

```
1: # Compiler and flags
2: CC      = g++
3: CFLAGS   = -g -Wall -lcrypto
4:
5: # Directories
6: OBJDIR    = build
7: SRCDIR    = src
8: BINDIR    = dist
9: HDRDIR    = include
10: DOCDIR    = doc
11: DOCTMPDIR = build/doc
12: FOLDERS   := $(strip $(shell find $(SRCDIR) -type d -printf '%P\n'))
13:
14: # List of targets
15: 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
16: TARGETS   = client/client server/server
17:
18: SRCS = $(addsuffix .cpp, $(addprefix $(SRCDIR)/,$(UTILS))) $(addsuffix .cpp, $(addprefix $(SRCDIR)/,$(TARGETS)))
19:
20: # Documentation output
21: DOCPDFNAME = documentation.pdf
22: SRCPDFNAME = source_code.pdf
23:
24: # Documentation config file
25: DOXYGENCFG = doxygen.cfg
26:
27: override CFLAGS += -I $(HDRDIR)
28: override LDFLAGS += -lpthread
29:
30: # Object files for utilities (aka libraries)
31: UTILS_OBJ = $(addsuffix .o, $(addprefix $(OBJDIR)/,$(UTILS)))
32:
33: # Builds only the executables: default rule
34: exe: $(addprefix $(BINDIR)/,$(TARGETS))
35:
36: # Utilities are secondary targets
37: .SECONDARY: $(UTILSOBJ)
38:
39: # Build targets
40: $(BINDIR)/%: $(OBJDIR)/%.o $(UTILS_OBJ) $(HDRDIR)/**/*.h $(HDRDIR)/*.h $(SRCDIR)/**/*.h
41:     $(CC) $(CFLAGS) -o $@ $(filter %.o,$^) $(LDFLAGS)
42:     chmod +x $@
43:
44: # Build generic .o file from .cpp file
45: $(OBJDIR)/%.o: $(SRCDIR)/%.cpp $(HDRDIR)/**/*.h $(HDRDIR)/*.h $(SRCDIR)/**/*.h
46:     $(CC) $(CFLAGS) -c $< -o $@ $(LDFLAGS)
47:
48: # Note that if I modify any header everything is built again
49: # This is not very effective but for such small project that's not an issue
50:
51: # Build documentation pdf
52: $(DOCDIR)/$(DOCPDFNAME): $(SRCS) $(HDRDIR)/**/*.h $(HDRDIR)/*.h $(DOXYGENCFG)
53:     doxygen $(DOXYGENCFG)
54:     ( cd $(DOCTMPDIR)/latex ; make )
55:     cp $(DOCTMPDIR)/latex/refman.pdf $(DOCDIR)/$(DOCPDFNAME)
56:
57:
58: # prepare sorted source list for source code pdf generation
59: # I want the header to appear right before the c source file
60: # the source files of client and server will be last
61: both = $(HDRDIR)/$(1).h $(SRCDIR)/$(1).cpp
62: ALL_SOURCES = $(foreach x,$(UTILS),$(call both,$(x)))
63: ALL_SOURCES += logging.h
64: ALL_SOURCES += $(addprefix $(SRCDIR)/,$(addsuffix .cpp,$(TARGETS)))
65:
```

```

66: # Build source code ps file
67: $(OBJDIR)/sources.ps : $(SRCDIR)/**/*.cpp $(SRCDIR)/**/*.h $(HDRDIR)/*.h
68:     echo $(ALL_SOURCES)
69:     enscript -C -fCourier9 --highlight=c -p$(OBJDIR)/sources.ps $(ALL_SOURCES)
70:
71: # Build makefile ps file
72: $(OBJDIR)/makefile.ps: Makefile
73:     enscript -C -fCourier9 --highlight=makefile -p$(OBJDIR)/makefile.ps Makefile
74:
75: $(OBJDIR)/sources.pdf: $(OBJDIR)/sources.ps
76:     ps2pdf $(OBJDIR)/sources.ps $(OBJDIR)/sources.pdf
77:
78: $(OBJDIR)/makefile.pdf: $(OBJDIR)/makefile.ps
79:     ps2pdf $(OBJDIR)/makefile.ps $(OBJDIR)/makefile.pdf
80:
81: # Builds source code pdf file
82: $(DOCDIR)/$(SRCPDFNAME) : $(OBJDIR)/sources.pdf $(OBJDIR)/makefile.pdf
83:     pdfunite $(OBJDIR)/makefile.pdf $(OBJDIR)/sources.pdf $(DOCDIR)/$(SRCPDFNAME)
84:
85: # Builds everything (ecutables and documentation)
86: all: exe $(DOCDIR)/$(DOCPDFNAME) $(DOCDIR)/$(SRCPDFNAME) report
87:
88: # clean everything
89: clean:
90:     $(RM) -r $(OBJDIR)/* $(BINDIR)/* $(DOCDIR)/$(DOCPDFNAME) $(DOCDIR)/$(SRCPDFNAME)
doc/report/report.pdf
91:
92: # clean everything and then rebuild
93: rebuild: clean all
94:
95: # just opens output documentation
96: doc_open: $(DOCDIR)/$(DOCPDFNAME)
97:     xdg-open $(DOCDIR)/$(DOCPDFNAME)
98:
99: # generates documentation and opens it in default pdf viewer
100: doc: $(DOCDIR)/$(DOCPDFNAME) doc_open
101:
102: # makes source code pdf and opens it
103: source: $(DOCDIR)/$(SRCPDFNAME)
104:     xdg-open doc/source_code.pdf
105:
106: test: exe
107:     @ i=0; \
108:         pass=0; \
109:         for test_script in tests/*.sh; do \
110:             echo -n "$$test_script ... "; \
111:             sh $$test_script > /dev/null 2> /dev/null; \
112:             if [ "$$?" -eq "0" ]; \
113:             then \
114:                 echo "PASS"; \
115:                 pass=$(( $$pass+1 )); \
116:             else \
117:                 echo "FAIL"; \
118:             fi; \
119:             i=$(( $$i+1 )); \
120:         done; \
121:         echo "Passed $$pass out of $$i"
122:
123: doc/report/report.pdf: doc/report/*.tex
124:     cd doc/report; latexmk -pdf report.tex
125:
126: report: doc/report/report.pdf
127:
128: help:
129:     @echo "all:          builds everything (both binaries and documentation)"
130:     @echo "clean:         deletes any intermediate or output file in build/, dist/ and d
oc/"
131:     @echo "report:        builds the report only"
132:     @echo "doc:           builds documentation only and opens pdf file"

```

```
Makefile      gio giu 18 02:28:54 2020      3

133:      @echo "doc_open:      opens documentation pdf"
134:      @echo "exe:          builds only binaries"
135:      @echo "help:          shows this message"
136:      @echo "rebuild:       same as calling clean and then all"
137:      @echo "source:        makes source code pdf and opens it"
138:      @echo "test:          runs all tests defined in tests/*.sh"
139:
140: # these targets aren't name of files
141: .PHONY: all exe clean rebuild doc_open doc help source report
142:
143: # build project structure
144: $(shell mkdir -p $(DOCDIR) $(addprefix $(OBJDIR)/,$(FOLDERS)) $(BINDIR)/client $(BINDIR)
/server test)
```

```
1: /**
2:  * @file connect4.cpp
3:  * @author Mirko Laruina
4:  *
5:  * @brief Implementation of connect4.h
6:  *
7:  * @date 2020-05-14
8:  *
9:  * @see connect4.h
10:  *
11:  */
12: #include "connect4.h"
13: using namespace std;
14:
15: void Connect4::print(ostream& os){
16:     os<<*this;
17: }
18:
19: Connect4::Connect4(int rows /* = 6 */, int columns /* = 7 */){
20:     rows_ = rows;
21:     cols_ = columns;
22:     size_ = rows*columns;
23:     full_ = false;
24:
25:     //Maybe check for overflow if we will use different board values
26:
27:     cells_ = new char[size_];
28:     memset(cells_, 0, size_);
29: }
30:
31: int8_t Connect4::play(int col, char player){
32:     // bool col_full = true;
33:     if(player == 0){
34:         player = player_;
35:     }
36:
37:     //Trying to play with a full board
38:     if(full_){
39:         return -2;
40:     }
41:
42:     for(int i = rows_-1; i>=0; --i){
43:         if(cells_[i*cols_+col] == 0){
44:             // col_full = false;
45:             cells_[i*cols_+col] = player;
46:             if( checkWin(i, col, player) ){
47:                 return 1;
48:             } else {
49:                 //All the board could be full now
50:                 if(i == 0 && checkFullTopRow()){
51:                     full_ = true;
52:                     return -2;
53:                 } else {
54:                     return 0;
55:                 }
56:             }
57:         }
58:     }
59:
60:     //We are sure the board is not full, otherwise we would have already exited
61:     //If a play was possible, we would have already exited too
62:     //Only possible case is full column
63:     return -1;
64: }
65:
66: int Connect4::countNexts(char player, int row, int col, int di, int dj){
67:     int count = 0;
68:     for(
69:         int i = row+di, j = col+dj;
```

```
70:         i >= 0 && j >= 0 && i < rows_ && j < cols_;
71:         i+=di, j+=dj)
72:     {
73:         if(cells_[i*cols_+j] != player){
74:             break;
75:         } else {
76:             LOG(LOG_DEBUG, "%d %d", i, j);
77:             count++;
78:         }
79:     }
80:     return count;
81: }
82:
83: bool Connect4::checkWin(int row, int col, char player){
84:     /*
85:         Take any of the 4 possible directions
86:         count how many token of the same player there are
87:         before and after the new one
88:         if more than 4, declare win
89:     */
90:
91:     LOG(LOG_DEBUG, "Checking (%d, %d)", row, col);
92:     if(player == 0){
93:         player = player_;
94:     }
95:
96:     for(int di = 1; di >= 0 && di != -1; --di){
97:         for(int dj = 1; dj >= 0 && dj != -1; --dj){
98:             // direction (0, 0) is useless, since we would miss diagonal (-1, 1)
99:             // we can exploit the loop to iterate over that
100:            if(di == 0 && dj == 0){
101:                di = -1;
102:                dj = 1;
103:            }
104:
105:            int count_forward = Connect4::countNexts(player, row, col, di, dj);
106:            int count_backward = Connect4::countNexts(player, row, col, -di, -dj);
107:
108:            // N_IN_A_ROW minus 1 since the token just inserted is excluded
109:            if(count_forward + count_backward >= (N_IN_A_ROW - 1)){
110:                return true;
111:            }
112:        }
113:    }
114:
115:    return false;
116:
117: }
118:
119: bool Connect4::checkFullTopRow(){
120:     for(int j = 0; j<cols_; ++j){
121:         if(cells_[j] == 0){
122:             return false;
123:         }
124:     }
125:     return true;
126: }
127:
128: int Connect4::getNumCols(){
129:     return cols_;
130: }
131:
132: bool Connect4::setPlayer(char player){
133:     if(player == 'X' || player == 'x'
134:        || player == 'O' || player == 'o'){
135:         player_ = toupper(player);
136:         adversary_ = player_ == 'X' ? 'O' : 'X';
137:         return true;
138:     }
```

```
139:     return false;
140: }
141:
142: char Connect4::getPlayer(){
143:     return player_;
144: }
145:
146: char Connect4::getAdv(){
147:     return adversary_;
148: }
149:
150: ostream& operator<<(ostream& os, const Connect4& c){
151:     int width = 2+3*(c.rows_+1);
152:     for(int i = 0; i<width; ++i){
153:         os<<'*';
154:     }
155:     os<<endl;
156:
157:     for(int i = 0; i<c.rows_; ++i){
158:         os<<"*";
159:         for(int j = 0; j<c.cols_; ++j){
160:             if(c.cells_[i*c.cols_+j] == 0){
161:                 os<<" ";
162:             } else {
163:                 os<<" " << (c.cells_[i*c.cols_+j] == 'X' ? "\033[31mX" : "\033[34mO") <<"
";
164:             }
165:         }
166:         os<<"\033[0m*"<<endl;
167:     }
168:
169:     for(int i = 0; i<width; ++i){
170:         os<<'*';
171:     }
172:     os<<endl;
173:
174:     for(int i = 0; i<width; ++i){
175:         if( (i+1)%3 == 0 ){
176:             os<<(i+1)/3;
177:         } else {
178:             os<<" ";
179:         }
180:     }
181:     os<<endl;
182:     return os;
183: }
```

```
1: /**
2:  * @file inet_utils.h
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Utility functions for managing inet addresses.
6:  *
7:  * This library provides functions for creating sockaddr_in structures from
8:  * IP address string and integer port number and for binding to a random
9:  * port (chosen using rand() builtin C function).
10:  *
11:  * @date 2020-05-17
12:  *
13:  * @see sockaddr_in
14:  * @see rand
15:  */
16:
17: #ifndef INET_UTILS
18: #define INET_UTILS
19:
20:
21: #include <sys/socket.h>
22: #include <netinet/in.h>
23: #include <string>
24:
25: using namespace std;
26:
27: /** Random port will be greater or equal to FROM_PORT */
28: #define FROM_PORT 49152
29:
30: /** Random port will be lower or equal to TO_PORT */
31: #define TO_PORT 65535
32:
33: /** Maximum number of trials before giving up opening a random port */
34: #define MAX_TRIES 256
35:
36: /**
37:  * Maximum number of characters of INET address to string
38:  * (eg 123.156.189.123:45678).
39:  */
40: #define MAX_SOCKADDR_STR_LEN 22
41:
42: /**
43:  * Size of a serialized sockaddr_in
44:  * 32 bit address + 16 bit port = 6 bytes
45:  */
46: #define SERIALIZED_SOCKADDR_IN_LEN 6
47:
48:
49: /**
50:  * Binds socket to a random port.
51:  *
52:  * @param socket    socket ID
53:  * @param addr      inet addr structure
54:  * @return          0 in case of failure, port it could bind to otherwise
55:  *
56:  * @see FROM_PORT
57:  * @see TO_PORT
58:  * @see MAX_TRIES
59:  */
60: int bind_random_port(int socket, struct sockaddr_in *addr);
61:
62: /**
63:  * Makes sockaddr_in structure given ip string and port of server.
64:  *
65:  * @param ip        ip address of server
66:  * @param port      port of the server
67:  * @return          sockaddr_in structure for the given server
68:  */
69: struct sockaddr_in make_sv_sockaddr_in(const char* ip, int port);
```

```
70:
71: /**
72:  * Makes sockaddr_in structure of this host.
73:  *
74:  * INADDR_ANY is used as IP address.
75:  *
76:  * @param port    port of the server
77:  * @return        sockaddr_in structure this host on given port
78:  */
79: struct sockaddr_in make_my_sockaddr_in(int port);
80:
81: /**
82:  * Compares INET addresses, returning 0 in case they're equal.
83:  *
84:  * @param sai1    first address
85:  * @param sai2    second address
86:  * @return        0 if they're equal, 1 otherwise
87:  */
88: int sockaddr_in_cmp(struct sockaddr_in sai1, struct sockaddr_in sai2);
89:
90: /**
91:  * Converts sockaddr_in structure to string to be printed.
92:  *
93:  * @param src     the input address
94:  */
95: string sockaddr_in_to_string(struct sockaddr_in src);
96:
97: /**
98:  * Serializes sockaddr_in structure to given buffer.
99:  *
100:  * @param src     the input address
101:  * @param buffer  the buffer
102:  */
103: int writeSockAddrIn(char* buffer, size_t buf_len, struct sockaddr_in src);
104:
105: /**
106:  * Deerializes sockaddr_in structure from given buffer.
107:  *
108:  * @param buffer  the buffer
109:  * @return        the built sockaddr_in struct
110:  */
111: int readSockAddrIn(struct sockaddr_in *src, char* buffer, size_t buf_len);
112:
113: #endif
```



```
1: /**
2:  * @file inet_utils.cpp
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Implementation of inet_utils.h.
6:  *
7:  * @see inet_utils.h
8:  *
9:  * @date 2020-05-17
10: */
11:
12: #include <stdlib.h>
13: #include <string>
14: #include <string.h>
15: #include <sys/socket.h>
16: #include <netinet/in.h>
17: #include <arpa/inet.h>
18: #include <stdint.h>
19:
20: #include "logging.h"
21:
22: #include "network/inet_utils.h"
23:
24: using namespace std;
25:
26: int bind_random_port(int socket, struct sockaddr_in *addr){
27:     int port, ret, i;
28:     for (i = 0; i < MAX_TRIES; i++){
29:         if (i == 0) // first I generate a random one
30:             port = rand() % (TO_PORT - FROM_PORT + 1) + FROM_PORT;
31:         else //if it's not free I scan the next one
32:             port = (port - FROM_PORT + 1) % (TO_PORT - FROM_PORT + 1) + FROM_PORT;
33:
34:         LOG(LOG_DEBUG, "Trying port %d...", port);
35:
36:         addr->sin_port = htons(port);
37:         ret = bind(socket, (struct sockaddr *)addr, sizeof(*addr));
38:         if (ret != -1)
39:             return port;
40:         // consider only some errors?
41:     }
42:     LOG(LOG_ERR, "Could not bind to random port after %d attempts", MAX_TRIES);
43:     return 0;
44: }
45:
46: struct sockaddr_in make_sv_sockaddr_in(const char *ip, int port){
47:     struct sockaddr_in addr;
48:     memset(&addr, 0, sizeof(addr));
49:     addr.sin_family = AF_INET;
50:     addr.sin_port = htons(port);
51:     inet_pton(AF_INET, ip, &addr.sin_addr);
52:     return addr;
53: }
54:
55: struct sockaddr_in make_my_sockaddr_in(int port){
56:     struct sockaddr_in addr;
57:     memset(&addr, 0, sizeof(addr));
58:     addr.sin_family = AF_INET;
59:     addr.sin_port = htons(port);
60:     addr.sin_addr.s_addr = htonl(INADDR_ANY);
61:     return addr;
62: }
63:
64: int sockaddr_in_cmp(struct sockaddr_in sai1, struct sockaddr_in sai2){
65:     if (sai1.sin_port == sai2.sin_port &&
66:         sai1.sin_addr.s_addr == sai2.sin_addr.s_addr)
67:         return 0;
68:     else
69:         return 1;
```

```
70: }
71:
72: string sockaddr_in_to_string(struct sockaddr_in src){
73:     char dst[MAX_SOCKADDR_STR_LEN];
74:     char port_str[6];
75:     const char *ret;
76:
77:     sprintf(port_str, "%d", ntohs(src.sin_port));
78:
79:     ret = inet_ntop(AF_INET, (void *)&src.sin_addr, dst, MAX_SOCKADDR_STR_LEN);
80:     if (ret != NULL){
81:         strcat(dst, ":");
82:         strcat(dst, port_str);
83:     } else {
84:         strcpy(dst, "ERROR");
85:     }
86:
87:     string s = dst;
88:
89:     return s;
90: }
91:
92: int writeSockAddrIn(char* buffer, size_t buf_len, struct sockaddr_in src){
93:     if (buf_len < SERIALIZED_SOCKADDR_IN_LEN)
94:         return -1;
95:
96:     memcpy(buffer,
97:            &src.sin_addr.s_addr,
98:            sizeof(src.sin_addr.s_addr));
99:     memcpy(buffer+sizeof(src.sin_addr.s_addr),
100:            &src.sin_port,
101:            sizeof(src.sin_port));
102:     return SERIALIZED_SOCKADDR_IN_LEN;
103: }
104:
105: int readSockAddrIn(struct sockaddr_in *dst, char* buffer, size_t buf_len){
106:     if (buf_len < SERIALIZED_SOCKADDR_IN_LEN){
107:         return -1;
108:     }
109:
110:     memset(dst, 0, sizeof(*dst));
111:     dst->sin_family = AF_INET;
112:     memcpy(&(dst->sin_addr.s_addr),
113:            buffer,
114:            sizeof(dst->sin_addr.s_addr));
115:     memcpy(&(dst->sin_port),
116:            buffer+sizeof(dst->sin_addr.s_addr),
117:            sizeof(dst->sin_port));
118:     return SERIALIZED_SOCKADDR_IN_LEN;
119: }
```

```
1: /**
2:  * @file messages.h
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Definition of messages
6:  *
7:  * @date 2020-05-17
8:  */
9:
10: #ifndef MESSAGES_H
11: #define MESSAGES_H
12:
13: #include <string>
14: #include <stdint.h>
15: #include <sys/socket.h>
16: #include <netinet/in.h>
17:
18: #include "logging.h"
19: #include "config.h"
20: #include "network/inet_utils.h"
21: #include "network/host.h"
22: #include "security/crypto.h"
23: #include "security/secure_host.h"
24:
25: using namespace std;
26:
27: /** Type of message length (first N bytes of packet) */
28: typedef uint16_t msglen_t;
29:
30: #define MSGLEN_HTON(x) htons((x))
31: #define MSGLEN_NTOH(x) ntohs((x))
32:
33: /** Utility for getting username length (excluding '\0') */
34: inline size_t usernameLength(string s){
35:     return min(strlen(s.c_str()), (size_t) MAX_USERNAME_LENGTH);
36: }
37:
38: /** Utility for getting username from buffer
39:  *
40:  * @param buf the buffer to read the string from
41:  * @param buflen the size of the buffer
42:  * @returns the read string
43:  */
44: inline int readUsername(string *username, char* buf, size_t buf_len){
45:     size_t size = min(strlen(buf, buf_len-1), (size_t) MAX_USERNAME_LENGTH);
46:     *username = string(buf, size);
47:     return MAX_USERNAME_LENGTH+1;
48: }
49:
50: /**
51:  * Utility for writing username to buffer
52:  *
53:  * NB: buffer must be large enough
54:  *
55:  * @param s the username string to be written on buffer
56:  * @param buf the buffer to write the string to
57:  * @returns number of written bytes
58:  */
59: inline int writeUsername(char* buf, size_t buf_len, string s){
60:     if (buf_len < MAX_USERNAME_LENGTH+1){
61:         return -1;
62:     }
63:     size_t strsize = usernameLength(s);
64:     strncpy(buf, s.c_str(), strsize);
65:     memset(&buf[strsize], 0, MAX_USERNAME_LENGTH-strsize+1);
66:     return MAX_USERNAME_LENGTH+1;
67: }
68:
69: /**
```

```
70:  * Possible type of messages.
71:  *
72:  * When adding a new message class, add a related type here and set its
73:  * getType method to return it.
74:  */
75: enum MessageType
76: {
77:     SECURE_MESSAGE,
78:     CLIENT_HELLO,
79:     SERVER_HELLO,
80:     CLIENT_VERIFY,
81:     START_GAME_PEER,
82:     MOVE,
83:     REGISTER,
84:     CHALLENGE,
85:     GAME_END,
86:     USERS_LIST,
87:     USERS_LIST_REQ,
88:     CHALLENGE_FWD,
89:     CHALLENGE_RESP,
90:     GAME_START,
91:     GAME_CANCEL,
92:     CERT_REQ,
93:     CERTIFICATE,
94: };
95:
96: /**
97:  * Abstract class for Messages.
98:  */
99: class Message
100: {
101: public:
102:     virtual ~Message(){};
103:     /**
104:      * Write message to buffer
105:      *
106:      * @returns >0 number of bytes written
107:      * @returns 0 in case of errors
108:      */
109:     virtual msglen_t write(char *buffer) = 0;
110:
111:     /**
112:      * Read message from buffer
113:      *
114:      * @returns 0 in case of success, something else in case of errors.
115:      *       Refer to the implementation for details
116:      */
117:     virtual msglen_t read(char *buffer, msglen_t len) = 0;
118:
119:     /**
120:      * Get message name (for debug purposes)
121:      */
122:     virtual string getName() = 0;
123:
124:     virtual MessageType getType() = 0;
125: };
126:
127: /**
128:  * Message that signals to start a new game.
129:  */
130: class StartGameMessage : public Message
131: {
132: public:
133:     StartGameMessage() {}
134:     ~StartGameMessage() {}
135:
136:     msglen_t write(char *buffer);
137:     msglen_t read(char *buffer, msglen_t len);
138: }
```

```
139:     string getName() { return "StartGame"; }
140:
141:     MessageType getType() { return START_GAME_PEER; }
142: };
143:
144: /**
145:  * Message that signals a move
146:  */
147: class MoveMessage : public Message
148: {
149: private:
150:     char col;
151:
152: public:
153:     MoveMessage() {}
154:     MoveMessage(char col) : col(col) {}
155:     ~MoveMessage() {}
156:
157:     msglen_t write(char *buffer);
158:     msglen_t read(char *buffer, msglen_t len);
159:
160:     string getName() { return "Move"; }
161:
162:     char getColumn() { return col; }
163:
164:     MessageType getType() { return MOVE; }
165: };
166:
167: /**
168:  * Message that permits the client to register to server
169:  */
170: class RegisterMessage : public Message
171: {
172: private:
173:     string username;
174:
175: public:
176:     RegisterMessage() {}
177:     RegisterMessage(string username) : username(username) {}
178:     ~RegisterMessage() {}
179:
180:     msglen_t write(char *buffer);
181:     msglen_t read(char *buffer, msglen_t len);
182:
183:     string getName() { return "Register"; }
184:
185:     string getUsername() { return username; }
186:
187:     MessageType getType() { return REGISTER; }
188: };
189:
190: /**
191:  * Message that permits the client to challenge another client
192:  * through the server.
