```
 1: # Compiler and flags
 2: CC        = gcc
 3: CFLAGS    = -Wall
 4:
 5: # Directories
 6: OBJDIR    = build
 7: SRCDIR    = src
 8: BINDIR    = dist
 9: HDRDIR    = src/include
10: DOCDIR    = doc
11: DOCTMPDIR = build/doc
12:
13: # List of targets
14: UTILS     = fblock tftp_msgs inet_utils debug_utils tftp netascii
15: TARGETS   = tftp_client tftp_server
16:
17: # Documentation output
18: DOCPDFNAME = TFTP_documentation.pdf
19: SRCPDFNAME = source_code.pdf
20:
21: # Documentation config file
22: DOXYGENCFG = doxygen.cfg
23:
24: # Test files
25: TESTS     = 0.txt 4.txt 512.txt 513.txt 62836.txt 131073.txt
26:
27: # Object files for utilities (aka libraries)
28: UTILS_OBJ = $(addsuffix .o, $(addprefix $(OBJDIR)/,$(UTILS)))
29:
30: # Builds only the executables: default rule
31: exe: $(addprefix $(BINDIR)/,$(TARGETS))
32:
33: # Utilities are secondary targets
34: .SECONDARY: $(UTILSOBJ)
35:
36: # Build targets
37: $(BINDIR)/%: $(OBJDIR)/%.o $(UTILS_OBJ) $(HDRDIR)/*.h
38:         $(CC) $(CFLAGS) -o $@ $(filter %.o,$^)
39:
40: # Build generic .o file from .c file
41: $(OBJDIR)/%.o: $(SRCDIR)/%.c $(HDRDIR)/*.h
42:         $(CC) $(CFLAGS) -c $< -o $@
43:
44: # Note that if I modify any header everything is built again
45: # This is not very effective but for such small project that's not an issue
46:
47: # Build documentation pdf
48: $(DOCDIR)/$(DOCPDFNAME): $(SRCDIR)/*.c $(HDRDIR)/*.h $(DOXYGENCFG)
49:         doxygen $(DOXYGENCFG)
50:         ( cd $(DOCTMPDIR)/latex ; make )
51:         cp $(DOCTMPDIR)/latex/refman.pdf $(DOCDIR)/$(DOCPDFNAME)
52:
53:
54: # prepare sorted source list for source code pdf generation
55: # I want the header to appear right before the c source file
56: # the source files of client and server will be last
57: both = $(HDRDIR)/$(1).h $(SRCDIR)/$(1).c
58: ALL_SOURCES = $(foreach x,$(UTILS),$(call both,$(x)))
59: ALL_SOURCES += logging.h
60: ALL_SOURCES += $(addprefix src/,$(addsuffix .c,$(TARGETS)))
61:
62: # Build source code ps file
63: $(OBJDIR)/sources.ps : $(SRCDIR)/*.c $(HDRDIR)/*.h
64:         echo $(ALL_SOURCES)
65:         enscript -C -fCourier9 --highlight=c -p$(OBJDIR)/sources.ps  $(ALL_SOURCES)
66:
67: # Build makefile ps file
68: $(OBJDIR)/makefile.ps: Makefile
69:         enscript -C -fCourier9 --highlight=makefile -p$(OBJDIR)/makefile.ps Makefile
```

```
 70:
 71: $(OBJDIR)/sources.pdf: $(OBJDIR)/sources.ps
 72:         ps2pdf $(OBJDIR)/sources.ps $(OBJDIR)/sources.pdf
 73:
 74: $(OBJDIR)/makefile.pdf: $(OBJDIR)/makefile.ps
 75:         ps2pdf $(OBJDIR)/makefile.ps $(OBJDIR)/makefile.pdf
 76:
 77: # Builds source code pdf file
 78: $(DOCDIR)/$(SRCPDFNAME): $(OBJDIR)/sources.pdf $(OBJDIR)/makefile.pdf
 79:         pdfunite $(OBJDIR)/makefile.pdf $(OBJDIR)/sources.pdf $(DOCDIR)/$(SRCPDFNAME)
 80:
 81: # Builds everything (ecutables and documentation)
 82: all: exe $(DOCDIR)/$(DOCPDFNAME) $(DOCDIR)/$(SRCPDFNAME)
 83:
 84: # clean everything
 85: clean:
 86:         $(RM) -r $(OBJDIR)/* $(BINDIR)/* $(DOCDIR)/$(DOCPDFNAME) $(DOCDIR)/$(SRCPDFNAME)
 87:
 88: # clean everything and then rebuild
 89: rebuild: clean all
 90:
 91: # just opens output documentation
 92: doc_open: $(DOCDIR)/$(DOCPDFNAME)
 93:         xdg-open $(DOCDIR)/$(DOCPDFNAME)
 94:
 95: # generates documentation and opens it in default pdf viewer
 96: doc: $(DOCDIR)/$(DOCPDFNAME) doc_open
 97:
 98: # makes source code pdf and opens it
 99: source: $(DOCDIR)/$(SRCPDFNAME)
100:         xdg-open doc/source_code.pdf
101:
102: # runs tests
103: # 1) removes old test files
104: # 2) runs server in background
105: # 3) downloads files in binary mode
106: # 4) downloads files in text mode
107: # 5) kills the server since it isn't needed anymore
108: # 6) compares bytes in binary outputs with original files
109: # 7) compares text in text outputs with original files
110: # if there are no errors, there should be no output after every "Comparing..." line
111: test: exe
112:         $(RM) test/test_*
113:         dist/tftp_server 9999 test &
114:         for test in $(TESTS); \
115:         do \
116:                 echo "--- $$test ---"; \
117:                 printf "!get $$test test/test_bin_$$test\n!quit\n" | dist/tftp_client 127.
