```
# Compiler and flags
CC         = g++
CFLAGS     = -g -Wall -lcrypto

# Directories
OBJDIR     = build
SRCDIR     = src
BINDIR     = dist
HDRDIR     = include
DOCDIR     = doc
DOCTMPDIR  = build/doc
FOLDERS    := $(strip $(shell find $(SRCDIR) -type d -printf '%P\n'))

# List of targets
UTILS      = client/connect4 network/inet_utils network/messages network/socket_wrapper security/secure_socket_wrapper security/crypto utils/dump_buffer network/host server/user_list utils/args client/single_player client/multi_player client/server client/server_lobby security/crypto_utils utils/buffer_io
TARGETS    = client/client server/server

SRCS = $(addsuffix .cpp, $(addprefix $(SRCDIR)/,$(UTILS))) $(addsuffix .cpp, $(addprefix $(SRCDIR)/,$(TARGETS)))

# Documentation output
DOCPDFNAME = documentation.pdf
SRCPDFNAME = source_code.pdf

# Documentation config file
DOXYGENCFG = doxygen.cfg

override CFLAGS += -I $(HDRDIR)
override LDFLAGS += -lpthread

# Object files for utilities (aka libraries)
UTILS_OBJ  = $(addsuffix .o, $(addprefix $(OBJDIR)/,$(UTILS)))

# Builds only the executables: default rule
exe: $(addprefix $(BINDIR)/,$(TARGETS))

# Utilities are secondary targets
.SECONDARY: $(UTILSOBJ)

# Build targets
$(BINDIR)/%: $(OBJDIR)/%.o $(UTILS_OBJ) $(HDRDIR)/**/*.h $(HDRDIR)/*.h $(SRCDIR)/**/*.h
	$(CC) $(CFLAGS) -o $@ $(filter %.o,$^) $(LDFLAGS)
	chmod +x $@

# Build generic .o file from .cpp file
$(OBJDIR)/%.o: $(SRCDIR)/%.cpp $(HDRDIR)/**/*.h $(HDRDIR)/*.h $(SRCDIR)/**/*.h
	$(CC) $(CFLAGS) -c $< -o $@ $(LDFLAGS)

# Note that if I modify any header everything is built again
# This is not very effective but for such small project that's not an issue

# Build documentation pdf
$(DOCDIR)/$(DOCPDFNAME): $(SRCS) $(HDRDIR)/**/*.h $(HDRDIR)/*.h $(DOXYGENCFG)
	doxygen $(DOXYGENCFG)
	( cd $(DOCTMPDIR)/latex ; make )
	cp $(DOCTMPDIR)/latex/refman.pdf $(DOCDIR)/$(DOCPDFNAME)


# prepare sorted source list for source code pdf generation
# I want the header to appear right before the c source file
# the source files of client and server will be last
both = $(HDRDIR)/$(1).h $(SRCDIR)/$(1).cpp
ALL_SOURCES = $(foreach x,$(UTILS),$(call both,$(x)))
ALL_SOURCES += logging.h
ALL_SOURCES += $(addprefix $(SRCDIR)/,$(addsuffix .cpp,$(TARGETS)))

```

```
# Build source code ps file
$(OBJDIR)/sources.ps : $(SRCDIR)/**/*.cpp $(SRCDIR)/**/*.h $(HDRDIR)/*.h
        echo $(ALL_SOURCES)
        enscript -C -fCourier9 --highlight=c -p$(OBJDIR)/sources.ps  $(ALL_SOURCES)

# Build makefile ps file
$(OBJDIR)/makefile.ps: Makefile
        enscript -C -fCourier9 --highlight=makefile -p$(OBJDIR)/makefile.ps Makefile

$(OBJDIR)/sources.pdf: $(OBJDIR)/sources.ps
        ps2pdf $(OBJDIR)/sources.ps $(OBJDIR)/sources.pdf

$(OBJDIR)/makefile.pdf: $(OBJDIR)/makefile.ps
        ps2pdf $(OBJDIR)/makefile.ps $(OBJDIR)/makefile.pdf

# Builds source code pdf file
$(DOCDIR)/$(SRCPDFNAME): $(OBJDIR)/sources.pdf $(OBJDIR)/makefile.pdf
        pdfunite $(OBJDIR)/makefile.pdf $(OBJDIR)/sources.pdf $(DOCDIR)/$(SRCPDFNAME)

# Builds everything (ecutables and documentation)
all: exe $(DOCDIR)/$(DOCPDFNAME) $(DOCDIR)/$(SRCPDFNAME) report

# clean everything
clean:
        $(RM) -r $(OBJDIR)/* $(BINDIR)/* $(DOCDIR)/$(DOCPDFNAME) $(DOCDIR)/$(SRCPDFNAME) doc/report/report.pdf

# clean everything and then rebuild
rebuild: clean all

# just opens output documentation
doc_open: $(DOCDIR)/$(DOCPDFNAME)
        xdg-open $(DOCDIR)/$(DOCPDFNAME)

# generates documentation and opens it in default pdf viewer
doc: $(DOCDIR)/$(DOCPDFNAME) doc_open

# makes source code pdf and opens it
source: $(DOCDIR)/$(SRCPDFNAME)
        xdg-open doc/source_code.pdf

test: exe
        @   i=0; \
                pass=0; \
                for test_script in tests/*.sh; do \
                        echo -n "$$test_script ... "; \
                        sh $$test_script > /dev/null 2> /dev/null; \
                        if [ "$$?" -eq "0" ]; \
                        then \
                                echo "PASS"; \
                                pass=$$(($$pass+1)); \
                        else \
                                echo "FAIL"; \
                        fi; \
                        i=$$(($$i+1)); \
                done; \
                echo "Passed $$pass out of $$i"

doc/report/report.pdf: doc/report/*.tex
        cd doc/report; latexmk -pdf report.tex

report: doc/report/report.pdf

help:
        @echo "all:         builds everything (both binaries and documentation)"
        @echo "clean:       deletes any intermediate or output file in build/, dist/ and doc/"
        @echo "report:      builds the report only"
        @echo "doc:         builds documentation only and opens pdf file"
```

```
	@echo "doc_open:	opens documentation pdf"
	@echo "exe:		builds only binaries"
	@echo "help:		shows this message"
	@echo "rebuild:	same as calling clean and then all"
	@echo "source:		makes source code pdf and opens it"
	@echo "test:		runs all tests defined in tests/*.sh"

# these targets aren’t name of files
.PHONY: all exe clean rebuild doc_open doc help source report

# build project structure
$(shell   mkdir -p $(DOCDIR) $(addprefix $(OBJDIR)/,$(FOLDERS)) $(BINDIR)/client $(BINDIR)/server  test)
```

```cpp
/**
 * @file connect4.cpp
 * @author Mirko Laruina
 *
 * @brief Implementation of connect4.h
 *
 * @date 2020-05-14
 *
 * @see connect4.h
 *
 */
#include "connect4.h"
using namespace std;

void Connect4::print(ostream& os){
    os<<*this;
}

Connect4::Connect4(int rows /* = 6 */, int columns /* = 7 */){
    rows_ = rows;
    cols_ = columns;
    size_ = rows*columns;
    full_ = false;

    //Maybe check for overflow if we will use different board values

    cells_ = new char[size_];
    memset(cells_, 0, size_);
}

int8_t Connect4::play(int col, char player){
    // bool col_full = true;
    if(player == 0){
        player = player_;
    }

    //Trying to play with a full board
    if(full_){
        return -2;
    }

    for(int i = rows_-1; i>=0; --i){
        if(cells_[i*cols_+col] == 0){
            // col_full = false;
            cells_[i*cols_+col] = player;
            if( checkWin(i, col, player) ){
                return 1;
            } else {
                //All the board could be full now
                if(i == 0 && checkFullTopRow()){
                    full_ = true;
                    return -2;
                } else {
                    return 0;
                }
            }
        }
    }

    //We are sure the board is not full, otherwise we would have already exited
    //If a play was possible, we would have already exited too
    //Only possible case is full column
    return -1;
}

int Connect4::countNexts(char player, int row, int col, int di, int dj){
    int count = 0;
    for(
        int i = row+di, j = col+dj;
        i >= 0 && j >= 0 && i < rows_ && j < cols_;
        i+=di, j+=dj)
    {
        if(cells_[i*cols_+j] != player){
            break;
        } else {
            LOG(LOG_DEBUG, "%d %d", i, j);
            count++;
        }
    }
    return count;
}

bool Connect4::checkWin(int row, int col, char player){
    /*
        Take any of the 4 possible directions
        count how many token of the same player there are
        before and after the new one
        if more than 4, declare win
    */

    LOG(LOG_DEBUG, "Checking (%d, %d)", row, col);
    if(player == 0){
        player = player_;
    }

    for(int di = 1; di >= 0 && di != -1; --di){
        for(int dj = 1; dj >= 0 && di != -1; --dj){
            // direction (0, 0) is useless, since we would miss diagonal (-1, 1)
            // we can exploit the loop to iterate over that
            if(di == 0 && dj == 0){
                di = -1;
                dj = 1;
            }

            int count_forward = Connect4::countNexts(player, row, col, di, dj);
            int count_backward = Connect4::countNexts(player, row, col, -di, -dj);

            // N_IN_A_ROW minus 1 since the token just inserted is excluded
            if(count_forward + count_backward >= (N_IN_A_ROW - 1)){
                return true;
            }
        }
    }

    return false;

}

bool Connect4::checkFullTopRow(){
    for(int j = 0; j<cols_; ++j){
        if(cells_[j] == 0){
            return false;
        }
    }
    return true;
}

int Connect4::getNumCols(){
    return cols_;
}

bool Connect4::setPlayer(char player){
    if(player == 'X' || player == 'x'
        || player == 'O' || player == 'o'){
        player_ = toupper(player);
        adversary_ = player_ == 'X' ? 'O' : 'X';
        return true;
    }
```

```cpp
    return false;
}

char Connect4::getPlayer(){
    return player_;
}

char Connect4::getAdv(){
    return adversary_;
}

ostream& operator<<(ostream& os, const Connect4& c){
    int width = 2+3*(c.rows_+1);
    for(int i = 0; i<width; ++i){
        os<<'*';
    }
    os<<endl;

    for(int i = 0; i<c.rows_; ++i){
        os<<"*";
        for(int j = 0; j<c.cols_; ++j){
            if(c.cells_[i*c.cols_+j] == 0){
                os<<"   ";
            } else {
                os<< " " << (c.cells_[i*c.cols_+j] == 'X' ? "\033[31mX" : "\033[34mO") <<" ";
            }
        }
        os<<"\033[0m*"<<endl;
    }

    for(int i = 0; i<width; ++i){
        os<<'*';
    }
    os<<endl;

    for(int i = 0; i<width; ++i){
        if( (i+1)%3 == 0  ){
            os<<(i+1)/3;
        } else {
            os<<" ";
        }
    }
    os<<endl;
    return os;
}
```

```
/**
 * @file inet_utils.h
 * @author Riccardo Mancini
 *
 * @brief Utility funcions for managing inet addresses.
 *
 * This library provides functions for creating sockaddr_in structures from
 * IP address string and integer port number and for binding to a random
 * port (chosen using rand() builtin C function).
 *
 * @date 2020-05-17
 *
 * @see sockaddr_in
 * @see rand
 */

#ifndef INET_UTILS
#define INET_UTILS


#include <sys/socket.h>
#include <netinet/in.h>
#include <string>

using namespace std;

/** Random port will be greater or equal to FROM_PORT */
#define FROM_PORT 49152

/** Random port will be lower or equal to TO_PORT */
#define TO_PORT   65535

/** Maximum number of trials before giving up opening a random port */
#define MAX_TRIES 256

/**
 * Maximum number of characters of INET address to string
 * (eg 123.156.189.123:45678).
 */
#define MAX_SOCKADDR_STR_LEN 22

/**
 * Size of a serialized sockaddr_in
 * 32 bit address + 16 bit port = 6 bytes
 */
#define SERIALIZED_SOCKADDR_IN_LEN 6


/**
 * Binds socket to a random port.
 *
 * @param socket    socket ID
 * @param addr      inet addr structure
 * @return          0 in case of failure, port it could bind to otherwise
 *
 * @see FROM_PORT
 * @see TO_PORT
 * @see MAX_TRIES
 */
int bind_random_port(int socket, struct sockaddr_in *addr);

/**
 * Makes sockaddr_in structure given ip string and port of server.
 *
 * @param ip      ip address of server
 * @param port    port of the server
 * @return        sockaddr_in structure for the given server
 */
struct sockaddr_in make_sv_sockaddr_in(const char* ip, int port);
```

```

/**
 * Makes sockaddr_in structure of this host.
 *
 * INADDR_ANY is used as IP address.
 *
 * @param port    port of the server
 * @return        sockaddr_in structure this host on given port
 */
struct sockaddr_in make_my_sockaddr_in(int port);

/**
 * Compares INET addresses, returning 0 in case they're equal.
 *
 * @param sai1  first address
 * @param sai2  second address
 * @return      0 if thery're equal, 1 otherwise
 */
int sockaddr_in_cmp(struct sockaddr_in sai1, struct sockaddr_in sai2);

/**
 * Converts sockaddr_in structure to string to be printed.
 *
 * @param src   the input address
 */
string sockaddr_in_to_string(struct sockaddr_in src);

/**
 * Serializes sockaddr_in structure to given buffer.
 *
 * @param src   the input address
 * @param buffer the buffer
 */
int writeSockAddrIn(char* buffer, size_t buf_len, struct sockaddr_in src);

/**
 * Deerializes sockaddr_in structure from given buffer.
 *
 * @param buffer the buffer
 * @return the built sockaddr_in struct
 */
int readSockAddrIn(struct sockaddr_in *src, char* buffer, size_t buf_len);

#endif
```

```
/**
 * @file inet_utils.cpp
 * @author Riccardo Mancini
 *
 * @brief Implementation of inet_utils.h.
 *
 * @see inet_utils.h
 *
 * @date 2020-05-17
 */

#include <stdlib.h>
#include <string>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdint.h>

#include "logging.h"

#include "network/inet_utils.h"

using namespace std;

int bind_random_port(int socket, struct sockaddr_in *addr){
    int port, ret, i;
    for (i = 0; i < MAX_TRIES; i++){
        if (i == 0) // first I generate a random one
            port = rand() % (TO_PORT - FROM_PORT + 1) + FROM_PORT;
        else //if it's not free I scan the next one
            port = (port - FROM_PORT + 1) % (TO_PORT - FROM_PORT + 1) + FROM_PORT;

        LOG(LOG_DEBUG, "Trying port %d...", port);

        addr->sin_port = htons(port);
        ret = bind(socket, (struct sockaddr *)addr, sizeof(*addr));
        if (ret != -1)
            return port;
        // consider only some errors?
    }
    LOG(LOG_ERR, "Could not bind to random port after %d attempts", MAX_TRIES);
    return 0;
}

struct sockaddr_in make_sv_sockaddr_in(const char *ip, int port){
    struct sockaddr_in addr;
    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);
    inet_pton(AF_INET, ip, &addr.sin_addr);
    return addr;
}

struct sockaddr_in make_my_sockaddr_in(int port){
    struct sockaddr_in addr;
    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    return addr;
}

int sockaddr_in_cmp(struct sockaddr_in sai1, struct sockaddr_in sai2){
    if (sai1.sin_port == sai2.sin_port &&
        sai1.sin_addr.s_addr == sai2.sin_addr.s_addr)
        return 0;
    else
        return 1;
}

string sockaddr_in_to_string(struct sockaddr_in src){
    char dst[MAX_SOCKADDR_STR_LEN];
    char port_str[6];
    const char *ret;

    sprintf(port_str, "%d", ntohs(src.sin_port));

    ret = inet_ntop(AF_INET, (void *)&src.sin_addr, dst, MAX_SOCKADDR_STR_LEN);
    if (ret != NULL){
        strcat(dst, ":");
        strcat(dst, port_str);
    } else {
        strcpy(dst, "ERROR");
    }

    string s = dst;

    return s;
}

int writeSockAddrIn(char* buffer, size_t buf_len, struct sockaddr_in src){
    if (buf_len < SERIALIZED_SOCKADDR_IN_LEN)
        return -1;

    memcpy(buffer,
            &src.sin_addr.s_addr,
            sizeof(src.sin_addr.s_addr));
    memcpy(buffer+sizeof(src.sin_addr.s_addr),
            &src.sin_port,
            sizeof(src.sin_port));
    return SERIALIZED_SOCKADDR_IN_LEN;
}

int readSockAddrIn(struct sockaddr_in *dst, char* buffer, size_t buf_len){
    if (buf_len < SERIALIZED_SOCKADDR_IN_LEN){
        return -1;
    }

    memset(dst, 0, sizeof(*dst));
    dst->sin_family = AF_INET;
    memcpy(&(dst->sin_addr.s_addr),
        buffer,
        sizeof(dst->sin_addr.s_addr));
    memcpy(&(dst->sin_port),
        buffer+sizeof(dst->sin_addr.s_addr),
        sizeof(dst->sin_port));
    return SERIALIZED_SOCKADDR_IN_LEN;
}
```

```
/**
 * @file messages.h
 * @author Riccardo Mancini
 *
 * @brief Definition of messages
 *
 * @date 2020-05-17
 */

#ifndef MESSAGES_H
#define MESSAGES_H

#include <string>
#include <stdint.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include "logging.h"
#include "config.h"
#include "network/inet_utils.h"
#include "network/host.h"
#include "security/crypto.h"
#include "security/secure_host.h"

using namespace std;

/** Type of message length (first N bytes of packet) */
typedef uint16_t msglen_t;

#define MSGLEN_HTON(x) htons((x))
#define MSGLEN_NTOH(x) ntohs((x))

/** Utility for getting username length (excluding '\0') */
inline size_t usernameLength(string s){
    return min(strlen(s.c_str()), (size_t) MAX_USERNAME_LENGTH);
}

/** Utility for getting username from buffer
 *
 * @param buf the buffer to read the string from
 * @param buflen the size of the buffer
 * @returns the read string
 */
inline int readUsername(string *username, char* buf, size_t buf_len){
    size_t size = min(strnlen(buf, buf_len-1), (size_t) MAX_USERNAME_LENGTH);
    *username = string(buf, size);
    return MAX_USERNAME_LENGTH+1;
}

/**
 * Utility for writing username to buffer
 *
 * NB: buffer must be large enough
 *
 * @param s the username string to be written on buffer
 * @param buf the buffer to write the string to
 * @returns number of written bytes
 */
inline int writeUsername(char* buf, size_t buf_len, string s){
    if (buf_len < MAX_USERNAME_LENGTH+1){
        return -1;
    }
    size_t strsize = usernameLength(s);
    strncpy(buf, s.c_str(), strsize);
    memset(&buf[strsize], 0, MAX_USERNAME_LENGTH-strsize+1);
    return MAX_USERNAME_LENGTH+1;
}

/**
```

```
 * Possible type of messages.
 *
 * When adding a new message class, add a related type here and set its
 * getType method to return it.
 */
enum MessageType
{
    SECURE_MESSAGE,
    CLIENT_HELLO,
    SERVER_HELLO,
    CLIENT_VERIFY,
    START_GAME_PEER,
    MOVE,
    REGISTER,
    CHALLENGE,
    GAME_END,
    USERS_LIST,
    USERS_LIST_REQ,
    CHALLENGE_FWD,
    CHALLENGE_RESP,
    GAME_START,
    GAME_CANCEL,
    CERT_REQ,
    CERTIFICATE,
};

/**
 * Abstract class for Messages.
 */
class Message
{
public:
    virtual ~Message(){};
    /**
     * Write message to buffer
     *
     * @returns >0 number of bytes written
     * @returns  0 in case of errors
     */
    virtual msglen_t write(char *buffer) = 0;

    /**
     * Read message from buffer
     *
     * @returns 0 in case of success, something else in case of errors.
     *          Refer to the implementation for details
     */
    virtual msglen_t read(char *buffer, msglen_t len) = 0;

    /**
     * Get message name (for debug purposes)
     */
    virtual string getName() = 0;

    virtual MessageType getType() = 0;
};

/**
 * Message that signals to start a new game.
 */
class StartGameMessage : public Message
{
public:
    StartGameMessage() {}
    ~StartGameMessage() {}

    msglen_t write(char *buffer);
    msglen_t read(char *buffer, msglen_t len);

    string getName() { return "StartGame"; }

    MessageType getType() { return START_GAME_PEER; }
};

/**
 * Message that signals a move
 */
class MoveMessage : public Message
{
private:
    char col;

public:
    MoveMessage() {}
    MoveMessage(char col) : col(col) {}
    ~MoveMessage() {}

    msglen_t write(char *buffer);
    msglen_t read(char *buffer, msglen_t len);

    string getName() { return "Move"; }

    char getColumn() { return col; }

    MessageType getType() { return MOVE; }
};

/**
 * Message that permits the client to register to server
 */
class RegisterMessage : public Message
{
private:
    string username;

public:
    RegisterMessage() {}
    RegisterMessage(string username) : username(username) {}
    ~RegisterMessage() {}

    msglen_t write(char *buffer);
    msglen_t read(char *buffer, msglen_t len);

    string getName() { return "Register"; }

    string getUsername() { return username; }

    MessageType getType() { return REGISTER; }
};

/**
 * Message that permits the client to challenge another client
 * through the server.
