Libcommunist

Libcommunist is a simple library for peer to peer messaging in Linux and Windows operating systems.

Building and installation

To build libcommunist from source go to source code directory and execute following commands:

meson -Dbuildtype=release build
ninja -C build install

You may need to set prefix by -Dprefix= option of meson (default prefix is /usr/local). Installation may need superuser privileges. Installation to custom directory can be done by following command:

DESTDIR=/path/to/your/directory ninja -C build install

Dependencies

Libcommunist requires following libraries to be installed: libtorrent-rasterbar (http://libtorrent.org/), libzip (https://libzip.org/), libgcrypt (https://www.gnupg.org/software/libgcrypt/index.html).

License

GPLv3

Functions documentation

LibCommunist.h

int **createProfile** (std::string *Username*, std::string *Password*, std::string *homepath*, std::vector<std::tuple<std::string, std::vector<char>> &profvect, std::array<char, 32> &*seed*)

Function creates user profile and saves it in *homepath*/.Communist/Profile. *Username* should be any non-empty string, *Password* should be any non-empty string. *homepath* must be a UTF-8 string, containing absolute path to any existing directory. *seed* is array, generated by **seedGenerate** (). *profvect* – vector of tuples *key*-*value*. Possible keys are the following: “Nick”, “Name”, “Surname”, “Avatar”. Tuple “Nick”-*value* is compulsory, *value* must be any non-empty string converted to a vector of chars. Tuple “Name”-*value* is not compulsory, *value* can be any string converted to a vector of chars. Tuple “Surname”-*value* is not compulsory, *value* can be any string converted to a vector of chars. Tuple “Avatar”-*value* is not compulsory, *value* must be a vector containing any jpeg picture. *seed* – is a seed generated by **seedGenerate** function. If profile creation was successful, function returns 1. In case of any critical error function returns -1. 0 will be returned in case of any errors withing avatar creation (such errors are not critical for profile creation). Profile is a zip archive, encrypted by AES256 CBC CTS algorithm. Archive contains Profile text file (nickname, name, surname), Key file (seed to generate ed25519 key pair), Contacts file (list of contact’s

indexes and keys), Avatar.jpeg – avatar (if any), RequestList file (list of keys to add to contact list if any) and RelayContacts file (list of contacts, messages to which should be sent by relay servers if any).

void **cryptFile** (std::string *Username*, std::string *Password*, std::string *infile*, std::string *outfile*)

Auxiliary function, encrypts given file by AES256 CBC CTS algorithm. *Username* – any non-empty string, *Password* – any non-empty string, *infile* – UTF-8 string, containing absolute path to file you need to encrypt, *outfile* – UTF-8 string, containing absolute path to output file. File must have size equal 16 bytes or more.

std::vector<char> **cryptStrm** (std::string *Username*, std::string *Password*, std::vector<char> &*input*)

Auxiliary function, encrypts given vector by AES256 CBC CTS algorithm. *Username* – any non-empty string, *Password* – any non-empty string, *input* – vector of chars, you need to encrypt. Vector must have size more than 16 bytes (elements) or 16 bytes (elements). Returns encrypted vector.

void **decryptFile** (std::string *Username*, std::string *Password*, std::string *infile*, std::string *outfile*)

Auxiliary function, decrypts given file (if it was encrypted by **cryptFile** function). *Username* – same string as was used in **cryptFile** function, *Password* – same string as was used in **cryptFile** function, *infile* – UTF-8 string, containing absolute path to file you need to decrypt, *outfile* – UTF-8 string, containing absolute path to output file. File must have size 16 bytes or more.

std::vector<char> **decryptStrm** (std::string *Username*, std::string *Password*, std::vector<char> &*input*)

Auxiliary function, decrypts given vector (if it was encrypted by **cryptStrm** function). *Username* – same string as was used in **cryptStrm** function, *Password* – same string as was used in **cryptStrm** function, *input* – vector of chars, you need to decrypt, must have size 16 bytes (elements) or more. Function returns decrypted vector.

