My Project

Generated by Doxygen 1.10.0

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:

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:

Chapter 4

Class Index

4.1 Class List

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

Chapter 5

File Index

5.1 File List

Here is a list of all files with brief descriptions:

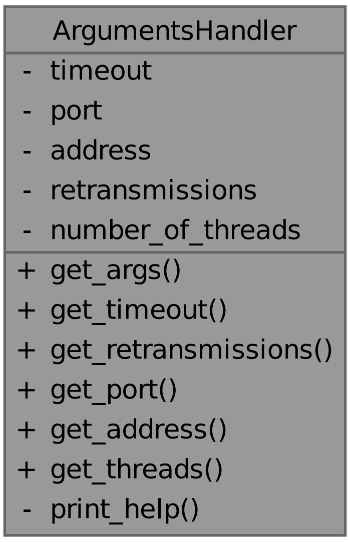
Chapter 6

Class Documentation

6.1 ArgumentsHandler Class Reference

`#include <ArgumentsHandler.h>`

Collaboration diagram for ArgumentsHandler:



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()

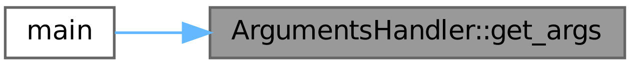
```
void ArgumentsHandler::get_args (
              int argc,
              char * argv[] )
```

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             }
```

```
        } else if (arg == "-r") {
            i++;
            if (i < argc) {
                try {
                    this->retransmissions = std::stoi(argv[i]);
                } catch (std::invalid_argument &) {
                    std::cout « "Passed non-int value to retransmissions" « std::endl;
                    exit(1);
                }
            } else {
                std::cout « "Nothing passed to retransmissions" « std::endl;
                exit(1);
            }
        } else if (arg == "-n"){
            i++;
            if (i < argc) {
                try {
                    this->number_of_threads = std::stoi(argv[i]);
                } catch (std::invalid_argument &) {
                    std::cout « "Passed non-int value to number of threads" « std::endl;
                    exit(1);
                }
            } 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.

```
                                  {
    return this->timeout;
}
```

6.1.2.6 get_timeout()

`int ArgumentsHandler::get_timeout ( )`

Definition at line 100 of file ArgumentsHandler.cpp.

```
                                      {
    return this->port;
}
```

6.1.2.7 print_help()

`void ArgumentsHandler::print_help ( ) [static], [private]`

Definition at line 8 of file ArgumentsHandler.cpp.

```
                                  {
    std::cout « R"(|Argument | Default values | Type

```

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.

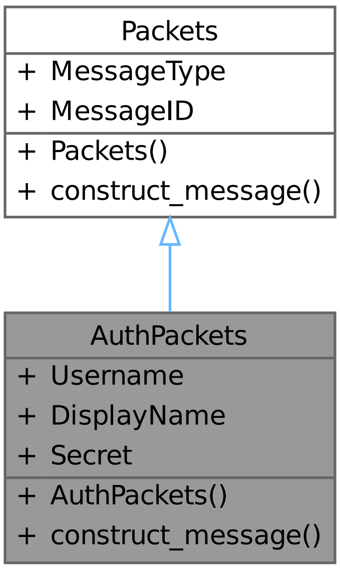
The documentation for this class was generated from the following files:

- ArgumentsHandler.h
- ArgumentsHandler.cpp

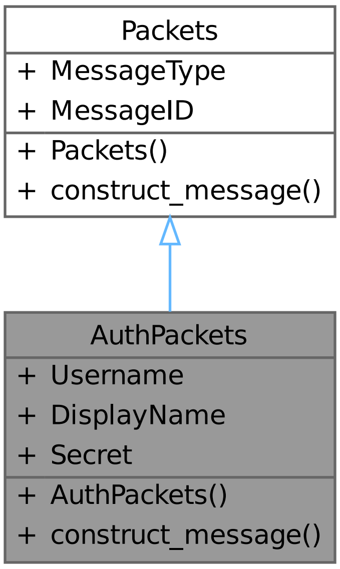
6.2 AuthPackets Struct Reference

`#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.

```
                                                                                              :
      Packets(type,
      id) {
            Username = std::move(u_n);
            DisplayName = std::move(disp_name);
            Secret = std::move(sec);
     }
```

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.

```
                                                  {
            memcpy(b, &this->MessageType, sizeof(this->MessageType));
            b += sizeof(this->MessageType);
            //std::cout«this->MessageType«std::endl;

            //uint16_t ID = htons(this->MessageID);
            uint16_t ID = this->MessageID;
            memcpy(b, &ID, sizeof(ID));
            b += sizeof(ID);

            memcpy(b, Username.c_str(), Username.length());
            b[Username.length()] = '\0';
            b += Username.length() + 1;

            memcpy(b, DisplayName.c_str(), DisplayName.length());
            b[DisplayName.length()] = '\0';
            b += DisplayName.length() + 1;

            memcpy(b, Secret.c_str(), Secret.length());
```

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

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.

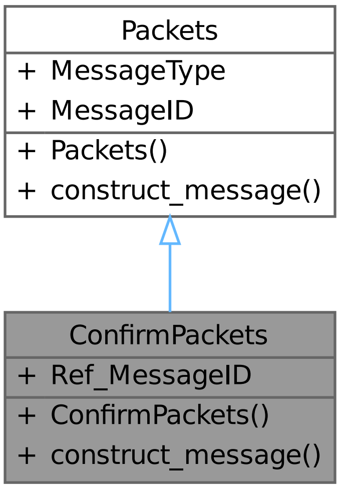
6.2.4.3 Username

`std::string AuthPackets::Username`
Definition at line 124 of file packets.h.
The documentation for this struct was generated from the following file:

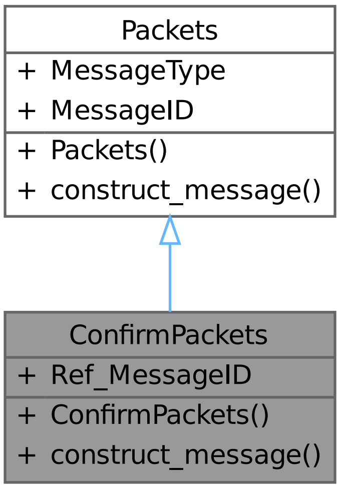
- packets.h

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.

```
                                                              : Packets(type, id) {
        Ref_MessageID = ref_id;
    }
```

6.3.3 Member Function Documentation

6.3.3.1 construct_message()

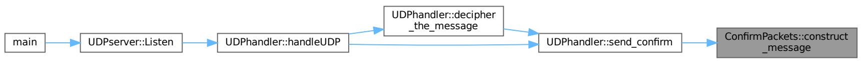
```
int ConfirmPackets::construct_message (
            uint8_t * b )  [inline], [override], [virtual]
```

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.

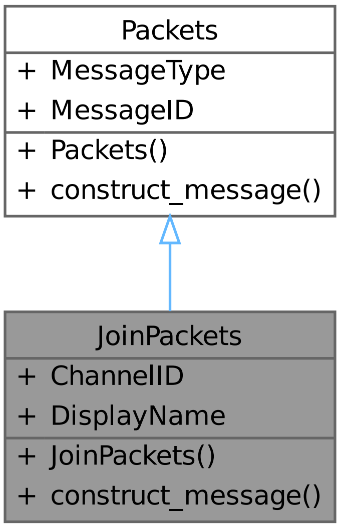
The documentation for this struct was generated from the following file:

- packets.h

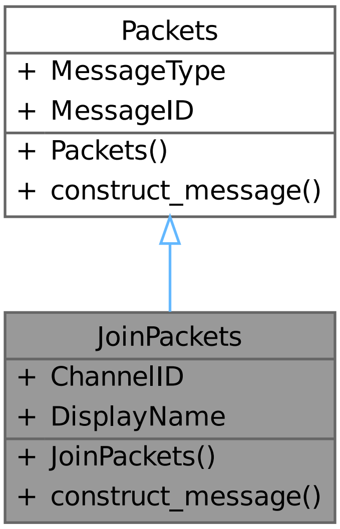
6.4 JoinPackets Struct Reference

```
#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.

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

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.

```
                                            {
        memcpy(b, &this->MessageType, sizeof(this->MessageType));
        b += sizeof(this->MessageType);

        //uint16_t ID = htons(this->MessageID);
        uint16_t ID = this->MessageID;
        memcpy(b, &ID, sizeof(ID));
        b += sizeof(ID);

        memcpy(b, ChannelID.c_str(), ChannelID.length());
        b[ChannelID.length()] = '\0';
        b += ChannelID.length() + 1;

        memcpy(b, DisplayName.c_str(), DisplayName.length());
        b[DisplayName.length()] = '\0';
        b += DisplayName.length() + 1;
        return sizeof(this->MessageType) + sizeof(ID) + ChannelID.length() + 1 + DisplayName.length()
      + 1;
    }
```

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.

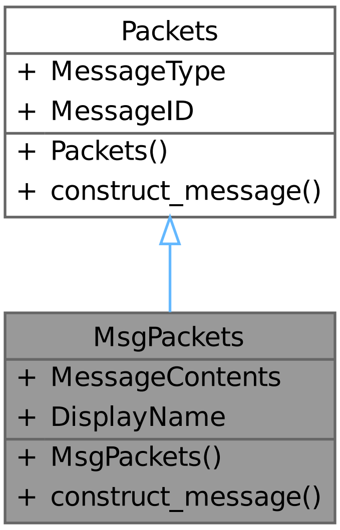
The documentation for this struct was generated from the following file:

- packets.h

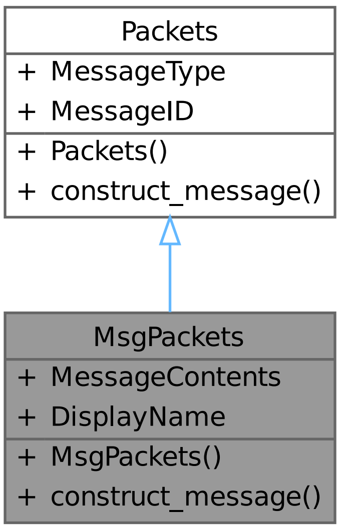
6.5 MsgPackets Struct Reference

```
#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         MessageContents = std::move(content);
00098         DisplayName = std::move(disp_name);
00099     }
```

6.5.3 Member Function Documentation

6.5.3.1 construct_message()

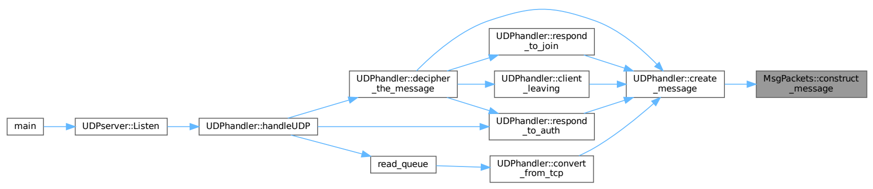
```
int MsgPackets::construct_message (
              uint8_t * b )  [inline], [override], [virtual]
```

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 +
     MessageContents.length() + 1;
00118     }
```

Here is the caller graph for this function:


main
UDPserver::Listen
UDPhandler::handleUDP
UDPhandler::decipher_the_message
read_queue
UDPhandler::respond_to_join
UDPhandler::client_leaving
UDPhandler::respond_to_auth
UDPhandler::convert_from_tcp
UDPhandler::create_message
MsgPackets::construct_message

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.

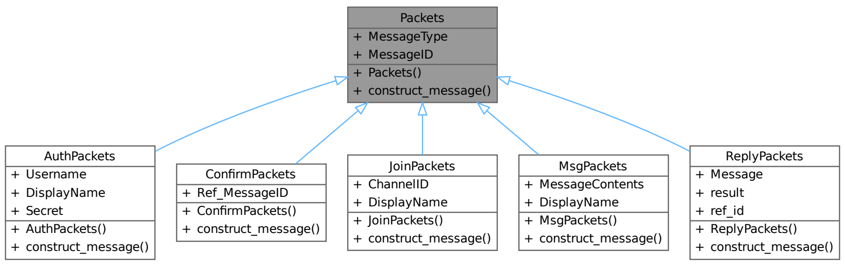
The documentation for this struct was generated from the following file:

- packets.h

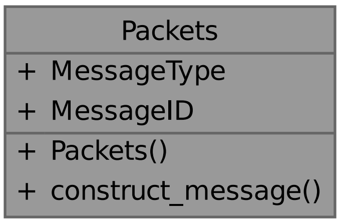
6.6 Packets Struct Reference

```
#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.

```
                                    {
        MessageType = type;
        MessageID = id;
    }
```

6.6.3 Member Function Documentation

6.6.3.1 construct_message()

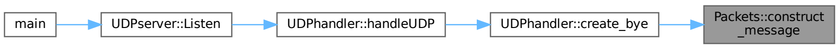
```
virtual int Packets::construct_message (
             uint8_t * b )  [inline], [virtual]
```

Reimplemented in ConfirmPackets, JoinPackets, MsgPackets, AuthPackets, and ReplyPackets.

Definition at line 25 of file packets.h.

```
                                            {
        memcpy(b, &this->MessageType, sizeof(this->MessageType));
        b += sizeof(this->MessageType);


        //uint16_t ID = htons(this->MessageID);
        uint16_t ID = this->MessageID;
        memcpy(b, &ID, sizeof(ID));
        b += sizeof(ID);
        return 3;
    }
```

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.

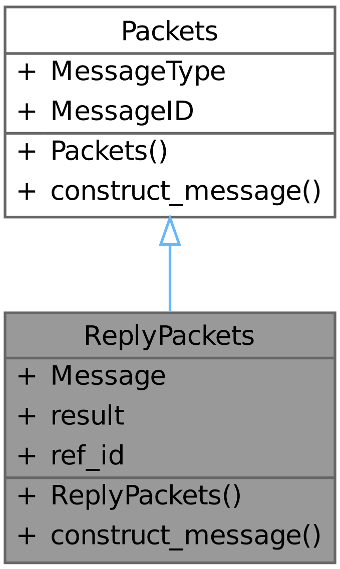
The documentation for this struct was generated from the following file:

- packets.h

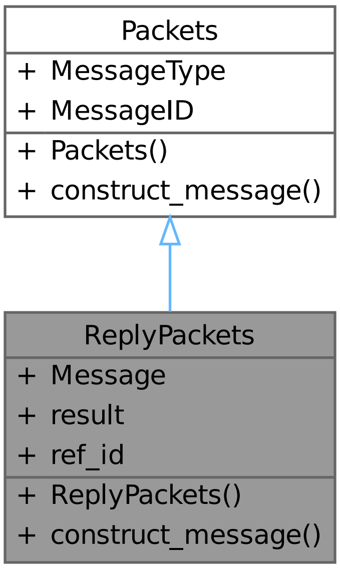
6.7 ReplyPackets Struct Reference

```
#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.

```
                                                                                  : Packets(type,
      id) {
            Message = std::move(mes);
            result = res;
            ref_id = ref;
        }
```

6.7.3 Member Function Documentation

6.7.3.1 construct_message()

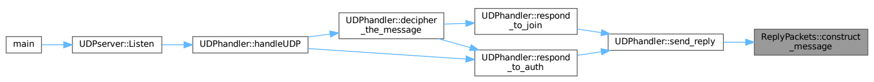
```
int ReplyPackets::construct_message (
              uint8_t * b )  [inline], [override], [virtual]
```

Reimplemented from Packets.

Definition at line 173 of file packets.h.

```
                                                   {
            memcpy(b, &this->MessageType, sizeof(this->MessageType));
            b += sizeof(this->MessageType);
            uint16_t ID = this->MessageID;
            memcpy(b, &ID, sizeof(ID));
            b += sizeof(ID);

            memcpy(b, &result, sizeof(result));
            b += sizeof(result);

            memcpy(b, &ref_id, sizeof(ref_id));
            b += sizeof(ref_id);

            memcpy(b, Message.c_str(), Message.length());
            b[Message.length()] = '\0';
            b += Message.length() + 1;

            return sizeof(this->MessageType) + sizeof(ID) + sizeof(result) + sizeof(ref_id) +
      Message.length() + 1;
        }
```

Here is the caller graph for this function:


main
UDPserver::Listen
UDPhandler::handleUDP
UDPhandler::decipher _the_message
UDPhandler::respond _to_join
UDPhandler::respond _to_auth
UDPhandler::send_reply
ReplyPackets::construct _message

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.

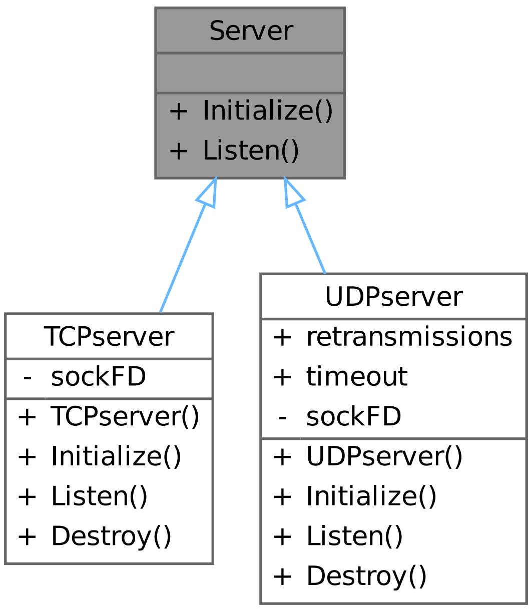
6.7.4.3 result

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

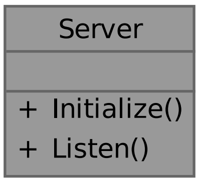
- packets.h

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.

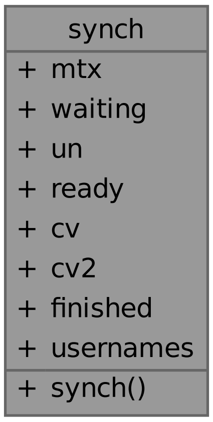
The documentation for this class was generated from the following file:

- server_classes.h

6.9 synch Struct Reference

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

Collaboration diagram for synch:



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.