 */
class ChallengeMessage : public Message
{
private:
    string username;

public:
    ChallengeMessage() {}
    ChallengeMessage(string username) : username(username) {}
    ~ChallengeMessage() {}

    msglen_t write(char *buffer);
    msglen_t read(char *buffer, msglen_t len);

    string getName() { return "Challenge"; }

    string getUsername() { return username; }

    MessageType getType() { return CHALLENGE; }
};

/**
 * Message that signals the server that the client is available
 */
class GameEndMessage : public Message
{
public:
    GameEndMessage() {}
    ~GameEndMessage() {}

    msglen_t write(char *buffer);
    msglen_t read(char *buffer, msglen_t len);

    string getName() { return "Game End"; }

    MessageType getType() { return GAME_END; }
};

/**
 * Message that the server sends the client with the list of users
 */
class UsersListMessage : public Message
{
private:
    string usernames;

public:
    UsersListMessage() {}
    UsersListMessage(string usernames) : usernames(usernames) {}
    ~UsersListMessage() {}

    msglen_t write(char *buffer);
    msglen_t read(char *buffer, msglen_t len);

    string getName() { return "User list"; }

    string getUsernames() { return usernames; }

    MessageType getType() { return USERS_LIST; }
};

/**
 * Message with which the client asks for the list of connected users.
 */
class UsersListRequestMessage : public Message
{
private:
    uint32_t offset;

public:
    UsersListRequestMessage() : offset(0) {}
    UsersListRequestMessage(unsigned int offset) : offset(offset) {}
    ~UsersListRequestMessage() {}

    msglen_t write(char *buffer);
    msglen_t read(char *buffer, msglen_t len);

    string getName() { return "Users list request"; }

    uint32_t getOffset() { return offset; }

    MessageType getType() { return USERS_LIST_REQ; }
};

```

```
/**
 * Message with which the server forwards a challenge.
 */
class ChallengeForwardMessage : public Message
{
private:
    string username;

public:
    ChallengeForwardMessage() {}
    ChallengeForwardMessage(string username) : username(username) {}
    ~ChallengeForwardMessage() {}

    msglen_t write(char *buffer);
    msglen_t read(char *buffer, msglen_t len);

    string getName() { return "Challenge forward"; }

    string getUsername() { return username; }

    MessageType getType() { return CHALLENGE_FWD; }
};

/**
 * Message with which the client replies to a challenge.
 */
class ChallengeResponseMessage : public Message
{
private:
    string username;
    bool response;
    uint16_t listen_port;

public:
    ChallengeResponseMessage() {}
    ChallengeResponseMessage(string username, bool response, uint16_t port)
        : username(username), response(response), listen_port(port) {}
    ~ChallengeResponseMessage() {}

    msglen_t write(char *buffer);
    msglen_t read(char *buffer, msglen_t len);

    string getName() { return "Challenge response"; }

    string getUsername() { return username; }
    bool getResponse() { return response; }
    uint16_t getListenPort() { return listen_port; }

    MessageType getType() { return CHALLENGE_RESP; }
};

/**
 * Message with which the server forwards a challenge rejectal or another
 * event that caused the game to be canceled.
 */
class GameCancelMessage : public Message
{
private:
    string username;

public:
    GameCancelMessage() {}
    GameCancelMessage(string username) : username(username) {}
    ~GameCancelMessage() {}

    msglen_t write(char *buffer);
    msglen_t read(char *buffer, msglen_t len);

    string getName() { return "Game cancel"; }

    string getUsername() { return username; }

    MessageType getType() { return GAME_CANCEL; }
};

/**
 * Message with which the server makes a new game start between clients.
 */
class GameStartMessage : public Message
{
private:
    string username;
    struct sockaddr_in addr;
    X509* cert;

public:
    GameStartMessage() {}
    GameStartMessage(string username, struct sockaddr_in addr, X509* opp_cert)
        : username(username), addr(addr), cert(opp_cert) {} //TODO cert
    ~GameStartMessage() {}

    msglen_t write(char *buffer);
    msglen_t read(char *buffer, msglen_t len);

    string getName() { return "Game start"; }

    string getUsername() { return username; }
    struct sockaddr_in getAddr() { return addr; }
    SecureHost getHost() { return SecureHost(addr, cert); }
    X509*      getCert() { return cert; }
    MessageType getType() { return GAME_START; }
};

class SecureMessage : public Message
{
private:
    char *ct;
    msglen_t ct_size;
    char* tag;

public:
    SecureMessage() : ct(NULL), ct_size(0), tag(NULL){}
    SecureMessage(char* ct, msglen_t ct_size, char* tag) : ct(ct), ct_size(ct_size), tag(t
ag){}
    ~SecureMessage(){ if (ct != NULL) free(ct); if (tag != NULL) free(tag); }

    MessageType getType() { return SECURE_MESSAGE; }
    string getName() { return "Secure message"; }

    void setCtSize(msglen_t s) { ct_size = s; }
    size_t getCtSize() { return ct_size; }
    char* getCt() { return ct; }
    char* getTag() { return tag; }

    msglen_t write(char *buffer);
    msglen_t read(char *buffer, msglen_t len);
};

class ClientHelloMessage: public Message
{
private:
    EVP_PKEY* eph_key;
    nonce_t nonce;
    string my_id;
    string other_id;

public:
    ClientHelloMessage() : eph_key(NULL) {}
    ClientHelloMessage(EVP_PKEY* eph_key, nonce_t nonce, string my_id, string other_id)
        : eph_key(eph_key), nonce(nonce), my_id(my_id), other_id(other_id) {}

    MessageType getType() {return CLIENT_HELLO; }
    string getName() { return "Client Hello message"; }

    nonce_t getNonce() { return nonce; }
    EVP_PKEY* getEphKey() { return eph_key; }
    void setEphKey(EVP_PKEY* eph_key) { this->eph_key=eph_key; }
    string getMyId() { return my_id; }
    string getOtherId() { return other_id; }

    msglen_t write(char* buffer);
    msglen_t read(char* buffer, msglen_t len);
};

class ClientVerifyMessage: public Message
{
private:
    char* ds;
    uint32_t ds_size;

public:
    ClientVerifyMessage() : ds(NULL), ds_size(0) {}
    ClientVerifyMessage(char* ds, uint32_t ds_size) : ds(ds), ds_size(ds_size) {}
    ~ClientVerifyMessage();

    MessageType getType() {return CLIENT_VERIFY; }
    string getName() { return "Client Verify message"; }

    char* getDs() { return ds; }
    uint32_t getDsSize() { return ds_size; }

    msglen_t write(char* buffer);
    msglen_t read(char* buffer, msglen_t len);
};

class ServerHelloMessage: public Message
{
private:
    EVP_PKEY* eph_key;
    nonce_t nonce;
    string my_id;
    string other_id;
    char* ds;
    uint32_t ds_size;

public:
    ServerHelloMessage() : eph_key(NULL), ds(NULL), ds_size(0) {}
    ServerHelloMessage(EVP_PKEY* eph_key, nonce_t nonce, string my_id, string other_id, ch
ar* ds, uint32_t ds_size)
        : eph_key(eph_key), nonce(nonce), my_id(my_id), other_id(other_id), ds(ds), ds_siz
e(ds_size) {}
    ~ServerHelloMessage();

    MessageType getType() {return SERVER_HELLO; }
    string getName() { return "Server Hello message"; }

    nonce_t getNonce() { return nonce; }
    EVP_PKEY* getEphKey() { return eph_key; }
    void setEphKey(EVP_PKEY* eph_key) { this->eph_key=eph_key; }
    string getMyId() { return my_id; }
    string getOtherId() { return other_id; }
    char* getDs() { return ds; }
    uint32_t getDsSize() { return ds_size; }

    msglen_t write(char* buffer);
    msglen_t read(char* buffer, msglen_t len);
};
```

```

class CertificateRequestMessage: public Message
{
public:
    CertificateRequestMessage(){}

    MessageType getType() {return CERT_REQ; }
    string getName() { return "Certificate Request message"; }

    msglen_t write(char* buffer);
    msglen_t read(char* buffer, msglen_t len);
};

class CertificateMessage: public Message
{
private:
    X509* cert;
public:
    CertificateMessage() : cert(NULL) {}
    CertificateMessage(X509* cert) : cert(cert) {}

    MessageType getType() {return CERTIFICATE; }
    string getName() { return "Certificate message"; }
    X509* getCert() { return cert; }

    msglen_t write(char* buffer);
    msglen_t read(char* buffer, msglen_t len);
};

/**
 * Reads the message using the correct class and returns a pointer to it.
 *
 * NB: remeber to dispose of the created Message when you are done with it.
 */
Message *readMessage(char *buffer, msglen_t len);

#endif // MESSAGES_H
```

```
 /**
* @file messages.cpp
* @author Riccardo Mancini
*
* @brief Implementation of messages.h
*
* @see messages.h
*/

#include <cstdlib>
#include <cstring>
#include <cmath>

#include "network/messages.h"
#include "network/inet_utils.h"

#include "security/crypto_utils.h"
#include "utils/buffer_io.h"

Message* readMessage(char *buffer, msglen_t len){
    Message *m;
    int ret;

    switch(buffer[0]){
        case SECURE_MESSAGE:
            m = new SecureMessage;
            break;
        case CLIENT_HELLO:
            m = new ClientHelloMessage;
            break;
        case SERVER_HELLO:
            m = new ServerHelloMessage;
            break;
        case CLIENT_VERIFY:
            m = new ClientVerifyMessage;
            break;
        case START_GAME_PEER:
            m = new StartGameMessage;
            break;
        case MOVE:
            m = new MoveMessage;
            break;
        case REGISTER:
            m = new RegisterMessage;
            break;
        case CHALLENGE:
            m = new ChallengeMessage;
            break;
        case GAME_END:
            m = new GameEndMessage;
            break;
        case USERS_LIST:
            m = new UsersListMessage;
            break;
        case USERS_LIST_REQ:
            m = new UsersListRequestMessage;
            break;
        case CHALLENGE_FWD:
            m = new ChallengeForwardMessage;
            break;
        case CHALLENGE_RESP:
            m = new ChallengeResponseMessage;
            break;
        case GAME_START:
            m = new GameStartMessage;
            break;
        case GAME_CANCEL:
            m = new GameCancelMessage;
            break;
        case CERT_REQ:
            m = new CertificateRequestMessage;
            break;
        case CERTIFICATE:
            m = new CertificateMessage;
            break;
        default:
            m = NULL;
            LOG(LOG_ERR, "Unrecognized message type %d", buffer[0]);
            return NULL;
    };

    ret = m->read(buffer, len);

    if (ret != 0){
        LOG(LOG_ERR, "Error reading message of type %d: %d", buffer[0], ret);
        return NULL;
    } else{
        return m;
    }
}

msglen_t StartGameMessage::write(char *buffer){
    int i = 0;
    int ret;

    if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) START_GAME_PEER)) < 0)
        return 0;
    i += ret;

    return i;
}

msglen_t StartGameMessage::read(char *buffer, msglen_t len){
    return 0;
}

msglen_t MoveMessage::write(char *buffer){
    int i = 0;
    int ret;

    if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) MOVE)) < 0)
        return 0;
    i += ret;

    if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, col)) < 0)
        return 0;
    i += ret;

    return i;
}

msglen_t MoveMessage::read(char *buffer, msglen_t len){
    int i = 1;
    int ret;

    if ((ret = readUInt8((uint8_t*)&col, &buffer[i], len-i)) < 0)
        return 1;
    i += ret;

    return 0;
}

msglen_t RegisterMessage::write(char *buffer){
    int i = 0;
    int ret;

    if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) REGISTER)) < 0)
        return 0;
    i += ret;

    if ((ret = writeUsername(&buffer[i], MAX_MSG_SIZE-i, username)) < 0)
        return 0;
    i += ret;

    return i;
}

msglen_t RegisterMessage::read(char *buffer, msglen_t len){
    int i = 1;
    int ret;

    if ((ret = readUsername(&username, &buffer[i], len-i)) < 0)
        return 1;
    i += ret;

    return 0;
}

msglen_t ChallengeMessage::write(char *buffer){
    int i = 0;
    int ret;

    if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) CHALLENGE)) < 0)
        return 0;
    i += ret;

    if ((ret = writeUsername(&buffer[i], MAX_MSG_SIZE-i, username)) < 0)
        return 0;
    i += ret;

    return i;
}
msglen_t ChallengeMessage::read(char *buffer, msglen_t len){
    int i = 1;
    int ret;

    if ((ret = readUsername(&username, &buffer[i], len-i)) < 0)
        return 1;
    i += ret;

    return 0;
}

msglen_t GameEndMessage::write(char *buffer){
    int i = 0;
    int ret;

    if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) GAME_END)) < 0)
        return 0;

    i += ret;

    return i;
}
msglen_t GameEndMessage::read(char *buffer, msglen_t len){
    return 0;
}

msglen_t UsersListMessage::write(char *buffer){
    int i = 0;
    int ret;

    if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) USERS_LIST)) < 0)
        return 0;
    i += ret;

    size_t strsize = min(usernames.size(),
                         (size_t) ((MAX_USERNAME_LENGTH+1)*MAX_USERS));
    size_t padded_size = (strsize+MAX_USERNAME_LENGTH)/(MAX_USERNAME_LENGTH+1)*(MAX_USERNA
ME_LENGTH+1);
    if ((int)padded_size > MAX_MSG_SIZE-i)
        return 0;
    strncpy(&buffer[i], usernames.c_str(), strsize);
    memset(&buffer[1+strsize], 0, padded_size-strsize+1);
    i += padded_size+1;

    return i;
}

msglen_t UsersListMessage::read(char *buffer, msglen_t len){
    int maxsize = min((MAX_USERNAME_LENGTH+1)*MAX_USERS, len-1);
    if (maxsize <= 0){
        return 1;
    }

    usernames = string(&buffer[1], maxsize);
    return 0;
}

msglen_t UsersListRequestMessage::write(char *buffer){
        int i = 0;
    int ret;

    if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) USERS_LIST_REQ)) < 0)
        return 0;
    i += ret;

    if ((ret = writeUInt32(&buffer[i], MAX_MSG_SIZE-i, offset)) < 0)
        return 0;
    i += ret;

    return i;
}

msglen_t UsersListRequestMessage::read(char *buffer, msglen_t len){
    int i = 1;
    int ret;

    if ((ret = readUInt32(&offset, &buffer[i], len-i)) < 0)
        return 1;
    i += ret;

    return 0;
}

msglen_t ChallengeForwardMessage::write(char *buffer){
    int i = 0;
    int ret;

    if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) CHALLENGE_FWD)) < 0)
        return 0;
    i += ret;

    if ((ret = writeUsername(&buffer[i], MAX_MSG_SIZE-i, username)) < 0)
        return 0;
    i += ret;

    return i;
}
msglen_t ChallengeForwardMessage::read(char *buffer, msglen_t len){
    int i = 1;
    int ret;

    if ((ret = readUsername(&username, &buffer[i], len-i)) < 0)
        return 1;
    i += ret;

    return 0;
}

msglen_t ChallengeResponseMessage::write(char *buffer){
    int i = 0;
    int ret;

    if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) CHALLENGE_RESP)) < 0)
        return 0;
    i += ret;

    if ((ret = writeBool(&buffer[i], MAX_MSG_SIZE-i, response)) < 0)
        return 0;
    i += ret;

    if ((ret = writeUInt16(&buffer[i], MAX_MSG_SIZE-i, listen_port)) < 0)
        return 0;
    i += ret;

    if ((ret = writeUsername(&buffer[i], MAX_MSG_SIZE-i, username)) < 0)
        return 0;
    i += ret;

    return i;
}
msglen_t ChallengeResponseMessage::read(char *buffer, msglen_t len){
    int i = 1;
    int ret;

    if ((ret = readBool(&response, &buffer[i], len-i)) < 0)
        return 1;
    i += ret;

    if ((ret = readUInt16(&listen_port, &buffer[i], len-i)) < 0)
        return 1;
    i += ret;

    if ((ret = readUsername(&username, &buffer[i], len-i)) < 0)
        return 1;
    i += ret;

    return 0;
}

msglen_t GameCancelMessage::write(char *buffer){
    int i = 0;
    int ret;

    if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) GAME_CANCEL)) < 0)
        return 0;
    i += ret;

    if ((ret = writeUsername(&buffer[i], MAX_MSG_SIZE-i, username)) < 0)
        return 0;
    i += ret;

    return i;
}
msglen_t GameCancelMessage::read(char *buffer, msglen_t len){
    int i = 1;
    int ret;

    if ((ret = readUsername(&username, &buffer[i], len-i)) < 0)
        return 1;
    i += ret;

    return 0;
}
```

```cpp

msglen_t GameStartMessage::write(char *buffer){
    int i = 0;
    int ret;

    if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) GAME_START)) < 0)
        return 0;
    i += ret;

    if ((ret = writeSockAddrIn(&buffer[i], MAX_MSG_SIZE-i, addr)) < 0)
        return 0;
    i += ret;

    if ((ret = writeUsername(&buffer[i], MAX_MSG_SIZE-i, username)) < 0)
        return 0;
    i += ret;

    if ((ret = cert2buf(cert, &buffer[i], MAX_MSG_SIZE - i)) < 0)
        return 0;
    i += ret;

    return i;
}

msglen_t GameStartMessage::read(char *buffer, msglen_t len){
    int i = 1;
    int ret;

    if ((ret = readSockAddrIn(&addr, &buffer[i], len-i)) < 0)
        return 1;
    i += ret;

    if ((ret = readUsername(&username, &buffer[i], len-i)) < 0)
        return 1;
    i += ret;

    if ((ret = buf2cert(&buffer[i], len - i, &cert)) < 0)
        return 1;
    i += ret;

    return 0;
}

msglen_t SecureMessage::write(char* buffer){
    int i = 0;
    int ret;

    if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) SECURE_MESSAGE)) < 0)
        return 0;
    i += ret;

    if ((ret = writeBuf(&buffer[i], MAX_MSG_SIZE-i, ct, ct_size)) < 0)
        return 0;
    i += ret;

    if ((ret = writeBuf(&buffer[i], MAX_MSG_SIZE-i, tag, TAG_SIZE)) < 0)
        return 0;
    i += ret;

    return i;
}

msglen_t SecureMessage::read(char* buffer, msglen_t len){
    ct_size = len-1-TAG_SIZE;
    ct = (char*) malloc(ct_size);
    tag = (char*) malloc(TAG_SIZE);
    if(!ct || !tag){
        LOG(LOG_WARN, "Malloc failed for message of length %d", len);
        return -1;
    }

    int i = 1;
    int ret;

    if ((ret = readBuf(ct, ct_size, &buffer[i], len-i)) < 0)
        return 1;
    i += ret;

    if ((ret = readBuf(tag, TAG_SIZE, &buffer[i], len-i)) < 0)
        return 1;
    i += ret;

    return 0;
}

msglen_t ClientHelloMessage::write(char* buffer){
    int i = 0;
    int ret;

    if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) CLIENT_HELLO)) < 0)
        return 0;
    i += ret;

    if ((ret = writeUInt32(&buffer[i], MAX_MSG_SIZE-i, nonce)) < 0)
        return 0;
    i += ret;

    if ((ret = writeUsername(&buffer[i], MAX_MSG_SIZE-i, my_id)) < 0)
        return 0;
    i += ret;

    if ((ret = writeUsername(&buffer[i], MAX_MSG_SIZE-i, other_id)) < 0)
        return 0;
    i += ret;

    if ((ret = pkey2buf(eph_key, &buffer[i], MAX_MSG_SIZE-i)) < 0)
        return 0;
    i += ret;

    return i;
}

msglen_t ClientHelloMessage::read(char* buffer, msglen_t len){
    int i = 1;
    int ret;

    if ((ret = readUInt32(&nonce, &buffer[i], len-i)) < 0)
        return 1;
    i += ret;

    if ((ret = readUsername(&my_id, &buffer[i], len-i)) < 0)
        return 1;
    i += ret;

    if ((ret = readUsername(&other_id, &buffer[i], len-i)) < 0)
        return 1;
    i += ret;

    if((ret = buf2pkey(&buffer[i], len-i, &eph_key)) < 0)
        return 1;
    i += ret;

    return 0;
}

ServerHelloMessage::~ServerHelloMessage(){
    if (ds != NULL){
        free(ds);
    }
}

msglen_t ServerHelloMessage::write(char* buffer){
    int i = 0;
    int ret;

    if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) SERVER_HELLO)) < 0)
        return 0;
    i += ret;

    if ((ret = writeUInt32(&buffer[i], MAX_MSG_SIZE-i, nonce)) < 0)
        return 0;
    i += ret;

    if ((ret = writeUsername(&buffer[i], MAX_MSG_SIZE-i, my_id)) < 0)
        return 0;
    i += ret;

    if ((ret = writeUsername(&buffer[i], MAX_MSG_SIZE-i, other_id)) < 0)
        return 0;
    i += ret;

    if ((ret = writeUInt32(&buffer[i], MAX_MSG_SIZE-i, ds_size)) < 0)
        return 0;
    i += ret;

    if ((ret = writeBuf(&buffer[i], MAX_MSG_SIZE-i, ds, ds_size)) < 0)
        return 0;
    i += ret;

    if ((ret = pkey2buf(eph_key, &buffer[i], MAX_MSG_SIZE-i)) < 0)
        return 0;
    i += ret;

    return i;
}

msglen_t ServerHelloMessage::read(char* buffer, msglen_t len){
    int i = 1;
    int ret;

    if ((ret = readUInt32(&nonce, &buffer[i], len-i)) < 0)
        return 1;
    i += ret;

    if ((ret = readUsername(&my_id, &buffer[i], len-i)) < 0)
        return 1;
    i += ret;

    if ((ret = readUsername(&other_id, &buffer[i], len-i)) < 0)
        return 1;
    i += ret;

    if ((ret = readUInt32(&ds_size, &buffer[i], len-i)) < 0)
        return 1;
    i += ret;

    ds = (char*) malloc(ds_size);
    if (!ds){
        LOG_PERROR(LOG_ERR, "Malloc failed: %s");
        return 1;
    }

    if ((ret = readBuf(ds, ds_size, &buffer[i], len-i)) < 0)
        return 1;
    i += ret;

    if((ret = buf2pkey(&buffer[i], len-i, &eph_key)) < 0)
```

```
        return 1;
    i += ret;

    return 0;
}

ClientVerifyMessage::~ClientVerifyMessage(){
    if (ds != NULL){
        free(ds);
    }
}

msglen_t ClientVerifyMessage::write(char* buffer){
    int i = 0;
    int ret;

    if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) CLIENT_VERIFY)) < 0)
        return 0;
    i += ret;

    if ((ret = writeUInt32(&buffer[i], MAX_MSG_SIZE-i, ds_size)) < 0)
        return 0;
    i += ret;

    if ((ret = writeBuf(&buffer[i], MAX_MSG_SIZE-i, ds, ds_size)) < 0)
        return 0;
    i += ret;

    return i;
}

msglen_t ClientVerifyMessage::read(char* buffer, msglen_t len){
   int i = 1;
    int ret;

    if ((ret = readUInt32(&ds_size, &buffer[i], len-i)) < 0)
        return 1;
    i += ret;

    ds = (char*) malloc(ds_size);
    if (!ds){
        LOG_PERROR(LOG_ERR, "Malloc failed: %s");
        return 1;
    }

    if ((ret = readBuf(ds, ds_size, &buffer[i], len-i)) < 0)
        return 1;
    i += ret;

    return 0;
}

msglen_t CertificateRequestMessage::write(char* buffer){
    int i = 0;
    int ret;

    if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) CERT_REQ)) < 0)
        return 0;

    i += ret;

    return i;
}

msglen_t CertificateRequestMessage::read(char* buffer, msglen_t len){
    return 0;
}

msglen_t CertificateMessage::write(char* buffer){
    int i = 0;
    int ret;

    if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) CERTIFICATE)) < 0)
        return 0;

    i += ret;

    if ((ret = cert2buf(cert, &buffer[i], MAX_MSG_SIZE-1)) < 0)
        return 0;
    i += ret;

    return i;
}

msglen_t CertificateMessage::read(char* buffer, msglen_t len){
    int ret = buf2cert(&buffer[1], len-1, &cert);
    return ret > 0 ? 0 : 1;
}
```

```
/**
 * @file socket_wrapper.h
 * @author Riccardo Mancini
 *
 * @brief Definition of the helper class "SocketWrapper" and derivatives
 *
 * @date 2020-05-17
 */

#ifndef SOCKET_WRAPPER_H
#define SOCKET_WRAPPER_H

#include <sys/socket.h>
#include <netinet/in.h>
#include "logging.h"
#include "network/messages.h"
#include "network/host.h"

/**
 * Wrapper class around sockaddr_in and socket descriptor
 *
 * It provides a more simple interface saving a lot of boiler-plate code.
