358             }
00359         }
00360     }
00361     return 0;
00362 }
```


Here is the caller graph for this function:



6.14.3.19 waiting_for_confirm()

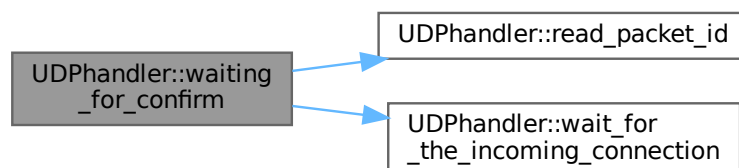
```
bool UDPhandler::waiting_for_confirm (
    uint8_t * buf,
    int len ) [private]
```

Definition at line 364 of file UDPhandler.cpp.

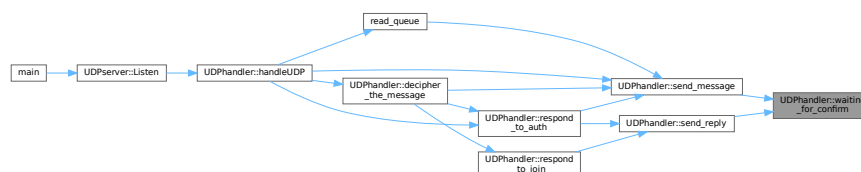
```

00364                                     {
00365     uint8_t buffer[1024];
00366     bool confirmed = false;
00367     for (int i = 0; i < this->retransmissions; ++i) {
00368         int result = this->wait_for_the_incoming_connection(buffer, this->timeout_chat);
00369         if (result > 0) {
00370             if (buffer[0] == 0x00 && read_packet_id(buffer) == read_packet_id(buf)) {
00371                 confirmed = true;
00372             }
00373         } else if (result == -1) {
00374             return true;
00375         }
00376         if (confirmed) {
00377             break;
00378         } else {
00379             socklen_t len_client = sizeof(client_addr);
00380             long bytes_tx = sendto(this->client_socket, buf, len, 0, (struct sockaddr *)
00381 &(this->client_addr),
00382                                     len_client);
00382             if (bytes_tx < 0) perror("ERROR: sendto");
00383         }
00384     }
00385     return confirmed;
00386 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.14.4 Member Data Documentation

6.14.4.1 auth

`bool UDPhandler::auth`

Definition at line 32 of file [UDPhandler.h](#).

6.14.4.2 channel_name

`std::string UDPhandler::channel_name`

Definition at line 35 of file [UDPhandler.h](#).

6.14.4.3 client_addr

`sockaddr_in UDPhandler::client_addr`

Definition at line 33 of file [UDPhandler.h](#).

6.14.4.4 client_socket

`int UDPhandler::client_socket`

Definition at line 28 of file [UDPhandler.h](#).

6.14.4.5 display_name

`std::string UDPhandler::display_name`

Definition at line 34 of file [UDPhandler.h](#).

6.14.4.6 epoll_fd

`int UDPhandler::epoll_fd`

Definition at line 31 of file [UDPhandler.h](#).

6.14.4.7 events

`epoll_event UDPhandler::events[2]`

Definition at line 30 of file [UDPhandler.h](#).

6.14.4.8 global_counter

`int UDPhandler::global_counter`

Definition at line 27 of file [UDPhandler.h](#).

6.14.4.9 retransmissions

`int UDPhandler::retransmissions`

Definition at line 25 of file [UDPhandler.h](#).

6.14.4.10 timeout_chat

`int UDPhandler::timeout_chat`

Definition at line 26 of file [UDPhandler.h](#).

6.14.4.11 user_n

`std::string UDPhandler::user_n`

Definition at line 36 of file [UDPhandler.h](#).

6.14.4.12 vec

```
std::vector<int> UDPhandler::vec
```

Definition at line 29 of file [UDPhandler.h](#).

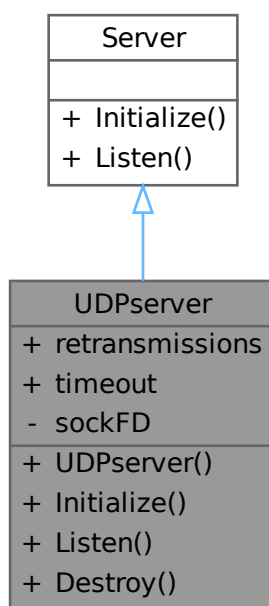
The documentation for this class was generated from the following files:

- [UDPhandler.h](#)
- [UDPhandler.cpp](#)

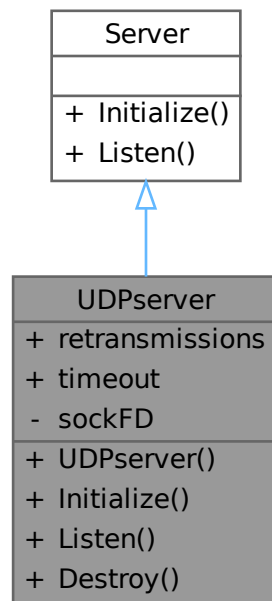
6.15 UDPserver Class Reference

```
#include <server_classes.h>
```

Inheritance diagram for UDPserver:



Collaboration diagram for UDPserver:



Public Member Functions

- [UDPserver](#) (int ret, int t)
- void [Initialize](#) (struct sockaddr_in *server_address) override
- void [Listen](#) (ThreadPool *tp, std::stack< [UserInfo](#) > *s, [synch](#) *synch_variables, int signal_listener) override
- void [Destroy](#) ()

Public Attributes

- int [retransmissions](#)
- int [timeout](#)

Private Attributes

- int [sockFD](#)

6.15.1 Detailed Description

Definition at line 48 of file [server_classes.h](#).

6.15.2 Constructor & Destructor Documentation

6.15.2.1 UDPserver()

```

UDPserver::UDPserver (
    int ret,
    int t ) [inline]
  
```

Definition at line 53 of file [server_classes.h](#).

```

00053         {
00054         this->retransmissions = ret;
  
```

```

00055         this->timeout = t;
00056     }

```

6.15.3 Member Function Documentation

6.15.3.1 Destroy()

```
void UDPserver::Destroy ( )
```

Definition at line 78 of file [server_classes.cpp](#).

```

00078     {
00079         close(this->sockFD);
00080     }

```

6.15.3.2 Initialize()

```
void UDPserver::Initialize (
```

```
    struct sockaddr_in * server_address ) [override], [virtual]
```

Implements [Server](#).

Definition at line 6 of file [server_classes.cpp](#).

```

00006     {
00007         if ((this->sockFD = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
00008             perror("socket creation failed");
00009             exit(EXIT_FAILURE);
00010         }
00011
00012         if (bind(this->sockFD, (const struct sockaddr *) server_address, sizeof(*server_address)) < 0) {
00013             perror("binding failed udp");
00014             exit(EXIT_FAILURE);
00015         }
00016     }

```

6.15.3.3 Listen()

```
void UDPserver::Listen (
```

```
    ThreadPool * tp,
    std::stack< UserInfo > * s,
    synch * synch_variables,
    int signal_listener ) [override], [virtual]
```

Implements [Server](#).

Definition at line 18 of file [server_classes.cpp](#).

```

00018     {
00019         int epoll_fd = epoll_create1(0);
00020         if (epoll_fd < 0) {
00021             std::cerr << "Unable to create epoll instance\n";
00022             exit(EXIT_FAILURE);
00023         }
00024
00025         epoll_event event;
00026         event.events = EPOLLIN | EPOLLET;
00027         event.data.fd = this->sockFD;
00028
00029         if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, this->sockFD, &event) < 0) {
00030             std::cerr << "Unable to add socket to epoll\n";
00031             exit(EXIT_FAILURE);
00032         }
00033
00034         event.data.fd = signal_listener;
00035         if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, signal_listener, &event) < 0) {
00036             std::cerr << "Unable to add socket to epoll\n";
00037             exit(EXIT_FAILURE);
00038         }
00039
00040         struct epoll_event events[2];
00041
00042         bool loop = true;
00043         while (loop) {
00044             int num_events = epoll_wait(epoll_fd, events, 2, -1); // 5 seconds timeout
00045
00046             if (num_events < 0) {
00047                 std::cerr << "Error in epoll_wait\n";
00048                 exit(EXIT_FAILURE);
00049             }
00050
00051             for (int i = 0; i < num_events; ++i) {
00052                 if (events[i].data.fd == this->sockFD) {

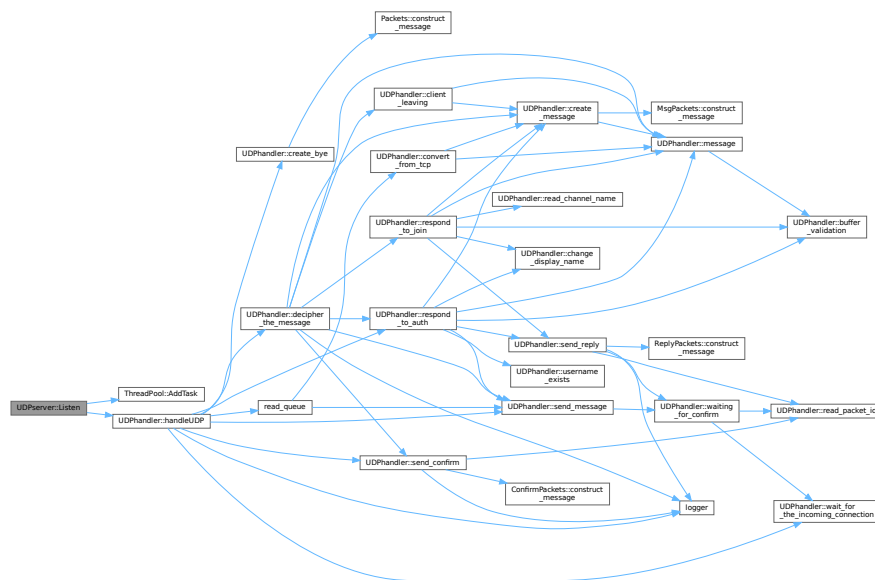
```

```

00053         uint8_t buf[1024];
00054         sockaddr_in client_addr;
00055         if (!(events[i].events & EPOLLIN))
00056             continue;
00057
00058         socklen_t len = sizeof(client_addr);
00059
00060         int n = recvfrom(this->sockFD, buf, 1024,
00061             0, (struct sockaddr *) &client_addr, &len);
00062
00063         if (n == -1) {
00064             std::cerr << "recvfrom failed. errno: " << errno << '\n';
00065             continue;
00066         }
00067
00068         tp->AddTask(std::bind(&UDPHandler::handleUDP, buf, client_addr, n,
00069             this->retransmissions, this->timeout,
00069                 &tp->busy_threads, s, synch_variables, signal_listener));
00069     } else {
00070         loop = false;
00071         break;
00072     }
00073 }
00074 }
00075 }
00076 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



6.15.4 Member Data Documentation

6.15.4.1 retransmissions

int UDPserver::retransmissions

Definition at line 50 of file [server_classes.h](#).

6.15.4.2 sockFD

```
int UDPServer::sockFD [private]
```

Definition at line 65 of file [server_classes.h](#).

6.15.4.3 timeout

```
int UDPServer::timeout
```

Definition at line 51 of file [server_classes.h](#).

The documentation for this class was generated from the following files:

- [server_classes.h](#)
- [server_classes.cpp](#)

6.16 UserInfo Struct Reference

```
#include <synch.h>
```

Collaboration diagram for UserInfo:

| UserInfo |
|--------------|
| + client |
| + tcp_socket |
| + buf |
| + length |
| + channel |
| + tcp |
| + UserInfo() |

Public Member Functions

- [UserInfo](#) (sockaddr_in c, uint8_t *m, int l, std::string name, bool t, int cs)

Public Attributes

- sockaddr_in [client](#)
- int [tcp_socket](#)
- uint8_t * [buf](#)
- int [length](#)
- std::string [channel](#)
- bool [tcp](#)

6.16.1 Detailed Description

Definition at line 39 of file [synch.h](#).

6.16.2 Constructor & Destructor Documentation

6.16.2.1 UserInfo()

```
UserInfo::UserInfo (
    sockaddr_in c,
    uint8_t * m,
    int l,
    std::string name,
    bool t,
    int cs ) [inline]
```

Definition at line 48 of file [synch.h](#).

```
00048                                     : client(c), buf(m),
00049     length(l),
00049     channel(std::move(name)), tcp(t),
00050                                     tcp_socket(cs) {};
```

6.16.3 Member Data Documentation

6.16.3.1 buf

```
uint8_t* UserInfo::buf
```

Definition at line 42 of file [synch.h](#).

6.16.3.2 channel

```
std::string UserInfo::channel
```

Definition at line 44 of file [synch.h](#).

6.16.3.3 client

```
sockaddr_in UserInfo::client
```

Definition at line 40 of file [synch.h](#).

6.16.3.4 length

```
int UserInfo::length
```

Definition at line 43 of file [synch.h](#).

6.16.3.5 tcp

```
bool UserInfo::tcp
```

Definition at line 45 of file [synch.h](#).

6.16.3.6 tcp_socket

```
int UserInfo::tcp_socket
```

Definition at line 41 of file [synch.h](#).

The documentation for this struct was generated from the following file:

- [synch.h](#)

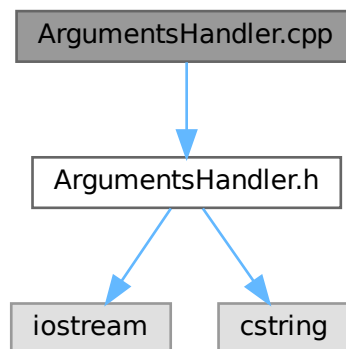
Chapter 7

File Documentation

7.1 ArgumentsHandler.cpp File Reference

```
#include "ArgumentsHandler.h"
```

Include dependency graph for ArgumentsHandler.cpp:



7.2 ArgumentsHandler.cpp

[Go to the documentation of this file.](#)

```
00001 //
00002 // Created by artem on 4/19/24.
00003 //
00004
00005
00006 #include "ArgumentsHandler.h"
00007
00008 void ArgumentsHandler::print_help() {
00009     std::cout << R"(|Argument | Default values | Type
00010
```

| | | | | |
|-------|---------|-----------|------------|--|
| 00011 | -l | 127.0.0.1 | IP address | Server listening IP address for welcome |
| | sockets | | | |
| 00012 | -p | 47356 | uint16 | Server listening port for welcome sockets |
| | | | | |
| 00013 | -d | 500 | uint16 | UDP confirmation timeout |
| | | | | |
| 00014 | -r | 3 | uint8 | Maximum number of UDP retransmissions |
| | | | | |
| 00015 | -n | 20 | uint16 | Maximum number of threads in the thread pool |
| | | | | |
| 00016 | -h | | | Prints program help output and exits |
| | | | | |