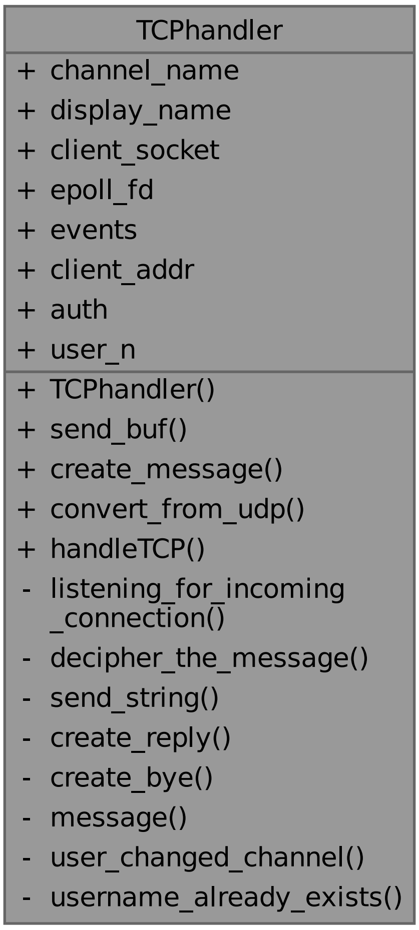
The documentation for this struct was generated from the following file:

- synch.h

6.10 TCPhandler Class Reference

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

Collaboration diagram for TCPhandler:



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.

```
                                            {
        this->channel_name = "general";
        this->client_socket = s;

        epoll_fd = epoll_create1(0);
        if (epoll_fd == -1) {
            std::cerr « "Failed to create epoll file descriptor\n";
            exit(EXIT_FAILURE);
        }

        // setup epoll event
        struct epoll_event ev;
        ev.events = EPOLLIN;
        ev.data.fd = this->client_socket;

        // add socket file descriptor to epoll
        if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, this->client_socket, &ev) == -1) {
            std::cerr « "Failed to add file descriptor to epoll\n";
            close(epoll_fd);
            exit(EXIT_FAILURE);
        }

        ev.data.fd = kill;
        if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, kill, &ev) < 0) {
            std::cerr « "Unable to add socket to epoll\n";
            exit(EXIT_FAILURE);
        }

        client_addr = c;

        auth = false;
    }
```

6.10.3 Member Function Documentation

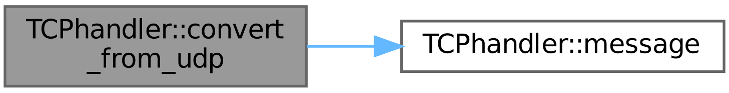
6.10.3.1 convert_from_udp()

```
int TCPhandler::convert_from_udp (
              uint8_t ∗ buf,
              uint8_t ∗ tcp_buf )
```

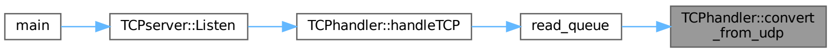
Definition at line 285 of file TCPhandler.cpp.

```
                                                                     {
    int i = 3;
    std::string display_n;
    std::string contents;

    while (udp_buf[i] != 0x00) {
        display_n.push_back(static_cast<char>(udp_buf[i]));
        i++;
    }

    i++;

    while (udp_buf[i] != 0x00) {
        contents.push_back(static_cast<char>(udp_buf[i]));
        i++;
    }

    std::string message = "MSG FROM " + display_n + " IS " + contents + "\r\n";

    memcpy(buf, message.c_str(), message.length());

    return message.length();
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



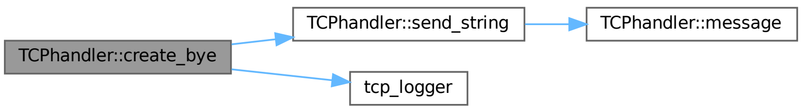
6.10.3.2 create_bye()

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

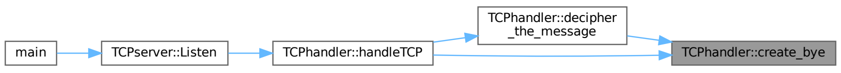
Definition at line 245 of file TCPhandler.cpp.

```
                            {
    std::string bye = "BYE\r\n";
    tcp_logger(this->client_addr, "BYE", "SENT");
    this->send_string(bye);
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



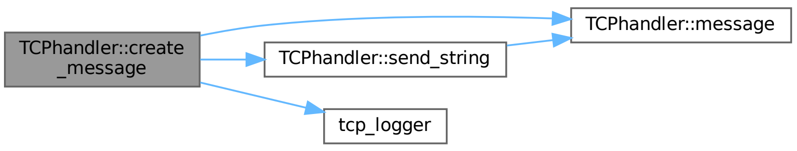
6.10.3.3 create_message()

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

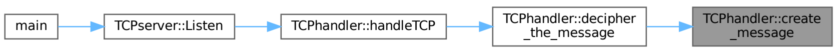
Definition at line 237 of file TCPhandler.cpp.

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

Here is the call graph for this function:



Here is the caller graph for this function:



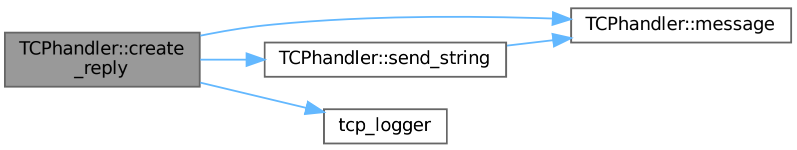
6.10.3.4 create_reply()

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

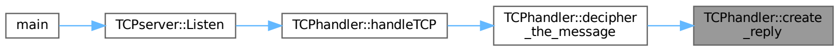
Definition at line 230 of file TCPhandler.cpp.

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

Here is the call graph for this function:



Here is the caller graph for this function:



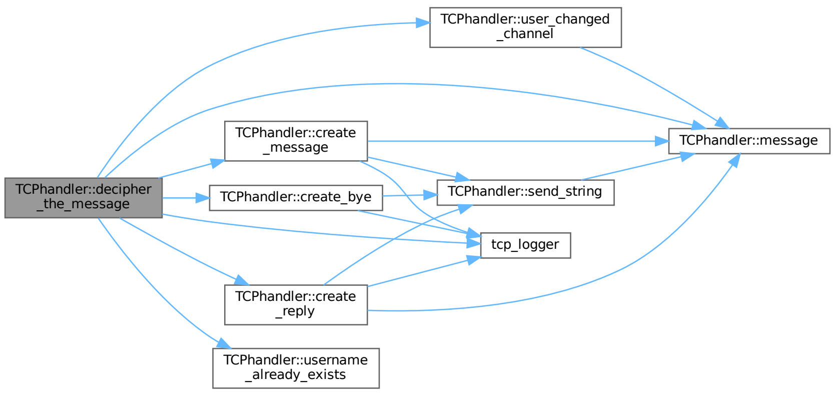
6.10.3.5 decipher_the_message()

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

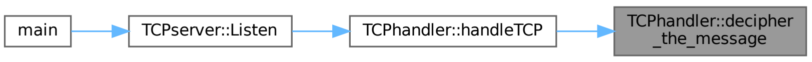
Definition at line 115 of file TCPhandler.cpp.

```
    {
    std::string out_str;
    for (int i = 0; i < length - 2; ++i) {
        out_str += static_cast<char>(buf[i]);
    }

    std::istringstream iss(out_str);
    std::vector<std::string> result;
    for (std::string element; std::getline(iss, element, ' ');) {
        result.push_back(element);
    }

    if (!this->auth) {
        if (result[0] != "AUTH") {
            if (result[0] != "BYE") {
                if (result[0] != "ERR") {
                    this->create_message(true, "You should log-in before doing anything else");
                    return true;
                }
            }
        }
    }

    if (result[0] == "AUTH") {
        std::regex e("^AUTH .{1,20} AS .{1,20} USING .{1,128}$");
        if (!std::regex_match(out_str, e)) {
            std::string mes = "Wrong AUTH format";
            std::cout « mes « std::endl;
            create_message(true, "Wrong AUTH format");
            std::this_thread::sleep_for(std::chrono::milliseconds(10));
            this->create_bye();
            return false;
        }
        synch_var->un.lock();
        bool exists = username_already_exists(result[1], synch_var);
        synch_var->un.unlock();
        tcp_logger(this->client_addr, "AUTH", "RECV");
        if (exists) {
            this->create_reply("NOK", "Username already exists");
        } else {
            this->create_reply("OK", "Authentication is successful");
            this->display_name = result[3];
            this->user_changed_channel(s, synch_var, "joined");
            this->auth = true;
        }

    } else if (result[0] == "MSG") {
        std::regex e("^MSG FROM .{1,20} IS .{1,1400}$");
        if (!std::regex_match(out_str, e)) {
            std::string mes = "Wrong MSG format";
            std::cout « mes « std::endl;
            create_message(true, "Wrong MSG format");
            std::this_thread::sleep_for(std::chrono::milliseconds(10));
            this->create_bye();
            return false;
        }
        tcp_logger(this->client_addr, "MSG", "RECV");
        this->display_name = result[2];
        this->message(buf, length, s, synch_var, this->channel_name);
    } else if (result[0] == "JOIN") {
        std::regex e("^JOIN .{1,20} AS .{1,20}$");
        if (!std::regex_match(out_str, e)) {
            std::string mes = "Wrong JOIN format";
            create_message(true, mes.c_str());
            create_message(true, "Wrong JOIN format");
            std::this_thread::sleep_for(std::chrono::milliseconds(10));
            this->create_bye();
            return false;
        }
        if (result[1] != this->channel_name) {
            tcp_logger(this->client_addr, "JOIN", "RECV");
            this->user_changed_channel(s, synch_var, "left");
            std::this_thread::sleep_for(std::chrono::milliseconds(30));
            this->channel_name = result[1];
            this->display_name = result[3];
            this->user_changed_channel(s, synch_var, "joined");
            this->create_reply("OK", "Join was successful");
        } else {
            this->create_reply("NOK", "Tried to join to the current channel");
        }
    } else if (result[0] == "BYE") {
        tcp_logger(this->client_addr, "BYE", "RECV");
        if (this->auth) {
            user_changed_channel(s, synch_var, "left");
        }
        return false;
    } else if (result[0] == "ERR") {
        tcp_logger(this->client_addr, "ERR", "RECV");
        if (this->auth) {
            user_changed_channel(s, synch_var, "left");
        }
        this->create_bye();
        return false;
    } else {
        tcp_logger(this->client_addr, "UNDEFINED", "RECV");
        this->create_message(true, "Unknown command");
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
        this->create_bye();
        return false;
    }

    return true;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.10.3.6 handleTCP()

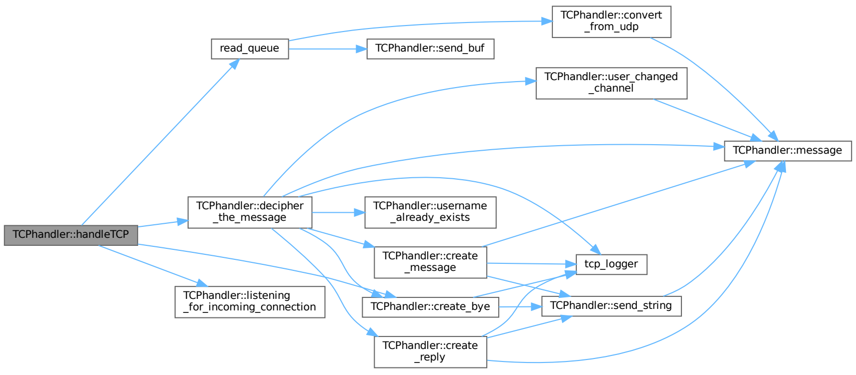
```
void TCPhandler::handleTCP (
               int client_socket,
               int * busy,
               std::stack< UserInfo > * s,
               synch * synch_var,
               sockaddr_in client,
               int signal_listener )  [static]
```

Definition at line 7 of file TCPhandler.cpp.

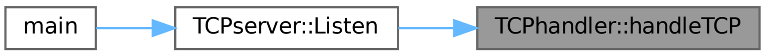
```
                                                {

    TCPhandler tcp(client_socket, client, signal_listener);

    bool end = false;

    std::thread sender(read_queue, s, &end, synch_var, busy, &tcp);

    uint8_t internal_buf[2048];

    while (true) {
        int length = tcp.listening_for_incoming_connection(internal_buf, 1024);
        if (length == 0)
            break;
        if (length == -1) {
            tcp.create_bye();
            break;
        }
        if (!tcp.decipher_the_message(internal_buf, length, s, synch_var))
            break;
    }

    if (synch_var->usernames.find(tcp.user_n) != synch_var->usernames.end())
        synch_var->usernames.erase(tcp.user_n);

    end = true;
    {
        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:


TCPhandler::handleTCP
read_queue
TCPhandler::send_buf
TCPhandler::convert_from_udp
TCPhandler::user_changed_channel
TCPhandler::message
TCPhandler::decipher_the_message
TCPhandler::username_already_exists
TCPhandler::create_message
tcp_logger
TCPhandler::listening_for_incoming_connection
TCPhandler::create_bye
TCPhandler::send_string
TCPhandler::create_reply

Here is the caller graph for this function:



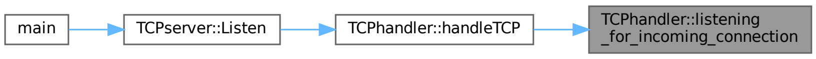
6.10.3.7 listening_for_incoming_connection()

```
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:



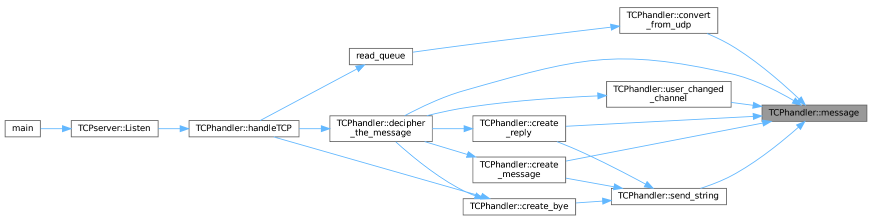
6.10.3.8 message()

```
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:



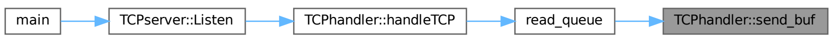
6.10.3.9 send_buf()

```
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:



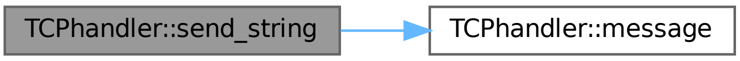
6.10.3.10 send_string()

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

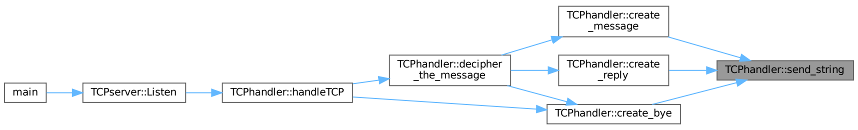
Definition at line 251 of file TCPhandler.cpp.

```
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:



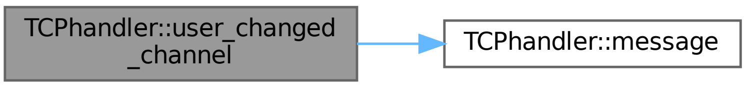
6.10.3.11 user_changed_channel()

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

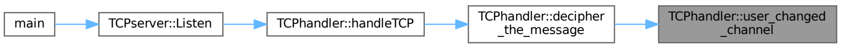
Definition at line 270 of file TCPhandler.cpp.

```
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:



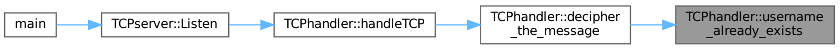
6.10.3.12 username_already_exists()

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

Definition at line 314 of file TCPhandler.cpp.

```
                                                                         {
    if (!synch_vars->usernames.empty()) {
        if (synch_vars->usernames.find(username) != synch_vars->usernames.end())
            return true;
    }
    synch_vars->usernames.insert(username);
    this->user_n = username;
    return false;
}
```

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
```

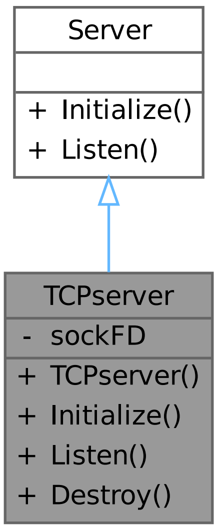
Definition at line 20 of file TCPhandler.h.
The documentation for this class was generated from the following files:

- TCPhandler.h
- TCPhandler.cpp

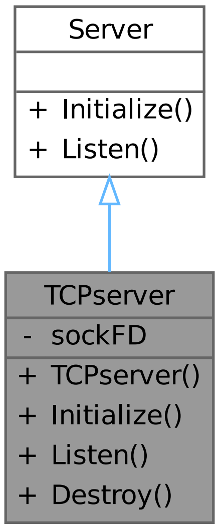
6.11 TCPserver Class Reference

```
#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.

```
                  {
        }
```

6.11.3 Member Function Documentation

6.11.3.1 Destroy()

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

Definition at line 148 of file server_classes.cpp.

```
                          {
        shutdown(this->sockFD, SHUT_RDWR);
        close(this->sockFD);
}
```

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.

```
                                                        {
    if ((this->sockFD = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    if (bind(this->sockFD, (const struct sockaddr *) server_address, sizeof(*server_address)) < 0) {
        perror("binding failed tcp");
        exit(EXIT_FAILURE);
    }
}
```

Here is the caller graph for this function:

main
TCPserver::Initialize

6.11.3.3 Listen()

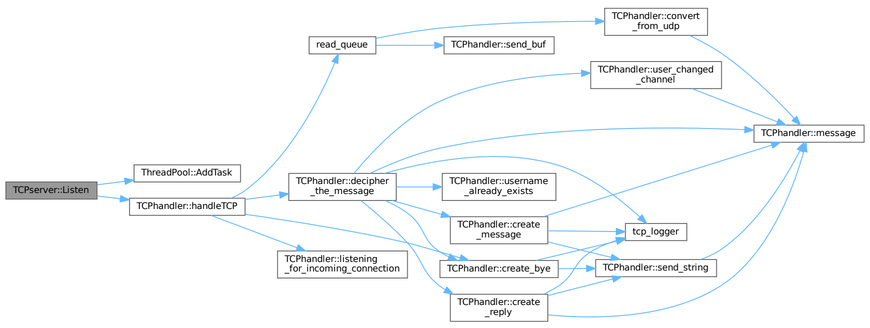
```
void TCPserver::Listen (
            ThreadPool * tp,
            std::stack< UserInfo > * s,
            synch * synch_variables,
            int signal_listener )  [override], [virtual]
```

Implements Server.

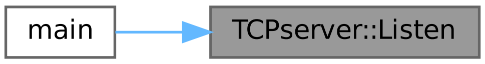
Definition at line 94 of file server_classes.cpp.

```
    {
    struct sockaddr_in client;
    listen(this->sockFD, 5);

    int epoll_fd = epoll_create1(0);
    if (epoll_fd < 0) {
        std::cerr « "Unable to create epoll instance\n";
        exit(EXIT_FAILURE);
    }

    epoll_event event;
    event.events = EPOLLIN | EPOLLET;
    event.data.fd = this->sockFD;

    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, this->sockFD, &event) < 0) {
        std::cerr « "Unable to add socket to epoll\n";
        exit(EXIT_FAILURE);
    }

    event.data.fd = signal_listener;
    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, signal_listener, &event) < 0) {
        std::cerr « "Unable to add socket to epoll\n";
        exit(EXIT_FAILURE);
    }

    struct epoll_event events[2];

    bool loop = true;
    while (loop) {
        int num_events = epoll_wait(epoll_fd, events, 2, -1); // 5 seconds timeout
        if (num_events < 0) {
            std::cerr « "Error in epoll_wait\n";
            exit(EXIT_FAILURE);
        }

        for (int i = 0; i < num_events; ++i) {
            if (events[i].data.fd == this->sockFD) {
                socklen_t len = sizeof(client);
                int clientSocket = accept(this->sockFD, (struct sockaddr *) &client, &len);
                if (clientSocket < 0) {
                    perror("accept failed");
                    continue;
                }
                tp->AddTask(
                        std::bind(&TCPhandler::handleTCP, clientSocket, &tp->busy_threads, s,
synch_variables, client,
                                  signal_listener));
            } else {
                loop = false;
                break;
            }
        }
    }
}
```

Here is the call graph for this function:


TCPserver::Listen
ThreadPool::AddTask
TCPhandler::handleTCP
read_queue
TCPhandler::decipher_the_message
TCPhandler::listening_for_incoming_connection
TCPhandler::send_buf
TCPhandler::convert_from_udp
TCPhandler::user_changed_channel
TCPhandler::username_already_exists
TCPhandler::create_message
TCPhandler::create_bye
TCPhandler::create_reply
tcp_logger
TCPhandler::send_string
TCPhandler::message

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.

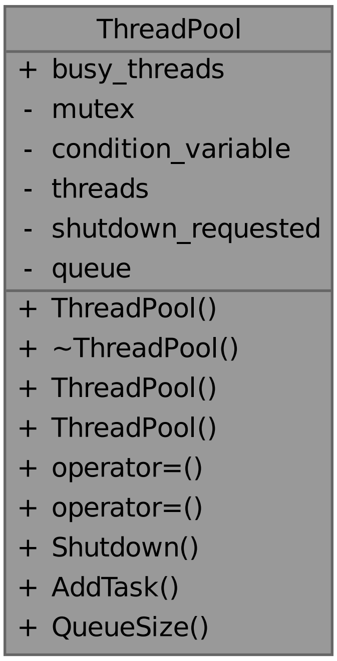
The documentation for this class was generated from the following files:

- server_classes.h
- server_classes.cpp

6.12 ThreadPool Class Reference

`#include <thread_pool.h>`

Collaboration diagram for ThreadPool:



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.

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

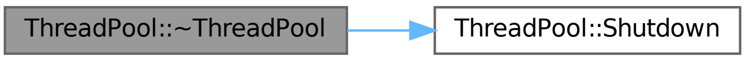
6.12.2.2 ∼ThreadPool()

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

Definition at line 20 of file thread_pool.h.