 * There are two subclasses: ClientSocketWrapper and ServerSocketWrapper.
 */
class SocketWrapper{
protected:
    /** Other host inet socket */
    struct sockaddr_in other_addr;

    /** Socket file descriptor */
    int socket_fd;

    /** Pre-allocated buffer for incoming messages */
    char buffer_in[MAX_MSG_SIZE];

    /** Pre-allocated buffer for outgoing messages */
    char buffer_out[MAX_MSG_SIZE];

    /** Index in the buffer that has been read up to now */
    msglen_t buf_idx;
public:
    /**
     * Initialize on a new socket
     */
    SocketWrapper();

    /**
     * Initialize using existing socket
     */
    SocketWrapper(int sd) : socket_fd(sd), buf_idx(0) {}

    ~SocketWrapper(){closeSocket();}

    /**
     * Returns current socket file descriptor
     */
    int getDescriptor(){return socket_fd;};

    /**
     * Read any new data from the socket but does not wait for the
     * whole message to be ready.
     *
     * This API is blocking iff socket was not ready.
     *
     * @param size the size of the temporary buffer
     * @returns the received message or null if an error occurred
     */
    Message* readPartMsg();

    /**
     * Receive any new message from the socket.
     *
     * This API is blocking.
     *
     * @returns the received message or null if an error occurred
     */
    Message* receiveAnyMsg();

    /**
     * Receive a new message of the given type from the socket.
     *
     * When a message of the wrong type is received it is simply ignored.
     *
     * This API is blocking.
     *
     * @param type the type to keep
     * @returns the received message or null if an error occurred
     */
    Message* receiveMsg(MessageType type);

    /**
     * Receive a new message of any of the given types from the socket.
     *
     * When a message of the wrong type is received it is simply ignored.
     *
     * This API is blocking.
     *
     * @param type the types to keep (array)
     * @param n_types the number of types to keep (array length)
     * @returns the received message or null if an error occurred
     */
    Message* receiveMsg(MessageType type[], int n_types);

    /**
     * Sends the given message to the peer host through the socket.
     *
     * @param msg the message to be sent
     * @returns 0 in case of success, something else otherwise
     */
    int sendMsg(Message *msg);

    /**
     * Closes the socket.
     */
    void closeSocket();

    /**
     * Sets the address of the other host.
     *
     * This is used when initializing a new SocketWrapper for a newly
     * accepter connection.
     */
    void setOtherAddr(struct sockaddr_in addr){other_addr = addr;}

    sockaddr_in* getOtherAddr() { return &other_addr;}

    /**
     * Returns connected host.
     */
    Host getConnectedHost(){return Host(other_addr);}
};

/**
 * SocketWrapper for a TCP client
 *
 * It provides a new function to connect to server.
 */
class ClientSocketWrapper : public SocketWrapper{
```

```
public:
    /**
     * Connects to a remote server.
     *
     * @returns 0 in case of success, something else otherwise
     */
    int connectServer(Host host);
};

/**
 * SocketWrapper for a TCP server
 *
 * It provides a new function to accept clients.
 * Constructor also set listen mode.
 */
class ServerSocketWrapper : public SocketWrapper{
private:
    /** Local inet socket */
    struct sockaddr_in my_addr;
public:
    /**
     * Binds the socket to the requested port.
     *
     * @param port the port you want to bind on
     * @returns 0 in case of success
     * @returns 1 otherwise
     */
    int bindPort(int port);

    /**
     * Binds the socket to a random port.
     *
     * @returns 0 in case of success
     * @returns 1 otherwise
     */
    int bindPort();

    /**
     * Accepts any incoming connection and returns the related SocketWrapper.
     */
    SocketWrapper* acceptClient();

    /**
     * Returns port the server is listening new connections on.
     */
    int getPort(){return ntohs(my_addr.sin_port);}
};

#endif // SOCKET_WRAPPER_Hln
```

```
 /**
 * @file socket_wrapper.cpp
 * @author Riccardo Mancini
 *
 * @brief Implementation of socket_wrapper.h
 *
 * @see socket_wrapper.h
 */

#include <assert.h>
#include "logging.h"
#include "network/socket_wrapper.h"
#include "utils/dump_buffer.h"
#include "network/inet_utils.h"

SocketWrapper::SocketWrapper() {
    socket_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (socket_fd < 0){
        LOG_PERROR(LOG_ERR, "Error creating socket: %s");
        return;
    }
    buf_idx = 0;
}
Message* SocketWrapper::readPartMsg(){
    int len;
    msglen_t msglen = 0;

    if (buf_idx < sizeof(msglen)){ // I first need to read msglen
        // read available message
        len = read(socket_fd, buffer_in+buf_idx, sizeof(msglen)-buf_idx);
        // I will read rest of it in another moment since I do not know
        // whether other data is available.
        // TODO: find a way to tell whether socket has other data
    } else{
        // read msg length
        msglen = MSGLEN_NTOH(*((msglen_t*)buffer_in));
        assert(msglen <= MAX_MSG_SIZE); // must not happen

        // read up to msg length
        len = read(socket_fd, buffer_in+buf_idx, msglen-buf_idx);
    }

    DUMP_BUFFER_HEX_DEBUG(buffer_in, len);

    if (len < 0){
        LOG_PERROR(LOG_ERR, "Error reading from socket: %s");
        throw "Error reading from socket";
    } else if (len == 0){
        throw "Connection lost";
    }

    // buf_idx is also the number of read bytes up to now
    buf_idx += len;

    if (buf_idx < sizeof(msglen)){
        LOG(LOG_DEBUG, "Too few bytes recevied from socket: %d < %lu",
            buf_idx, sizeof(msglen));
        return NULL;
    }

    // read msg length
    msglen = MSGLEN_NTOH(*((msglen_t*)buffer_in));

    if (msglen > MAX_MSG_SIZE){
        throw("Message is too big");
    }

    if (buf_idx != msglen){
        LOG(LOG_DEBUG, "Too few bytes received from socket: %d < %d",
            buf_idx, msglen);
        return NULL;
    }

    Message *m = readMessage(buffer_in+sizeof(msglen), msglen-sizeof(msglen));

    // reset buffer
    buf_idx = 0;

    return m;
}

Message* SocketWrapper::receiveAnyMsg(){
    int len;
    msglen_t msglen;

    // read msg length
    len = recv(socket_fd, buffer_in, sizeof(msglen), MSG_WAITALL);

    DUMP_BUFFER_HEX_DEBUG(buffer_in, len);

    if (len == 0){
        throw "Connection lost";
    } else if (len != sizeof(msglen)){
        LOG(LOG_ERR, "Too few bytes recevied from socket: %d < %lu",
            len, sizeof(msglen));
        return NULL;
    }

    // read msg payload
    msglen = MSGLEN_NTOH(*((msglen_t*)buffer_in));

    if (msglen > MAX_MSG_SIZE){
        throw("Message is too big");
    }

    len += recv(socket_fd, buffer_in+len, msglen-len, MSG_WAITALL);

    DUMP_BUFFER_HEX_DEBUG(buffer_in, len);

    if (len == 0){
        throw "Received EOF";
    } else if (len != msglen){
        LOG(LOG_ERR, "Too few bytes recevied from socket: %d < %d",
            len, msglen);
        return NULL;
    }

    Message *m = readMessage(buffer_in+sizeof(msglen), msglen-sizeof(msglen));

    return m;
}

Message* SocketWrapper::receiveMsg(MessageType type){
    return this->receiveMsg(&type, 1);
}

Message* SocketWrapper::receiveMsg(MessageType type[], int n_types){
    Message *m = NULL;
    while (m == NULL){
        try{
            m = receiveAnyMsg();
        } catch(const char* msg){
            LOG(LOG_ERR, "%s", msg);
            return NULL;
        }
        if (m != NULL){
            for (int i = 0; i < n_types; i++){
                if (m->getType() == type[i]){
                    return m;
                }
            }
            LOG(LOG_WARN, "Received unexpected message of type %s",  m->getName().c_str())
;
        }
    }
    //TODO: add timeout?
    return NULL;
}


int SocketWrapper::sendMsg(Message *msg){
    msglen_t msglen, pktlen;
    int len;

    msglen = msg->write(buffer_out+sizeof(msglen));
    if (msglen == 0)
        return 1;

    pktlen = msglen + sizeof(msglen);
    *((msglen_t*)buffer_out) = MSGLEN_HTON(pktlen);

    LOG(LOG_DEBUG, "Sending %s", msg->getName().c_str());

    DUMP_BUFFER_HEX_DEBUG(buffer_out, pktlen);

    len = send(socket_fd, buffer_out, pktlen, 0);
    if (len != pktlen){
        LOG(LOG_ERR, "Error sending %s: len (%d) != msglen (%d)",
            msg->getName().c_str(),
            len,
            msglen
        );
        return 1;
    }

    LOG(LOG_DEBUG, "Sent message %s", msg->getName().c_str());

    return 0;
}

void SocketWrapper::closeSocket(){
    close(socket_fd);
}

int ClientSocketWrapper::connectServer(Host host){
    int ret;

    other_addr = host.getAddress();

    ret = connect(
        socket_fd,
        (struct sockaddr*) &other_addr,
        sizeof(other_addr)
    );

    if (ret != 0){
        LOG_PERROR(LOG_ERR, "Error connecting to %s: %s",
            sockaddr_in_to_string(host.getAddress()).c_str());
        return ret;
    }

    return ret;
}

int ServerSocketWrapper::bindPort(){
    my_addr = make_my_sockaddr_in(0);
    int ret = bind_random_port(socket_fd, &my_addr);
    if (ret <= 0){
        LOG_PERROR(LOG_ERR, "Error in binding: %s");
        return ret;
    }

    ret = listen(socket_fd, 10);
    if (ret != 0){
        LOG_PERROR(LOG_ERR, "Error in setting socket to listen mode: %s");
    }

    return ret;
}

int ServerSocketWrapper::bindPort(int port){
    my_addr = make_my_sockaddr_in(port);
    int ret = bind(socket_fd, (struct sockaddr*) &my_addr, sizeof(my_addr));
    if (ret != 0){
        LOG_PERROR(LOG_ERR, "Error in binding: %s");
        return ret;
    }

    ret = listen(socket_fd, 10);
    if (ret != 0){
        LOG_PERROR(LOG_ERR, "Error in setting socket to listen mode: %s");
    }

    return ret;
}

SocketWrapper* ServerSocketWrapper::acceptClient(){
    socklen_t len = sizeof(other_addr);
    int new_sd = accept(
        socket_fd,
        (struct sockaddr*) &other_addr,
        &len
    );

    SocketWrapper *sw = new SocketWrapper(new_sd);
    sw->setOtherAddr(other_addr);
    return sw;
}
```

```
/**
 * @file secure_socket_wrapper.h
 * @author Mirko Laruina
 *
 * @brief Header file for SecureSocketWrapper
 *
 * @date 2020-06-09
 */

#ifndef SECURE_SOCKET_WRAPPER_H
#define SECURE_SOCKET_WRAPPER_H

#include <sys/socket.h>
#include <netinet/in.h>
#include "logging.h"
#include "network/messages.h"
#include "network/socket_wrapper.h"
#include "security/crypto.h"
#include "security/crypto_utils.h"
#include "security/secure_host.h"
#include "utils/dump_buffer.h"

#define MAX_MSG_TO_SIGN_SIZE (2*MAX_USERNAME_LENGTH + 2 * sizeof(nonce_t) + 2 * KEY_BIO_MA
X_SIZE )
#define MAX_SEC_MSG_SIZE (MAX_MSG_SIZE - TAG_SIZE - sizeof(msglen_t) - 1)

#define AAD_SIZE (sizeof(msglen_t) + 1)

class SecureSocketWrapper
{
protected:
    SocketWrapper *sw;

    char send_key[KEY_SIZE];
    char recv_key[KEY_SIZE];
    char send_iv_static[IV_SIZE];
    char recv_iv_static[IV_SIZE];
    char send_iv[IV_SIZE];
    char recv_iv[IV_SIZE];
    uint64_t send_seq_num;
    uint64_t recv_seq_num;
    string my_id;
    string other_id;
    nonce_t sv_nonce;
    nonce_t cl_nonce;
    EVP_PKEY *my_eph_key;
    EVP_PKEY *other_eph_key;
    X509 *my_cert;
    X509 *other_cert;
    X509_STORE *store;
    EVP_PKEY *my_priv_key;

    bool peer_authenticated;

    char msg_to_sign_buf[MAX_MSG_TO_SIGN_SIZE];

    /**
     * Empty constructor to use in child classes.
     */
    SecureSocketWrapper(){};

    /**
     * @brief Derives the key
     *
     * @param role           Role in the communication
     */
    void generateKeys(const char *role);

    /**
```

```
     * @brief Calculates the IV to use when sending the next message
     *
     */
    void updateSendIV();

    /**
     * @brief Calculates the IV to use when receiving a new message
     *
     */
    void updateRecvIV();

    /** Internal initialization */
    void init(X509 *cert, EVP_PKEY *my_priv_key, X509_STORE *store);

    /**
     * @brief Decrypts a Secure Message into a Message
     *
     * @param sm             Secure message ptr
     * @return Message*      Read message
     */
    Message *decryptMsg(SecureMessage *sm);

    /**
     * @brief Encrypts a Message into a SecureMessage
     *
     * @param m                   Message to encrypt
     * @return SecureMessage*     Encrypted SecureMessage
     */
    SecureMessage *encryptMsg(Message *m);

    /**
     * Make the signature for the handshake protocol
     */
    int makeSignature(const char *role, char** ds);

    /**
     * Checks the signature for the handshake protocol
     */
    bool checkSignature(char *ds, size_t ds_size, const char *role);

    /**
     * Builds the message to be signed.
     *
     * @param role the role of this peer
     * @param msg the buffer to write the message to
     * @returns number of written bytes
     */
    int buildMsgToSign(const char *role, char *msg);

    /**
     * Builds the aad of a message.
     *
     * I.e. this function builds the message header as SocketWrapper would.
     *
     * @param msg_type the type of the message
     * @param len the length of the message
     * @param aad the aad buffer to write to (it must be AAD_SIZE long)
     * @returns number of written bytes
     *
     * @see AAD_SIZE
     */
    void makeAAD(MessageType msg_type, msglen_t len, char* aad);

public:
    /**
     * Initialize on a new socket
     */
    SecureSocketWrapper(X509 *cert, EVP_PKEY *my_priv_key, X509_STORE *store);

    /**
     * Initialize using existing socket
     */
    SecureSocketWrapper(X509 *cert, EVP_PKEY *my_priv_key, X509_STORE *store, int sd);

    /**
     * Constructor to generate connection socket wrappers
     */
    SecureSocketWrapper(X509 *cert, EVP_PKEY *my_priv_key, X509_STORE *store, SocketWrappe
r *sw);

    /**
     * Destructor
     */
    ~SecureSocketWrapper();

    /**
     * Read any new data from the socket but does not wait for the
     * whole message to be ready. This does not decrypt the message!
     *
     * This API is blocking iff socket was not ready.
     *
     * @param size the size of the temporary buffer
     * @returns the received message or null if an error occurred
     */
    Message *readPartMsg();

    /**
     * Receive any new message from the socket.
     *
     * This API is blocking.
     *
     * @returns the received message or null if an error occurred
     */
    Message *receiveAnyMsg();

    /**
     * Receive a new message of the given type from the socket.
     *
     * When a message of the wrong type is received it is simply ignored.
     *
     * This API is blocking.
     *
     * @param type the type to keep
     * @returns the received message or null if an error occurred
     */
    Message *receiveMsg(MessageType type);

    /**
     * Receive a new message of any of the given types from the socket.
     *
     * When a message of the wrong type is received it is simply ignored.
     *
     * This API is blocking.
     *
     * @param type the types to keep (array)
     * @param n_types the number of types to keep (array length)
     * @returns the received message or null if an error occurred
     */
    Message *receiveMsg(MessageType type[], int n_types);

    Message *handleMsg(Message *msg);


    int sendCertRequest();

    int handleCertResponse(CertificateMessage* cm);

    int handleClientHello(ClientHelloMessage *chm);
    int handleServerHello(ServerHelloMessage *shm);
    int handleClientVerify(ClientVerifyMessage *cvm);

    int sendClientHello();
    int sendServerHello();
    int sendClientVerify();

    int sendPlain(Message *msg);
    /**
     * Sends the given message to the peer host through the socket.
     *
     * @param msg the message to be sent
     * @returns 0 in case of success, something else otherwise
     */
    int sendMsg(Message *msg);

    /**
     * @brief Estiblishes a secure connection over the already specified socket. To be run server-side.
     *
     * @return int  0 in case of success, something else otherwise
     */
    int handshakeServer();

    /**
     * @brief Estiblishes a secure connection over the already specified socket. To be run client-side.
     *
     * @return int  0 in case of success, something else otherwise
     */
    int handshakeClient();

    /**
     * Sets the peer certificate.
     */
    bool setOtherCert(X509 *other_cert);

    /**
     * Returns current socket file descriptor
     */
    int getDescriptor() { return sw->getDescriptor(); };

    /**
     * Closes the socket.
     */
    void closeSocket() { sw->closeSocket(); }

    /**
     * Sets the address of the other host.
     *
     * This is used when initializing a new SocketWrapper for a newly
     * accepter connection.