```

00017 );
00018 }
00019
00020 void ArgumentsHandler::get_args(int argc, char **argv) {
00021     this->timeout = 500;
00022     this->retransmissions = 3;
00023     this->port = 47356;
00024     this->address = new char[13];
00025     this->number_of_threads = 20;
00026     strcpy(address, "127.0.0.1");
00027
00028     for (int i = 0; i < argc; i++) {
00029         std::string arg = argv[i];
00030
00031         if (arg == "-h") {
00032             print_help();
00033             exit(0);
00034         } else if (arg == "-l") {
00035             i++;
00036             if (i < argc) {
00037                 address = argv[i];
00038             } else {
00039                 std::cout << "Nothing passed to address" << std::endl;
00040                 exit(1);
00041             }
00042         } else if (arg == "-p") {
00043             i++;
00044             if (i < argc) {
00045                 try {
00046                     port = std::stoi(argv[i]);
00047                 } catch (std::invalid_argument &) {
00048                     std::cout << "Passed non-int value to port" << std::endl;
00049                     exit(1);
00050                 }
00051             } else {
00052                 std::cout << "Nothing passed to port" << std::endl;
00053                 exit(1);
00054             }
00055         } else if (arg == "-d") {
00056             i++;
00057             if (i < argc) {
00058                 try {
00059                     this->timeout = std::stoi(argv[i]);
00060                 } catch (std::invalid_argument &) {
00061                     std::cout << "Passed non-int value to timeout" << std::endl;
00062                     exit(1);
00063                 }
00064             } else {
00065                 std::cout << "Nothing passed to timeout" << std::endl;
00066                 exit(1);
00067             }
00068         } else if (arg == "-r") {
00069             i++;
00070             if (i < argc) {
00071                 try {
00072                     this->retransmissions = std::stoi(argv[i]);
00073                 } catch (std::invalid_argument &) {
00074                     std::cout << "Passed non-int value to retransmissions" << std::endl;
00075                     exit(1);
00076                 }
00077             } else {
00078                 std::cout << "Nothing passed to retransmissions" << std::endl;
00079                 exit(1);
00080             }
00081         } else if (arg == "-n") {
00082             i++;
00083             if (i < argc) {
00084                 try {
00085                     this->number_of_threads = std::stoi(argv[i]);
00086                 } catch (std::invalid_argument &) {
00087                     std::cout << "Passed non-int value to number of threads" << std::endl;
00088                     exit(1);
00089                 }
00090             } else {
00091                 std::cout << "Nothing passed to number of threads" << std::endl;
00092                 exit(1);
00093             }
00094         }
00095     }
00096
00097 }
00098
00099
00100 int ArgumentsHandler::get_port() const {
00101     return this->port;
00102 }
00103

```

```

00104 int ArgumentsHandler::get_retransmissions() {
00105     return this->retransmissions;
00106 }
00107
00108 int ArgumentsHandler::get_timeout() {
00109     return this->timeout;
00110 }
00111
00112 char *ArgumentsHandler::get_address() {
00113     return this->address;
00114 }
00115
00116 int ArgumentsHandler::get_threads() {
00117     return this->number_of_threads;
00118 }

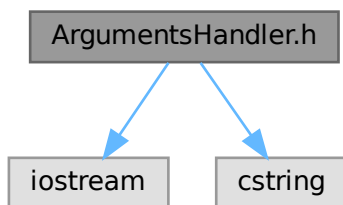
```

7.3 ArgumentsHandler.h File Reference

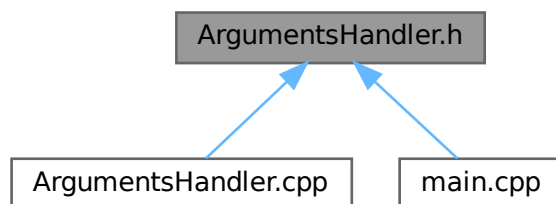
```
#include <iostream>
```

```
#include <cstring>
```

Include dependency graph for ArgumentsHandler.h:



This graph shows which files directly or indirectly include this file:



Classes

- class [ArgumentsHandler](#)

7.4 ArgumentsHandler.h

[Go to the documentation of this file.](#)

```

00001 //
00002 // Created by artem on 4/19/24.

```

```

00003 //
00004
00005 #ifndef IPK_SERVER_ARGUMENTSHANDLER_H
00006 #define IPK_SERVER_ARGUMENTSHANDLER_H
00007
00008 #include <iostream>
00009 #include <cstring>
00010
00011 class ArgumentsHandler {
00012 public:
00013     void get_args(int argc, char *argv[]);
00014
00015     int get_timeout();
00016
00017     int get_retransmissions();
00018
00019     int get_port() const;
00020
00021     char *get_address();
00022
00023     int get_threads();
00024
00025 private:
00026     int timeout;
00027     int port;
00028     char *address;
00029     int retransmissions;
00030     int number_of_threads;
00031
00032     static void print_help();
00033 };
00034
00035
00036 #endif //IPK_SERVER_ARGUMENTSHANDLER_H

```

7.5 CHAGELOG.md File Reference

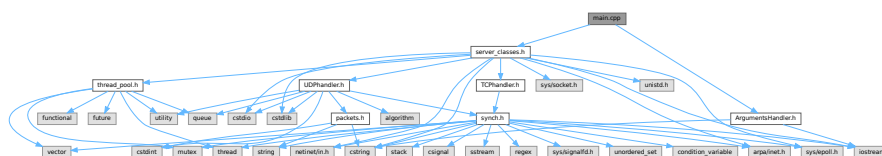
7.6 main.cpp File Reference

```

#include "server_classes.h"
#include "ArgumentsHandler.h"

```

Include dependency graph for main.cpp:



Functions

- void [init](#) (struct sockaddr_in *server_addr, int port, const char *addr)
- void [handle_sigint](#) (int sig)
- int [main](#) (int argc, char *argv[])

Variables

- int [pipefd](#) [2]

7.6.1 Function Documentation

7.6.1.1 [handle_sigint\(\)](#)

```

void handle_sigint (
    int sig )

```

Definition at line 52 of file [main.cpp](#).

```

00052 {

```

```
00053     write(pipefd[1], "X", 1);
00054 }
```

Here is the caller graph for this function:



7.6.1.2 init()

```
void init (
    struct sockaddr_in * server_addr,
    int port,
    const char * addr )
```

Definition at line 45 of file [main.cpp](#).

```
00045 {
00046     memset(server_addr, 0, sizeof(*server_addr));
00047     server_addr->sin_family = AF_INET;
00048     server_addr->sin_port = htons(port);
00049     server_addr->sin_addr.s_addr = inet_addr(addr);
00050 }
```

Here is the caller graph for this function:



7.6.1.3 main()

```
int main (
    int argc,
    char * argv[ ] )
```

Definition at line 10 of file [main.cpp](#).

```
00010 {
00011
00012     ArgumentsHandler ah{};
00013     ah.get_args(argc, argv);
00014
00015     ThreadPool tp{ah.get_threads()};
00016     std::stack<UserInfo> s;
00017     synch synch_variables(0);
00018     struct sockaddr_in *server_addr = new sockaddr_in;
00019
00020     init(server_addr, ah.get_port(), ah.get_address());
00021
00022     signal(SIGINT, handle_sigint);
00023     pipe(pipefd);
00024
00025     UDPserver udp{ah.get_retransmissions(), ah.get_timeout()};
00026     TCPserver tcp{};
00027
00028     udp.Initialize(server_addr);
00029     tcp.Initialize(server_addr);
```



```

00014
00015     ThreadPool tp{ah.get_threads()};
00016     std::stack<UserInfo> s;
00017     synch synch_variables(0);
00018     struct sockaddr_in *server_addr = new sockaddr_in;
00019
00020     init(server_addr, ah.get_port(), ah.get_address());
00021
00022     signal(SIGINT, handle_sigint);
00023     pipe(pipefd);
00024
00025     UDPserver udp{ah.get_retransmissions(), ah.get_timeout()};
00026     TCPserver tcp{};
00027
00028     udp.Initialize(server_addr);
00029     tcp.Initialize(server_addr);
00030
00031     std::thread tcpThread(&TCPserver::Listen, &tcp, &tp, &s, &synch_variables, pipefd[0]);
00032     std::thread udpThread(&UDPserver::Listen, &udp, &tp, &s, &synch_variables, pipefd[0]);
00033
00034     tcpThread.join();
00035     udpThread.join();
00036
00037     tp.Shutdown();
00038
00039     tcp.Destroy();
00040     udp.Destroy();
00041
00042     delete server_addr;
00043 }
00044
00045 void init(struct sockaddr_in *server_addr, int port, const char *addr) {
00046     memset(server_addr, 0, sizeof(*server_addr));
00047     server_addr->sin_family = AF_INET;
00048     server_addr->sin_port = htons(port);
00049     server_addr->sin_addr.s_addr = inet_addr(addr);
00050 }
00051
00052 void handle_sigint(int sig) {
00053     write(pipefd[1], "X", 1);
00054 }
00055
00056

```

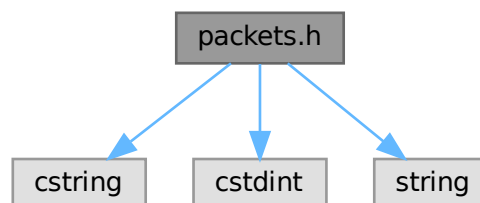
7.8 packets.h File Reference

```

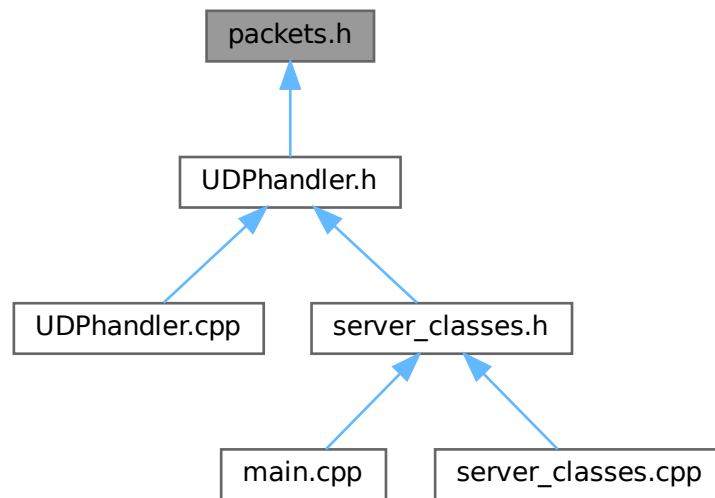
#include <cstring>
#include <cstdint>
#include <string>

```

Include dependency graph for packets.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct [Packets](#)
- struct [ConfirmPackets](#)
- struct [JoinPackets](#)
- struct [MsgPackets](#)
- struct [AuthPackets](#)
- struct [ReplyPackets](#)

Typedefs

- typedef struct [Packets](#) [Packet](#)
- typedef [ConfirmPackets](#) [ConfirmPacket](#)
- typedef [JoinPackets](#) [JoinPacket](#)
- typedef [MsgPackets](#) [MsgPacket](#)
- typedef [AuthPackets](#) [AuthPacket](#)
- typedef [ReplyPackets](#) [ReplyPacket](#)

7.8.1 Typedef Documentation

7.8.1.1 AuthPacket

```
typedef AuthPackets AuthPacket
```

7.8.1.2 ConfirmPacket

```
typedef ConfirmPackets ConfirmPacket
```

7.8.1.3 JoinPacket

```
typedef JoinPackets JoinPacket
```


7.8.1.4 MsgPacket

```
typedef MsgPackets MsgPacket
```

7.8.1.5 Packet

```
typedef struct Packets Packet
```

7.8.1.6 ReplyPacket

```
typedef ReplyPackets ReplyPacket
```

7.9 packets.h

[Go to the documentation of this file.](#)

```
00001 //
00002 // Created by artem on 4/14/24.
00003 //
00004
00005 #ifndef IPK_SERVER_PACKETS_H
00006
00007 #define IPK_SERVER_PACKETS_H
00008
00009 #include <cstring>
00010 #include <stdint>
00011 #include <string>
00012
00013 #endif //IPK_SERVER_PACKETS_H
00014
00015
00016 typedef struct Packets {
00017     uint8_t MessageType;
00018     uint16_t MessageID;
00019
00020     Packets(uint8_t type, uint16_t id) {
00021         MessageType = type;
00022         MessageID = id;
00023     }
00024
00025     virtual int construct_message(uint8_t *b) {
00026         memcpy(b, &this->MessageType, sizeof(this->MessageType));
00027         b += sizeof(this->MessageType);
00028
00029         //uint16_t ID = htons(this->MessageID);
00030         uint16_t ID = this->MessageID;
00031         memcpy(b, &ID, sizeof(ID));
00032         b += sizeof(ID);
00033         return 3;
00034     }
00035 } Packet;
00036
00037
00038 typedef struct ConfirmPackets : public Packets {
00039
00040     uint16_t Ref_MessageID;
00041
00042     ConfirmPackets(uint8_t type, uint16_t id, uint16_t ref_id) : Packets(type, id) {
00043         Ref_MessageID = ref_id;
00044     }
00045
00046     int construct_message(uint8_t *b) override {
00047         memcpy(b, &this->MessageType, sizeof(this->MessageType));
00048         b += sizeof(this->MessageType);
00049
00050         uint16_t ID = this->Ref_MessageID;
00051         memcpy(b, &ID, sizeof(ID));
00052         b += sizeof(ID);
00053         return sizeof(this->MessageType) + sizeof(ID);
00054     }
00055 } ConfirmPacket;
00056
00057
00058 typedef struct JoinPackets : public Packets {
00059
00060     std::string ChannelID;
00061     std::string DisplayName;
00062
00063     JoinPackets(uint8_t type, uint16_t id, std::string ch_id, std::string disp_name) : Packets(type,
00064 id) {
```