```
                    {
        Shutdown();
    }
```

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

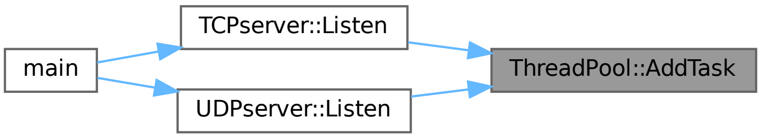
6.12.3.1 AddTask()

```
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.

```
                                                                                    {

        auto task_ptr = std::make_shared<std::packaged_task<decltype(f(args...))()»(
                std::bind(std::forward<F>(f), std::forward<Args>(args)...));

        auto wrapper_func = [task_ptr]() { (*task_ptr)(); };
        {
            std::lock_guard<std::mutex> lock(mutex);
            queue.push(wrapper_func);
            // Wake up one thread if its waiting
            condition_variable.notify_one();
        }

        // Return future from promise
        return task_ptr->get_future();
    }
```

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.

```
                      {
        std::unique_lock<std::mutex> lock(mutex);
        return queue.size();
    }
```

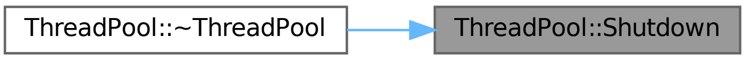
6.12.3.5 Shutdown()

```
void ThreadPool::Shutdown ( ) [inline]
```

Definition at line 33 of file thread_pool.h.

```
                      {
        {
            std::lock_guard<std::mutex> lock(mutex);
            shutdown_requested = true;
            condition_variable.notify_all();
        }

        for (size_t i = 0; i < threads.size(); ++i) {
            if (threads[i].joinable()) {
                threads[i].join();
            }
        }
    }
```

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]`

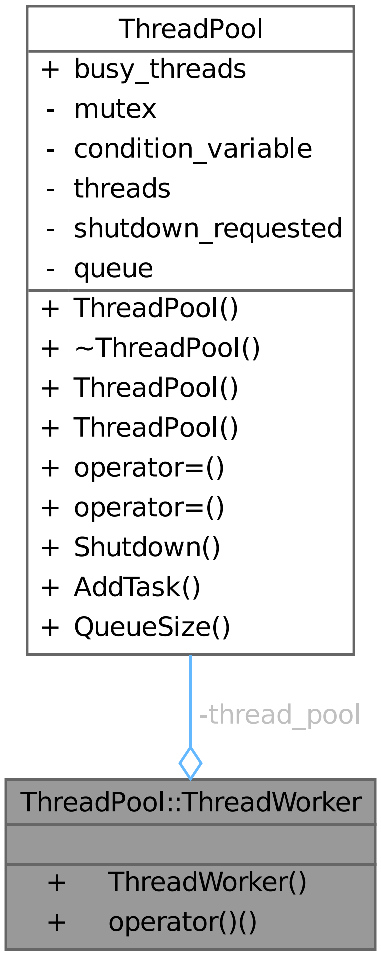
Definition at line 110 of file thread_pool.h.

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.

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

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.

```
                             {
            std::unique_lock<std::mutex> lock(thread_pool->mutex);
            while (!thread_pool->shutdown_requested ||
                   (thread_pool->shutdown_requested && !thread_pool->queue.empty())) {
                thread_pool->busy_threads--;
                thread_pool->condition_variable.wait(lock, [this] {
                    return this->thread_pool->shutdown_requested || !this->thread_pool->queue.empty();
                });
                thread_pool->busy_threads++;

                if (!this->thread_pool->queue.empty()) {

                    auto func = thread_pool->queue.front();

                    thread_pool->queue.pop();

                    lock.unlock();
                    func();
                    lock.lock();
                }
            }
        }
```

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.

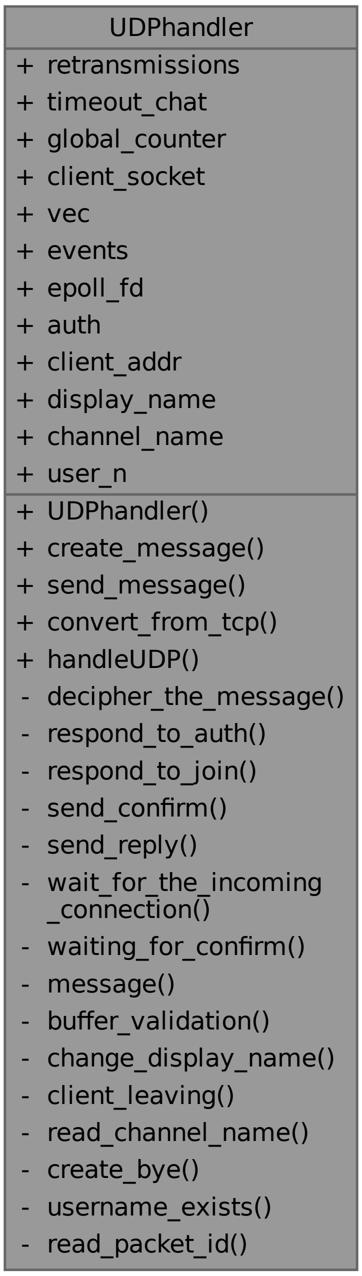
The documentation for this class was generated from the following file:

- thread_pool.h

6.14 UDPhandler Class Reference

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

Collaboration diagram for UDPhandler:



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                                                            {
        this->retransmissions = ret;
        this->timeout_chat = t;
        this->global_counter = 0;
        this->client_socket = socket(AF_INET, SOCK_DGRAM, 0);
        if (this->client_socket < 0) {
            perror("Problem with creating response socket");
            exit(EXIT_FAILURE);
        }


        epoll_fd = epoll_create1(0);
        if (epoll_fd == -1) {
            std::cerr « "Failed to create epoll file descriptor\n";
            exit(EXIT_FAILURE);
        }

        // setup epoll event
        struct epoll_event ev;
        ev.events = EPOLLIN | EPOLLET;
        ev.data.fd = this->client_socket;

        // add socket file descriptor to epoll
        if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, this->client_socket, &ev) == -1) {
            std::cerr « "Failed to add file descriptor to epoll\n";
            close(epoll_fd);
            exit(EXIT_FAILURE);
        }

        ev.data.fd = kill;
        if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, kill, &ev) < 0) {
            std::cerr « "Unable to add socket to epoll\n";
            exit(EXIT_FAILURE);
        }

        auth = false;

        client_addr = client;

        channel_name = "general";

    }
```

6.14.3 Member Function Documentation

6.14.3.1 buffer_validation()

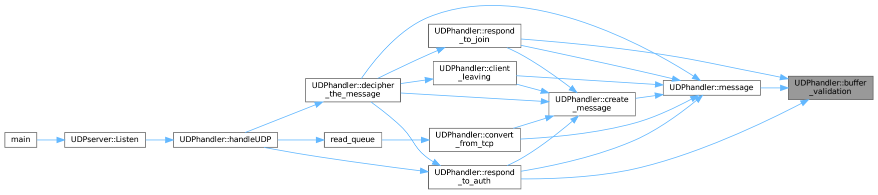
```
bool UDPhandler::buffer_validation (
            uint8_t * buf,
            int message_length,
            int start_position,
            int minimal_length,
            int amount_of_fields = 2,
            int first_limit = 20,
            int second_limit = 20,
            int third_limit =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

Definition at line 388 of file UDPhandler.cpp.

```
      {

    if (message_length < minimal_length)
        return false;

    size_t i = start_position;

    size_t count = 0;
    while (i < message_length && buf[i] != 0x00 && count < first_limit) {
        i++;
        count++;
    }

    if (i >= message_length || buf[i] != 0x00 || count < 1) {
        return false;
    }
    ++i;

    count = 0;
    while (i < message_length && buf[i] != 0x00 && count < second_limit) {
        ++i;
        ++count;
    }
```

```
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:


main
UDPserver::Listen
UDPhandler::handleUDP
read_queue
UDPhandler::decipher_the_message
UDPhandler::respond_to_join
UDPhandler::client_leaving
UDPhandler::convert_from_tcp
UDPhandler::respond_to_auth
UDPhandler::create_message
UDPhandler::message
UDPhandler::buffer_validation

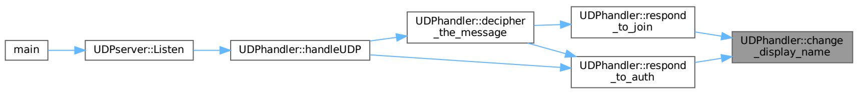
6.14.3.2 change_display_name()

```
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:


main
UDPserver::Listen
UDPhandler::handleUDP
UDPhandler::decipher_the_message
UDPhandler::respond_to_join
UDPhandler::respond_to_auth
UDPhandler::change_display_name

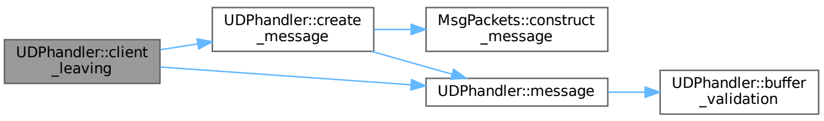
6.14.3.3 client_leaving()

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

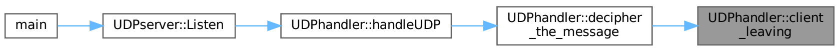
Definition at line 258 of file UDPhandler.cpp.

```
00258                                                                              {
    std::stringstream ss;
    ss « this->display_name « " has left " « this->channel_name « ".";
    std::string message = ss.str();
    uint8_t buf_message[1024];
    std::string name = "Server";
    int length = this->create_message(buf_message, message, false, name);
    this->message(buf_message, length, s, synch_var, this->channel_name);
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
}
```

Here is the call graph for this function:


UDPhandler::client_leaving
UDPhandler::create_message
MsgPackets::construct_message
UDPhandler::message
UDPhandler::buffer_validation

Here is the caller graph for this function:



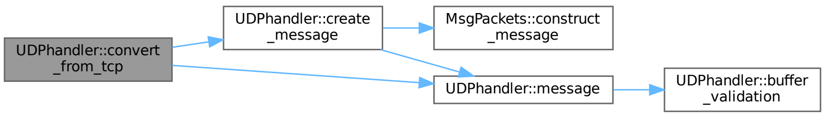
6.14.3.4 convert_from_tcp()

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

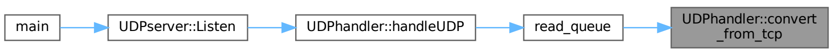
Definition at line 458 of file UDPhandler.cpp.

```
                                                            {

    std::string message;
    int i = 0;
    while (tcp_buf[i] != 0x0d) {
        message.push_back(static_cast<char>(tcp_buf[i]));
        i++;
    }

    std::regex patternFromToIs(R"(FROM\s(.*?)\sIS)");
    std::smatch matchFromToIs;
    std::regex_search(message, matchFromToIs, patternFromToIs);
    std::string name = matchFromToIs[1].str();

    std::regex patternAfterIs(R"(IS\s(.*))");
    std::smatch matchAfterIs;
    std::regex_search(message, matchAfterIs, patternAfterIs);
    std::string msg = matchAfterIs[1].str();

    return this->create_message(buf, msg, false, name);
}
```

Here is the call graph for this function:


UDPhandler::convert_from_tcp
UDPhandler::create_message
MsgPackets::construct_message
UDPhandler::message
UDPhandler::buffer_validation

Here is the caller graph for this function:



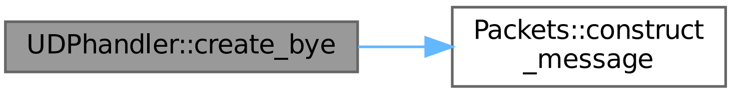
6.14.3.5 create_bye()

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

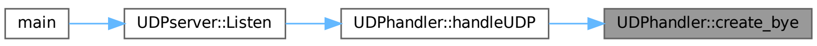
Definition at line 326 of file UDPhandler.cpp.

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

Here is the call graph for this function:



Here is the caller graph for this function:



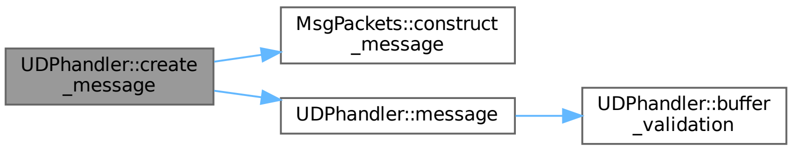
6.14.3.6 create_message()

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

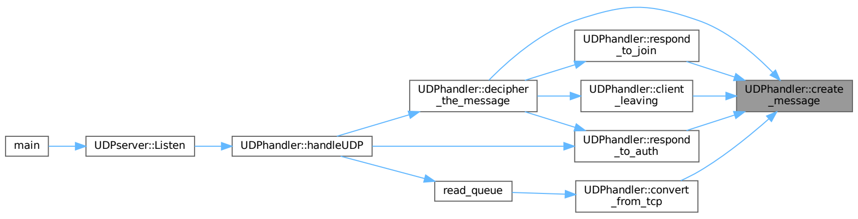
Definition at line 321 of file UDPhandler.cpp.

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

Here is the call graph for this function:


UDPhandler::create_message
MsgPackets::construct_message
UDPhandler::message
UDPhandler::buffer_validation

Here is the caller graph for this function:


main
UDPserver::Listen
UDPhandler::handleUDP
UDPhandler::decipher_the_message
read_queue
UDPhandler::respond_to_join
UDPhandler::client_leaving
UDPhandler::respond_to_auth
UDPhandler::convert_from_tcp
UDPhandler::create_message

6.14.3.7 decipher_the_message()

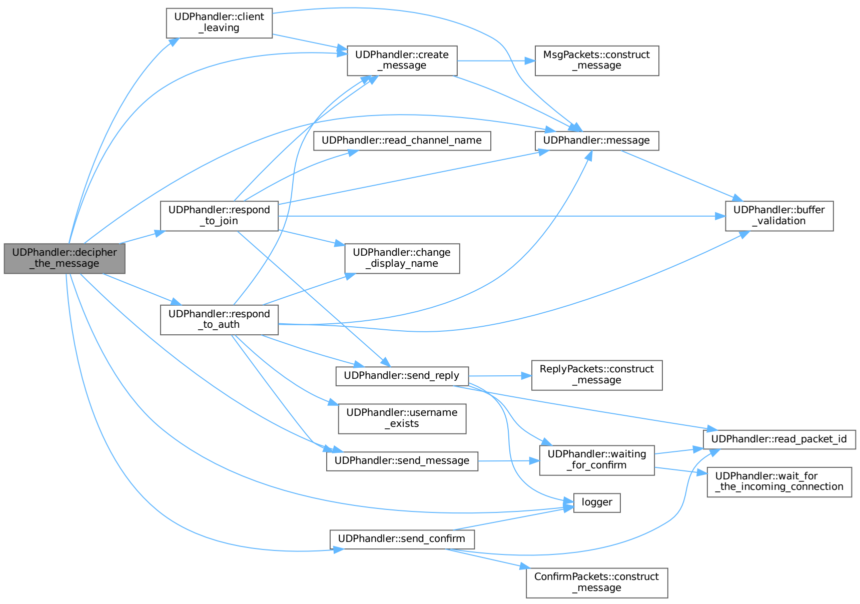
```
bool UDPhandler::decipher_the_message (
            uint8_t * buf,
            int length,
            std::stack< UserInfo > * s,
            synch * synch_var )  [private]
```

Definition at line 89 of file UDPhandler.cpp.

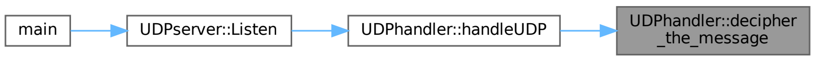
```
00089
     {
00090     if (!this->auth) {
00091         if (buf[0] != 0x02) {
00092             if (buf[0] != 0xFF) {
00093                 if (buf[0] != 0xFE) {
00094                     uint8_t buf_int[1024];
00095                     std::string message = "You should log-in before doing anything 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", "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");
```

```
            send_confirm(buf);
            respond_to_join(buf, length, s, synch_var);
            break;
        case 0x04://MSG
            logger(this->client_addr, "MSG", "RECV");
            send_confirm(buf);
            this->message(buf, length, s, synch_var, this->channel_name);
            break;
        case 0xFF://BYE
            if (this->auth) {
                this->client_leaving(s, synch_var);
            }
            logger(this->client_addr, "BYE", "RECV");
            send_confirm(buf);
            return false;
        case 0xFE://ERR
            if (this->auth) {
                this->client_leaving(s, synch_var);
            }
            logger(this->client_addr, "ERR", "RECV");
            send_confirm(buf);
            return false;
        default:
            uint8_t buf[1024];
            std::string message = "Unknown instruction";
            std::string name = "Server";
            int length_err = this->create_message(buf, message, true, name);
            this->send_message(buf, length_err, false);
            return false;
    }
    return true;
}
```

Here is the call graph for this function:


UDPhandler::client_leaving
UDPhandler::create_message
MsgPackets::construct_message
UDPhandler::read_channel_name
UDPhandler::message
UDPhandler::respond_to_join
UDPhandler::buffer_validation
UDPhandler::decipher_the_message
UDPhandler::change_display_name
UDPhandler::respond_to_auth
UDPhandler::send_reply
ReplyPackets::construct_message
UDPhandler::username_exists
UDPhandler::read_packet_id
UDPhandler::waiting_for_confirm
UDPhandler::send_message
UDPhandler::wait_for_the_incoming_connection
logger
UDPhandler::send_confirm
ConfirmPackets::construct_message

Here is the caller graph for this function:



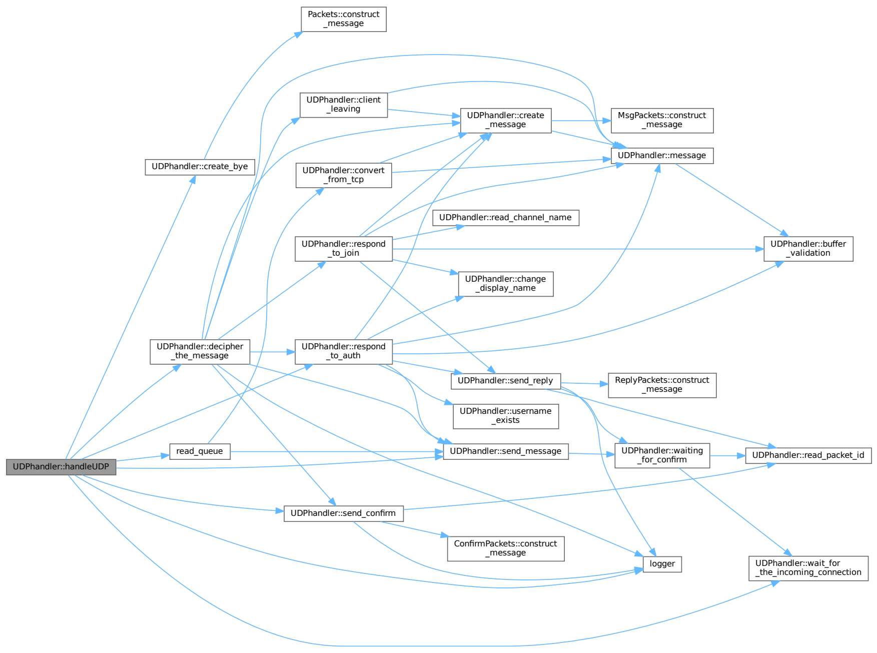
6.14.3.8 handleUDP()

```
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]
```