int **editProfile** (std::string *Username*, std::string *Password*, std::string *homepath*, std::vector<std::tuple<std::string, std::string>> *&profvect*)

Function replaces information in Profile by given information. *Username* – same string as was used in **createProfile** function, *Password* – same string as was used in **createProfile** function, *profvect* – vector of tuples *key-value*. Possible keys are the following: “Nick”, “Name”, “Surname”, “Avatar”. Tuple “Nick”-*value* is compulsory, *value* must be any non-empty string. Tuple “Name”-*value* is not compulsory, *value* can be any string. Tuple “Surname”-*value* is not compulsory, *value* can be any string. Tuple “Avatar”-*value* is not compulsory, *value* must be UTF-8 string containing absolute path to any jpeg picture. *homepath* must be a UTF-8 string, containing absolute path to profiles’ directory (same as used in **createProfile** function). Function returns -1 in case of critical errors, 0 – in case of avatar adding errors, 1 – in case of success.

std::string **genFriendPasswd** (std::string *key*, std::array<char, 32> &*seed*)

Function creates password for sent messages. *key* should be UTF-8 string 64 chars in length, *seed* – is a seed generated by **seedGenerate**. Function returns UTF-8 string containing password.

std::string **genPasswd** (std::string *key*, std::array<char, 32> &*seed*)

Function returns password for received messages. *key* should be UTF-8 string 64 chars in length, *seed* – is a seed generated by **seedGenerate**. Function returns UTF-8 string containing password.

std::string **getContactMsgLog** (std::string *homepath*, std::string *Username*, std::string *Password*, int *contind*, std::string *outfolderpath*)

Function returns UTF-8 absolute path to decrypted contact's message log file (file will not exist or will be empty, if no messages were sent or received). Log file contains contact's public key, time in seconds (a value of integral type holding the number of seconds since the Epoch) and message index. *homepath* must be a UTF-8 string, containing absolute path to profiles' directory (same as was used in **createProfile** function). *Username* – same string as was used in **createProfile** function, *Password* - same string as was used in **createProfile** function. *contind* – index of contact (see **readContacts** function). *outfolderpath* – path to directory you want decrypted log file to be placed to. Warning! If directory exists, it will be deleted before decryption.

std::string **getKeyFmSeed** (std::array<char, 32> &*seed*)

Function returns public key for given *seed* in hex format.

std::string **getSecreteKeyFmSeed** (std::array<char, 32> *&seed*)

Function returns secrete key for given *seed* in hex format.

std::vector<std::filesystem::path> **listMessages** (std::string *homepath*, std::string *key*, std::string *Username*, std::string *Password*)

Function returns list of all messages received from and sent to contact. *homepath* – UTF-8 string, containing absolute path, same as was used in **createProfile** function, *key* – string containing hex value of contact's public key, *Username* – same string as was used in **createProfile** function, *Password* - same string as was used in **createProfile** function. Function returns vector, containing absolute paths to messages of selected contact.

void **msgAutoRemove** (std::string *homepath*, std::string *Username*, std::string *Password*, std::string *mode*)

Function removes all messages for all contacts older, then set by *mode* value. *homepath* – UTF-8 string, containing absolute path, same as was used in **createProfile** function, *Username* – same string as was used in **createProfile** function, *Password* – same string as was used in **createProfile** function, *mode* – UTF-8 string, containing "1", "2", "3" or "4" (other values will be ignored). "1" – function will remove messages older than 1 day, "2" – older than 1 week, "3" – older than 31 days, "4" – older than 365 days.

int **openFriendProfile** (std::string *homepath*, std::string *Username*, std::string *Password*, std::string *friendkey*, std::string *outfoldername*)

Function decrypts friend's profile and places it to given directory. *homepath* – UTF-8 string, containing absolute path, same as was used in **createProfile** function, *Username* – same string as was used in **createProfile** function, *Password* – same string as was used in **createProfile** function. *friendkey* – string, containing hex value of friend's public key, *outfoldername* – UTF-8 string,

containing absolute path to directory where result should be placed. Function returns 1 in case of success, -1 in case of any errors.