```

00065         ChannelID = std::move(ch_id);
00066         DisplayName = std::move(displ_name);
00067     }
00068
00069     int construct_message(uint8_t *b) override {
00070         memcpy(b, &this->MessageType, sizeof(this->MessageType));
00071         b += sizeof(this->MessageType);
00072
00073         //uint16_t ID = htons(this->MessageID);
00074         uint16_t ID = this->MessageID;
00075         memcpy(b, &ID, sizeof(ID));
00076         b += sizeof(ID);
00077
00078         memcpy(b, ChannelID.c_str(), ChannelID.length());
00079         b[ChannelID.length()] = '\0';
00080         b += ChannelID.length() + 1;
00081
00082         memcpy(b, DisplayName.c_str(), DisplayName.length());
00083         b[DisplayName.length()] = '\0';
00084         b += DisplayName.length() + 1;
00085         return sizeof(this->MessageType) + sizeof(ID) + ChannelID.length() + 1 + DisplayName.length()
+ 1;
00086     }
00087
00088 } JoinPacket;
00089
00090 // To create ERR use MsgPackets struct
00091 typedef struct MsgPackets : public Packets {
00092
00093     std::string MessageContents;
00094     std::string DisplayName;
00095
00096     MsgPackets(uint8_t type, uint16_t id, std::string content, std::string disp_name) : Packets(type,
id) {
00097         MessageContents = std::move(content);
00098         DisplayName = std::move(disp_name);
00099     }
00100
00101     int construct_message(uint8_t *b) override {
00102         memcpy(b, &this->MessageType, sizeof(this->MessageType));
00103         b += sizeof(this->MessageType);
00104
00105         //uint16_t ID = htons(this->MessageID);
00106         uint16_t ID = this->MessageID;
00107         memcpy(b, &ID, sizeof(ID));
00108         b += sizeof(ID);
00109
00110         memcpy(b, DisplayName.c_str(), DisplayName.length());
00111         b[DisplayName.length()] = '\0';
00112         b += DisplayName.length() + 1;
00113
00114         memcpy(b, MessageContents.c_str(), MessageContents.length());
00115         b[MessageContents.length()] = '\0';
00116         b += MessageContents.length() + 1;
00117         return sizeof(this->MessageType) + sizeof(ID) + DisplayName.length() + 1 +
MessageContents.length() + 1;
00118     }
00119
00120 } MsgPacket;
00121
00122 typedef struct AuthPackets : public Packets {
00123
00124     std::string Username;
00125     std::string DisplayName;
00126     std::string Secret;
00127
00128     AuthPackets(uint8_t type, uint16_t id, std::string u_n, std::string disp_name, std::string sec) :
Packets(type,
id) {
00130         Username = std::move(u_n);
00131         DisplayName = std::move(disp_name);
00132         Secret = std::move(sec);
00133     }
00134
00135     int construct_message(uint8_t *b) override {
00136         memcpy(b, &this->MessageType, sizeof(this->MessageType));
00137         b += sizeof(this->MessageType);
00138         //std::cout<<this->MessageType<<std::endl;
00139
00140         //uint16_t ID = htons(this->MessageID);
00141         uint16_t ID = this->MessageID;
00142         memcpy(b, &ID, sizeof(ID));
00143         b += sizeof(ID);
00144
00145         memcpy(b, Username.c_str(), Username.length());
00146         b[Username.length()] = '\0';

```

```

00147         b += Username.length() + 1;
00148
00149         memcpy(b, DisplayName.c_str(), DisplayName.length());
00150         b[DisplayName.length()] = '\0';
00151         b += DisplayName.length() + 1;
00152
00153         memcpy(b, Secret.c_str(), Secret.length());
00154         b[Secret.length()] = '\0';
00155         b += Secret.length() + 1;
00156         return sizeof(this->MessageType) + sizeof(ID) + Username.length() + 1 + DisplayName.length() +
1 +
00157             Secret.length() + 1;
00158     }
00159 }
00160 } AuthPacket;
00161
00162 typedef struct ReplyPackets : public Packets {
00163     std::string Message;
00164     uint8_t result;
00165     uint16_t ref_id;
00166
00167     ReplyPackets(uint8_t type, uint16_t id, std::string mes, uint8_t res, uint16_t ref) :
Packets(type, id) {
00168         Message = std::move(mes);
00169         result = res;
00170         ref_id = ref;
00171     }
00172
00173     int construct_message(uint8_t *b) override {
00174         memcpy(b, &this->MessageType, sizeof(this->MessageType));
00175         b += sizeof(this->MessageType);
00176         uint16_t ID = this->MessageID;
00177         memcpy(b, &ID, sizeof(ID));
00178         b += sizeof(ID);
00179
00180         memcpy(b, &result, sizeof(result));
00181         b += sizeof(result);
00182
00183         memcpy(b, &ref_id, sizeof(ref_id));
00184         b += sizeof(ref_id);
00185
00186         memcpy(b, Message.c_str(), Message.length());
00187         b[Message.length()] = '\0';
00188         b += Message.length() + 1;
00189
00190         return sizeof(this->MessageType) + sizeof(ID) + sizeof(result) + sizeof(ref_id) +
Message.length() + 1;
00191     }
00192 } ReplyPacket;

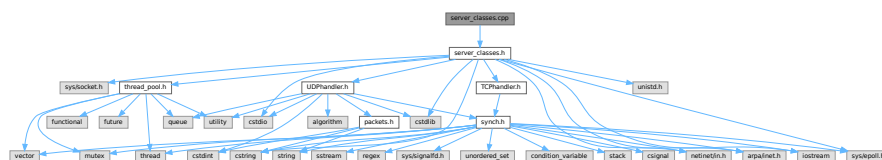
```

7.10 README.md File Reference

7.11 server_classes.cpp File Reference

```
#include "server_classes.h"
```

Include dependency graph for server_classes.cpp:



7.12 server_classes.cpp

[Go to the documentation of this file.](#)

```

00001 //
00002 // Created by artem on 4/13/24.
00003 //
00004 #include "server_classes.h"
00005
00006 void UDPserver::Initialize(struct sockaddr_in *server_address) {

```

```

00007     if ((this->sockFD = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
00008         perror("socket creation failed");
00009         exit(EXIT_FAILURE);
00010     }
00011
00012     if (bind(this->sockFD, (const struct sockaddr *) server_address, sizeof(*server_address)) < 0) {
00013         perror("binding failed udp");
00014         exit(EXIT_FAILURE);
00015     }
00016 }
00017
00018 void UDPServer::Listen(ThreadPool *tp, std::stack<UserInfo> *s, synch *synch_variables, int
signal_listener) {
00019     int epoll_fd = epoll_create1(0);
00020     if (epoll_fd < 0) {
00021         std::cerr << "Unable to create epoll instance\n";
00022         exit(EXIT_FAILURE);
00023     }
00024
00025     epoll_event event;
00026     event.events = EPOLLIN | EPOLLET;
00027     event.data.fd = this->sockFD;
00028
00029     if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, this->sockFD, &event) < 0) {
00030         std::cerr << "Unable to add socket to epoll\n";
00031         exit(EXIT_FAILURE);
00032     }
00033
00034     event.data.fd = signal_listener;
00035     if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, signal_listener, &event) < 0) {
00036         std::cerr << "Unable to add socket to epoll\n";
00037         exit(EXIT_FAILURE);
00038     }
00039
00040     struct epoll_event events[2];
00041
00042     bool loop = true;
00043     while (loop) {
00044         int num_events = epoll_wait(epoll_fd, events, 2, -1); // 5 seconds timeout
00045
00046         if (num_events < 0) {
00047             std::cerr << "Error in epoll_wait\n";
00048             exit(EXIT_FAILURE);
00049         }
00050
00051         for (int i = 0; i < num_events; ++i) {
00052             if (events[i].data.fd == this->sockFD) {
00053                 uint8_t buf[1024];
00054                 sockaddr_in client_addr;
00055                 if (!(events[i].events & EPOLLIN))
00056                     continue;
00057
00058                 socklen_t len = sizeof(client_addr);
00059
00060                 int n = recvfrom(this->sockFD, buf, 1024,
00061                                0, (struct sockaddr *) &client_addr, &len);
00062
00063                 if (n == -1) {
00064                     std::cerr << "recvfrom failed. errno: " << errno << '\n';
00065                     continue;
00066                 }
00067
00068                 tp->AddTask(std::bind(&UDPHandler::handleUDP, buf, client_addr, n,
this->retransmissions, this->timeout,
00069                                     &tp->busy_threads, s, synch_variables, signal_listener));
00070             } else {
00071                 loop = false;
00072                 break;
00073             }
00074         }
00075     }
00076 }
00077
00078 void UDPServer::Destroy() {
00079     close(this->sockFD);
00080 }
00081
00082 void TCPserver::Initialize(struct sockaddr_in *server_address) {
00083     if ((this->sockFD = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
00084         perror("socket creation failed");
00085         exit(EXIT_FAILURE);
00086     }
00087
00088     if (bind(this->sockFD, (const struct sockaddr *) server_address, sizeof(*server_address)) < 0) {
00089         perror("binding failed tcp");
00090         exit(EXIT_FAILURE);
00091     }

```

```

00092 }
00093
00094 void TCPserver::Listen(ThreadPool *tp, std::stack<UserInfo> *s, synch *synch_variables, int
    signal_listener) {
00095     struct sockaddr_in client;
00096     listen(this->sockFD, 5);
00097
00098     int epoll_fd = epoll_create1(0);
00099     if (epoll_fd < 0) {
00100         std::cerr << "Unable to create epoll instance\n";
00101         exit(EXIT_FAILURE);
00102     }
00103
00104     epoll_event event;
00105     event.events = EPOLLIN | EPOLLET;
00106     event.data.fd = this->sockFD;
00107
00108     if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, this->sockFD, &event) < 0) {
00109         std::cerr << "Unable to add socket to epoll\n";
00110         exit(EXIT_FAILURE);
00111     }
00112
00113     event.data.fd = signal_listener;
00114     if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, signal_listener, &event) < 0) {
00115         std::cerr << "Unable to add socket to epoll\n";
00116         exit(EXIT_FAILURE);
00117     }
00118
00119     struct epoll_event events[2];
00120
00121     bool loop = true;
00122     while (loop) {
00123         int num_events = epoll_wait(epoll_fd, events, 2, -1); // 5 seconds timeout
00124         if (num_events < 0) {
00125             std::cerr << "Error in epoll_wait\n";
00126             exit(EXIT_FAILURE);
00127         }
00128
00129         for (int i = 0; i < num_events; ++i) {
00130             if (events[i].data.fd == this->sockFD) {
00131                 socklen_t len = sizeof(client);
00132                 int clientSocket = accept(this->sockFD, (struct sockaddr *) &client, &len);
00133                 if (clientSocket < 0) {
00134                     perror("accept failed");
00135                     continue;
00136                 }
00137                 tp->AddTask(
00138                     std::bind(&TCPHandler::handleTCP, clientSocket, &tp->busy_threads, s,
00139                         synch_variables, client,
00140                             signal_listener));
00141             } else {
00142                 loop = false;
00143                 break;
00144             }
00145         }
00146     }
00147
00148 void TCPserver::Destroy() {
00149     shutdown(this->sockFD, SHUT_RDWR);
00150     close(this->sockFD);
00151 }

```

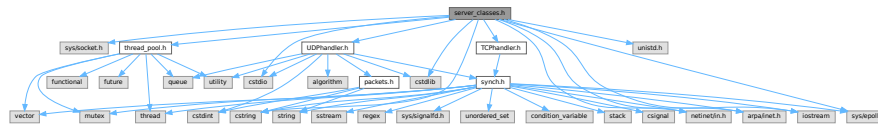
7.13 server_classes.h File Reference

```

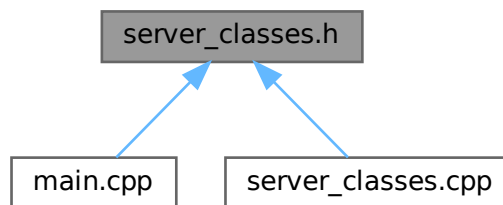
#include <sys/socket.h>
#include <cstdio>
#include <cstdlib>
#include "netinet/in.h"
#include <cstring>
#include <arpa/inet.h>
#include "thread_pool.h"
#include <iostream>
#include <sys/epoll.h>
#include <unistd.h>
#include "UDPhandler.h"
#include "TCPHandler.h"

```

Include dependency graph for `server_classes.h`:



This graph shows which files directly or indirectly include this file:



Classes

- class [Server](#)
- class [TCPserver](#)
- class [UDPserver](#)

Macros

- `#define` [PORT](#) 47356

7.13.1 Macro Definition Documentation

7.13.1.1 PORT

```
#define PORT 47356
```

Definition at line 21 of file [server_classes.h](#).

7.14 server_classes.h

[Go to the documentation of this file.](#)

```
00001 //
00002 // Created by artem on 4/13/24.
00003 //
00004
00005 #ifndef IPK_SERVER_SERVER_CLASSES_H
00006 #define IPK_SERVER_SERVER_CLASSES_H
00007
00008 #include <sys/socket.h>
00009 #include <cstdio>
00010 #include <cstdlib>
00011 #include "netinet/in.h"
00012 #include <cstring>
00013 #include <arpa/inet.h>
00014 #include "thread_pool.h"
00015 #include <iostream>
00016 #include <sys/epoll.h>
00017 #include <unistd.h>
00018 #include "UDPHandler.h"
```

```

00019 #include "TCPHandler.h"
00020
00021 #define PORT 47356
00022 #endif //IPK_SERVER_SERVER_CLASSES_H
00023
00024 class Server {
00025 public:
00026     virtual void Initialize(struct sockaddr_in *server_address) = 0;
00027
00028     virtual void Listen(ThreadPool *tp, std::stack<UserInfo> *s, synch *synch_variables, int
        signal_listener) = 0;
00029 };
00030
00031 class TCPserver : public Server {
00032 public:
00033
00034     TCPserver() {
00035
00036     }
00037
00038     void Initialize(struct sockaddr_in *server_address) override;
00039
00040     void Listen(ThreadPool *tp, std::stack<UserInfo> *s, synch *synch_variables, int signal_listener)
        override;
00041
00042     void Destroy();
00043
00044 private:
00045     int sockFD;
00046 };
00047
00048 class UDPserver : public Server {
00049 public:
00050     int retransmissions;
00051     int timeout;
00052
00053     UDPserver(int ret, int t) {
00054         this->retransmissions = ret;
00055         this->timeout = t;
00056     }
00057
00058     void Initialize(struct sockaddr_in *server_address) override;
00059
00060     void Listen(ThreadPool *tp, std::stack<UserInfo> *s, synch *synch_variables, int signal_listener)
        override;
00061
00062     void Destroy();
00063
00064 private:
00065     int sockFD;
00066 };
00067

```

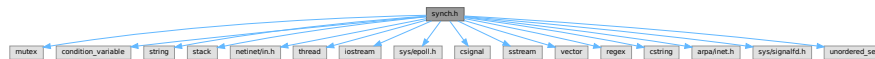
7.15 synch.h File Reference

```

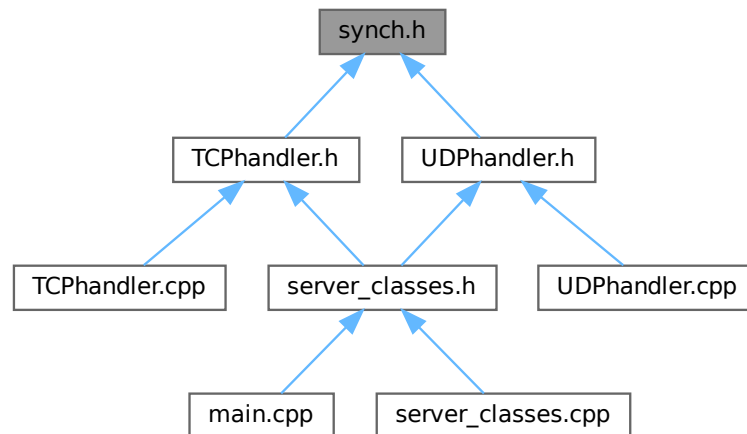
#include <mutex>
#include <condition_variable>
#include <string>
#include <stack>
#include <netinet/in.h>
#include <thread>
#include <iostream>
#include <sys/epoll.h>
#include <csignal>
#include <sstream>
#include <vector>
#include <regex>
#include <cstring>
#include <arpa/inet.h>
#include <sys/signalfd.h>
#include <unordered_set>

```

Include dependency graph for synch.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct [synch](#)
- struct [UserInfo](#)

7.16 synch.h

[Go to the documentation of this file.](#)