193:  */
194: class ChallengeMessage : public Message
195: {
196: private:
197:     string username;
198:
199: public:
200:     ChallengeMessage() {}
201:     ChallengeMessage(string username) : username(username) {}
202:     ~ChallengeMessage() {}
203:
204:     msglen_t write(char *buffer);
205:     msglen_t read(char *buffer, msglen_t len);
206:
207:     string getName() { return "Challenge"; }
```

```
208:
209:     string getUsername() { return username; }
210:
211:     MessageType getType() { return CHALLENGE; }
212: };
213:
214: /**
215:  * Message that signals the server that the client is available
216:  */
217: class GameEndMessage : public Message
218: {
219: public:
220:     GameEndMessage() {}
221:     ~GameEndMessage() {}
222:
223:     msglen_t write(char *buffer);
224:     msglen_t read(char *buffer, msglen_t len);
225:
226:     string getName() { return "Game End"; }
227:
228:     MessageType getType() { return GAME_END; }
229: };
230:
231: /**
232:  * Message that the server sends the client with the list of users
233:  */
234: class UsersListMessage : public Message
235: {
236: private:
237:     string usernames;
238:
239: public:
240:     UsersListMessage() {}
241:     UsersListMessage(string usernames) : usernames(usernames) {}
242:     ~UsersListMessage() {}
243:
244:     msglen_t write(char *buffer);
245:     msglen_t read(char *buffer, msglen_t len);
246:
247:     string getName() { return "User list"; }
248:
249:     string getUsernames() { return usernames; }
250:
251:     MessageType getType() { return USERS_LIST; }
252: };
253:
254: /**
255:  * Message with which the client asks for the list of connected users.
256:  */
257: class UsersListRequestMessage : public Message
258: {
259: private:
260:     uint32_t offset;
261:
262: public:
263:     UsersListRequestMessage() : offset(0) {}
264:     UsersListRequestMessage(unsigned int offset) : offset(offset) {}
265:     ~UsersListRequestMessage() {}
266:
267:     msglen_t write(char *buffer);
268:     msglen_t read(char *buffer, msglen_t len);
269:
270:     string getName() { return "Users list request"; }
271:
272:     uint32_t getOffset() { return offset; }
273:
274:     MessageType getType() { return USERS_LIST_REQ; }
275: };
276:
```

```
277: /**
278:  * Message with which the server forwards a challenge.
279:  */
280: class ChallengeForwardMessage : public Message
281: {
282: private:
283:     string username;
284:
285: public:
286:     ChallengeForwardMessage() {}
287:     ChallengeForwardMessage(string username) : username(username) {}
288:     ~ChallengeForwardMessage() {}
289:
290:     msglen_t write(char *buffer);
291:     msglen_t read(char *buffer, msglen_t len);
292:
293:     string getName() { return "Challenge forward"; }
294:
295:     string getUsername() { return username; }
296:
297:     MessageType getType() { return CHALLENGE_FWD; }
298: };
299:
300: /**
301:  * Message with which the client replies to a challenge.
302:  */
303: class ChallengeResponseMessage : public Message
304: {
305: private:
306:     string username;
307:     bool response;
308:     uint16_t listen_port;
309:
310: public:
311:     ChallengeResponseMessage() {}
312:     ChallengeResponseMessage(string username, bool response, uint16_t port)
313:         : username(username), response(response), listen_port(port) {}
314:     ~ChallengeResponseMessage() {}
315:
316:     msglen_t write(char *buffer);
317:     msglen_t read(char *buffer, msglen_t len);
318:
319:     string getName() { return "Challenge response"; }
320:
321:     string getUsername() { return username; }
322:     bool getResponse() { return response; }
323:     uint16_t getListenPort() { return listen_port; }
324:
325:     MessageType getType() { return CHALLENGE_RESP; }
326: };
327:
328: /**
329:  * Message with which the server forwards a challenge rejectal or another
330:  * event that caused the game to be canceled.
331:  */
332: class GameCancelMessage : public Message
333: {
334: private:
335:     string username;
336:
337: public:
338:     GameCancelMessage() {}
339:     GameCancelMessage(string username) : username(username) {}
340:     ~GameCancelMessage() {}
341:
342:     msglen_t write(char *buffer);
343:     msglen_t read(char *buffer, msglen_t len);
344:
345:     string getName() { return "Game cancel"; }
```

```

346:
347:     string getUsername() { return username; }
348:
349:     MessageType getType() { return GAME_CANCEL; }
350: };
351:
352: /**
353:  * Message with which the server makes a new game start between clients.
354:  */
355: class GameStartMessage : public Message
356: {
357: private:
358:     string username;
359:     struct sockaddr_in addr;
360:     X509* cert;
361:
362: public:
363:     GameStartMessage() {}
364:     GameStartMessage(string username, struct sockaddr_in addr, X509* opp_cert)
365:         : username(username), addr(addr), cert(opp_cert) {} //TODO cert
366:     ~GameStartMessage() {}
367:
368:     msglen_t write(char *buffer);
369:     msglen_t read(char *buffer, msglen_t len);
370:
371:     string getName() { return "Game start"; }
372:
373:     string getUsername() { return username; }
374:     struct sockaddr_in getAddr() { return addr; }
375:     SecureHost getHost() { return SecureHost(addr, cert); }
376:     X509* getCert() { return cert; }
377:     MessageType getType() { return GAME_START; }
378: };
379:
380: class SecureMessage : public Message
381: {
382: private:
383:     char *ct;
384:     msglen_t ct_size;
385:     char* tag;
386:
387: public:
388:     SecureMessage() : ct(NULL), ct_size(0), tag(NULL){}
389:     SecureMessage(char* ct, msglen_t ct_size, char* tag) : ct(ct), ct_size(ct_size), tag(t
ag){}
390:     ~SecureMessage(){ if (ct != NULL) free(ct); if (tag != NULL) free(tag); }
391:
392:     MessageType getType() { return SECURE_MESSAGE; }
393:     string getName() { return "Secure message"; }
394:
395:     void setCtSize(msglen_t s) { ct_size = s; }
396:     size_t getCtSize() { return ct_size; }
397:     char* getCt() { return ct; }
398:     char* getTag() { return tag; }
399:
400:     msglen_t write(char *buffer);
401:     msglen_t read(char *buffer, msglen_t len);
402: };
403:
404: class ClientHelloMessage: public Message
405: {
406: private:
407:     EVP_PKEY* eph_key;
408:     nonce_t nonce;
409:     string my_id;
410:     string other_id;
411:
412: public:
413:     ClientHelloMessage() : eph_key(NULL) {}

```



```
414:     ClientHelloMessage(EVP_PKEY* eph_key, nonce_t nonce, string my_id, string other_id)
415:         : eph_key(eph_key), nonce(nonce), my_id(my_id), other_id(other_id) {}
416:
417:     MessageType getType() {return CLIENT_HELLO; }
418:     string getName() { return "Client Hello message"; }
419:
420:     nonce_t getNonce() { return nonce; }
421:     EVP_PKEY* getEphKey() { return eph_key; }
422:     void setEphKey(EVP_PKEY* eph_key) { this->eph_key=eph_key; }
423:     string getMyId() { return my_id; }
424:     string getOtherId() { return other_id; }
425:
426:     msglen_t write(char* buffer);
427:     msglen_t read(char* buffer, msglen_t len);
428: };
429:
430: class ClientVerifyMessage: public Message
431: {
432: private:
433:     char* ds;
434:     uint32_t ds_size;
435:
436: public:
437:     ClientVerifyMessage() : ds(NULL), ds_size(0) {}
438:     ClientVerifyMessage(char* ds, uint32_t ds_size) : ds(ds), ds_size(ds_size) {}
439:     ~ClientVerifyMessage();
440:
441:     MessageType getType() {return CLIENT_VERIFY; }
442:     string getName() { return "Client Verify message"; }
443:
444:     char* getDs() { return ds; }
445:     uint32_t getDsSize() { return ds_size; }
446:
447:     msglen_t write(char* buffer);
448:     msglen_t read(char* buffer, msglen_t len);
449: };
450:
451: class ServerHelloMessage: public Message
452: {
453: private:
454:     EVP_PKEY* eph_key;
455:     nonce_t nonce;
456:     string my_id;
457:     string other_id;
458:     char* ds;
459:     uint32_t ds_size;
460:
461: public:
462:     ServerHelloMessage() : eph_key(NULL), ds(NULL), ds_size(0) {}
463:     ServerHelloMessage(EVP_PKEY* eph_key, nonce_t nonce, string my_id, string other_id, ch
ar* ds, uint32_t ds_size)
464:         : eph_key(eph_key), nonce(nonce), my_id(my_id), other_id(other_id), ds(ds), ds_siz
e(ds_size) {}
465:     ~ServerHelloMessage();
466:
467:     MessageType getType() {return SERVER_HELLO; }
468:     string getName() { return "Server Hello message"; }
469:
470:     nonce_t getNonce() { return nonce; }
471:     EVP_PKEY* getEphKey() { return eph_key; }
472:     void setEphKey(EVP_PKEY* eph_key) { this->eph_key=eph_key; }
473:     string getMyId() { return my_id; }
474:     string getOtherId() { return other_id; }
475:     char* getDs() { return ds; }
476:     uint32_t getDsSize() { return ds_size; }
477:
478:     msglen_t write(char* buffer);
479:     msglen_t read(char* buffer, msglen_t len);
480: };
```

```
481:
482: class CertificateRequestMessage: public Message
483: {
484: public:
485:     CertificateRequestMessage(){}
486:
487:     MessageType getType() {return CERT_REQ; }
488:     string getName() { return "Certificate Request message"; }
489:
490:     msglen_t write(char* buffer);
491:     msglen_t read(char* buffer, msglen_t len);
492: };
493:
494: class CertificateMessage: public Message
495: {
496: private:
497:     X509* cert;
498: public:
499:     CertificateMessage() : cert(NULL) {}
500:     CertificateMessage(X509* cert) : cert(cert) {}
501:
502:     MessageType getType() {return CERTIFICATE; }
503:     string getName() { return "Certificate message"; }
504:     X509* getCert() { return cert; }
505:
506:     msglen_t write(char* buffer);
507:     msglen_t read(char* buffer, msglen_t len);
508: };
509:
510: /**
511:  * Reads the message using the correct class and returns a pointer to it.
512:  *
513:  * NB: remeber to dispose of the created Message when you are done with it.
514:  */
515: Message *readMessage(char *buffer, msglen_t len);
516:
517: #endif // MESSAGES_H
```

```
1:  /**
2:  * @file messages.cpp
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Implementation of messages.h
6:  *
7:  * @see messages.h
8:  */
9:
10: #include <cstdlib>
11: #include <cstring>
12: #include <cmath>
13:
14: #include "network/messages.h"
15: #include "network/inet_utils.h"
16:
17: #include "security/crypto_utils.h"
18: #include "utils/buffer_io.h"
19:
20: Message* readMessage(char *buffer, msglen_t len){
21:     Message *m;
22:     int ret;
23:
24:     switch(buffer[0]){
25:         case SECURE_MESSAGE:
26:             m = new SecureMessage;
27:             break;
28:         case CLIENT_HELLO:
29:             m = new ClientHelloMessage;
30:             break;
31:         case SERVER_HELLO:
32:             m = new ServerHelloMessage;
33:             break;
34:         case CLIENT_VERIFY:
35:             m = new ClientVerifyMessage;
36:             break;
37:         case START_GAME_PEER:
38:             m = new StartGameMessage;
39:             break;
40:         case MOVE:
41:             m = new MoveMessage;
42:             break;
43:         case REGISTER:
44:             m = new RegisterMessage;
45:             break;
46:         case CHALLENGE:
47:             m = new ChallengeMessage;
48:             break;
49:         case GAME_END:
50:             m = new GameEndMessage;
51:             break;
52:         case USERS_LIST:
53:             m = new UsersListMessage;
54:             break;
55:         case USERS_LIST_REQ:
56:             m = new UsersListRequestMessage;
57:             break;
58:         case CHALLENGE_FWD:
59:             m = new ChallengeForwardMessage;
60:             break;
61:         case CHALLENGE_RESP:
62:             m = new ChallengeResponseMessage;
63:             break;
64:         case GAME_START:
65:             m = new GameStartMessage;
66:             break;
67:         case GAME_CANCEL:
68:             m = new GameCancelMessage;
69:             break;
```

```
70:         case CERT_REQ:
71:             m = new CertificateRequestMessage;
72:             break;
73:         case CERTIFICATE:
74:             m = new CertificateMessage;
75:             break;
76:         default:
77:             m = NULL;
78:             LOG(LOG_ERR, "Unrecognized message type %d", buffer[0]);
79:             return NULL;
80:     };
81:
82:     ret = m->read(buffer, len);
83:
84:     if (ret != 0){
85:         LOG(LOG_ERR, "Error reading message of type %d: %d", buffer[0], ret);
86:         return NULL;
87:     } else{
88:         return m;
89:     }
90: }
91:
92: msglen_t StartGameMessage::write(char *buffer){
93:     int i = 0;
94:     int ret;
95:
96:     if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) START_GAME_PEER)) < 0)
97:         return 0;
98:     i += ret;
99:
100:     return i;
101: }
102:
103: msglen_t StartGameMessage::read(char *buffer, msglen_t len){
104:     return 0;
105: }
106:
107: msglen_t MoveMessage::write(char *buffer){
108:     int i = 0;
109:     int ret;
110:
111:     if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) MOVE)) < 0)
112:         return 0;
113:     i += ret;
114:
115:     if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, col)) < 0)
116:         return 0;
117:     i += ret;
118:
119:     return i;
120: }
121:
122: msglen_t MoveMessage::read(char *buffer, msglen_t len){
123:     int i = 1;
124:     int ret;
125:
126:     if ((ret = readUInt8((uint8_t*)&col, &buffer[i], len-i)) < 0)
127:         return 1;
128:     i += ret;
129:
130:     return 0;
131: }
132:
133: msglen_t RegisterMessage::write(char *buffer){
134:     int i = 0;
135:     int ret;
136:
137:     if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) REGISTER)) < 0)
138:         return 0;
```

```
139:     i += ret;
140:
141:     if ((ret = writeUsername(&buffer[i], MAX_MSG_SIZE-i, username)) < 0)
142:         return 0;
143:     i += ret;
144:
145:     return i;
146: }
147:
148: msglen_t RegisterMessage::read(char *buffer, msglen_t len){
149:     int i = 1;
150:     int ret;
151:
152:     if ((ret = readUsername(&username, &buffer[i], len-i)) < 0)
153:         return 1;
154:     i += ret;
155:
156:     return 0;
157: }
158:
159: msglen_t ChallengeMessage::write(char *buffer){
160:     int i = 0;
161:     int ret;
162:
163:     if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) CHALLENGE)) < 0)
164:         return 0;
165:     i += ret;
166:
167:     if ((ret = writeUsername(&buffer[i], MAX_MSG_SIZE-i, username)) < 0)
168:         return 0;
169:     i += ret;
170:
171:     return i;
172: }
173: msglen_t ChallengeMessage::read(char *buffer, msglen_t len){
174:     int i = 1;
175:     int ret;
176:
177:     if ((ret = readUsername(&username, &buffer[i], len-i)) < 0)
178:         return 1;
179:     i += ret;
180:
181:     return 0;
182: }
183:
184: msglen_t GameEndMessage::write(char *buffer){
185:     int i = 0;
186:     int ret;
187:
188:     if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) GAME_END)) < 0)
189:         return 0;
190:
191:     i += ret;
192:
193:     return i;
194: }
195: msglen_t GameEndMessage::read(char *buffer, msglen_t len){
196:     return 0;
197: }
198:
199: msglen_t UsersListMessage::write(char *buffer){
200:     int i = 0;
201:     int ret;
202:
203:     if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) USERS_LIST)) < 0)
204:         return 0;
205:     i += ret;
206:
207:     size_t strsize = min(usernames.size(),
```

```
208:         (size_t) ((MAX_USERNAME_LENGTH+1)*MAX_USERS));
209:     size_t padded_size = (strsize+MAX_USERNAME_LENGTH)/(MAX_USERNAME_LENGTH+1)*(MAX_USERNAME_LENGTH+1);
210:     if ((int)padded_size > MAX_MSG_SIZE-i)
211:         return 0;
212:     strncpy(&buffer[i], usernames.c_str(), strsize);
213:     memset(&buffer[i+strsize], 0, padded_size-strsize+1);
214:     i += padded_size+1;
215:
216:     return i;
217: }
218:
219: msglen_t UsersListMessage::read(char *buffer, msglen_t len){
220:     int maxsize = min((MAX_USERNAME_LENGTH+1)*MAX_USERS, len-1);
221:     if (maxsize <= 0){
222:         return 1;
223:     }
224:
225:     usernames = string(&buffer[i], maxsize);
226:     return 0;
227: }
228:
229: msglen_t UsersListRequestMessage::write(char *buffer){
230:     int i = 0;
231:     int ret;
232:
233:     if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) USERS_LIST_REQ)) < 0)
234:         return 0;
235:     i += ret;
236:
237:     if ((ret = writeUInt32(&buffer[i], MAX_MSG_SIZE-i, offset)) < 0)
238:         return 0;
239:     i += ret;
240:
241:     return i;
242: }
243:
244: msglen_t UsersListRequestMessage::read(char *buffer, msglen_t len){
245:     int i = 1;
246:     int ret;
247:
248:     if ((ret = readUInt32(&offset, &buffer[i], len-i)) < 0)
249:         return 1;
250:     i += ret;
251:
252:     return 0;
253: }
254:
255: msglen_t ChallengeForwardMessage::write(char *buffer){
256:     int i = 0;
257:     int ret;
258:
259:     if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) CHALLENGE_FWD)) < 0)
260:         return 0;
261:     i += ret;
262:
263:     if ((ret = writeUsername(&buffer[i], MAX_MSG_SIZE-i, username)) < 0)
264:         return 0;
265:     i += ret;
266:
267:     return i;
268: }
269: msglen_t ChallengeForwardMessage::read(char *buffer, msglen_t len){
270:     int i = 1;
271:     int ret;
272:
273:     if ((ret = readUsername(&username, &buffer[i], len-i)) < 0)
274:         return 1;
275:     i += ret;
```

```
276:
277:     return 0;
278: }
279:
280: msglen_t ChallengeResponseMessage::write(char *buffer){
281:     int i = 0;
282:     int ret;
283:
284:     if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) CHALLENGE_RESP)) < 0)
285:         return 0;
286:     i += ret;
287:
288:     if ((ret = writeBool(&buffer[i], MAX_MSG_SIZE-i, response)) < 0)
289:         return 0;
290:     i += ret;
291:
292:     if ((ret = writeUInt16(&buffer[i], MAX_MSG_SIZE-i, listen_port)) < 0)
293:         return 0;
294:     i += ret;
295:
296:     if ((ret = writeUsername(&buffer[i], MAX_MSG_SIZE-i, username)) < 0)
297:         return 0;
298:     i += ret;
299:
300:     return i;
301: }
302: msglen_t ChallengeResponseMessage::read(char *buffer, msglen_t len){
303:     int i = 1;
304:     int ret;
305:
306:     if ((ret = readBool(&response, &buffer[i], len-i)) < 0)
307:         return 1;
308:     i += ret;
309:
310:     if ((ret = readUInt16(&listen_port, &buffer[i], len-i)) < 0)
311:         return 1;
312:     i += ret;
313:
314:     if ((ret = readUsername(&username, &buffer[i], len-i)) < 0)
315:         return 1;
316:     i += ret;
317:
318:     return 0;
319: }
320:
321: msglen_t GameCancelMessage::write(char *buffer){
322:     int i = 0;
323:     int ret;
324:
325:     if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) GAME_CANCEL)) < 0)
326:         return 0;
327:     i += ret;
328:
329:     if ((ret = writeUsername(&buffer[i], MAX_MSG_SIZE-i, username)) < 0)
330:         return 0;
331:     i += ret;
332:
333:     return i;
334: }
335: msglen_t GameCancelMessage::read(char *buffer, msglen_t len){
336:     int i = 1;
337:     int ret;
338:
339:     if ((ret = readUsername(&username, &buffer[i], len-i)) < 0)
340:         return 1;
341:     i += ret;
342:
343:     return 0;
344: }
```

```
345:
346: msglen_t GameStartMessage::write(char *buffer){
347:     int i = 0;
348:     int ret;
349:
350:     if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) GAME_START)) < 0)
351:         return 0;
352:     i += ret;
353:
354:     if ((ret = writeSockAddrIn(&buffer[i], MAX_MSG_SIZE-i, addr)) < 0)
355:         return 0;
356:     i += ret;
357:
358:     if ((ret = writeUsername(&buffer[i], MAX_MSG_SIZE-i, username)) < 0)
359:         return 0;
360:     i += ret;
361:
362:     if ((ret = cert2buf(cert, &buffer[i], MAX_MSG_SIZE - i)) < 0)
363:         return 0;
364:     i += ret;
365:
366:     return i;
367: }
368:
369: msglen_t GameStartMessage::read(char *buffer, msglen_t len){
370:     int i = 1;
371:     int ret;
372:
373:     if ((ret = readSockAddrIn(&addr, &buffer[i], len-i)) < 0)
374:         return 1;
375:     i += ret;
376:
377:     if ((ret = readUsername(&username, &buffer[i], len-i)) < 0)
378:         return 1;
379:     i += ret;
380:
381:     if ((ret = buf2cert(&buffer[i], len - i, &cert)) < 0)
382:         return 1;
383:     i += ret;
384:
385:     return 0;
386: }
387:
388: msglen_t SecureMessage::write(char* buffer){
389:     int i = 0;
390:     int ret;
391:
392:     if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) SECURE_MESSAGE)) < 0)
393:         return 0;
394:     i += ret;
395:
396:     if ((ret = writeBuf(&buffer[i], MAX_MSG_SIZE-i, ct, ct_size)) < 0)
397:         return 0;
398:     i += ret;
399:
400:     if ((ret = writeBuf(&buffer[i], MAX_MSG_SIZE-i, tag, TAG_SIZE)) < 0)
401:         return 0;
402:     i += ret;
403:
404:     return i;
405: }
406:
407: msglen_t SecureMessage::read(char* buffer, msglen_t len){
408:     ct_size = len-1-TAG_SIZE;
409:     ct = (char*) malloc(ct_size);
410:     tag = (char*) malloc(TAG_SIZE);
411:     if(!ct || !tag){
412:         LOG(LOG_WARN, "Malloc failed for message of length %d", len);
413:         return -1;
```



```
414:     }
415:
416:     int i = 1;
417:     int ret;
418:
419:     if ((ret = readBuf(ct, ct_size, &buffer[i], len-i)) < 0)
420:         return 1;
421:     i += ret;
422:
423:     if ((ret = readBuf(tag, TAG_SIZE, &buffer[i], len-i)) < 0)
424:         return 1;
425:     i += ret;
426:
427:     return 0;
428: }
429:
430: msglen_t ClientHelloMessage::write(char* buffer){
431:     int i = 0;
432:     int ret;
433:
434:     if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) CLIENT_HELLO)) < 0)
435:         return 0;
436:     i += ret;
437:
438:     if ((ret = writeUInt32(&buffer[i], MAX_MSG_SIZE-i, nonce)) < 0)
439:         return 0;
440:     i += ret;
441:
442:     if ((ret = writeUsername(&buffer[i], MAX_MSG_SIZE-i, my_id)) < 0)
443:         return 0;
444:     i += ret;
445:
446:     if ((ret = writeUsername(&buffer[i], MAX_MSG_SIZE-i, other_id)) < 0)
447:         return 0;
448:     i += ret;
449:
450:     if ((ret = pkey2buf(eph_key, &buffer[i], MAX_MSG_SIZE-i)) < 0)
451:         return 0;
452:     i += ret;
453:
454:     return i;
455: }
456:
457: msglen_t ClientHelloMessage::read(char* buffer, msglen_t len){
458:     int i = 1;
459:     int ret;
460:
461:     if ((ret = readUInt32(&nonce, &buffer[i], len-i)) < 0)
462:         return 1;
463:     i += ret;
464:
465:     if ((ret = readUsername(&my_id, &buffer[i], len-i)) < 0)
466:         return 1;
467:     i += ret;
468:
469:     if ((ret = readUsername(&other_id, &buffer[i], len-i)) < 0)
470:         return 1;
471:     i += ret;
472:
473:     if((ret = buf2pkey(&buffer[i], len-i, &eph_key)) < 0)
474:         return 1;
475:     i += ret;
476:
477:     return 0;
478: }
479:
480: ServerHelloMessage::~ServerHelloMessage(){
481:     if (ds != NULL){
482:         free(ds);
```

```
483:     }
484: }
485:
486: msglen_t ServerHelloMessage::write(char* buffer){
487:     int i = 0;
488:     int ret;
489:
490:     if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) SERVER_HELLO)) < 0)
491:         return 0;
492:     i += ret;
493:
494:     if ((ret = writeUInt32(&buffer[i], MAX_MSG_SIZE-i, nonce)) < 0)
495:         return 0;
496:     i += ret;
497:
498:     if ((ret = writeUsername(&buffer[i], MAX_MSG_SIZE-i, my_id)) < 0)
499:         return 0;
500:     i += ret;
501:
502:     if ((ret = writeUsername(&buffer[i], MAX_MSG_SIZE-i, other_id)) < 0)
503:         return 0;
504:     i += ret;
505:
506:     if ((ret = writeUInt32(&buffer[i], MAX_MSG_SIZE-i, ds_size)) < 0)
507:         return 0;
508:     i += ret;
509:
510:     if ((ret = writeBuf(&buffer[i], MAX_MSG_SIZE-i, ds, ds_size)) < 0)
511:         return 0;
512:     i += ret;
513:
514:     if ((ret = pkey2buf(eph_key, &buffer[i], MAX_MSG_SIZE-i)) < 0)
515:         return 0;
516:     i += ret;
517:
518:     return i;
519: }
520:
521: msglen_t ServerHelloMessage::read(char* buffer, msglen_t len){
522:     int i = 1;
523:     int ret;
524:
525:     if ((ret = readUInt32(&nonce, &buffer[i], len-i)) < 0)
526:         return 1;
527:     i += ret;
528:
529:     if ((ret = readUsername(&my_id, &buffer[i], len-i)) < 0)
530:         return 1;
531:     i += ret;
532:
533:     if ((ret = readUsername(&other_id, &buffer[i], len-i)) < 0)
534:         return 1;
535:     i += ret;
536:
537:     if ((ret = readUInt32(&ds_size, &buffer[i], len-i)) < 0)
538:         return 1;
539:     i += ret;
540:
541:     ds = (char*) malloc(ds_size);
542:     if (!ds){
543:         LOG_PERROR(LOG_ERR, "Malloc failed: %s");
544:         return 1;
545:     }
546:
547:     if ((ret = readBuf(ds, ds_size, &buffer[i], len-i)) < 0)
548:         return 1;
549:     i += ret;
550:
551:     if((ret = buf2pkey(&buffer[i], len-i, &eph_key)) < 0)
```

```
552:         return 1;
553:     i += ret;
554:
555:     return 0;
556: }
557:
558: ClientVerifyMessage::~ClientVerifyMessage() {
559:     if (ds != NULL) {
560:         free(ds);
561:     }
562: }
563:
564: msglen_t ClientVerifyMessage::write(char* buffer) {
565:     int i = 0;
566:     int ret;
567:
568:     if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) CLIENT_VERIFY)) < 0)
569:         return 0;
570:     i += ret;
571:
572:     if ((ret = writeUInt32(&buffer[i], MAX_MSG_SIZE-i, ds_size)) < 0)
573:         return 0;
574:     i += ret;
575:
576:     if ((ret = writeBuf(&buffer[i], MAX_MSG_SIZE-i, ds, ds_size)) < 0)
577:         return 0;
578:     i += ret;
579:
580:     return i;
581: }
582:
583: msglen_t ClientVerifyMessage::read(char* buffer, msglen_t len) {
584:     int i = 1;
585:     int ret;
586:
587:     if ((ret = readUInt32(&ds_size, &buffer[i], len-i)) < 0)
588:         return 1;
589:     i += ret;
590:
591:     ds = (char*) malloc(ds_size);
592:     if (!ds) {
593:         LOG_PERROR(LOG_ERR, "Malloc failed: %s");
594:         return 1;
595:     }
596:
597:     if ((ret = readBuf(ds, ds_size, &buffer[i], len-i)) < 0)
598:         return 1;
599:     i += ret;
600:
601:     return 0;
602: }
603:
604: msglen_t CertificateRequestMessage::write(char* buffer) {
605:     int i = 0;
606:     int ret;
607:
608:     if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) CERT_REQ)) < 0)
609:         return 0;
610:
611:     i += ret;
612:
613:     return i;
614: }
615:
616: msglen_t CertificateRequestMessage::read(char* buffer, msglen_t len) {
617:     return 0;
618: }
619:
620: msglen_t CertificateMessage::write(char* buffer) {
```

```
621:     int i = 0;
622:     int ret;
623:
624:     if ((ret = writeUInt8(&buffer[i], MAX_MSG_SIZE-i, (char) CERTIFICATE)) < 0)
625:         return 0;
626:
627:     i += ret;
628:
629:     if ((ret = cert2buf(cert, &buffer[i], MAX_MSG_SIZE-1)) < 0)
630:         return 0;
631:     i += ret;
632:
633:     return i;
634: }
635:
636: msglen_t CertificateMessage::read(char* buffer, msglen_t len){
637:     int ret = buf2cert(&buffer[1], len-1, &cert);
638:     return ret > 0 ? 0 : 1;
639: }
```

```
1: /**
2:  * @file socket_wrapper.h
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Definition of the helper class "SocketWrapper" and derivatives
6:  *
7:  * @date 2020-05-17
8:  */
9:
10: #ifndef SOCKET_WRAPPER_H
11: #define SOCKET_WRAPPER_H
12:
13: #include <sys/socket.h>
14: #include <netinet/in.h>
15: #include "logging.h"
16: #include "network/messages.h"
17: #include "network/host.h"
18:
19: /**
20:  * Wrapper class around sockaddr_in and socket descriptor
21:  *
22:  * It provides a more simple interface saving a lot of boiler-plate code.