std::string **openMessage** (std::filesystem::path *msgpath*, std::string *friendkey*, std::string *Username*, std::string *Password*)

Function opens given message. *msgpath* – absolute path to message (see **listMessages** for details), *friendkey* – string, containing hex value of friend's public key, *Username* – same string as was used in **createProfile** function, *Password* – same string as was used in **createProfile** function. Function returns UTF-8 string, containing absolute path to opened message. Message header contains several lines: *From:* (contains hex value of opponent's public key), *Resend from:* (contains message first author nickname if message has been resent, empty otherwise), *To:* (contains hex value of contacts' public key message has been addressed to), Creation time: (contains time in seconds message has been created – a value of integral type holding the number of seconds since the Epoch), *Type:* (contains message type, two types are available now: *text message* and *file message*), *Reply to:* (contains quotation of replied message if message is a reply to any other message, empty otherwise). *Message:* (contains message text beginning from next line).

int **openProfile** (std::string *homepath*, std::string *Username*, std::string *Password*, std::string *outfoldername*)

Function decrypts Profile file and puts it to *outfoldername* directory. *homepath* – UTF-8 string, containing absolute path, same as was used in **createProfile** function, *Username* – same string as was used in **createProfile** function, *Password* - same string as was used in **createProfile** function, *outfoldername* – must be UTF-8 string, containing absolute path to directory you want to place result to. Function returns -1 in case if Profile file was not found, 0 – in case of any other errors, and 1 in case of successful completion of operation. Warning! You should remove *outfoldername* directory by yourself after usage to avoid private information leakage, library does not control *outfoldername* directory behavior.

int **packing** (std::string *source*, std::string *out*)

Auxiliary function, packs file or directory to zip archive. *source* – UTF-8 string, containing absolute path to file or directory you need to pack, *out* – UTF-8 string, containing absolute path to result archive. Function returns 1 in case of success, -1 if *source* file does not exist, -2 in case of any errors (if *source* is a directory), -3 or -4 in case of any errors (if *source* is a file).

std::string **randomFileName** ()

Function returns UTF-8 string, containing random file name (function based on std::random()).

std::vector<std::tuple<int, std::string>> **readContacts** (std::string *homepath*, std::string *Username*, std::string *Password*)

Function returns vector of existing contacts tuples *index-key*. *index* – contact's number in order, *key* – hex value of contact's public key. *homepath* must be a UTF-8 string, containing absolute path to profile directory (same as was used in **createProfile** function). *Username* – same string as was used in **createProfile** function, *Password* - same string as was used in **createProfile** function.

std::vector<std::tuple<std::string, std::string>> **readFriendProfile** (std::string *homepath*, std::string *friendkey*, std::string *Username*, std::string *Password*)

Function reads friend's Profile file to vector of tuples. *homepath* – UTF-8 string, containing absolute path, same as was used in **createProfile** function, *friendkey* – string, containing hex value of friend's public key, *Username* – same string as was used in **createProfile** function, *Password* - same string as was used in **createProfile** function. Function returns vector of tuples *key-value* (see **createProfile** function *profvect*).

std::vector<std::tuple<std::string, std::vector<char>> **readProfile** (std::string *homepath*, std::string *Username*, std::string *Password*)

Function reads Profile file to vector of tuples. *homepath* – UTF-8 string, containing absolute path, same as was used in **createProfile** function, *Username* – same string as was used in **createProfile** function, *Password* – same string as was used in **createProfile** function. Function returns vector of tuples *key-value* (see **createProfile** function *profvect*).

std::vector<std::string> **readRelayContacts** (std::string *homepath*, std::string *Username*, std::string *Password*)

Function returns list of contact's keys to send messages to by relay servers. *homepath* – UTF-8 string, containing absolute path, same as was used in **createProfile** function, *Username* – same string as was used in **createProfile** function, *Password* – same string as was used in **createProfile** function. Function returns vector of hex values of contacts public keys.

std::vector<std::string> **readRequestList** (std::string *homepath*, std::string *Username*, std::string *Password*)