```

00001 //
00002 // Created by artem on 4/20/24.
00003 //
00004
00005 #ifndef IPK_SERVER_SYNCH_H
00006
00007 #define IPK_SERVER_SYNCH_H
00008
00009 #include <mutex>
00010 #include <condition_variable>
00011 #include <string>
00012 #include <stack>
00013 #include <netinet/in.h>
00014 #include <thread>
00015 #include <iostream>
00016 #include <sys/epoll.h>
00017 #include <csignal>
00018 #include <sstream>
00019 #include <vector>
00020 #include <regex>
00021 #include <cstring>
00022 #include <arpa/inet.h>
00023 #include <sys/signalfd.h>
00024 #include <unordered_set>
00025
00026 struct synch {
00027     std::mutex mtx;
00028     std::mutex waiting;
00029     std::mutex un;
  
```



```

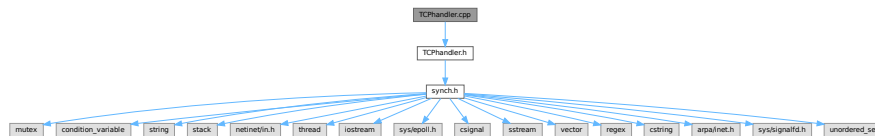
00030     bool ready;
00031     std::condition_variable cv;
00032     std::condition_variable cv2;
00033     int finished;
00034     std::unordered_set<std::string> usernames;
00035
00036     explicit synch(int b) : finished(b), ready(false) {};
00037 };
00038
00039 struct UserInfo {
00040     sockaddr_in client;
00041     int tcp_socket;
00042     uint8_t *buf;
00043     int length;
00044     std::string channel;
00045     bool tcp;
00046
00047     UserInfo(sockaddr_in c, uint8_t *m, int l, std::string name, bool t, int cs) : client(c), buf(m),
00048         length(l),
00049         channel(std::move(name)), tcp(t),
00050
00051                                     tcp_socket(cs) {};
00052 };
00053
00054 #endif //IPK_SERVER_SYNCH_H

```

7.17 TCPhandler.cpp File Reference

#include "TCPhandler.h"

Include dependency graph for TCPhandler.cpp:



Functions

- void `read_queue` (std::stack< `UserInfo` > *s, bool *terminate, `synch` *synch_vars, int *busy, `TCPhandler` *tcp)
- void `tcp_logger` (sockaddr_in client, const char *type, const char *operation)

7.17.1 Function Documentation

7.17.1.1 read_queue()

```

void read_queue (
    std::stack< UserInfo > * s,
    bool * terminate,
    synch * synch_vars,
    int * busy,
    TCPhandler * tcp )

```

Definition at line 45 of file `TCPhandler.cpp`.

```

00045 {
00046     while (!*terminate) {
00047         std::unique_lock<std::mutex> lock(synch_vars->mtx);
00048         synch_vars->cv.wait(lock, [&synch_vars] { return synch_vars->ready; });
00049
00050         synch_vars->waiting.lock();
00051         synch_vars->finished++;
00052         synch_vars->waiting.unlock();
00053
00054         if (!s->empty() && tcp->auth) {
00055             synch_vars->waiting.lock();
00056             UserInfo new_uf = s->top();
00057

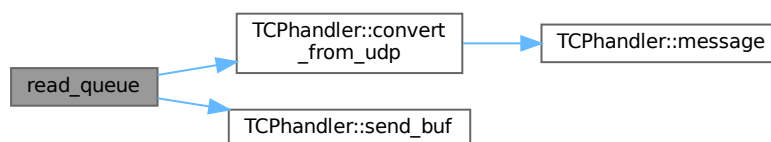
```

```

00058         synch_vars->waiting.unlock();
00059
00060         if (!new_uf.tcp) {
00061             if (new_uf.channel == tcp->channel_name) {
00062                 uint8_t buf[3048];
00063                 int length = tcp->convert_from_udp(buf, new_uf.buf);
00064                 tcp->send_buf(buf, length);
00065             }
00066         } else {
00067             if (new_uf.tcp_socket != tcp->client_socket && new_uf.channel == tcp->channel_name) {
00068                 tcp->send_buf(new_uf.buf, new_uf.length);
00069             }
00070         }
00071     }
00072
00073     if (synch_vars->finished == *busy) {
00074         synch_vars->finished = 0;
00075         synch_vars->ready = false;
00076         if (!s->empty())
00077             s->pop();
00078     }
00079
00080     lock.unlock();
00081     std::this_thread::sleep_for(std::chrono::milliseconds(100));
00082     lock.lock();
00083 }
00084 }
00085 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



7.17.1.2 tcp_logger()

```

void tcp_logger (
    sockaddr_in client,
    const char * type,
    const char * operation )

```

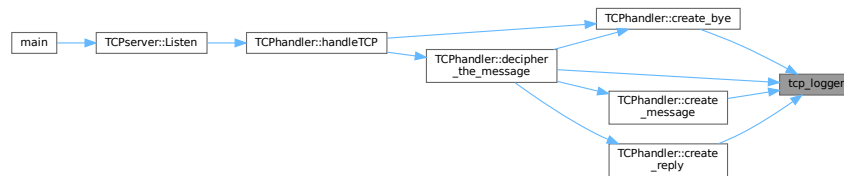
Definition at line 309 of file [TCPHandler.cpp](#).

```

00309     {
00310         std::cout << operation << " " << inet_ntoa(client.sin_addr) << ":" << ntohs(client.sin_port) << " | " <<
type
00311             << std::endl;
00312     }

```

Here is the caller graph for this function:



7.18 TCPHandler.cpp

[Go to the documentation of this file.](#)

```

00001 //
00002 // Created by artem on 4/20/24.
00003 //
00004 #include "TCPHandler.h"
00005
00006 void
00007 TCPHandler::handleTCP(int client_socket, int *busy, std::stack<UserInfo> *s, synch *synch_var,
00008 sockaddr_in client,
00009 int signal_listener) {
00010     TCPHandler tcp(client_socket, client, signal_listener);
00011
00012     bool end = false;
00013
00014     std::thread sender(read_queue, s, &end, synch_var, busy, &tcp);
00015
00016     uint8_t internal_buf[2048];
00017
00018     while (true) {
00019         int length = tcp.listening_for_incoming_connection(internal_buf, 1024);
00020         if (length == 0)
00021             break;
00022         if (length == -1) {
00023             tcp.create_bye();
00024             break;
00025         }
00026         if (!tcp.decipher_the_message(internal_buf, length, s, synch_var))
00027             break;
00028     }
00029
00030     if (synch_var->usernames.find(tcp.user_n) != synch_var->usernames.end())
00031         synch_var->usernames.erase(tcp.user_n);
00032
00033     end = true;
00034     {
00035         std::lock_guard<std::mutex> lock(synch_var->mtx);
00036         synch_var->ready = true;
00037     }
00038     synch_var->cv.notify_all();
00039
00040     sender.join();
00041     shutdown(tcp.client_socket, SHUT_RDWR);
00042     close(tcp.client_socket);
00043 }
00044
00045 void read_queue(std::stack<UserInfo> *s, bool *terminate, synch *synch_vars, int *busy, TCPHandler
00046 *tcp) {
00047     while (!*terminate) {
00048         std::unique_lock<std::mutex> lock(synch_vars->mtx);
00049         synch_vars->cv.wait(lock, [&synch_vars] { return synch_vars->ready; });
00050
00051         synch_vars->waiting.lock();
00052         synch_vars->finished++;
00053         synch_vars->waiting.unlock();
00054
00055         if (!s->empty() && tcp->auth) {
00056             synch_vars->waiting.lock();
00057             UserInfo new_uf = s->top();
00058             synch_vars->waiting.unlock();
00059
00060             if (!new_uf.tcp) {
00061                 if (new_uf.channel == tcp->channel_name) {
00062                     uint8_t buf[3048];

```

```

00063         int length = tcp->convert_from_udp(buf, new_uf.buf);
00064         tcp->send_buf(buf, length);
00065     }
00066     } else {
00067         if (new_uf.tcp_socket != tcp->client_socket && new_uf.channel == tcp->channel_name) {
00068             tcp->send_buf(new_uf.buf, new_uf.length);
00069         }
00070     }
00071 }
00072
00073 if (synch_vars->finished == *busy) {
00074     synch_vars->finished = 0;
00075     synch_vars->ready = false;
00076     if (!s->empty())
00077         s->pop();
00078 }
00079
00080 lock.unlock();
00081 std::this_thread::sleep_for(std::chrono::milliseconds(100));
00082 lock.lock();
00083
00084 }
00085 }
00086
00087 int TCPHandler::listening_for_incoming_connection(uint8_t *buf, int len) {
00088     int event_count = epoll_wait(this->epoll_fd, this->events, 2, -1);
00089
00090     if (event_count == -1) {
00091         perror("epoll_wait");
00092         close(this->epoll_fd);
00093         exit(EXIT_FAILURE);
00094     } else if (event_count > 0) {
00095         for (int j = 0; j < event_count; j++) {
00096             if (events[j].data.fd == this->client_socket) { // check if EPOLLIN event has occurred
00097                 int n = recv(this->client_socket, buf, len, 0);
00098                 if (n == -1) {
00099                     std::cerr << "recvfrom failed. errno: " << errno << '\n';
00100                     continue;
00101                 } else if (n == 0) {
00102                     return 0;
00103                 } else if (n > 0) {
00104                     return n;
00105                 }
00106             } else {
00107                 return -1;
00108             }
00109         }
00110     }
00111 }
00112 return 0;
00113 }
00114
00115 bool TCPHandler::decipher_the_message(uint8_t *buf, int length, std::stack<UserInfo> *s, synch
*synch_var) {
00116     std::string out_str;
00117     for (int i = 0; i < length - 2; ++i) {
00118         out_str += static_cast<char>(buf[i]);
00119     }
00120
00121     std::stringstream iss(out_str);
00122     std::vector<std::string> result;
00123     for (std::string element; std::getline(iss, element, ' ');) {
00124         result.push_back(element);
00125     }
00126
00127     if (!this->auth) {
00128         if (result[0] != "AUTH") {
00129             if (result[0] != "BYE") {
00130                 if (result[0] != "ERR") {
00131                     this->create_message(true, "You should log-in before doing anything else");
00132                     return true;
00133                 }
00134             }
00135         }
00136     }
00137
00138     if (result[0] == "AUTH") {
00139         std::regex e("^AUTH.{1,20} AS.{1,20} USING.{1,128}$");
00140         if (!std::regex_match(out_str, e)) {
00141             std::string mes = "Wrong AUTH format";
00142             std::cout << mes << std::endl;
00143             create_message(true, "Wrong AUTH format");
00144             std::this_thread::sleep_for(std::chrono::milliseconds(10));
00145             this->create_bye();
00146             return false;
00147         }
00148         synch_var->un.lock();

```

```

00149         bool exists = username_already_exists(result[1], synch_var);
00150         synch_var->un.lock();
00151         tcp_logger(this->client_addr, "AUTH", "RECV");
00152         if (exists) {
00153             this->create_reply("NOK", "Username already exists");
00154         } else {
00155             this->create_reply("OK", "Authentication is successful");
00156             this->display_name = result[3];
00157             this->user_changed_channel(s, synch_var, "joined");
00158             this->auth = true;
00159         }
00160
00161     } else if (result[0] == "MSG") {
00162         std::regex e("^MSG FROM .{1,20} IS .{1,1400}$");
00163         if (!std::regex_match(out_str, e)) {
00164             std::string mes = "Wrong MSG format";
00165             std::cout << mes << std::endl;
00166             create_message(true, "Wrong MSG format");
00167             std::this_thread::sleep_for(std::chrono::milliseconds(10));
00168             this->create_bye();
00169             return false;
00170         }
00171         tcp_logger(this->client_addr, "MSG", "RECV");
00172         this->display_name = result[2];
00173         this->message(buf, length, s, synch_var, this->channel_name);
00174     } else if (result[0] == "JOIN") {
00175         std::regex e("^JOIN .{1,20} AS .{1,20}$");
00176         if (!std::regex_match(out_str, e)) {
00177             std::string mes = "Wrong JOIN format";
00178             create_message(true, mes.c_str());
00179             create_message(true, "Wrong JOIN format");
00180             std::this_thread::sleep_for(std::chrono::milliseconds(10));
00181             this->create_bye();
00182             return false;
00183         }
00184         if (result[1] != this->channel_name) {
00185             tcp_logger(this->client_addr, "JOIN", "RECV");
00186             this->user_changed_channel(s, synch_var, "left");
00187             std::this_thread::sleep_for(std::chrono::milliseconds(30));
00188             this->channel_name = result[1];
00189             this->display_name = result[3];
00190             this->user_changed_channel(s, synch_var, "joined");
00191             this->create_reply("OK", "Join was successful");
00192         } else {
00193             this->create_reply("NOK", "Tried to join to the current channel");
00194         }
00195     } else if (result[0] == "BYE") {
00196         tcp_logger(this->client_addr, "BYE", "RECV");
00197         if (this->auth) {
00198             user_changed_channel(s, synch_var, "left");
00199         }
00200         return false;
00201     } else if (result[0] == "ERR") {
00202         tcp_logger(this->client_addr, "ERR", "RECV");
00203         if (this->auth) {
00204             user_changed_channel(s, synch_var, "left");
00205         }
00206         this->create_bye();
00207         return false;
00208     } else {
00209         tcp_logger(this->client_addr, "UNDEFINED", "RECV");
00210         this->create_message(true, "Unknown command");
00211         std::this_thread::sleep_for(std::chrono::milliseconds(10));
00212         this->create_bye();
00213         return false;
00214     }
00215
00216     return true;
00217 }
00218
00219 void TCPhandler::message(uint8_t *buf, int message_length, std::stack<UserInfo> *s, synch *synch_var,
00220                          std::string &channel) {
00221     struct sockaddr_in blank;
00222     {
00223         std::lock_guard<std::mutex> lock(synch_var->mtx);
00224         s->emplace(blank, buf, message_length, channel, true, this->client_socket);
00225         synch_var->ready = true;
00226     }
00227     synch_var->cv.notify_all();
00228 }
00229
00230 void TCPhandler::create_reply(const char *status, const char *msg) {
00231     std::string message;
00232     message = "REPLY " + std::string(status) + " IS " + std::string(msg) + "\r\n";
00233     tcp_logger(this->client_addr, "REPLY", "SENT");
00234     this->send_string(message);
00235 }