0.0.1 9999; \
118:         done
119:         for test in $(TESTS); \
120:         do \
121:                 echo "--- $$test ---"; \
122:                 printf "!mode txt\n!get $$test test/test_txt_$$test\n!quit\n" | dist/tftp_
client 127.0.0.1 9999; \
123:         done
124:         pkill tftp_server
125:         @for test in $(TESTS); \
126:         do \
127:                 echo "Comparing $$test ($$(stat --printf="%s" test/$$test)) test_bin_$$tes
t ($$(stat --printf="%s" test/test_bin_$$test))"; \
128:                 cmp test/$$test test/test_bin_$$test; \
129:         done
130:         @for test in $(TESTS); \
131:         do \
132:                 echo "Comparing $$test ($$(stat --printf="%s" test/$$test)) test_txt_$$tes
t ($$(stat --printf="%s" test/test_txt_$$test))"; \
133:                 diff test/$$test test/test_txt_$$test; \
134:         done
```

```
135:
136: help:
137:          @echo "all:        builds everything (both binaries and documentation)"
138:          @echo "clean:      deletes any intermediate or output file in build/, dist/ and d
oc/"
139:          @echo "doc:        builds documentation only and opens pdf file"
140:          @echo "doc_open:   opens documentation pdf"
141:          @echo "exe:        builds only binaries"
142:          @echo "help:       shows this message"
143:          @echo "rebuild:    same as calling clean and then all"
144:          @echo "source:     makes source code pdf and opens it"
145:          @echo "test:       runs tests"
146:
147: # these targets aren't name of files
148: .PHONY: all exe clean rebuild doc_open doc test help source
149:
150: # build project structure
151: $(shell   mkdir -p $(SRCDIR) $(HDRDIR) $(DOCDIR) $(OBJDIR) $(BINDIR) test)
```

```
 1: /**
 2:  * @file
 3:  * @author Riccardo Mancini
 4:  *
 5:  * @brief File block read and write.
 6:  *
 7:  * This library provides functions for reading and writing a text or binary
 8:  * file using a predefined block size.
 9:  */
10:
11: #ifndef FBLOCK
12: #define FBLOCK
13:
14:
15: #include <stdio.h>
16:
17:
18: /** Mask for getting text/binary mode */
19: #define FBLOCK_MODE_MASK    0b01
20:
21: /** Open file in text mode */
22: #define FBLOCK_MODE_TEXT    0b00
23:
24: /** Open file in binary mode */
25: #define FBLOCK_MODE_BINARY 0b01
26:
27: /** Mask for getting r/w mode */
28: #define FBLOCK_RW_MASK      0b10
29:
30: /** Open file in read mode */
31: #define FBLOCK_READ         0b00
32:
33: /** Open file in write mode */
34: #define FBLOCK_WRITE        0b10
35:
36:
37: /**
38:  * Structure which defines a file.
39:  */
40: struct fblock{
41:   FILE *file; /**< Pointer to the file */
42:   int block_size;  /**< Predefined block size for i/o operations */
43:   char mode;  /**< Can be read xor write, text xor binary. */
44:   union{
45:     unsigned int written;  /**< Bytes already written (for future use) */
46:     unsigned int remaining;  /**< Remaining bytes to read  */
47:   };
48: };
49:
50:
51: /**
52:  * Opens a file.
53:  *
54:  * @param filename    name of the file
55:  * @param block_size  size of the blocks
56:  * @param mode        mode (read, write, text, binary)
57:  * @return            fblock structure
58:  *
59:  * @see FBLOCK_MODE_TEXT
60:  * @see FBLOCK_MODE_BINARY
61:  * @see FBLOCK_WRITE
62:  * @see FBLOCK_READ
63:  */
64: struct fblock fblock_open(char* filename, int block_size, char mode);
65:
66: /**
67:  * Reads next block_size bytes from file.
68:  *
69:  * @param m_fblock    fblock instance
```

```
70:  * @param buffer      block_size bytes buffer
71:  * @return            0 in case of success, otherwise number of bytes it could
72:  *                    not read.
73:  */
74: int fblock_read(struct fblock *m_fblock, char* buffer);
75:
76: /**
77:  * Writes next block_size bytes to file.
78:  *
79:  * @param m_fblock    fblock instance
80:  * @param buffer      block_size bytes buffer
81:  * @param block_size  if set to a non-0 value, override block_size defined in
82:  *                    fblock.
83:  * @return            0 in case of success, otherwise number of bytes it could
84:  *                    not write.
85:  */
86: int fblock_write(struct fblock *m_fblock, char* buffer, int block_size);
87:
88: /**
89:  * Closes a file.
90:  *
91:  * @param m_fblock    fblock instance to be closed
92:  * @return            0 in case of success, EOF in case of failure
93:  *
94:  * @see fclose
95:  */
96: int fblock_close(struct fblock *m_fblock);
97:
98:
99: #endif
```

```
 1: /**
 2:  * @file
 3:  * @author Riccardo Mancini
 4:  *
 5:  * @brief Implementation of fblock.h.