Function returns vector of contacts you want to establish connection with. *homepath* must be a UTF-8 string, containing absolute path to profiles' directory (same as was used in **createProfile** function). *Username* – same string as was used in **createProfile** function, *Password* - same string as was used in **createProfile** function.

int **readSeed** (std::string *homepath*, std::string *Username*, std::string *Password*, std::array<char, 32> *&seed*)

Function reads seed saved in your Profile archive to array. *homepath* – UTF-8 string, containing absolute path, same as was used in **createProfile** function, *Username* – same string as was used in **createProfile** function, *Password* – same string as was used in **createProfile** function, *seed* – char array information will be read to. Array must have 32 chars size. Function returns 1 in case of success, -1 in case of any errors.

std::array<char, 32> **seedGenerate** ()

Function generates random seed to create key pair.

int **unpacking** (std::string *archadress*, std::string *outfolder*)

Auxiliary function, unpack zip archives. *archadress* – UTF-8 string, containing absolute path to archive you need to unpack. *outfolder* – UTF-8 string, containing absolute path to directory where result of unpacking will be placed. Function returns 1 in case of success, -1 or -2 in case of any errors.

NetOperationsComm.h

Callback functions

std::function<void (std::string, std::filesystem::path)> **filehasherr_signal**

This function will be called if hash received from opponent and actual hash of file are not equal (this means error on file receiving). First argument is opponent's public key in hex format, second argument is absolute path to file on receiving of which error occurred.

std::function<void (std::string, std::filesystem::path, uint64_t)> **filepartrcvd_signal**

This function will be called if part of file has been received. First argument is opponent's public key in hex format, second argument is absolute path to file, third argument is current size of the file. This function may be useful to control file receiving progress.

std::function<void (std::string, std::filesystem::path, uint64_t)> **filepartsend_signal**

This function will be called if part of file has been successfully sent. First argument is opponent's public key in hex format, second argument is absolute path to file, third argument is quantity of bytes have been totally sent. This function may be useful to control file sending progress.

std::function<void (std::string, std::filesystem::path)> **filercvd_signal**

This function will be called after file has been successfully received. First argument is opponent's public key in hex format, second argument is absolute path to file has been received.

std::function<void (std::string, std::filesystem::path)> **fileRejected_signal**

This function will be called after opponent has rejected your request to receive file. First argument is opponent's public key in hex format, second argument is absolute path to file, which has been rejected.

std::function<void (std::string, uint64_t, uint64_t, std::string)> **filerequest_signal**

This function will be called after request to receive file from an opponent have been obtained. First argument is opponent's public key in hex format, second argument is time request has been created by opponent (a value of integral type holding the number of seconds since the Epoch), third argument is file size in bytes, fourth argument is file name.

std::function<void (std::string, std::filesystem::path)> **filesent_signal**

This function will be called after file has been sent. First argument is opponent's public key in hex format, second argument is absolute path to file has been sent.

std::function<void (std::string, std::filesystem::path)> **filesenterror_signal**

This function will be called if any errors occurred while sending a file. First argument is opponent's public key in hex format, second argument is absolute path to the file.

std::function<void ()> **friendBlocked_signal**

This function will be called after contact has been blocked (see **blockFriend** function).

std::function<void (std::string)> **friendDeleted_signal**

This function will be called after deletion of contact has been completed. Argument is opponent's public key in hex format.

std::function<void ()> **friendDelPulse_signal**

This function will be called time to time by contact's deletion process (it can be useful for indication that process is not frozen).

std::function<void (std::string)> **ipv4_signal**

This function will be called if machine has more than one available ipv4 address. One address – one function call. After all addresses have been listed **ipv4finished_signal** will be called. Argument is ipv4 address in string format (like a "192.168.1.1").

std::function<void ()> **ipv4finished_signal**

This function will be called after all ipv4 addresses have been listed (see **ipv4_signal**). Program execution will be stopped after this function call until **setIPv4** function call.