```

```

00236
00237 void TCPHandler::create_message(bool error, const char *msg) {
00238     std::string message;
00239     error ? message = "ERR FROM SERVER IS " + std::string(msg) + "\r\n" : message = "MSG FROM SERVER
IS " +
00240                                     std::string(msg) +
"\r\n";
00241     tcp_logger(this->client_addr, "MSG", "SENT");
00242     this->send_string(message);
00243 }
00244
00245 void TCPHandler::create_bye() {
00246     std::string bye = "BYE\r\n";
00247     tcp_logger(this->client_addr, "BYE", "SENT");
00248     this->send_string(bye);
00249 }
00250
00251 void TCPHandler::send_string(std::string &msg) const {
00252     const char *message = msg.c_str();
00253     size_t bytes_left = msg.size();
00254
00255     ssize_t tx = send(this->client_socket, message, bytes_left, 0);
00256
00257     if (tx < 0) {
00258         perror("Error sending message");
00259     }
00260 }
00261
00262 void TCPHandler::send_buf(uint8_t *buf, int length) const {
00263     ssize_t tx = send(this->client_socket, buf, length, 0);
00264
00265     if (tx < 0) {
00266         perror("Error sending message");
00267     }
00268 }
00269
00270 void TCPHandler::user_changed_channel(std::stack<UserInfo> *s, synch *synch_var, const char *action) {
00271
00272     std::stringstream ss;
00273     ss << this->display_name << " has " << std::string(action) << " " << this->channel_name << ".";
00274     std::string content = ss.str();
00275
00276     std::string message = "MSG FROM Server IS " + content + "\r\n";
00277
00278     uint8_t buffer[1024];
00279
00280     memcpy(buffer, message.c_str(), message.length());
00281
00282     this->message(buffer, message.length(), s, synch_var, this->channel_name);
00283 }
00284
00285 int TCPHandler::convert_from_udp(uint8_t *buf, uint8_t *udp_buf) {
00286     int i = 3;
00287     std::string display_n;
00288     std::string contents;
00289
00290     while (udp_buf[i] != 0x00) {
00291         display_n.push_back(static_cast<char>(udp_buf[i]));
00292         i++;
00293     }
00294
00295     i++;
00296
00297     while (udp_buf[i] != 0x00) {
00298         contents.push_back(static_cast<char>(udp_buf[i]));
00299         i++;
00300     }
00301
00302     std::string message = "MSG FROM " + display_n + " IS " + contents + "\r\n";
00303
00304     memcpy(buf, message.c_str(), message.length());
00305
00306     return message.length();
00307 }
00308
00309 void tcp_logger(sockaddr_in client, const char *type, const char *operation) {
00310     std::cout << operation << " " << inet_ntoa(client.sin_addr) << ":" << ntohs(client.sin_port) << " | " <<
type
00311         << std::endl;
00312 }
00313
00314 bool TCPHandler::username_already_exists(std::string &username, synch *synch_vars) {
00315     if (!synch_vars->usernames.empty()) {
00316         if (synch_vars->usernames.find(username) != synch_vars->usernames.end())
00317             return true;
00318     }
00319     synch_vars->usernames.insert(username);

```

```

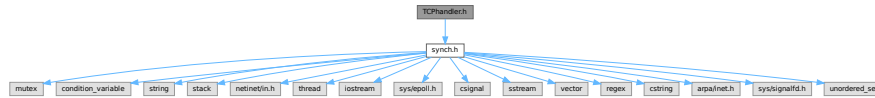
00320     this->user_n = username;
00321     return false;
00322 }

```

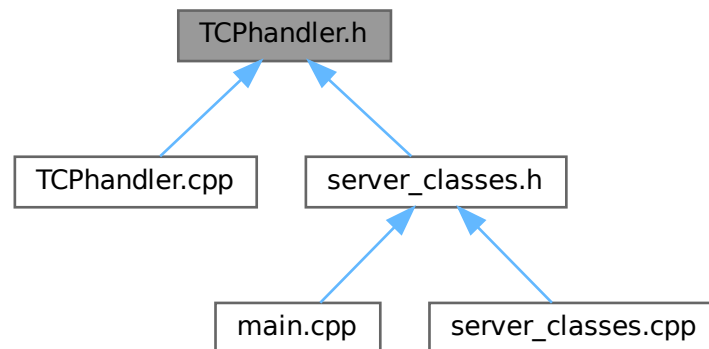
7.19 TCPhandler.h File Reference

```
#include "synch.h"
```

Include dependency graph for TCPhandler.h:



This graph shows which files directly or indirectly include this file:



Classes

- class [TCPhandler](#)

Functions

- void [tcp_logger](#) (sockaddr_in client, const char *type, const char *operation)
- void [read_queue](#) (std::stack< [UserInfo](#) > *s, bool *terminate, [synch](#) *synch_vars, int *busy, [TCPhandler](#) *tcp)

7.19.1 Function Documentation

7.19.1.1 read_queue()

```

void read_queue (
    std::stack< UserInfo > * s,
    bool * terminate,
    synch * synch_vars,
    int * busy,
    TCPhandler * tcp )

```

Definition at line 45 of file [TCPhandler.cpp](#).

```

00045 {

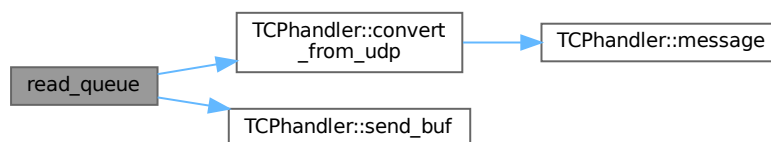
```

```

00046     while (!*terminate) {
00047         std::unique_lock<std::mutex> lock(synch_vars->mtx);
00048         synch_vars->cv.wait(lock, [&synch_vars] { return synch_vars->ready; });
00049
00050         synch_vars->waiting.lock();
00051         synch_vars->finished++;
00052         synch_vars->waiting.unlock();
00053
00054         if (!s->empty() && tcp->auth) {
00055
00056             synch_vars->waiting.lock();
00057             UserInfo new_uf = s->top();
00058             synch_vars->waiting.unlock();
00059
00060             if (!new_uf.tcp) {
00061                 if (new_uf.channel == tcp->channel_name) {
00062                     uint8_t buf[3048];
00063                     int length = tcp->convert_from_udp(buf, new_uf.buf);
00064                     tcp->send_buf(buf, length);
00065                 }
00066             } else {
00067                 if (new_uf.tcp_socket != tcp->client_socket && new_uf.channel == tcp->channel_name) {
00068                     tcp->send_buf(new_uf.buf, new_uf.length);
00069                 }
00070             }
00071         }
00072
00073         if (synch_vars->finished == *busy) {
00074             synch_vars->finished = 0;
00075             synch_vars->ready = false;
00076             if (!s->empty())
00077                 s->pop();
00078         }
00079
00080         lock.unlock();
00081         std::this_thread::sleep_for(std::chrono::milliseconds(100));
00082         lock.lock();
00083     }
00084 }
00085 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



7.19.1.2 tcp_logger()

```

void tcp_logger (
    sockaddr_in client,
    const char * type,
    const char * operation )

```

Definition at line 309 of file [TCPHandler.cpp](#).

```
00309
```

```
{
```

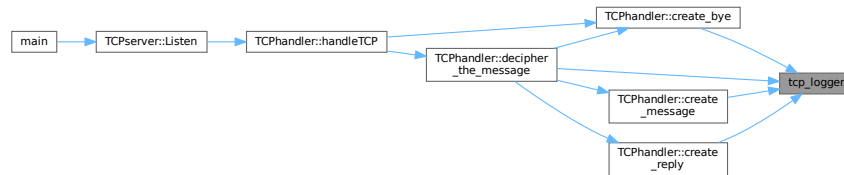


```

00310     std::cout << operation << " " << inet_ntoa(client.sin_addr) << ":" << ntohs(client.sin_port) << " | " <<
type
00311         << std::endl;
00312 }

```

Here is the caller graph for this function:



7.20 TCPHandler.h

[Go to the documentation of this file.](#)

```

00001 //
00002 // Created by artem on 4/20/24.
00003 //
00004
00005 #ifndef IPK_SERVER_TCPHANDLER_H
00006 #define IPK_SERVER_TCPHANDLER_H
00007
00008 #include "synch.h"
00009
00010 class TCPHandler {
00011 public:
00012
00013     std::string channel_name;
00014     std::string display_name;
00015     int client_socket;
00016     int epoll_fd;
00017     epoll_event events[2];
00018     sockaddr_in client_addr;
00019     bool auth;
00020     std::string user_n;
00021
00022     TCPHandler(int s, sockaddr_in c, int kill) {
00023         this->channel_name = "general";
00024         this->client_socket = s;
00025
00026         epoll_fd = epoll_create1(0);
00027         if (epoll_fd == -1) {
00028             std::cerr << "Failed to create epoll file descriptor\n";
00029             exit(EXIT_FAILURE);
00030         }
00031
00032         // setup epoll event
00033         struct epoll_event ev;
00034         ev.events = EPOLLIN;
00035         ev.data.fd = this->client_socket;
00036
00037         // add socket file descriptor to epoll
00038         if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, this->client_socket, &ev) == -1) {
00039             std::cerr << "Failed to add file descriptor to epoll\n";
00040             close(epoll_fd);
00041             exit(EXIT_FAILURE);
00042         }
00043
00044         ev.data.fd = kill;
00045         if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, kill, &ev) < 0) {
00046             std::cerr << "Unable to add socket to epoll\n";
00047             exit(EXIT_FAILURE);
00048         }
00049
00050         client_addr = c;
00051
00052         auth = false;
00053     }
00054
00055     static void handleTCP(int client_socket, int *busy, std::stack<UserInfo> *s, synch *synch_var,
sockaddr_in client,
00056                             int signal_listener);
00057
00058     void send_buf(uint8_t *buf, int length) const;
00059

```

```

00060     void create_message(bool error, const char *msg);
00061
00062     int convert_from_udp(uint8_t *buf, uint8_t *tcp_buf);
00063
00064 private:
00065     int listening_for_incoming_connection(uint8_t *buf, int len);
00066
00067     bool decipher_the_message(uint8_t *buf, int length, std::stack<UserInfo> *s, synch *synch_var);
00068
00069     void send_string(std::string &msg) const;
00070
00071     void create_reply(const char *status, const char *msg);
00072
00073     void create_bye();
00074
00075     void message(uint8_t *buf, int message_length, std::stack<UserInfo> *s, synch *synch_var,
00076                 std::string &channel);
00077
00078     void user_changed_channel(std::stack<UserInfo> *s, synch *synch_var, const char *action);
00079
00080     bool username_already_exists(std::string &username, synch *synch_vars);
00081
00082 };
00083
00084 void tcp_logger(sockaddr_in client, const char *type, const char *operation);
00085
00086 void read_queue(std::stack<UserInfo> *s, bool *terminate, synch *synch_vars, int *busy, TCPHandler
00087                *tcp);
00088
00089
00090 #endif //IPK_SERVER_TCPHANDLER_H

```

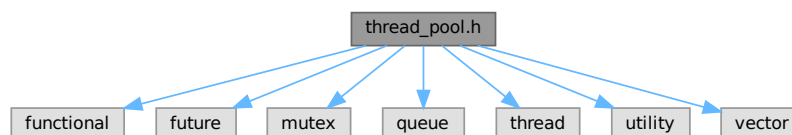
7.21 thread_pool.h File Reference

```

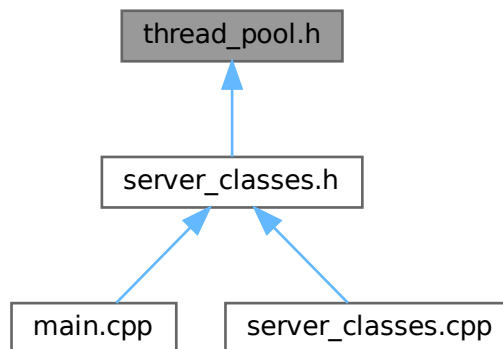
#include <functional>
#include <future>
#include <mutex>
#include <queue>
#include <thread>
#include <utility>
#include <vector>

```

Include dependency graph for thread_pool.h:



This graph shows which files directly or indirectly include this file:



Classes

- class `ThreadPool`
- class `ThreadPool::ThreadWorker`

7.22 thread_pool.h

[Go to the documentation of this file.](#)

```

00001 #pragma once
00002
00003 #include <functional>
00004 #include <future>
00005 #include <mutex>
00006 #include <queue>
00007 #include <thread>
00008 #include <utility>
00009 #include <vector>
00010
00011 class ThreadPool {
00012 public:
00013     ThreadPool(const int size) : busy_threads(size), threads(std::vector<std::thread>(size)),
00014                               shutdown_requested(false) {
00015         for (size_t i = 0; i < size; ++i) {
00016             threads[i] = std::thread(ThreadWorker(this));
00017         }
00018     }
00019
00020     ~ThreadPool() {
00021         Shutdown();
00022     }
00023
00024     ThreadPool(const ThreadPool &) = delete;
00025
00026     ThreadPool(ThreadPool &&) = delete;
00027
00028     ThreadPool &operator=(const ThreadPool &) = delete;
00029
00030     ThreadPool &operator=(ThreadPool &&) = delete;
00031
00032     // Waits until threads finish their current task and shutdowns the pool
00033     void Shutdown() {
00034         {
00035             std::lock_guard<std::mutex> lock(mutex);
00036             shutdown_requested = true;
00037             condition_variable.notify_all();
00038         }
00039
00040         for (size_t i = 0; i < threads.size(); ++i) {
00041             if (threads[i].joinable()) {
00042                 threads[i].join();
00043             }
00044         }
00045     }
00046
00047 private:
00048     int busy_threads;
00049     std::vector<std::thread> threads;
00050     bool shutdown_requested;
00051     std::condition_variable condition_variable;
00052     std::mutex mutex;
00053 };
  
```