23:  * There are two subclasses: ClientSocketWrapper and ServerSocketWrapper.
24:  */
25: class SocketWrapper{
26: protected:
27:     /** Other host inet socket */
28:     struct sockaddr_in other_addr;
29:
30:     /** Socket file descriptor */
31:     int socket_fd;
32:
33:     /** Pre-allocated buffer for incoming messages */
34:     char buffer_in[MAX_MSG_SIZE];
35:
36:     /** Pre-allocated buffer for outgoing messages */
37:     char buffer_out[MAX_MSG_SIZE];
38:
39:     /** Index in the buffer that has been read up to now */
40:     msglen_t buf_idx;
41: public:
42:     /**
43:      * Initialize on a new socket
44:      */
45:     SocketWrapper();
46:
47:     /**
48:      * Initialize using existing socket
49:      */
50:     SocketWrapper(int sd) : socket_fd(sd), buf_idx(0) {}
51:
52:     ~SocketWrapper(){closeSocket();}
53:
54:     /**
55:      * Returns current socket file descriptor
56:      */
57:     int getDescriptor(){return socket_fd;};
58:
59:     /**
60:      * Read any new data from the socket but does not wait for the
61:      * whole message to be ready.
62:      *
63:      * This API is blocking iff socket was not ready.
64:      *
65:      * @param size the size of the temporary buffer
66:      * @returns the received message or null if an error occurred
67:      */
68:     Message* readPartMsg();
69: }
```

```
70:    /**
71:     * Receive any new message from the socket.
72:     *
73:     * This API is blocking.
74:     *
75:     * @returns the received message or null if an error occurred
76:     */
77:    Message* receiveAnyMsg();
78:
79:    /**
80:     * Receive a new message of the given type from the socket.
81:     *
82:     * When a message of the wrong type is received it is simply ignored.
83:     *
84:     * This API is blocking.
85:     *
86:     * @param type the type to keep
87:     * @returns the received message or null if an error occurred
88:     */
89:    Message* receiveMsg(MessageType type);
90:
91:    /**
92:     * Receive a new message of any of the given types from the socket.
93:     *
94:     * When a message of the wrong type is received it is simply ignored.
95:     *
96:     * This API is blocking.
97:     *
98:     * @param type the types to keep (array)
99:     * @param n_types the number of types to keep (array length)
100:    * @returns the received message or null if an error occurred
101:    */
102:    Message* receiveMsg(MessageType type[], int n_types);
103:
104:    /**
105:     * Sends the given message to the peer host through the socket.
106:     *
107:     * @param msg the message to be sent
108:     * @returns 0 in case of success, something else otherwise
109:     */
110:    int sendMsg(Message *msg);
111:
112:    /**
113:     * Closes the socket.
114:     */
115:    void closeSocket();
116:
117:    /**
118:     * Sets the address of the other host.
119:     *
120:     * This is used when initializing a new SocketWrapper for a newly
121:     * acceptor connection.
122:     */
123:    void setOtherAddr(struct sockaddr_in addr){other_addr = addr;}
124:
125:    sockaddr_in* getOtherAddr() { return &other_addr;}
126:
127:    /**
128:     * Returns connected host.
129:     */
130:    Host getConnectedHost(){return Host(other_addr);}
131: };
132:
133: /**
134:  * SocketWrapper for a TCP client
135:  *
136:  * It provides a new function to connect to server.
137:  */
138: class ClientSocketWrapper : public SocketWrapper{
```

```
139: public:
140:     /**
141:      * Connects to a remote server.
142:      *
143:      * @returns 0 in case of success, something else otherwise
144:      */
145:     int connectServer(Host host);
146: };
147:
148: /**
149:  * SocketWrapper for a TCP server
150:  *
151:  * It provides a new function to accept clients.
152:  * Constructor also set listen mode.
153:  */
154: class ServerSocketWrapper : public SocketWrapper{
155: private:
156:     /** Local inet socket */
157:     struct sockaddr_in my_addr;
158: public:
159:     /**
160:      * Binds the socket to the requested port.
161:      *
162:      * @param port the port you want to bind on
163:      * @returns 0 in case of success
164:      * @returns 1 otherwise
165:      */
166:     int bindPort(int port);
167:
168:     /**
169:      * Binds the socket to a random port.
170:      *
171:      * @returns 0 in case of success
172:      * @returns 1 otherwise
173:      */
174:     int bindPort();
175:
176:     /**
177:      * Accepts any incoming connection and returns the related SocketWrapper.
178:      */
179:     SocketWrapper* acceptClient();
180:
181:     /**
182:      * Returns port the server is listening new connections on.
183:      */
184:     int getPort(){return ntohs(my_addr.sin_port);}
185: };
186:
187: #endif // SOCKET_WRAPPER_Hln
```

```
1:  /**
2:  * @file socket_wrapper.cpp
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Implementation of socket_wrapper.h
6:  *
7:  * @see socket_wrapper.h
8:  */
9:
10: #include <assert.h>
11: #include "logging.h"
12: #include "network/socket_wrapper.h"
13: #include "utils/dump_buffer.h"
14: #include "network/inet_utils.h"
15:
16: SocketWrapper::SocketWrapper() {
17:     socket_fd = socket(AF_INET, SOCK_STREAM, 0);
18:     if (socket_fd < 0){
19:         LOG_PERROR(LOG_ERR, "Error creating socket: %s");
20:         return;
21:     }
22:     buf_idx = 0;
23: }
24: Message* SocketWrapper::readPartMsg(){
25:     int len;
26:     msglen_t msglen = 0;
27:
28:     if (buf_idx < sizeof(msglen)){ // I first need to read msglen
29:         // read available message
30:         len = read(socket_fd, buffer_in+buf_idx, sizeof(msglen)-buf_idx);
31:         // I will read rest of it in another moment since I do not know
32:         // whether other data is available.
33:         // TODO: find a way to tell whether socket has other data
34:     } else{
35:         // read msg length
36:         msglen = MSGLEN_NTOH(*(msglen_t*)buffer_in);
37:         assert(msglen <= MAX_MSG_SIZE); // must not happen
38:
39:         // read up to msg length
40:         len = read(socket_fd, buffer_in+buf_idx, msglen-buf_idx);
41:     }
42:
43:     DUMP_BUFFER_HEX_DEBUG(buffer_in, len);
44:
45:     if (len < 0){
46:         LOG_PERROR(LOG_ERR, "Error reading from socket: %s");
47:         throw "Error reading from socket";
48:     } else if (len == 0){
49:         throw "Connection lost";
50:     }
51:
52:     // buf_idx is also the number of read bytes up to now
53:     buf_idx += len;
54:
55:     if (buf_idx < sizeof(msglen)){
56:         LOG(LOG_DEBUG, "Too few bytes received from socket: %d < %lu",
57:             buf_idx, sizeof(msglen));
58:         return NULL;
59:     }
60:
61:     // read msg length
62:     msglen = MSGLEN_NTOH(*(msglen_t*)buffer_in);
63:
64:     if (msglen > MAX_MSG_SIZE){
65:         throw("Message is too big");
66:     }
67:
68:     if (buf_idx != msglen){
69:         LOG(LOG_DEBUG, "Too few bytes received from socket: %d < %d",
```



```
70:         buf_idx, msglen);
71:         return NULL;
72:     }
73:
74:     Message *m = readMessage(buffer_in+sizeof(msglen), msglen-sizeof(msglen));
75:
76:     // reset buffer
77:     buf_idx = 0;
78:
79:     return m;
80: }
81:
82: Message* SocketWrapper::receiveAnyMsg(){
83:     int len;
84:     msglen_t msglen;
85:
86:     // read msg length
87:     len = recv(socket_fd, buffer_in, sizeof(msglen), MSG_WAITALL);
88:
89:     DUMP_BUFFER_HEX_DEBUG(buffer_in, len);
90:
91:     if (len == 0){
92:         throw "Connection lost";
93:     } else if (len != sizeof(msglen)){
94:         LOG(LOG_ERR, "Too few bytes received from socket: %d < %lu",
95:             len, sizeof(msglen));
96:         return NULL;
97:     }
98:
99:     // read msg payload
100:    msglen = MSGLEN_NTOH*((msglen_t*)buffer_in);
101:
102:    if (msglen > MAX_MSG_SIZE){
103:        throw("Message is too big");
104:    }
105:
106:    len += recv(socket_fd, buffer_in+len, msglen-len, MSG_WAITALL);
107:
108:    DUMP_BUFFER_HEX_DEBUG(buffer_in, len);
109:
110:    if (len == 0){
111:        throw "Received EOF";
112:    } else if (len != msglen){
113:        LOG(LOG_ERR, "Too few bytes received from socket: %d < %d",
114:            len, msglen);
115:        return NULL;
116:    }
117:
118:    Message *m = readMessage(buffer_in+sizeof(msglen), msglen-sizeof(msglen));
119:
120:    return m;
121: }
122:
123: Message* SocketWrapper::receiveMsg(MessageType type){
124:     return this->receiveMsg(&type, 1);
125: }
126:
127: Message* SocketWrapper::receiveMsg(MessageType type[], int n_types){
128:     Message *m = NULL;
129:     while (m == NULL){
130:         try{
131:             m = receiveAnyMsg();
132:         } catch(const char* msg){
133:             LOG(LOG_ERR, "%s", msg);
134:             return NULL;
135:         }
136:         if (m != NULL){
137:             for (int i = 0; i < n_types; i++){
138:                 if (m->getType() == type[i]){
```

```
139:         return m;
140:     }
141: }
142:     LOG(LOG_WARN, "Received unexpected message of type %s", m->getName().c_str())
;
143: }
144: }
145: //TODO: add timeout?
146: return NULL;
147: }
148:
149:
150: int SocketWrapper::sendMsg(Message *msg){
151:     msglen_t msglen, pktlen;
152:     int len;
153:
154:     msglen = msg->write(buffer_out+sizeof(msglen));
155:     if (msglen == 0)
156:         return 1;
157:
158:     pktlen = msglen + sizeof(msglen);
159:     *((msglen_t*)buffer_out) = MSGLEN_HTON(pktlen);
160:
161:     LOG(LOG_DEBUG, "Sending %s", msg->getName().c_str());
162:
163:     DUMP_BUFFER_HEX_DEBUG(buffer_out, pktlen);
164:
165:     len = send(socket_fd, buffer_out, pktlen, 0);
166:     if (len != pktlen){
167:         LOG(LOG_ERR, "Error sending %s: len (%d) != msglen (%d)",
168:             msg->getName().c_str(),
169:             len,
170:             msglen
171:         );
172:         return 1;
173:     }
174:
175:     LOG(LOG_DEBUG, "Sent message %s", msg->getName().c_str());
176:
177:     return 0;
178: }
179:
180: void SocketWrapper::closeSocket(){
181:     close(socket_fd);
182: }
183:
184: int ClientSocketWrapper::connectServer(Host host){
185:     int ret;
186:
187:     other_addr = host.getAddress();
188:
189:     ret = connect(
190:         socket_fd,
191:         (struct sockaddr*) &other_addr,
192:         sizeof(other_addr)
193:     );
194:
195:     if (ret != 0){
196:         LOG_PERROR(LOG_ERR, "Error connecting to %s: %s",
197:             sockaddr_in_to_string(host.getAddress()).c_str());
198:         return ret;
199:     }
200:
201:     return ret;
202: }
203:
204: int ServerSocketWrapper::bindPort(){
205:     my_addr = make_my_sockaddr_in(0);
206:     int ret = bind_random_port(socket_fd, &my_addr);
```

```
207:     if (ret <= 0){
208:         LOG_PERROR(LOG_ERR, "Error in binding: %s");
209:         return ret;
210:     }
211:
212:     ret = listen(socket_fd, 10);
213:     if (ret != 0){
214:         LOG_PERROR(LOG_ERR, "Error in setting socket to listen mode: %s");
215:     }
216:
217:     return ret;
218: }
219:
220: int ServerSocketWrapper::bindPort(int port){
221:     my_addr = make_my_sockaddr_in(port);
222:     int ret = bind(socket_fd, (struct sockaddr*) &my_addr, sizeof(my_addr));
223:     if (ret != 0){
224:         LOG_PERROR(LOG_ERR, "Error in binding: %s");
225:         return ret;
226:     }
227:
228:     ret = listen(socket_fd, 10);
229:     if (ret != 0){
230:         LOG_PERROR(LOG_ERR, "Error in setting socket to listen mode: %s");
231:     }
232:
233:     return ret;
234: }
235:
236: SocketWrapper* ServerSocketWrapper::acceptClient(){
237:     socklen_t len = sizeof(other_addr);
238:     int new_sd = accept(
239:         socket_fd,
240:         (struct sockaddr*) &other_addr,
241:         &len
242:     );
243:
244:     SocketWrapper *sw = new SocketWrapper(new_sd);
245:     sw->setOtherAddr(other_addr);
246:     return sw;
247: }
```

```
1: /**
2:  * @file secure_socket_wrapper.h
3:  * @author Mirko Laruina
4:  *
5:  * @brief Header file for SecureSocketWrapper
6:  *
7:  * @date 2020-06-09
8:  */
9:
10: #ifndef SECURE_SOCKET_WRAPPER_H
11: #define SECURE_SOCKET_WRAPPER_H
12:
13: #include <sys/socket.h>
14: #include <netinet/in.h>
15: #include "logging.h"
16: #include "network/messages.h"
17: #include "network/socket_wrapper.h"
18: #include "security/crypto.h"
19: #include "security/crypto_utils.h"
20: #include "security/secure_host.h"
21: #include "utils/dump_buffer.h"
22:
23: #define MAX_MSG_TO_SIGN_SIZE (2*MAX_USERNAME_LENGTH + 2 * sizeof(nonce_t) + 2 * KEY_BIO_MA
X_SIZE )
24: #define MAX_SEC_MSG_SIZE (MAX_MSG_SIZE - TAG_SIZE - sizeof(msglen_t) - 1)
25:
26: #define AAD_SIZE (sizeof(msglen_t) + 1)
27:
28: class SecureSocketWrapper
29: {
30: protected:
31:     SocketWrapper *sw;
32:
33:     char send_key[KEY_SIZE];
34:     char recv_key[KEY_SIZE];
35:     char send_iv_static[IV_SIZE];
36:     char recv_iv_static[IV_SIZE];
37:     char send_iv[IV_SIZE];
38:     char recv_iv[IV_SIZE];
39:     uint64_t send_seq_num;
40:     uint64_t recv_seq_num;
41:     string my_id;
42:     string other_id;
43:     nonce_t sv_nonce;
44:     nonce_t cl_nonce;
45:     EVP_PKEY *my_eph_key;
46:     EVP_PKEY *other_eph_key;
47:     X509 *my_cert;
48:     X509 *other_cert;
49:     X509_STORE *store;
50:     EVP_PKEY *my_priv_key;
51:
52:     bool peer_authenticated;
53:
54:     char msg_to_sign_buf[MAX_MSG_TO_SIGN_SIZE];
55:
56:     /**
57:      * Empty constructor to use in child classes.
58:      */
59:     SecureSocketWrapper(){};
60:
61:     /**
62:      * @brief Derives the key
63:      *
64:      * @param role          Role in the communication
65:      */
66:     void generateKeys(const char *role);
67:
68:     /**
```

```
69:      * @brief Calculates the IV to use when sending the next message
70:      *
71:      */
72:      void updateSendIV();
73:
74:      /**
75:      * @brief Calculates the IV to use when receiving a new message
76:      *
77:      */
78:      void updateRecvIV();
79:
80:      /** Internal initialization */
81:      void init(X509 *cert, EVP_PKEY *my_priv_key, X509_STORE *store);
82:
83:      /**
84:      * @brief Decrypts a Secure Message into a Message
85:      *
86:      * @param sm          Secure message ptr
87:      * @return Message*   Read message
88:      */
89:      Message *decryptMsg(SecureMessage *sm);
90:
91:      /**
92:      * @brief Encrypts a Message into a SecureMessage
93:      *
94:      * @param m          Message to encrypt
95:      * @return SecureMessage*   Encrypted SecureMessage
96:      */
97:      SecureMessage *encryptMsg(Message *m);
98:
99:      /**
100:      * Make the signature for the handshake protocol
101:      */
102:      int makeSignature(const char *role, char** ds);
103:
104:      /**
105:      * Checks the signature for the handshake protocol
106:      */
107:      bool checkSignature(char *ds, size_t ds_size, const char *role);
108:
109:      /**
110:      * Builds the message to be signed.
111:      *
112:      * @param role the role of this peer
113:      * @param msg the buffer to write the message to
114:      * @returns number of written bytes
115:      */
116:      int buildMsgToSign(const char *role, char *msg);
117:
118:      /**
119:      * Builds the aad of a message.
120:      *
121:      * I.e. this function builds the message header as SocketWrapper would.
122:      *
123:      * @param msg_type the type of the message
124:      * @param len the length of the message
125:      * @param aad the aad buffer to write to (it must be AAD_SIZE long)
126:      * @returns number of written bytes
127:      *
128:      * @see AAD_SIZE
129:      */
130:      void makeAAD(MessageType msg_type, msglen_t len, char* aad);
131:
132: public:
133:      /**
134:      * Initialize on a new socket
135:      */
136:      SecureSocketWrapper(X509 *cert, EVP_PKEY *my_priv_key, X509_STORE *store);
137:
```

```
138:    /**
139:     * Initialize using existing socket
140:     */
141:    SecureSocketWrapper(X509 *cert, EVP_PKEY *my_priv_key, X509_STORE *store, int sd);
142:
143:    /**
144:     * Constructor to generate connection socket wrappers
145:     */
146:    SecureSocketWrapper(X509 *cert, EVP_PKEY *my_priv_key, X509_STORE *store, SocketWrapper
r *sw);
147:
148:    /**
149:     * Destructor
150:     */
151:    ~SecureSocketWrapper();
152:
153:    /**
154:     * Read any new data from the socket but does not wait for the
155:     * whole message to be ready. This does not decrypt the message!
156:     *
157:     * This API is blocking iff socket was not ready.
158:     *
159:     * @param size the size of the temporary buffer
160:     * @returns the received message or null if an error occurred
161:     */
162:    Message *readPartMsg();
163:
164:    /**
165:     * Receive any new message from the socket.
166:     *
167:     * This API is blocking.
168:     *
169:     * @returns the received message or null if an error occurred
170:     */
171:    Message *receiveAnyMsg();
172:
173:    /**
174:     * Receive a new message of the given type from the socket.
175:     *
176:     * When a message of the wrong type is received it is simply ignored.
177:     *
178:     * This API is blocking.
179:     *
180:     * @param type the type to keep
181:     * @returns the received message or null if an error occurred
182:     */
183:    Message *receiveMsg(MessageType type);
184:
185:    /**
186:     * Receive a new message of any of the given types from the socket.
187:     *
188:     * When a message of the wrong type is received it is simply ignored.
189:     *
190:     * This API is blocking.