std::function<void (std::string)> **ipv6_signal**

This function will be called if machine has more than one available ipv6 address. One address – one function call. After all addresses have been listed **ipv6finished_signal** will be called. Argument is ipv6 address in a string format.

std::function<void ()> **ipv6finished_signal**

This function will be called after all ipv6 addresses have been listed (see **ipv6_signal**). Program execution will be stopped after this function call until **setIPv6** function call.

std::function<void (std::string, std::filesystem::path)> **messageReceived_signal**

This function will be called on message receiving. First argument is an opponent's public key in hex format, second argument is absolute path to received message file.

std::function<void (std::string, std::filesystem::path)> **msgSent_signal**

This function will be called after message or file has been sent. First argument is an opponent's public key in hex format, second argument is absolute path to message file.

std::function<void ()> **net_op_canceled_signal**

This function will be called after all network operations have been finished.

std::function<void (std::string, int)> **profReceived_signal**

This function will be called on profile receiving. First argument is an opponent's public key in hex format, second argument is contacts' index (see **readContacts** of **LibCommunist.h**).

std::function<void (std::string, uint64_t)> **smthrcvd_signal**

This function will be called if maintenance message has been received (it means, that direct or relay connection has been established). Such messages are being emitted every 3-5 seconds to maintain connection. First argument is opponent's public key in hex format, second argument is time stamp (a value of integral type holding the number of seconds since the Epoch).

Functions

void **blockFriend** (std::string *key*)

Function temporary blocks all net operations for selected contact. Operations can be resumed by **startFriend** function (or by recreation of NetOperationsComm object). *key* – opponent's public key in hex format.

void **cancelNetOperations** ()

Function cancels all net operations. On operation completion **net_op_canceled_signal** will be called.

void **cancelReceivFile** (std::string *key*, std::filesystem::path *filepath*)

Function interrupts file receiving operations. *key* – opponents' public key in hex format (UTF-8 string), *filepath* – absolute path to file which receiving should be interrupted.

void **cancelSendFile** (std::string *key*, std::filesystem::path *filepath*)

Function interrupts file sending operations. *key* – opponents' public key in hex format (UTF-8 string), *filepath* – absolute path to file which sending should be interrupted.

bool **checkIfMsgSent** (std::filesystem::path *p*)

This function checks if message was sent. *p* – absolute path to message file (see **listMessages** of **LibCommunist.h**). Function returns *true* if message was sent and *false* otherwise.

static void **cleanMemory** (NetOperationsComm **noc*)

Function destroys NetOperationsComm object. *noc* – the object which should be destroyed (see **create_object**). This function must be called only after **net_op_canceled_signal** call, otherwise library threads will not be finished correctly.

static NetOperationsComm* **create_object** ()

Function creates class object. This function must be called before other functions from **NetOperationsComm.h**. Function returns pointer to created NetOperationsComm object. You should use **cleanMemory** function when you do not need it anymore.

void **editContByRelay** (std::vector<std::string> *&sendbyrel*)

Function sets list of contacts messages to which must be sent by relay servers. *sendbyrel* – vector of contacts' public keys in hex format (UTF-8 strings).

void **fileAccept** (std::string *key*, uint64_t *tm*, std::filesystem::path *sp*)

Function sends "file has been accepted signal" to your opponent. *key* – opponents' public key in hex format (UTF-8 string), *tm* – time file request has been created by opponent (see **filerequest_signal**), *sp* – absolute path (including filename and extension) you want save incoming file to (parent directory must exist before this function call).

void **fileReject** (std::string *key*, uint64_t *tm*)

Function sends "file has been rejected" signal to your opponent. *key* – opponents' public key in hex format (UTF-8 string), *tm* – is time file request has been created by opponent (see **filerequest_signal**).

void **getNewFriends** (std::string *key*)

This function is to be used to add new contact. *key* is public key in hex format of the opponent you want to establish connection with.

void **removeFriend** (std::string *key*)

This function removes contact from contact list and removes all messages were sent to and received from contact. *key* is a string, containing hex value of friend's public key. **friendDelPulse_signal** will be called time to time on function execution. On function completion **friendDeleted_signal** will be called.