     */
    void setOtherAddr(struct sockaddr_in addr) { sw->setOtherAddr(addr); }

    sockaddr_in *getOtherAddr() { return sw->getOtherAddr(); }

    /**
     * Returns connected host.
     */
    SecureHost getConnectedHost() { return SecureHost(*getOtherAddr(), other_cert); }

    /**
     * Returns the certificate of this host
     */
    X509* getCert(){ return my_cert;}
};

/**
```

```
 * SocketWrapper for a TCP client
 *
 * It provides a new function to connect to server.
 */
class ClientSecureSocketWrapper : public SecureSocketWrapper
{
private:
    ClientSocketWrapper *csw;

public:
    /**
     * Initialize a new socket on a random port.
     *
     * @param port the port you want to bind on
     */
    ClientSecureSocketWrapper(X509 *cert, EVP_PKEY *my_priv_key, X509_STORE *store);

    /**
     * Connects to a remote server.
     *
     * @returns 0 in case of success, something else otherwise
     */
    int connectServer(SecureHost host);

};

/**
 * SocketWrapper for a TCP server
 *
 * It provides a new function to accept clients.
 * Constructor also set listen mode.
 */
class ServerSecureSocketWrapper : public SecureSocketWrapper
{
private:
    ServerSocketWrapper *ssw;

public:
    /**
     * Initialize a new socket on a random port.
     *
     * @param port the port you want to bind on
     */
    ServerSecureSocketWrapper(X509 *cert, EVP_PKEY *my_priv_key, X509_STORE *store);

   /**
     * Binds the socket to the requested port.
     *
     * @param port the port you want to bind on
     * @returns 0 in case of success
     * @returns 1 otherwise
     */
    int bindPort(int port) { return ssw->bindPort(port); }

    /**
     * Binds the socket to a random port.
     *
     * @returns 0 in case of success
     * @returns 1 otherwise
     */
    int bindPort() { return ssw->bindPort(); }

    /**
     * Accepts any incoming connection and returns the related SocketWrapper.
     */
    SecureSocketWrapper *acceptClient();

    /**
     * Accepts any incoming connection and returns the related SocketWrapper.
     *
     * The certificate is set as the expected certificate of the peer.
     */
    SecureSocketWrapper *acceptClient(X509 *other_cert);

    /**
     * Returns port the server is listening new connections on.
     */
    int getPort() { return ssw->getPort(); }
};

#endif
```

```
#include "security/secure_socket_wrapper.h"
#include "security/crypto_utils.h"

SecureSocketWrapper::SecureSocketWrapper(X509* cert, EVP_PKEY* my_priv_key, X509_STORE* store)
{
    sw = new SocketWrapper();
    init(cert, my_priv_key, store);
}

SecureSocketWrapper::SecureSocketWrapper(X509* cert, EVP_PKEY* my_priv_key, X509_STORE* store, int sd)
{
    sw = new SocketWrapper(sd);
    init(cert, my_priv_key, store);
}

SecureSocketWrapper::SecureSocketWrapper(X509* cert, EVP_PKEY* my_priv_key, X509_STORE* store, SocketWrapper *sw)
{
    this->sw = sw;
    init(cert, my_priv_key, store);
}

void SecureSocketWrapper::init(X509* cert, EVP_PKEY* my_priv_key, X509_STORE* store){
    send_seq_num = 0;
    recv_seq_num = 0;
    cl_nonce = 0;
    sv_nonce = 0;
    my_eph_key = NULL;
    other_eph_key = NULL;
    this->my_cert = cert;
    this->my_priv_key = my_priv_key;
    this->store = store;
    other_cert = NULL;
    peer_authenticated = false;
    my_id = usernameFromCert(cert);
}

SecureSocketWrapper::~SecureSocketWrapper(){
    delete sw;

    // TODO do not wait this much time to delete them
    if (my_eph_key != NULL){
        EVP_PKEY_free(my_eph_key);
    }
    if (other_eph_key != NULL){
        EVP_PKEY_free(other_eph_key);
    }
    // TODO free certs too?
}

Message *SecureSocketWrapper::decryptMsg(SecureMessage *sm)
{
    if (!peer_authenticated){
        LOG(LOG_WARN, "Unauthenticated peer sent encrypted message");
        return NULL;
    }
    int ret;

    msglen_t pt_len = sm->getCtSize();
    LOG(LOG_DEBUG, "Received SecureMessage of size %d", pt_len);
    if (pt_len > MAX_SEC_MSG_SIZE){
        LOG(LOG_ERR, "Message is too big");
        return NULL;
    }

    LOG(LOG_DEBUG, "Payload");
    DUMP_BUFFER_HEX_DEBUG(sm->getCt(), pt_len);
    LOG(LOG_DEBUG, "TAG");
    DUMP_BUFFER_HEX_DEBUG(sm->getTag(), TAG_SIZE);

    char *buffer_pt = (char*) malloc(pt_len);
    char buffer_aad[AAD_SIZE];

    if (!buffer_pt){
        LOG_PERROR(LOG_ERR, "Malloc failed: %s");
        return NULL;
    }

    makeAAD(SECURE_MESSAGE, pt_len+TAG_SIZE+AAD_SIZE, buffer_aad);
    DUMP_BUFFER_HEX_DEBUG(buffer_aad, AAD_SIZE);

    updateRecvIV();

    try{
        ret = aes_gcm_decrypt(sm->getCt(), pt_len, buffer_aad, AAD_SIZE,
                                recv_key, recv_iv,
                                buffer_pt, sm->getTag());
    } catch (const char *err_msg){
        LOG(LOG_ERR, "Error: %s", err_msg);
        free(buffer_pt);
        return NULL;
    }

    if (ret <= 0)
    {
        LOG(LOG_ERR, "Could not decrypt the message");
        return NULL;
    }

    LOG(LOG_DEBUG, "Decrypted message (%d):", ret);
    DUMP_BUFFER_HEX_DEBUG(buffer_pt, ret);

    Message *m = readMessage(buffer_pt, pt_len);

    free(buffer_pt);

    if (m != NULL){
        LOG(LOG_INFO, "Decrypted message of type %s", m->getName().c_str());
    } else{
        LOG(LOG_WARN, "Malformed message");
    }
    return m;
}

SecureMessage *SecureSocketWrapper::encryptMsg(Message *m)
{
    if (!peer_authenticated)
        return NULL;
    int ret;

    char buffer_pt[MAX_MSG_SIZE];
    msglen_t buf_len = m->write(buffer_pt);

    if (buf_len > MAX_SEC_MSG_SIZE){
        LOG(LOG_ERR, "Message is too big: %s", m->getName().c_str());
        return NULL;
    }

    char* buffer_ct = (char*) malloc(MAX_MSG_SIZE);
    char* buffer_tag = (char*) malloc(TAG_SIZE);
    char  buffer_aad[AAD_SIZE];

    if (!buffer_ct || !buffer_tag){
        LOG_PERROR(LOG_ERR, "Malloc failed: %s");
        return NULL;
    }

    LOG(LOG_DEBUG, "Encrypting message of size %d", buf_len);
    DUMP_BUFFER_HEX_DEBUG(buffer_pt, IV_SIZE);

    updateSendIV();
    makeAAD(SECURE_MESSAGE, buf_len+TAG_SIZE+AAD_SIZE, buffer_aad);
    DUMP_BUFFER_HEX_DEBUG(buffer_aad, AAD_SIZE);
    try{
        ret = aes_gcm_encrypt(buffer_pt, buf_len, buffer_aad, AAD_SIZE,
                              send_key, send_iv,
                              buffer_ct, buffer_tag);
    } catch(const char* err_msg){
        LOG(LOG_ERR, "Error: %s", err_msg);
        free(buffer_ct);
        free(buffer_tag);
        return NULL;
    }

    LOG(LOG_DEBUG, "Message encrypted %d bytes with iv: ", ret);
    DUMP_BUFFER_HEX_DEBUG(send_iv, IV_SIZE);
    LOG(LOG_DEBUG, "and tag: ");
    DUMP_BUFFER_HEX_DEBUG(buffer_tag, TAG_SIZE);
    LOG(LOG_DEBUG, "SecureMessage of size %d", buf_len + 1 + TAG_SIZE);

    if (ret <= 0)
    {
        LOG(LOG_ERR, "Could not encrypt the message");
        return NULL;
    }

    SecureMessage *sm = new SecureMessage(buffer_ct, ret, buffer_tag);
    return sm;
}

void SecureSocketWrapper::makeAAD(MessageType msg_type, msglen_t len, char* aad){
    *((msglen_t*)aad) = MSGLEN_HTON(len);
    aad[2] = msg_type;
}

Message *SecureSocketWrapper::readPartMsg()
{
    return sw->readPartMsg();
}

Message *SecureSocketWrapper::receiveAnyMsg()
{
    Message *m = sw->receiveAnyMsg();

    return handleMsg(m);
}

Message *SecureSocketWrapper::handleMsg(Message* msg)
{
    Message* dm;
    if (msg == NULL)
        return NULL;
    switch(msg->getType()){
        case SECURE_MESSAGE:
            dm = decryptMsg((SecureMessage*) msg);
            if (dm != NULL)
                recv_seq_num++;
            delete msg;
            return dm;
        case CLIENT_HELLO:
            if (cl_nonce == 0 && !peer_authenticated){
                return msg;
            } else{
                // already received
                LOG(LOG_WARN, "Client sent CLIENT_HELLO twice");
                return NULL;
            }
        case SERVER_HELLO:
            if (!peer_authenticated){
                return msg;
            } else{
                // already authenticated!
                LOG(LOG_WARN, "Server sent SERVER_HELLO twice");
                return NULL;
            }
        case CLIENT_VERIFY:
            if (!peer_authenticated){
                return msg;
            } else{
                // already authenticated
                LOG(LOG_WARN, "Client sent CLIENT_VERIFY twice");
                return NULL;
            }
        case CERTIFICATE:
        case CERT_REQ:
            return msg;
        default:
            LOG(LOG_WARN, "Peer sent %s in cleartext!", msg->getName().c_str());
            return NULL;
    }
}

int SecureSocketWrapper::handleClientHello(ClientHelloMessage* chm)
{
    cl_nonce = chm->getNonce();
    other_eph_key = chm->getEphKey();
    return sendServerHello();
}

int SecureSocketWrapper::handleServerHello(ServerHelloMessage* shm)
{
    sv_nonce = shm->getNonce();
    other_eph_key = shm->getEphKey();

    //Deriving the symmetric key
    generateKeys("client");

    bool check = checkSignature(shm->getDs(), shm->getDsSize(), "client");
    if (!check){
        LOG(LOG_ERR, "Digital Signature verification failure!");
        return -1;
    }

    peer_authenticated = true;

    return sendClientVerify();
}

int SecureSocketWrapper::handleClientVerify(ClientVerifyMessage* cvm)
{
    bool check = checkSignature(cvm->getDs(), cvm->getDsSize(), "server");
    if (!check){
        LOG(LOG_ERR, "Digital Signature verification failure!");
        return -1;
    }

    LOG(LOG_INFO, "Digital Signature verification succeded!");

    peer_authenticated = true;

    return 0;
}

int SecureSocketWrapper::sendPlain(Message *msg)
```

```
{
    int ret = sw->sendMsg(msg);
    if(ret == 0){
        return 0;
    }
    return 1;
}

int SecureSocketWrapper::sendMsg(Message *msg)
{
    SecureMessage *sm = encryptMsg(msg);
    if (!sm)
    {
        return 1;
    }
    int ret = sw->sendMsg(sm);
    delete sm;
    if (ret == 0){
        send_seq_num++;
        return 0;
    } else{
        return 1;
    }
}

int SecureSocketWrapper::sendCertRequest(){
    CertificateRequestMessage crm;
    return sw->sendMsg(&crm);
}

int SecureSocketWrapper::sendClientHello(){
    cl_nonce = get_rand();
    get_ecdh_key(&my_eph_key);

    ClientHelloMessage chm(my_eph_key, cl_nonce, my_id, other_id);
    return sw->sendMsg(&chm);
}

int SecureSocketWrapper::sendServerHello(){
    sv_nonce = get_rand();
    get_ecdh_key(&my_eph_key);

    //Deriving the symmetric key
    generateKeys("server");

    char *ds = NULL;
    int ret = makeSignature("server", &ds);
    if (ret > 0){
        ServerHelloMessage shm(my_eph_key, sv_nonce, my_id, other_id, ds, ret);
        return sw->sendMsg(&shm);
    } else {
        return ret;
    }
}

int SecureSocketWrapper::sendClientVerify(){
    char *ds = NULL;
    int ret = makeSignature("client", &ds);
    if (ret > 0){
        ClientVerifyMessage cvm(ds, ret);
        return sw->sendMsg(&cvm);
    } else{
        return ret;
    }
}

void SecureSocketWrapper::generateKeys(const char* role){
    char *shared_secret = NULL;

    int size = dhke(my_eph_key, other_eph_key, &shared_secret);

    const char* other_role;
    if (strcmp(role, "client") == 0){
        other_role = "server";
    } else if (strcmp(role, "server") == 0){
        other_role = "client";
    } else{
        LOG(LOG_ERR, "Wrong role %s", role);
    }
    char my_key_str[11] = "key_";
    char other_key_str[11] = "key_";
    char my_iv_str[11] = "iv__";
    char other_iv_str[11] = "iv__";
    strcat(my_key_str, role);
    strcat(other_key_str, other_role);
    strcat(my_iv_str, role);
    strcat(other_iv_str, other_role);

    hkdf(shared_secret, size, sv_nonce, cl_nonce, my_key_str, send_key, KEY_SIZE);
    hkdf(shared_secret, size, sv_nonce, cl_nonce, other_key_str, recv_key, KEY_SIZE);
    hkdf(shared_secret, size, sv_nonce, cl_nonce, my_iv_str, send_iv_static, IV_SIZE);
    hkdf(shared_secret, size, sv_nonce, cl_nonce, other_iv_str, recv_iv_static, IV_SIZE);

    LOG(LOG_DEBUG, "HKDF parameters BEGIN --------");
    LOG(LOG_DEBUG, "Shared secret:");
    DUMP_BUFFER_HEX_DEBUG(shared_secret, size);
    LOG(LOG_DEBUG, "sv_nonce=%d", sv_nonce);
    LOG(LOG_DEBUG, "cl_nonce=%d", cl_nonce);
    LOG(LOG_DEBUG, "HKDF parameters END --------");
    LOG(LOG_DEBUG, "Generated keys BEGIN --------");
    LOG(LOG_DEBUG, "Send key (%s):", my_key_str);
    DUMP_BUFFER_HEX_DEBUG(send_key, KEY_SIZE);
    LOG(LOG_DEBUG, "Send IV (%s):", my_iv_str);
    DUMP_BUFFER_HEX_DEBUG(send_iv_static, IV_SIZE);
    LOG(LOG_DEBUG, "Recv key (%s):", other_key_str);
    DUMP_BUFFER_HEX_DEBUG(recv_key, KEY_SIZE);
    LOG(LOG_DEBUG, "Recv IV (%s):", other_iv_str);
    DUMP_BUFFER_HEX_DEBUG(recv_iv_static, IV_SIZE);
    LOG(LOG_DEBUG, "Generated keys END --------");

    free(shared_secret);
}

int SecureSocketWrapper::buildMsgToSign(const char* role, char* msg){
    int i = 0;
    size_t size;
    string A;
    string B;
    EVP_PKEY *A_eph_key;
    EVP_PKEY *B_eph_key;
    if (strcmp(role, "server") == 0){
        A = other_id;
        A_eph_key = other_eph_key;
        B = my_id;
        B_eph_key = my_eph_key;
    } else if (strcmp(role, "client") == 0){
        A = my_id;
        A_eph_key = my_eph_key;
        B = other_id;
        B_eph_key = other_eph_key;
    } else{
        LOG(LOG_ERR, "Unrecognized role %s", role);
        return -1;
    }

    size = min((int)A.size(),MAX_USERNAME_LENGTH);
    strncpy(&msg[i],
         A.c_str(),
            size);
    i += size;

    size = min((int)B.size(),MAX_USERNAME_LENGTH);
    strncpy(&msg[i],
            B.c_str(),
            size);
    i += size;

    size = sizeof(nonce_t);
    memcpy(&msg[i], &cl_nonce, size);
    i += size;

    size = sizeof(nonce_t);
    memcpy(&msg[i], &sv_nonce, size);
    i += size;

    size = pkey2buf(A_eph_key, &msg[i], MAX_MSG_TO_SIGN_SIZE-i);
    if (size <= 0){
        LOG(LOG_ERR, "Error copying key to buffer");
        return 0;
    }
    i += size;

    size = pkey2buf(B_eph_key, &msg[i], MAX_MSG_TO_SIGN_SIZE-i);
    if (size <= 0){
        LOG(LOG_ERR, "Error copying key to buffer");
        return 0;
    }
    i += size;

    return i;
}

int SecureSocketWrapper::makeSignature(const char *role, char** ds){
    size_t msglen = buildMsgToSign(role, msg_to_sign_buf);

    if (msglen <= 0){
        LOG(LOG_ERR, "Error building message to sign!");
        return -1;
    }

    return dsa_sign(msg_to_sign_buf, msglen, ds, my_priv_key);
}

bool SecureSocketWrapper::checkSignature(char* ds, size_t ds_size, const char* role){
    size_t msglen = buildMsgToSign(role, msg_to_sign_buf);

    if (msglen <= 0){
        LOG(LOG_ERR, "Error building message to sign!");
        return false;
    }

    bool ret = dsa_verify(msg_to_sign_buf, msglen, ds, ds_size,
                          X509_get_pubkey(other_cert));

    return ret;
}

void updateIV(uint64_t seq, char* iv_static, char* iv){
    char* seq_bytes = (char*) &seq;
    for (size_t i=0; i<IV_SIZE; i++){
        if (i<sizeof(seq)){
            iv[i] = iv_static[i] ^ seq_bytes[i];
        } else{
            iv[i] = iv_static[i];
        }
    }
}
```

```

void SecureSocketWrapper::updateSendIV(){
    updateIV(send_seq_num, send_iv_static, send_iv);
}

void SecureSocketWrapper::updateRecvIV(){
    updateIV(recv_seq_num, recv_iv_static, recv_iv);
}

int SecureSocketWrapper::handshakeClient(){
    if(sendClientHello() != 0){
        LOG(LOG_ERR, "ClientHello send failed!");
        return 1;
    }
    LOG(LOG_INFO, "Client Hello sent");
    ServerHelloMessage *shm = dynamic_cast<ServerHelloMessage*>(receiveMsg(SERVER_HELLO));
    if (shm == NULL || handleServerHello(shm) != 0){
        LOG(LOG_ERR, "Error handling ServerHello!");
        return 1;
    } else{
        LOG(LOG_INFO, "Server Hello handled");
        return 0;
    }
}

int SecureSocketWrapper::handshakeServer(){
    ClientHelloMessage *chm = dynamic_cast<ClientHelloMessage*>(receiveMsg(CLIENT_HELLO));

    if (chm == NULL || handleClientHello(chm) != 0){
        LOG(LOG_ERR, "Error handling ClientHello!");
        return 1;
    }

    ClientVerifyMessage *cvm = dynamic_cast<ClientVerifyMessage*>(receiveMsg(CLIENT_VERIFY
));

    if (handleClientVerify(cvm) != 0){
        LOG(LOG_ERR, "Error handling ClientVerify!");
        return 1;
    }

    return 0;
}

bool SecureSocketWrapper::setOtherCert(X509* other_cert){
    if (!verify_peer_cert(store, other_cert)){
        LOG(LOG_ERR, "Peer certificate validation failed!");
        return false;
    }
    this->other_cert = other_cert;
    this->other_id = usernameFromCert(other_cert);
    return true;
}

Message* SecureSocketWrapper::receiveMsg(MessageType type){
    return this->receiveMsg(&type, 1);
}

Message* SecureSocketWrapper::receiveMsg(MessageType type[], int n_types){
    Message *m = NULL;
    while (m == NULL){
        try{
            m = receiveAnyMsg();
        } catch(const char* msg){
            LOG(LOG_ERR, "%s", msg);
            return NULL;
        }
        if (m != NULL){
            for (int i = 0; i < n_types; i++){
                if (m->getType() == type[i]){
                    return m;
                }
            }
            LOG(LOG_WARN, "Received unexpected message of type %s",  m->getName().c_str());
        }
    }
    //TODO: add timeout?
    return NULL;
}

ClientSecureSocketWrapper::ClientSecureSocketWrapper(X509* cert, EVP_PKEY* my_priv_key, X509_STORE* store){
    csw = new ClientSocketWrapper();
    sw = csw;
    init(cert, my_priv_key, store);
}

int ClientSecureSocketWrapper::connectServer(SecureHost host)
{
    if (host.getCert() != NULL && !setOtherCert(host.getCert())){
        LOG(LOG_ERR, "Peer certificate validation failed!");
        return -1;
    }
    return csw->connectServer(host);
}

ServerSecureSocketWrapper::ServerSecureSocketWrapper(X509* cert, EVP_PKEY* my_priv_key, X509_STORE* store){
    ssw = new ServerSocketWrapper();
    sw = ssw;
    init(cert, my_priv_key, store);
}

SecureSocketWrapper *ServerSecureSocketWrapper::acceptClient()
{
    return new SecureSocketWrapper(my_cert, my_priv_key, store, ssw->acceptClient());
}

SecureSocketWrapper *ServerSecureSocketWrapper::acceptClient(X509* other_cert)
{
    SecureSocketWrapper* sec_sw = new SecureSocketWrapper(my_cert, my_priv_key, store, ssw->acceptClient());
    if (sec_sw->setOtherCert(other_cert))
        return sec_sw;
    else{
        delete sec_sw;
        return NULL;
    }
}
```

```
/**
 * @file crypto.h
 * @author Mirko Laruina
 * @brief Header for crypto algorithms
 * @date 2020-06-07
 *
 */
#ifndef CRYPTO_H
#define CRYPTO_H
#include <openssl/conf.h>
#include <openssl/evp.h>
#include <openssl/err.h>
#include <openssl/rand.h>
#include <openssl/pem.h>
#include <openssl/hmac.h>
#include <openssl/kdf.h>
#include <string.h>
#include "logging.h"

/** AES-256 GCM */
#define TAG_SIZE     16
#define IV_SIZE      12
#define KEY_SIZE     16

typedef uint32_t nonce_t;

/**
 * @brief Print OpenSSL errors
 */
#define handleErrorsNoException(level) { \
    LOG((level), "OpenSSL Exception"); \
    FILE* stream; \
    if (level < LOG_ERR) \
        stream = stdout; \
    else \
        stream = stderr; \
    ERR_print_errors_fp(stream); \
}

/**
 * @brief Print OpenSSL errors and throw exception
 */
#define handleErrors() { \
    handleErrorsNoException(LOG_ERR); \
    throw "OpenSSL Error"; \
}

/**
 * Encrypts using AES in GCM mode
 *
 * @param plaintext     buffer where the plaintext is stored
 * @param plaintext_len length of said buffer
 * @param aad           additional authenticated data buffer
 * @param aad_len       length of said buffer
 * @param key           encryption key
 * @param iv            initialization vector
 * @param ciphertext    buffer (already allocated) where the ct will be stored
 * @param tag           tag buffer
 *
 * @return number of written bytes
 */
int aes_gcm_encrypt(char *plaintext, int plaintext_len,
                    char *aad, int aad_len,
                    char *key, char *iv,
                    char *ciphertext,
                    char *tag);

/**
 * Decrypts using AES in GCM mode
 *
 * @param ciphertext        buffer where the ciphertext is stored
 * @param ciphertext_len    length of said buffer
 * @param aad               additional authenticated data buffer
 * @param aad_len           length of said buffer
 * @param key               decryption key
 * @param iv                initialization vector
 * @param plaintext         buffer (already allocated) where the pt will be stored
 * @param tag               tag buffer
 *
 * @retval -1               on error
 * @retval n                number of written bytes
 */
int aes_gcm_decrypt(char *ciphertext, int ciphertext_len,
                    char *aad, int aad_len,
                    char *key,
                    char *iv,
                    char *plaintext,
                    char *tag);

/**
 * @brief Generate a ECDH key
 *
 * Example of usage:
 * EVP_PKEY *key=NULL;
 * int ret = get_ecdh_key(&key);
 *
 * @param key   the generated key
 * @return int  ???