 6:  *
 7:  * @see fblock.h
 8:  */
 9:
10:
11: #include "include/fblock.h"
12: #include <stdio.h>
13: #include <string.h>
14: #include "include/logging.h"
15:
16:
17: /** LOG_LEVEL will be defined in another file */
18: extern const int LOG_LEVEL;
19:
20:
21: /**
22:  * Returns file length
23:  *
24:  * @param f  file pointer
25:  * @return   file length in bytes
26:  */
27: int get_length(FILE *f){
28:   int size;
29:   fseek(f, 0, SEEK_END); // seek to end of file
30:   size = ftell(f); // get current file pointer
31:   fseek(f, 0, SEEK_SET); // seek back to beginning of file
32:   return size;
33: }
34:
35:
36: struct fblock fblock_open(char* filename, int block_size, char mode){
37:   struct fblock m_fblock;
38:   m_fblock.block_size = block_size;
39:   m_fblock.mode = mode;
40:
41:   char mode_str[4] = "";
42:
43:   LOG(LOG_DEBUG, "Opening file %s (%s %s), block_size = %d",
44:       filename,
45:       (mode & FBLOCK_MODE_MASK) == FBLOCK_MODE_BINARY ? "binary" : "text",
46:       (mode & FBLOCK_RW_MASK) == FBLOCK_WRITE ? "write" : "read",
47:       block_size
48:   );
49:
50:   if ((mode & FBLOCK_RW_MASK) == FBLOCK_WRITE){
51:     strcat(mode_str, "w");
52:     m_fblock.written = 0;
53:   } else {
54:     strcat(mode_str, "r");
55:   }
56:
57:   if ((mode & FBLOCK_MODE_MASK) == FBLOCK_MODE_BINARY)
58:     strcat(mode_str, "b");
59:   // text otherwise
60:
61:   m_fblock.file = fopen(filename, mode_str);
62:   if (m_fblock.file == NULL){
63:     LOG(LOG_ERR, "Error while opening file %s", filename);
64:     return m_fblock;
65:   }
66:   if ((mode & FBLOCK_RW_MASK) == FBLOCK_READ)
67:     m_fblock.remaining = get_length(m_fblock.file);
68:
69:   LOG(LOG_DEBUG, "Successfully opened file");
```

```
 70:    return m_fblock;
 71: }
 72:
 73:
 74: int fblock_read(struct fblock *m_fblock, char* buffer){
 75:    int bytes_read, bytes_to_read;
 76:
 77:    if (m_fblock->remaining > m_fblock->block_size)
 78:      bytes_to_read = m_fblock->block_size;
 79:    else
 80:      bytes_to_read = m_fblock->remaining;
 81:
 82:    bytes_read = fread(buffer, sizeof(char), bytes_to_read, m_fblock->file);
 83:    m_fblock->remaining -= bytes_read;
 84:
 85:    return bytes_to_read - bytes_read;
 86: }
 87:
 88:
 89: int fblock_write(struct fblock *m_fblock, char* buffer, int block_size){
 90:    int written_bytes;
 91:
 92:    if (!block_size)
 93:      block_size = m_fblock->block_size;
 94:
 95:    written_bytes = fwrite(buffer, sizeof(char), block_size, m_fblock->file);
 96:    m_fblock->written += written_bytes;
 97:    return block_size - written_bytes;
 98: }
 99:
100: int fblock_close(struct fblock *m_fblock){
101:    return fclose(m_fblock->file);
102: }
```

```
 1: /**
 2:  * @file
 3:  * @author Riccardo Mancini
 4:  *
 5:  * @brief Contructor for TFTP messages.
 6:  *
 7:  * This library provides functions for building TFTP messages.
 8:  * There are 5 types of messages:
 9:  *  - 1: Read request (RRQ)
10:  *  - 2: Write request (WRQ)
11:  *  - 3: Data (DATA)
12:  *  - 4: Acknowledgment (ACK)
13:  *  - 5: Error (ERROR)
14:  */
15:
16: #ifndef TFTP_MSGS
17: #define TFTP_MSGS
18:
19:
20: /** Read request message type */
21: #define TFTP_TYPE_RRQ   1
22:
23: /** Write request message type */
24: #define TFTP_TYPE_WRQ   2
25:
26: /** Data message type */
27: #define TFTP_TYPE_DATA  3
28:
29: /** Acknowledgment message type */
30: #define TFTP_TYPE_ACK   4
31:
32: /** Error message type */
33: #define TFTP_TYPE_ERROR 5
34:
35: /** String for netascii */
36: #define TFTP_STR_NETASCII "netascii"
37:
38: /** String for octet */
39: #define TFTP_STR_OCTET "octet"
40:
41: /** Maximum filename length (do not defined in RFC) */
42: #define TFTP_MAX_FILENAME_LEN 255
43:
44: /**
45:  * Maximum mode field string length
46:  *
47:  * Since there are only two options: 'netascii' and 'octet', len('netascii') is
48:  * the TFTP_MAX_MODE_LEN.
49:  */
50: #define TFTP_MAX_MODE_LEN 8
51:
52: /** Maximum error message length (do not defined in RFC) */
53: #define TFTP_MAX_ERROR_LEN 255
54:
55: /** Data block size as defined in RFC */
56: #define TFTP_DATA_BLOCK 512
57:
58: /** Data message max size is equal to TFTP_DATA_BLOCK + 4 (header) */
59: #define TFTP_MAX_DATA_MSG_SIZE 516
60:
61:
62: /**
63:  * Retuns msg type given a message buffer.
64:  *
65:  * @param buffer the buffer
66:  * @return message type
67:  *
68:  * @see TFTP_TYPE_RRQ
69:  * @see TFTP_TYPE_WRQ
```

```
 70:  * @see TFTP_TYPE_DATA
 71:  * @see TFTP_TYPE_ACK
 72:  * @see TFTP_TYPE_ERROR
 73:  */
 74: int tftp_msg_type(char *buffer);
 75:
 76:
 77: /**
 78:  * Builds a read request message.
 79:  *
 80:  * ```
 81:  *  2 bytes    string    1 byte    string    1 byte
 82:  *  -------------------------------------------------
 83:  * |   01  | Filename  |   0  |   Mode    |   0  |
 84:  *  -------------------------------------------------
 85:  * ```
 86:  *
 87:  * @param filename  name of the file
 88:  * @param mode      requested transfer mode ("netascii" or "octet")
 89:  * @param buffer    data buffer where to build the message
 90:  */
 91: void tftp_msg_build_rrq(char* filename, char* mode, char* buffer);
 92:
 93: /**
 94:  * Unpacks a read request message.
 95:  *
 96:  * @param buffer      data buffer where the message to read is [in]
 97:  * @param buffer_len  length of the buffer [in]
 98:  * @param filename    name of the file [out]
 99:  * @param mode        requested transfer mode ("netascii" or "octet") [out]
100:  * @return
101:  * - 0 in case of success.
102:  * - 1 in case of wrong operation code.
103:  * - 2 in case of unexpected fields inside message.