191:     *
192:     * @param type the types to keep (array)
193:     * @param n_types the number of types to keep (array length)
194:     * @returns the received message or null if an error occurred
195:     */
196:    Message *receiveMsg(MessageType type[], int n_types);
197:
198:    Message *handleMsg(Message *msg);
199:
200:
201:    int sendCertRequest();
202:
203:    int handleCertResponse(CertificateMessage* cm);
204:
205:    int handleClientHello(ClientHelloMessage *chm);
```

```
206:     int handleServerHello(ServerHelloMessage *shm);
207:     int handleClientVerify(ClientVerifyMessage *cvm);
208:
209:     int sendClientHello();
210:     int sendServerHello();
211:     int sendClientVerify();
212:
213:     int sendPlain(Message *msg);
214:     /**
215:      * Sends the given message to the peer host through the socket.
216:      *
217:      * @param msg the message to be sent
218:      * @returns 0 in case of success, something else otherwise
219:      */
220:     int sendMsg(Message *msg);
221:
222:     /**
223:      * @brief Establishes a secure connection over the already specified socket. To be run
server-side.
224:      *
225:      * @return int 0 in case of success, something else otherwise
226:      */
227:     int handshakeServer();
228:
229:     /**
230:      * @brief Establishes a secure connection over the already specified socket. To be run
client-side.
231:      *
232:      * @return int 0 in case of success, something else otherwise
233:      */
234:     int handshakeClient();
235:
236:     /**
237:      * Sets the peer certificate.
238:      */
239:     bool setOtherCert(X509 *other_cert);
240:
241:     /**
242:      * Returns current socket file descriptor
243:      */
244:     int getDescriptor() { return sw->getDescriptor(); };
245:
246:     /**
247:      * Closes the socket.
248:      */
249:     void closeSocket() { sw->closeSocket(); }
250:
251:     /**
252:      * Sets the address of the other host.
253:      *
254:      * This is used when initializing a new SocketWrapper for a newly
255:      * acceptor connection.
256:      */
257:     void setOtherAddr(struct sockaddr_in addr) { sw->setOtherAddr(addr); }
258:
259:     sockaddr_in *getOtherAddr() { return sw->getOtherAddr(); }
260:
261:     /**
262:      * Returns connected host.
263:      */
264:     SecureHost getConnectedHost() { return SecureHost(*getOtherAddr(), other_cert); }
265:
266:     /**
267:      * Returns the certificate of this host
268:      */
269:     X509* getCert(){ return my_cert;}
270: };
271:
272: /**
```

```
273:  * SocketWrapper for a TCP client
274:  *
275:  * It provides a new function to connect to server.
276:  */
277: class ClientSecureSocketWrapper : public SecureSocketWrapper
278: {
279: private:
280:     ClientSocketWrapper *csw;
281:
282: public:
283:     /**
284:      * Initialize a new socket on a random port.
285:      *
286:      * @param port the port you want to bind on
287:      */
288:     ClientSecureSocketWrapper(X509 *cert, EVP_PKEY *my_priv_key, X509_STORE *store);
289:
290:     /**
291:      * Connects to a remote server.
292:      *
293:      * @returns 0 in case of success, something else otherwise
294:      */
295:     int connectServer(SecureHost host);
296:
297: };
298:
299: /**
300:  * SocketWrapper for a TCP server
301:  *
302:  * It provides a new function to accept clients.
303:  * Constructor also set listen mode.
304:  */
305: class ServerSecureSocketWrapper : public SecureSocketWrapper
306: {
307: private:
308:     ServerSocketWrapper *ssw;
309:
310: public:
311:     /**
312:      * Initialize a new socket on a random port.
313:      *
314:      * @param port the port you want to bind on
315:      */
316:     ServerSecureSocketWrapper(X509 *cert, EVP_PKEY *my_priv_key, X509_STORE *store);
317:
318:     /**
319:      * Binds the socket to the requested port.
320:      *
321:      * @param port the port you want to bind on
322:      * @returns 0 in case of success
323:      * @returns 1 otherwise
324:      */
325:     int bindPort(int port) { return ssw->bindPort(port); }
326:
327:     /**
328:      * Binds the socket to a random port.
329:      *
330:      * @returns 0 in case of success
331:      * @returns 1 otherwise
332:      */
333:     int bindPort() { return ssw->bindPort(); }
334:
335:     /**
336:      * Accepts any incoming connection and returns the related SocketWrapper.
337:      */
338:     SecureSocketWrapper *acceptClient();
339:
340:     /**
341:      * Accepts any incoming connection and returns the related SocketWrapper.
```



```
342:      *
343:      * The certificate is set as the expected certificate of the peer.
344:      */
345:      SecureSocketWrapper *acceptClient(X509 *other_cert);
346:
347:      /**
348:      * Returns port the server is listening new connections on.
349:      */
350:      int getPort() { return ssw->getPort(); }
351: };
352:
353: #endif
```

```
1: #include "security/secure_socket_wrapper.h"
2: #include "security/crypto_utils.h"
3:
4: SecureSocketWrapper::SecureSocketWrapper(X509* cert, EVP_PKEY* my_priv_key, X509_STORE* store)
5: {
6:     sw = new SocketWrapper();
7:     init(cert, my_priv_key, store);
8: }
9:
10: SecureSocketWrapper::SecureSocketWrapper(X509* cert, EVP_PKEY* my_priv_key, X509_STORE* store, int sd)
11: {
12:     sw = new SocketWrapper(sd);
13:     init(cert, my_priv_key, store);
14: }
15:
16: SecureSocketWrapper::SecureSocketWrapper(X509* cert, EVP_PKEY* my_priv_key, X509_STORE* store, SocketWrapper *sw)
17: {
18:     this->sw = sw;
19:     init(cert, my_priv_key, store);
20: }
21:
22: void SecureSocketWrapper::init(X509* cert, EVP_PKEY* my_priv_key, X509_STORE* store){
23:     send_seq_num = 0;
24:     rcv_seq_num = 0;
25:     cl_nonce = 0;
26:     sv_nonce = 0;
27:     my_eph_key = NULL;
28:     other_eph_key = NULL;
29:     this->my_cert = cert;
30:     this->my_priv_key = my_priv_key;
31:     this->store = store;
32:     other_cert = NULL;
33:     peer_authenticated = false;
34:     my_id = usernameFromCert(cert);
35: }
36:
37: SecureSocketWrapper::~SecureSocketWrapper(){
38:     delete sw;
39:
40:     // TODO do not wait this much time to delete them
41:     if (my_eph_key != NULL){
42:         EVP_PKEY_free(my_eph_key);
43:     }
44:     if (other_eph_key != NULL){
45:         EVP_PKEY_free(other_eph_key);
46:     }
47:     // TODO free certs too?
48: }
49:
50: Message *SecureSocketWrapper::decryptMsg(SecureMessage *sm)
51: {
52:     if (!peer_authenticated){
53:         LOG(LOG_WARN, "Unauthenticated peer sent encrypted message");
54:         return NULL;
55:     }
56:     int ret;
57:
58:     msglen_t pt_len = sm->getCtSize();
59:     LOG(LOG_DEBUG, "Received SecureMessage of size %d", pt_len);
60:     if (pt_len > MAX_SEC_MSG_SIZE){
61:         LOG(LOG_ERR, "Message is too big");
62:         return NULL;
63:     }
64:
65:     LOG(LOG_DEBUG, "Payload");
66:     DUMP_BUFFER_HEX_DEBUG(sm->getCt(), pt_len);
```

```
67:     LOG(LOG_DEBUG, "TAG");
68:     DUMP_BUFFER_HEX_DEBUG(sm->getTag(), TAG_SIZE);
69:
70:     char *buffer_pt = (char*) malloc(pt_len);
71:     char buffer_aad[AAD_SIZE];
72:
73:     if (!buffer_pt){
74:         LOG_PERROR(LOG_ERR, "Malloc failed: %s");
75:         return NULL;
76:     }
77:
78:     makeAAD(SECURE_MESSAGE, pt_len+TAG_SIZE+AAD_SIZE, buffer_aad);
79:     DUMP_BUFFER_HEX_DEBUG(buffer_aad, AAD_SIZE);
80:
81:     updateRecvIV();
82:
83:     try{
84:         ret = aes_gcm_decrypt(sm->getCt(), pt_len, buffer_aad, AAD_SIZE,
85:                               recv_key, recv_iv,
86:                               buffer_pt, sm->getTag());
87:     } catch (const char *err_msg){
88:         LOG(LOG_ERR, "Error: %s", err_msg);
89:         free(buffer_pt);
90:         return NULL;
91:     }
92:
93:     if (ret <= 0)
94:     {
95:         LOG(LOG_ERR, "Could not decrypt the message");
96:         return NULL;
97:     }
98:
99:     LOG(LOG_DEBUG, "Decrypted message (%d):", ret);
100:    DUMP_BUFFER_HEX_DEBUG(buffer_pt, ret);
101:
102:    Message *m = readMessage(buffer_pt, pt_len);
103:
104:    free(buffer_pt);
105:
106:    if (m != NULL){
107:        LOG(LOG_INFO, "Decrypted message of type %s", m->getName().c_str());
108:    } else{
109:        LOG(LOG_WARN, "Malformed message");
110:    }
111:    return m;
112: }
113:
114: SecureMessage *SecureSocketWrapper::encryptMsg(Message *m)
115: {
116:     if (!peer_authenticated)
117:         return NULL;
118:     int ret;
119:
120:     char buffer_pt[MAX_MSG_SIZE];
121:     msglen_t buf_len = m->write(buffer_pt);
122:
123:     if (buf_len > MAX_SEC_MSG_SIZE){
124:         LOG(LOG_ERR, "Message is too big: %s", m->getName().c_str());
125:         return NULL;
126:     }
127:
128:     char* buffer_ct = (char*) malloc(MAX_MSG_SIZE);
129:     char* buffer_tag = (char*) malloc(TAG_SIZE);
130:     char buffer_aad[AAD_SIZE];
131:
132:     if (!buffer_ct || !buffer_tag){
133:         LOG_PERROR(LOG_ERR, "Malloc failed: %s");
134:         return NULL;
135:     }
```

```
136:
137:     LOG(LOG_DEBUG, "Encrypting message of size %d", buf_len);
138:     DUMP_BUFFER_HEX_DEBUG(buffer_pt, IV_SIZE);
139:
140:     updateSendIV();
141:     makeAAD(SECURE_MESSAGE, buf_len+TAG_SIZE+AAD_SIZE, buffer_aad);
142:     DUMP_BUFFER_HEX_DEBUG(buffer_aad, AAD_SIZE);
143:     try{
144:         ret = aes_gcm_encrypt(buffer_pt, buf_len, buffer_aad, AAD_SIZE,
145:                               send_key, send_iv,
146:                               buffer_ct, buffer_tag);
147:     } catch(const char* err_msg){
148:         LOG(LOG_ERR, "Error: %s", err_msg);
149:         free(buffer_ct);
150:         free(buffer_tag);
151:         return NULL;
152:     }
153:
154:     LOG(LOG_DEBUG, "Message encrypted %d bytes with iv: ", ret);
155:     DUMP_BUFFER_HEX_DEBUG(send_iv, IV_SIZE);
156:     LOG(LOG_DEBUG, "and tag: ");
157:     DUMP_BUFFER_HEX_DEBUG(buffer_tag, TAG_SIZE);
158:     LOG(LOG_DEBUG, "SecureMessage of size %d", buf_len + 1 + TAG_SIZE);
159:
160:     if (ret <= 0)
161:     {
162:         LOG(LOG_ERR, "Could not encrypt the message");
163:         return NULL;
164:     }
165:
166:     SecureMessage *sm = new SecureMessage(buffer_ct, ret, buffer_tag);
167:     return sm;
168: }
169:
170: void SecureSocketWrapper::makeAAD(MessageType msg_type, msglen_t len, char* aad){
171:     *((msglen_t*)aad) = MSGLEN_HTON(len);
172:     aad[2] = msg_type;
173: }
174:
175: Message *SecureSocketWrapper::readPartMsg()
176: {
177:     return sw->readPartMsg();
178: }
179:
180: Message *SecureSocketWrapper::receiveAnyMsg()
181: {
182:     Message *m = sw->receiveAnyMsg();
183:
184:     return handleMsg(m);
185: }
186:
187: Message *SecureSocketWrapper::handleMsg(Message* msg)
188: {
189:     Message* dm;
190:     if (msg == NULL)
191:         return NULL;
192:     switch(msg->getType()){
193:         case SECURE_MESSAGE:
194:             dm = decryptMsg((SecureMessage*) msg);
195:             if (dm != NULL)
196:                 recv_seq_num++;
197:             delete msg;
198:             return dm;
199:         case CLIENT_HELLO:
200:             if (cl_nonce == 0 && !peer_authenticated){
201:                 return msg;
202:             } else{
203:                 // already received
204:                 LOG(LOG_WARN, "Client sent CLIENT_HELLO twice");
```

```
205:         return NULL;
206:     }
207:     case SERVER_HELLO:
208:         if (!peer_authenticated){
209:             return msg;
210:         } else{
211:             // already authenticated!
212:             LOG(LOG_WARN, "Server sent SERVER_HELLO twice");
213:             return NULL;
214:         }
215:     case CLIENT_VERIFY:
216:         if (!peer_authenticated){
217:             return msg;
218:         } else{
219:             // already authenticated
220:             LOG(LOG_WARN, "Client sent CLIENT_VERIFY twice");
221:             return NULL;
222:         }
223:     case CERTIFICATE:
224:     case CERT_REQ:
225:         return msg;
226:     default:
227:         LOG(LOG_WARN, "Peer sent %s in cleartext!", msg->getName().c_str());
228:         return NULL;
229: }
230: }
231:
232: int SecureSocketWrapper::handleClientHello(ClientHelloMessage* chm)
233: {
234:     cl_nonce = chm->getNonce();
235:     other_eph_key = chm->getEphKey();
236:     return sendServerHello();
237: }
238:
239: int SecureSocketWrapper::handleServerHello(ServerHelloMessage* shm)
240: {
241:     sv_nonce = shm->getNonce();
242:     other_eph_key = shm->getEphKey();
243:
244:     //Deriving the symmetric key
245:     generateKeys("client");
246:
247:     bool check = checkSignature(shm->getDs(), shm->getDsSize(), "client");
248:     if (!check){
249:         LOG(LOG_ERR, "Digital Signature verification failure!");
250:         return -1;
251:     }
252:
253:     peer_authenticated = true;
254:
255:     return sendClientVerify();
256: }
257:
258: int SecureSocketWrapper::handleClientVerify(ClientVerifyMessage* cvm)
259: {
260:     bool check = checkSignature(cvm->getDs(), cvm->getDsSize(), "server");
261:     if (!check){
262:         LOG(LOG_ERR, "Digital Signature verification failure!");
263:         return -1;
264:     }
265:
266:     LOG(LOG_INFO, "Digital Signature verification succeeded!");
267:
268:     peer_authenticated = true;
269:
270:     return 0;
271: }
272:
273: int SecureSocketWrapper::sendPlain(Message *msg)
```

```
274: {
275:     int ret = sw->sendMsg(msg);
276:     if(ret == 0){
277:         return 0;
278:     }
279:     return 1;
280: }
281:
282: int SecureSocketWrapper::sendMsg(Message *msg)
283: {
284:     SecureMessage *sm = encryptMsg(msg);
285:     if (!sm)
286:     {
287:         return 1;
288:     }
289:     int ret = sw->sendMsg(sm);
290:     delete sm;
291:     if (ret == 0){
292:         send_seq_num++;
293:         return 0;
294:     } else{
295:         return 1;
296:     }
297: }
298:
299: int SecureSocketWrapper::sendCertRequest(){
300:     CertificateRequestMessage crm;
301:     return sw->sendMsg(&crm);
302: }
303:
304: int SecureSocketWrapper::sendClientHello(){
305:     cl_nonce = get_rand();
306:     get_ecdh_key(&my_eph_key);
307:
308:     ClientHelloMessage chm(my_eph_key, cl_nonce, my_id, other_id);
309:     return sw->sendMsg(&chm);
310: }
311:
312: int SecureSocketWrapper::sendServerHello(){
313:     sv_nonce = get_rand();
314:     get_ecdh_key(&my_eph_key);
315:
316:     //Deriving the symmetric key
317:     generateKeys("server");
318:
319:     char *ds = NULL;
320:     int ret = makeSignature("server", &ds);
321:     if (ret > 0){
322:         ServerHelloMessage shm(my_eph_key, sv_nonce, my_id, other_id, ds, ret);
323:         return sw->sendMsg(&shm);
324:     } else {
325:         return ret;
326:     }
327: }
328:
329: int SecureSocketWrapper::sendClientVerify(){
330:     char *ds = NULL;
331:     int ret = makeSignature("client", &ds);
332:     if (ret > 0){
333:         ClientVerifyMessage cvm(ds, ret);
334:         return sw->sendMsg(&cvm);
335:     } else{
336:         return ret;
337:     }
338: }
339:
340: void SecureSocketWrapper::generateKeys(const char* role){
341:     char *shared_secret = NULL;
342: }
```

```
343:     int size = dhke(my_eph_key, other_eph_key, &shared_secret);
344:
345:     const char* other_role;
346:     if (strcmp(role, "client") == 0){
347:         other_role = "server";
348:     } else if (strcmp(role, "server") == 0){
349:         other_role = "client";
350:     } else{
351:         LOG(LOG_ERR, "Wrong role %s", role);
352:     }
353:     char my_key_str[11] = "key_";
354:     char other_key_str[11] = "key_";
355:     char my_iv_str[11] = "iv_";
356:     char other_iv_str[11] = "iv_";
357:     strcat(my_key_str, role);
358:     strcat(other_key_str, other_role);
359:     strcat(my_iv_str, role);
360:     strcat(other_iv_str, other_role);
361:
362:     hkdf(shared_secret, size, sv_nonce, cl_nonce, my_key_str, send_key, KEY_SIZE);
363:     hkdf(shared_secret, size, sv_nonce, cl_nonce, other_key_str, recv_key, KEY_SIZE);
364:     hkdf(shared_secret, size, sv_nonce, cl_nonce, my_iv_str, send_iv_static, IV_SIZE);
365:     hkdf(shared_secret, size, sv_nonce, cl_nonce, other_iv_str, recv_iv_static, IV_SIZE);
366:
367:     LOG(LOG_DEBUG, "HKDF parameters BEGIN -----");
368:     LOG(LOG_DEBUG, "Shared secret:");
369:     DUMP_BUFFER_HEX_DEBUG(shared_secret, size);
370:     LOG(LOG_DEBUG, "sv_nonce=%d", sv_nonce);
371:     LOG(LOG_DEBUG, "cl_nonce=%d", cl_nonce);
372:     LOG(LOG_DEBUG, "HKDF parameters END -----");
373:     LOG(LOG_DEBUG, "Generated keys BEGIN -----");
374:     LOG(LOG_DEBUG, "Send key (%s):", my_key_str);
375:     DUMP_BUFFER_HEX_DEBUG(send_key, KEY_SIZE);
376:     LOG(LOG_DEBUG, "Send IV (%s):", my_iv_str);
377:     DUMP_BUFFER_HEX_DEBUG(send_iv_static, IV_SIZE);
378:     LOG(LOG_DEBUG, "Recv key (%s):", other_key_str);
379:     DUMP_BUFFER_HEX_DEBUG(recv_key, KEY_SIZE);
380:     LOG(LOG_DEBUG, "Recv IV (%s):", other_iv_str);
381:     DUMP_BUFFER_HEX_DEBUG(recv_iv_static, IV_SIZE);
382:     LOG(LOG_DEBUG, "Generated keys END -----");
383:
384:     free(shared_secret);
385: }
386:
387: int SecureSocketWrapper::buildMsgToSign(const char* role, char* msg){
388:     int i = 0;
389:     size_t size;
390:     string A;
391:     string B;
392:     EVP_PKEY *A_eph_key;
393:     EVP_PKEY *B_eph_key;
394:     if (strcmp(role, "server") == 0){
395:         A = other_id;
396:         A_eph_key = other_eph_key;
397:         B = my_id;
398:         B_eph_key = my_eph_key;
399:     } else if (strcmp(role, "client") == 0){
400:         A = my_id;
401:         A_eph_key = my_eph_key;
402:         B = other_id;
403:         B_eph_key = other_eph_key;
404:     } else{
405:         LOG(LOG_ERR, "Unrecognized role %s", role);
406:         return -1;
407:     }
408:
409:     size = min((int)A.size(), MAX_USERNAME_LENGTH);
410:     strncpy(&msg[i],
411:         A.c_str(),
```

```
412:         size);
413:     i += size;
414:
415:     size = min((int)B.size(),MAX_USERNAME_LENGTH);
416:     strncpy(&msg[i],
417:         B.c_str(),
418:         size);
419:     i += size;
420:
421:     size = sizeof(nonce_t);
422:     memcpy(&msg[i], &cl_nonce, size);
423:     i += size;
424:
425:     size = sizeof(nonce_t);
426:     memcpy(&msg[i], &sv_nonce, size);
427:     i += size;
428:
429:     size = pkey2buf(A_eph_key, &msg[i], MAX_MSG_TO_SIGN_SIZE-i);
430:     if (size <= 0){
431:         LOG(LOG_ERR, "Error copying key to buffer");
432:         return 0;
433:     }
434:     i += size;
435:
436:     size = pkey2buf(B_eph_key, &msg[i], MAX_MSG_TO_SIGN_SIZE-i);
437:     if (size <= 0){
438:         LOG(LOG_ERR, "Error copying key to buffer");
439:         return 0;
440:     }
441:     i += size;
442:
443:     return i;
444: }
445:
446: int SecureSocketWrapper::makeSignature(const char *role, char** ds){
447:     size_t msglen = buildMsgToSign(role, msg_to_sign_buf);
448:
449:     if (msglen <= 0){
450:         LOG(LOG_ERR, "Error building message to sign!");
451:         return -1;
452:     }
453:
454:     return dsa_sign(msg_to_sign_buf, msglen, ds, my_priv_key);
455: }
456:
457: bool SecureSocketWrapper::checkSignature(char* ds, size_t ds_size, const char* role){
458:     size_t msglen = buildMsgToSign(role, msg_to_sign_buf);
459:
460:     if (msglen <= 0){
461:         LOG(LOG_ERR, "Error building message to sign!");
462:         return false;
463:     }
464:
465:     bool ret = dsa_verify(msg_to_sign_buf, msglen, ds, ds_size,
466:         X509_get_pubkey(other_cert));
467:
468:     return ret;
469: }
470:
471: void updateIV(uint64_t seq, char* iv_static, char* iv){
472:     char* seq_bytes = (char*) &seq;
473:     for (size_t i=0; i<IV_SIZE; i++){
474:         if (i<sizeof(seq)){
475:             iv[i] = iv_static[i] ^ seq_bytes[i];
476:         } else{
477:             iv[i] = iv_static[i];
478:         }
479:     }
480: }
```



```
481:
482: void SecureSocketWrapper::updateSendIV(){
483:     updateIV(send_seq_num, send_iv_static, send_iv);
484: }
485:
486: void SecureSocketWrapper::updateRecvIV(){
487:     updateIV(recv_seq_num, recv_iv_static, recv_iv);
488: }
489:
490: int SecureSocketWrapper::handshakeClient(){
491:     if(sendClientHello() != 0){
492:         LOG(LOG_ERR, "ClientHello send failed!");
493:         return 1;
494:     }
495:     LOG(LOG_INFO, "Client Hello sent");
496:     ServerHelloMessage *shm = dynamic_cast<ServerHelloMessage*>(receiveMsg(SERVER_HELLO));
497:     if (shm == NULL || handleServerHello(shm) != 0){
498:         LOG(LOG_ERR, "Error handling ServerHello!");
499:         return 1;
500:     } else{
501:         LOG(LOG_INFO, "Server Hello handled");
502:         return 0;
503:     }
504: }
505:
506: int SecureSocketWrapper::handshakeServer(){
507:     ClientHelloMessage *chm = dynamic_cast<ClientHelloMessage*>(receiveMsg(CLIENT_HELLO));
508:
509:     if (chm == NULL || handleClientHello(chm) != 0){
510:         LOG(LOG_ERR, "Error handling ClientHello!");
511:         return 1;
512:     }
513:
514:     ClientVerifyMessage *cvm = dynamic_cast<ClientVerifyMessage*>(receiveMsg(CLIENT_VERIFY
));
515:
516:     if (handleClientVerify(cvm) != 0){
517:         LOG(LOG_ERR, "Error handling ClientVerify!");
518:         return 1;
519:     }
520:
521:     return 0;
522: }
523:
524: bool SecureSocketWrapper::setOtherCert(X509* other_cert){
525:     if (!verify_peer_cert(store, other_cert)){
526:         LOG(LOG_ERR, "Peer certificate validation failed!");
527:         return false;
528:     }
529:     this->other_cert = other_cert;
530:     this->other_id = usernameFromCert(other_cert);
531:     return true;
532: }
533:
534: Message* SecureSocketWrapper::receiveMsg(MessageType type){
535:     return this->receiveMsg(&type, 1);
536: }
537:
538: Message* SecureSocketWrapper::receiveMsg(MessageType type[], int n_types){
539:     Message *m = NULL;
540:     while (m == NULL){
541:         try{
542:             m = receiveAnyMsg();
543:         } catch(const char* msg){
544:             LOG(LOG_ERR, "%s", msg);
545:             return NULL;
546:         }
547:         if (m != NULL){
548:             for (int i = 0; i < n_types; i++){
```

```
549:         if (m->getType() == type[i]){
550:             return m;
551:         }
552:     }
553:     LOG(LOG_WARN, "Received unexpected message of type %s", m->getName().c_str())
;
554:     }
555: }
556: //TODO: add timeout?