```

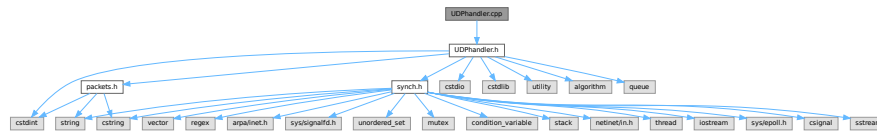
00044     }
00045 }
00046
00047 template<typename F, typename... Args>
00048 auto AddTask(F &&f, Args &&... args) -> std::future<decltype(f(args...))> {
00049
00050     auto task_ptr = std::make_shared<std::packaged_task<decltype(f(args...)) ()>>(
00051         std::bind(std::forward<F>(f), std::forward<Args>(args)...));
00052
00053     auto wrapper_func = [task_ptr]() { (*task_ptr)(); };
00054     {
00055         std::lock_guard<std::mutex> lock(mutex);
00056         queue.push(wrapper_func);
00057         // Wake up one thread if its waiting
00058         condition_variable.notify_one();
00059     }
00060
00061     // Return future from promise
00062     return task_ptr->get_future();
00063 }
00064
00065 int QueueSize() {
00066     std::unique_lock<std::mutex> lock(mutex);
00067     return queue.size();
00068 }
00069
00070 private:
00071     class ThreadWorker {
00072     public:
00073         ThreadWorker(ThreadPool *pool) : thread_pool(pool) {
00074         }
00075
00076         void operator() () {
00077             std::unique_lock<std::mutex> lock(thread_pool->mutex);
00078             while (!thread_pool->shutdown_requested ||
00079                 (thread_pool->shutdown_requested && !thread_pool->queue.empty())) {
00080                 thread_pool->busy_threads--;
00081                 thread_pool->condition_variable.wait(lock, [this] {
00082                     return this->thread_pool->shutdown_requested || !this->thread_pool->queue.empty();
00083                 });
00084                 thread_pool->busy_threads++;
00085
00086                 if (!this->thread_pool->queue.empty()) {
00087                     auto func = thread_pool->queue.front();
00088                     thread_pool->queue.pop();
00089
00090                     lock.unlock();
00091                     func();
00092                     lock.lock();
00093                 }
00094             }
00095         }
00096     };
00097
00098     private:
00099         ThreadPool *thread_pool;
00100     };
00101
00102     public:
00103         int busy_threads;
00104
00105     private:
00106         mutable std::mutex mutex;
00107         std::condition_variable condition_variable;
00108
00109         std::vector<std::thread> threads;
00110         bool shutdown_requested;
00111
00112         std::queue<std::function<void()>> queue;
00113     };
00114 };

```

7.23 UDPhandler.cpp File Reference

```
#include "UDPhandler.h"
```

Include dependency graph for UDPhandler.cpp:



Functions

- void [read_queue](#) (std::stack< [UserInfo](#) > *s, bool *terminate, [synch](#) *synch_vars, int *busy, [UDPhandler](#) *udp)
- void [logger](#) (sockaddr_in client, const char *type, const char *operation)

7.23.1 Function Documentation

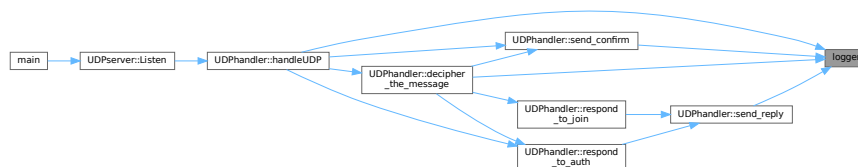
7.23.1.1 logger()

```
void logger (
    sockaddr_in client,
    const char * type,
    const char * operation )
```

Definition at line 480 of file [UDPhandler.cpp](#).

```
00480
00481     std::cout << operation << " " << inet_ntoa(client.sin_addr) << ":" << ntohs(client.sin_port) << " | " <<
    type
00482         << std::endl;
00483 }
```

Here is the caller graph for this function:



7.23.1.2 read_queue()

```
void read_queue (
    std::stack< UserInfo > * s,
    bool * terminate,
    synch * synch_vars,
    int * busy,
    UDPhandler * udp )
```

Definition at line 49 of file [UDPhandler.cpp](#).

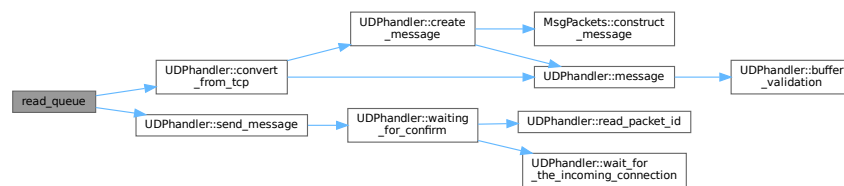
```
00049
{
00050     while (!*terminate) {
00051         std::unique_lock<std::mutex> lock(synch_vars->mtx);
00052         synch_vars->cv.wait(lock, [&synch_vars] { return synch_vars->ready; });
00053
00054         synch_vars->waiting.lock();
00055         synch_vars->finished++;
00056         synch_vars->waiting.unlock();
00057     }
```

```

00058         if (!s->empty() && udp->auth) {
00059
00060             synch_vars->waiting.lock();
00061             UserInfo new_uf = s->top();
00062             synch_vars->waiting.unlock();
00063
00064             if (new_uf.tcp) {
00065                 if (new_uf.channel == udp->channel_name) {
00066                     uint8_t buf[3048];
00067                     int length = udp->convert_from_tcp(buf, new_uf.buf);
00068                     udp->send_message(buf, length, false);
00069                 }
00070             } else {
00071                 if ((new_uf.client.sin_port != udp->client_addr.sin_port && new_uf.channel ==
udp->channel_name)) {
00072                     udp->send_message(new_uf.buf, new_uf.length, false);
00073                 }
00074             }
00075         }
00076         if (synch_vars->finished == *busy) {
00077             synch_vars->finished = 0;
00078             synch_vars->ready = false;
00079             if (!s->empty())
00080                 s->pop();
00081         }
00082         lock.unlock();
00083         std::this_thread::sleep_for(std::chrono::milliseconds(100));
00084         lock.lock();
00085     }
00086 }
00087 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



7.24 UDHandler.cpp

[Go to the documentation of this file.](#)

```

00001 //
00002 // Created by artem on 4/14/24.
00003 //
00004 #include "UDHandler.h"
00005
00006 void UDHandler::handleUDP(uint8_t *buf, sockaddr_in client_addr, int length, int retransmissions, int
timeout,
00007                          int *busy, std::stack<UserInfo> *s, synch *synch_var, int signal_listener)
00008 {
00009     UDHandler udp(retransmissions, timeout, client_addr, signal_listener);
00010
00011     bool end = false;
00012
00013     std::thread sender(read_queue, s, &end, synch_var, busy, &udp);
00014
00015     uint8_t internal_buf[2048];
00016     logger(udp.client_addr, "AUTH", "RCV");

```

```

00017     udp.send_confirm(buf);
00018     int result = udp.respond_to_auth(buf, length, s, synch_var);
00019
00020     if (result != -1) {
00021         while (true) {
00022             int length_internal = udp.wait_for_the_incoming_connection(internal_buf);
00023             if (length_internal == -1) {
00024                 uint8_t buf_int[256];
00025                 int length_int = udp.create_bye(buf_int);
00026                 udp.send_message(buf_int, length_int, true);
00027                 break;
00028             }
00029             if (!udp.decipher_the_message(internal_buf, length_internal, s, synch_var)) {
00030                 break;
00031             }
00032         }
00033     }
00034
00035     if (synch_var->usernames.find(udp.user_n) != synch_var->usernames.end())
00036         synch_var->usernames.erase(udp.user_n);
00037
00038     end = true;
00039     {
00040         std::lock_guard<std::mutex> lock(synch_var->mtx);
00041         synch_var->ready = true;
00042     }
00043     synch_var->cv.notify_all();
00044
00045     sender.join();
00046     close(udp.client_socket);
00047 }
00048
00049 void read_queue(std::stack<UserInfo> *s, bool *terminate, synch *synch_vars, int *busy, UDPhandler
*udp) {
00050     while (!*terminate) {
00051         std::unique_lock<std::mutex> lock(synch_vars->mtx);
00052         synch_vars->cv.wait(lock, [&synch_vars] { return synch_vars->ready; });
00053
00054         synch_vars->waiting.lock();
00055         synch_vars->finished++;
00056         synch_vars->waiting.unlock();
00057
00058         if (!s->empty() && udp->auth) {
00059             synch_vars->waiting.lock();
00060             UserInfo new_uf = s->top();
00061             synch_vars->waiting.unlock();
00062
00063             if (new_uf.tcp) {
00064                 if (new_uf.channel == udp->channel_name) {
00065                     uint8_t buf[3048];
00066                     int length = udp->convert_from_tcp(buf, new_uf.buf);
00067                     udp->send_message(buf, length, false);
00068                 }
00069             } else {
00070                 if ((new_uf.client.sin_port != udp->client_addr.sin_port && new_uf.channel ==
udp->channel_name)) {
00071                     udp->send_message(new_uf.buf, new_uf.length, false);
00072                 }
00073             }
00074         }
00075     }
00076     if (synch_vars->finished == *busy) {
00077         synch_vars->finished = 0;
00078         synch_vars->ready = false;
00079         if (!s->empty())
00080             s->pop();
00081     }
00082     lock.unlock();
00083     std::this_thread::sleep_for(std::chrono::milliseconds(100));
00084     lock.lock();
00085 }
00086
00087 }
00088
00089 bool UDPhandler::decipher_the_message(uint8_t *buf, int length, std::stack<UserInfo> *s, synch
*synch_var) {
00090     if (!this->auth) {
00091         if (buf[0] != 0x02) {
00092             if (buf[0] != 0xFF) {
00093                 if (buf[0] != 0xFE) {
00094                     uint8_t buf_int[1024];
00095                     std::string message = "You should log-in before doing anything else";
00096                     std::string name = "Server";
00097                     int length = this->create_message(buf_int, message, false, name);
00098                     this->send_message(buf_int, length, false);
00099                     return true;
00100                 }

```

```

00101     }
00102     }
00103 }
00104
00105 switch (buf[0]) {
00106     case 0x00://CONFIRM
00107         break;
00108     case 0x02://AUTH
00109         logger(this->client_addr, "AUTH", "RECV");
00110         send_confirm(buf);
00111         if (!this->auth) {
00112             respond_to_auth(buf, length, s, synch_var);
00113         } else {
00114             uint8_t buf_err[1024];
00115             std::string message = "Already authed";
00116             std::string name = "Server";
00117             int length_err = this->create_message(buf_err, message, true, name);
00118             this->send_message(buf_err, length_err, false);
00119         }
00120         break;
00121     case 0x03://JOIN
00122         logger(this->client_addr, "JOIN", "RECV");
00123         send_confirm(buf);
00124         respond_to_join(buf, length, s, synch_var);
00125         break;
00126     case 0x04://MSG
00127         logger(this->client_addr, "MSG", "RECV");
00128         send_confirm(buf);
00129         this->message(buf, length, s, synch_var, this->channel_name);
00130         break;
00131     case 0xFF://BYE
00132         if (this->auth) {
00133             this->client_leaving(s, synch_var);
00134         }
00135         logger(this->client_addr, "BYE", "RECV");
00136         send_confirm(buf);
00137         return false;
00138     case 0xFE://ERR
00139         if (this->auth) {
00140             this->client_leaving(s, synch_var);
00141         }
00142         logger(this->client_addr, "ERR", "RECV");
00143         send_confirm(buf);
00144         return false;
00145     default:
00146         uint8_t buf[1024];
00147         std::string message = "Unknown instruction";
00148         std::string name = "Server";
00149         int length_err = this->create_message(buf, message, true, name);
00150         this->send_message(buf, length_err, false);
00151         return false;
00152 }
00153 return true;
00154 }
00155
00156 int UDPHandler::respond_to_auth(uint8_t *buf, int message_length, std::stack<UserInfo> *s, synch
*synch_var) {
00157     bool valid_message = true;
00158
00159     if (buf[0] == 0xFF) {
00160         return -1;
00161     }
00162
00163     if (buf[0] != 0x02) {
00164         uint8_t buf_int[1024];
00165         std::string message = "You should log-in before doing anything else";
00166         std::string name = "Server";
00167         int length = this->create_message(buf_int, message, false, name);
00168         this->send_message(buf_int, length, false);
00169         return 0;
00170     }
00171
00172     if (!this->buffer_validation(buf, message_length, 3, 7, 3))
00173         valid_message = false;
00174
00175     if (valid_message) {
00176         synch_var->un.lock();
00177         bool exists = username_exists(buf, synch_var);
00178         synch_var->un.unlock();
00179         if (!exists) {
00180             this->change_display_name(buf, true);
00181             std::string success = "Authentication is succesful";
00182             send_reply(buf, success, true);
00183
00184             std::stringstream ss;
00185             ss << this->display_name << " has joined general.";
00186             std::string message = ss.str();

```



```

00187         uint8_t buf_message[1024];
00188         std::string name = "Server";
00189         int length = this->create_message(buf_message, message, false, name);
00190         this->message(buf_message, length, s, synch_var, this->channel_name);
00191         this->auth = true;
00192     } else {
00193         std::string failure = "Username already exists";
00194         send_reply(buf, failure, false);
00195     }
00196 } else {
00197     std::string failure = "Authentication is not succesful";
00198     send_reply(buf, failure, false);
00199 }
00200
00201 return 0;
00202 }
00203
00204 void UDPhandler::respond_to_join(uint8_t *buf, int message_length, std::stack<UserInfo> *s, synch
*synch_var) {
00205     bool valid = true;
00206
00207     if (!this->buffer_validation(buf, message_length, 3, 2))
00208         valid = false;
00209
00210     if (valid) {
00211         this->change_display_name(buf, true);
00212         std::string success = "Join is succesful";
00213         send_reply(buf, success, true);
00214
00215         std::stringstream ss;
00216         ss << this->display_name << " has left " << this->channel_name << ".";
00217         std::string message = ss.str();
00218         uint8_t buf_message[1024];
00219         std::string name = "Server";
00220         int length = this->create_message(buf_message, message, false, name);
00221         this->message(buf_message, length, s, synch_var, this->channel_name);
00222
00223         std::this_thread::sleep_for(std::chrono::milliseconds(10));
00224
00225         memset(buf_message, 0, 1024);
00226         std::stringstream joined;
00227         this->channel_name = this->read_channel_name(buf);
00228         joined << this->display_name << " has joined " << this->channel_name << ".";
00229         std::string message_new = joined.str();
00230         length = this->create_message(buf_message, message_new, false, name);
00231         this->message(buf_message, length, s, synch_var, this->channel_name);
00232
00233     } else {
00234         std::string failure = "Join is not succesful";
00235         send_reply(buf, failure, false);
00236     }
00237 }
00238
00239 void UDPhandler::message(uint8_t *buf, int message_length, std::stack<UserInfo> *s, synch *synch_var,
std::string &channel) {
00240     bool valid_message = true;
00241
00242     if (!this->buffer_validation(buf, message_length, 3, 2, 2, 20, 1400))
00243         valid_message = false;
00244
00245     if (valid_message) {
00246         {
00247             std::lock_guard<std::mutex> lock(synch_var->mtx);
00248             s->emplace(this->client_addr, buf, message_length, channel, false, 0);
00249             synch_var->ready = true;
00250         }
00251         synch_var->cv.notify_all();
00252     } else {
00253         std::cout << "Invalid message" << std::endl;
00254     }
00255 }
00256
00257 void UDPhandler::client_leaving(std::stack<UserInfo> *s, synch *synch_var) {
00258     std::stringstream ss;
00259     ss << this->display_name << " has left " << this->channel_name << ".";
00260     std::string message = ss.str();
00261     uint8_t buf_message[1024];
00262     std::string name = "Server";
00263     int length = this->create_message(buf_message, message, false, name);
00264     this->message(buf_message, length, s, synch_var, this->channel_name);
00265     std::this_thread::sleep_for(std::chrono::milliseconds(10));
00266 }
00267
00268 void UDPhandler::send_confirm(uint8_t *buf) {
00269     uint8_t buf_out[4];
00270 }
00271
00272