104:  * - 3 in case of filename exceeding TFTP_MAX_FILENAME_LEN.
105:  * - 4 in case of mode string exceeding TFTP_MAX_MODE_LEN.
106:  * - 5 in case of unrecognized transfer mode.
107:  *
108:  * @see TFTP_TYPE_RRQ
109:  * @see TFTP_MAX_FILENAME_LEN
110:  * @see TFTP_MAX_MODE_LEN
111:  * @see TFTP_STR_NETASCII
112:  * @see TFTP_STR_OCTET
113:  */
114: int tftp_msg_unpack_rrq(char* buffer, int buffer_len, char* filename,
115:                         char* mode);
116:
117: /**
118:  * Returns size in bytes of a read request message.
119:  *
120:  * @param filename  name of the file
121:  * @param mode      requested transfer mode ("netascii" or "octet")
122:  * @return          size in bytes
123:  */
124: int tftp_msg_get_size_rrq(char* filename, char* mode);
125:
126: /**
127:  * Builds a write request message.
128:  *
129:  * Message format:
130:  * ```
131:  *  2 bytes    string    1 byte    string    1 byte
132:  *  -------------------------------------------------
133:  * |   02  | Filename  |   0  |   Mode    |   0  |
134:  *  -------------------------------------------------
135:  * ```
136:  *
137:  * @param filename  name of the file
138:  * @param mode      requested transfer mode ("netascii" or "octet")
```

```
139:  * @param buffer    data buffer where to build the message
140:  */
141: void tftp_msg_build_wrq(char* filename, char* mode, char* buffer);
142:
143: /**
144:  * Unpacks a write request message.
145:  *
146:  * @param buffer      data buffer where the message to read is [in]
147:  * @param buffer_len  length of the buffer [in]
148:  * @param filename    name of the file [out]
149:  * @param mode        requested transfer mode ("netascii" or "octet") [out]
150:  * @return
151:  * - 0 in case of success.
152:  * - 1 in case of wrong operation code.
153:  * - 2 in case of unexpected fields inside message.
154:  * - 3 in case of filename exceeding TFTP_MAX_FILENAME_LEN.
155:  * - 4 in case of mode string exceeding TFTP_MAX_MODE_LEN.
156:  * - 5 in case of unrecognized transfer mode.
157:  *
158:  * @see TFTP_TYPE_WRQ
159:  * @see TFTP_MAX_FILENAME_LEN
160:  * @see TFTP_MAX_MODE_LEN
161:  * @see TFTP_STR_NETASCII
162:  * @see TFTP_STR_OCTET
163:  */
164: int tftp_msg_unpack_rrq(char* buffer, int buffer_len, char* filename,
165:                         char* mode);
166:
167: /**
168:  * Returns size in bytes of a write request message.
169:  *
170:  * @param filename  name of the file
171:  * @param mode      requested transfer mode ("netascii" or "octet")
172:  * @return          size in bytes
173:  */
174: int tftp_msg_get_size_wrq(char* filename, char* mode);
175:
176: /**
177:  * Builds a data message.
178:  *
179:  * Message format:
180:  * '''
181:  *  2 bytes    2 bytes       n bytes
182:  *  -------------------------------
183:  * | 03    |   Block # |   Data    |
184:  *  -------------------------------
185:  * '''
186:  *
187:  * @param block_n   block sequence number
188:  * @param data      pointer to the buffer containing the data to be transfered
189:  * @param data_size data buffer size
190:  * @param buffer    data buffer where to build the message
191:  */
192: void tftp_msg_build_data(int block_n, char* data, int data_size, char* buffer);
193:
194: /**
195:  * Unpacks a data message.
196:  *
197:  * @param buffer      data buffer where the message to read is [in]
198:  * @param buffer_len  length of the buffer [in]
199:  * @param block_n     pointer where block_n will be written [out]
200:  * @param data        pointer where to copy data [out]
201:  * @return
202:  * - 0 in case of success.
203:  * - 1 in case of wrong operation code.
204:  * - 2 in case of missing fields (packet size is too small).
205:  *
206:  * @see TFTP_TYPE_DATA
207:  */
```

```
208: int tftp_msg_unpack_data(char* buffer, int buffer_len, int* block_n, char* data,
209:                          int* data_size);
210:
211: /**
212:  * Returns size in bytes of a data message.
213:  *
214:  * It just sums 4 to data_size.
215:  *
216:  * @param data_size data buffer size
217:  * @return          size in bytes
218:  */
219: int tftp_msg_get_size_data(int data_size);
220:
221: /**
222:  * Builds an acknowledgment message.
223:  *
224:  * Message format:
225:  * ```
226:  *  2 bytes    2 bytes
227:  *  -------------------
228:  * | 04     |  Block # |
229:  *  -------------------
230:  * ```
231:  *
232:  * @param block_n   block sequence number
233:  * @param buffer    data buffer where to build the message
234:  */
235: void tftp_msg_build_ack(int block_n, char* buffer);
236:
237: /**
238:  * Unpacks an acknowledgment message.
239:  *
240:  * @param buffer     data buffer where the message to read is [in]
241:  * @param buffer_len length of the buffer [in]
242:  * @param block_n    pointer where block_n will be written [out]
243:  * @param data       pointer inside buffer where the data is [out]
244:  * @return
245:  * - 0 in case of success.
246:  * - 1 in case of wrong operation code.
247:  * - 2 in case of wrong packet size.
248:  *
249:  * @see TFTP_TYPE_ACK
250:  */
251: int tftp_msg_unpack_ack(char* buffer, int buffer_len, int* block_n);
252:
253: /**
254:  * Returns size in bytes of an acknowledgment message.
255:  *
256:  * It just returns 4.
257:  *
258:  * @param data_size data buffer size
259:  * @return          size in bytes
260:  */
261: int tftp_msg_get_size_ack();
262:
263: /**
264:  * Builds an error message.