557: return NULL;
558: }
559:
560: ClientSecureSocketWrapper::ClientSecureSocketWrapper(X509* cert, EVP_PKEY* my_priv_key, X5
09_STORE* store){
561:     csw = new ClientSocketWrapper();
562:     sw = csw;
563:     init(cert, my_priv_key, store);
564: }
565:
566: int ClientSecureSocketWrapper::connectServer(SecureHost host)
567: {
568:     if (host.getCert() != NULL && !setOtherCert(host.getCert())){
569:         LOG(LOG_ERR, "Peer certificate validation failed!");
570:         return -1;
571:     }
572:     return csw->connectServer(host);
573: }
574:
575: ServerSecureSocketWrapper::ServerSecureSocketWrapper(X509* cert, EVP_PKEY* my_priv_key, X5
09_STORE* store){
576:     ssw = new ServerSocketWrapper();
577:     sw = ssw;
578:     init(cert, my_priv_key, store);
579: }
580:
581: SecureSocketWrapper *ServerSecureSocketWrapper::acceptClient()
582: {
583:     return new SecureSocketWrapper(my_cert, my_priv_key, store, ssw->acceptClient());
584: }
585:
586: SecureSocketWrapper *ServerSecureSocketWrapper::acceptClient(X509* other_cert)
587: {
588:     SecureSocketWrapper* sec_sw = new SecureSocketWrapper(my_cert, my_priv_key, store, ssw
->acceptClient());
589:     if (sec_sw->setOtherCert(other_cert))
590:         return sec_sw;
591:     else{
592:         delete sec_sw;
593:         return NULL;
594:     }
595: }
```

```
1: /**
2:  * @file crypto.h
3:  * @author Mirko Laruina
4:  * @brief Header for crypto algorithms
5:  * @date 2020-06-07
6:  *
7:  */
8: #ifndef CRYPTO_H
9: #define CRYPTO_H
10: #include <openssl/conf.h>
11: #include <openssl/evp.h>
12: #include <openssl/err.h>
13: #include <openssl/rand.h>
14: #include <openssl/pem.h>
15: #include <openssl/hmac.h>
16: #include <openssl/kdf.h>
17: #include <string.h>
18: #include "logging.h"
19:
20: /** AES-256 GCM */
21: #define TAG_SIZE 16
22: #define IV_SIZE 12
23: #define KEY_SIZE 16
24:
25: typedef uint32_t nonce_t;
26:
27: /**
28:  * @brief Print OpenSSL errors
29:  */
30: #define handleErrorsNoException(level) { \
31:     LOG((level), "OpenSSL Exception"); \
32:     FILE* stream; \
33:     if (level < LOG_ERR) \
34:         stream = stdout; \
35:     else \
36:         stream = stderr; \
37:     ERR_print_errors_fp(stream); \
38: }
39:
40: /**
41:  * @brief Print OpenSSL errors and throw exception
42:  */
43: #define handleErrors() { \
44:     handleErrorsNoException(LOG_ERR); \
45:     throw "OpenSSL Error"; \
46: }
47:
48: /**
49:  * Encrypts using AES in GCM mode
50:  *
51:  * @param plaintext    buffer where the plaintext is stored
52:  * @param plaintext_len length of said buffer
53:  * @param aad           additional authenticated data buffer
54:  * @param aad_len       length of said buffer
55:  * @param key           encryption key
56:  * @param iv           initialization vector
57:  * @param ciphertext    buffer (already allocated) where the ct will be stored
58:  * @param tag           tag buffer
59:  *
60:  * @return number of written bytes
61:  */
62: int aes_gcm_encrypt(char *plaintext, int plaintext_len,
63:                    char *aad, int aad_len,
64:                    char *key, char *iv,
65:                    char *ciphertext,
66:                    char *tag);
67:
68: /**
69:  * Decrypts using AES in GCM mode
```

```
70: *
71: * @param ciphertext      buffer where the ciphertext is stored
72: * @param ciphertext_len  length of said buffer
73: * @param aad             additional authenticated data buffer
74: * @param aad_len         length of said buffer
75: * @param key             decryption key
76: * @param iv              initialization vector
77: * @param plaintext       buffer (already allocated) where the pt will be stored
78: * @param tag             tag buffer
79: *
80: * @retval -1             on error
81: * @retval n              number of written bytes
82: */
83: int aes_gcm_decrypt(char *ciphertext, int ciphertext_len,
84:                    char *aad, int aad_len,
85:                    char *key,
86:                    char *iv,
87:                    char *plaintext,
88:                    char *tag);
89:
90: /**
91:  * @brief Generate a ECDH key
92:  *
93:  * Example of usage:
94:  * EVP_PKEY *key=NULL;
95:  * int ret = get_ecdh_key(&key);
96:  *
97:  * @param key    the generated key
98:  * @return int   ???
99:  */
100: int get_ecdh_key(EVP_PKEY **key);
101:
102: /**
103:  * @brief Apply the DHKE to derive a shared secret
104:  *
105:  * @param my_key      first key
106:  * @param peer_pubkey second key
107:  * @param shared_key  output buffer location (unallocated), it will contained the shared
key
108:  * @return int        shared_key length
109:  */
110: int dhke(EVP_PKEY *my_key, EVP_PKEY *peer_pubkey, char **shared_key);
111:
112: /**
113:  * @brief Get a random number
114:  *
115:  * @return nonce_t  the random number
116:  */
117: nonce_t get_rand();
118:
119: /**
120:  * @brief Fills a buffer with a random value
121:  *
122:  * @param char  buffer to fill
123:  * @param bytes number of bytes (buffer length)
124:  */
125: void get_rand(char* buffer, int bytes);
126:
127: /**
128:  * @brief Load a certificate from file
129:  *
130:  * @param file_name    file name of the certificate
131:  * @return X509*       the certificate ptr, NULL if not read correctly
132:  */
133: X509 *load_cert_file(const char *file_name);
134:
135: /**
136:  * @brief Load certificate revocation list from file
137:  *
```

```
138: * @param file_name      file name
139: * @return X509_CRL*      the CRL, NULL if not read correctly
140: */
141: X509_CRL *load_crl_file(const char *file_name);
142:
143: /**
144: * @brief Load a key from file
145: *
146: * @param file_name      file name of the key file
147: * @param password      key password
148: * @return X509*         the key ptr, NULL if not read correctly
149: */
150: EVP_PKEY *load_key_file(const char *file_name, const char* password);
151:
152: /**
153: * @brief Build a CA store from CA certificate and CRL
154: *
155: * @param cacert          CA certificate
156: * @param crl             CRL
157: * @return X509_STORE*    the store
158: */
159: X509_STORE *build_store(X509 *cacert, X509_CRL *crl);
160:
161:
162: /**
163: * @brief
164: *
165: * @param store           Certificate store
166: * @param cert            Certificate
167: * @return true           if validation is successful
168: * @return false          otherwise
169: */
170: bool verify_peer_cert(X509_STORE *store, X509 *cert);
171:
172: /**
173: * @brief Calculate HMAC of the msg
174: *
175: * @param msg             message of which we need the HMAC
176: * @param msg_len          size of said message
177: * @param key              key to use for the HMAC
178: * @param keylen           size of said key
179: * @param hmac             output buffer (uninitialized, it will be allocated)
180: * @return int             size of the HMAC
181: */
182: int hmac(char *msg, int msg_len, char *key, unsigned int keylen,
183:          char *hmac);
184:
185: /**
186: * @brief Compare two HMAC in a secure way
187: *
188: * @param hmac_expected    Expected HMAC
189: * @param hmac_rcv         Received HMAC
190: * @param len              Length of the buffers to compare
191: * @return true             if they are the same
192: * @return false            otherwise
193: */
194: bool compare_hmac(char *hmac_expected, char *hmac_rcv, unsigned int len);
195:
196: /**
197: * @brief Apply HKDF, takes only one info field
198: *
199: * @param key              Key to use
200: * @param key_len          Size of said key
201: * @param info              Info field to use
202: * @param info_len         Size of said info field
203: * @param out               Output buffer (allocated)
204: * @param outlen           Output len
205: */
206: void hkdf_one_info(char *key, size_t key_len,
```

```
207:         char *info, size_t info_len,
208:         char *out, size_t outlen);
209:
210: /**
211:  * @brief Apply HKDF, takes two nonces and a label field
212:  *
213:  * @param key          Key to use
214:  * @param key_len      Size of said key
215:  * @param noncel       First nonce
216:  * @param nonce2       Second nonce
217:  * @param label        Label field
218:  * @param out          Output buffer (allocated)
219:  * @param outlen       Output len
220:  */
221: void hkdf(char *key, size_t key_len,
222:          nonce_t noncel, nonce_t nonce2,
223:          char *label,
224:          char *out, size_t outlen);
225:
226: /**
227:  * Signs the given message
228:  *
229:  * @param msg the message to be signed
230:  * @param msglen the length of the message to be signed
231:  * @param signature pointer to the output signature
232:  * @param prvkey the private key
233:  * @returns the length of the signature
234:  */
235: int dsa_sign(char* msg, int msglen, char** signature,
236:             EVP_PKEY *prvkey);
237:
238: /**
239:  * Checks the given signature on the given message
240:  *
241:  * @param msg the message to be signed
242:  * @param msglen the length of the message to be signed
243:  * @param signature the signature
244:  * @param prvkey the public key
245:  * @returns true if message is authentic, false otherwise
246:  */
247: bool dsa_verify(char* msg, int msglen,
248:                char* signature, int sign_len,
249:                EVP_PKEY *pkey);
250:
251: #endif
```

```
1: #include "security/crypto.h"
2: #include "logging.h"
3:
4: int aes_gcm_encrypt(char *plaintext, int plaintext_len,
5:                    char *aad, int aad_len,
6:                    char *key,
7:                    char *iv,
8:                    char *ciphertext,
9:                    char *tag)
10: {
11:     EVP_CIPHER_CTX *ctx;
12:
13:     int len;
14:
15:     int ciphertext_len;
16:
17:     /* Create and initialise the context */
18:     if (!(ctx = EVP_CIPHER_CTX_new()))
19:         handleErrors();
20:
21:     /* Initialise the encryption operation. */
22:     if (1 != EVP_EncryptInit(ctx, EVP_aes_128_gcm(), (unsigned char*) key, (unsigned char*
) iv))
23:         handleErrors();
24:
25:     /*
26:      * Provide any AAD data. This can be called zero or more times as
27:      * required
28:      */
29:     if (1 != EVP_EncryptUpdate(ctx, NULL, &len, (unsigned char*) aad, aad_len))
30:         handleErrors();
31:
32:     /*
33:      * Provide the message to be encrypted, and obtain the encrypted output.
34:      * EVP_EncryptUpdate can be called multiple times if necessary
35:      */
36:     if (1 != EVP_EncryptUpdate(ctx, (unsigned char*) ciphertext, &len, (unsigned char*) pl
aintext, plaintext_len))
37:         handleErrors();
38:     ciphertext_len = len;
39:
40:     /*
41:      * Finalise the encryption. Normally ciphertext bytes may be written at
42:      * this stage, but this does not occur in GCM mode
43:      */
44:     if (1 != EVP_EncryptFinal(ctx, (unsigned char*) ciphertext + len, &len))
45:         handleErrors();
46:     ciphertext_len += len;
47:
48:     /* Get the tag */
49:     if (1 != EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_GET_TAG, 16, tag))
50:         handleErrors();
51:
52:     /* Clean up */
53:     EVP_CIPHER_CTX_free(ctx);
54:
55:     return ciphertext_len;
56: }
57:
58: int aes_gcm_decrypt(char *ciphertext, int ciphertext_len,
59:                    char *aad, int aad_len,
60:                    char *key,
61:                    char *iv,
62:                    char *plaintext,
63:                    char *tag
64:                    )
65: {
66:     EVP_CIPHER_CTX *ctx;
67:     int len;
```

```
68:     int plaintext_len;
69:     int ret;
70:
71:     /* Create and initialise the context */
72:     if (!(ctx = EVP_CIPHER_CTX_new()))
73:         handleErrors();
74:
75:     /* Initialise the decryption operation. */
76:     if (!EVP_DecryptInit(ctx, EVP_aes_128_gcm(), (unsigned char*) key, (unsigned char*) iv
))
77:         handleErrors();
78:
79:     /*
80:      * Provide any AAD data. This can be called zero or more times as
81:      * required
82:      */
83:     if (!EVP_DecryptUpdate(ctx, NULL, &len, (unsigned char*) aad, aad_len))
84:         handleErrors();
85:
86:     /*
87:      * Provide the message to be decrypted, and obtain the plaintext output.
88:      * EVP_DecryptUpdate can be called multiple times if necessary
89:      */
90:     if (!EVP_DecryptUpdate(ctx, (unsigned char*) plaintext, &len, (unsigned char*) ciphert
ext, ciphertext_len))
91:         handleErrors();
92:     plaintext_len = len;
93:
94:     /* Set expected tag value. Works in OpenSSL 1.0.1d and later */
95:     if (!EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_SET_TAG, 16, tag))
96:         handleErrors();
97:
98:     /*
99:      * Finalise the decryption. A positive return value indicates success,
100:      * anything else is a failure - the plaintext is not trustworthy.
101:      */
102:     ret = EVP_DecryptFinal(ctx, (unsigned char*) plaintext + len, &len);
103:
104:     /* Clean up */
105:     EVP_CIPHER_CTX_free(ctx);
106:
107:     if (ret > 0)
108:     {
109:         /* Success */
110:         plaintext_len += len;
111:         return plaintext_len;
112:     }
113:     else
114:     {
115:         /* Verify failed */
116:         return -1;
117:     }
118: }
119:
120: int get_ecdh_key(EVP_PKEY **key)
121: {
122:     EVP_PKEY *dh_params = NULL;
123:     EVP_PKEY_CTX *ctx_params;
124:     int ret;
125:
126:     ctx_params = EVP_PKEY_CTX_new_id(EVP_PKEY_EC, NULL);
127:     if (!ctx_params)
128:     {
129:         handleErrors();
130:     }
131:     ret = EVP_PKEY_paramgen_init(ctx_params);
132:     if (!ret)
133:     {
134:         handleErrors();
```



```
135:     }
136:     //Using NID_X9_62_prime256v1 curve
137:     ret = EVP_PKEY_CTX_set_ec_paramgen_curve_nid(
138:         ctx_params,
139:         NID_X9_62_prime256v1);
140:     if (!ret)
141:     {
142:         handleErrors();
143:     }
144:
145:     ret = EVP_PKEY_paramgen(ctx_params, &dh_params);
146:     if (!ret)
147:     {
148:         handleErrors();
149:     }
150:     //check
151:     if (dh_params == NULL)
152:     {
153:         handleErrors();
154:         return ret;
155:     }
156:
157:     EVP_PKEY_CTX_free(ctx_params);
158:
159:     // creating the context for key generation
160:     EVP_PKEY_CTX *ctx = EVP_PKEY_CTX_new(dh_params, NULL);
161:     if (ctx == NULL)
162:     {
163:         handleErrors();
164:     }
165:     // Generating the key
166:     ret = EVP_PKEY_keygen_init(ctx);
167:     if (!ret)
168:     {
169:         handleErrors();
170:     }
171:
172:     ret = EVP_PKEY_keygen(ctx, key);
173:     if (!ret)
174:     {
175:         handleErrors();
176:     }
177:
178:     //check
179:     if (key == NULL)
180:     {
181:         handleErrors();
182:         return ret;
183:     }
184:
185:     EVP_PKEY_CTX_free(ctx);
186:     return ret;
187: }
188:
189: /**
190:  * @brief Apply the DHKE to derive a shared secret
191:  *
192:  * @param my_key          first key
193:  * @param peer_pubkey     second key
194:  * @param shared_key      output buffer location (unallocated), it will contained the shared
key
195:  * @return int            ???
196:  */
197: int dhke(EVP_PKEY *my_key, EVP_PKEY *peer_pubkey, char **shared_key)
198: {
199:     EVP_PKEY_CTX *derivation_ctx;
200:     size_t shared_key_len;
201:
202:     derivation_ctx = EVP_PKEY_CTX_new(my_key, NULL);
```

```
203:     if (derivation_ctx == NULL)
204:     {
205:         handleErrors();
206:     }
207:     if (EVP_PKEY_derive_init(derivation_ctx) <= 0)
208:     {
209:         handleErrors();
210:     }
211:
212:     if (EVP_PKEY_derive_set_peer(derivation_ctx, peer_pubkey) <= 0)
213:     {
214:         handleErrors();
215:     }
216:
217:     // "Dummy" derivation to extract key len
218:     EVP_PKEY_derive(derivation_ctx, NULL, &shared_key_len);
219:     *shared_key = (char *)malloc(shared_key_len);
220:     if (!*shared_key)
221:     {
222:         LOG_PERROR(LOG_ERR, "Malloc failed: %s");
223:         handleErrors();
224:     }
225:
226:     // Real derivation
227:     if (EVP_PKEY_derive(derivation_ctx, (unsigned char*) *shared_key, &shared_key_len) <=
0)
228:     {
229:         handleErrors();
230:     }
231:
232:     EVP_PKEY_CTX_free(derivation_ctx);
233:     return shared_key_len;
234: }
235:
236: /**
237:  * @brief Get a random number
238:  *
239:  * @return nonce_t the random number
240:  */
241: nonce_t get_rand()
242: {
243:     nonce_t random_num;
244:     RAND_bytes((unsigned char *)&random_num, sizeof(random_num));
245:     return random_num;
246: }
247:
248: /**
249:  * @brief Fills a buffer with a random value
250:  *
251:  * @param char buffer to fill
252:  * @param bytes number of bytes (buffer length)
253:  */
254: void get_rand(char* buffer, int bytes){
255:     RAND_bytes((unsigned char*) buffer, bytes);
256: }
257:
258: /**
259:  * @brief Load a certificate from file
260:  *
261:  * @param file_name file name of the certificate
262:  * @return X509* the certificate ptr, NULL if not read correctly
263:  */
264: X509 *load_cert_file(const char *file_name)
265: {
266:     FILE *cert_file = fopen(file_name, "r");
267:     if (!cert_file)
268:     {
269:         return NULL;
270:     }
```

```
271:     X509 *cert = PEM_read_X509(cert_file, NULL, NULL, NULL);
272:     fclose(cert_file);
273:     return cert;
274: }
275:
276: /**
277:  * @brief Load a key from file
278:  *
279:  * @param file_name      file name of the key file
280:  * @param password      key password
281:  * @return X509*        the key ptr, NULL if not read correctly
282:  */
283: EVP_PKEY *load_key_file(const char *file_name, const char* password)
284: {
285:     FILE *key_file = fopen(file_name, "r");
286:     if (!key_file)
287:     {
288:         return NULL;
289:     }
290:     EVP_PKEY *key = PEM_read_PrivateKey(key_file, NULL, NULL, (void*) password);
291:     fclose(key_file);
292:     return key;
293: }
294:
295: /**
296:  * @brief Load certificate revocation list from file
297:  *
298:  * @param file_name      file name
299:  * @return X509_CRL*     the CRL, NULL if not read correctly
300:  */
301: X509_CRL *load_crl_file(const char *file_name)
302: {
303:     FILE *crl_file = fopen(file_name, "r");
304:     if (!crl_file)
305:     {
306:         return NULL;
307:     }
308:     X509_CRL *crl = PEM_read_X509_CRL(crl_file, NULL, NULL, NULL);
309:     fclose(crl_file);
310:     return crl;
311: }
312:
313: /**
314:  * @brief Build a CA store from CA certificate and CRL
315:  *
316:  * @param cacert          CA certificate
317:  * @param crl             CRL
318:  * @return X509_STORE*    the store
319:  */
320: X509_STORE *build_store(X509 *cacert, X509_CRL *crl)
321: {
322:     X509_STORE *store = X509_STORE_new();
323:     if (!store)
324:     {
325:         handleErrors();
326:     }
327:     if (1 != X509_STORE_add_cert(store, cacert))
328:     {
329:         handleErrors();
330:     }
331:     if (1 != X509_STORE_add_crl(store, crl))
332:     {
333:         handleErrors();
334:     }
335:     if (1 != X509_STORE_set_flags(store, X509_V_FLAG_CRL_CHECK))
336:     {
337:         handleErrors();
338:     }
339: }
```

```
340:     return store;
341: }
342:
343: /**
344:  * @brief
345:  *
346:  * @param store
347:  * @param cert
348:  * @return true
349:  * @return false
350:  */
351: bool verify_peer_cert(X509_STORE *store, X509 *cert)
352: {
353:     X509_STORE_CTX *verify_ctx = X509_STORE_CTX_new();
354:     if (!verify_ctx)
355:     {
356:         handleErrors();
357:     }
358:
359:     if (1 != X509_STORE_CTX_init(verify_ctx, store, cert, NULL))
360:     {
361:         handleErrors();
362:     }
363:
364:     int ret = X509_verify_cert(verify_ctx);
365:     if (ret == 1){
366:         return true;
367:     } else if (ret == 0){
368:         return false;
369:     } else {
370:         handleErrorsNoException(LOG_DEBUG);
371:         return false;
372:     }
373: }
374:
375: int hmac(char *msg, int msg_len, char *key, unsigned int keylen,
376:          char *hmac)
377: {
378:     const EVP_MD *md = EVP_sha256();
379:     unsigned int hash_size = EVP_MD_size(md);
380:
381:     HMAC_CTX *ctx = HMAC_CTX_new();
382:     HMAC_Init_ex(ctx, key, keylen, md, NULL);
383:
384:     HMAC_Update(ctx, (unsigned char*) msg, sizeof(msg));
385:     HMAC_Final(ctx, (unsigned char*) hmac, &hash_size);
386:
387:     HMAC_CTX_free(ctx);
388:     /*
389:     while ((bytes_read = fread(buffer, 1, hash_size, file)) > 0)
390:     {
391:         HMAC_Update(ctx, buffer, bytes_read);
392:         printf("len: %d\n", bytes_read);
393:     }
394:     */
395:     return hash_size;
396: }
397:
398: bool compare_hmac(char *hmac_expected, char *hmac_rcv, unsigned int len)
399: {
400:     if (0 != CRYPTO_memcmp(hmac_expected, hmac_rcv, len))
401:     {
402:         return false;
403:     }
404:     else
405:     {
406:         return true;
407:     }
408: }
```

```
409:
410: void hkdf_one_info(char *key, size_t key_len,
411:                   char *info, size_t info_len,
412:                   char *out, size_t outlen)
413: {
414:     EVP_PKEY_CTX *ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_HKDF, NULL);
415:     if (!ctx)
416:     {
417:         handleErrors();
418:     }
419:
420:     if (EVP_PKEY_derive_init(ctx) <= 0)
421:     {
422:         handleErrors();
423:     }
424:
425:     if (EVP_PKEY_CTX_set_hkdf_md(ctx, EVP_sha256()) <= 0)
426:     {
427:         handleErrors();
428:     }
429:     if (EVP_PKEY_CTX_set1_hkdf_key(ctx, key, key_len) <= 0)
430:     {
431:         handleErrors();
432:     }
433:     if (EVP_PKEY_CTX_add1_hkdf_info(ctx, (unsigned char *)info, info_len) <= 0)
434:     {
435:         handleErrors();
436:     }
437:     if (EVP_PKEY_derive(ctx, (unsigned char*) out, &outlen) <= 0)
438:     {
439:         handleErrors();
440:     }
441:     EVP_PKEY_CTX_free(ctx);
442: }
443:
444: void hkdf(char *key, size_t key_len,
445:          nonce_t noncel, nonce_t nonce2,
446:          char *label,
447:          char *out, size_t outlen)
448: {
449:     // label is a string, we remove the termination null char
450:     size_t info_len = sizeof(nonce_t) * 2 + strlen(label);
451:     char *info = (char *)malloc(info_len);
452:     if (!info){
453:         LOG_PERROR(LOG_ERR, "Malloc failed: %ss");
454:         throw "Malloc error";
455:     }
456:     char *info_buf = info;
457:     strcpy((char *)info_buf, label);
458:     info_buf += strlen(label);
459:     memcpy(info_buf, (void *)&noncel, sizeof(nonce1));
460:     info_buf += sizeof(nonce1);
461:     memcpy(info_buf, (void *)&nonce2, sizeof(nonce2));
462:
463:     hkdf_one_info(key, key_len, info, info_len, out, outlen);
464:     free(info);
465: }
466:
467: int dsa_sign(char* msg, int msglen, char** signature,
468:             EVP_PKEY *prvkey){
469:     unsigned int sign_len;
470:
471:     EVP_MD_CTX* ctx = EVP_MD_CTX_new();
472:     if (ctx == NULL){
473:         handleErrors();
474:     }
475:
476:     *signature = (char*) malloc(EVP_PKEY_size(prvkey));
477:     if (*signature == NULL){
```

```
478:         LOG_PERROR(LOG_ERR, "Malloc failed: %s");
479:         return -1;
480:     }
481:
482:     if (EVP_SignInit(ctx, EVP_sha256()) != 1){
483:         handleErrors();
484:     }
485:
486:     if (EVP_SignUpdate(ctx, (unsigned char*) msg, msglen) != 1){
487:         handleErrors();
488:     }
489:
490:     if (EVP_SignFinal(ctx, (unsigned char*) *signature, &sign_len, prvkey) != 1){
491:         handleErrors();
492:     }
493:
494:     EVP_MD_CTX_free(ctx);
495:
496:     return sign_len;
497: }
498:
499: bool dsa_verify(char* msg, int msglen,
500:                char* signature, int sign_len,
501:                EVP_PKEY *pkey){
502:     EVP_MD_CTX* ctx = EVP_MD_CTX_new();
503:
504:     if (ctx == NULL){
505:         handleErrors();
506:     }
507:
508:     if (EVP_VerifyInit(ctx, EVP_sha256()) != 1){
509:         handleErrors();
510:     }
511:
512:     if (EVP_VerifyUpdate(ctx, msg, msglen) != 1){
513:         handleErrors();
514:     }
515:
516:     int ret = EVP_VerifyFinal(ctx, (unsigned char*) signature, sign_len, pkey);
517:     if (ret != 1){
518:         return false;
519:     }
520:     return true;
521: }
```

```
1: /**
2:  * @file dump_buffer.h
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Utility function for dumping a buffer as hex string.