```

```

00273     ConfirmPacket confirm(0x00, this->global_counter, read_packet_id(buf));
00274
00275     int len = confirm.construct_message(buf_out);
00276
00277     socklen_t address_size = sizeof(this->client_addr);
00278
00279     long bytes_tx = sendto(this->client_socket, buf_out, len, 0, (struct sockaddr *)
&(this->client_addr),
00280                           address_size);
00281     if (bytes_tx < 0) perror("ERROR: sendto");
00282
00283     logger(this->client_addr, "CONFIRM", "SENT");
00284 }
00285 }
00286
00287 void UDPHandler::send_reply(uint8_t *buf, std::string &message, bool OK) {
00288     uint8_t buf_out[1024];
00289
00290     socklen_t address_size = sizeof(this->client_addr);
00291
00292     ReplyPacket reply(0x01, this->global_counter, message, OK ? 1 : 0, read_packet_id(buf));
00293     this->global_counter++;
00294
00295     int len = reply.construct_message(buf_out);
00296     long bytes_tx = sendto(this->client_socket, buf_out, len, 0, (struct sockaddr *)
&(this->client_addr),
00297                           address_size);
00298     if (bytes_tx < 0) perror("ERROR: sendto");
00299
00300     if (!waiting_for_confirm(buf_out, len))
00301         std::cout << "Client didn't confirm" << std::endl;
00302
00303     logger(this->client_addr, "REPLY", "SENT");
00304 }
00305
00306 void UDPHandler::send_message(uint8_t *buf, int message_length, bool terminate) {
00307     socklen_t address_size = sizeof(this->client_addr);
00308
00309     sockaddr_in backup = this->client_addr;
00310
00311     sendto(this->client_socket, buf, message_length, 0, (struct sockaddr *) &this->client_addr,
address_size);
00312     this->global_counter++;
00313
00314     if (!terminate) {
00315         if (!waiting_for_confirm(buf, message_length))
00316             std::cout << "Client didn't confirm" << std::endl;
00317     }
00318     this->client_addr = backup;
00319 }
00320
00321 int UDPHandler::create_message(uint8_t *buf_out, std::string &msg, bool error, std::string &name) {
00322     MsgPacket message(error ? 0xFE : 0x04, this->global_counter, msg, name);
00323     return message.construct_message(buf_out);
00324 }
00325
00326 int UDPHandler::create_bye(uint8_t *buf) {
00327     Packet bye(0xFF, this->global_counter);
00328     return bye.construct_message(buf);
00329 }
00330
00331 int UDPHandler::read_packet_id(uint8_t *buf) {
00332     int result = buf[1] << 8 | buf[2];
00333     return ntohs(result);
00334 }
00335
00336 int UDPHandler::wait_for_the_incoming_connection(uint8_t *buf_out, int timeout) {
00337     int event_count = epoll_wait(this->epoll_fd, this->events, 2, timeout);
00338
00339     if (event_count == -1) {
00340         perror("epoll_wait");
00341         close(this->epoll_fd);
00342         exit(EXIT_FAILURE);
00343     } else if (event_count > 0) {
00344         socklen_t len_client = sizeof(this->client_addr);
00345         for (int j = 0; j < event_count; j++) {
00346             if (events[j].data.fd == this->client_socket) { // check if EPOLLIN event has occurred
00347                 int n = recvfrom(this->client_socket, buf_out, 1024, 0, (struct sockaddr *)
&this->client_addr,
00348                                &len_client);
00349                 if (n == -1) {
00350                     std::cerr << "recvfrom failed. errno: " << errno << '\n';
00351                     continue;
00352                 }
00353                 if (n > 0) {
00354                     return n;
00355                 }

```

```

00356         } else {
00357             return -1;
00358         }
00359     }
00360 }
00361 return 0;
00362 }
00363
00364 bool UDPhandler::waiting_for_confirm(uint8_t *buf, int len) {
00365     uint8_t buffer[1024];
00366     bool confirmed = false;
00367     for (int i = 0; i < this->retransmissions; ++i) {
00368         int result = this->wait_for_the_incoming_connection(buffer, this->timeout_chat);
00369         if (result > 0) {
00370             if (buffer[0] == 0x00 && read_packet_id(buffer) == read_packet_id(buf)) {
00371                 confirmed = true;
00372             }
00373         } else if (result == -1) {
00374             return true;
00375         }
00376         if (confirmed) {
00377             break;
00378         } else {
00379             socklen_t len_client = sizeof(client_addr);
00380             long bytes_tx = sendto(this->client_socket, buf, len, 0, (struct sockaddr *)
00381 &(this->client_addr),
00382                                 len_client);
00383             if (bytes_tx < 0) perror("ERROR: sendto");
00384         }
00385     }
00386     return confirmed;
00387 }
00388 bool UDPhandler::buffer_validation(uint8_t *buf, int message_length, int start_position, int
00389 minimal_length,
00390                                   int amount_of_fields, int first_limit, int second_limit, int
00391 third_limit) {
00392     if (message_length < minimal_length)
00393         return false;
00394     size_t i = start_position;
00395
00396     size_t count = 0;
00397     while (i < message_length && buf[i] != 0x00 && count < first_limit) {
00398         ++i;
00399         count++;
00400     }
00401
00402     if (i >= message_length || buf[i] != 0x00 || count < 1) {
00403         return false;
00404     }
00405     ++i;
00406
00407     count = 0;
00408     while (i < message_length && buf[i] != 0x00 && count < second_limit) {
00409         ++i;
00410         ++count;
00411     }
00412
00413     if (i >= message_length || buf[i] != 0x00 || count < 1) {
00414         return false;
00415     }
00416
00417     ++i;
00418
00419     if (amount_of_fields == 3) {
00420         count = 0;
00421         while (i < message_length && buf[i] != 0x00 && count < third_limit) {
00422             ++i;
00423             ++count;
00424         }
00425
00426         if (i >= message_length || buf[i] != 0x00 || count < third_limit) {
00427             return false;
00428         }
00429     }
00430
00431     return true;
00432 }
00433
00434 std::string UDPhandler::read_channel_name(uint8_t *buf) {
00435     int i = 3;
00436     std::string channel;
00437     while (buf[i] != 0x00) {
00438         channel.push_back(static_cast<char>(buf[i]));
00439         ++i;

```

```

00440     }
00441     return channel;
00442 }
00443
00444 void UDPHandler::change_display_name(uint8_t *buf, bool second) {
00445     this->display_name.clear();
00446     int i = 3;
00447     if (second) {
00448         while (buf[i] != 0x00)
00449             i++;
00450     }
00451     i++;
00452     while (buf[i] != 0x00) {
00453         this->display_name.push_back(static_cast<char>(buf[i]));
00454         i++;
00455     }
00456 }
00457
00458 int UDPHandler::convert_from_tcp(uint8_t *buf, uint8_t *tcp_buf) {
00459     std::string message;
00460     int i = 0;
00461     while (tcp_buf[i] != 0x0d) {
00462         message.push_back(static_cast<char>(tcp_buf[i]));
00463         i++;
00464     }
00465
00466     std::regex patternFromToIs(R"(FROM\s(.*)\sIS)");
00467     std::smatch matchFromToIs;
00468     std::regex_search(message, matchFromToIs, patternFromToIs);
00469     std::string name = matchFromToIs[1].str();
00470
00471     std::regex patternAfterIs(R"(IS\s(.*))");
00472     std::smatch matchAfterIs;
00473     std::regex_search(message, matchAfterIs, patternAfterIs);
00474     std::string msg = matchAfterIs[1].str();
00475
00476     return this->create_message(buf, msg, false, name);
00477 }
00478
00479 void logger(sockaddr_in client, const char *type, const char *operation) {
00480     std::cout << operation << " " << inet_ntoa(client.sin_addr) << ":" << ntohs(client.sin_port) << " | " <<
00481         type
00482         << std::endl;
00483 }
00484
00485 bool UDPHandler::username_exists(uint8_t *buf, synch *synch_vars) {
00486     std::string username;
00487     int i = 3;
00488     while (buf[i] != 0x00) {
00489         username.push_back(static_cast<char>(buf[i]));
00490         i++;
00491     }
00492
00493     if (!synch_vars->usernames.empty()) {
00494         if (synch_vars->usernames.find(username) != synch_vars->usernames.end())
00495             return true;
00496     }
00497     synch_vars->usernames.insert(username);
00498     this->user_n = username;
00499     return false;
00500 }

```

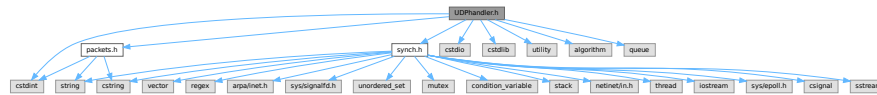
7.25 UDPhandler.h File Reference

```

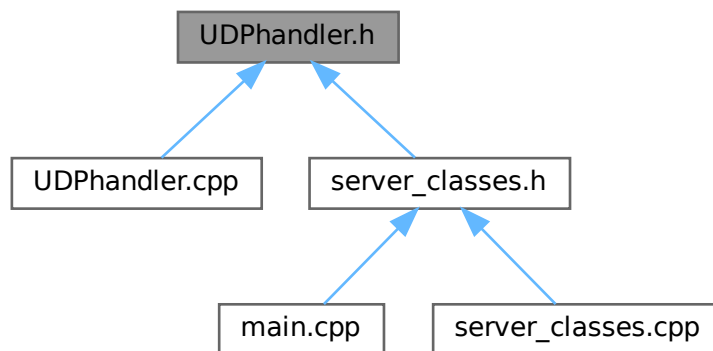
#include <cstdint>
#include <cstdio>
#include <cstdlib>
#include "packets.h"
#include <utility>
#include <algorithm>
#include <queue>
#include "synch.h"

```

Include dependency graph for UDPhandler.h:



This graph shows which files directly or indirectly include this file:



Classes

- class [UDPhandler](#)

Functions

- void [read_queue](#) (std::stack< [UserInfo](#) > *s, bool *terminate, [synch](#) *synch_vars, int *busy, [UDPhandler](#) *udp)
- void [logger](#) (sockaddr_in client, const char *type, const char *operation)

7.25.1 Function Documentation

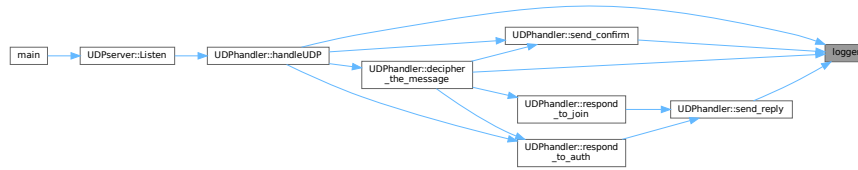
7.25.1.1 logger()

```
void logger (
    sockaddr_in client,
    const char * type,
    const char * operation )
```

Definition at line 480 of file [UDPhandler.cpp](#).

```
00480                                     {
00481     std::cout << operation << " " << inet_ntoa(client.sin_addr) << ":" << ntohs(client.sin_port) << " | " <<
type
00482         << std::endl;
00483 }
```

Here is the caller graph for this function:



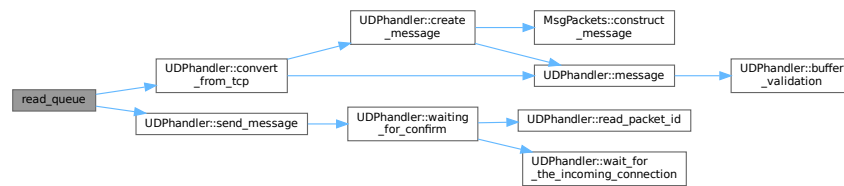
7.25.1.2 read_queue()

```
void read_queue (
    std::stack< UserInfo > * s,
    bool * terminate,
    synch * synch_vars,
    int * busy,
    UDPhandler * udp )
```

Definition at line 49 of file UDPhandler.cpp.

```
00049 {
00050     while (!*terminate) {
00051         std::unique_lock<std::mutex> lock(synch_vars->mtx);
00052         synch_vars->cv.wait(lock, [&synch_vars] { return synch_vars->ready; });
00053
00054         synch_vars->waiting.lock();
00055         synch_vars->finished++;
00056         synch_vars->waiting.unlock();
00057
00058         if (!s->empty() && udp->auth) {
00059             synch_vars->waiting.lock();
00060             UserInfo new_uf = s->top();
00061             synch_vars->waiting.unlock();
00062
00063             if (new_uf.tcp) {
00064                 if (new_uf.channel == udp->channel_name) {
00065                     uint8_t buf[3048];
00066                     int length = udp->convert_from_tcp(buf, new_uf.buf);
00067                     udp->send_message(buf, length, false);
00068                 }
00069             } else {
00070                 if ((new_uf.client.sin_port != udp->client_addr.sin_port && new_uf.channel ==
00071 udp->channel_name)) {
00072                     udp->send_message(new_uf.buf, new_uf.length, false);
00073                 }
00074             }
00075         }
00076         if (synch_vars->finished == *busy) {
00077             synch_vars->finished = 0;
00078             synch_vars->ready = false;
00079             if (!s->empty())
00080                 s->pop();
00081         }
00082         lock.unlock();
00083         std::this_thread::sleep_for(std::chrono::milliseconds(100));
00084         lock.lock();
00085     }
00086 }
00087 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



7.26 UDPhandler.h

[Go to the documentation of this file.](#)

```

00001 //
00002 // Created by artem on 4/14/24.
00003 //
00004
00005 #ifndef IPK_SERVER_UDPHANDLER_H
00006 #define IPK_SERVER_UDPHANDLER_H
00007
00008 #include <cstdint>
00009
00010 #include <cstdio>
00011 #include <cstdlib>
00012
00013 #include "packets.h"
00014
00015 #include <utility>
00016
00017 #include <algorithm>
00018
00019 #include <queue>
00020 #include "synch.h"
00021
00022
00023 class UDPhandler {
00024 public:
00025     int retransmissions;
00026     int timeout_chat;
00027     int global_counter;
00028     int client_socket;
00029     std::vector<int> vec;
00030     epoll_event events[2];
00031     int epoll_fd;
00032     bool auth;
00033     sockaddr_in client_addr;
00034     std::string display_name;
00035     std::string channel_name;
00036     std::string user_n;
00037
00038     UDPhandler(int ret, int t, sockaddr_in client, int kill) {
00039         this->retransmissions = ret;
00040         this->timeout_chat = t;
00041         this->global_counter = 0;
00042         this->client_socket = socket(AF_INET, SOCK_DGRAM, 0);
00043         if (this->client_socket < 0) {
00044             perror("Problem with creating response socket");
00045             exit(EXIT_FAILURE);
00046         }
00047
00048         epoll_fd = epoll_create1(0);
00049         if (epoll_fd == -1) {

```