265:  *
266:  * Message format:
267:  * ```
268:  *   2 bytes  2 bytes         string    1 byte
269:  *  ----------------------------------------
270:  * | 05     | ErrorCode |   ErrMsg   |  0  |
271:  *  ----------------------------------------
272:  * ```
273:  *
274:  * Error code meaning:
275:  * - 0: Not defined, see error message (if any).
276:  * - 1: File not found.
```

```
277:  * – 2: Access violation.
278:  * – 3: Disk full or allocation exceeded.
279:  * – 4: Illegal TFTP operation.
280:  * – 5: Unknown transfer ID.
281:  * – 6: File already exists.
282:  * – 7: No such user.
283:  *
284:  * In current implementation only errors 1 and 4 are implemented.
285:  *
286:  * @param error_code error code (from 0 to 7)
287:  * @param error_msg  error message
288:  * @param buffer     data buffer where to build the message
289:  */
290: void tftp_msg_build_error(int error_code, char* error_msg, char* buffer);
291:
292: /**
293:  * Unpacks an error message.
294:  *
295:  * @param buffer      data buffer where the message to read is [in]
296:  * @param buffer_len length of the buffer [in]
297:  * @param error_code pointer where error_code will be written [out]
298:  * @param error_msg  pointer to error message inside the message [out]
299:  * @return
300:  * – 0 in case of success.
301:  * – 1 in case of wrong operation code.
302:  * – 2 in case of unexpected fields.
303:  * – 3 in case of error string exceeding TFTP_MAX_ERROR_LEN.
304:  * – 4 in case of unrecognize error code (must be within 0 and 7).
305:  *
306:  * @see TFTP_TYPE_ERROR
307:  * @see TFTP_MAX_ERROR_LEN
308:  */
309: int tftp_msg_unpack_error(char* buffer, int buffer_len, int* error_code,
310:                           char* error_msg);
311:
312: /**
313:  * Returns size in bytes of an error message.
314:  *
315:  * @param error_msg  error message
316:  * @return           size in bytes
317:  */
318: int tftp_msg_get_size_error(char* error_msg);
319:
320:
321: #endif
```

```
  1: /**
  2:  * @file
  3:  * @author Riccardo Mancini
  4:  *
  5:  * @brief Implementation of tftp_msgs.h .
  6:  *
  7:  * @see tftp_msgs.h
  8:  */
  9:
 10:
 11: #include "include/tftp_msgs.h"
 12: #include "include/logging.h"
 13: #include <string.h>
 14: #include <strings.h>
 15: #include <stdio.h>
 16: #include <arpa/inet.h>
 17: #include <stdint.h>
 18:
 19:
 20: /** LOG_LEVEL will be defined in another file */
 21: extern const int LOG_LEVEL;
 22:
 23:
 24: int tftp_msg_type(char *buffer){
 25:   return (int) ntohs(*(uint16_t*)buffer);
 26: }
 27:
 28:
 29: void tftp_msg_build_rrq(char* filename, char* mode, char* buffer){
 30:   *((uint16_t*)buffer) = htons(TFTP_TYPE_RRQ);
 31:   buffer += 2;
 32:   strcpy(buffer, filename);
 33:   buffer += strlen(filename)+1;
 34:   strcpy(buffer, mode);
 35: }
 36:
 37:
 38: int tftp_msg_unpack_rrq(char* buffer, int buffer_len, char* filename, char* mode){
 39:   int offset = 0;
 40:   if (tftp_msg_type(buffer) != TFTP_TYPE_RRQ){
 41:     LOG(LOG_ERR, "Expected RRQ message (1), found %d", tftp_msg_type(buffer));
 42:     return 1;
 43:   }
 44:
 45:   offset += 2;
 46:   if (strlen(buffer+offset) > TFTP_MAX_FILENAME_LEN){
 47:     LOG(LOG_ERR, "Filename too long (%d > %d): %s",
 48:         (int) strlen(buffer+offset),
 49:         TFTP_MAX_FILENAME_LEN, buffer+offset
 50:     );
 51:     return 3;
 52:   }
 53:   strcpy(filename, buffer+offset);
 54:
 55:   offset += strlen(filename)+1;
 56:   if (strlen(buffer+offset) > TFTP_MAX_MODE_LEN){
 57:     LOG(LOG_ERR, "Mode string too long (%d > %d): %s",
 58:         (int) strlen(buffer+offset),
 59:         TFTP_MAX_MODE_LEN,
 60:         buffer+offset
 61:     );
 62:     return 4;
 63:   }
 64:   strcpy(mode, buffer+offset);
 65:
 66:   offset += strlen(mode)+1;
 67:   if (buffer_len != offset){
 68:     LOG(LOG_ERR, "Packet contains unexpected fields");
 69:     return 2;
```

```
 70:     }
 71:     if (strcasecmp(mode, TFTP_STR_NETASCII) == 0 ||
 72:         strcasecmp(mode, TFTP_STR_OCTET) == 0)
 73:       return 0;
 74:     else{
 75:       LOG(LOG_ERR, "Unrecognized transfer mode: %s", mode);
 76:       return 5;
 77:     }
 78: }
 79:
 80:
 81: int tftp_msg_get_size_rrq(char* filename, char* mode){
 82:   return 4 + strlen(filename) + strlen(mode);
 83: }
 84:
 85:
 86: void tftp_msg_build_wrq(char* filename, char* mode, char* buffer){
 87:   *((uint16_t*)buffer) = htons(TFTP_TYPE_WRQ);
 88:   buffer += 2;
 89:   strcpy(buffer, filename);
 90:   buffer += strlen(filename)+1;
 91:   strcpy(buffer, mode);
 92: }
 93:
 94:
 95: int tftp_msg_unpack_wrq(char* buffer, int buffer_len, char* filename,
 96:                         char* mode){
 97:   int offset = 0;
 98:   if (tftp_msg_type(buffer) != TFTP_TYPE_WRQ){
 99:     LOG(LOG_ERR, "Expected WRQ message (2), found %d", tftp_msg_type(buffer));
100:     return 1;
101:   }
102:
103:   offset += 2;
104:   if (strlen(buffer+offset) > TFTP_MAX_FILENAME_LEN){