6:  *
7:  * @date 2020-05-17
8:  */
9:
10: #ifndef DUMP_BUFFER_H
11: #define DUMP_BUFFER_H
12:
13:
14: /**
15:  * Prints content of buffer to stdout, showing it as hex values.
16:  *
17:  * It uses the logging infrastructure to print.
18:  *
19:  * @param buffer    pointer to the buffer to be printed
20:  * @param len        the length (in bytes) of the buffer
21:  */
22: void dump_buffer_hex(char* buffer, int len, int log_level, const char* name);
23:
24: #if LOG_LEVEL == LOG_DEBUG
25: #define DUMP_BUFFER_HEX_DEBUG(buffer, len) dump_buffer_hex(buffer, len, LOG_DEBUG, #buffer
26: )
27: #else
28: #define DUMP_BUFFER_HEX_DEBUG(buffer, len)
29: #endif
30:
31: #endif // DUMP_BUFFER_H
```

```
1: /**
2:  * @file dump_buffer.cpp
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Implementation of dump_buffer.h.
6:  *
7:  * @see dump_buffer.h
8:  */
9:
10: #include "utls/dump_buffer.h"
11: #include "logging.h"
12: #include <stdio.h>
13: #include <stdlib.h>
14: #include <string.h>
15: #include <cctype>
16:
17: #define ROW 32
18:
19:
20: void dump_buffer_hex(char* buffer, int len, int log_level, const char* name){
21:     char *str, tmp3[4], tmp1[2];
22:     int i, j;
23:     int n_rows = (len+ROW-1)/ROW;
24:
25:     str = (char*) malloc(n_rows*(3*ROW + 4 + ROW + 1)+1);
26:     if (!str){
27:         LOG_PERROR(LOG_ERR, "Malloc failed: %s");
28:         return;
29:     }
30:
31:     const char* col_sep = " ";
32:     const char* row_sep = "\n";
33:
34:     str[0] = '\0';
35:     for (i=0; i<n_rows; i++){
36:         for (j=0; j<ROW; j++){
37:             int idx = i*ROW+j;
38:             if (idx < len){
39:                 sprintf(tmp3, "%02x ", (unsigned char) buffer[idx]);
40:             } else {
41:                 sprintf(tmp3, " ");
42:             }
43:             strcat(str, tmp3);
44:         }
45:
46:         strcat(str, col_sep);
47:
48:         for (j=0; j<ROW; j++){
49:             int idx = i*ROW+j;
50:             if (idx >= len)
51:                 break;
52:
53:             if (isprint(buffer[idx])){
54:                 sprintf(tmp1, "%c", buffer[idx]);
55:             } else{
56:                 sprintf(tmp1, ".");
57:             }
58:             strcat(str, tmp1);
59:         }
60:         if (i != n_rows - 1)
61:             strcat(str, row_sep);
62:     }
63:     LOG(log_level, "Dumping %s", name);
64:     if (log_level >= LOG_LEVEL)
65:         printf("%s%s\033[0m\n", logColor(log_level), str);
66:     free(str);
67: }
```



```
1: /**
2:  * @file host.h
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Definition of the helper class "Host"
6:  *
7:  * @date 2020-05-17
8:  */
9:
10: #ifndef HOST_H
11: #define HOST_H
12:
13: #include <sys/socket.h>
14: #include <netinet/in.h>
15: #include <string>
16: #include "logging.h"
17:
18: using namespace std;
19:
20: /**
21:  * Class that holds a host information
22:  *
23:  * OpenSSL certificates are held in SecureHost class.
24:  */
25: class Host{
26: private:
27:     struct sockaddr_in addr;
28:
29: public:
30:     /**
31:      * Constructs new empty instance
32:      */
33:     Host() {}
34:
35:     /**
36:      * Constructs new instance from given inet address
37:      *
38:      * @param addr the inet address of the remote host
39:      */
40:     Host(struct sockaddr_in addr)
41:         : addr(addr) {}
42:
43:     /**
44:      * Constructs new instance from IP/port pair
45:      *
46:      * @param ip the IP address the remote host
47:      * @param port the port the remote host
48:      */
49:     Host(const char* ip, int port);
50:
51:     /** Returns the inet address of the host */
52:     struct sockaddr_in getAddress(){return addr;}
53:
54:     /** Returns the inet address of the host */
55:     string toString();
56:
57: };
58:
59: #endif // HOST_H
```

```
1: /**
2:  * @file host.cpp
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Implementation of host.h.
6:  *
7:  * @see host.h
8:  *
9:  * @date 2020-05-20
10: */
11:
12: #include "network/host.h"
13: #include "network/inet_utils.h"
14:
15: Host::Host(const char* ip, int port){
16:     addr = make_sv_sockaddr_in(ip, port);
17: }
18: string Host::toString(){
19:     return sockaddr_in_to_string(addr);
20: }
```

```
1:  /**
2:   * @file user_list.h
3:   * @author Riccardo Mancini
4:   *
5:   * @brief Implementation of the UserList class
6:   *
7:   * @date 2020-05-23
8:   */
9:
10: #include <sstream>
11: #include <string>
12: #include <vector>
13: #include <iterator>
14: #include <iostream>
15: #include "user_list.h"
16: #include "config.h"
17:
18: using namespace std;
19:
20: typedef map<string, User*>::iterator Iterator;
21:
22: UserList::UserList() {
23:     pthread_mutex_init(&mutex, NULL);
24: }
25:
26: bool UserList::add(User *u) {
27:     bool success;
28:     pthread_mutex_lock(&mutex);
29:     if ((success = user_map_by_fd.size() < MAX_USERS)) {
30:         user_map_by_fd.insert(
31:             pair<int, User*>(u->getSocketWrapper()->getDescriptor(), u)
32:         );
33:         if (!u->getUsername().empty()) {
34:             user_map_by_username.insert(
35:                 pair<string, User*>(u->getUsername(), u)
36:             );
37:         }
38:     }
39:     pthread_mutex_unlock(&mutex);
40:
41:     return success;
42: }
43:
44: User* UserList::get(string username) {
45:     User* u = NULL;
46:     pthread_mutex_lock(&mutex);
47:     if (user_map_by_username.find(username) != user_map_by_username.end()) {
48:         u = user_map_by_username.at(username);
49:         u->increaseRefs();
50:         LOG(LOG_DEBUG, "Thread %ld got reference to user %d",
51:             pthread_self(), u->getSocketWrapper()->getDescriptor()
52:         );
53:     }
54:     pthread_mutex_unlock(&mutex);
55:     return u;
56: }
57:
58: User* UserList::get(int fd) {
59:     User* u = NULL;
60:     pthread_mutex_lock(&mutex);
61:     if (user_map_by_fd.find(fd) != user_map_by_fd.end()) {
62:         u = user_map_by_fd.at(fd);
63:         u->increaseRefs();
64:         LOG(LOG_DEBUG, "Thread %ld got reference to user %d",
65:             pthread_self(), u->getSocketWrapper()->getDescriptor()
66:         );
67:     }
68:     pthread_mutex_unlock(&mutex);
69:     return u;
```

```
70: }
71:
72: bool UserList::exists(string username){
73:     bool res;
74:     pthread_mutex_lock(&mutex);
75:     res = user_map_by_username.find(username) != user_map_by_username.end();
76:     pthread_mutex_unlock(&mutex);
77:     return res;
78: }
79:
80: bool UserList::exists(int fd){
81:     bool res;
82:     pthread_mutex_lock(&mutex);
83:     res = user_map_by_fd.find(fd) != user_map_by_fd.end();
84:     pthread_mutex_unlock(&mutex);
85:     return res;
86: }
87:
88: void UserList::yield(User* u){
89:     pthread_mutex_lock(&mutex);
90:     u->decreaseRefs();
91:     LOG(LOG_DEBUG, "Thread %ld yielded user %d (refcount: %d)",
92:         pthread_self(), u->getSocketWrapper()->getDescriptor(), u->countRefs()
93:     );
94:     if (u->countRefs() == 0 && u->getState() == DISCONNECTED){
95:         if (user_map_by_username.find(u->getUsername())
96:             != user_map_by_username.end()
97:         ){
98:             user_map_by_username.erase(u->getUsername());
99:         }
100:         if (user_map_by_fd.find(u->getSocketWrapper()->getDescriptor())
101:             != user_map_by_fd.end()
102:         ){
103:             user_map_by_fd.erase(u->getSocketWrapper()->getDescriptor());
104:         }
105:         delete u;
106:     }
107:     pthread_mutex_unlock(&mutex);
108: }
109:
110: string UserList::listAvailableFromTo(int from){
111:     ostringstream os;
112:     int n = 0;
113:
114:     pthread_mutex_lock(&mutex);
115:     for (Iterator it = user_map_by_username.begin();
116:          n < from+MAX_USERS_IN_MESSAGE && it != user_map_by_username.end();
117:          ++it
118:     ){
119:         if (it->second->getState() == AVAILABLE){
120:             if (n < from)
121:                 continue;
122:
123:             os << it->first;
124:             if (n < from+MAX_USERS_IN_MESSAGE-1)
125:                 os << ",";
126:
127:             n++;
128:         }
129:     }
130:
131:     pthread_mutex_unlock(&mutex);
132:     os << '\0';
133:
134:     return os.str();
135: }
136:
137: int UserList::size(){
138:     int sz;
```

```
139:     pthread_mutex_lock(&mutex);
140:     sz = user_map_by_fd.size();
141:     pthread_mutex_unlock(&mutex);
142:     return sz;
143: }
```

```
1:  /**
2:  * @file args.h
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Definition of the Args class
6:  *
7:  * @date 2020-05-27
8:  */
9:
10: #ifndef ARGS_H
11: #define ARGS_H
12:
13: #include <cstring>
14: #include <string>
15: #include <list>
16: #include <iostream>
17: #include <vector>
18:
19: using namespace std;
20:
21: /**
22:  * Utility class that parses an input line into a list of arguments (argc, argv)
23:  */
24: class Args {
25: private:
26:     int status;
27:     vector<string> argv;
28:
29:     void parseLine(string s);
30: public:
31:     /**
32:      * Default constructor
33:      */
34:     Args() : status(0) {}
35:
36:     /**
37:      * Constructor that parses the given input line
38:      */
39:     Args(char* line);
40:
41:     /**
42:      * Constructor that reads from the given input stream.
43:      */
44:     Args(std::istream &is);
45:
46:     /**
47:      * Operator overload for printing the arguments with cout
48:      *
49:      * Format: ["arg1", "arg2"]
50:      */
51:     friend std::ostream& operator<<(std::ostream& os, const Args& b);
52:
53:     /**
54:      * Returns argument count.
55:      *
56:      * @returns >=0 argument count
57:      * @returns -1 error reading stream (may be caused by EOF)
58:      */
59:     int getArgc(){ return status == 0 ? argv.size() : -1;}
60:
61:     /**
62:      * Returns nth argument
63:      */
64:     const char* getArgv(unsigned int i) {
65:         if (status == 0 && i < argv.size())
66:             return argv.at(i).c_str();
67:         else
68:             return NULL;
69:     }
```

```
70:
71:     /**
72:      * Returns user friendly content as a C string.
73:      */
74:     const char* c_str();
75: };
76:
77: #endif //PARSE_ARGS_H
```

```
1:  /**
2:   * @file args.cpp
3:   * @author Riccardo Mancini
4:   *
5:   * @brief Implementation of the Args class
6:   *
7:   * Adapted from https://stackoverflow.com/a/14266139
8:   *
9:   * @date 2020-05-27
10:  */
11:
12: #include "utls/args.h"
13: #include <sstream>
14:
15: using namespace std;
16:
17: void Args::parseLine(string s){
18:     string delimiter = " ";
19:     size_t pos = 0;
20:     string token;
21:
22:     while ((pos = s.find(delimiter)) != string::npos) {
23:         token = s.substr(0, pos);
24:         argv.push_back(token);
25:         s.erase(0, pos + delimiter.length());
26:     }
27:
28:     if (!s.empty()){
29:         argv.push_back(s);
30:     }
31: }
32:
33: Args::Args(char* line){
34:     parseLine(string(line));
35: }
36:
37: Args::Args(istream &is){
38:     string line;
39:     getline(is, line);
40:     if (!is){
41:         status = 1;
42:     } else {
43:         status = 0;
44:         parseLine(line);
45:     }
46: }
47:
48: ostream& operator<<(ostream& os, const Args& a){
49:     os << "[";
50:
51:     for (vector<string>::const_iterator it = a.argv.begin(); it != a.argv.end(); it++){
52:         os << *it;
53:         if (it != a.argv.end()-1)
54:             os << ",";
55:     }
56:
57:     os << "]";
58:     return os;
59: }
60:
61: const char* Args::c_str(){
62:     ostringstream os;
63:     os << *this;
64:     return os.str().c_str();
65: }
```



```
1: /**
2:  * @file single_player.h
3:  * @author Mirko Laruina
4:  * @author Riccardo Mancini
5:  *
6:  * @brief Implementation of the single player game main function
7:  *
8:  * @date 2020-05-27
9:  */
10:
11: #include "single_player.h"
12: #include <iostream>
13: #include "utils/args.h"
14: #include "connect4.h"
15:
16: using namespace std;
17:
18: int playSinglePlayer(){
19:     int choosen_col, adv_col;
20:     int win;
21:     Connect4 c;
22:
23:     cout<<"Who do you want to be? X or O ?"<<endl;
24:
25:     do {
26:         cout<<"> "<<flush;
27:         Args args(cin);
28:         if (args.getArgc() < 0){
29:             return 1;
30:         } else if (args.getArgc() == 1 && c.setPlayer(args.getArgv(0)[0])){
31:             break;
32:         } else{
33:             continue;
34:         }
35:     } while (1);
36:
37:     cout<<"You are playing as "<<c.getPlayer()<<endl;
38:
39:     cout<<"This is the starting board:"<<endl;
40:     cout<<c;
41:
42:     srand(time(NULL));
43:
44:     do {
45:         cout<<"Write the column you want to insert the token to"<<endl;
46:         do {
47:             cout<<"> "<<flush;
48:             Args args(cin);
49:             if (args.getArgc() == 1){
50:                 choosen_col = args.getArgv(0)[0] - '0';
51:             } else if (args.getArgc() < 0){
52:                 return 1;
53:             } else {
54:                 choosen_col = -1;
55:             }
56:         } while(choosen_col < 0 || choosen_col > 7);
57:
58:         win = c.play(choosen_col-1, c.getPlayer());
59:         cout<<c;
60:         if(win == 1){
61:             cout<<"Congratulation, you won!"<<endl;
62:         } else if(win == -1){
63:             cout<<"The column is full, choose a different one!"<<endl;
64:             continue;
65:         } else if(win == -2){
66:             cout<<"The entire board is filled: it is a draw!"<<endl;
67:             break;
68:         }
69:     }
```

```
70:         if(win != 1){
71:             do {
72:                 adv_col = rand()%c.getNumCols();
73:                 cout<<"Your enemy has chosen column "<<adv_col<<endl;
74:                 win = c.play(adv_col, c.getAdv());
75:                 cout<<c;
76:                 if(win == 1){
77:                     cout<<"Damn! You lost!"<<endl;
78:                 } else if(win == -1){
79:                     cout<<"The column is full, the adversary has to chose a different one!"
"<<endl;
80:                     continue;
81:                 } else if(win == -2){
82:                     cout<<"The entire board is filled: it is a draw!"<<endl;
83:                     break;
84:                 }
85:             } while (win == -1);
86:         }
87:     } while (win == -1 || win == 0);
88:     return 0;
89: }
```

```
1: /**
2:  * @file multi_player.h
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Implementation of the multi player game main function and
6:  *        connection with peer functions
7:  *
8:  * @date 2020-05-29
9:  */
10:
11: #include "multi_player.h"
12: #include "connect4.h"
13: #include <iostream>
14: #include "utils/args.h"
15:
16: int playWithPlayer(int turn, SecureSocketWrapper *sw){
17:     int choosen_col, adv_col;
18:     int win;
19:
20:     Connect4 c;
21:     cout<<"Who do you want to be? X or O ?"<<endl;
22:
23:     do {
24:         cout<<"> "<<flush;
25:         Args args(cin);
26:         if (args.getArgc() == -1){
27:             return 1;
28:         } else if (args.getArgc() == 1 && c.setPlayer(args.getArgv(0)[0])){
29:             break;
30:         } else if (args.getArgc() < 0) {
31:             return 1;
32:         } else{
33:             continue;
34:         }
35:     } while (1);
36:
37:     cout<<"You are playing as "<<c.getPlayer()<<endl;
38:
39:     cout<<"This is the starting board:"<<endl;
40:     cout<<c;
41:
42:     do {
43:         if (turn == MY_TURN){
44:             cout<<"Write the column you want to insert the token to"<<endl;
45:             do {
46:                 cout<<"> "<<flush;
47:                 Args args(cin);
48:                 if (args.getArgc() == 1){
49:                     choosen_col = args.getArgv(0)[0]-'0';
50:                 } else if (args.getArgc() < 0){
51:                     return 1;
52:                 } else {
53:                     choosen_col = -1;
54:                 }
55:             } while(choosen_col < 0 || choosen_col > 7);
56:
57:             win = c.play(choosen_col-1, c.getPlayer());
58:             cout<<c;
59:
60:             if (win != -1){
61:                 MoveMessage mm(choosen_col-1);
62:                 int ret = sw->sendMsg(&mm);
63:                 if (ret != 0){
64:                     LOG(LOG_ERR, "Connection error");
65:                     return 1;
66:                 }
67:             }
68:
69:             if(win == 1){
```

```
70:         cout<<"Congratulation, you won!"<<endl;
71:     } else if(win == -1){
72:         cout<<"The column is full, choose a different one!"<<endl;
73:         continue;
74:     } else if(win == -2){
75:         cout<<"The entire board is filled: it is a draw!"<<endl;
76:         break;
77:     } else{
78:         turn = THEIR_TURN;
79:     }
80:
81: } else{        // THEIR_TURN
82:     do {
83:         MoveMessage *mm;
84:         mm = dynamic_cast<MoveMessage*>(sw->receiveMsg(MOVE));
85:         if (mm == NULL){
86:             LOG(LOG_ERR, "Connection error");
87:             return 1;
88:         }
89:         adv_col = mm->getColumn();
90:         cout<<"Your enemy has chosen column "<<adv_col<<endl;
91:         win = c.play(adv_col, c.getAdv());
92:         cout<<c;
93:         if(win == 1){
94:             cout<<"Damn! You lost!"<<endl;
95:         } else if(win == -1){
96:             cout<<"The column is full, the adversary has lost!"<<endl;
97:             break;
98:         } else if(win == -2){
99:             cout<<"The entire board is filled: it is a draw!"<<endl;
100:            break;
101:        }
102:    } while (win == -1);
103:    turn = MY_TURN;
104: }
105: } while (win == -1 || win == 0);
106: return 0;
107: }
108:
109: SecureSocketWrapper* waitForPeer(int port, SecureHost host, X509* cert, EVP_PKEY* key, X50
9_STORE* store){
110:     int ret;
111:
112:     ServerSecureSocketWrapper *ssw;
113:     ssw = new ServerSecureSocketWrapper(cert, key, store);
114:
115:
116:     ret = ssw->bindPort(port);
117:     if (ret != 0){
118:         cout<<"Could not bind to port: "<<ssw->getPort()<<endl;
119:         delete ssw;
120:         return NULL;
121:     }
122:
123:     cout<<"Waiting for connection on port: "<<ssw->getPort()<<endl;
124:
125:     SecureSocketWrapper *sw = ssw->acceptClient(host.getCert());
126:
127:     if (sw == NULL){
128:         LOG(LOG_ERR, "Connection error: no client with valid certificate connected");
129:         return NULL;
130:     }
131:
132:     ret = sw->handshakeServer();
133:
134:     if (ret != 0){
135:         LOG(LOG_ERR, "Handshake error");
136:         return NULL;
137:     }
```

```
138:
139:     Host p = sw->getConnectedHost();
140:     cout<<"Accepted client: "<<p.toString()<<endl;
141:
142:     StartGameMessage *sgm = dynamic_cast<StartGameMessage*>(sw->receiveMsg(START_GAME_PEER
));
143:
144:     if (sgm == NULL){
145:         LOG(LOG_ERR, "Connection error");
146:         return NULL;
147:     }
148:
149:     LOG(LOG_INFO, "Connected to %s", p.toString().c_str());
150:     return sw;
151: }
152:
153: SecureSocketWrapper* connectToPeer(SecureHost peer, X509* cert, EVP_PKEY* key, X509_STORE*
store){
154:     cout<<"Connecting to: "<<peer.toString()<<endl;
155:
156:     ClientSecureSocketWrapper *csw = new ClientSecureSocketWrapper(cert, key, store);
157:
158:     int ret;
159:     int retry = 10;
160:     do {
161:         retry--;
162:         ret = csw->connectServer(peer);
163:         if (ret != 0){
164:             if (retry != 0){
165:                 LOG(LOG_INFO, "Peer is not online yet, retrying in 1 second.");
166:                 sleep(1);
167:             } else {
168:                 break;
169:             }
170:         }
171:     } while(retry != 0);
172:
173:     if (ret != 0){
174:         LOG(LOG_ERR, "Connection error");
175:         return NULL;
176:     }
177:
178:     ret = csw->handshakeClient();
179:
180:     if (ret != 0){
181:         LOG(LOG_ERR, "Handshake error");
182:         return NULL;
183:     }
184:
185:     StartGameMessage m;
186:     ret = csw->sendMsg(&m);
187:
188:     if (ret != 0){
189:         LOG(LOG_ERR, "Connection error");
190:         return NULL;
191:     }
192:
193:     LOG(LOG_INFO, "Connected to %s", peer.toString().c_str());
194:
195:     return csw;
196: }
```

```
1: /**
2:  * @file server.h
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Implementation of the utility class used to communicate with the server
6:  *
7:  * @date 2020-05-29
8:  */
9:
10: #include "server.h"
11: #include "network/messages.h"
12: #include "network/inet_utils.h"
13: #include <iostream>
14:
15: Server::~Server()
16: {
17:     if (sw != NULL)
18:     {
19:         sw->closeSocket();
20:         delete sw;
21:     }
22: }
23:
24: int Server::getServerCert()
25: {
26:     sw->sendCertRequest();
27:     CertificateMessage *crm = dynamic_cast<CertificateMessage *>(sw->receiveMsg(CERTIFICAT
E));
28:     if (crm != NULL && sw->setOtherCert(crm->getCert()))
29:     {
30:         return 0;
31:     }
32:     return 1;
33: }
34:
35: int Server::registerToServer()
36: {
37:     if (sw->connectServer(host) != 0)
38:     {
39:         connected = false;
40:         return 1;
41:     }
42:
43:     if (host.getCert() == NULL)
44:     {
45:         if (getServerCert() != 0)
46:         {
47:             connected = false;
48:             return 1;
49:         }
50:     }
51:
52:     if (sw->handshakeClient() != 0)
53:     {
54:         connected = false;
55:         return 1;
56:     }
57:
58:     RegisterMessage msg(getPlayerUsername());
59:
60:     int ret = sw->sendMsg(&msg);
61:     connected = ret == 0;
62:     return ret;
63: }
64:
65: string Server::getUserList()
66: {
67:     UsersListRequestMessage req_msg;
68: }
```

```
69:     if (sw->sendMsg(&req_msg) != 0)
70:     {
71:         connected = false;
72:         return "";
73:     }
74:
75:     UsersListMessage *res_msg;
76:
77:     try
78:     {
79:         res_msg = dynamic_cast<UsersListMessage *>(sw->receiveMsg(USERS_LIST));
80:     }
81:     catch (const char *error_msg)
82:     {
83:         cerr << "Could not connect to server " << host.toString();
84:         cerr << " : " << error_msg << endl;
85:         connected = false;
86:         return "";
87:     }
88:
89:     if (res_msg == NULL)
90:         return "";
91:
92:     string usernames = res_msg->getUsernames();
93:     delete res_msg;
94:
95:     return usernames;
96: }
97:
98: int Server::challengePeer(string username, SecureHost *peerHost)
99: {
100:     ChallengeMessage req_msg(username);
101:
102:     if (sw->sendMsg(&req_msg) != 0)
103:     {
104:         connected = false;
105:         return 1;
106:     }
107:     Message *res_msg;
108:
109:     try
110:     {
111:         MessageType accept_types[] = {GAME_START, GAME_CANCEL};
112:         res_msg = sw->receiveMsg(accept_types, 2);
113:     }
114:     catch (const char *error_msg)
115:     {
116:         cerr << "Could not connect to server " << host.toString();
117:         cerr << " : " << error_msg << endl;
118:         connected = false;
119:         return 1;
120:     }
121:
122:     if (res_msg == NULL)
123:         return 1;
124:
125:     if (res_msg->getType() == GAME_CANCEL)
126:     {
127:         // Game refused
128:         delete res_msg;
129:         return -1;
130:     }
131:     else
132:     {
133:         GameStartMessage *gsm = dynamic_cast<GameStartMessage *>(res_msg);
134:         *peerHost = gsm->getHost();
135:         delete gsm;
136:         return 0;
137:     }
```

```
138: }
139:
140: int Server::replyPeerChallenge(string username, bool response, SecureHost *peerHost, uint1
6_t *listen_port)
141: {
142:     // TODO handle port already busy ?