 */
int get_ecdh_key(EVP_PKEY **key);

/**
 * @brief Apply the DHKE to derive a shared secret
 *
 * @param my_key        first key
 * @param peer_pubkey   second key
 * @param shared_key    output buffer location (unallocated), it will contained the shared
key
 * @return int          shared_key length
 */
int dhke(EVP_PKEY *my_key, EVP_PKEY *peer_pubkey, char **shared_key);

/**
 * @brief Get a random number
 *
 * @return nonce_t  the random number
 */
nonce_t get_rand();

/**
 * @brief Fills a buffer with a random value
 *
 * @param char  buffer to fill
 * @param bytes number of bytes (buffer length)
 */
void get_rand(char* buffer, int bytes);

/**
 * @brief Load a certificate from file
 *
 * @param file_name     file name of the certificate
 * @return X509*        the certificate ptr, NULL if not read correctly
 */
X509 *load_cert_file(const char *file_name);

/**
 * @brief Load certificate revocation list from file
 *
 * @param file_name    file name
 * @return X509_CRL*   the CRL, NULL if not read correctly
 */
X509_CRL *load_crl_file(const char *file_name);

/**
 * @brief Load a key from file
 *
 * @param file_name     file name of the key file
 * @param password      key password
 * @return X509*        the key ptr, NULL if not read correctly
 */
EVP_PKEY *load_key_file(const char *file_name, const char* password);

/**
 * @brief Build a CA store from CA certificate and CRL
 *
 * @param cacert        CA certificate
 * @param crl           CRL
 * @return X509_STORE*  the store
 */
X509_STORE *build_store(X509 *cacert, X509_CRL *crl);


/**
 * @brief
 *
 * @param store     Certificate store
 * @param cert      Certificate
 * @return true     if validation is successful
 * @return false    otherwise
 */
bool verify_peer_cert(X509_STORE *store, X509 *cert);

/**
 * @brief Calculate HMAC of the msg
 *
 * @param msg       message of which we need the HMAC
 * @param msg_len   size of said message
 * @param key       key to use for the HMAC
 * @param keylen    size of said key
 * @param hmac      output buffer (uninitialized, it will be allocated)
 * @return int      size of the HMAC
 */
int hmac(char *msg, int msg_len, char *key, unsigned int keylen,
         char *hmac);

/**
 * @brief Compare two HMAC in a secure way
 *
 * @param hmac_expected     Expected HMAC
 * @param hmac_rcv          Received HMAC
 * @param len               Length of the buffers to compare
 * @return true             if they are the same
 * @return false            otherwise
 */
bool compare_hmac(char *hmac_expected, char *hmac_rcv, unsigned int len);

/**
 * @brief Apply HKDF, takes only one info field
 *
 * @param key               Key to use
 * @param key_len           Size of said key
 * @param info              Info field to use
 * @param info_len          Size of said info field
 * @param out               Output buffer (allocated)
 * @param outlen            Output len
 */
void hkdf_one_info(char *key, size_t key_len,
                  char *info, size_t info_len,
                  char *out, size_t outlen);

/**
 * @brief Apply HKDF, takes two nonces and a label field
 *
 * @param key               Key to use
 * @param key_len           Size of said key
 * @param nonce1            First nonce
 * @param nonce2            Second nonce
 * @param label             Label field
 * @param out               Output buffer (allocated)
 * @param outlen            Output len
 */
void hkdf(char *key, size_t key_len,
          nonce_t nonce1, nonce_t nonce2,
          char *label,
          char *out, size_t outlen);

/**
 * Signs the given message
 *
 * @param msg the message to be signed
 * @param msglen the length of the message to be signed
 * @param signature pointer to the output signature
 * @param prvkey the private key
 * @returns the length of the signature
 */
int dsa_sign(char* msg, int msglen, char** signature,
             EVP_PKEY *prvkey);

/**
 * Checks the given signature on the given message
 *
 * @param msg the message to be signed
 * @param msglen the length of the message to be signed
 * @param signature the signature
 * @param prvkey the public key
 * @returns true if message is authentic, false otherwise
 */
bool dsa_verify(char* msg, int msglen,
            char* signature, int sign_len,
            EVP_PKEY *pkey);

#endif
```

```cpp
#include "security/crypto.h"
#include "logging.h"

int aes_gcm_encrypt(char *plaintext, int plaintext_len,
                    char *aad, int aad_len,
                    char *key,
                    char *iv,
                    char *ciphertext,
                    char *tag)
{
    EVP_CIPHER_CTX *ctx;

    int len;

    int ciphertext_len;

    /* Create and initialise the context */
    if (!(ctx = EVP_CIPHER_CTX_new()))
        handleErrors();

    /* Initialise the encryption operation. */
    if (1 != EVP_EncryptInit(ctx, EVP_aes_128_gcm(), (unsigned char*) key, (unsigned char*) iv))
        handleErrors();

    /*
     * Provide any AAD data. This can be called zero or more times as
     * required
     */
    if (1 != EVP_EncryptUpdate(ctx, NULL, &len, (unsigned char*) aad, aad_len))
        handleErrors();

    /*
     * Provide the message to be encrypted, and obtain the encrypted output.
     * EVP_EncryptUpdate can be called multiple times if necessary
     */
    if (1 != EVP_EncryptUpdate(ctx, (unsigned char*) ciphertext, &len, (unsigned char*) plaintext, plaintext_len))
        handleErrors();
    ciphertext_len = len;

    /*
     * Finalise the encryption. Normally ciphertext bytes may be written at
     * this stage, but this does not occur in GCM mode
     */
    if (1 != EVP_EncryptFinal(ctx, (unsigned char*) ciphertext + len, &len))
        handleErrors();
    ciphertext_len += len;

    /* Get the tag */
    if (1 != EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_GET_TAG, 16, tag))
        handleErrors();

    /* Clean up */
    EVP_CIPHER_CTX_free(ctx);

    return ciphertext_len;
}

int aes_gcm_decrypt(char *ciphertext, int ciphertext_len,
                    char *aad, int aad_len,
                    char *key,
                    char *iv,
                    char *plaintext,
                    char *tag
                    )
{
    EVP_CIPHER_CTX *ctx;
    int len;
    int plaintext_len;
    int ret;

    /* Create and initialise the context */
    if (!(ctx = EVP_CIPHER_CTX_new()))
        handleErrors();

    /* Initialise the decryption operation. */
    if (!EVP_DecryptInit(ctx, EVP_aes_128_gcm(), (unsigned char*) key, (unsigned char*) iv
))
        handleErrors();

    /*
     * Provide any AAD data. This can be called zero or more times as
     * required
     */
    if (!EVP_DecryptUpdate(ctx, NULL, &len, (unsigned char*) aad, aad_len))
        handleErrors();

    /*
     * Provide the message to be decrypted, and obtain the plaintext output.
     * EVP_DecryptUpdate can be called multiple times if necessary
     */
    if (!EVP_DecryptUpdate(ctx, (unsigned char*) plaintext, &len, (unsigned char*) ciphert
ext, ciphertext_len))
        handleErrors();
    plaintext_len = len;

    /* Set expected tag value. Works in OpenSSL 1.0.1d and later */
    if (!EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_SET_TAG, 16, tag))
        handleErrors();

    /*
     * Finalise the decryption. A positive return value indicates success,
     * anything else is a failure - the plaintext is not trustworthy.
     */
    ret = EVP_DecryptFinal(ctx, (unsigned char*) plaintext + len, &len);

    /* Clean up */
    EVP_CIPHER_CTX_free(ctx);

    if (ret > 0)
    {
        /* Success */
        plaintext_len += len;
        return plaintext_len;
    }
    else
    {
        /* Verify failed */
        return -1;
    }
}

int get_ecdh_key(EVP_PKEY **key)
{
    EVP_PKEY *dh_params = NULL;
    EVP_PKEY_CTX *ctx_params;
    int ret;

    ctx_params = EVP_PKEY_CTX_new_id(EVP_PKEY_EC, NULL);
    if (!ctx_params)
    {
        handleErrors();
    }
    ret = EVP_PKEY_paramgen_init(ctx_params);
    if (!ret)
    {
        handleErrors();
    }
    //Using NID_X9_62_prime256v1 curve
    ret = EVP_PKEY_CTX_set_ec_paramgen_curve_nid(
        ctx_params,
        NID_X9_62_prime256v1);
    if (!ret)
    {
        handleErrors();
    }

    ret = EVP_PKEY_paramgen(ctx_params, &dh_params);
    if (!ret)
    {
        handleErrors();
    }
    //check
    if (dh_params == NULL)
    {
        handleErrors();
        return ret;
    }

    EVP_PKEY_CTX_free(ctx_params);

    // creating the context for key generation
    EVP_PKEY_CTX *ctx = EVP_PKEY_CTX_new(dh_params, NULL);
    if (ctx == NULL)
    {
        handleErrors();
    }
    // Generating the key
    ret = EVP_PKEY_keygen_init(ctx);
    if (!ret)
    {
        handleErrors();
    }

    ret = EVP_PKEY_keygen(ctx, key);
    if (!ret)
    {
        handleErrors();
    }

    //check
    if (key == NULL)
    {
        handleErrors();
        return ret;
    }

    EVP_PKEY_CTX_free(ctx);
    return ret;
}

/**
 * @brief Apply the DHKE to derive a shared secret
 *
 * @param my_key         first key
 * @param peer_pubkey    second key
 * @param shared_key     output buffer location (unallocated), it will contained the shared
key
 * @return int           ???
 */
int dhke(EVP_PKEY *my_key, EVP_PKEY *peer_pubkey, char **shared_key)
{
    EVP_PKEY_CTX *derivation_ctx;
    size_t shared_key_len;

    derivation_ctx = EVP_PKEY_CTX_new(my_key, NULL);
    if (derivation_ctx == NULL)
    {
        handleErrors();
    }
    if (EVP_PKEY_derive_init(derivation_ctx) <= 0)
    {
        handleErrors();
    }

    if (EVP_PKEY_derive_set_peer(derivation_ctx, peer_pubkey) <= 0)
    {
        handleErrors();
    }

    // "Dummy" derivation to extract key len
    EVP_PKEY_derive(derivation_ctx, NULL, &shared_key_len);
    *shared_key = (char *)malloc(shared_key_len);
    if (!*shared_key)
    {
        LOG_PERROR(LOG_ERR, "Malloc failed: %s");
        handleErrors();
    }

    // Real derivation
    if (EVP_PKEY_derive(derivation_ctx, (unsigned char*) *shared_key, &shared_key_len) <=
0)
    {
        handleErrors();
    }

    EVP_PKEY_CTX_free(derivation_ctx);
    return shared_key_len;
}

/**
 * @brief Get a random number
 *
 * @return nonce_t  the random number
 */
nonce_t get_rand()
{
    nonce_t random_num;
    RAND_bytes((unsigned char *)&random_num, sizeof(random_num));
    return random_num;
}

/**
 * @brief Fills a buffer with a random value
 *
 * @param char  buffer to fill
 * @param bytes number of bytes (buffer length)
 */
void get_rand(char* buffer, int bytes){
    RAND_bytes((unsigned char*) buffer, bytes);
}

/**
 * @brief Load a certificate from file
 *
 * @param file_name     file name of the certificate
 * @return X509*        the certificate ptr, NULL if not read correctly
 */
X509 *load_cert_file(const char *file_name)
{
    FILE *cert_file = fopen(file_name, "r");
    if (!cert_file)
    {
        return NULL;
    }
    X509 *cert = PEM_read_X509(cert_file, NULL, NULL, NULL);
    fclose(cert_file);
    return cert;
}

/**
 * @brief Load a key from file
 *
 * @param file_name     file name of the key file
 * @param password      key password
 * @return X509*        the key ptr, NULL if not read correctly
 */
EVP_PKEY *load_key_file(const char *file_name, const char* password)
{
    FILE *key_file = fopen(file_name, "r");
    if (!key_file)
    {
        return NULL;
    }
    EVP_PKEY *key = PEM_read_PrivateKey(key_file, NULL, NULL, (void*) password);
    fclose(key_file);
    return key;
}

/**
 * @brief Load certificate revocation list from file
 *
 * @param file_name    file name
 * @return X509_CRL*   the CRL, NULL if not read correctly
 */
X509_CRL *load_crl_file(const char *file_name)
{
    FILE *crl_file = fopen(file_name, "r");
    if (!crl_file)
    {
        return NULL;
    }
    X509_CRL *crl = PEM_read_X509_CRL(crl_file, NULL, NULL, NULL);
    fclose(crl_file);
    return crl;
}

/**
 * @brief Build a CA store from CA certificate and CRL
 *
 * @param cacert        CA certificate
 * @param crl           CRL
 * @return X509_STORE*  the store
 */
X509_STORE *build_store(X509 *cacert, X509_CRL *crl)
{
    X509_STORE *store = X509_STORE_new();
    if (!store)
    {
        handleErrors();
    }
    if (1 != X509_STORE_add_cert(store, cacert))
    {
        handleErrors();
    }
    if (1 != X509_STORE_add_crl(store, crl))
    {
        handleErrors();
    }
    if (1 != X509_STORE_set_flags(store, X509_V_FLAG_CRL_CHECK))
    {
        handleErrors();
    }

    return store;
}

/**
 * @brief
 *
 * @param store
 * @param cert
 * @return true
 * @return false
 */
bool verify_peer_cert(X509_STORE *store, X509 *cert)
{
    X509_STORE_CTX *verify_ctx = X509_STORE_CTX_new();
    if (!verify_ctx)
    {
        handleErrors();
    }

    if (1 != X509_STORE_CTX_init(verify_ctx, store, cert, NULL))
    {
        handleErrors();
    }

    int ret = X509_verify_cert(verify_ctx);
    if (ret == 1){
        return true;
    } else if (ret == 0){
        return false;
    } else {
        handleErrorsNoException(LOG_DEBUG);
        return false;
    }
}

int hmac(char *msg, int msg_len, char *key, unsigned int keylen,
         char *hmac)
{
    const EVP_MD *md = EVP_sha256();
    unsigned int hash_size = EVP_MD_size(md);

    HMAC_CTX *ctx = HMAC_CTX_new();
    HMAC_Init_ex(ctx, key, keylen, md, NULL);

    HMAC_Update(ctx, (unsigned char*) msg, sizeof(msg));
    HMAC_Final(ctx, (unsigned char*) hmac, &hash_size);

    HMAC_CTX_free(ctx);
    /*
    while ((bytes_read = fread(buffer, 1, hash_size, file)) > 0)
    {
        HMAC_Update(ctx, buffer, bytes_read);
        printf("len: %d\n", bytes_read);
    }
    */
    return hash_size;
}

bool compare_hmac(char *hmac_expected, char *hmac_rcv, unsigned int len)
{
    if (0 != CRYPTO_memcmp(hmac_expected, hmac_rcv, len))
    {
        return false;
    }
    else
    {
        return true;
    }
}
```

```

void hkdf_one_info(char *key, size_t key_len,
                   char *info, size_t info_len,
                   char *out, size_t outlen)
{
    EVP_PKEY_CTX *ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_HKDF, NULL);
    if (!ctx)
    {
        handleErrors();
    }

    if (EVP_PKEY_derive_init(ctx) <= 0)
    {
        handleErrors();
    }

    if (EVP_PKEY_CTX_set_hkdf_md(ctx, EVP_sha256()) <= 0)
    {
        handleErrors();
    }
    if (EVP_PKEY_CTX_set1_hkdf_key(ctx, key, key_len) <= 0)
    {
        handleErrors();
    }
    if (EVP_PKEY_CTX_add1_hkdf_info(ctx, (unsigned char *)info, info_len) <= 0)
    {
        handleErrors();
    }
    if (EVP_PKEY_derive(ctx, (unsigned char*) out, &outlen) <= 0)
    {
        handleErrors();
    }
    EVP_PKEY_CTX_free(ctx);
}

void hkdf(char *key, size_t key_len,
          nonce_t nonce1, nonce_t nonce2,
          char *label,
          char *out, size_t outlen)
{
    // label is a string, we remove the termination null char
    size_t info_len = sizeof(nonce_t) * 2 + strlen(label);
    char *info = (char *)malloc(info_len);
    if (!info){
        LOG_PERROR(LOG_ERR, "Malloc failed: %ss");
        throw "Malloc error";
    }
    char *info_buf = info;
    strcpy((char *)info_buf, label);
    info_buf += strlen(label);
    memcpy(info_buf, (void *)&nonce1, sizeof(nonce1));
    info_buf += sizeof(nonce1);
    memcpy(info_buf, (void *)&nonce2, sizeof(nonce2));

    hkdf_one_info(key, key_len, info, info_len, out, outlen);
    free(info);
}

int dsa_sign(char* msg, int msglen, char** signature,
             EVP_PKEY *prvkey){
            unsigned int sign_len;

    EVP_MD_CTX* ctx = EVP_MD_CTX_new();
    if (ctx == NULL){
        handleErrors();
    }

    *signature = (char*) malloc(EVP_PKEY_size(prvkey));
    if (*signature == NULL){
        LOG_PERROR(LOG_ERR, "Malloc failed: %s");
        return -1;
    }

    if (EVP_SignInit(ctx, EVP_sha256()) != 1){
        handleErrors();
    }

    if (EVP_SignUpdate(ctx, (unsigned char*) msg, msglen) != 1){
        handleErrors();
    }

    if (EVP_SignFinal(ctx, (unsigned char*) *signature, &sign_len, prvkey) != 1){
        handleErrors();
    }

    EVP_MD_CTX_free(ctx);

    return sign_len;
}

bool dsa_verify(char* msg, int msglen,
            char* signature, int sign_len,
            EVP_PKEY *pkey){
    EVP_MD_CTX* ctx = EVP_MD_CTX_new();

    if (ctx == NULL){
        handleErrors();
    }

    if (EVP_VerifyInit(ctx, EVP_sha256()) != 1){
        handleErrors();
    }

    if (EVP_VerifyUpdate(ctx, msg, msglen) != 1){
        handleErrors();
    }

    int ret = EVP_VerifyFinal(ctx, (unsigned char*) signature, sign_len, pkey);
    if(ret != 1){
        return false;
    }
    return true;
}
```

```c
/**
 * @file dump_buffer.h
 * @author Riccardo Mancini
 *
 * @brief Utility function for dumping a buffer as hex string.
 *
 * @date 2020-05-17
 */

#ifndef DUMP_BUFFER_H
#define DUMP_BUFFER_H


/**
 * Prints content of buffer to stdout, showing it as hex values.