105:     LOG(LOG_ERR, "Filename too long (%d > %d): %s",
106:         (int) strlen(buffer+offset),
107:         TFTP_MAX_FILENAME_LEN,
108:         buffer+offset
109:     );
110:     return 3;
111:   }
112:
113:   strcpy(filename, buffer+offset);
114:   offset += strlen(filename)+1;
115:   if (strlen(buffer+offset) > TFTP_MAX_MODE_LEN){
116:     LOG(LOG_ERR, "Mode string too long (%d > %d): %s",
117:         (int) strlen(buffer+offset),
118:         TFTP_MAX_MODE_LEN,
119:         buffer+offset
120:     );
121:     return 4;
122:   }
123:
124:   strcpy(mode, buffer+offset);
125:   offset += strlen(mode)+1;
126:   if (buffer_len != offset){
127:     LOG(LOG_ERR, "Packet contains unexpected fields");
128:     return 2;
129:   }
130:
131:   if (strcmp(mode, TFTP_STR_NETASCII) == 0 || strcmp(mode, TFTP_STR_OCTET) == 0)
132:     return 0;
133:   else{
134:     LOG(LOG_ERR, "Unrecognized transfer mode: %s", mode);
135:     return 5;
136:   }
137: }
138:
```

```
139:
140: int tftp_msg_get_size_wrq(char* filename, char* mode){
141:   return 4 + strlen(filename) + strlen(mode);
142: }
143:
144:
145: void tftp_msg_build_data(int block_n, char* data, int data_size, char* buffer){
146:   *((uint16_t*)buffer) = htons(TFTP_TYPE_DATA);
147:   *((uint16_t*)(buffer+2)) = htons((uint16_t) block_n);
148:   buffer += 4;
149:   memcpy(buffer, data, data_size);
150: }
151:
152:
153: int tftp_msg_unpack_data(char* buffer, int buffer_len, int* block_n, char* data,
154:                          int* data_size){
155:   if (tftp_msg_type(buffer) != TFTP_TYPE_DATA){
156:     LOG(LOG_ERR, "Expected DATA message (3), found %d", tftp_msg_type(buffer));
157:     return 1;
158:   }
159:
160:   if (buffer_len < 4){
161:     LOG(LOG_ERR, "Packet size too small for DATA: %d > 4", buffer_len);
162:     return 2;
163:   }
164:
165:   *block_n = (int) ntohs(*((uint16_t*)(buffer+2)));
166:   *data_size = buffer_len - 4;
167:   if (*data_size > 0)
168:     memcpy(data, buffer+4, *data_size);
169:   return 0;
170: }
171:
172:
173: int tftp_msg_get_size_data(int data_size){
174:   return data_size + 4;
175: }
176:
177:
178: void tftp_msg_build_ack(int block_n, char* buffer){
179:   *((uint16_t*)buffer) = htons(TFTP_TYPE_ACK);
180:   *((uint16_t*)(buffer+2)) = htons((uint16_t) block_n);
181: }
182:
183:
184: int tftp_msg_unpack_ack(char* buffer, int buffer_len, int* block_n){
185:   if (tftp_msg_type(buffer) != TFTP_TYPE_ACK){
186:     LOG(LOG_ERR, "Expected ACK message (4), found %d", tftp_msg_type(buffer));
187:     return 1;
188:   }
189:
190:   if (buffer_len != 4){
191:     LOG(LOG_ERR, "Wrong packet size for ACK: %d != 4", buffer_len);
192:     return 2;
193:   }
194:   *block_n = (int) ntohs(*((uint16_t*)(buffer+2)));
195:   return 0;
196: }
197:
198:
199: int tftp_msg_get_size_ack(){
200:   return 4;
201: }
202:
203:
204: void tftp_msg_build_error(int error_code, char* error_msg, char* buffer){
205:   *((uint16_t*)buffer) = htons(TFTP_TYPE_ERROR);
206:   *((uint16_t*)(buffer+2)) = htons((uint16_t) error_code);
207:   buffer += 4;
```

```
208:    strcpy(buffer, error_msg);
209: }
210:
211:
212: int tftp_msg_unpack_error(char* buffer, int buffer_len, int* error_code,
213:                           char* error_msg){
214:    if (tftp_msg_type(buffer) != TFTP_TYPE_ERROR){
215:        LOG(LOG_ERR, "Expected ERROR message (5), found %d",
216:            tftp_msg_type(buffer)
217:        );
218:        return 1;
219:    }
220:
221:    *error_code = (int) ntohs(*((uint16_t*)(buffer+2)));
222:    if (*error_code < 0 || *error_code > 7){
223:        LOG(LOG_ERR, "Unrecognized error code: %d", *error_code);
224:        return 4;
225:    }
226:
227:    buffer += 4;
228:    if(strlen(buffer) > TFTP_MAX_ERROR_LEN){
229:        LOG(LOG_ERR, "Error string too long (%d > %d): %s",
230:            (int) strlen(buffer),
231:            TFTP_MAX_ERROR_LEN,
232:            buffer
233:        );
234:        return 3;
235:    }
236:
237:    strcpy(error_msg, buffer);
238:    if (buffer_len != strlen(error_msg)+5){
239:        LOG(LOG_WARN, "Packet contains unexpected fields");
240:        return 2;
241:    }
242:    return 0;
243: }
244:
245:
246: int tftp_msg_get_size_error(char* error_msg){
247:    return 5 + strlen(error_msg);
248: }
```

```c
 1: /**
 2:  * @file
 3:  * @author Riccardo Mancini
 4:  *
 5:  * @brief Utility funcions 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:  * @see sockaddr_in
12:  * @see rand
13:  */
14:
15: #ifndef INET_UTILS
16: #define INET_UTILS
17:
18:
19: #include <sys/socket.h>
20: #include <netinet/in.h>
21:
22: /** Random port will be greater or equal to FROM_PORT */
23: #define FROM_PORT 49152
24:
25: /** Random port will be lower or equal to TO_PORT */
26: #define TO_PORT   65535
27:
28: /** Maximum number of trials before giving up opening a random port */
29: #define MAX_TRIES 256
30:
31: /**
32:  * Maximum number of characters of INET address to string
33:  * (eg 123.156.189.123:45678).