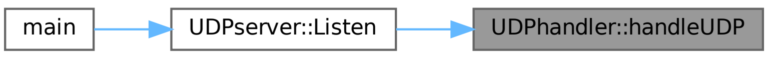
Definition at line 6 of file UDPhandler.cpp.

```
                                                                                                                      {

    UDPhandler udp(retransmissions, timeout, client_addr, signal_listener);

    bool end = false;

    std::thread sender(read_queue, s, &end, synch_var, busy, &udp);

    uint8_t internal_buf[2048];
    logger(udp.client_addr, "AUTH", "RECV");
    udp.send_confirm(buf);
    int result = udp.respond_to_auth(buf, length, s, synch_var);

    if (result != -1) {
        while (true) {
            int length_internal = udp.wait_for_the_incoming_connection(internal_buf);
            if (length_internal == -1) {
                uint8_t buf_int[256];
                int length_int = udp.create_bye(buf_int);
                udp.send_message(buf_int, length_int, true);
                break;
            }
            if (!udp.decipher_the_message(internal_buf, length_internal, s, synch_var)) {
                break;
            }
        }
    }

    if (synch_var->usernames.find(udp.user_n) != synch_var->usernames.end())
        synch_var->usernames.erase(udp.user_n);

    end = true;
    {
        std::lock_guard<std::mutex> lock(synch_var->mtx);
        synch_var->ready = true;
    }
    synch_var->cv.notify_all();

    sender.join();
    close(udp.client_socket);
}
```

Here is the call graph for this function:


Packets::construct_message
UDPhandler::client_leaving
UDPhandler::create_message
MsgPackets::construct_message
UDPhandler::create_bye
UDPhandler::message
UDPhandler::convert_from_tcp
UDPhandler::read_channel_name
UDPhandler::respond_to_join
UDPhandler::buffer_validation
UDPhandler::change_display_name
UDPhandler::decipher_the_message
UDPhandler::respond_to_auth
UDPhandler::send_reply
ReplyPackets::construct_message
UDPhandler::username_exists
read_queue
UDPhandler::send_message
UDPhandler::waiting_for_confirm
UDPhandler::read_packet_id
UDPhandler::handleUDP
UDPhandler::send_confirm
ConfirmPackets::construct_message
logger
UDPhandler::wait_for_the_incoming_connection

Here is the caller graph for this function:



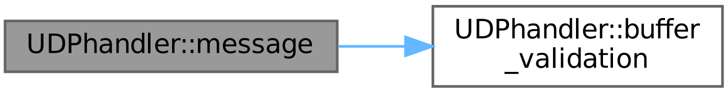
6.14.3.9 message()

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

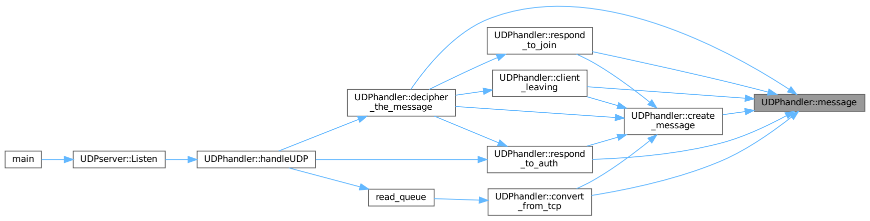
Definition at line 239 of file UDPhandler.cpp.

```
                                                  {
    bool valid_message = true;

    if (!this->buffer_validation(buf, message_length, 3, 2, 2, 20, 1400))
        valid_message = false;

    if (valid_message) {
        {
            std::lock_guard<std::mutex> lock(synch_var->mtx);
            s->emplace(this->client_addr, buf, message_length, channel, false, 0);
            synch_var->ready = true;
        }
        synch_var->cv.notify_all();
    } else {
        std::cout « "Invalid message" « std::endl;
    }
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



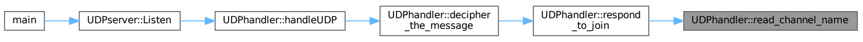
6.14.3.10 read_channel_name()

```
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:



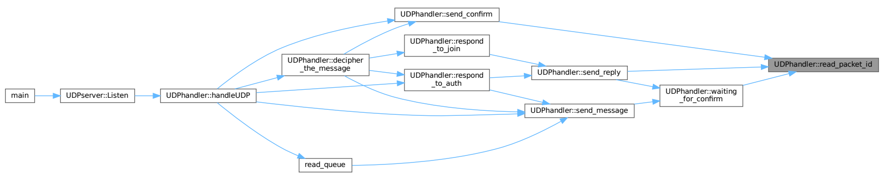
6.14.3.11 read_packet_id()

```
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:


main
UDPserver::Listen
UDPhandler::handleUDP
UDPhandler::decipher_the_message
read_queue
UDPhandler::send_confirm
UDPhandler::respond_to_join
UDPhandler::respond_to_auth
UDPhandler::send_reply
UDPhandler::send_message
UDPhandler::waiting_for_confirm
UDPhandler::read_packet_id

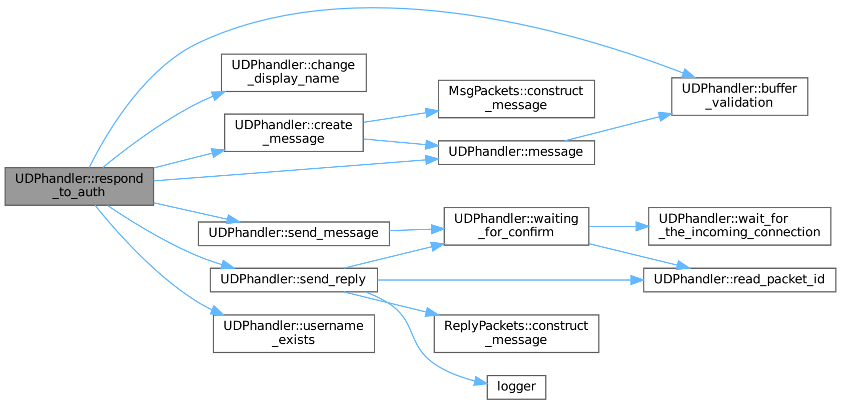
6.14.3.12 respond_to_auth()

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

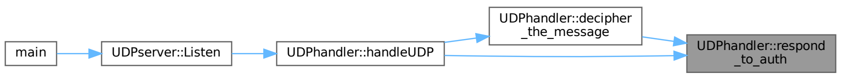
Definition at line 156 of file UDPhandler.cpp.

```
00156
      {
00157
00158     bool valid_message = true;
00159
00160     if (buf[0] == 0xFF) {
00161         return -1;
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 }
```

Here is the call graph for this function:


UDPhandler::respond_to_auth
UDPhandler::change_display_name
UDPhandler::create_message
MsgPackets::construct_message
UDPhandler::message
UDPhandler::buffer_validation
UDPhandler::send_message
UDPhandler::waiting_for_confirm
UDPhandler::wait_for_the_incoming_connection
UDPhandler::send_reply
UDPhandler::read_packet_id
UDPhandler::username_exists
ReplyPackets::construct_message
logger

Here is the caller graph for this function:



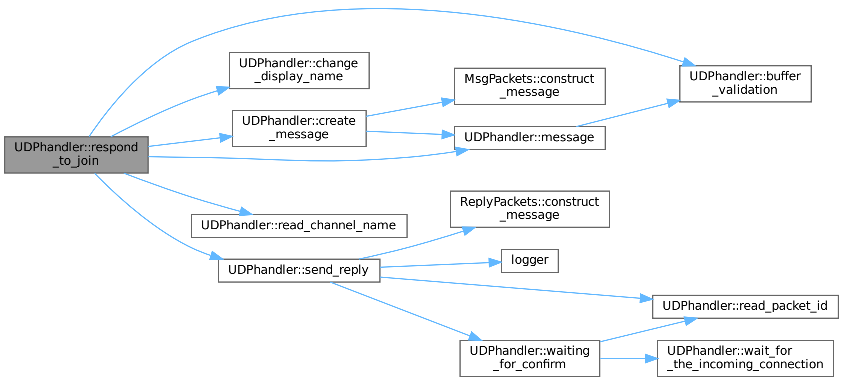
6.14.3.13 respond_to_join()

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

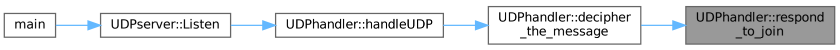
Definition at line 204 of file UDPhandler.cpp.

```
{
    bool valid = true;

    if (!this->buffer_validation(buf, message_length, 3, 2))
        valid = false;

    if (valid) {
        this->change_display_name(buf, true);
        std::string success = "Join is succesful";
        send_reply(buf, success, true);

        std::stringstream ss;
        ss « this->display_name « " has left " « this->channel_name « ".";
        std::string message = ss.str();
        uint8_t buf_message[1024];
        std::string name = "Server";
        int length = this->create_message(buf_message, message, false, name);
        this->message(buf_message, length, s, synch_var, this->channel_name);

        std::this_thread::sleep_for(std::chrono::milliseconds(10));

        memset(buf_message, 0, 1024);
        std::stringstream joined;
        this->channel_name = this->read_channel_name(buf);
        joined « this->display_name « " has joined " « this->channel_name « ".";
        std::string message_new = joined.str();
        length = this->create_message(buf_message, message_new, false, name);
        this->message(buf_message, length, s, synch_var, this->channel_name);

    } else {
        std::string failure = "Join is not succesful";
00235         send_reply(buf, failure, false);
00236     }
00237 }
```

Here is the call graph for this function:


UDPhandler::respond_to_join
UDPhandler::change_display_name
UDPhandler::create_message
MsgPackets::construct_message
UDPhandler::message
UDPhandler::buffer_validation
UDPhandler::read_channel_name
UDPhandler::send_reply
ReplyPackets::construct_message
logger
UDPhandler::read_packet_id
UDPhandler::waiting_for_confirm
UDPhandler::wait_for_the_incoming_connection

Here is the caller graph for this function:



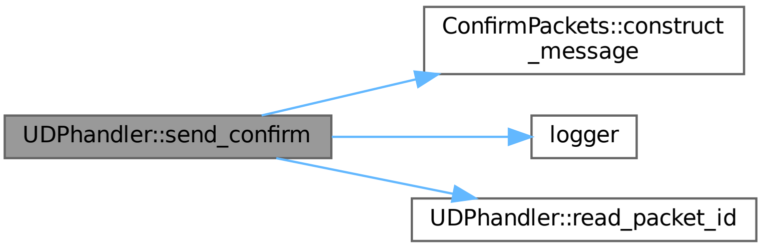
6.14.3.14 send_confirm()

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

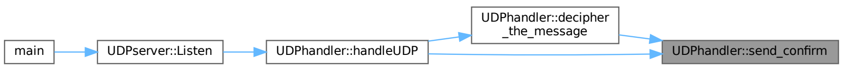
Definition at line 270 of file UDPhandler.cpp.

```
00270                                            {
00271     uint8_t buf_out[4];
00272 
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 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



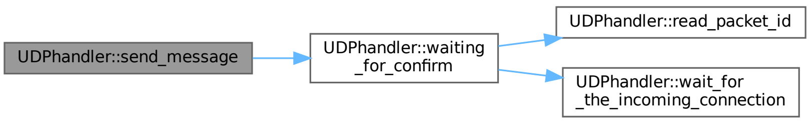
6.14.3.15 send_message()

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

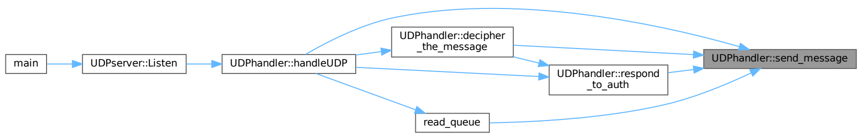
Definition at line 306 of file UDPhandler.cpp.

```
                                                                                 {
    socklen_t address_size = sizeof(this->client_addr);

    sockaddr_in backup = this->client_addr;

    sendto(this->client_socket, buf, message_length, 0, (struct sockaddr *) &this->client_addr,
      address_size);
    this->global_counter++;

    if (!terminate) {
        if (!waiting_for_confirm(buf, message_length))
            std::cout « "Client didn't confirm" « std::endl;
    }
    this->client_addr = backup;
}
```

Here is the call graph for this function:


UDPhandler::send_message
UDPhandler::waiting_for_confirm
UDPhandler::read_packet_id
UDPhandler::wait_for_the_incoming_connection

Here is the caller graph for this function:



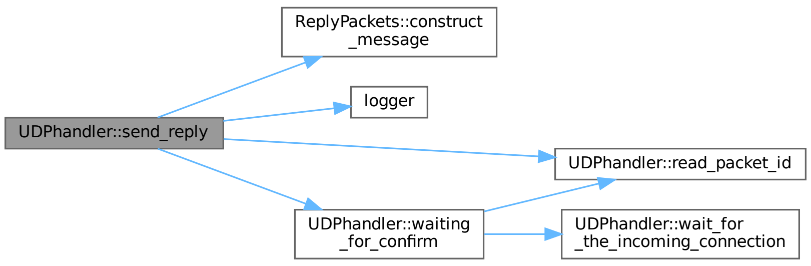
6.14.3.16 send_reply()

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

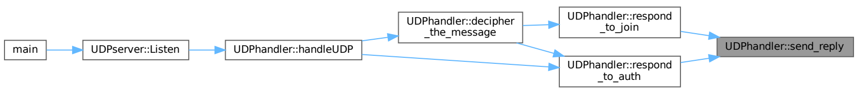
Definition at line 287 of file UDPhandler.cpp.

```
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 *)
      &(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 }
```

Here is the call graph for this function:


ReplyPackets::construct_message
logger
UDPhandler::send_reply
UDPhandler::read_packet_id
UDPhandler::waiting_for_confirm
UDPhandler::wait_for_the_incoming_connection

Here is the caller graph for this function:


main
UDPserver::Listen
UDPhandler::handleUDP
UDPhandler::decipher_the_message
UDPhandler::respond_to_join
UDPhandler::respond_to_auth
UDPhandler::send_reply

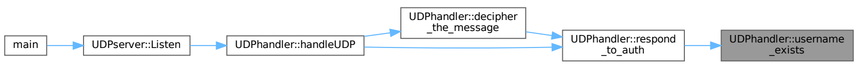
6.14.3.17 username_exists()

```
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:


main
UDPserver::Listen
UDPhandler::handleUDP
UDPhandler::decipher_the_message
UDPhandler::respond_to_auth
UDPhandler::username_exists

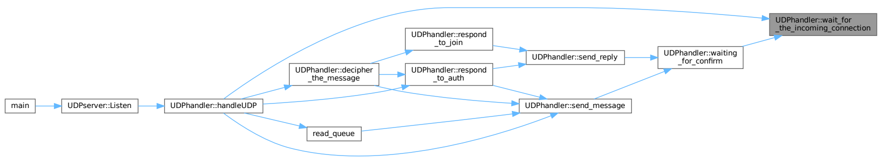
6.14.3.18 wait_for_the_incoming_connection()

```
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:


main
UDPserver::Listen
UDPhandler::handleUDP
UDPhandler::decipher_the_message
UDPhandler::respond_to_join
UDPhandler::respond_to_auth
UDPhandler::send_reply
UDPhandler::waiting_for_confirm
UDPhandler::wait_for_the_incoming_connection
UDPhandler::send_message
read_queue

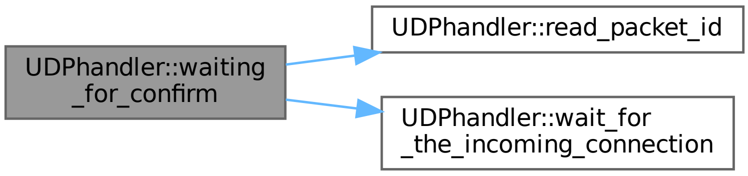
6.14.3.19 waiting_for_confirm()

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

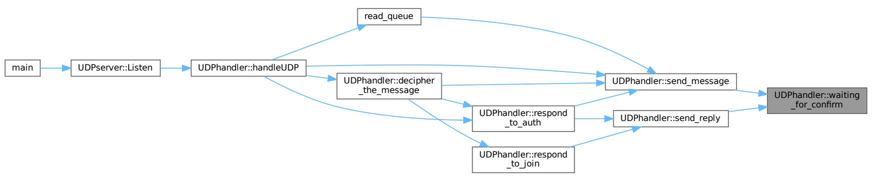
Definition at line 364 of file UDPhandler.cpp.

```
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 *)
      &(this->client_addr),
00381                                    len_client);
00382             if (bytes_tx < 0) perror("ERROR: sendto");
00383         }
00384     }
00385     return confirmed;
00386 }
```

Here is the call graph for this function:


UDPhandler::waiting_for_confirm
UDPhandler::read_packet_id
UDPhandler::wait_for_the_incoming_connection

Here is the caller graph for this function:


main
UDPserver::Listen
UDPhandler::handleUDP
read_queue
UDPhandler::decipher_the_message
UDPhandler::respond_to_auth
UDPhandler::respond_to_join
UDPhandler::send_message
UDPhandler::send_reply
UDPhandler::waiting_for_confirm

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.

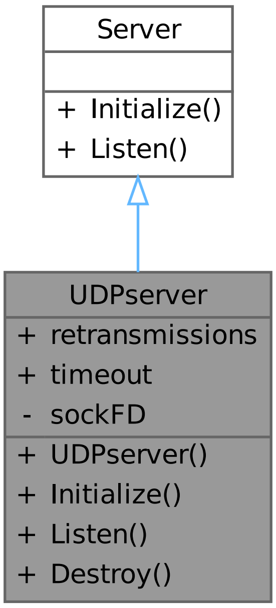
The documentation for this class was generated from the following files:

- UDPhandler.h
- UDPhandler.cpp

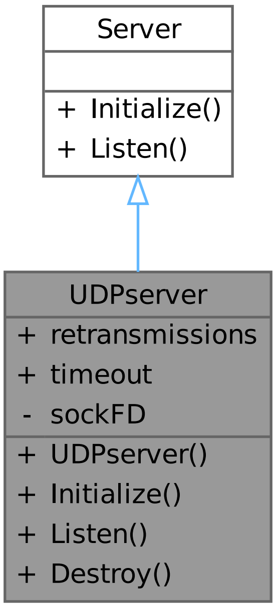
6.15 UDPserver Class Reference

```
#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;
```

```
        this->timeout = t;
    }
```

6.15.3 Member Function Documentation

6.15.3.1 Destroy()

void UDPserver::Destroy ( )

Definition at line 78 of file server_classes.cpp.

```
                                {
    close(this->sockFD);
}
```

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.

```
                                                            {
    if ((this->sockFD = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    if (bind(this->sockFD, (const struct sockaddr *) server_address, sizeof(*server_address)) < 0) {
        perror("binding failed udp");
        exit(EXIT_FAILURE);
    }
}
```

6.15.3.3 Listen()

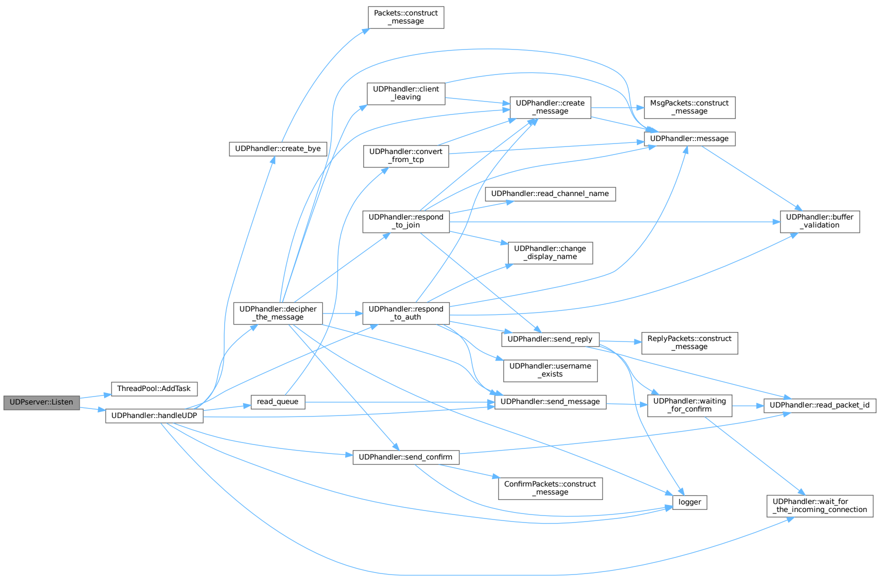
```
void UDPserver::Listen (
            ThreadPool * tp,
            std::stack< UserInfo > * s,
            synch * synch_variables,
            int signal_listener )  [override], [virtual]
```

Implements Server.