std::filesystem::path **removeMsg** (std::string *key*, std::filesystem::path *msgpath*)

Function removes selected message (only local machine). *key* – opponents' public key in hex format (UTF-8 string), *msgpath* – absolute path to message (see **listMessages** of **LibCommunist.h** for details). If message is "file message" function returns absolute path to file (see **sendFile**), otherwise returned path will be empty.

void **renewProfile** (std::string *key*)

Function sends your profile to selected opponent (in case you have edited it for example). *key* – opponents' public key in hex format (UTF-8 string).

std::filesystem::path **sendFile** (std::string *key*, std::string *nick*, std::string *replstr*, std::string *pathtofile*)

Function sends file. *key* – public key in hex format (UTF-8 string) of the opponent you want to send file to, *nick* – nickname to be added to *Resend from:* line (see **openMessage** function) if it is needed (empty string otherwise), *replstr* – quotation of message to add to *Reply to:* line (see **openMessage** function, 40 chars max length, it will be cut otherwise) if it is needed (empty string otherwise), *pathtofile* – absolute path to file you want to send. Function returns absolute path to encrypted message formed on file creation.

std::filesystem::path **sendMessage** (std::string *key*, std::string *nick*, std::string *replstr*, std::filesystem::path *msgpath*)

Function sends message. *key* – public key in hex format (UTF-8 string) of the opponent you want to send message to, *nick* – nickname to be added to *Resend from:* line (see **openMessage** function) if it is needed (empty string otherwise), *replstr* – quotation of message to add to *Reply to:* line (see **openMessage** function, 40 chars max length, line will be cut otherwise) if it is needed (empty string otherwise), *msgpath* – absolute path to file, containing message text. Function returns absolute path to encrypted message.

std::filesystem::path **sendMessage** (std::string *key*, std::string *nick*, std::string *replstr*, std::string *msgstring*)

Function sends message. *key* – public key in hex format (UTF-8 string) of the opponent you want to send message to, *nick* – nickname to be added to *Resend from:* line (see **openMessage** function) if it is needed (empty string otherwise), *replstr* – quotation of message to add to *Reply to:* line (see **openMessage** function, 40 chars max length, it will be cut otherwise) if it is needed (empty string otherwise), *msgstring* – a string, containing message text. Function returns absolute path to encrypted message.

void **setHomePath** (std::string *homepath*)

Function sets path to directory containing *.Communist* directory (see **createProfile** of **LibCommunist.h**). *homepath* – UTF-8 string, containing absolute path, same as was used in **createProfile** function.

void **setIPv4** (std::string *ip*)

Function sets library internal ipv4 address, resumes program execution after **ipv4finished_signal** call. *ip* – selected ipv4 (see **ipv4_signal**).

void **setIPv6** (std::string *ip*)

Function sets library internal ipv6 address, resumes program execution after **ipv6finished_signal** call. *ip* – selected ipv6 (see **ipv6_signal**).

void **setPrefVector** (std::vector<std::tuple<std::string, std::string>> *&prefvect*)

Function sets vector os settings for network operations. Vector is array of tuple *key-value*. Keys and values must be UTF-8 strings. Possible keys are the following: “Netmode”, “Maxmsgsz”, “Partsize”, “ShutTmt”, “TmtTear”, “Stunport”, “Stun”, “DirectInet”, “Relayport”, “RelaySrv”, “RelayListPath”, “Listenifcs”, “Bootstrapdht”, “Locip6port”, “Locip4port”. All tuples are not compulsory, if any key is not found, default value will be used. Possible values are listed in table below.