34:  */
35: #define MAX_SOCKADDR_STR_LEN 22
36:
37:
38: /**
39:  * Binds socket to a random port.
40:  *
41:  * @param socket    socket ID
42:  * @param addr      inet addr structure
43:  * @return          0 in case of failure, port it could bind to otherwise
44:  *
45:  * @see FROM_PORT
46:  * @see TO_PORT
47:  * @see MAX_TRIES
48:  */
49: int bind_random_port(int socket, struct sockaddr_in *addr);
50:
51: /**
52:  * Makes sockaddr_in structure given ip string and port of server.
53:  *
54:  * @param ip      ip address of server
55:  * @param port    port of the server
56:  * @return        sockaddr_in structure for the given server
57:  */
58: struct sockaddr_in make_sv_sockaddr_in(char* ip, int port);
59:
60: /**
61:  * Makes sockaddr_in structure of this host.
62:  *
63:  * INADDR_ANY is used as IP address.
64:  *
65:  * @param port    port of the server
66:  * @return        sockaddr_in structure this host on given port
67:  */
68: struct sockaddr_in make_my_sockaddr_in(int port);
69:
```

```
70: /**
71:  * Compares INET addresses, returning 0 in case they're equal.
72:  *
73:  * @param sai1  first address
74:  * @param sai2  second address
75:  * @return      0 if thery're equal, 1 otherwise
76:  */
77: int sockaddr_in_cmp(struct sockaddr_in sai1, struct sockaddr_in sai2);
78:
79: /**
80:  * Converts sockaddr_in structure to string to be printed.
81:  *
82:  * @param src   the input address
83:  * @param dst   the output string (must be at least MAX_SOCKADDR_STR_LEN long)
84:  */
85: void sockaddr_in_to_string(struct sockaddr_in src, char *dst);
86:
87:
88: #endif
```

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

```
70:     return 1;
71: }
72:
73: void sockaddr_in_to_string(struct sockaddr_in src, char *dst){
74:    char* port_str;
75:    const char *ret;
76:
77:    port_str = malloc(6);
78:    sprintf(port_str, "%d", ntohs(src.sin_port));
79:
80:    ret = inet_ntop(AF_INET, (void*) &src.sin_addr, dst, MAX_SOCKADDR_STR_LEN);
81:    if (ret != NULL){
82:       strcat(dst, ":");
83:       strcat(dst, port_str);
84:    } else{
85:       strcpy(dst, "ERROR");
86:    }
87:
88:    free(port_str);
89: }
```

```
 1: /**
 2:  * @file
 3:  * @author Riccardo Mancini
 4:  *
 5:  * @brief Utility functions for debugging.
 6:  *
 7:  * At the moment, this library implements only one function for dumping a
 8:  * buffer using hexadecimal.
 9:  */
10:
11: #ifndef DEBUG_UTILS
12: #define DEBUG_UTILS
13:
14:
15: /**
16:  * Prints content of buffer to stdout, showing it as hex values.
17:  *
18:  * @param buffer    pointer to the buffer to be printed
19:  * @param len       the length (in bytes) of the buffer
20:  */
21: void dump_buffer_hex(char* buffer, int len);
22:
23:
24: #endif
```

```
 1: /**
 2:  * @file
 3:  * @author Riccardo Mancini
 4:  *
 5:  * @brief Implementation of debug_utils.h.
 6:  *
 7:  * @see debug_utils.h
 8:  */
 9:
10:
11: #include "include/debug_utils.h"
12: #include "include/logging.h"
13: #include <stdio.h>
14: #include <stdlib.h>
15: #include <string.h>
16:
17:
18: /** LOG_LEVEL will be defined in another file */
19: extern const int LOG_LEVEL;
20:
21:
22: void dump_buffer_hex(char* buffer, int len){
23:   char *str, tmp[4];
24:   int i;
25:
26:   str = malloc(len*3+1);
27:
28:   str[0] = '\0';
29:   for (i=0; i<len; i++){
30:     sprintf(tmp, "%02x ", (unsigned char) buffer[i]);
31:     strcat(str, tmp);
32:   }
33:
34:   LOG(LOG_DEBUG, "%s", str);
35:   free(str);
36: }
```

```
 1: /**
 2:  * @file
 3:  * @author Riccardo Mancini
 4:  *
 5:  * @brief Common functions for TFTP client and server.
 6:  *
 7:  * This library provides functions for sending requests, errors and exchanging
 8:  * files using the TFTP protocol.
 9:  *
10:  * Even though the project assignment does not require the client to send files
11:  * to the server, I still decided to include those functions in a common library
12:  * in case in the future I decide to complete the TFTP implementation.
13:  */
14:
15: #ifndef TFTP
16: #define TFTP
17:
18:
19: #include <sys/socket.h>
20: #include <netinet/in.h>
21: #include "fblock.h"
22:
23: /** Maximum file size to prevent block # overflow */
24: #define TFTP_MAX_FILE_SIZE 33554431
25:
26:
27: /**
28:  * Send a RRQ message to a server.
29:  *
30:  * @param filename  the name of the requested file
31:  * @param mode      the desired mode of transfer (netascii or octet)
32:  * @param sd        socket id of the (UDP) socket to be used to send the message
33:  * @param addr      address of the server
34:  * @return          0 in case of success, 1 otherwise
35:  *
36:  * @see TFTP_STR_NETASCII
37:  * @see TFTP_STR_OCTET
38:  */
39: int tftp_send_rrq(char* filename, char *mode, int sd, struct sockaddr_in *addr);
40:
41: /**
42:  * Send a WRQ message to a server.