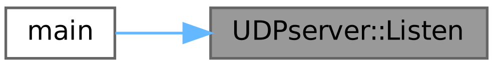
Definition at line 18 of file server_classes.cpp.

```

{
    int epoll_fd = epoll_create1(0);
    if (epoll_fd < 0) {
        std::cerr « "Unable to create epoll instance\n";
        exit(EXIT_FAILURE);
    }

    epoll_event event;
    event.events = EPOLLIN | EPOLLET;
    event.data.fd = this->sockFD;

    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, this->sockFD, &event) < 0) {
        std::cerr « "Unable to add socket to epoll\n";
        exit(EXIT_FAILURE);
    }

    event.data.fd = signal_listener;
    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, signal_listener, &event) < 0) {
        std::cerr « "Unable to add socket to epoll\n";
        exit(EXIT_FAILURE);
    }

    struct epoll_event events[2];

    bool loop = true;
    while (loop) {
        int num_events = epoll_wait(epoll_fd, events, 2, -1); // 5 seconds timeout

        if (num_events < 0) {
            std::cerr « "Error in epoll_wait\n";
            exit(EXIT_FAILURE);
        }

        for (int i = 0; i < num_events; ++i) {
            if (events[i].data.fd == this->sockFD) {
                uint8_t buf[1024];
                sockaddr_in client_addr;
                if (!(events[i].events & EPOLLIN))
                    continue;

                socklen_t len = sizeof(client_addr);

                int n = recvfrom(this->sockFD, buf, 1024,
                                 0, (struct sockaddr *) &client_addr, &len);

                if (n == -1) {
                    std::cerr « "recvfrom failed. errno: " « errno « '\n';
                    continue;
                }

                tp->AddTask(std::bind(&UDPhandler::handleUDP, buf, client_addr, n,
this->retransmissions, this->timeout,
                                      &tp->busy_threads, s, synch_variables, signal_listener));
            } else {
                loop = false;
                break;
            }
        }
    }
}
```

Here is the call graph for this function:


Packets::construct_message
UDPhandler::client_leaving
UDPhandler::create_message
MsgPackets::construct_message
UDPhandler::create_bye
UDPhandler::convert_from_tcp
UDPhandler::message
UDPhandler::read_channel_name
UDPhandler::respond_to_join
UDPhandler::buffer_validation
UDPhandler::change_display_name
UDPhandler::decipher_the_message
UDPhandler::respond_to_auth
UDPhandler::send_reply
ReplyPackets::construct_message
UDPhandler::username_exists
ThreadPool::AddTask
UDPserver::Listen
read_queue
UDPhandler::send_message
UDPhandler::waiting_for_confirm
UDPhandler::read_packet_id
UDPhandler::handleUDP
UDPhandler::send_confirm
ConfirmPackets::construct_message
logger
UDPhandler::wait_for_the_incoming_connection

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.

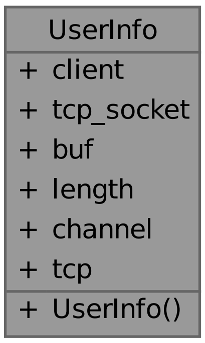
The documentation for this class was generated from the following files:

- server_classes.h
- server_classes.cpp

6.16 UserInfo Struct Reference

`#include <synch.h>`

Collaboration diagram for UserInfo:



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.

```
: client(c), buf(m),
     length(l),
     channel(std::move(name)), tcp(t),
                                                         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

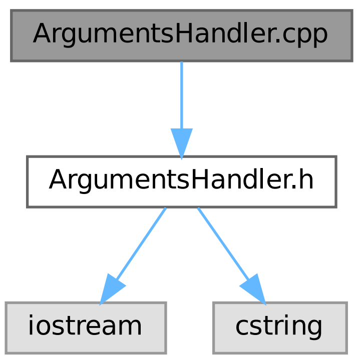
Chapter 7

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.

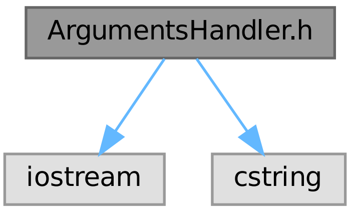
```
//
// Created by artem on 4/19/24.
//


#include "ArgumentsHandler.h"

void ArgumentsHandler::print_help() {
    std::cout « R"(|Argument | Default values | Type

──────────────────────────────────────────────────────────────────────────────────────────────
| -l      | 127.0.0.1      | IP address                   | Server listening IP address for welcome sockets     |
| -p      | 47356          | uint16                       | Server listening port for welcome sockets |
| -d      | 500            | uint16                       | UDP confirmation timeout |
| -r      | 3              | uint8                        | Maximum number of UDP retransmissions |
| -n      | 20             | uint16                       | Maximum number of threads in the thread pool |
| -h      |                |                              | Prints program help output and exits |
)";
}

void ArgumentsHandler::get_args(int argc, char **argv) {
    this->timeout = 500;
    this->retransmissions = 3;
    this->port = 47356;
    this->address = new char[13];
    this->number_of_threads = 20;
    strcpy(address, "127.0.0.1");

    for (int i = 0; i < argc; i++) {
        std::string arg = argv[i];

        if (arg == "-h") {
            print_help();
            exit(0);
        } else if (arg == "-l") {
            i++;
            if (i < argc) {
                address = argv[i];
            } else {
                std::cout « "Nothing passed to address" « std::endl;
                exit(1);
            }
        } else if (arg == "-p") {
            i++;
            if (i < argc) {
                try {
                    port = std::stoi(argv[i]);
                } catch (std::invalid_argument &) {
                    std::cout « "Passed non-int value to port" « std::endl;
                    exit(1);
                }
            } else {
                std::cout « "Nothing passed to port" « std::endl;
                exit(1);
            }
        } else if (arg == "-d") {
            i++;
            if (i < argc) {
                try {
                    this->timeout = std::stoi(argv[i]);
                } catch (std::invalid_argument &) {
                    std::cout « "Passed non-int value to timeout" « std::endl;
                    exit(1);
                }
            } else {
                std::cout « "Nothing passed to timeout" « std::endl;
                exit(1);
            }
        } else if (arg == "-r") {
            i++;
            if (i < argc) {
                try {
                    this->retransmissions = std::stoi(argv[i]);
                } catch (std::invalid_argument &) {
                    std::cout « "Passed non-int value to retransmissions" « std::endl;
                    exit(1);
                }
            } else {
                std::cout « "Nothing passed to retransmissions" « std::endl;
                exit(1);
            }
        } else if (arg == "-n"){
            i++;
            if (i < argc) {
                try {
                    this->number_of_threads = std::stoi(argv[i]);
                } catch (std::invalid_argument &) {
                    std::cout « "Passed non-int value to number of threads" « std::endl;
                    exit(1);
                }
            } else {
                std::cout « "Nothing passed to number of threads" « std::endl;
                exit(1);
            }
        }
    }


}

int ArgumentsHandler::get_port() const {
    return this->port;
}

```

```
int ArgumentsHandler::get_retransmissions() {
    return this->retransmissions;
}

int ArgumentsHandler::get_timeout() {
    return this->timeout;
}

char *ArgumentsHandler::get_address() {
    return this->address;
}

int ArgumentsHandler::get_threads() {
    return this->number_of_threads;
}
```

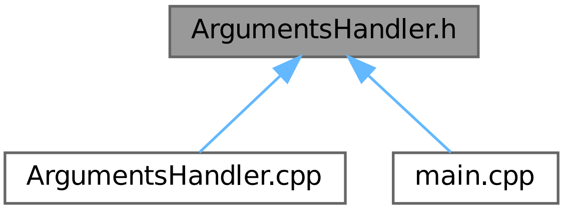
7.3 ArgumentsHandler.h File Reference

```
#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.

```
//
// Created by artem on 4/19/24.
```

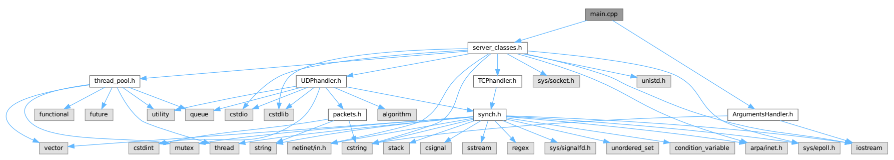
```
//

#ifndef IPK_SERVER_ARGUMENTSHANDLER_H
#define IPK_SERVER_ARGUMENTSHANDLER_H

#include <iostream>
#include <cstring>

class ArgumentsHandler {
public:
    void get_args(int argc, char *argv[]);

    int get_timeout();

    int get_retransmissions();

    int get_port() const;

    char *get_address();

    int get_threads();

private:
    int timeout;
    int port;
    char *address;
    int retransmissions;
    int number_of_threads;

    static void print_help();
};


#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

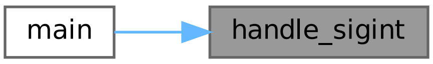
7.6.1.1 handle_sigint()

```
void handle_sigint (
            int sig )
```

Definition at line 52 of file main.cpp.

```
00052                                {
    write(pipefd[1], "X", 1);
}
```

Here is the caller graph for this function:



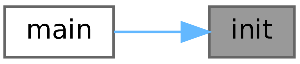
7.6.1.2 init()

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

Definition at line 45 of file main.cpp.

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

Here is the caller graph for this function:



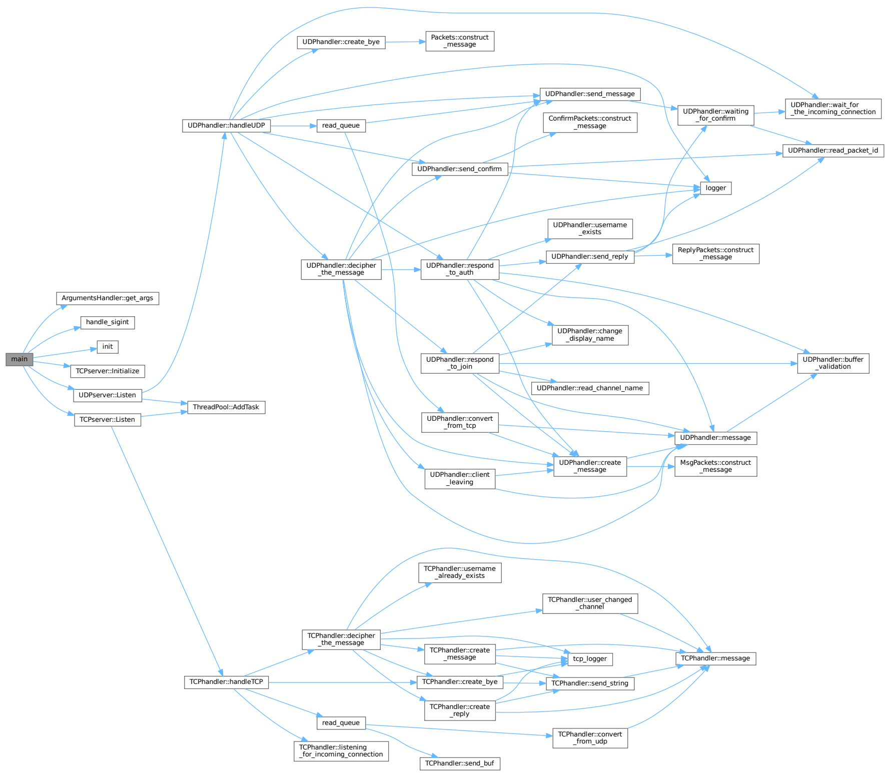
7.6.1.3 main()

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

Definition at line 10 of file main.cpp.

```
                                  {

    ArgumentsHandler ah{};
    ah.get_args(argc, argv);

    ThreadPool tp{ah.get_threads()};
    std::stack<UserInfo> s;
    synch synch_variables(0);
    struct sockaddr_in *server_addr = new sockaddr_in;

    init(server_addr, ah.get_port(), ah.get_address());

    signal(SIGINT, handle_sigint);
    pipe(pipefd);

    UDPserver udp{ah.get_retransmissions(), ah.get_timeout()};
    TCPserver tcp{};

    udp.Initialize(server_addr);
    tcp.Initialize(server_addr);

    std::thread tcpThread(&TCPserver::Listen, &tcp, &tp, &s, &synch_variables, pipefd[0]);
    std::thread udpThread(&UDPserver::Listen, &udp, &tp, &s, &synch_variables, pipefd[0]);

    tcpThread.join();
    udpThread.join();

    tp.Shutdown();

    tcp.Destroy();
    udp.Destroy();

    delete server_addr;
}
```

Here is the call graph for this function:



main
ArgumentsHandler::get_args
handle_sigint
init
TCPserver::Initialize
UDPserver::Listen
TCPserver::Listen
ThreadPool::AddTask
UDPhandler::handleUDP
UDPhandler::create_bye
Packets::construct_message
read_queue
UDPhandler::send_message
ConfirmPackets::construct_message
UDPhandler::waiting_for_confirm
UDPhandler::wait_for_the_incoming_connection
UDPhandler::read_packet_id
UDPhandler::send_confirm
logger
UDPhandler::username_exists
UDPhandler::send_reply
ReplyPackets::construct_message
UDPhandler::decipher_the_message
UDPhandler::respond_to_auth
UDPhandler::change_display_name
UDPhandler::respond_to_join
UDPhandler::buffer_validation
UDPhandler::read_channel_name
UDPhandler::convert_from_tcp
UDPhandler::message
UDPhandler::create_message
MsgPackets::construct_message
UDPhandler::client_leaving
TCPhandler::username_already_exists
TCPhandler::user_changed_channel
TCPhandler::decipher_the_message
TCPhandler::create_message
tcp_logger
TCPhandler::message
TCPhandler::handleTCP
TCPhandler::create_bye
TCPhandler::send_string
TCPhandler::create_reply
read_queue
TCPhandler::convert_from_udp
TCPhandler::listening_for_incoming_connection
TCPhandler::send_buf

7.6.2 Variable Documentation

7.6.2.1 pipefd

```
int pipefd[2]
```

Definition at line 4 of file main.cpp.

7.7 main.cpp

Go to the documentation of this file.

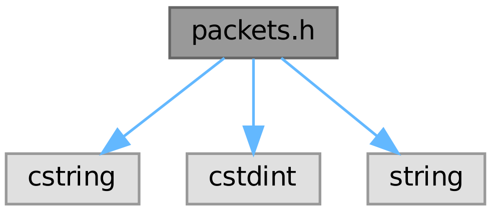
```
00001 #include "server_classes.h"
00002 #include "ArgumentsHandler.h"
00003
00004 int pipefd[2];
00005
00006 void init(struct sockaddr_in *server_addr, int port, const char *addr);
00007
00008 void handle_sigint(int sig);
00009
00010 int main(int argc, char *argv[]) {
00011
00012     ArgumentsHandler ah{};
00013     ah.get_args(argc, argv);
```

```

    ThreadPool tp{ah.get_threads()};
    std::stack<UserInfo> s;
    synch synch_variables(0);
    struct sockaddr_in *server_addr = new sockaddr_in;

    init(server_addr, ah.get_port(), ah.get_address());

    signal(SIGINT, handle_sigint);
    pipe(pipefd);

    UDPserver udp{ah.get_retransmissions(), ah.get_timeout()};
    TCPserver tcp{};

    udp.Initialize(server_addr);
    tcp.Initialize(server_addr);

    std::thread tcpThread(&TCPserver::Listen, &tcp, &tp, &s, &synch_variables, pipefd[0]);
    std::thread udpThread(&UDPserver::Listen, &udp, &tp, &s, &synch_variables, pipefd[0]);

    tcpThread.join();
    udpThread.join();

    tp.Shutdown();

    tcp.Destroy();
    udp.Destroy();

    delete server_addr;
}

void init(struct sockaddr_in *server_addr, int port, const char *addr) {
    memset(server_addr, 0, sizeof(*server_addr));
    server_addr->sin_family = AF_INET;
    server_addr->sin_port = htons(port);
    server_addr->sin_addr.s_addr = inet_addr(addr);
}

void handle_sigint(int sig) {
    write(pipefd[1], "X", 1);
}


```

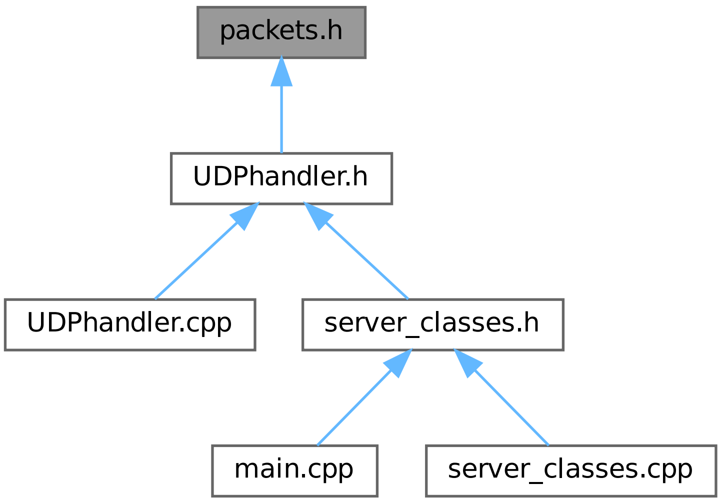
7.8 packets.h File Reference

```
#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.