143:     *listen_port = rand() % (TO_PORT - FROM_PORT + 1) + FROM_PORT;
144:
145:     ChallengeResponseMessage msg(username, response, *listen_port);
146:     if (sw->sendMsg(&msg) != 0)
147:     {
148:         connected = false;
149:         return 1;
150:     }
151:
152:     if (response)
153:     {
154:         Message *res_msg;
155:         try
156:         {
157:             MessageType accept_types[] = {GAME_START, GAME_CANCEL};
158:             res_msg = sw->receiveMsg(accept_types, 2);
159:         }
160:         catch (const char *error_msg)
161:         {
162:             cerr << "Could not connect to server " << host.toString();
163:             cerr << " : " << error_msg << endl;
164:             connected = false;
165:             return 1;
166:         }
167:
168:         if (res_msg == NULL)
169:             return 1;
170:
171:         if (res_msg->getType() == GAME_CANCEL)
172:         {
173:             // Game refused
174:             delete res_msg;
175:             return -1;
176:         }
177:         else
178:         {
179:             GameStartMessage *gsm = dynamic_cast<GameStartMessage *>(res_msg);
180:             *peerHost = gsm->getHost();
181:             delete gsm;
182:             return 0;
183:         }
184:     }
185:     else
186:     {
187:         return -1;
188:     }
189: }
190:
191: int Server::signalGameEnd()
192: {
193:     GameEndMessage msg;
194:     return sw->sendMsg(&msg);
195: }
196:
197: void Server::disconnect()
198: {
199:     connected = false;
200:     sw->closeSocket();
201:     delete sw;
202:     sw = NULL;
203: }
```



```

1: /**
2:  * @file server_lobby.cpp
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Implementation of the function that handles user and network input
6:  *         while the user is in the server lobby waiting for a game to start
7:  *
8:  * The select implementation was inspired by
9:  * https://www.gnu.org/software/libc/manual/html\_node/Server-Example.html
10:  *
11:  * @date 2020-05-29
12:  */
13:
14: #include <sys/select.h>
15: #include <stdio.h>
16: #include <cstring>
17: #include <iostream>
18: #include <sstream>
19:
20: #include "utils/args.h"
21: #include "security/secure_socket_wrapper.h"
22: #include "security/crypto_utils.h"
23:
24: #include "server_lobby.h"
25: #include "server.h"
26:
27: using namespace std;
28:
29: /** stdin has file descriptor 0 in Unix */
30: #define STDIN (0)
31:
32: void printAvailableActions(){
33:     cout<<"You can list users, challenge a user, exit or simply wait for other users to ch
allenge you."<< endl;
34:     cout<<"To list users type: `list`"<< endl;
35:     cout<<"To challenge a user type: `challenge username`"<< endl;
36:     cout<<"To disconnect type: `exit`"<< endl;
37:     cout<<"NB: you cannot receive challenges if you are challenging another user"<< endl;
38: }
39:
40: int doAction(Args args, Server *server, SecureHost* peer_host){
41:     LOG(LOG_DEBUG, "Args: %s", args.c_str());
42:     if (args.getArgc() == 1 && strcmp(args.getArgv(0), "exit") == 0){
43:         return -2;
44:     } else if (args.getArgc() == 1 && strcmp(args.getArgv(0), "list") == 0){
45:         cout<<"Retrieving the list of users..."<<endl;
46:         string userlist = server->getUserList();
47:         if (userlist.empty()){
48:             return 1;
49:         }
50:         cout<<"Online users: "<<userlist<<endl;
51:         return 0;
52:     } else if (args.getArgc() == 2 && strcmp(args.getArgv(0), "challenge") == 0){
53:         cout<<"Sending challenge to "<<args.getArgv(1)<<" and waiting for response..."<<en
dl;
54:         string username(args.getArgv(1));
55:         int ret = server->challengePeer(username, peer_host);
56:         switch (ret){
57:             case -1: // refused
58:                 cout<<username<<" refused your challenge"<<endl;
59:                 return 0;
60:             case 0: //accepted
61:                 cout<<username<<" accepted your challenge"<<endl;
62:                 return -1;
63:             default:
64:                 cout<<"Error connecting to server!"<<endl;
65:                 return 1;
66:         }
67:     } else if (args.getArgc() < 0) {

```

```
68:         return -2; //exit
69:     } else {
70:         return 0;
71:     }
72:
73: }
74:
75: int handleReceivedChallenge(Server *server,
76:                             ChallengeForwardMessage* msg,
77:                             SecureHost* peer_host,
78:                             uint16_t* listen_port){
79:     cout<<endl<<"You received a challenge from "<<msg->getUsername()<<endl;
80:     cout<<"Do you want to accept? (y/n)";
81:
82:     bool response;
83:
84:     do{
85:         cout<<"> "<<flush;
86:         Args args(cin);
87:         LOG(LOG_DEBUG, "Args: %s", args.c_str());
88:         if (args.getArgc() == 1 && strcmp(args.getArgv(0), "y") == 0){
89:             response = true;
90:             break;
91:         } else if (args.getArgc() == 1 && strcmp(args.getArgv(0), "n") == 0){
92:             response = false;
93:             break;
94:         } else if (args.getArgc() < 0){ // EOF
95:             response = false;
96:             break;
97:         } else{
98:             continue;
99:         }
100:     } while(1);
101:
102:
103:     return server->replyPeerChallenge(msg->getUsername(), response, peer_host, listen_port
);
104: }
105:
106: ConnectionMode handleMessage(Message* msg, Server* server){
107:     ChallengeForwardMessage* cfm;
108:     SecureHost peer_host;
109:     uint16_t listen_port;
110:
111:     int ret;
112:     LOG(LOG_INFO, "Server sent message %s", msg->getName().c_str());
113:     switch(msg->getType()){
114:         case CHALLENGE_FWD:
115:             cfm = dynamic_cast<ChallengeForwardMessage*>(msg);
116:             ret = handleReceivedChallenge(server, cfm, &peer_host, &listen_port);
117:             switch (ret){
118:                 case -1: // game canceled
119:                     cout<<"Game was canceled"<<endl;
120:                     return ConnectionMode(CONTINUE);
121:                 case 0:
122:                     cout<<"Starting game..."<<endl;
123:                     return ConnectionMode(WAIT_FOR_PEER, peer_host, listen_port);
124:                 default:
125:                     cout<<"Error"<<endl;
126:                     return ConnectionMode(EXIT, CONNECTION_ERROR);
127:             }
128:             break;
129:         default:
130:             // other messages are handled internally to
131:             // Server since they require the user to wait
132:             LOG(LOG_WARN, "Received unexpected message %s", msg->getName().c_str());
133:             return ConnectionMode(CONTINUE);
134:     }
135: }
```

```
136:
137: ConnectionMode handleStdin(Server* server){
138:     SecureHost peer_host;
139:
140:     int ret;
141:     // Input from user
142:     Args args(cin);
143:     if (args.getArgc() < 0){
144:         ret = -2; // received EOF
145:     } else{
146:         ret = doAction(args, server, &peer_host);
147:     }
148:     switch (ret){
149:         case 0: // do nothing
150:             LOG(LOG_DEBUG, "No action");
151:             return ConnectionMode(CONTINUE);
152:         case 1: // error
153:             cout<<"Error!"<<endl;
154:             return ConnectionMode(EXIT, CONNECTION_ERROR);
155:         case -1: // challenge accepted
156:             cout<<"Starting game..."<<endl;
157:             return ConnectionMode(CONNECT_TO_PEER, peer_host, 0);
158:         case -2:
159:             cout<<"Bye"<<endl;
160:             return ConnectionMode(EXIT, OK);
161:         default:
162:             return ConnectionMode(CONTINUE);
163:     }
164: }
165:
166:
167: ConnectionMode serverLobby(Server* server){
168:     fd_set active_fd_set, read_fd_set;
169:
170:     string username = server->getPlayerUsername();
171:
172:     if (!server->isConnected()){
173:         cout<<"Registering to "<<server->getHost().toString()<<" as "<<username<<endl;
174:         if (server->registerToServer() != 0){
175:             cout<<"Connection to "<<server->getHost().toString()<<" failed!"<<endl;
176:             return ConnectionMode(EXIT, CONNECTION_ERROR);
177:         }
178:         LOG(LOG_INFO, "Server %s is now connected",
179:             server->getHost().toString().c_str());
180:     } else {
181:         LOG(LOG_INFO, "Server %s was already connected",
182:             server->getHost().toString().c_str());
183:     }
184:
185:     SecureHost peer_host;
186:
187:     /* Initialize the set of active sockets. */
188:     FD_ZERO(&active_fd_set);
189:     FD_SET(server->getSocketWrapper()->getDescriptor(), &active_fd_set);
190:     FD_SET(STDIN, &active_fd_set);
191:
192:     printAvailableActions();
193:
194:     while (1){
195:         cout<<endl<<"> "<<flush;
196:
197:         /* Block until input arrives on one or more active sockets. */
198:         read_fd_set = active_fd_set;
199:         if (select(FD_SETSIZE, &read_fd_set, NULL, NULL, NULL) < 0) {
200:             LOG_PERROR(LOG_ERR, "Error in select: %s");
201:             return ConnectionMode(EXIT, GENERIC_ERROR);
202:         }
203:
204:         /* Service all the socketsexit(1); with input pending. */
```

```
205:         for (int i = 0; i < FD_SETSIZE; ++i){
206:             if (FD_ISSET(i, &read_fd_set)){
207:                 if (i == server->getSocketWrapper()->getDescriptor()){
208:                     // Message from server.
209:                     Message* msg;
210:                     try{
211:                         msg = server->getSocketWrapper()->receiveAnyMsg();
212:                     } catch(const char* msg){
213:                         LOG(LOG_ERR, "Error: %s", msg);
214:                         return ConnectionMode(EXIT, CONNECTION_ERROR);
215:                     }
216:
217:                     ConnectionMode m = handleMessage(msg, server);
218:                     if (m.connection_type != CONTINUE){
219:                         return m;
220:                     }
221:                 } else if (i == STDIN){
222:                     ConnectionMode m = handleStdin(server);
223:                     if (m.connection_type != CONTINUE){
224:                         return m;
225:                     }
226:                 }
227:             }
228:         }
229:     }
230: }
231:
```

```
1: /**
2:  * @file crypto.h
3:  * @author Riccardo Mancini
4:  * @brief Header for crypto utilities
5:  * @date 2020-06-07
6:  *
7:  */
8:
9: #ifndef CRYPTO_UTILS_H
10: #define CRYPTO_UTILS_H
11: #include <openssl/conf.h>
12: #include <openssl/evp.h>
13: #include <openssl/err.h>
14: #include <openssl/rand.h>
15: #include <openssl/pem.h>
16: #include <openssl/hmac.h>
17: #include <openssl/kdf.h>
18: #include <string>
19: #include "logging.h"
20: #include <map>
21:
22: using namespace std;
23:
24: #define KEY_BIO_MAX_SIZE 256
25:
26: /**
27:  * Writes the key internal byte representation to the given buffer.
28:  *
29:  * @param key the key to serialize
30:  * @param buf the buffer
31:  * @param buflen the buffer length
32:  * @returns the number of written bytes
33:  */
34: int pkey2buf(EVP_PKEY *key, char* buf, int buflen);
35:
36: /**
37:  * Reads the pkey from the given buffer.
38:  *
39:  * @param buf the buffer
40:  * @param buflen the buffer length
41:  * @param key the key
42:  * @returns 1 in case of success, <=0 otherwise
43:  */
44: int buf2pkey(char* buf, int buflen, EVP_PKEY **key);
45:
46: /**
47:  * Writes the cert internal byte representation to the given buffer.
48:  *
49:  * @param cert the key to serialize
50:  * @param buf the buffer
51:  * @param buflen the buffer length
52:  * @returns the number of written bytes
53:  */
54: int cert2buf(X509 *cert, char* buf, int buflen);
55:
56: /**
57:  * Reads the cert from the given buffer.
58:  *
59:  * @param buf the buffer
60:  * @param buflen the buffer length
61:  * @param cert the key
62:  * @returns 1 in case of success, <=0 otherwise
63:  */
64: int buf2cert(char* buf, int buflen, X509 **cert);
65:
66:
67: /**
68:  * Extracts the username (aka CN) from the given certificate
69:  */
```

```
70: string usernameFromCert(X509* cert);
71:
72: /**
73:  * Builds a map username-certificate from the given directory
74:  *
75:  * This function matches the pattern *_cert.pem inside the directory.
76:  */
77: map<string,X509*> buildCertMapFromDirectory(string dir);
78:
79: #endif // CRYPTO_UTILS_H
```

```
1: #include "security/crypto_utils.h"
2: #include "security/crypto.h"
3: #include "dirent.h"
4:
5: int pkey2buf(EVP_PKEY *key, char* buf, int buflen){
6:     unsigned char* i2dbuff = NULL;
7:     int size = i2d_PUBKEY(key, &i2dbuff);
8:     if(size < 0 ){
9:         handleErrors();
10:        return -1;
11:    }
12:
13:    if(buflen < size){
14:        return -1;
15:    }
16:
17:    memcpy(buf, i2dbuff, size);
18:    OPENSSL_free(i2dbuff);
19:    return size;
20: }
21:
22:
23: int buf2pkey(char* buf, int buflen, EVP_PKEY **key){
24:     const unsigned char **p = (const unsigned char**) &buf;
25:     if (d2i_PUBKEY(key, p, buflen) == NULL){
26:         handleErrors();
27:         return -1;
28:     }
29:     return 1;
30: }
31:
32: int cert2buf(X509 *cert, char* buf, int buflen){
33:     unsigned char* i2dbuff = NULL;
34:     int size = i2d_X509(cert, &i2dbuff);
35:     if(size < 0 ){
36:         handleErrors();
37:         return -1;
38:     }
39:
40:     if(buflen < size){
41:         return -1;
42:     }
43:
44:     memcpy(buf, i2dbuff, size);
45:     OPENSSL_free(i2dbuff);
46:     return size;
47: }
48:
49: int buf2cert(char* buf, int buflen, X509 **cert){
50:     const unsigned char **p = (const unsigned char**) &buf;
51:     if (d2i_X509(cert, p, buflen) == NULL){
52:         handleErrors();
53:         return -1;
54:     }
55:     return 1;
56: }
57:
58: string usernameFromCert(X509* cert){
59:     string username;
60:     X509_NAME* subj_name = X509_get_subject_name(cert);
61:     char* subj_name_cstr = X509_NAME_oneline(subj_name, NULL, 0);
62:
63:     string subj_name_str(subj_name_cstr);
64:     size_t pos = subj_name_str.find("/CN=");
65:     if (pos != string::npos){
66:         username = subj_name_str.substr(pos+4);
67:         LOG(LOG_DEBUG, "%s has size %ld", username.c_str(), username.size());
68:     } else{
69:         LOG(LOG_WARN, "Common name not found in cert: %s", subj_name_str.c_str());
```

```
70:     username = subj_name_str;
71: }
72:
73: free(subj_name_cstr);
74: return username;
75: }
76:
77: map<string,X509*> buildCertMapFromDirectory(string dir_name){
78:     const char* PATTERN = "_cert.pem";
79:     char path[1024]; //should be always big enough
80:     map<string,X509*> cert_map;
81:     DIR *dir;
82:     struct dirent *ent;
83:     if((dir = opendir(dir_name.c_str())) != NULL) {
84:         while((ent = readdir (dir)) != NULL) {
85:             LOG(LOG_DEBUG, "%s", ent->d_name);
86:             if (strstr(ent->d_name, PATTERN) != NULL){
87:                 LOG(LOG_DEBUG, "Match");
88:                 snprintf(path, 1024, "%s/%s", dir_name.c_str(), ent->d_name);
89:                 X509* cert = load_cert_file(path);
90:                 string username = usernameFromCert(cert);
91:                 cert_map.insert(pair<string,X509*>(username, cert));
92:             }
93:         }
94:         closedir(dir);
95:     } else {
96:         LOG(LOG_ERR, "Could not open certificate directory");
97:     }
98:     return cert_map;
99: }
```



```
1: /**
2:  * @file dump_buffer.h
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Utility functions for writing and reading data from a buffer
6:  *
7:  * These functions are buffer-overflow-safe, i.e. they check the remaining
8:  * buffer length before writing/reading. The return is -1 in case of errors,
9:  * the written/read size otherwise.
10:  *
11:  * @date 2020-06-16
12:  */
13:
14: #ifndef BUFFER_IO_H
15: #define BUFFER_IO_H
16:
17: #include <stdlib.h>
18: #include <stdint.h>
19:
20: using namespace std;
21:
22: /**
23:  * Reads a boolean.
24:  *
25:  * @param val the dest value
26:  * @param buf the source buffer
27:  * @param buf_size the size of the buffer
28:  * @returns -1 in case of errors
29:  * @returns 1 number of read bytes
30:  */
31: int readBool(bool *val, char* buf, size_t buf_size);
32:
33: /**
34:  * Writes a boolean.
35:  *
36:  * @param buf the dest buffer
37:  * @param buf_size the size of the buffer
38:  * @param val the source value
39:  * @returns -1 in case of errors
40:  * @returns 1 number of read bytes
41:  */
42: int writeBool(char* buf, size_t buf_size, bool val);
43:
44: /**
45:  * Reads a uint32_t.
46:  *
47:  * @param val the dest value
48:  * @param buf the source buffer
49:  * @param buf_size the size of the buffer
50:  * @returns -1 in case of errors
51:  * @returns 4 number of read bytes
52:  */
53: int readUInt32(uint32_t *val, char* buf, size_t buf_size);
54:
55: /**
56:  * Writes a uint32_t.
57:  *
58:  * @param buf the dest buffer
59:  * @param buf_size the size of the buffer
60:  * @param val the source value
61:  * @returns -1 in case of errors
62:  * @returns 4 number of read bytes
63:  */
64: int writeUInt32(char* buf, size_t buf_size, uint32_t val);
65:
66: /**
67:  * Reads a uint16_t.
68:  *
69:  * @param val the dest value
```

```
70:  * @param buf the source buffer
71:  * @param buf_size the size of the buffer
72:  * @returns -1 in case of errors
73:  * @returns 2 number of read bytes
74:  */
75: int readUInt16(uint16_t *val, char* buf, size_t buf_size);
76:
77: /**
78:  * Writes a uint16_t.
79:  *
80:  * @param buf the dest buffer
81:  * @param buf_size the size of the buffer
82:  * @param val the source value
83:  * @returns -1 in case of errors
84:  * @returns 2 number of read bytes
85:  */
86: int writeUInt16(char* buf, size_t buf_size, uint16_t val);
87:
88: /**
89:  * Reads a uint8_t.
90:  *
91:  * @param val the dest value
92:  * @param buf the source buffer
93:  * @param buf_size the size of the buffer
94:  * @returns -1 in case of errors
95:  * @returns 1 number of read bytes
96:  */
97: int readUInt8(uint8_t *val, char* buf, size_t buf_size);
98:
99: /**
100:  * Writes a uint8_t.
101:  *
102:  * @param buf the dest buffer
103:  * @param buf_size the size of the buffer
104:  * @param val the source value
105:  * @returns -1 in case of errors
106:  * @returns 1 number of read bytes
107:  */
108: int writeUInt8(char* buf, size_t buf_size, uint8_t val);
109:
110:
111: /**
112:  * Reads a char array.
113:  *
114:  * @param val the dest buffer
115:  * @param len number of bytes to read
116:  * @param buf the source buffer
117:  * @param buf_size the size of the buffer
118:  * @returns -1 in case of errors
119:  * @returns len number of read bytes
120:  */
121: int readBuf(char *val, size_t len, char* buf, size_t buf_size);
122:
123: /**
124:  * Writes a char array.