```

00051         std::cerr << "Failed to create epoll file descriptor\n";
00052         exit(EXIT_FAILURE);
00053     }
00054
00055     // setup epoll event
00056     struct epoll_event ev;
00057     ev.events = EPOLLIN | EPOLLET;
00058     ev.data.fd = this->client_socket;
00059
00060     // add socket file descriptor to epoll
00061     if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, this->client_socket, &ev) == -1) {
00062         std::cerr << "Failed to add file descriptor to epoll\n";
00063         close(epoll_fd);
00064         exit(EXIT_FAILURE);
00065     }
00066
00067     ev.data.fd = kill;
00068     if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, kill, &ev) < 0) {
00069         std::cerr << "Unable to add socket to epoll\n";
00070         exit(EXIT_FAILURE);
00071     }
00072
00073     auth = false;
00074
00075     client_addr = client;
00076
00077     channel_name = "general";
00078
00079 }
00080
00081 static void
00082 handleUDP(uint8_t *buf, sockaddr_in client_addr, int length, int retransmissions, int timeout, int
*busy,
00083         std::stack<UserInfo> *s, synch *synch_var, int signal_listener);
00084
00085 int create_message(uint8_t *buf_out, std::string &msg, bool error, std::string &name);
00086
00087 void send_message(uint8_t *buf, int message_length, bool terminate);
00088
00089 int convert_from_tcp(uint8_t *buf, uint8_t *tcp_buf);
00090
00091 private:
00092     bool decipher_the_message(uint8_t *buf, int length, std::stack<UserInfo> *s, synch *synch_var);
00093
00094     int respond_to_auth(uint8_t *buf, int length, std::stack<UserInfo> *s, synch *synch_var);
00095
00096     void respond_to_join(uint8_t *buf, int length, std::stack<UserInfo> *s, synch *synch_var);
00097
00098     void send_confirm(uint8_t *buf);
00099
00100     void send_reply(uint8_t *buf, std::string &message, bool OK);
00101
00102     static int read_packet_id(uint8_t *buf);
00103
00104     int wait_for_the_incoming_connection(uint8_t *buf_out, int timeout = -1);
00105
00106     bool waiting_for_confirm(uint8_t *buf, int len);
00107
00108     void message(uint8_t *buf, int message_length, std::stack<UserInfo> *s, synch *synch_var,
std::string &channel);
00109
00110     bool buffer_validation(uint8_t *buf, int message_length, int start_position, int minimal_length,
int amount_of_fields = 2, int first_limit = 20, int second_limit = 20, int
third_limit = 5);
00111
00112     void change_display_name(uint8_t *buf, bool second);
00113
00114     void client_leaving(std::stack<UserInfo> *s, synch *synch_var);
00115
00116     std::string read_channel_name(uint8_t *buf);
00117
00118     int create_bye(uint8_t *buf);
00119
00120     bool username_exists(uint8_t *buf, synch *synch_vars);
00121
00122 };
00123
00124 void read_queue(std::stack<UserInfo> *s, bool *terminate, synch *synch_vars, int *busy, UDPHandler
*udp);
00125
00126 void logger(sockaddr_in client, const char *type, const char *operation);
00127
00128 #endif //IPK_SERVER_UDPHANDLER_H
00129
00130
00131
00132
00133

```


Index

- ~ThreadPool
 - ThreadPool, [49](#)
- address
 - ArgumentsHandler, [14](#)
- AddTask
 - ThreadPool, [49](#)
- ArgumentsHandler, [11](#)
 - address, [14](#)
 - get_address, [12](#)
 - get_args, [12](#)
 - get_port, [13](#)
 - get_retransmissions, [13](#)
 - get_threads, [13](#)
 - get_timeout, [13](#)
 - number_of_threads, [14](#)
 - port, [14](#)
 - print_help, [13](#)
 - retransmissions, [14](#)
 - timeout, [14](#)
- ArgumentsHandler.cpp, [79](#)
- ArgumentsHandler.h, [81](#)
- auth
 - TCPHandler, [43](#)
 - UDPhandler, [72](#)
- AuthPacket
 - packets.h, [86](#)
- AuthPackets, [14](#)
 - AuthPackets, [16](#)
 - construct_message, [16](#)
 - DisplayName, [17](#)
 - Secret, [17](#)
 - Username, [17](#)
- buf
 - UserInfo, [78](#)
- buffer_validation
 - UDPhandler, [56](#)
- busy_threads
 - ThreadPool, [51](#)
- CHAGELOG.md, [82](#)
- change_display_name
 - UDPhandler, [57](#)
- channel
 - UserInfo, [78](#)
- channel_name
 - TCPHandler, [43](#)
 - UDPhandler, [72](#)
- ChannelID
 - JoinPackets, [21](#)
- client
 - UserInfo, [78](#)
- client_addr
 - TCPHandler, [43](#)
 - UDPhandler, [72](#)
- client_leaving
 - UDPhandler, [57](#)
- client_socket
 - TCPHandler, [43](#)
 - UDPhandler, [72](#)
- condition_variable
 - ThreadPool, [51](#)
- ConfirmPacket
 - packets.h, [86](#)
- ConfirmPackets, [17](#)
 - ConfirmPackets, [18](#)
 - construct_message, [19](#)
 - Ref_MessageID, [19](#)
- construct_message
 - AuthPackets, [16](#)
 - ConfirmPackets, [19](#)
 - JoinPackets, [21](#)
 - MsgPackets, [24](#)
 - Packets, [26](#)
 - ReplyPackets, [28](#)
- convert_from_tcp
 - UDPhandler, [58](#)
- convert_from_udp
 - TCPHandler, [34](#)
- create_bye
 - TCPHandler, [35](#)
 - UDPhandler, [59](#)
- create_message
 - TCPHandler, [36](#)
 - UDPhandler, [59](#)
- create_reply
 - TCPHandler, [36](#)
- cv
 - synch, [31](#)
- cv2
 - synch, [31](#)
- decipher_the_message
 - TCPHandler, [37](#)
 - UDPhandler, [60](#)
- Destroy
 - TCPserver, [45](#)
 - UDPserver, [75](#)
- display_name

- TCPHandler, [44](#)
 - UDPHandler, [72](#)
- DisplayName
 - AuthPackets, [17](#)
 - JoinPackets, [21](#)
 - MsgPackets, [24](#)
- epoll_fd
 - TCPHandler, [44](#)
 - UDPHandler, [72](#)
- events
 - TCPHandler, [44](#)
 - UDPHandler, [72](#)
- finished
 - synch, [32](#)
- Functionality, [1](#)
- get_address
 - ArgumentsHandler, [12](#)
- get_args
 - ArgumentsHandler, [12](#)
- get_port
 - ArgumentsHandler, [13](#)
- get_retransmissions
 - ArgumentsHandler, [13](#)
- get_threads
 - ArgumentsHandler, [13](#)
- get_timeout
 - ArgumentsHandler, [13](#)
- global_counter
 - UDPHandler, [72](#)
- handle_sigint
 - main.cpp, [82](#)
- handleTCP
 - TCPHandler, [39](#)
- handleUDP
 - UDPHandler, [61](#)
- init
 - main.cpp, [83](#)
- Initialize
 - Server, [30](#)
 - TCPserver, [45](#)
 - UDPserver, [75](#)
- IPK - first project - Client-chat, [3](#)
- JoinPacket
 - packets.h, [86](#)
- JoinPackets, [19](#)
 - ChannelID, [21](#)
 - construct_message, [21](#)
 - DisplayName, [21](#)
 - JoinPackets, [21](#)
- length
 - UserInfo, [78](#)
- Listen
 - Server, [30](#)
- TCPserver, [46](#)
 - UDPserver, [75](#)
- listening_for_incoming_connection
 - TCPHandler, [40](#)
- logger
 - UDPHandler.cpp, [107](#)
 - UDPHandler.h, [115](#)
- main
 - main.cpp, [83](#)
- main.cpp, [82](#)
 - handle_sigint, [82](#)
 - init, [83](#)
 - main, [83](#)
 - pipefd, [84](#)
- Message
 - ReplyPackets, [29](#)
- message
 - TCPHandler, [41](#)
 - UDPHandler, [63](#)
- MessageContents
 - MsgPackets, [24](#)
- MessageID
 - Packets, [26](#)
- MessageType
 - Packets, [26](#)
- MsgPacket
 - packets.h, [86](#)
- MsgPackets, [22](#)
 - construct_message, [24](#)
 - DisplayName, [24](#)
 - MessageContents, [24](#)
 - MsgPackets, [23](#)
- mtx
 - synch, [32](#)
- mutex
 - ThreadPool, [51](#)
- number_of_threads
 - ArgumentsHandler, [14](#)
- operator()
 - ThreadPool::ThreadWorker, [53](#)
- operator=
 - ThreadPool, [50](#)
- Packet
 - packets.h, [87](#)
- Packets, [24](#)
 - construct_message, [26](#)
 - MessageID, [26](#)
 - MessageType, [26](#)
 - Packets, [25](#)
- packets.h, [85](#)
 - AuthPacket, [86](#)
 - ConfirmPacket, [86](#)
 - JoinPacket, [86](#)
 - MsgPacket, [86](#)
 - Packet, [87](#)

- ReplyPacket, 87
- pipefd
 - main.cpp, 84
- PORT
 - server_classes.h, 92
- port
 - ArgumentsHandler, 14
- print_help
 - ArgumentsHandler, 13
- queue
 - ThreadPool, 51
- QueueSize
 - ThreadPool, 50
- read_channel_name
 - UDPhandler, 64
- read_packet_id
 - UDPhandler, 64
- read_queue
 - TCPHandler.cpp, 95
 - TCPHandler.h, 101
 - UDPhandler.cpp, 107
 - UDPhandler.h, 116
- README.md, 89
- ready
 - synch, 32
- ref_id
 - ReplyPackets, 29
- Ref_MessageID
 - ConfirmPackets, 19
- ReplyPacket
 - packets.h, 87
- ReplyPackets, 26
 - construct_message, 28
 - Message, 29
 - ref_id, 29
 - ReplyPackets, 28
 - result, 29
- respond_to_auth
 - UDPhandler, 65
- respond_to_join
 - UDPhandler, 66
- result
 - ReplyPackets, 29
- retransmissions
 - ArgumentsHandler, 14
 - UDPhandler, 72
 - UDPserver, 76
- Secret
 - AuthPackets, 17
- send_buf
 - TCPHandler, 41
- send_confirm
 - UDPhandler, 67
- send_message
 - UDPhandler, 68
- send_reply
 - UDPhandler, 69
- send_string
 - TCPHandler, 41
- Server, 29
 - Initialize, 30
 - Listen, 30
- server_classes.cpp, 89
- server_classes.h, 91
 - PORT, 92
- Shutdown
 - ThreadPool, 50
- shutdown_requested
 - ThreadPool, 51
- sockFD
 - TCPserver, 47
 - UDPserver, 76
- synch, 30
 - cv, 31
 - cv2, 31
 - finished, 32
 - mtx, 32
 - ready, 32
 - synch, 31
 - un, 32
 - usernames, 32
 - waiting, 32
- synch.h, 93
- tcp
 - UserInfo, 78
- tcp_logger
 - TCPHandler.cpp, 96
 - TCPHandler.h, 102
- tcp_socket
 - UserInfo, 78
- TCPHandler, 32
 - auth, 43
 - channel_name, 43
 - client_addr, 43
 - client_socket, 43
 - convert_from_udp, 34
 - create_bye, 35
 - create_message, 36
 - create_reply, 36
 - decipher_the_message, 37
 - display_name, 44
 - epoll_fd, 44
 - events, 44
 - handleTCP, 39
 - listening_for_incoming_connection, 40
 - message, 41
 - send_buf, 41
 - send_string, 41
 - TCPHandler, 34
 - user_changed_channel, 42
 - user_n, 44
 - username_already_exists, 43
- TCPHandler.cpp, 95
 - read_queue, 95

- tcp_logger, 96
- TCPHandler.h, 101
 - read_queue, 101
 - tcp_logger, 102
- TCPserver, 44
 - Destroy, 45
 - Initialize, 45
 - Listen, 46
 - sockFD, 47
 - TCPserver, 45
- thread_pool
 - ThreadPool::ThreadWorker, 53
- thread_pool.h, 104
- ThreadPool, 47
 - ~ThreadPool, 49
 - AddTask, 49
 - busy_threads, 51
 - condition_variable, 51
 - mutex, 51
 - operator=, 50
 - queue, 51
 - QueueSize, 50
 - Shutdown, 50
 - shutdown_requested, 51
 - ThreadPool, 49
 - threads, 51
- ThreadPool::ThreadWorker, 52
 - operator(), 53
 - thread_pool, 53
 - ThreadWorker, 53
- threads
 - ThreadPool, 51
- ThreadWorker
 - ThreadPool::ThreadWorker, 53
- timeout
 - ArgumentsHandler, 14
 - UDPserver, 77
- timeout_chat
 - UDPHandler, 72
- UDPHandler, 53
 - auth, 72
 - buffer_validation, 56
 - change_display_name, 57
 - channel_name, 72
 - client_addr, 72
 - client_leaving, 57
 - client_socket, 72
 - convert_from_tcp, 58
 - create_bye, 59
 - create_message, 59
 - decipher_the_message, 60
 - display_name, 72
 - epoll_fd, 72
 - events, 72
 - global_counter, 72
 - handleUDP, 61
 - message, 63
 - read_channel_name, 64
 - read_packet_id, 64
 - respond_to_auth, 65
 - respond_to_join, 66
 - retransmissions, 72
 - send_confirm, 67
 - send_message, 68
 - send_reply, 69
 - timeout_chat, 72
 - UDPHandler, 55
 - user_n, 72
 - username_exists, 69
 - vec, 72
 - wait_for_the_incoming_connection, 70
 - waiting_for_confirm, 71
- UDPHandler.cpp, 107
 - logger, 107
 - read_queue, 107
- UDPHandler.h, 114
 - logger, 115
 - read_queue, 116
- UDPserver, 73
 - Destroy, 75
 - Initialize, 75
 - Listen, 75
 - retransmissions, 76
 - sockFD, 76
 - timeout, 77
 - UDPserver, 74
- un
 - synch, 32
- user_changed_channel
 - TCPHandler, 42
- user_n
 - TCPHandler, 44
 - UDPHandler, 72
- UserInfo, 77
 - buf, 78
 - channel, 78
 - client, 78
 - length, 78
 - tcp, 78
 - tcp_socket, 78
 - UserInfo, 78
- Username
 - AuthPackets, 17
- username_already_exists
 - TCPHandler, 43
- username_exists
 - UDPHandler, 69
- usernames
 - synch, 32
- vec
 - UDPHandler, 72
- wait_for_the_incoming_connection
 - UDPHandler, 70
- waiting
 - synch, 32

waiting_for_confirm
 UDPHandler, [71](#)