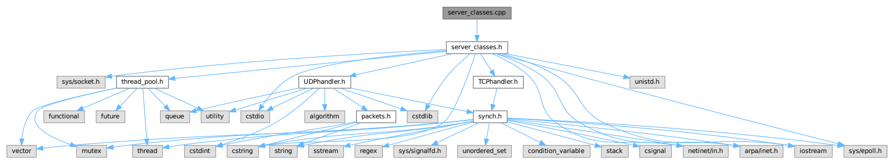
```
//
// Created by artem on 4/14/24.
//

#ifndef IPK_SERVER_PACKETS_H

#define IPK_SERVER_PACKETS_H

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

#endif //IPK_SERVER_PACKETS_H


typedef struct Packets {
    uint8_t MessageType;
    uint16_t MessageID;

    Packets(uint8_t type, uint16_t id) {
        MessageType = type;
        MessageID = id;
    }

    virtual int construct_message(uint8_t *b) {
        memcpy(b, &this->MessageType, sizeof(this->MessageType));
        b += sizeof(this->MessageType);


        //uint16_t ID = htons(this->MessageID);
        uint16_t ID = this->MessageID;
        memcpy(b, &ID, sizeof(ID));
        b += sizeof(ID);
        return 3;
    }

} Packet;

typedef struct ConfirmPackets : public Packets {

    uint16_t Ref_MessageID;

    ConfirmPackets(uint8_t type, uint16_t id, uint16_t ref_id) : Packets(type, id) {
        Ref_MessageID = ref_id;
    }

    int construct_message(uint8_t *b) override {
        memcpy(b, &this->MessageType, sizeof(this->MessageType));
        b += sizeof(this->MessageType);

        uint16_t ID = this->Ref_MessageID;
        memcpy(b, &ID, sizeof(ID));
        b += sizeof(ID);
        return sizeof(this->MessageType) + sizeof(ID);
    }

} ConfirmPacket;

typedef struct JoinPackets : public Packets {

    std::string ChannelID;
    std::string DisplayName;

    JoinPackets(uint8_t type, uint16_t id, std::string ch_id, std::string disp_name) : Packets(type,
id) {
        ChannelID = std::move(ch_id);
        DisplayName = std::move(disp_name);
    }

    int construct_message(uint8_t *b) override {
        memcpy(b, &this->MessageType, sizeof(this->MessageType));
        b += sizeof(this->MessageType);

        //uint16_t ID = htons(this->MessageID);
        uint16_t ID = this->MessageID;
        memcpy(b, &ID, sizeof(ID));
        b += sizeof(ID);

        memcpy(b, ChannelID.c_str(), ChannelID.length());
        b[ChannelID.length()] = '\0';
        b += ChannelID.length() + 1;

        memcpy(b, DisplayName.c_str(), DisplayName.length());
        b[DisplayName.length()] = '\0';
        b += DisplayName.length() + 1;
        return sizeof(this->MessageType) + sizeof(ID) + ChannelID.length() + 1 + DisplayName.length()
+ 1;
    }

} JoinPacket;

// To create ERR use MsgPackets struct
typedef struct MsgPackets : public Packets {

    std::string MessageContents;
    std::string DisplayName;

    MsgPackets(uint8_t type, uint16_t id, std::string content, std::string disp_name) : Packets(type,
id) {
        MessageContents = std::move(content);
        DisplayName = std::move(disp_name);
    }

    int construct_message(uint8_t *b) override {
        memcpy(b, &this->MessageType, sizeof(this->MessageType));
        b += sizeof(this->MessageType);

        //uint16_t ID = htons(this->MessageID);
        uint16_t ID = this->MessageID;
        memcpy(b, &ID, sizeof(ID));
        b += sizeof(ID);

        memcpy(b, DisplayName.c_str(), DisplayName.length());
        b[DisplayName.length()] = '\0';
        b += DisplayName.length() + 1;

        memcpy(b, MessageContents.c_str(), MessageContents.length());
        b[MessageContents.length()] = '\0';
        b += MessageContents.length() + 1;
        return sizeof(this->MessageType) + sizeof(ID) + DisplayName.length() + 1 +
MessageContents.length() + 1;
    }

} MsgPacket;

typedef struct AuthPackets : public Packets {

    std::string Username;
    std::string DisplayName;
    std::string Secret;

    AuthPackets(uint8_t type, uint16_t id, std::string u_n, std::string disp_name, std::string sec) :
Packets(type,

id) {
        Username = std::move(u_n);
        DisplayName = std::move(disp_name);
        Secret = std::move(sec);
    }

    int construct_message(uint8_t *b) override {
        memcpy(b, &this->MessageType, sizeof(this->MessageType));
        b += sizeof(this->MessageType);
        //std::cout«this->MessageType«std::endl;

        //uint16_t ID = htons(this->MessageID);
        uint16_t ID = this->MessageID;
        memcpy(b, &ID, sizeof(ID));
        b += sizeof(ID);

        memcpy(b, Username.c_str(), Username.length());
        b[Username.length()] = '\0';
        b += Username.length() + 1;

        memcpy(b, DisplayName.c_str(), DisplayName.length());
        b[DisplayName.length()] = '\0';
        b += DisplayName.length() + 1;

        memcpy(b, Secret.c_str(), Secret.length());
        b[Secret.length()] = '\0';
        b += Secret.length() + 1;
        return sizeof(this->MessageType) + sizeof(ID) + Username.length() + 1 + DisplayName.length() +
1 +
               Secret.length() + 1;
    }

} AuthPacket;

typedef struct ReplyPackets : public Packets {
    std::string Message;
    uint8_t result;
    uint16_t ref_id;

    ReplyPackets(uint8_t type, uint16_t id, std::string mes, uint8_t res, uint16_t ref) :
Packets(type, id) {
        Message = std::move(mes);
        result = res;
        ref_id = ref;
    }

    int construct_message(uint8_t *b) override {
        memcpy(b, &this->MessageType, sizeof(this->MessageType));
        b += sizeof(this->MessageType);
        uint16_t ID = this->MessageID;
        memcpy(b, &ID, sizeof(ID));
        b += sizeof(ID);

        memcpy(b, &result, sizeof(result));
        b += sizeof(result);

        memcpy(b, &ref_id, sizeof(ref_id));
        b += sizeof(ref_id);

        memcpy(b, Message.c_str(), Message.length());
        b[Message.length()] = '\0';
        b += Message.length() + 1;

        return sizeof(this->MessageType) + sizeof(ID) + sizeof(result) + sizeof(ref_id) +
Message.length() + 1;
    }
} 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.

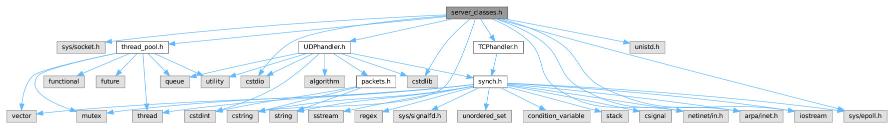
```
//
// Created by artem on 4/13/24.
//
#include "server_classes.h"

void UDPserver::Initialize(struct sockaddr_in *server_address) {
    if ((this->sockFD = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    if (bind(this->sockFD, (const struct sockaddr *) server_address, sizeof(*server_address)) < 0) {
        perror("binding failed udp");
        exit(EXIT_FAILURE);
    }
}

void UDPserver::Listen(ThreadPool *tp, std::stack<UserInfo> *s, synch *synch_variables, int signal_listener) {
    int epoll_fd = epoll_create1(0);
    if (epoll_fd < 0) {
        std::cerr « "Unable to create epoll instance\n";
        exit(EXIT_FAILURE);
    }

    epoll_event event;
    event.events = EPOLLIN | EPOLLET;
    event.data.fd = this->sockFD;

    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, this->sockFD, &event) < 0) {
        std::cerr « "Unable to add socket to epoll\n";
        exit(EXIT_FAILURE);
    }

    event.data.fd = signal_listener;
    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, signal_listener, &event) < 0) {
        std::cerr « "Unable to add socket to epoll\n";
        exit(EXIT_FAILURE);
    }

    struct epoll_event events[2];

    bool loop = true;
    while (loop) {
        int num_events = epoll_wait(epoll_fd, events, 2, -1); // 5 seconds timeout

        if (num_events < 0) {
            std::cerr « "Error in epoll_wait\n";
            exit(EXIT_FAILURE);
        }

        for (int i = 0; i < num_events; ++i) {
            if (events[i].data.fd == this->sockFD) {
                uint8_t buf[1024];
                sockaddr_in client_addr;
                if (!(events[i].events & EPOLLIN))
                    continue;

                socklen_t len = sizeof(client_addr);

                int n = recvfrom(this->sockFD, buf, 1024,
                                 0, (struct sockaddr *) &client_addr, &len);

                if (n == -1) {
                    std::cerr « "recvfrom failed. errno: " « errno « ’\n’;
                    continue;
                }

                tp->AddTask(std::bind(&UDPhandler::handleUDP, buf, client_addr, n, this->retransmissions, this->timeout,
                                      &tp->busy_threads, s, synch_variables, signal_listener));
            } else {
                loop = false;
                break;
            }
        }
    }
}

void UDPserver::Destroy() {
    close(this->sockFD);
}

void TCPserver::Initialize(struct sockaddr_in *server_address) {
    if ((this->sockFD = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    if (bind(this->sockFD, (const struct sockaddr *) server_address, sizeof(*server_address)) < 0) {
        perror("binding failed tcp");
        exit(EXIT_FAILURE);
    }
```

```
}

void TCPserver::Listen(ThreadPool *tp, std::stack<UserInfo> *s, synch *synch_variables, int signal_listener) {
    struct sockaddr_in client;
    listen(this->sockFD, 5);

    int epoll_fd = epoll_create1(0);
    if (epoll_fd < 0) {
        std::cerr « "Unable to create epoll instance\n";
        exit(EXIT_FAILURE);
    }

    epoll_event event;
    event.events = EPOLLIN | EPOLLET;
    event.data.fd = this->sockFD;

    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, this->sockFD, &event) < 0) {
        std::cerr « "Unable to add socket to epoll\n";
        exit(EXIT_FAILURE);
    }

    event.data.fd = signal_listener;
    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, signal_listener, &event) < 0) {
        std::cerr « "Unable to add socket to epoll\n";
        exit(EXIT_FAILURE);
    }

    struct epoll_event events[2];

    bool loop = true;
    while (loop) {
        int num_events = epoll_wait(epoll_fd, events, 2, -1); // 5 seconds timeout
        if (num_events < 0) {
            std::cerr « "Error in epoll_wait\n";
            exit(EXIT_FAILURE);
        }

        for (int i = 0; i < num_events; ++i) {
            if (events[i].data.fd == this->sockFD) {
                socklen_t len = sizeof(client);
                int clientSocket = accept(this->sockFD, (struct sockaddr *) &client, &len);
                if (clientSocket < 0) {
                    perror("accept failed");
                    continue;
                }
                tp->AddTask(
                        std::bind(&TCPhandler::handleTCP, clientSocket, &tp->busy_threads, s, synch_variables, client,
                                  signal_listener));
            } else {
                loop = false;
                break;
            }
        }
    }
}

void TCPserver::Destroy() {
    shutdown(this->sockFD, SHUT_RDWR);
    close(this->sockFD);
}
```

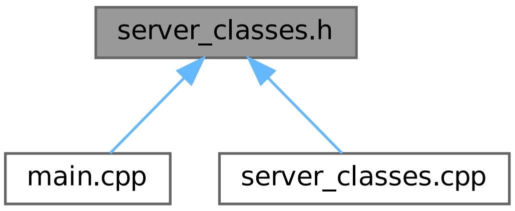
7.13 server_classes.h File Reference

```
#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.

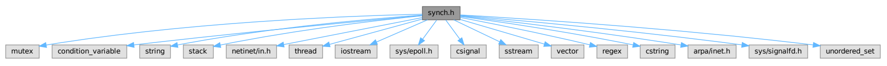
```
00001 //
00002 // Created by artem on 4/13/24.
00003 //
00004
00005 #ifndef IPK_SERVER_SERVER_CLASSES_H
00006 #define IPK_SERVER_SERVER_CLASSES_H
00007
00008 #include <sys/socket.h>
00009 #include <cstdio>
00010 #include <cstdlib>
00011 #include "netinet/in.h"
00012 #include <cstring>
00013 #include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

```
#include "TCPhandler.h"

#define PORT 47356
#endif //IPK_SERVER_SERVER_CLASSES_H

class Server {
public:
    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;
};

class TCPserver : public Server {
public:

    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:
    int sockFD;
};

class UDPserver : public Server {
public:
    int retransmissions;
    int timeout;

    UDPserver(int ret, int t) {
        this->retransmissions = ret;
        this->timeout = 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();

private:
    int sockFD;
};

```

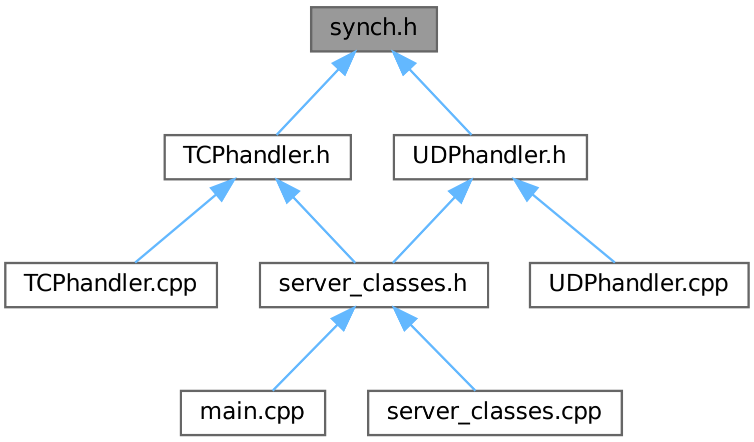
7.15 synch.h File Reference

```
#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

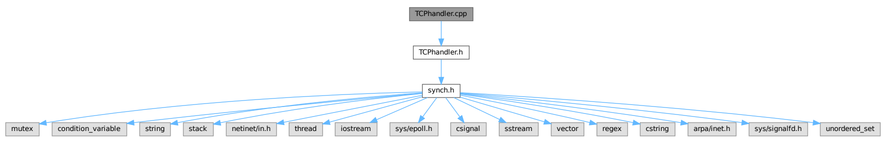
Go to the documentation of this file.

```
//
// Created by artem on 4/20/24.
//

#ifndef IPK_SERVER_SYNCH_H

#define IPK_SERVER_SYNCH_H

#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>

struct synch {
    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;

    explicit synch(int b) : finished(b), ready(false){};
};

struct UserInfo {
    sockaddr_in client;
    int tcp_socket;
    uint8_t *buf;
    int length;
    std::string channel;
    bool tcp;


    UserInfo(sockaddr_in c, uint8_t *m, int l, std::string name, bool t, int cs) : client(c), buf(m),
length(l),

channel(std::move(name)), tcp(t),
                                                                                tcp_socket(cs) {};
};


#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()

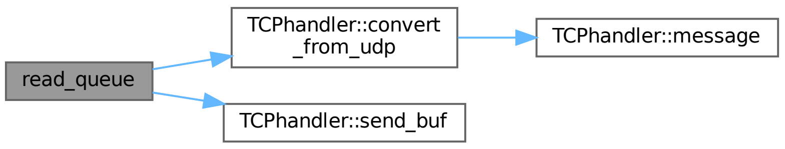
```
void read_queue (
            std::stack< UserInfo > * s,
            bool * terminate,
            synch * synch_vars,
            int * busy,
            TCPhandler * tcp )
```

Definition at line 45 of file TCPhandler.cpp.

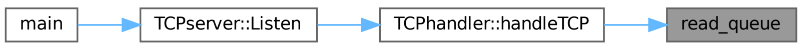
```
00045
      {
00046     while (!*terminate) {
00047         std::unique_lock<std::mutex> lock(synch_vars->mtx);
00048         synch_vars->cv.wait(lock, [&synch_vars] { return synch_vars->ready; });
00049
00050         synch_vars->waiting.lock();
00051         synch_vars->finished++;
00052       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

```
            synch_vars->waiting.unlock();

            if (!new_uf.tcp) {
                if (new_uf.channel == tcp->channel_name) {
                    uint8_t buf[3048];
                    int length = tcp->convert_from_udp(buf, new_uf.buf);
                    tcp->send_buf(buf, length);
                }
            } else {
                if (new_uf.tcp_socket != tcp->client_socket && new_uf.channel == tcp->channel_name) {
                    tcp->send_buf(new_uf.buf, new_uf.length);
                }
            }
        }

        if (synch_vars->finished == *busy) {
            synch_vars->finished = 0;
            synch_vars->ready = false;
            if (!s->empty())
                s->pop();
        }

        lock.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        lock.lock();

    }
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



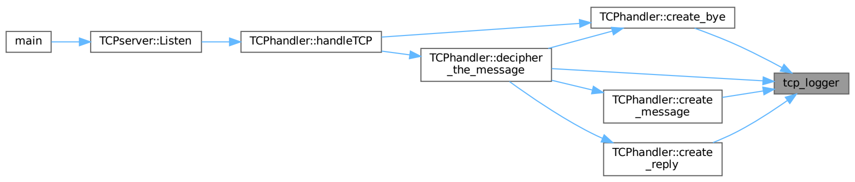
7.17.1.2 tcp_logger()

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

Definition at line 309 of file TCPhandler.cpp.

```
                                                                        {
    std::cout « operation « " " « inet_ntoa(client.sin_addr) « ":" « ntohs(client.sin_port) « " | " «
      type
              « std::endl;
}
```

Here is the caller graph for this function:



7.18 TCPhandler.cpp

Go to the documentation of this file.

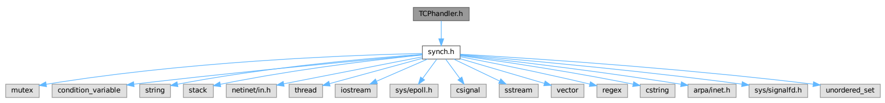
```
//
// Created by artem on 4/20/24.
//
#include "TCPhandler.h"

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

    TCPhandler tcp(client_socket, client, signal_listener);

    bool end = false;

    std::thread sender(read_queue, s, &end, synch_var, busy, &tcp);

    uint8_t internal_buf[2048];

    while (true) {
        int length = tcp.listening_for_incoming_connection(internal_buf, 1024);
        if (length == 0)
            break;
        if (length == -1) {
            tcp.create_bye();
            break;
        }
        if (!tcp.decipher_the_message(internal_buf, length, s, synch_var))
            break;
    }

    if (synch_var->usernames.find(tcp.user_n) != synch_var->usernames.end())
        synch_var->usernames.erase(tcp.user_n);

    end = true;
    {
        std::lock_guard<std::mutex> lock(synch_var->mtx);
        synch_var->ready = true;
    }
    synch_var->cv.notify_all();

    sender.join();
    shutdown(tcp.client_socket, SHUT_RDWR);
    close(tcp.client_socket);
}

void read_queue(std::stack<UserInfo> *s, bool *terminate, synch *synch_vars, int *busy, TCPhandler
*tcp) {
    while (!*terminate) {
        std::unique_lock<std::mutex> lock(synch_vars->mtx);
        synch_vars->cv.wait(lock, [&synch_vars] { return synch_vars->ready; });

        synch_vars->waiting.lock();
        synch_vars->finished++;
        synch_vars->waiting.unlock();

        if (!s->empty() && tcp->auth) {

            synch_vars->waiting.lock();
            UserInfo new_uf = s->top();
            synch_vars->waiting.unlock();

            if (!new_uf.tcp) {
                if (new_uf.channel == tcp->channel_name) {
                    uint8_t buf[3048];
                    int length = tcp->convert_from_udp(buf, new_uf.buf);
                    tcp->send_buf(buf, length);
                }
            } else {
                if (new_uf.tcp_socket != tcp->client_socket && new_uf.channel == tcp->channel_name) {
                    tcp->send_buf(new_uf.buf, new_uf.length);
                }
            }
        }

        if (synch_vars->finished == *busy) {
            synch_vars->finished = 0;
            synch_vars->ready = false;
            if (!s->empty())
                s->pop();
        }

        lock.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        lock.lock();