 *
 * It uses the logging infrastructure to print.
 *
 * @param buffer    pointer to the buffer to be printed
 * @param len       the length (in bytes) of the buffer
 */
void dump_buffer_hex(char* buffer, int len, int log_level, const char* name);

#if LOG_LEVEL == LOG_DEBUG
#define DUMP_BUFFER_HEX_DEBUG(buffer, len) dump_buffer_hex(buffer, len, LOG_DEBUG, #buffer
)
#else
#define DUMP_BUFFER_HEX_DEBUG(buffer, len)
#endif


#endif // DUMP_BUFFER_H
```

```
/**
 * @file dump_buffer.cpp
 * @author Riccardo Mancini
 *
 * @brief Implementation of dump_buffer.h.
 *
 * @see dump_buffer.h
 */

#include "utils/dump_buffer.h"
#include "logging.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cctype>

#define ROW 32


void dump_buffer_hex(char* buffer, int len, int log_level, const char* name){
  char *str, tmp3[4], tmp1[2];
  int i, j;
  int n_rows = (len+ROW-1)/ROW;

  str = (char*) malloc(n_rows*(3*ROW + 4 + ROW + 1)+1);
  if (!str){
    LOG_PERROR(LOG_ERR, "Malloc failed: %s");
    return;
  }

  const char* col_sep = "    ";
  const char* row_sep = "\n";

  str[0] = '\0';
  for (i=0; i<n_rows; i++){
    for (j=0; j<ROW; j++){
      int idx = i*ROW+j;
      if (idx < len){
        sprintf(tmp3, "%02x ", (unsigned char) buffer[idx]);
      } else {
        sprintf(tmp3, "   ");
      }
      strcat(str, tmp3);
    }

    strcat(str, col_sep);

    for (j=0; j<ROW; j++){
      int idx = i*ROW+j;
      if (idx >= len)
        break;

      if (isprint(buffer[idx])){
        sprintf(tmp1, "%c", buffer[idx]);
      } else{
        sprintf(tmp1, ".");
      }
      strcat(str, tmp1);
    }
    if (i != n_rows - 1)
      strcat(str, row_sep);
  }
  LOG(log_level, "Dumping %s", name);
  if (log_level >= LOG_LEVEL)
    printf("%s%s\033[0m\n", logColor(log_level), str);
  free(str);
}
```

```
/**
 * @file host.h
 * @author Riccardo Mancini
 *
 * @brief Definition of the helper class "Host"
 *
 * @date 2020-05-17
 */

#ifndef HOST_H
#define HOST_H

#include <sys/socket.h>
#include <netinet/in.h>
#include <string>
#include "logging.h"

using namespace std;

/**
 * Class that holds a host information
 *
 * OpenSSL certificates are held in SecureHost class.
 */
class Host{
private:
    struct sockaddr_in addr;

public:
    /**
     * Constructs new empty instance
     */
    Host() {}

    /**
     * Constructs new instance from given inet address
     *
     * @param addr the inet address of the remote host
     */
    Host(struct sockaddr_in addr)
        : addr(addr) {}

    /**
     * Constructs new instance from IP/port pair
     *
     * @param ip the IP address the remote host
     * @param port the port the remote host
     */
    Host(const char* ip, int port);

    /** Returns the inet address of the host */
    struct sockaddr_in getAddress(){return addr;}

    /** Returns the inet address of the host */
    string toString();

};

#endif // HOST_H
```

```
/**
 * @file host.cpp
 * @author Riccardo Mancini
 *
 * @brief Implementation of host.h.
 *
 * @see host.h
 *
 * @date 2020-05-20
 */

#include "network/host.h"
#include "network/inet_utils.h"

Host::Host(const char* ip, int port){
    addr = make_sv_sockaddr_in(ip, port);
}
string Host::toString(){
    return sockaddr_in_to_string(addr);
}
```

```cpp
/**
 * @file user_list.h
 * @author Riccardo Mancini
 *
 * @brief Implementation of the UserList class
 *
 * @date 2020-05-23
 */

#include <sstream>
#include <string>
#include <vector>
#include <iterator>
#include <iostream>
#include "user_list.h"
#include "config.h"

using namespace std;

typedef map<string,User*>::iterator Iterator;

UserList::UserList(){
    pthread_mutex_init(&mutex, NULL);
}

bool UserList::add(User *u){
    bool success;
    pthread_mutex_lock(&mutex);
    if ((success = user_map_by_fd.size() < MAX_USERS)){
        user_map_by_fd.insert(
            pair<int,User*>(u->getSocketWrapper()->getDescriptor(), u)
        );
        if (!u->getUsername().empty()){
            user_map_by_username.insert(
                pair<string,User*>(u->getUsername(), u)
            );
        }
    }
    pthread_mutex_unlock(&mutex);

    return success;
}

User* UserList::get(string username){
    User* u = NULL;
    pthread_mutex_lock(&mutex);
    if (user_map_by_username.find(username) != user_map_by_username.end()){
        u = user_map_by_username.at(username);
        u->increaseRefs();
        LOG(LOG_DEBUG, "Thread %ld got reference to user %d",
            pthread_self(), u->getSocketWrapper()->getDescriptor()
        );
    }
    pthread_mutex_unlock(&mutex);
    return u;
}

User* UserList::get(int fd){
    User* u = NULL;
    pthread_mutex_lock(&mutex);
    if (user_map_by_fd.find(fd) != user_map_by_fd.end()){
        u = user_map_by_fd.at(fd);
        u->increaseRefs();
        LOG(LOG_DEBUG, "Thread %ld got reference to user %d",
          pthread_self(), u->getSocketWrapper()->getDescriptor()
        );
    }
    pthread_mutex_unlock(&mutex);
    return u;
```

```
}

bool UserList::exists(string username){
    bool res;
    pthread_mutex_lock(&mutex);
    res = user_map_by_username.find(username) != user_map_by_username.end();
    pthread_mutex_unlock(&mutex);
    return res;
}

bool UserList::exists(int fd){
    bool res;
    pthread_mutex_lock(&mutex);
    res = user_map_by_fd.find(fd) != user_map_by_fd.end();
    pthread_mutex_unlock(&mutex);
    return res;
}

void UserList::yield(User* u){
    pthread_mutex_lock(&mutex);
    u->decreaseRefs();
    LOG(LOG_DEBUG, "Thread %ld yielded user %d (refcount: %d)",
        pthread_self(), u->getSocketWrapper()->getDescriptor(), u->countRefs()
    );
    if (u->countRefs() == 0 && u->getState() == DISCONNECTED){
        if (user_map_by_username.find(u->getUsername())
                != user_map_by_username.end()
        ){
            user_map_by_username.erase(u->getUsername());
        }
        if (user_map_by_fd.find(u->getSocketWrapper()->getDescriptor())
                != user_map_by_fd.end()
        ){
            user_map_by_fd.erase(u->getSocketWrapper()->getDescriptor());
        }
        delete u;
    }
    pthread_mutex_unlock(&mutex);
}

string UserList::listAvailableFromTo(int from){
    ostringstream os;
    int n = 0;

    pthread_mutex_lock(&mutex);
    for (Iterator it = user_map_by_username.begin();
        n < from+MAX_USERS_IN_MESSAGE && it != user_map_by_username.end();
        ++it
    ){
        if (it->second->getState() == AVAILABLE){
            if (n < from)
                continue;

            os << it->first;
            if (n < from+MAX_USERS_IN_MESSAGE-1)
                os << ",";

            n++;
        }

    }
    pthread_mutex_unlock(&mutex);
    os << '\0';

    return os.str();
}

int UserList::size(){
    int sz;
    pthread_mutex_lock(&mutex);
    sz = user_map_by_fd.size();
    pthread_mutex_unlock(&mutex);
    return sz;
}
```

```
/**
 * @file args.h
 * @author Riccardo Mancini
 *
 * @brief Definition of the Args class
 *
 * @date 2020-05-27
 */

#ifndef ARGS_H
#define ARGS_H

#include <cstring>
#include <string>
#include <list>
#include <iostream>
#include <vector>

using namespace std;

/**
 * Utility class that parses an input line into a list of arguments (argc, argv)
 */
class Args {
private:
    int status;
    vector<string> argv;

    void parseLine(string s);
public:
    /**
     * Default constructor
     */
    Args() : status(0) {}

    /**
     * Constructor that parses the given input line
     */
    Args(char* line);

    /**
     * Constructor that reads from the given input stream.
     */
    Args(std::istream &is);

    /**
     * Operator overload for printing the arguments with cout
     *
     * Format: ["arg1", "arg2"]
     */
    friend std::ostream& operator<<(std::ostream& os, const Args& b);

    /**
     * Returns argument count.
     *
     * @returns >=0 argument count
     * @retuns -1 error reading stream (may be caused by EOF)
     */
    int getArgc(){ return status == 0 ? argv.size() : -1;}

    /**
     * Returns nth argument
     */
    const char* getArgv(unsigned int i) {
        if (status == 0 && i < argv.size())
            return argv.at(i).c_str();
        else
            return NULL;
    }

    /**
     * Returns user friendly content as a C string.
     */
    const char* c_str();
};

#endif //PARSE_ARGS_H
```

```
 /**
 * @file args.cpp
 * @author Riccardo Mancini
 *
 * @brief Implementation of the Args class
 *
 * Adapted from https://stackoverflow.com/a/14266139
 *
 * @date 2020-05-27
 */

#include "utils/args.h"
#include <sstream>

using namespace std;

void Args::parseLine(string s){
    string delimiter = " ";
    size_t pos = 0;
    string token;

    while ((pos = s.find(delimiter)) != string::npos) {
        token = s.substr(0, pos);
        argv.push_back(token);
        s.erase(0, pos + delimiter.length());
    }

    if (!s.empty()){
        argv.push_back(s);
    }
}

Args::Args(char* line){
    parseLine(string(line));
}

Args::Args(istream &is){
    string line;
    getline(is, line);
    if (!is){
        status = 1;
    } else {
        status = 0;
        parseLine(line);
    }
}

ostream& operator<<(ostream& os, const Args& a){
    os << "[";

    for (vector<string>::const_iterator it = a.argv.begin(); it != a.argv.end(); it++){
        os << *it;
        if (it != a.argv.end()-1)
            os << ",";
    }

    os << "]";
    return os;
}

const char* Args::c_str(){
    ostringstream os;
    os << *this;
    return os.str().c_str();
}
```

```
/**
 * @file single_player.h
 * @author Mirko Laruina
 * @author Riccardo Mancini
 *
 * @brief Implementation of the single player game main function
 *
 * @date 2020-05-27
 */

#include "single_player.h"
#include <iostream>
#include "utils/args.h"
#include "connect4.h"

using namespace std;

int playSinglePlayer(){
    int choosen_col, adv_col;
    int win;
    Connect4 c;

    cout<<"Who do you want to be? X or O ?"<<endl;

    do {
        cout<<"> "<<flush;
        Args args(cin);
        if (args.getArgc() <0){
            return 1;
        } else if (args.getArgc() == 1 && c.setPlayer(args.getArgv(0)[0])){
            break;
        } else{
            continue;
        }
    } while (1);

    cout<<"You are playing as "<<c.getPlayer()<<endl;

    cout<<"This is the starting board:"<<endl;
    cout<<c;

    srand(time(NULL));

    do {
        cout<<"Write the column you want to insert the token to"<<endl;
        do {
            cout<<"> "<<flush;
            Args args(cin);
            if (args.getArgc() == 1){
                choosen_col = args.getArgv(0)[0]-'0';
            } else if (args.getArgc() < 0){
                return 1;
            } else {
                choosen_col = -1;
            }
        } while(choosen_col < 0 || choosen_col > 7);

        win = c.play(choosen_col-1, c.getPlayer());
        cout<<c;
        if(win == 1){
            cout<<"Congratulation, you won!"<<endl;
        } else if(win == -1){
            cout<<"The column is full, choose a different one!"<<endl;
            continue;
        } else if(win == -2){
            cout<<"The entire board is filled: it is a draw!"<<endl;
            break;
        }

```

```cpp
        if(win != 1){
            do {
                adv_col = rand()%c.getNumCols();
                cout<<"Your enemy has chosen column "<<adv_col<<endl;
                win = c.play(adv_col, c.getAdv());
                cout<<c;
                if(win == 1){
                    cout<<"Damn! You lost!"<<endl;
                } else if(win == -1){
                    cout<<"The column is full, the adversary has to chose a different one!"<<endl;
                    continue;
                } else if(win == -2){
                    cout<<"The entire board is filled: it is a draw!"<<endl;
                    break;
                }
            } while (win == -1);
        }
    } while (win == -1 || win == 0);
    return 0;
}
```

```
/**
 * @file multi_player.h
 * @author Riccardo Mancini
 *
 * @brief Implementation of the multi player game main function and
 *          connection with peer functions
 *
 * @date 2020-05-29
 */

#include "multi_player.h"
#include "connect4.h"
#include <iostream>
#include "utils/args.h"

int playWithPlayer(int turn, SecureSocketWrapper *sw){
    int choosen_col, adv_col;
    int win;

    Connect4 c;
    cout<<"Who do you want to be? X or O ?"<<endl;

    do {
        cout<<"> "<<flush;
        Args args(cin);
        if (args.getArgc() == -1){
            return 1;
        } else if (args.getArgc() == 1 && c.setPlayer(args.getArgv(0)[0])){
            break;
        } else if (args.getArgc() < 0) {
            return 1;
        } else{
            continue;
        }
    } while (1);

    cout<<"You are playing as "<<c.getPlayer()<<endl;

    cout<<"This is the starting board:"<<endl;
    cout<<c;

    do {
        if (turn == MY_TURN){
            cout<<"Write the column you want to insert the token to"<<endl;
            do {
                cout<<"> "<<flush;
                Args args(cin);
                if (args.getArgc() == 1){
                    choosen_col = args.getArgv(0)[0]-'0';
                } else if (args.getArgc() < 0){
                    return 1;
                } else {
                    choosen_col = -1;
                }
            } while(choosen_col < 0 || choosen_col > 7);

            win = c.play(choosen_col-1, c.getPlayer());
            cout<<c;

            if (win != -1){
                MoveMessage mm(choosen_col-1);
                int ret = sw->sendMsg(&mm);
                if (ret != 0){
                    LOG(LOG_ERR, "Connection error");
                    return 1;
                }
            }

            if(win == 1){
                cout<<"Congratulation, you won!"<<endl;
            } else if(win == -1){
                cout<<"The column is full, choose a different one!"<<endl;
                continue;
            } else if(win == -2){
                cout<<"The entire board is filled: it is a draw!"<<endl;
                break;
            } else{
                turn = THEIR_TURN;
            }

        } else{     // THEIR_TURN
            do {
                MoveMessage *mm;
                mm = dynamic_cast<MoveMessage*>(sw->receiveMsg(MOVE));
                if (mm == NULL){
                    LOG(LOG_ERR, "Connection error");
                    return 1;
                }
                adv_col = mm->getColumn();
                cout<<"Your enemy has chosen column "<<adv_col<<endl;
                win = c.play(adv_col, c.getAdv());
                cout<<c;
                if(win == 1){
                    cout<<"Damn! You lost!"<<endl;
                } else if(win == -1){
                    cout<<"The column is full, the adversary has lost!"<<endl;
                    break;
                } else if(win == -2){
                    cout<<"The entire board is filled: it is a draw!"<<endl;
                    break;
                }
            } while (win == -1);
            turn = MY_TURN;
        }
    } while (win == -1 || win == 0);
    return 0;
}

SecureSocketWrapper* waitForPeer(int port, SecureHost host, X509* cert, EVP_PKEY* key, X50
9_STORE* store){
    int ret;

    ServerSecureSocketWrapper *ssw;
    ssw = new ServerSecureSocketWrapper(cert, key, store);


    ret = ssw->bindPort(port);
    if (ret != 0){
        cout<<"Could not bind to port: "<<ssw->getPort()<<endl;
        delete ssw;
        return NULL;
    }

    cout<<"Waiting for connection on port: "<<ssw->getPort()<<endl;

    SecureSocketWrapper *sw = ssw->acceptClient(host.getCert());

    if (sw == NULL){
        LOG(LOG_ERR, "Connection error: no client with valid certificate connected");
        return NULL;
    }

    ret = sw->handshakeServer();

    if (ret != 0){
        LOG(LOG_ERR, "Handshake error");
        return NULL;
    }

    Host p = sw->getConnectedHost();
    cout<<"Accepted client: "<<p.toString()<<endl;

    StartGameMessage *sgm = dynamic_cast<StartGameMessage*>(sw->receiveMsg(START_GAME_PEER
));

    if (sgm == NULL){
        LOG(LOG_ERR, "Connection error");
        return NULL;
    }

    LOG(LOG_INFO, "Connected to %s", p.toString().c_str());
    return sw;
}

SecureSocketWrapper* connectToPeer(SecureHost peer, X509* cert, EVP_PKEY* key, X509_STORE*
 store){
    cout<<"Connecting to: "<<peer.toString()<<endl;

    ClientSecureSocketWrapper *csw = new ClientSecureSocketWrapper(cert, key, store);

    int ret;
    int retry = 10;
    do {
        retry--;
        ret = csw->connectServer(peer);
        if (ret != 0){
            if (retry != 0){
                LOG(LOG_INFO, "Peer is not online yet, retrying in 1 second.");
                sleep(1);
            } else {
                break;
            }
        }
    } while(ret != 0);

    if (ret != 0){
        LOG(LOG_ERR, "Connection error");
        return NULL;
    }

    ret = csw->handshakeClient();

    if (ret != 0){
        LOG(LOG_ERR, "Handshake error");
        return NULL;
    }

    StartGameMessage m;
    ret = csw->sendMsg(&m);

    if (ret != 0){
        LOG(LOG_ERR, "Connection error");
        return NULL;
    }

    LOG(LOG_INFO, "Connected to %s", peer.toString().c_str());

    return csw;
}
```

```cpp
/**
 * @file server.h
 * @author Riccardo Mancini
 *
 * @brief Implementation of the utility class used to communicate with the server
 *
 * @date 2020-05-29
 */

#include "server.h"
#include "network/messages.h"
#include "network/inet_utils.h"
#include <iostream>

Server::~Server()
{
    if (sw != NULL)
    {
        sw->closeSocket();
        delete sw;
    }
}

int Server::getServerCert()
{
    sw->sendCertRequest();
    CertificateMessage *crm = dynamic_cast<CertificateMessage *>(sw->receiveMsg(CERTIFICATE));
    if (crm != NULL && sw->setOtherCert(crm->getCert()))
    {
        return 0;
    }
    return 1;
}

int Server::registerToServer()
{
    if (sw->connectServer(host) != 0)
    {
        connected = false;
        return 1;
    }

    if (host.getCert() == NULL)
    {
        if (getServerCert() != 0)
        {
            connected = false;
            return 1;
        }
    }

    if (sw->handshakeClient() != 0)
    {
        connected = false;
        return 1;
    }

    RegisterMessage msg(getPlayerUsername());

    int ret = sw->sendMsg(&msg);
    connected = ret == 0;
    return ret;
}

string Server::getUserList()
{
    UsersListRequestMessage req_msg;

    if (sw->sendMsg(&req_msg) != 0)
    {
        connected = false;
        return "";
    }

    UsersListMessage *res_msg;

    try
    {
        res_msg = dynamic_cast<UsersListMessage *>(sw->receiveMsg(USERS_LIST));
    }
    catch (const char *error_msg)
    {
        cerr << "Could not connect to server " << host.toString();
        cerr << " : " << error_msg << endl;
        connected = false;
        return "";
    }

    if (res_msg == NULL)
        return "";

    string usernames = res_msg->getUsernames();
    delete res_msg;

    return usernames;
}

int Server::challengePeer(string username, SecureHost *peerHost)
{
    ChallengeMessage req_msg(username);

    if (sw->sendMsg(&req_msg) != 0)
    {
        connected = false;
        return 1;
    }
    Message *res_msg;

    try
    {
        MessageType accept_types[] = {GAME_START, GAME_CANCEL};
        res_msg = sw->receiveMsg(accept_types, 2);
    }
    catch (const char *error_msg)
    {
        cerr << "Could not connect to server " << host.toString();
        cerr << " : " << error_msg << endl;
        connected = false;
        return 1;
    }

    if (res_msg == NULL)
        return 1;

    if (res_msg->getType() == GAME_CANCEL)
    {
        // Game refused
        delete res_msg;
        return -1;
    }
    else
    {
        GameStartMessage *gsm = dynamic_cast<GameStartMessage *>(res_msg);
        *peerHost = gsm->getHost();
        delete gsm;
        return 0;
    }
```

```
}

int Server::replyPeerChallenge(string username, bool response, SecureHost *peerHost, uint16_t *listen_port)
{
    // TODO handle port already busy ?