125:  *
126:  * @param buf the dest buffer
127:  * @param buf_size the size of the buffer
128:  * @param val the source buffer
129:  * @param len number of bytes to write
130:  * @returns -1 in case of errors
131:  * @returns len number of read bytes
132:  */
133: int writeBuf(char* buf, size_t buf_size, char* val, size_t len);
134:
135: #endif // BUFFER_IO_H
```

```
1: /**
2:  * @file dump_buffer.h
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Utility functions for writing and reading data from a buffer
6:  *
7:  * @date 2020-06-16
8:  */
9:
10: #ifndef BUFFER_IO_H
11: #define BUFFER_IO_H
12:
13: #include <cstring>
14: #include <stdint.h>
15: #include "utls/buffer_io.h"
16: #include "network/inet_utils.h"
17:
18: using namespace std;
19:
20: int readBool(bool *val, char* buf, size_t buf_size){
21:     if (buf_size < sizeof(bool))
22:         return -1;
23:
24:     *val = (bool) buf[0];
25:     return sizeof(bool);
26: }
27:
28: int writeBool(char* buf, size_t buf_size, bool val){
29:     if (buf_size < sizeof(bool))
30:         return -1;
31:
32:     buf[0] = (char) val;
33:     return sizeof(bool);
34: }
35:
36: int readUInt32(uint32_t *val, char* buf, size_t buf_size){
37:     if (buf_size < sizeof(uint32_t))
38:         return -1;
39:
40:     *val = ntohl(*(uint32_t*) buf);
41:     return sizeof(uint32_t);
42: }
43:
44: int writeUInt32(char* buf, size_t buf_size, uint32_t val){
45:     if (buf_size < sizeof(uint32_t))
46:         return -1;
47:
48:     *((uint32_t*)buf) = htonl(val);
49:     return sizeof(uint32_t);
50: }
51:
52: int readUInt16(uint16_t *val, char* buf, size_t buf_size){
53:     if (buf_size < sizeof(uint16_t))
54:         return -1;
55:
56:     *val = ntohs(*(uint16_t*) buf);
57:     return sizeof(uint16_t);
58: }
59:
60: int writeUInt16(char* buf, size_t buf_size, uint16_t val){
61:     if (buf_size < sizeof(uint16_t))
62:         return -1;
63:
64:     *((uint16_t*)buf) = htons(val);
65:     return sizeof(uint16_t);
66: }
67:
68: int readUInt8(uint8_t *val, char* buf, size_t buf_size){
69:     if (buf_size < sizeof(uint8_t))
```

```
70:         return -1;
71:
72:     *val = (uint8_t) buf[0];
73:     return sizeof(uint8_t);
74: }
75:
76: int writeUInt8(char* buf, size_t buf_size, uint8_t val){
77:     if (buf_size < sizeof(uint8_t))
78:         return -1;
79:
80:     buf[0] = val;
81:     return sizeof(uint8_t);
82: }
83:
84: int readBuf(char *val, size_t len, char* buf, size_t buf_size){
85:     if (buf_size < len)
86:         return -1;
87:
88:     memcpy(val, buf, len);
89:     return len;
90: }
91:
92: int writeBuf(char* buf, size_t buf_size, char* val, size_t len){
93:     if (buf_size < len)
94:         return -1;
95:
96:     memcpy(buf, val, len);
97:     return len;
98: }
99:
100: #endif // BUFFER_IO_H
```

[illegible]

```
69:         } else if (args.getArgc() >= 3 && strcmp(args.getArgv(0), "server") == 0){
70:             X509* cert;
71:             if(args.getArgc() == 4)
72:                 cert = load_cert_file(args.getArgv(3));
73:             else
74:                 cert = NULL;
75:             return ConnectionMode(CONNECT_TO_SERVER, args.getArgv(1),
76:                                   atoi(args.getArgv(2)), cert, 0);
77:
78:         } else if (args.getArgc() == 1 && strcmp(args.getArgv(0), "offline") == 0){
79:             return ConnectionMode(SINGLE_PLAYER);
80:
81:         } else if (args.getArgc() == 1 && strcmp(args.getArgv(0), "exit") == 0){
82:             cout << "Bye" << endl;
83:             return ConnectionMode(EXIT, OK);
84:         } else if (args.getArgc() == 0){
85:             return ConnectionMode(CONTINUE);
86:         } else if (args.getArgc() == -1){ // EOF
87:             cout << "Bye" << endl;
88:             return ConnectionMode(EXIT, OK);
89:         } else{
90:             cout << "Could not parse arguments: " << args << endl;
91:         }
92:     } while (true);
93: }
94:
95:
96: int main(int argc, char** argv){
97:     SecureSocketWrapper *sw = NULL;
98:     Server* server = NULL;
99:
100:    if (argc < 5){
101:        print_help(argv[0]);
102:        return 1;
103:    }
104:
105:    X509* cert = load_cert_file(argv[1]);
106:    X509* cacert = load_cert_file(argv[3]);
107:    X509_CRL* crl = load_crl_file(argv[4]);
108:    X509_STORE* store = build_store(cacert, crl);
109:
110:    srand(time(NULL));
111:
112:    int ret;
113:
114:    printWelcome();
115:    cout<<endl<<"Welcome to 4-in-a-row!"<<endl;
116:    cout<<"The rules of the game are simple: you win when you have 4 connected tokens along
g any direction."<<endl;
117:
118:    EVP_PKEY* key;
119:    do {
120:        key = load_key_file(argv[2], NULL);
121:        if( key == NULL ){
122:            cout<<"Wrong password"<<endl;
123:        }
124:    } while(!key);
125:    cout<<endl;
126:
127:    do{
128:        struct ConnectionMode ucc = promptChooseConnection();
129:
130:        if (ucc.connection_type == EXIT && ucc.exit_code == OK){
131:            exit(0); // Bye
132:        }
133:
134:        if (ucc.connection_type == CONNECT_TO_SERVER){
135:            server = new Server(ucc.host, cert, key, store);
136:        }
```

```
137:
138:     bool loopLobby = true;
139:
140:     do{
141:         try{
142:             if (server != NULL){
143:                 ucc = serverLobby(server);
144:             }
145:
146:             switch(ucc.connection_type){
147:                 case WAIT_FOR_PEER:
148:                     sw = waitForPeer(ucc.listen_port, ucc.host, cert, key, store);
149:                     if (sw != NULL)
150:                         ret = playWithPlayer(MY_TURN, sw);
151:                     else
152:                         ret = CONNECTION_ERROR;
153:
154:                     loopLobby = true;
155:                     break;
156:                 case CONNECT_TO_PEER:
157:                     sw = connectToPeer(ucc.host, cert, key, store);
158:                     if (sw != NULL)
159:                         ret = playWithPlayer(THEIR_TURN, sw);
160:                     else
161:                         ret = CONNECTION_ERROR;
162:
163:                     loopLobby = true;
164:                     break;
165:                 case SINGLE_PLAYER:
166:                     ret = playSinglePlayer();
167:                     loopLobby = false;
168:                     break;
169:                 case EXIT:
170:                     ret = ucc.exit_code;
171:                     loopLobby = false;
172:                     break;
173:                 case CONNECT_TO_SERVER:
174:                     ret = FATAL_ERROR;
175:                     loopLobby = false;
176:                     break;
177:                 case CONTINUE:
178:                     ret = OK;
179:                     loopLobby = true;
180:                     break;
181:             }
182:
183:             if (loopLobby && ret == OK &&
184:                 server != NULL && server->isConnected())
185:             ){
186:                 server->signalGameEnd();
187:             }
188:         } catch(const char* error_msg){
189:             LOG(LOG_ERR, "Caught error: %s", error_msg);
190:             ret = GENERIC_ERROR;
191:             loopLobby = false;
192:         }
193:
194:     } while (loopLobby && server != NULL && server->isConnected());
195:     if (server != NULL){
196:         delete server;
197:         server = NULL;
198:     }
199: } while(ret != FATAL_ERROR);
200:
201: return ret;
202: }
```

```
1: /**
2:  * @file server.cpp
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Implementation of a 4-in-a-row online server
6:  *
7:  * The select implementation was inspired by
8:  * https://www.gnu.org/software/libc/manual/html\_node/Server-Example.html
9:  *
10:  * @date 2020-05-23
11:  */
12: #include <iostream>
13: #include <cstdlib>
14: #include <ctime>
15: #include <pthread.h>
16: #include <map>
17: #include <queue>
18: #include <utility>
19: #include <stdio.h>
20: #include <errno.h>
21: #include <stdlib.h>
22: #include <unistd.h>
23: #include <sys/types.h>
24: #include <sys/socket.h>
25: #include <netinet/in.h>
26: #include <netdb.h>
27:
28: #include "logging.h"
29: #include "config.h"
30: #include "network/socket_wrapper.h"
31: #include "network/host.h"
32:
33: #include "user.h"
34: #include "user_list.h"
35: #include "utils/message_queue.h"
36:
37: #include "security/crypto_utils.h"
38:
39: using namespace std;
40:
41: typedef pair<int, Message*> msgqueue_t;
42: typedef map<string, X509*> cert_map_t;
43:
44: static UserList user_list;
45: static MessageQueue<msgqueue_t, MAX_QUEUE_LENGTH> message_queue;
46: static pthread_t threads[N_THREADS];
47: static cert_map_t cert_map;
48: static X509* cert;
49:
50: void logUnexpectedMessage(User* u, Message* m){
51:     LOG(LOG_WARN, "User %s (state %d) was not expecting a message of type %d",
52:         u->getUsername().c_str(), (int)u->getState(), (int)m->getType());
53: }
54:
55: void doubleLock(User* u_with_lock, User* u_without_lock){
56:     // prevent deadlocks
57:     if (u_without_lock->getUsername() < u_with_lock->getUsername()){
58:         u_with_lock->unlock();
59:         u_without_lock->lock();
60:         u_with_lock->lock();
61:     } else{
62:         u_without_lock->lock();
63:     }
64: }
65:
66: void doubleUnlock(User* u_keep_lock, User* u_unlock){
67:     // prevent deadlocks
68:     if (u_unlock->getUsername() < u_keep_lock->getUsername()){
69:         u_keep_lock->unlock();
```



```
70:         u_unlock->unlock();
71:         u_keep_lock->lock();
72:     } else{
73:         u_unlock->unlock();
74:     }
75: }
76:
77: bool handleRegisterMessage(User* u, RegisterMessage* msg){
78:     string username = msg->getUsername();
79:     string usernameCert = usernameFromCert(u->getSocketWrapper()->getCert());
80:     if (username.compare(usernameCert) == 0){
81:         LOG(LOG_WARN, "Malicious operation: %s tried to register as %s",
82:             usernameCert.c_str(), username.c_str());
83:         return false;
84:     }
85:     u->setUsername(username);
86:
87:     if (!user_list.exists(username)){
88:         u->setState(AVAILABLE);
89:         // readd with username
90:         user_list.add(u);
91:         return true;
92:     } else {
93:         LOG(LOG_WARN, "User %s already registered!", username.c_str());
94:         //TODO send error
95:         u->setState(DISCONNECTED);
96:         return false;
97:     }
98: }
99:
100: bool handleChallengeMessage(User* u, ChallengeMessage* msg){
101:     bool res;
102:     string chlg_username = msg->getUsername();
103:     User* challenged = user_list.get(chlg_username);
104:     if (challenged == NULL || challenged == u){
105:         GameCancelMessage cancel_msg(chlg_username);
106:         return u->getSocketWrapper()->sendMsg(&cancel_msg) == 0;
107:     }
108:
109:     doubleLock(u, challenged);
110:
111:     if (u->getState() != AVAILABLE){
112:         // someone else thing concurrently happened, ignore
113:         doubleUnlock(u, challenged);
114:         user_list.yield(challenged);
115:         return true;
116:     }
117:
118:     if (challenged->getState() != AVAILABLE){
119:         // someone else thing concurrently happened, abort
120:         GameCancelMessage cancel_msg(chlg_username);
121:         res = u->getSocketWrapper()->sendMsg(&cancel_msg) == 0;
122:
123:         doubleUnlock(u, challenged);
124:         user_list.yield(challenged);
125:         return res;
126:     }
127:
128:     // both are available, send challenge
129:     ChallengeForwardMessage fwd_msg(u->getUsername());
130:     if (challenged->getSocketWrapper()->sendMsg(&fwd_msg) == 0){
131:         // challenge sent, mark them as playing until I receive a response
132:         u->setState(CHALLENGED);
133:         u->setOpponent(challenged->getUsername());
134:         challenged->setState(CHALLENGED);
135:         challenged->setOpponent(u->getUsername());
136:         res = true;
137:     } else{
138:         // connection error -> assume disconnected and notify u
```

```
139:         GameCancelMessage cancel_msg(chlg_username);
140:         challenged->setState(DISCONNECTED);
141:         res = u->getSocketWrapper()->sendMsg(&cancel_msg) == 0;
142:     }
143:
144:     doubleUnlock(u, challenged);
145:     user_list.yield(challenged);
146:
147:     return res;
148: }
149:
150: bool handleGameEndMessage(User* u, GameEndMessage* msg){
151:     u->setState(AVAILABLE);
152:     return true;
153: }
154:
155: bool handleUsersListRequestMessage(User* u, UsersListRequestMessage* msg){
156:     UsersListMessage ul_msg(user_list.listAvailableFromTo(msg->getOffset()));
157:     return u->getSocketWrapper()->sendMsg(&ul_msg) == 0;
158: }
159:
160: bool handleChallengeResponseMessage(User* u, ChallengeResponseMessage* msg){
161:     bool res;
162:     User *opponent = user_list.get(u->getOpponent());
163:     if (opponent == NULL || opponent == u){
164:         // opponent disconnected or invalid opponent -> cancel
165:         u->setState(AVAILABLE);
166:         GameCancelMessage cancel_msg(u->getOpponent());
167:         return u->getSocketWrapper()->sendMsg(&cancel_msg) == 0;
168:     }
169:
170:     doubleLock(u, opponent);
171:
172:     if (msg->getResponse()){ // accepted
173:         if (u->getState() != CHALLENGED){
174:             // someother thing concurrently happened, abort
175:             // maybe this refers to old challenge
176:             doubleUnlock(u, opponent);
177:             user_list.yield(opponent);
178:             return true;
179:         }
180:
181:         if (opponent->getState() != CHALLENGED){
182:             // opponent is in wrong state...
183:             // maybe this refers to old challenge
184:             // notify u of opponent not ready
185:             GameCancelMessage cancel_msg(opponent->getUsername());
186:             res = u->getSocketWrapper()->sendMsg(&cancel_msg) == 0;
187:             doubleUnlock(u, opponent);
188:             user_list.yield(opponent);
189:             return res;
190:         }
191:
192:         struct sockaddr_in opp_addr = opponent->getSocketWrapper()
193:             ->getConnectedHost().getAddress();
194:         opp_addr.sin_port = 0;
195:         cert_map_t::iterator opp_pair = cert_map.find(opponent->getUsername());
196:         if(opp_pair == cert_map.end()) {
197:             doubleUnlock(u, opponent);
198:             user_list.yield(opponent);
199:             return false;
200:         }
201:         GameStartMessage msg_to_u(opponent->getUsername(), opp_addr, opp_pair->second);
202:
203:         struct sockaddr_in u_addr = u->getSocketWrapper()
204:             ->getConnectedHost().getAddress();
205:         u_addr.sin_port = htons(msg->getListenPort());
206:         cert_map_t::iterator u_pair = cert_map.find(u->getUsername());
207:         if(u_pair == cert_map.end()) {
```

```
208:         doubleUnlock(u, opponent);
209:         user_list.yield(opponent);
210:         return false;
211:     }
212:     GameStartMessage msg_to_opp(u->getUsername(), u_addr, u_pair->second);
213:
214:     int res_u = u->getSocketWrapper()->sendMsg(&msg_to_u);
215:     int res_opp = opponent->getSocketWrapper()->sendMsg(&msg_to_opp);
216:
217:     if (res_u == 0 && res_opp == 0){
218:         //success
219:         u->setState(PLAYING);
220:         opponent->setState(PLAYING);
221:
222:         res = true;
223:     } else if (res_u != 0 && res_opp != 0){
224:         // both disconnected
225:         opponent->setState(DISCONNECTED);
226:         u->setState(DISCONNECTED);
227:         res = false;
228:     } else {
229:         if (res_u != 0){ // just u disconnected => notify opp
230:             GameCancelMessage cancel_msg(u->getUsername());
231:             if (opponent->getSocketWrapper()->sendMsg(&cancel_msg) == 0){
232:                 opponent->setState(AVAILABLE);
233:             } else {
234:                 opponent->setState(DISCONNECTED);
235:             }
236:         } else if (res_opp != 0){ // just opp disconnected => notify u
237:             GameCancelMessage cancel_msg(opponent->getUsername());
238:             if (u->getSocketWrapper()->sendMsg(&cancel_msg) == 0){
239:                 u->setState(AVAILABLE);
240:                 res = true;
241:             } else {
242:                 u->setState(DISCONNECTED);
243:                 res = false;
244:             }
245:         }
246:     }
247: } else{ // rejected
248:     u->setState(AVAILABLE);
249:     GameCancelMessage cancel_msg(u->getUsername());
250:     if (opponent->getSocketWrapper()->sendMsg(&cancel_msg) == 0){
251:         opponent->setState(AVAILABLE);
252:     } else{
253:         opponent->setState(DISCONNECTED);
254:     }
255: }
256:
257: doubleUnlock(u, opponent);
258: user_list.yield(opponent);
259:
260: return res;
261: }
262:
263: bool handleClientHelloMessage(User* u, ClientHelloMessage* chm){
264:     string username = chm->getMyId();
265:     cert_map_t::iterator res;
266:     SecureSocketWrapper *sw = u->getSocketWrapper();
267:
268:     if ((res = cert_map.find(username)) != cert_map.end()){
269:         sw->setOtherCert(res->second);
270:         int ret = u->getSocketWrapper()->handleClientHello(chm);
271:         return ret == 0;
272:     } else{
273:         LOG(LOG_WARN, "User %s not found in cert_map", username.c_str());
274:         return false;
275:     }
276: }
```

```
277:
278: bool handleClientVerifyMessage(User* u, ClientVerifyMessage* cvm){
279:     int ret = u->getSocketWrapper()->handleClientVerify(cvm);
280:     if(ret == 0){
281:         u->setState(SECURELY_CONNECTED);
282:         return true;
283:     } else {
284:         LOG(LOG_ERR, "Client Verify failed!");
285:         u->setState(DISCONNECTED);
286:         return false;
287:     }
288: }
289:
290: bool handleCertificateRequestMessage(User* u, CertificateRequestMessage* crm){
291:     CertificateMessage cm(cert);
292:     int ret = u->getSocketWrapper()->sendPlain(&cm);
293:     if(ret == 0){
294:         return true;
295:     } else {
296:         LOG(LOG_ERR, "Error sending certificate to client! Error %d", ret);
297:         u->setState(DISCONNECTED);
298:         return false;
299:     }
300: }
301:
302: bool handleMessage(User* user, Message* raw_msg){
303:     bool res = true;
304:
305:     user->lock();
306:
307:     try{
308:
309:         Message* msg = user->getSocketWrapper()->handleMsg(raw_msg);
310:
311:         if (msg == NULL)
312:             return false;
313:
314:         LOG(LOG_INFO, "User %s (state %d) received a message of type %s",
315:             user->getUsername().c_str(), (int) user->getState(), msg->getName().c_str());
316:
317:         switch(user->getState()){
318:             case JUST_CONNECTED:
319:                 switch(msg->getType()){
320:                     case CLIENT_HELLO:
321:                         res = handleClientHelloMessage(user,
322:                             dynamic_cast<ClientHelloMessage*>(msg));
323:                         break;
324:                     case CLIENT_VERIFY:
325:                         res = handleClientVerifyMessage(user,
326:                             dynamic_cast<ClientVerifyMessage*>(msg));
327:                         break;
328:                     case CERT_REQ:
329:                         res = handleCertificateRequestMessage(user,
330:                             dynamic_cast<CertificateRequestMessage*>(msg));
331:                         // TODO: handle cert request
332:                     default:
333:                         logUnexpectedMessage(user, msg);
334:                 }
335:                 break;
336:             case SECURELY_CONNECTED:
337:                 switch(msg->getType()){
338:                     case REGISTER:
339:                         res = handleRegisterMessage(user,
340:                             dynamic_cast<RegisterMessage*>(msg));
341:                         break;
342:                     default:
343:                         logUnexpectedMessage(user, msg);
344:                 }
345:                 break;
```

```
346:         case AVAILABLE:
347:             switch(msg->getType()){
348:                 case CHALLENGE:
349:                     res = handleChallengeMessage(user,
350:                         dynamic_cast<ChallengeMessage*>(msg));
351:                     break;
352:                 case USERS_LIST_REQ:
353:                     res = handleUsersListRequestMessage(user,
354:                         dynamic_cast<UsersListRequestMessage*>(msg));
355:                     break;
356:                 default:
357:                     logUnexpectedMessage(user, msg);
358:             }
359:             break;
360:         case CHALLENGED:
361:             switch(msg->getType()){
362:                 case CHALLENGE_RESP:
363:                     res = handleChallengeResponseMessage(user,
364:                         dynamic_cast<ChallengeResponseMessage*>(msg));
365:                     break;
366:                 default:
367:                     logUnexpectedMessage(user, msg);
368:             }
369:             break;
370:         case PLAYING:
371:             switch(msg->getType()){
372:                 case GAME_END:
373:                     res = handleGameEndMessage(user,
374:                         dynamic_cast<GameEndMessage*>(msg));
375:                     break;
376:                 default:
377:                     logUnexpectedMessage(user, msg);
378:             }
379:             break;
380:         default:
381:             LOG(LOG_ERR, "User %s is in unrecognized state %d",
382:                 user->getUsername().c_str(), (int) user->getState());
383:     }
384:
385:     delete msg;
386: } catch(const char* error_msg){
387:     LOG(LOG_ERR, "Caught error: %s", error_msg);
388:     res = false;
389: }
390:
391: user->unlock();
392: return res;
393: }
394:
395: void* worker(void *args){
396:     while (1){
397:         msgqueue_t p = message_queue.pullWait();
398:         User* u = user_list.get(p.first);
399:         if (u != NULL){
400:             if (!handleMessage(u, p.second)){
401:                 // Connection error -> assume disconnected
402:                 u->setState(DISCONNECTED);
403:             }
404:             user_list.yield(u);
405:         }
406:     }
407: }
408: }
409:
410: void init_threads(){
411:     for (int i=0; i < N_THREADS; i++){
412:         pthread_create(&threads[i], NULL, worker, NULL);
413:     }
414: }
```

```

415:
416: bool checkCertsInCertMap(X509_STORE* store, cert_map_t cert_map){
417:     for (cert_map_t::iterator it = cert_map.begin();
418:          it != cert_map.end();
419:          ++it
420:     ){
421:         if (!verify_peer_cert(store, it->second)){
422:             LOG(LOG_ERR, "Validation failed for certificate in directory: %s",
423:                 it->first.c_str());
424:             return false;
425:         }
426:     }
427:     return true;
428: }
429: }
430:
431: int main(int argc, char** argv){
432:     fd_set active_fd_set, read_fd_set;
433:
434:     if (argc < 7){
435:         cout<<"Usage: "<<argv[0]<<" port cert.pem key.pem cacert.pem crl.pem"<<endl;
436:         exit(1);
437:     }
438:
439:     int port = atoi(argv[1]);
440:     cert = load_cert_file(argv[2]);
441:     EVP_PKEY* key = load_key_file(argv[3], NULL);
442:     X509* cacert = load_cert_file(argv[4]);
443:     X509_CRL* crl = load_crl_file(argv[5]);
444:     X509_STORE* store = build_store(cacert, crl);
445:     cert_map = buildCertMapFromDirectory(argv[6]);
446:
447:     if (cert_map.size() == 0){
448:         LOG(LOG_ERR, "No certificates found in directory");
449:         return 1;
450:     }
451:
452:     if (!checkCertsInCertMap(store, cert_map)){
453:         return 1;
454:     }
455:
456:     LOG(LOG_INFO, "Loaded certificates from %s", argv[6]);
457:
458:     ServerSecureSocketWrapper server_sw(cert, key, store);
459:
460:     int ret = server_sw.bindPort(port);
461:     if (ret != 0){
462:         LOG(LOG_FATAL, "Error binding to port %d", port);
463:         exit(1);
464:     }
465:
466:     LOG(LOG_INFO, "Bound to port %d", port);
467:
468:     init_threads();
469:
470:     LOG(LOG_INFO, "Started %d worker threads", N_THREADS);
471:
472:     /* Initialize the set of active sockets. */
473:     FD_ZERO(&active_fd_set);
474:     FD_SET(server_sw.getDescriptor(), &active_fd_set);
475:
476:     LOG(LOG_INFO, "Polling open sockets");
477:
478:     while (1){
479:         /* Block until input arrives on one or more active sockets. */
480:         read_fd_set = active_fd_set;
481:         if (select(FD_SETSIZE, &read_fd_set, NULL, NULL, NULL) < 0) {
482:             if (errno == EBADF){ // clean closed sockets
483:                 LOG(LOG_DEBUG, "Bad file descriptor");

```

```
484:         for (int i = 0; i < FD_SETSIZE; ++i){
485:             if (FD_ISSET(i, &active_fd_set)){
486:                 if (i != server_sw.getDescriptor()
487:                     && !user_list.exists(i)
488:                 ){
489:                     // user was disconnected but I still need to clear it
490:                     LOG(LOG_DEBUG, "Cleared fd %d", i);
491:                     FD_CLR(i, &active_fd_set);
492:                 }
493:             }
494:         }
495:         continue;
496:     }
497:
498:     LOG_ERROR(LOG_FATAL, "Error in select: %s");
499:     exit(1);
500: }
501:
502: /* Service all the sockets with input pending. */
503: for (int i = 0; i < FD_SETSIZE; ++i){
504:     if (FD_ISSET(i, &read_fd_set)){
505:         if (i == server_sw.getDescriptor()) {
506:             /* Connection request on original socket. */
507:             SecureSocketWrapper* sw = server_sw.acceptClient();
508:
509:             LOG(LOG_INFO, "New connection from %s",
510:                 sw->getConnectedHost().toString().c_str());
511:
512:             FD_SET(sw->getDescriptor(), &active_fd_set);
513:
514:             User *u = new User(sw);
515:             user_list.add(u);
516:         } else {
517:             User *u = user_list.get(i);
518:             if (u->getState() == DISCONNECTED){
519:                 LOG(LOG_DEBUG, "Received message from disconnected user with count
Refs = %d", u->countRefs());
520:                 user_list.yield(u);
521:                 FD_CLR(i, &active_fd_set); // ignore him
522:                 continue;
523:             }
524:             const char* u_addr_str = u->getSocketWrapper()
525:                 ->getConnectedHost().toString().c_str();
526:             LOG(LOG_INFO, "Available message from %s (%s)",
527:                 u->getUsername().c_str(), u_addr_str);
528:             try{
529:                 Message* m = u->getSocketWrapper()->readPartMsg();
530:                 if (m != NULL)
531:                     message_queue.pushSignal(msgqueue_t(i, m));
532:             } catch(const char* msg){
533:                 LOG(LOG_WARN, "Client %s disconnected: %s",
534:                     u_addr_str, msg);
535:                 u->setState(DISCONNECTED);
536:             }
537:             user_list.yield(u);
538:             if (!user_list.exists(i)){
539:                 // user was disconnected -> clear it
540:                 FD_CLR(i, &active_fd_set);
541:             }
542:         }
543:     } else if (FD_ISSET(i, &active_fd_set)){
544:         if (i != server_sw.getDescriptor()
545:             && !user_list.exists(i)
546:         ){
547:             LOG(LOG_DEBUG, "Cleared fd %d", i);
548:             // user was disconnected but I still need to clear it
549:             FD_CLR(i, &active_fd_set);
550:         }
551:     }
```

```
552:         }  
553:     }  
554: }  
555:  
556:
```