    }
}

int TCPhandler::listening_for_incoming_connection(uint8_t *buf, int len) {

    int event_count = epoll_wait(this->epoll_fd, this->events, 2, -1);

    if (event_count == -1) {
        perror("epoll_wait");
        close(this->epoll_fd);
        exit(EXIT_FAILURE);
    } else if (event_count > 0) {
        for (int j = 0; j < event_count; j++) {
            if (events[j].data.fd == this->client_socket) { // check if EPOLLIN event has occurred
                int n = recv(this->client_socket, buf, len, 0);
                if (n == -1) {
                    std::cerr « "recvfrom failed. errno: " « errno « '\n';
                    continue;
                } else if (n == 0) {
                    return 0;
                } else if (n > 0) {
                    return n;
                }
            } else {
                return -1;
            }
        }
    }
    return 0;
}

bool TCPhandler::decipher_the_message(uint8_t *buf, int length, std::stack<UserInfo> *s, synch
*synch_var) {
    std::string out_str;
    for (int i = 0; i < length - 2; ++i) {
        out_str += static_cast<char>(buf[i]);
    }

    std::istringstream iss(out_str);
    std::vector<std::string> result;
    for (std::string element; std::getline(iss, element, ' ');) {
        result.push_back(element);
    }

    if (!this->auth) {
        if (result[0] != "AUTH") {
            if (result[0] != "BYE") {
                if (result[0] != "ERR") {
                    this->create_message(true, "You should log-in before doing anything else");
                    return true;
                }
            }
        }
    }

    if (result[0] == "AUTH") {
        std::regex e("^AUTH .{1,20} AS .{1,20} USING .{1,128}$");
        if (!std::regex_match(out_str, e)) {
            std::string mes = "Wrong AUTH format";
            std::cout « mes « std::endl;
            create_message(true, "Wrong AUTH format");
            std::this_thread::sleep_for(std::chrono::milliseconds(10));
            this->create_bye();
            return false;
        }
        synch_var->un.lock();
        bool exists = username_already_exists(result[1], synch_var);
        synch_var->un.unlock();
        tcp_logger(this->client_addr, "AUTH", "RECV");
        if (exists) {
            this->create_reply("NOK", "Username already exists");
        } else {
            this->create_reply("OK", "Authentication is successful");
            this->display_name = result[3];
            this->user_changed_channel(s, synch_var, "joined");
            this->auth = true;
        }

    } else if (result[0] == "MSG") {
        std::regex e("^MSG FROM .{1,20} IS .{1,1400}$");
        if (!std::regex_match(out_str, e)) {
            std::string mes = "Wrong MSG format";
            std::cout « mes « std::endl;
            create_message(true, "Wrong MSG format");
            std::this_thread::sleep_for(std::chrono::milliseconds(10));
            this->create_bye();
            return false;
        }
        tcp_logger(this->client_addr, "MSG", "RECV");
        this->display_name = result[2];
        this->message(buf, length, s, synch_var, this->channel_name);
    } else if (result[0] == "JOIN") {
        std::regex e("^JOIN .{1,20} AS .{1,20}$");
        if (!std::regex_match(out_str, e)) {
            std::string mes = "Wrong JOIN format";
            create_message(true, mes.c_str());
            create_message(true, "Wrong JOIN format");
            std::this_thread::sleep_for(std::chrono::milliseconds(10));
            this->create_bye();
            return false;
        }
        if (result[1] != this->channel_name) {
            tcp_logger(this->client_addr, "JOIN", "RECV");
            this->user_changed_channel(s, synch_var, "left");
            std::this_thread::sleep_for(std::chrono::milliseconds(30));
            this->channel_name = result[1];
            this->display_name = result[3];
            this->user_changed_channel(s, synch_var, "joined");
            this->create_reply("OK", "Join was successful");
        } else {
            this->create_reply("NOK", "Tried to join to the current channel");
        }
    } else if (result[0] == "BYE") {
        tcp_logger(this->client_addr, "BYE", "RECV");
        if (this->auth) {
            user_changed_channel(s, synch_var, "left");
        }
        return false;
    } else if (result[0] == "ERR") {
        tcp_logger(this->client_addr, "ERR", "RECV");
        if (this->auth) {
            user_changed_channel(s, synch_var, "left");
        }
        this->create_bye();
        return false;
    } else {
        tcp_logger(this->client_addr, "UNDEFINED", "RECV");
        this->create_message(true, "Unknown command");
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
        this->create_bye();
        return false;
    }

    return true;
}

void TCPhandler::message(uint8_t *buf, int message_length, std::stack<UserInfo> *s, synch *synch_var,
                         std::string &channel) {
    struct sockaddr_in blank;
    {
        std::lock_guard<std::mutex> lock(synch_var->mtx);
        s->emplace(blank, buf, message_length, channel, true, this->client_socket);
        synch_var->ready = true;
    }
    synch_var->cv.notify_all();
}

void TCPhandler::create_reply(const char *status, const char *msg) {
    std::string message;
    message = "REPLY " + std::string(status) + " IS " + std::string(msg) + "\r\n";
    tcp_logger(this->client_addr, "REPLY", "SENT");
    this->send_string(message);
}
```

```cpp

void TCPhandler::create_message(bool error, const char *msg) {
    std::string message;
    error ? message = "ERR FROM SERVER IS " + std::string(msg) + "\r\n" : message = "MSG FROM SERVER
IS " +
                                                                                        std::string(msg) +
"\r\n";
    tcp_logger(this->client_addr, "MSG", "SENT");
    this->send_string(message);
}

void TCPhandler::create_bye() {
    std::string bye = "BYE\r\n";
    tcp_logger(this->client_addr, "BYE", "SENT");
    this->send_string(bye);
}

void TCPhandler::send_string(std::string &msg) const {
    const char *message = msg.c_str();
    size_t bytes_left = msg.size();

    ssize_t tx = send(this->client_socket, message, bytes_left, 0);

    if (tx < 0) {
        perror("Error sending message");
    }
}

void TCPhandler::send_buf(uint8_t *buf, int length) const {
    ssize_t tx = send(this->client_socket, buf, length, 0);

    if (tx < 0) {
        perror("Error sending message");
    }
}

void TCPhandler::user_changed_channel(std::stack<UserInfo> *s, synch *synch_var, const char *action) {

    std::stringstream ss;
    ss « this->display_name « " has " « std::string(action) « " " « this->channel_name « ".";
    std::string content = ss.str();

    std::string message = "MSG FROM Server IS " + content + "\r\n";

    uint8_t buffer[1024];

    memcpy(buffer, message.c_str(), message.length());

    this->message(buffer, message.length(), s, synch_var, this->channel_name);
}

int TCPhandler::convert_from_udp(uint8_t *buf, uint8_t *udp_buf) {
    int i = 3;
    std::string display_n;
    std::string contents;

    while (udp_buf[i] != 0x00) {
        display_n.push_back(static_cast<char>(udp_buf[i]));
        i++;
    }

    i++;

    while (udp_buf[i] != 0x00) {
        contents.push_back(static_cast<char>(udp_buf[i]));
        i++;
    }

    std::string message = "MSG FROM " + display_n + " IS " + contents + "\r\n";

    memcpy(buf, message.c_str(), message.length());

    return message.length();
}

void tcp_logger(sockaddr_in client, const char *type, const char *operation) {
    std::cout « operation « " " « inet_ntoa(client.sin_addr) « ":" « ntohs(client.sin_port) « " | " «
type
                « std::endl;
}

bool TCPhandler::username_already_exists(std::string &username, synch *synch_vars) {
    if (!synch_vars->usernames.empty()) {
        if (synch_vars->usernames.find(username) != synch_vars->usernames.end())
            return true;
    }
    synch_vars->usernames.insert(username);
00320     this->user_n = username;
00321     return false;
00322 }
```

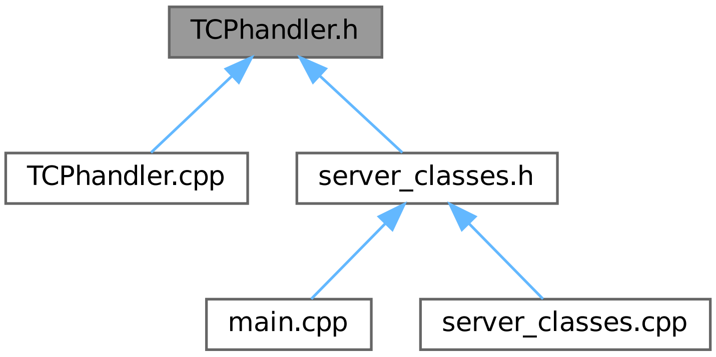
7.19 TCPhandler.h File Reference

```
#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()

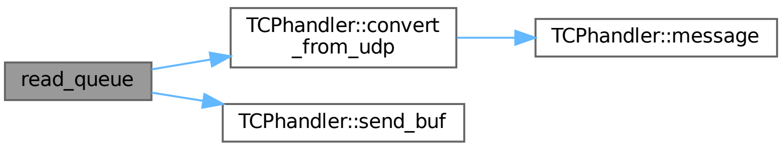
```
void read_queue (
            std::stack< UserInfo > ∗ s,
            bool ∗ terminate,
            synch ∗ synch_vars,
            int ∗ busy,
            TCPhandler ∗ tcp )
```

Definition at line 45 of file TCPhandler.cpp.

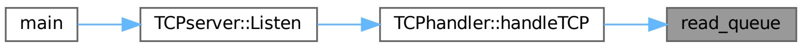
```
00045
      {
```

```
    while (!*terminate) {
        std::unique_lock<std::mutex> lock(synch_vars->mtx);
        synch_vars->cv.wait(lock, [&synch_vars] { return synch_vars->ready; });

        synch_vars->waiting.lock();
        synch_vars->finished++;
        synch_vars->waiting.unlock();

        if (!s->empty() && tcp->auth) {

            synch_vars->waiting.lock();
            UserInfo new_uf = s->top();
            synch_vars->waiting.unlock();

            if (!new_uf.tcp) {
                if (new_uf.channel == tcp->channel_name) {
                    uint8_t buf[3048];
                    int length = tcp->convert_from_udp(buf, new_uf.buf);
                    tcp->send_buf(buf, length);
                }
            } else {
                if (new_uf.tcp_socket != tcp->client_socket && new_uf.channel == tcp->channel_name) {
                    tcp->send_buf(new_uf.buf, new_uf.length);
                }
            }
        }

        if (synch_vars->finished == *busy) {
            synch_vars->finished = 0;
            synch_vars->ready = false;
            if (!s->empty())
                s->pop();
        }

        lock.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        lock.lock();

    }
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



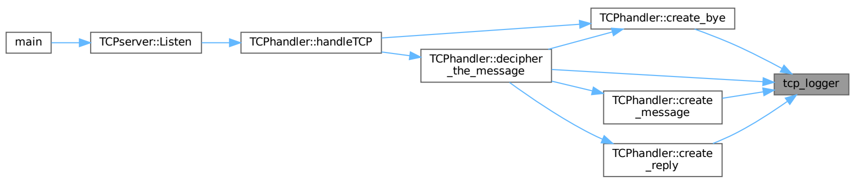
7.19.1.2 tcp_logger()

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

Definition at line 309 of file TCPhandler.cpp.

```
{
    std::cout « operation « " " « inet_ntoa(client.sin_addr) « ":" « ntohs(client.sin_port) « " | " « type
                « std::endl;
}
```

Here is the caller graph for this function:



7.20 TCPhandler.h

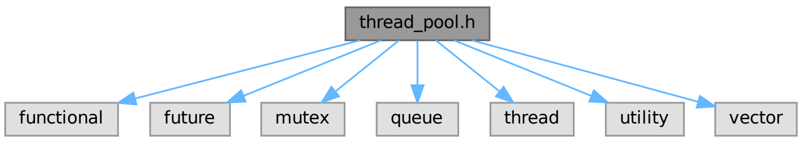
Go to the documentation of this file.

```
//
// Created by artem on 4/20/24.
//

#ifndef IPK_SERVER_TCPHANDLER_H
#define IPK_SERVER_TCPHANDLER_H

#include "synch.h"

class TCPhandler {
public:

    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;

    TCPhandler(int s, sockaddr_in c, int kill) {
        this->channel_name = "general";
        this->client_socket = s;

        epoll_fd = epoll_create1(0);
        if (epoll_fd == -1) {
            std::cerr « "Failed to create epoll file descriptor\n";
            exit(EXIT_FAILURE);
        }

        // setup epoll event
        struct epoll_event ev;
        ev.events = EPOLLIN;
        ev.data.fd = this->client_socket;

        // add socket file descriptor to epoll
        if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, this->client_socket, &ev) == -1) {
            std::cerr « "Failed to add file descriptor to epoll\n";
            close(epoll_fd);
            exit(EXIT_FAILURE);
        }

        ev.data.fd = kill;
        if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, kill, &ev) < 0) {
            std::cerr « "Unable to add socket to epoll\n";
            exit(EXIT_FAILURE);
        }

        client_addr = c;

        auth = false;
    }

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

    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);

private:
    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);

};

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);



#endif //IPK_SERVER_TCPHANDLER_H
```

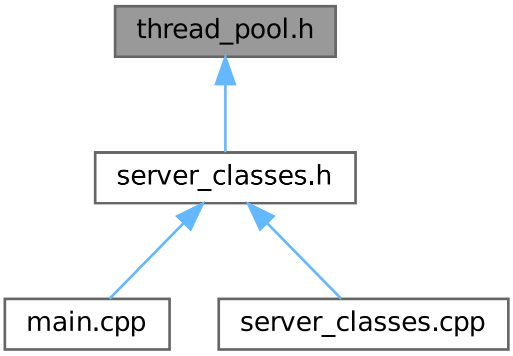
7.21 thread_pool.h File Reference

```
#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

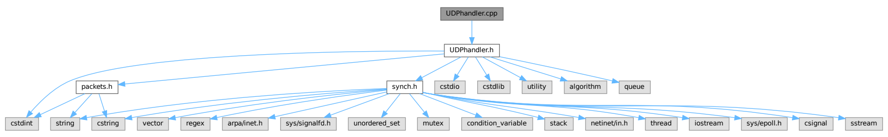
7.22 thread_pool.h

Go to the documentation of this file.

```
#pragma once

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

class ThreadPool {
public:
    ThreadPool(const int size) : busy_threads(size), threads(std::vector<std::thread>(size)),
                                 shutdown_requested(false) {
        for (size_t i = 0; i < size; ++i) {
            threads[i] = std::thread(ThreadWorker(this));
        }
    }

    ~ThreadPool() {
        Shutdown();
    }

    ThreadPool(const ThreadPool &) = delete;

    ThreadPool(ThreadPool &&) = delete;

    ThreadPool &operator=(const ThreadPool &) = delete;

    ThreadPool &operator=(ThreadPool &&) = delete;

    // Waits until threads finish their current task and shutdowns the pool
    void Shutdown() {
        {
            std::lock_guard<std::mutex> lock(mutex);
            shutdown_requested = true;
            condition_variable.notify_all();
        }

        for (size_t i = 0; i < threads.size(); ++i) {
            if (threads[i].joinable()) {
                threads[i].join();
            }
        }
    }

    template<typename F, typename... Args>
    auto AddTask(F &&f, Args &&... args) -> std::future<decltype(f(args...))> {

        auto task_ptr = std::make_shared<std::packaged_task<decltype(f(args...))()»(
                std::bind(std::forward<F>(f), std::forward<Args>(args)...));

        auto wrapper_func = [task_ptr]() { (*task_ptr)(); };
        {
            std::lock_guard<std::mutex> lock(mutex);
            queue.push(wrapper_func);
            // Wake up one thread if its waiting
            condition_variable.notify_one();
        }

        // Return future from promise
        return task_ptr->get_future();
    }

    int QueueSize() {
        std::unique_lock<std::mutex> lock(mutex);
        return queue.size();
    }

private:
    class ThreadWorker {
    public:
        ThreadWorker(ThreadPool *pool) : thread_pool(pool) {
        }

        void operator()() {
            std::unique_lock<std::mutex> lock(thread_pool->mutex);
            while (!thread_pool->shutdown_requested ||
                   (thread_pool->shutdown_requested && !thread_pool->queue.empty())) {
                thread_pool->busy_threads--;
                thread_pool->condition_variable.wait(lock, [this] {
                    return this->thread_pool->shutdown_requested || !this->thread_pool->queue.empty();
                });
                thread_pool->busy_threads++;

                if (!this->thread_pool->queue.empty()) {

                    auto func = thread_pool->queue.front();

                    thread_pool->queue.pop();

                    lock.unlock();
                    func();
                    lock.lock();
                }
            }
        }

    private:
        ThreadPool *thread_pool;
    };

public:
    int busy_threads;

private:
    mutable std::mutex mutex;
    std::condition_variable condition_variable;

    std::vector<std::thread> threads;
    bool shutdown_requested;

    std::queue<std::function<void()» queue;
};
```

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

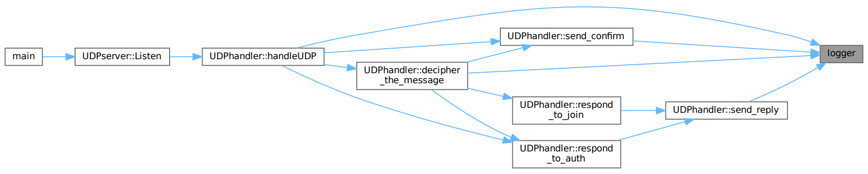
7.23.1.1 logger()

```
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()

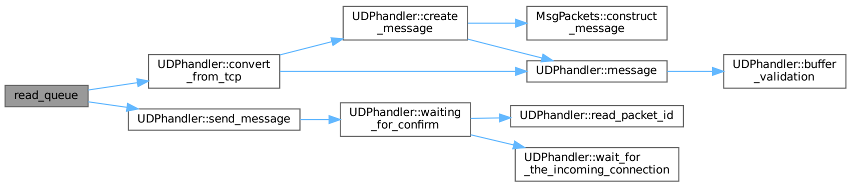
```
void read_queue (
            std::stack< UserInfo > * s,
            bool * terminate,
            synch * synch_vars,
            int * busy,
            UDPhandler * udp )
```

Definition at line 49 of file UDPhandler.cpp.

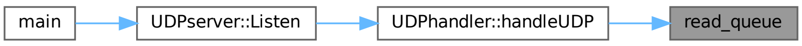
```
00049
      {
00050     while (!*terminate) {
00051         std::unique_lock<std::mutex> lock(synch_vars->mtx);
00052         synch_vars->cv.wait(lock, [&synch_vars] { return synch_vars->ready; });
00053
00054         synch_vars->waiting.lock();
00055         synch_vars->finished++;
00056       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

```
        if (!s->empty() && udp->auth) {

            synch_vars->waiting.lock();
            UserInfo new_uf = s->top();
            synch_vars->waiting.unlock();

            if (new_uf.tcp) {
                if (new_uf.channel == udp->channel_name) {
                    uint8_t buf[3048];
                    int length = udp->convert_from_tcp(buf, new_uf.buf);
                    udp->send_message(buf, length, false);
                }
            } else {
                if ((new_uf.client.sin_port != udp->client_addr.sin_port && new_uf.channel == udp->channel_name)) {
                    udp->send_message(new_uf.buf, new_uf.length, false);
                }
            }
        }
        if (synch_vars->finished == *busy) {
            synch_vars->finished = 0;
            synch_vars->ready = false;
            if (!s->empty())
                s->pop();
        }
        lock.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        lock.lock();

    }
}
```

Here is the call graph for this function:


UDPhandler::create _message
MsgPackets::construct _message
UDPhandler::convert _from_tcp
UDPhandler::message
UDPhandler::buffer _validation
read_queue
UDPhandler::send_message
UDPhandler::waiting _for_confirm
UDPhandler::read_packet_id
UDPhandler::wait_for _the_incoming_connection

Here is the caller graph for this function:



7.24 UDPhandler.cpp

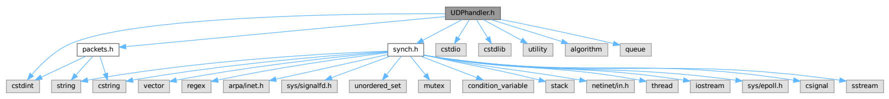
Go to the documentation of this file.

```
//
// Created by artem on 4/14/24.
//
#include "UDPhandler.h"

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) {

    UDPhandler udp(retransmissions, timeout, client_addr, signal_listener);

    bool end = false;

    std::thread sender(read_queue, s, &end, synch_var, busy, &udp);

    uint8_t internal_buf[2048];
    logger(udp.client_addr, "AUTH", "RECV");
    udp.send_confirm(buf);
    int result = udp.respond_to_auth(buf, length, s, synch_var);

    if (result != -1) {
        while (true) {
            int length_internal = udp.wait_for_the_incoming_connection(internal_buf);
            if (length_internal == -1) {
                uint8_t buf_int[256];
                int length_int = udp.create_bye(buf_int);
                udp.send_message(buf_int, length_int, true);
                break;
            }
            if (!udp.decipher_the_message(internal_buf, length_internal, s, synch_var)) {
                break;
            }
        }
    }

    if (synch_var->usernames.find(udp.user_n) != synch_var->usernames.end())
        synch_var->usernames.erase(udp.user_n);

    end = true;
    {
        std::lock_guard<std::mutex> lock(synch_var->mtx);
        synch_var->ready = true;
    }
    synch_var->cv.notify_all();

    sender.join();
    close(udp.client_socket);
}

void read_queue(std::stack<UserInfo> *s, bool *terminate, synch *synch_vars, int *busy, UDPhandler *udp) {
    while (!*terminate) {
        std::unique_lock<std::mutex> lock(synch_vars->mtx);
        synch_vars->cv.wait(lock, [&synch_vars] { return synch_vars->ready; });

        synch_vars->waiting.lock();
        synch_vars->finished++;
        synch_vars->waiting.unlock();

        if (!s->empty() && udp->auth) {

            synch_vars->waiting.lock();
            UserInfo new_uf = s->top();
            synch_vars->waiting.unlock();

            if (new_uf.tcp) {
                if (new_uf.channel == udp->channel_name) {
                    uint8_t buf[3048];
                    int length = udp->convert_from_tcp(buf, new_uf.buf);
                    udp->send_message(buf, length, false);
                }
            } else {
                if ((new_uf.client.sin_port != udp->client_addr.sin_port && new_uf.channel == udp->channel_name)) {
                    udp->send_message(new_uf.buf, new_uf.length, false);
                }
            }
        }
        if (synch_vars->finished == *busy) {
            synch_vars->finished = 0;
            synch_vars->ready = false;
            if (!s->empty())
                s->pop();
        }
        lock.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        lock.lock();

    }
}

bool UDPhandler::decipher_the_message(uint8_t *buf, int length, std::stack<UserInfo> *s, synch *synch_var) {
    if (!this->auth) {
        if (buf[0] != 0x02) {
            if (buf[0] != 0xFF) {
                if (buf[0] != 0xFE) {
                    uint8_t buf_int[1024];
                    std::string message = "You should log-in before doing anything else";
                    std::string name = "Server";
                    int length = this->create_message(buf_int, message, false, name);
                    this->send_message(buf_int, length, false);
                    return true;
                }
            }
        }
    }

    switch (buf[0]) {
        case 0x00://CONFIRM
            break;
        case 0x02://AUTH
            logger(this->client_addr, "AUTH", "RECV");
            send_confirm(buf);
            if (!this->auth) {
                respond_to_auth(buf, length, s, synch_var);
            } else {
                uint8_t buf_err[1024];
                std::string message = "Already authed";
                std::string name = "Server";
                int length_err = this->create_message(buf_err, message, true, name);
                this->send_message(buf_err, length_err, false);
            }
            break;
        case 0x03://JOIN
            logger(this->client_addr, "JOIN", "RECV");
            send_confirm(buf);
            respond_to_join(buf, length, s, synch_var);
            break;
        case 0x04://MSG
            logger(this->client_addr, "MSG", "RECV");
            send_confirm(buf);
            this->message(buf, length, s, synch_var, this->channel_name);
            break;
        case 0xFF://BYE
            if (this->auth) {
                this->client_leaving(s, synch_var);
            }
            logger(this->client_addr, "BYE", "RECV");
            send_confirm(buf);
            return false;
        case 0xFE://ERR
            if (this->auth) {
                this->client_leaving(s, synch_var);
            }
            logger(this->client_addr, "ERR", "RECV");
            send_confirm(buf);
            return false;
        default:
            uint8_t buf[1024];
            std::string message = "Unknown instruction";
            std::string name = "Server";
            int length_err = this->create_message(buf, message, true, name);
            this->send_message(buf, length_err, false);
            return false;
    }
    return true;
}

int UDPhandler::respond_to_auth(uint8_t *buf, int message_length, std::stack<UserInfo> *s, synch
    *synch_var) {

    bool valid_message = true;

    if (buf[0] == 0xFF) {
        return -1;
    }
    if (buf[0] != 0x02) {
        uint8_t buf_int[1024];
        std::string message = "You should log-in before doing anything else";
        std::string name = "Server";
        int length = this->create_message(buf_int, message, false, name);
        this->send_message(buf_int, length, false);
        return 0;
    }

    if (!this->buffer_validation(buf, message_length, 3, 7, 3))
        valid_message = false;

    if (valid_message) {
        synch_var->un.lock();
        bool exists = username_exists(buf, synch_var);
        synch_var->un.unlock();
        if (!exists) {
            this->change_display_name(buf, true);
            std::string success = "Authentication is succesful";
            send_reply(buf, success, true);

            std::stringstream ss;
            ss « this->display_name « " has joined general.";
            std::string message = ss.str();
            uint8_t buf_message[1024];
            std::string name = "Server";
            int length = this->create_message(buf_message, message, false, name);
            this->message(buf_message, length, s, synch_var, this->channel_name);
            this->auth = true;
        } else {
            std::string failure = "Username already exists";
            send_reply(buf, failure, false);
        }
    } else {
        std::string failure = "Authentication is not succesful";
        send_reply(buf, failure, false);
    }

    return 0;
}

void UDPhandler::respond_to_join(uint8_t *buf, int message_length, std::stack<UserInfo> *s, synch
*synch_var) {
    bool valid = true;

    if (!this->buffer_validation(buf, message_length, 3, 2))
        valid = false;

    if (valid) {
        this->change_display_name(buf, true);
        std::string success = "Join is succesful";
        send_reply(buf, success, true);

        std::stringstream ss;
        ss « this->display_name « " has left " « this->channel_name « ".";
        std::string message = ss.str();
        uint8_t buf_message[1024];
        std::string name = "Server";
        int length = this->create_message(buf_message, message, false, name);
        this->message(buf_message, length, s, synch_var, this->channel_name);

        std::this_thread::sleep_for(std::chrono::milliseconds(10));

        memset(buf_message, 0, 1024);
        std::stringstream joined;
        this->channel_name = this->read_channel_name(buf);
        joined « this->display_name « " has joined " « this->channel_name « ".";
        std::string message_new = joined.str();
        length = this->create_message(buf_message, message_new, false, name);
        this->message(buf_message, length, s, synch_var, this->channel_name);

    } else {
        std::string failure = "Join is not succesful";
        send_reply(buf, failure, false);
    }
}

void UDPhandler::message(uint8_t *buf, int message_length, std::stack<UserInfo> *s, synch *synch_var,
                         std::string &channel) {
    bool valid_message = true;

    if (!this->buffer_validation(buf, message_length, 3, 2, 2, 20, 1400))
        valid_message = false;

    if (valid_message) {
        {
            std::lock_guard<std::mutex> lock(synch_var->mtx);
            s->emplace(this->client_addr, buf, message_length, channel, false, 0);
            synch_var->ready = true;
        }
        synch_var->cv.notify_all();
    } else {
        std::cout « "Invalid message" « std::endl;
    }
}

void UDPhandler::client_leaving(std::stack<UserInfo> *s, synch *synch_var) {
    std::stringstream ss;
    ss « this->display_name « " has left " « this->channel_name « ".";
    std::string message = ss.str();
    uint8_t buf_message[1024];
    std::string name = "Server";
    int length = this->create_message(buf_message, message, false, name);
    this->message(buf_message, length, s, synch_var, this->channel_name);
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
}


void UDPhandler::send_confirm(uint8_t *buf) {
    uint8_t buf_out[4];

    ConfirmPacket confirm(0x00, this->global_counter, read_packet_id(buf));

    int len = confirm.construct_message(buf_out);

    socklen_t address_size = sizeof(this->client_addr);

    long bytes_tx = sendto(this->client_socket, buf_out, len, 0, (struct sockaddr *)
&(this->client_addr),
                           address_size);
    if (bytes_tx < 0) perror("ERROR: sendto");

    logger(this->client_addr, "CONFIRM", "SENT");

}

void UDPhandler::send_reply(uint8_t *buf, std::string &message, bool OK) {
    uint8_t buf_out[1024];

    socklen_t address_size = sizeof(this->client_addr);

    ReplyPacket reply(0x01, this->global_counter, message, OK ? 1 : 0, read_packet_id(buf));
    this->global_counter++;

    int len = reply.construct_message(buf_out);
    long bytes_tx = sendto(this->client_socket, buf_out, len, 0, (struct sockaddr *)
&(this->client_addr),
                           address_size);
    if (bytes_tx < 0) perror("ERROR: sendto");

    if (!waiting_for_confirm(buf_out, len))
        std::cout « "Client didn’t confirm" « std::endl;

    logger(this->client_addr, "REPLY", "SENT");
}

void UDPhandler::send_message(uint8_t *buf, int message_length, bool terminate) {
    socklen_t address_size = sizeof(this->client_addr);

    sockaddr_in backup = this->client_addr;

    sendto(this->client_socket, buf, message_length, 0, (struct sockaddr *) &this->client_addr,
address_size);
    this->global_counter++;

    if (!terminate) {
        if (!waiting_for_confirm(buf, message_length))
            std::cout « "Client didn’t confirm" « std::endl;
    }
    this->client_addr = backup;
}

int UDPhandler::create_message(uint8_t *buf_out, std::string &msg, bool error, std::string &name) {
    MsgPacket message(error ? 0xFE : 0x04, this->global_counter, msg, name);
    return message.construct_message(buf_out);
}

int UDPhandler::create_bye(uint8_t *buf) {
    Packet bye(0xFF, this->global_counter);
    return bye.construct_message(buf);
}

int UDPhandler::read_packet_id(uint8_t *buf) {
    int result = buf[1] « 8 | buf[2];
    return ntohs(result);
}

int UDPhandler::wait_for_the_incoming_connection(uint8_t *buf_out, int timeout) {
    int event_count = epoll_wait(this->epoll_fd, this->events, 2, timeout);

    if (event_count == -1) {
        perror("epoll_wait");
        close(this->epoll_fd);
        exit(EXIT_FAILURE);
    } else if (event_count > 0) {
        socklen_t len_client = sizeof(this->client_addr);
        for (int j = 0; j < event_count; j++) {
            if (events[j].data.fd == this->client_socket) { // check if EPOLLIN event has occurred
                int n = recvfrom(this->client_socket, buf_out, 1024, 0, (struct sockaddr *)
&this->client_addr,
                                 &len_client);
                if (n == -1) {
                    std::cerr « "recvfrom failed. errno: " « errno « ’\n’;
                    continue;
                }
                if (n > 0) {
                    return n;
                }
            } else {
                return -1;
            }
        }
    }
    return 0;
}

bool UDPhandler::waiting_for_confirm(uint8_t *buf, int len) {
    uint8_t buffer[1024];
    bool confirmed = false;
    for (int i = 0; i < this->retransmissions; ++i) {
        int result = this->wait_for_the_incoming_connection(buffer, this->timeout_chat);
        if (result > 0) {
            if (buffer[0] == 0x00 && read_packet_id(buffer) == read_packet_id(buf)) {
                confirmed = true;
            }
        } else if (result == -1) {
            return true;
        }
        if (confirmed) {
            break;
        } else {
            socklen_t len_client = sizeof(client_addr);
            long bytes_tx = sendto(this->client_socket, buf, len, 0, (struct sockaddr *)
&(this->client_addr),
                                   len_client);
            if (bytes_tx < 0) perror("ERROR: sendto");
        }
    }
    return confirmed;
}

bool UDPhandler::buffer_validation(uint8_t *buf, int message_length, int start_position, int
minimal_length,
                                   int amount_of_fields, int first_limit, int second_limit, int
third_limit) {

    if (message_length < minimal_length)
        return false;

    size_t i = start_position;

    size_t count = 0;
    while (i < message_length && buf[i] != 0x00 && count < first_limit) {
        i++;
        count++;
    }

    if (i >= message_length || buf[i] != 0x00 || count < 1) {
        return false;
    }
    ++i;

    count = 0;
    while (i < message_length && buf[i] != 0x00 && count < second_limit) {
        ++i;
        ++count;
    }

    if (i >= message_length || buf[i] != 0x00 || count < 1) {
        return false;
    }

    ++i;

    if (amount_of_fields == 3) {
        count = 0;
        while (i < message_length && buf[i] != 0x00 && count < third_limit) {
            ++i;
            ++count;
        }

        if (i >= message_length || buf[i] != 0x00 || count < third_limit) {
            return false;
        }
    }

    return true;
}

std::string UDPhandler::read_channel_name(uint8_t *buf) {
    int i = 3;
    std::string channel;
    while (buf[i] != 0x00) {
        channel.push_back(static_cast<char>(buf[i]));
        i++;
    }
    return channel;
}

void UDPhandler::change_display_name(uint8_t *buf, bool second) {
    this->display_name.clear();
    int i = 3;
    if (second) {
        while (buf[i] != 0x00)
            i++;
    }
    i++;
    while (buf[i] != 0x00) {
        this->display_name.push_back(static_cast<char>(buf[i]));
        i++;
    }
}

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

    std::string message;
    int i = 0;
    while (tcp_buf[i] != 0x0d) {
        message.push_back(static_cast<char>(tcp_buf[i]));
        i++;
    }

    std::regex patternFromToIs(R"(FROM\s(.*?)\sIS)");
    std::smatch matchFromToIs;
    std::regex_search(message, matchFromToIs, patternFromToIs);
    std::string name = matchFromToIs[1].str();

    std::regex patternAfterIs(R"(IS\s(.*))");
    std::smatch matchAfterIs;
    std::regex_search(message, matchAfterIs, patternAfterIs);
    std::string msg = matchAfterIs[1].str();

    return this->create_message(buf, msg, false, name);
}

void logger(sockaddr_in client, const char *type, const char *operation) {
    std::cout « operation « " " « inet_ntoa(client.sin_addr) « ":" « ntohs(client.sin_port) « " | " «
type
              « std::endl;
}

bool UDPhandler::username_exists(uint8_t *buf, synch *synch_vars) {
    std::string username;
    int i = 3;
    while (buf[i] != 0x00) {
        username.push_back(static_cast<char>(buf[i]));
        i++;
    }

    if (!synch_vars->usernames.empty()) {
        if (synch_vars->usernames.find(username) != synch_vars->usernames.end())
            return true;
    }
    synch_vars->usernames.insert(username);
    this->user_n = username;
    return false;
}
```

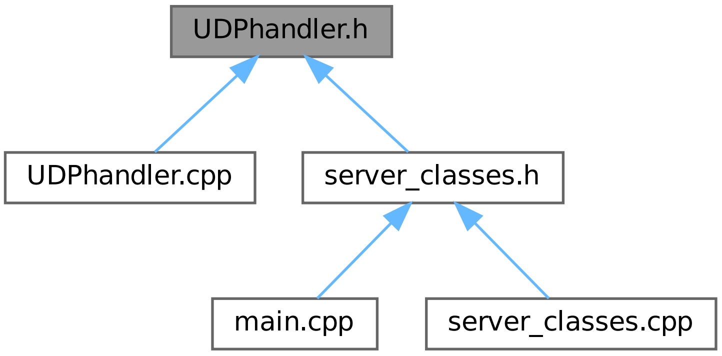
7.25 UDPhandler.h File Reference

```
#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

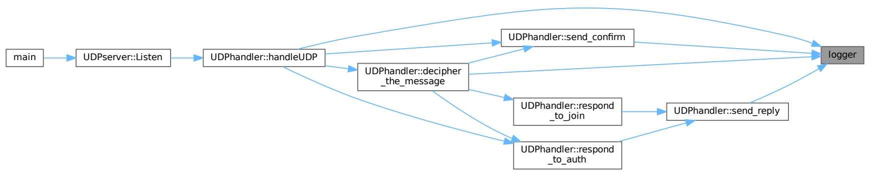
7.25.1.1 logger()

```
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:



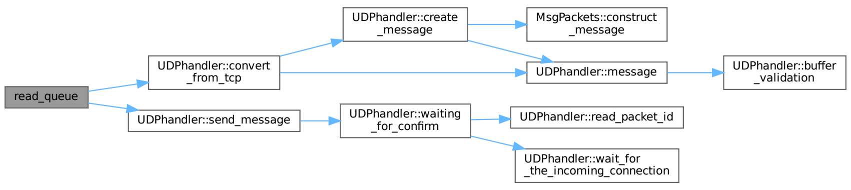
7.25.1.2 read_queue()

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

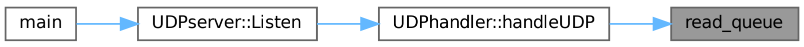
Definition at line 49 of file UDPhandler.cpp.

```
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
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:


read_queue
UDPhandler::convert_from_tcp
UDPhandler::send_message
UDPhandler::create_message
MsgPackets::construct_message
UDPhandler::message
UDPhandler::buffer_validation
UDPhandler::waiting_for_confirm
UDPhandler::read_packet_id
UDPhandler::wait_for_the_incoming_connection

Here is the caller graph for this function:



7.26 UDPhandler.h

Go to the documentation of this file.

```
//
// Created by artem on 4/14/24.
//

#ifndef IPK_SERVER_UDPHANDLER_H
#define IPK_SERVER_UDPHANDLER_H

#include <cstdint>

#include <cstdio>
#include <cstdlib>

#include "packets.h"

#include <utility>

#include <algorithm>

#include <queue>
#include "synch.h"


class UDPhandler {
public:
    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;

    UDPhandler(int ret, int t, sockaddr_in client, int kill) {
        this->retransmissions = ret;
        this->timeout_chat = t;
        this->global_counter = 0;
        this->client_socket = socket(AF_INET, SOCK_DGRAM, 0);
        if (this->client_socket < 0) {
            perror("Problem with creating response socket");
            exit(EXIT_FAILURE);
        }


        epoll_fd = epoll_create1(0);
        if (epoll_fd == -1) {
            std::cerr « "Failed to create epoll file descriptor\n";
            exit(EXIT_FAILURE);
        }

        // setup epoll event
        struct epoll_event ev;
        ev.events = EPOLLIN | EPOLLET;
        ev.data.fd = this->client_socket;

        // add socket file descriptor to epoll
        if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, this->client_socket, &ev) == -1) {
            std::cerr « "Failed to add file descriptor to epoll\n";
            close(epoll_fd);
            exit(EXIT_FAILURE);
        }

        ev.data.fd = kill;
        if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, kill, &ev) < 0) {
            std::cerr « "Unable to add socket to epoll\n";
            exit(EXIT_FAILURE);
        }

        auth = false;

        client_addr = client;

        channel_name = "general";

    }

    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);

    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);

private:
    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);

    static int read_packet_id(uint8_t *buf);

    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);

};

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);



#endif //IPK_SERVER_UDPHANDLER_H

```

Index