    *listen_port = rand() % (TO_PORT - FROM_PORT + 1) + FROM_PORT;

    ChallengeResponseMessage msg(username, response, *listen_port);
    if (sw->sendMsg(&msg) != 0)
    {
        connected = false;
        return 1;
    }

    if (response)
    {
        Message *res_msg;
        try
        {
            MessageType accept_types[] = {GAME_START, GAME_CANCEL};
            res_msg = sw->receiveMsg(accept_types, 2);
        }
        catch (const char *error_msg)
        {
            cerr << "Could not connect to server " << host.toString();
            cerr << " : " << error_msg << endl;
            connected = false;
            return 1;
        }

        if (res_msg == NULL)
            return 1;

        if (res_msg->getType() == GAME_CANCEL)
        {
            // Game refused
            delete res_msg;
            return -1;
        }
        else
        {
            GameStartMessage *gsm = dynamic_cast<GameStartMessage *>(res_msg);
            *peerHost = gsm->getHost();
            delete gsm;
            return 0;
        }
    }
    else
    {
        return -1;
    }
}

int Server::signalGameEnd()
{
    GameEndMessage msg;
    return sw->sendMsg(&msg);
}

void Server::disconnect()
{
    connected = false;
    sw->closeSocket();
    delete sw;
    sw = NULL;
}
```

```
/**
 * @file server_lobby.cpp
 * @author Riccardo Mancini
 *
 * @brief Implementation of the function that handles user and network input
 *          while the user is in the server lobby waiting for a game to start
 *
 * The select implementation was inspired by
 * https://www.gnu.org/software/libc/manual/html_node/Server-Example.html
 *
 * @date 2020-05-29
 */

#include <sys/select.h>
#include <stdio.h>
#include <cstring>
#include <iostream>
#include <sstream>

#include "utils/args.h"
#include "security/secure_socket_wrapper.h"
#include "security/crypto_utils.h"

#include "server_lobby.h"
#include "server.h"

using namespace std;

/** stdin has file descriptor 0 in Unix */
#define STDIN (0)

void printAvailableActions(){
    cout<<"You can list users, challenge a user, exit or simply wait for other users to challenge you."<< endl;
    cout<<"To list users type: `list`"<< endl;
    cout<<"To challenge a user type: `challenge username`"<< endl;
    cout<<"To disconnect type: `exit`"<< endl;
    cout<<"NB: you cannot receive challenges if you are challenging another user"<< endl;
}

int doAction(Args args, Server *server, SecureHost* peer_host){
    LOG(LOG_DEBUG, "Args: %s", args.c_str());
    if (args.getArgc() == 1 && strcmp(args.getArgv(0), "exit") == 0){
        return -2;
    } else if (args.getArgc() == 1 && strcmp(args.getArgv(0), "list") == 0){
        cout<<"Retrieving the list of users..."<<endl;
        string userlist = server->getUserList();
        if (userlist.empty()){
            return 1;
        }
        cout<<"Online users: "<<userlist<<endl;
        return 0;
    } else if (args.getArgc() == 2 && strcmp(args.getArgv(0), "challenge") == 0){
        cout<<"Sending challenge to "<<args.getArgv(1)<<" and waiting for response..."<<endl;
        string username(args.getArgv(1));
        int ret = server->challengePeer(username, peer_host);
        switch (ret){
            case -1: // refused
                cout<<username<<" refused your challenge"<<endl;
                return 0;
            case 0: //accepted
                cout<<username<<" accepted your challenge"<<endl;
                return -1;
            default:
                cout<<"Error connecting to server!"<<endl;
                return 1;
        }
    } else if (args.getArgc() < 0) {
        return -2; //exit
    } else {
        return 0;
    }

}

int handleReceivedChallenge(Server *server,
                            ChallengeForwardMessage* msg,
                            SecureHost* peer_host,
                            uint16_t* listen_port){
    cout<<endl<<"You received a challenge from "<<msg->getUsername()<<endl;
    cout<<"Do you want to accept? (y/n)";

    bool response;

    do{
        cout<<"> "<<flush;
        Args args(cin);
        LOG(LOG_DEBUG, "Args: %s", args.c_str());
        if (args.getArgc() == 1 && strcmp(args.getArgv(0), "y") == 0){
            response = true;
            break;
        } else if (args.getArgc() == 1 && strcmp(args.getArgv(0), "n") == 0){
            response = false;
            break;
        } else if (args.getArgc() < 0){ // EOF
            response = false;
            break;
        } else{
            continue;
        }
    } while(1);


    return server->replyPeerChallenge(msg->getUsername(), response, peer_host, listen_port
);
}

ConnectionMode handleMessage(Message* msg, Server* server){
    ChallengeForwardMessage* cfm;
    SecureHost peer_host;
    uint16_t listen_port;

    int ret;
    LOG(LOG_INFO, "Server sent message %s", msg->getName().c_str());
    switch(msg->getType()){
        case CHALLENGE_FWD:
            cfm = dynamic_cast<ChallengeForwardMessage*>(msg);
            ret = handleReceivedChallenge(server, cfm, &peer_host, &listen_port);
            switch (ret){
                case -1: // game canceled
                    cout<<"Game was canceled"<<endl;
                    return ConnectionMode(CONTINUE);
                case 0:
                    cout<<"Starting game..."<<endl;
                    return ConnectionMode(WAIT_FOR_PEER, peer_host, listen_port);
                default:
                    cout<<"Error"<<endl;
                    return ConnectionMode(EXIT, CONNECTION_ERROR);
            }
            break;
        default:
            // other messages are handled internally to
            // Server since they require the user to wait
            LOG(LOG_WARN, "Received unexpected message %s", msg->getName().c_str());
            return ConnectionMode(CONTINUE);
    }
}
```

```

ConnectionMode handleStdin(Server* server){
    SecureHost peer_host;

    int ret;
    // Input from user
    Args args(cin);
    if (args.getArgc() < 0){
        ret = -2; // received EOF
    } else{
        ret = doAction(args, server, &peer_host);
    }
    switch (ret){
        case 0: // do nothing
            LOG(LOG_DEBUG, "No action");
            return ConnectionMode(CONTINUE);
        case 1: // error
            cout<<"Error!"<<endl;
            return ConnectionMode(EXIT, CONNECTION_ERROR);
        case -1: // challenge accepted
            cout<<"Starting game..."<<endl;
            return ConnectionMode(CONNECT_TO_PEER, peer_host, 0);
        case -2:
            cout<<"Bye"<<endl;
            return ConnectionMode(EXIT, OK);
    default:
        return ConnectionMode(CONTINUE);
    }
}


ConnectionMode serverLobby(Server* server){
    fd_set active_fd_set, read_fd_set;

    string username = server->getPlayerUsername();

    if (!server->isConnected()){
        cout<<"Registering to "<<server->getHost().toString()<<" as "<<username<<endl;
        if (server->registerToServer() != 0){
            cout<<"Connection to "<<server->getHost().toString()<<" failed!"<<endl;
            return ConnectionMode(EXIT, CONNECTION_ERROR);
        }
        LOG(LOG_INFO, "Server %s is now connected",
                        server->getHost().toString().c_str());
    } else {
        LOG(LOG_INFO, "Server %s was already connected",
                        server->getHost().toString().c_str());
    }

    SecureHost peer_host;

    /* Initialize the set of active sockets. */
    FD_ZERO(&active_fd_set);
    FD_SET(server->getSocketWrapper()->getDescriptor(), &active_fd_set);
    FD_SET(STDIN, &active_fd_set);

    printAvailableActions();

    while (1){
        cout<<endl<<"> "<<flush;

        /* Block until input arrives on one or more active sockets. */
        read_fd_set = active_fd_set;
        if (select(FD_SETSIZE, &read_fd_set, NULL, NULL, NULL) < 0) {
            LOG_PERROR(LOG_ERR, "Error in select: %s");
            return ConnectionMode(EXIT, GENERIC_ERROR);
        }

        /* Service all the socketsexit(1); with input pending. */
        for (int i = 0; i < FD_SETSIZE; ++i){
            if (FD_ISSET(i, &read_fd_set)){
                if (i == server->getSocketWrapper()->getDescriptor()){
                    // Message from server.
                    Message* msg;
                    try{
                        msg = server->getSocketWrapper()->receiveAnyMsg();
                    } catch(const char* msg){
                        LOG(LOG_ERR, "Error: %s", msg);
                        return ConnectionMode(EXIT, CONNECTION_ERROR);
                    }

                    ConnectionMode m = handleMessage(msg, server);
                    if (m.connection_type != CONTINUE){
                        return m;
                    }
                } else if (i == STDIN){
                    ConnectionMode m = handleStdin(server);
                    if (m.connection_type != CONTINUE){
                        return m;
                    }
                }
            }
        }
    }
}

```

```
/**
 * @file crypto.h
 * @author Riccardo Mancini
 * @brief Header for crypto utilities
 * @date 2020-06-07
 *
 */

#ifndef CRYPTO_UTILS_H
#define CRYPTO_UTILS_H
#include <openssl/conf.h>
#include <openssl/evp.h>
#include <openssl/err.h>
#include <openssl/rand.h>
#include <openssl/pem.h>
#include <openssl/hmac.h>
#include <openssl/kdf.h>
#include <string>
#include "logging.h"
#include <map>

using namespace std;

#define KEY_BIO_MAX_SIZE 256

/**
 * Writes the key internal byte representation to the given buffer.
 *
 * @param key the key to serialize
 * @param buf the buffer
 * @param buflen the buffer length
 * @returns the number of written bytes
 */
int pkey2buf(EVP_PKEY *key, char* buf, int buflen);

/**
 * Reads the pkey from the given buffer.
 *
 * @param buf the buffer
 * @param buflen the buffer length
 * @param key the key
 * @returns 1 in case of success, <=0 otherwise
 */
int buf2pkey(char* buf, int buflen, EVP_PKEY **key);

/**
 * Writes the cert internal byte representation to the given buffer.
 *
 * @param cert the key to serialize
 * @param buf the buffer
 * @param buflen the buffer length
 * @returns the number of written bytes
 */
int cert2buf(X509 *cert, char* buf, int buflen);

/**
 * Reads the cert from the given buffer.
 *
 * @param buf the buffer
 * @param buflen the buffer length
 * @param cert the key
 * @returns 1 in case of success, <=0 otherwise
 */
int buf2cert(char* buf, int buflen, X509 **cert);


/**
 * Extracts the username (aka CN) from the given certificate
 */
```

```
string usernameFromCert(X509* cert);

/**
 * Builds a map username-certificate from the given directory
 *
 * This function matches the pattern *_cert.pem inside the directory.
 */
map<string,X509*> buildCertMapFromDirectory(string dir);

#endif // CRYPTO_UTILS_H
```

```cpp
#include "security/crypto_utils.h"
#include "security/crypto.h"
#include "dirent.h"

int pkey2buf(EVP_PKEY *key, char* buf, int buflen){
    unsigned char* i2dbuff = NULL;
    int size = i2d_PUBKEY(key, &i2dbuff);
    if(size < 0 ){
        handleErrors();
        return -1;
    }

    if(buflen < size){
        return -1;
    }

    memcpy(buf, i2dbuff, size);
    OPENSSL_free(i2dbuff);
    return size;
}


int buf2pkey(char* buf, int buflen, EVP_PKEY **key){
    const unsigned char **p = (const unsigned char**) &buf;
    if (d2i_PUBKEY(key, p, buflen) == NULL){
        handleErrors();
        return -1;
    }
    return 1;
}

int cert2buf(X509 *cert, char* buf, int buflen){
    unsigned char* i2dbuff = NULL;
    int size = i2d_X509(cert, &i2dbuff);
    if(size < 0 ){
        handleErrors();
        return -1;
    }

    if(buflen < size){
        return -1;
    }

    memcpy(buf, i2dbuff, size);
    OPENSSL_free(i2dbuff);
    return size;
}

int buf2cert(char* buf, int buflen, X509 **cert){
    const unsigned char **p = (const unsigned char**) &buf;
    if (d2i_X509(cert, p, buflen) == NULL){
        handleErrors();
        return -1;
    }
    return 1;
}

string usernameFromCert(X509* cert){
    string username;
    X509_NAME* subj_name = X509_get_subject_name(cert);
    char* subj_name_cstr = X509_NAME_oneline(subj_name, NULL, 0);

    string subj_name_str(subj_name_cstr);
    size_t pos = subj_name_str.find("/CN=");
    if (pos != string::npos){
        username = subj_name_str.substr(pos+4);
        LOG(LOG_DEBUG, "%s has size %ld", username.c_str(), username.size());
    } else{
        LOG(LOG_WARN, "Common name not found in cert: %s", subj_name_str.c_str());
        username = subj_name_str;
    }

    free(subj_name_cstr);
    return username;
}

map<string,X509*> buildCertMapFromDirectory(string dir_name){
    const char* PATTERN = "_cert.pem";
    char path[1024]; //should be always big enough
    map<string,X509*> cert_map;
    DIR *dir;
    struct dirent *ent;
    if((dir = opendir(dir_name.c_str())) != NULL) {
        while((ent = readdir (dir)) != NULL) {
            LOG(LOG_DEBUG, "%s", ent->d_name);
            if (strstr(ent->d_name, PATTERN) != NULL){
                LOG(LOG_DEBUG, "Match");
                snprintf(path, 1024, "%s/%s", dir_name.c_str(), ent->d_name);
                X509* cert = load_cert_file(path);
                string username = usernameFromCert(cert);
                cert_map.insert(pair<string,X509*>(username, cert));
            }
        }
        closedir(dir);
    } else {
        LOG(LOG_ERR, "Could not open certificate directory");
    }
    return cert_map;
}
```

```
/**
 * @file dump_buffer.h
 * @author Riccardo Mancini
 *
 * @brief Utility functions for writing and reading data from a buffer
 *
 * These functions are buffer-overflow-safe, i.e. they check the remaining
 * buffer length before writing/reading. The return is -1 in case of errors,
 * the written/read size otherwise.
 *
 * @date 2020-06-16
 */

#ifndef BUFFER_IO_H
#define BUFFER_IO_H

#include <cstdlib>
#include <stdint.h>

using namespace std;

/**
 * Reads a boolean.
 *
 * @param val the dest value
 * @param buf the source buffer
 * @param buf_size the size of the buffer
 * @returns -1 in case of errors
 * @returns 1 number of read bytes
 */
int readBool(bool *val, char* buf, size_t buf_size);

/**
 * Writes a boolean.
 *
 * @param buf the dest buffer
 * @param buf_size the size of the buffer
 * @param val the source value
 * @returns -1 in case of errors
 * @returns 1 number of read bytes
 */
int writeBool(char* buf, size_t buf_size, bool val);

/**
 * Reads a uint32_t.
 *
 * @param val the dest value
 * @param buf the source buffer
 * @param buf_size the size of the buffer
 * @returns -1 in case of errors
 * @returns 4 number of read bytes
 */
int readUInt32(uint32_t *val, char* buf, size_t buf_size);

/**
 * Writes a uint32_t.
 *
 * @param buf the dest buffer
 * @param buf_size the size of the buffer
 * @param val the source value
 * @returns -1 in case of errors
 * @returns 4 number of read bytes
 */
int writeUInt32(char* buf, size_t buf_size, uint32_t val);

/**
 * Reads a uint16_t.
 *
 * @param val the dest value
 * @param buf the source buffer
 * @param buf_size the size of the buffer
 * @returns -1 in case of errors
 * @returns 2 number of read bytes
 */
int readUInt16(uint16_t *val, char* buf, size_t buf_size);

/**
 * Writes a uint16_t.
 *
 * @param buf the dest buffer
 * @param buf_size the size of the buffer
 * @param val the source value
 * @returns -1 in case of errors
 * @returns 2 number of read bytes
 */
int writeUInt16(char* buf, size_t buf_size, uint16_t val);

/**
 * Reads a uint8_t.
 *
 * @param val the dest value
 * @param buf the source buffer
 * @param buf_size the size of the buffer
 * @returns -1 in case of errors
 * @returns 1 number of read bytes
 */
int readUInt8(uint8_t *val, char* buf, size_t buf_size);

/**
 * Writes a uint8_t.
 *
 * @param buf the dest buffer
 * @param buf_size the size of the buffer
 * @param val the source value
 * @returns -1 in case of errors
 * @returns 1 number of read bytes
 */
int writeUInt8(char* buf, size_t buf_size, uint8_t val);


/**
 * Reads a char array.
 *
 * @param val the dest buffer
 * @param len number of bytes to read
 * @param buf the source buffer
 * @param buf_size the size of the buffer
 * @returns -1 in case of errors
 * @returns len number of read bytes
 */
int readBuf(char *val, size_t len, char* buf, size_t buf_size);

/**
 * Writes a char array.
 *
 * @param buf the dest buffer
 * @param buf_size the size of the buffer
 * @param val the source buffer
 * @param len number of bytes to write
 * @returns -1 in case of errors
 * @returns len number of read bytes
 */
int writeBuf(char* buf, size_t buf_size, char* val, size_t len);

#endif // BUFFER_IO_H
```

```cpp
/**
 * @file dump_buffer.h
 * @author Riccardo Mancini
 *
 * @brief Utility functions for writing and reading data from a buffer
 *
 * @date 2020-06-16
 */

#ifndef BUFFER_IO_H
#define BUFFER_IO_H

#include <cstring>
#include <stdint.h>
#include "utils/buffer_io.h"
#include "network/inet_utils.h"

using namespace std;

int readBool(bool *val, char* buf, size_t buf_size){
    if (buf_size < sizeof(bool))
        return -1;

    *val = (bool) buf[0];
    return sizeof(bool);
}

int writeBool(char* buf, size_t buf_size, bool val){
    if (buf_size < sizeof(bool))
        return -1;

    buf[0] = (char) val;
    return sizeof(bool);
}

int readUInt32(uint32_t *val, char* buf, size_t buf_size){
    if (buf_size < sizeof(uint32_t))
        return -1;

    *val = ntohl(*((uint32_t*) buf));
    return sizeof(uint32_t);
}

int writeUInt32(char* buf, size_t buf_size, uint32_t val){
    if (buf_size < sizeof(uint32_t))
        return -1;

    *((uint32_t*)buf) = htonl(val);
    return sizeof(uint32_t);
}

int readUInt16(uint16_t *val, char* buf, size_t buf_size){
    if (buf_size < sizeof(uint16_t))
        return -1;

    *val = ntohs(*((uint16_t*) buf));
    return sizeof(uint16_t);
}

int writeUInt16(char* buf, size_t buf_size, uint16_t val){
    if (buf_size < sizeof(uint16_t))
        return -1;

    *((uint16_t*)buf) = htons(val);
    return sizeof(uint16_t);
}

int readUInt8(uint8_t *val, char* buf, size_t buf_size){
    if (buf_size < sizeof(uint8_t))
```

```
        return -1;

    *val = (uint8_t) buf[0];
    return sizeof(uint8_t);
}

int writeUInt8(char* buf, size_t buf_size, uint8_t val){
    if (buf_size < sizeof(uint8_t))
        return -1;

    buf[0] = val;
    return sizeof(uint8_t);
}

int readBuf(char *val, size_t len, char* buf, size_t buf_size){
    if (buf_size < len)
        return -1;

    memcpy(val, buf, len);
    return len;
}

int writeBuf(char* buf, size_t buf_size, char* val, size_t len){
    if (buf_size < len)
        return -1;

    memcpy(buf, val, len);
    return len;
}

#endif // BUFFER_IO_H
```

```
/**
 * @file client.cpp
 * @author Mirko Laruina
 *
 * @brief Implementation of a 4-in-a-row game
 *
 * @date 2020-05-14
 */
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "connect4.h"
#include "logging.h"
#include "network/socket_wrapper.h"
#include "network/host.h"
#include "utils/args.h"
#include "single_player.h"
#include "multi_player.h"
#include "connection_mode.h"
#include "server_lobby.h"
#include "security/crypto.h"
#include "server.h"

using namespace std;

static const char players[] = {'X', 'O'};

/**
 * Prints command usage information.