Key	Value	Default value
Bootstrapdht	UTF-8 string containing comma separated pairs ipv4:port, [ipv6]:port or address:port. This addresses will be used to bootstrap to DHT.	router.bittorrent.com:6881
DirectInet	direct – if set library will not try	notdirect

	to determine machines' ip address by STUN requests and will use addresses received from OS. notdirect – if set, library will try to obtain own ip addresses by STUN requests.	
HolePunchMech	Library has mechanism for symmetric NAT hole punch (see NetworkOperations.cpp holePunch function). Warning! This mechanism was not properly tested – use it only if other variants (relay servers, VPN, proxy) to connect are not available. active – activates mechanism. notactive – disables mechanism.	notactive
Listenifcs	UTF-8 string containing comma separated pairs ipv4:port or [ipv6]:port, which will be used to bind DHT sockets to.	0.0.0.0:0,[::]:0
Locip4port	Port number to use in ipv4 group calls (0 – 65535).	48655
Locip6port	Port number to use in ipv6 group calls (0 – 65535).	48666
Maxmsgsz	Maximal permitted size of incoming messages in bytes. Messages with size exceeding this value will be ignored.	1048576
Netmode	0 – "internet mode". Library will use DHT to establish connection with opponents. 1 – "local network mode". DHT disabled, library will use ipv4 and ipv6 group calls to establish connection with opponents.	0
Partsize	All messages and files are divided into parts before sending. This parameter is a size of such part in bytes. It is not recommended to set this parameter too small (it will increase traffic consumption) or	1371

	too big (in case of any errors part will be sent again, and obviously it will increase traffic consumption too).	
RelayListPath	UTF-8 string containing absolute path to file with relay servers addresses. Each address in file must be in separate line and has format “192.168.1.1:1234”.	<Empty>
Relayport	Port number to use in relay operations.	3029
RelaySrv	enabled – library has internal relay server. If set, it will be enabled. disabled – if set, internal relay server will be disabled.	disabled
ShutTmt	Time in seconds after which contact will be considered “offline” if no any datagrams from it were received. If contact is “offline”, library will not try to establish connection with it until new ip addresses are not received (by DHT or group calls). It means technical messages sending stop.	600
Stun	active – enables internal STUN server. notactive – if set internal STUN server will be disabled.	notactive
Stunport	Port number to use in STUN operations.	3478
TmtTear	Time in seconds after which connection with contact will be considered broken. If connection is broken library will not try to send any datagrams except maintenance messages.	20

void **setStunListPath** (std::string *path*)

Function sets path to list of STUN servers. *path* – UTF-8 string containing absolute path to list file. Each stun server in the list must start from new line and has format *address port* or *ip port.* Example:
stun.phonepower.com 3478

192.168.122.1 5000

void **setUsernamePasswd** (std::string *Username*, std::string *Password*)

This function sets username and password for internal usage in net operations. *Username* – same string as was used in **createProfile** (see **LibCommunist.h**) function, *Password* – same string as was used in **createProfile** (see **LibCommunist.h**) function.

void **startFriend** (std::string *key*, int *ind*)

Function resumes net operations after **blockFriend** function call. *key* – opponents' public key in hex format (UTF-8 string), *ind* – index of contact (see **readContacts** function of **LibCommunist.h**).

void **startNetOperations** ()

This function starts net operations loop. To stop it use **cancelNetOperations** function. Do not call this function after **cancelNetOperation** call. You need to recreate NetOperationsComm object to restart net operations in such case.

Tutorial

First of all you need to create profile. Include **LibCommunist.h** header to your project and create a seed by **seedGenerate** function. Save it to proper array (it will be needed on next step). Then you need to create profile file itself. This can be done by **createProfile** function. After that you need to add contacts to communicate with.

"Contact add" operation can be done as follows. Include **NetOperationsComm.h** in your project and create NetOperationsComm object by **create_object** function. Set home directory (see **createProfile**) by **setHomePath** function, set vector of preferences by **setPrefVector** function, set path to STUN list file by **setStunListPath** function, set username and password by **setUsernamePasswd** function. Bind all callback functions you need to receive information from and call **startNetOperations** function. After that you can call **getNewFriends** function. See networkOp function of MainWindow.cpp of Communist project (https://github.com/ProfessorNavigator/communist) as example.

After contacts were added, you can use library for direct peer to peer communication (see this document and https://github.com/ProfessorNavigator/communist as example).