 */
void print_help(char* argv0){
  cout<<"Usage: "<<argv0<<" cert.pem key.pem cacert.pem crl.pem [other_cert.pem]"<<endl;
}

void printWelcome(){
    cout<<"****************************************************************\n"
        <<"*        __ __     _                                           *\n"
        <<"*       / // /    (_)___     ____ _   _________ _      __      *\n"
        <<"*      / // /_   / / __ \\   / __ `/  / ___/ __ \\ | /| / /      *\n"
        <<"*     /__  __/  / / / / /  / /_/ /  / /  / /_/ / |/ |/ /       *\n"
        <<"*       /_/    /_/_/ /_/   \\__,_/  /_/   \\____/|__/|__/        *\n"
        <<"*                                                              *\n"
        <<"****************************************************************"
        <<endl;
}

struct ConnectionMode promptChooseConnection(){
    cout<<"You can connect to a server, wait for a peer or connect to a peer"<< endl;
    cout<<"To connect to a server type: `server host port [path/to/server_cert.pem]`"<< endl;
    cout<<"To connect to a peer type: `peer host port path/to/peer_cert.pem`"<< endl;
    cout<<"To wait for a peer type: `peer listen_port path/to/peer_cert.pem`"<< endl;
    cout<<"To play offline type: `offline`"<< endl;
    cout<<"To exit type: `exit`"<< endl;

    do {
        cout<<"> "<<flush;
        Args args(cin);
        if (args.getArgc() == 3 && strcmp(args.getArgv(0), "peer") == 0){
            X509* cert = load_cert_file(args.getArgv(2));
            char dummy_ip[] = "127.0.0.1";
            return ConnectionMode(WAIT_FOR_PEER, dummy_ip,
                                0, cert, atoi(args.getArgv(1)));

        } else if (args.getArgc() == 4 && strcmp(args.getArgv(0), "peer") == 0){
            X509* cert = load_cert_file(args.getArgv(3));
            return ConnectionMode(CONNECT_TO_PEER, args.getArgv(1),
                                        atoi(args.getArgv(2)), cert, 0);

        } else if (args.getArgc() >= 3 && strcmp(args.getArgv(0), "server") == 0){
            X509* cert;
            if(args.getArgc() == 4)
                cert = load_cert_file(args.getArgv(3));
            else
                cert = NULL;
            return ConnectionMode(CONNECT_TO_SERVER, args.getArgv(1),
                                        atoi(args.getArgv(2)), cert, 0);

        } else if (args.getArgc() == 1 && strcmp(args.getArgv(0), "offline") == 0){
            return ConnectionMode(SINGLE_PLAYER);

        } else if (args.getArgc() == 1 && strcmp(args.getArgv(0), "exit") == 0){
            cout << "Bye" << endl;
            return ConnectionMode(EXIT, OK);
        } else if (args.getArgc() == 0){
            return ConnectionMode(CONTINUE);
        } else if (args.getArgc() == -1){ // EOF
            cout << "Bye" << endl;
            return ConnectionMode(EXIT, OK);
        }else{
            cout << "Could not parse arguments: "<< args << endl;
        }
    } while (true);
}


int main(int argc, char** argv){
    SecureSocketWrapper *sw = NULL;
    Server* server = NULL;

    if (argc < 5){
        print_help(argv[0]);
        return 1;
    }

    X509* cert = load_cert_file(argv[1]);
    X509* cacert = load_cert_file(argv[3]);
    X509_CRL* crl = load_crl_file(argv[4]);
    X509_STORE* store = build_store(cacert, crl);

    srand(time(NULL));

    int ret;

    printWelcome();
    cout<<endl<<"Welcome to 4-in-a-row!"<<endl;
    cout<<"The rules of the game are simple: you win when you have 4 connected tokens alon
g any direction."<<endl;

    EVP_PKEY* key;
    do {
        key = load_key_file(argv[2], NULL);
        if( key == NULL ){
            cout<<"Wrong password"<<endl;
        }
    } while(!key);
    cout<<endl;

    do{
        struct ConnectionMode ucc = promptChooseConnection();

        if (ucc.connection_type == EXIT && ucc.exit_code == OK){
            exit(0); // Bye
        }

        if (ucc.connection_type == CONNECT_TO_SERVER){
            server = new Server(ucc.host, cert, key, store);
        }

        bool loopLobby = true;

        do{
            try{
                if (server != NULL){
                    ucc = serverLobby(server);
                }

                switch(ucc.connection_type){
                    case WAIT_FOR_PEER:
                        sw = waitForPeer(ucc.listen_port, ucc.host, cert, key, store);
                        if (sw != NULL)
                            ret = playWithPlayer(MY_TURN, sw);
                        else
                            ret = CONNECTION_ERROR;

                        loopLobby = true;
                        break;
                    case CONNECT_TO_PEER:
                        sw = connectToPeer(ucc.host, cert, key, store);
                        if (sw != NULL)
                            ret = playWithPlayer(THEIR_TURN, sw);
                        else
                            ret = CONNECTION_ERROR;

                        loopLobby = true;
                        break;
                    case SINGLE_PLAYER:
                        ret = playSinglePlayer();
                        loopLobby = false;
                        break;
                    case EXIT:
                        ret = ucc.exit_code;
                        loopLobby = false;
                        break;
                    case CONNECT_TO_SERVER:
                        ret = FATAL_ERROR;
                        loopLobby = false;
                        break;
                    case CONTINUE:
                        ret = OK;
                        loopLobby = true;
                        break;
                }

                if (loopLobby && ret == OK &&
                    server != NULL && server->isConnected()
                ){
                    server->signalGameEnd();
                }
            } catch(const char* error_msg){
                LOG(LOG_ERR, "Caught error: %s", error_msg);
                ret = GENERIC_ERROR;
                loopLobby = false;
            }

        } while (loopLobby && server != NULL && server->isConnected());
        if (server != NULL){
            delete server;
            server = NULL;
        }
    } while(ret != FATAL_ERROR);

    return ret;
}
```

```
/**
 * @file server.cpp
 * @author Riccardo Mancini
 *
 * @brief Implementation of a 4-in-a-row online server
 *
 * The select implementation was inspired by
 * https://www.gnu.org/software/libc/manual/html_node/Server-Example.html
 *
 * @date 2020-05-23
 */
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <pthread.h>
#include <map>
#include <queue>
#include <utility>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#include "logging.h"
#include "config.h"
#include "network/socket_wrapper.h"
#include "network/host.h"

#include "user.h"
#include "user_list.h"
#include "utils/message_queue.h"

#include "security/crypto_utils.h"

using namespace std;

typedef pair<int,Message*> msgqueue_t;
typedef map<string,X509*> cert_map_t;

static UserList user_list;
static MessageQueue<msgqueue_t,MAX_QUEUE_LENGTH> message_queue;
static pthread_t threads[N_THREADS];
static cert_map_t cert_map;
static X509* cert;

void logUnexpectedMessage(User* u, Message* m){
    LOG(LOG_WARN, "User %s (state %d) was not expecting a message of type %d",
        u->getUsername().c_str(), (int)u->getState(), (int)m->getType());
}

void doubleLock(User* u_with_lock, User* u_without_lock){
    // prevent deadlocks
    if (u_without_lock->getUsername() < u_with_lock->getUsername()){
        u_with_lock->unlock();
        u_without_lock->lock();
        u_with_lock->lock();
    } else{
        u_without_lock->lock();
    }
}

void doubleUnlock(User* u_keep_lock, User* u_unlock){
    // prevent deadlocks
    if (u_unlock->getUsername() < u_keep_lock->getUsername()){
        u_keep_lock->unlock();
        u_unlock->unlock();
        u_keep_lock->lock();
    } else{
        u_unlock->unlock();
    }
}

bool handleRegisterMessage(User* u, RegisterMessage* msg){
    string username = msg->getUsername();
    string usernameCert = usernameFromCert(u->getSocketWrapper()->getCert());
    if (username.compare(usernameCert) == 0){
        LOG(LOG_WARN, "Malicious operation: %s tried to register as %s",
                usernameCert.c_str(), username.c_str());
        return false;
    }
    u->setUsername(username);

    if (!user_list.exists(username)){
        u->setState(AVAILABLE);
        // readd with username
        user_list.add(u);
        return true;
    } else {
        LOG(LOG_WARN, "User %s already registered!", username.c_str());
        //TODO send error
        u->setState(DISCONNECTED);
        return false;
    }
}

bool handleChallengeMessage(User* u, ChallengeMessage* msg){
    bool res;
    string chlg_username = msg->getUsername();
    User* challenged = user_list.get(chlg_username);
    if (challenged == NULL || challenged == u){
        GameCancelMessage cancel_msg(chlg_username);
        return u->getSocketWrapper()->sendMsg(&cancel_msg) == 0;
    }

    doubleLock(u, challenged);

    if (u->getState() != AVAILABLE){
        // someother thing concurrently happened, ignore
        doubleUnlock(u, challenged);
        user_list.yield(challenged);
        return true;
    }

    if (challenged->getState() != AVAILABLE){
        // someother thing concurrently happened, abort
        GameCancelMessage cancel_msg(chlg_username);
        res = u->getSocketWrapper()->sendMsg(&cancel_msg) == 0;

        doubleUnlock(u, challenged);
        user_list.yield(challenged);
        return res;
    }

    // both are available, send challenge
    ChallengeForwardMessage fwd_msg(u->getUsername());
    if (challenged->getSocketWrapper()->sendMsg(&fwd_msg) == 0){
        // challenge sent, mark them as playing until I receive a response
        u->setState(CHALLENGED);
        u->setOpponent(challenged->getUsername());
        challenged->setState(CHALLENGED);
        challenged->setOpponent(u->getUsername());
        res = true;
    } else{
        // connection error -> assume disconnected and notify u
        GameCancelMessage cancel_msg(chlg_username);
        challenged->setState(DISCONNECTED);
        res = u->getSocketWrapper()->sendMsg(&cancel_msg) == 0;
    }

    doubleUnlock(u, challenged);
    user_list.yield(challenged);

    return res;
}

bool handleGameEndMessage(User* u, GameEndMessage* msg){
    u->setState(AVAILABLE);
    return true;
}

bool handleUsersListRequestMessage(User* u, UsersListRequestMessage* msg){
    UsersListMessage ul_msg(user_list.listAvailableFromTo(msg->getOffset()));
    return u->getSocketWrapper()->sendMsg(&ul_msg) == 0;
}

bool handleChallengeResponseMessage(User* u, ChallengeResponseMessage* msg){
    bool res;
    User *opponent = user_list.get(u->getOpponent());
    if (opponent == NULL || opponent == u){
        // opponent disconnected or invalid opponent -> cancel
        u->setState(AVAILABLE);
        GameCancelMessage cancel_msg(u->getOpponent());
        return u->getSocketWrapper()->sendMsg(&cancel_msg) == 0;
    }

    doubleLock(u, opponent);

    if (msg->getResponse()){ // accepted
        if (u->getState() != CHALLENGED){
            // someother thing concurrently happened, abort
            // maybe this refers to old challenge
            doubleUnlock(u, opponent);
            user_list.yield(opponent);
            return true;
        }

        if (opponent->getState() != CHALLENGED){
            // opponent is in wrong state...
            // maybe this refers to old challenge
            // notify u of opponent not ready
            GameCancelMessage cancel_msg(opponent->getUsername());
            res = u->getSocketWrapper()->sendMsg(&cancel_msg) == 0;
            doubleUnlock(u, opponent);
            user_list.yield(opponent);
            return res;
        }

        struct sockaddr_in opp_addr = opponent->getSocketWrapper()
                                        ->getConnectedHost().getAddress();
        opp_addr.sin_port = 0;
        cert_map_t::iterator opp_pair = cert_map.find(opponent->getUsername());
        if(opp_pair == cert_map.end()) {
            doubleUnlock(u, opponent);
            user_list.yield(opponent);
            return false;
        }
        GameStartMessage msg_to_u(opponent->getUsername(), opp_addr, opp_pair->second);

        struct sockaddr_in u_addr = u->getSocketWrapper()
                                        ->getConnectedHost().getAddress();
        u_addr.sin_port = htons(msg->getListenPort());
        cert_map_t::iterator u_pair = cert_map.find(u->getUsername());
        if(u_pair == cert_map.end()) {
            doubleUnlock(u, opponent);
            user_list.yield(opponent);
            return false;
        }
        GameStartMessage msg_to_opp(u->getUsername(), u_addr, u_pair->second);

        int res_u = u->getSocketWrapper()->sendMsg(&msg_to_u);
        int res_opp = opponent->getSocketWrapper()->sendMsg(&msg_to_opp);

        if (res_u == 0 && res_opp == 0){
            //success
            u->setState(PLAYING);
            opponent->setState(PLAYING);

            res = true;
        } else if (res_u != 0 && res_opp != 0){
            // both disconnected
            opponent->setState(DISCONNECTED);
            u->setState(DISCONNECTED);
            res = false;
        } else {
            if (res_u != 0){ // just u disconnected => notify opp
                GameCancelMessage cancel_msg(u->getUsername());
                if (opponent->getSocketWrapper()->sendMsg(&cancel_msg) == 0){
                    opponent->setState(AVAILABLE);
                } else {
                    opponent->setState(DISCONNECTED);
                }
            } else if (res_opp != 0){ // just opp disconnected => notify u
                GameCancelMessage cancel_msg(opponent->getUsername());
                if (u->getSocketWrapper()->sendMsg(&cancel_msg) == 0){
                    u->setState(AVAILABLE);
                    res = true;
                } else {
                    u->setState(DISCONNECTED);
                    res = false;
                }
            }
        }
    } else{ // rejected
        u->setState(AVAILABLE);
        GameCancelMessage cancel_msg(u->getUsername());
        if (opponent->getSocketWrapper()->sendMsg(&cancel_msg) == 0){
            opponent->setState(AVAILABLE);
        } else{
            opponent->setState(DISCONNECTED);
        }
    }

    doubleUnlock(u, opponent);
    user_list.yield(opponent);

    return res;
}

bool handleClientHelloMessage(User* u, ClientHelloMessage* chm){
    string username = chm->getMyId();
    cert_map_t::iterator res;
    SecureSocketWrapper *sw = u->getSocketWrapper();

    if ((res = cert_map.find(username)) != cert_map.end()){
        sw->setOtherCert(res->second);
        int ret = u->getSocketWrapper()->handleClientHello(chm);
        return ret == 0;
    } else{
        LOG(LOG_WARN, "User %s not found in cert_map", username.c_str());
        return false;
    }
}
```

```

bool handleClientVerifyMessage(User* u, ClientVerifyMessage* cvm){
    int ret = u->getSocketWrapper()->handleClientVerify(cvm);
    if(ret == 0){
        u->setState(SECURELY_CONNECTED);
        return true;
    } else {
        LOG(LOG_ERR, "Client Verify failed!");
        u->setState(DISCONNECTED);
        return false;
    }
}

bool handleCertificateRequestMessage(User* u, CertificateRequestMessage* crm){
    CertificateMessage cm(cert);
    int ret = u->getSocketWrapper()->sendPlain(&cm);
    if(ret == 0){
        return true;
    } else {
        LOG(LOG_ERR, "Error sending certificate to client! Error %d", ret);
        u->setState(DISCONNECTED);
        return false;
    }
}

bool handleMessage(User* user, Message* raw_msg){
    bool res = true;

    user->lock();

    try{

        Message* msg = user->getSocketWrapper()->handleMsg(raw_msg);

        if (msg == NULL)
            return false;

        LOG(LOG_INFO, "User %s (state %d) received a message of type %s",
            user->getUsername().c_str(), (int) user->getState(), msg->getName().c_str());

        switch(user->getState()){
            case JUST_CONNECTED:
                switch(msg->getType()){
                    case CLIENT_HELLO:
                        res = handleClientHelloMessage(user,
                            dynamic_cast<ClientHelloMessage*>(msg));
                        break;
                    case CLIENT_VERIFY:
                        res = handleClientVerifyMessage(user,
                            dynamic_cast<ClientVerifyMessage*>(msg));
                        break;
                    case CERT_REQ:
                        res = handleCertificateRequestMessage(user,
                            dynamic_cast<CertificateRequestMessage*>(msg));
                    // TODO: handle cert request
                    default:
                        logUnexpectedMessage(user, msg);
                }
                break;
            case SECURELY_CONNECTED:
                switch(msg->getType()){
                    case REGISTER:
                        res = handleRegisterMessage(user,
                            dynamic_cast<RegisterMessage*>(msg));
                        break;
                    default:
                        logUnexpectedMessage(user, msg);
                }
                break;
            case AVAILABLE:
                switch(msg->getType()){
                    case CHALLENGE:
                        res = handleChallengeMessage(user,
                            dynamic_cast<ChallengeMessage*>(msg));
                        break;
                    case USERS_LIST_REQ:
                        res = handleUsersListRequestMessage(user,
                            dynamic_cast<UsersListRequestMessage*>(msg));
                        break;
                    default:
                        logUnexpectedMessage(user, msg);
                }
                break;
            case CHALLENGED:
                switch(msg->getType()){
                    case CHALLENGE_RESP:
                        res = handleChallengeResponseMessage(user,
                            dynamic_cast<ChallengeResponseMessage*>(msg));
                        break;
                    default:
                        logUnexpectedMessage(user, msg);
                }
                break;
            case PLAYING:
                switch(msg->getType()){
                    case GAME_END:
                        res = handleGameEndMessage(user,
                            dynamic_cast<GameEndMessage*>(msg));
                        break;
                    default:
                        logUnexpectedMessage(user, msg);
                }
                break;
            default:
                LOG(LOG_ERR, "User %s is in unrecognized state %d",
                    user->getUsername().c_str(), (int) user->getState());
        }

        delete msg;
    } catch(const char* error_msg){
        LOG(LOG_ERR, "Caught error: %s", error_msg);
        res = false;
    }

    user->unlock();
    return res;
}

void* worker(void *args){
    while (1){
        msgqueue_t p = message_queue.pullWait();
        User* u = user_list.get(p.first);
        if (u != NULL){
            if (!handleMessage(u, p.second)){
                // Connection error -> assume disconnected
                u->setState(DISCONNECTED);
            }
            user_list.yield(u);
        }
    }

}

void init_threads(){
    for (int i=0; i < N_THREADS; i++){
        pthread_create(&threads[i], NULL, worker, NULL);
    }
}
```

```

bool checkCertsInCertMap(X509_STORE* store, cert_map_t cert_map){
    for (cert_map_t::iterator it = cert_map.begin();
        it != cert_map.end();
        ++it
    ){
        if (!verify_peer_cert(store, it->second)){
            LOG(LOG_ERR, "Validation failed for certificate in directory: %s",
                    it->first.c_str());
            return false;
        }
    }
    return true;

}

int main(int argc, char** argv){
    fd_set active_fd_set, read_fd_set;

    if (argc < 7){
        cout<<"Usage: "<<argv[0]<<" port cert.pem key.pem cacert.pem crl.pem"<<endl;
        exit(1);
    }

    int port = atoi(argv[1]);
    cert = load_cert_file(argv[2]);
    EVP_PKEY* key = load_key_file(argv[3], NULL);
    X509* cacert = load_cert_file(argv[4]);
    X509_CRL* crl = load_crl_file(argv[5]);
    X509_STORE* store = build_store(cacert, crl);
    cert_map = buildCertMapFromDirectory(argv[6]);

    if (cert_map.size() == 0){
        LOG(LOG_ERR, "No certificates found in directory");
        return 1;
    }

    if (!checkCertsInCertMap(store, cert_map)){
        return 1;
    }

    LOG(LOG_INFO, "Loaded certificates from %s", argv[6]);

    ServerSecureSocketWrapper server_sw(cert, key, store);

    int ret = server_sw.bindPort(port);
    if (ret != 0){
        LOG(LOG_FATAL, "Error binding to port %d", port);
        exit(1);
    }

    LOG(LOG_INFO, "Binded to port %d", port);

    init_threads();

    LOG(LOG_INFO, "Started %d worker threads", N_THREADS);

    /* Initialize the set of active sockets. */
    FD_ZERO(&active_fd_set);
    FD_SET(server_sw.getDescriptor(), &active_fd_set);

    LOG(LOG_INFO, "Polling open sockets");

    while (1){
        /* Block until input arrives on one or more active sockets. */
        read_fd_set = active_fd_set;
        if (select(FD_SETSIZE, &read_fd_set, NULL, NULL, NULL) < 0) {
            if (errno == EBADF){ // clean closed sockets
                LOG(LOG_DEBUG, "Bad file descriptor");
                for (int i = 0; i < FD_SETSIZE; ++i){
                    if (FD_ISSET(i, &active_fd_set)){
                        if (i != server_sw.getDescriptor()
                                && !user_list.exists(i)
                            ){
                            // user was disconnected but I still need to clear it
                            LOG(LOG_DEBUG, "Cleared fd %d", i);
                            FD_CLR(i, &active_fd_set);
                        }
                    }
                }
                continue;
            }

            LOG_PERROR(LOG_FATAL, "Error in select: %s");
            exit(1);
        }

        /* Service all the sockets with input pending. */
        for (int i = 0; i < FD_SETSIZE; ++i){
            if (FD_ISSET(i, &read_fd_set)){
                if (i == server_sw.getDescriptor()) {
                    /* Connection request on original socket. */
                    SecureSocketWrapper* sw = server_sw.acceptClient();

                    LOG(LOG_INFO, "New connection from %s",
                        sw->getConnectedHost().toString().c_str());

                    FD_SET(sw->getDescriptor(), &active_fd_set);

                    User *u = new User(sw);
                    user_list.add(u);
                } else {
                    User *u = user_list.get(i);
                    if (u->getState() == DISCONNECTED){
                        LOG(LOG_DEBUG, "Received message from disconnected user with count
Refs = %d", u->countRefs());
                        user_list.yield(u);
                        FD_CLR(i, &active_fd_set); // ignore him
                        continue;
                    }
                    const char* u_addr_str = u->getSocketWrapper()
                            ->getConnectedHost().toString().c_str();
                    LOG(LOG_INFO, "Available message from %s (%s)",
                        u->getUsername().c_str(), u_addr_str);
                    try{
                        Message* m = u->getSocketWrapper()->readPartMsg();
                        if (m != NULL)
                            message_queue.pushSignal(msgqueue_t(i, m));
                    } catch(const char* msg){
                        LOG(LOG_WARN, "Client %s disconnected: %s",
                            u_addr_str, msg);
                        u->setState(DISCONNECTED);
                    }
                    user_list.yield(u);
                    if (!user_list.exists(i)){
                        // user was disconnected -> clear it
                        FD_CLR(i, &active_fd_set);
                    }
                }
            } else if (FD_ISSET(i, &active_fd_set)){
                if (i != server_sw.getDescriptor()
                            && !user_list.exists(i)
                ){
                    LOG(LOG_DEBUG, "Cleared fd %d", i);
                    // user was disconnected but I still need to clear it
                    FD_CLR(i, &active_fd_set);
                }
            }
        }
    }
}


```