43:  *
44:  * Do not used in current implementation.
45:  *
46:  * @param filename  the name of the requested file
47:  * @param mode      the desired mode of transfer (netascii or octet)
48:  * @param sd        socket id of the (UDP) socket to be used to send the message
49:  * @param addr      address of the server
50:  * @return          0 in case of success, 1 otherwise
51:  *
52:  * @see TFTP_STR_NETASCII
53:  * @see TFTP_STR_OCTET
54:  */
55: int tftp_send_wrq(char* filename, char *mode, int sd, struct sockaddr_in *addr);
56:
57: /**
58:  * Send an ERROR message to the client (server).
59:  *
60:  * In current implementation it is only used for sending File Not Found and
61:  * Illegal TFTP Operation errors to clients.
62:  *
63:  * @param error_code the code of the error (must be within 0 and 7)
64:  * @param error_msg  the message explaining the error
65:  * @param sd         socket id of the (UDP) socket to be used to send the
66:  *                   message
67:  * @param addr       address of the client (server)
68:  * @return           0 in case of success, 1 otherwise
69:  */
```

```
 70: int tftp_send_error(int error_code, char* error_msg, int sd,
 71:                     struct sockaddr_in *addr);
 72:
 73: /**
 74:  * Send an ACK message.
 75:  *
 76:  * In current implementation it is only used for sending ACKs from client to
 77:  * server.
 78:  *
 79:  * @param block_n    sequence number of the block to be acknowledged.
 80:  * @param out_buffer buffer to be used for sending the ACK (useful for recycling
 81:  *                   the same buffer)
 82:  * @param sd         socket id of the (UDP) socket to be used to send the
 83:  *                   message
 84:  * @param addr       address of recipient of the ACK
 85:  * @return           0 in case of success, 1 otherwise
 86:  */
 87: int tftp_send_ack(int block_n, char* out_buffer, int sd,
 88:                   struct sockaddr_in *addr);
 89:
 90: /**
 91:  * Handle the entire workflow required to receive a file.
 92:  *
 93:  * In current implementation it is only used in client but it could be also
 94:  * used on the server side, potentially (some tweaks may be needed, though!).
 95:  *
 96:  * @param m_fblock   block file where to write incoming data to
 97:  * @param sd         socket id of the (UDP) socket to be used to send ACK
 98:  *                   messages
 99:  * @param addr       address of the recipient of ACKs
100:  * @return
101:  * - 0 in case of success.
102:  * - 1 in case of file not found.
103:  * - 2 in case of error while sending ACK.
104:  * - 3 in case of unexpected sequence number.
105:  * - 4 in case of an error while unpacking data.
106:  * - 5 in case of an error while unpacking an incoming error message.
107:  * - 6 in case of en error while writing to the file.
108:  * - 7 in case of an error message different from File Not Found (since it is
109:  * the only erorr available in current implementation).
110:  * - 8 in case of the incoming message is neither DATA nor ERROR.
111:  */
112: int tftp_receive_file(struct fblock *m_fblock, int sd,
113:                       struct sockaddr_in *addr);
114:
115: /**
116:  * Receive an ACK message.
117:  *
118:  * In current implementation it is only used for receiving ACKs from client.
119:  *
120:  * @param block_n [out] sequence number of the acknowledged block.
121:  * @param in_buffer     buffer to be used for receiving the ACK (useful for
122:  *                      recycling the same buffer)
123:  * @param sd [in]       socket id of the (UDP) socket to be used to send the
124:  *                      message
125:  * @param addr [in]     address of recipient of the ACK
126:  * @return
127:  * - 0 in case of success
128:  * - 1 in case of failure while receiving the message
129:  * - 2 in case of address and/or port mismatch in sender sockaddr
130:  * - error unpacking ACK message otherwise (8 + result of tftp_msg_unpack_ack)
131:  *
132:  * @see tftp_msg_unpack_ack
133:  */
134: int tftp_receive_ack(int *block_n, char* in_buffer, int sd,
135:                      struct sockaddr_in *addr);
136:
137: /**
138:  * Handle the entire workflow required to send a file.
```

```
139:  *
140:  * In current implementation it is only used in server but it could be also
141:  * used on the client side, potentially (some tweaks may be needed, though!).
142:  *
143:  * @param m_fblock   block file where to read incoming data from
144:  * @param sd         socket id of the (UDP) socket to be used to send DATA
145:  *                   messages
146:  * @param addr       address of the recipient of the file
147:  * @return
148:  * - 0 in case of success.
149:  * - 1 in case of error sending a packet.
150:  * - 2 in case of error while receiving the ack.
151:  * - 3 in case of unexpected sequence number in ack.
152:  * - 4 in case of file too big
153:  */
154: int tftp_send_file(struct fblock *m_fblock, int sd, struct sockaddr_in *addr);
155:
156:
157: #endif
```

```
 1: /**
 2:  * @file
 3:  * @author Riccardo Mancini
 4:  *
 5:  * @brief Implementation of tftp.h.
 6:  *
 7:  * @see tftp.h
 8:  */
 9:
10:
11: #include "include/fblock.h"
12: #include "include/tftp_msgs.h"
13: #include "include/tftp.h"
14: #include "include/debug_utils.h"
15: #include "include/inet_utils.h"
16: #include "include/logging.h"
17: #include <arpa/inet.h>
18: #include <sys/socket.h>
19: #include <netinet/in.h>
20: #include <stdlib.h>
21:
22:
23: /** LOG_LEVEL will be defined in another file */
24: extern const int LOG_LEVEL;
25:
26:
27: int tftp_send_rrq(char* filename, char *mode, int sd, struct sockaddr_in *addr){
28:   int msglen, len;
29:   char *out_buffer;
30:
31:   msglen = tftp_msg_get_size_rrq(filename, mode);
32:   out_buffer = malloc(msglen);
33:
34:   tftp_msg_build_rrq(filename, mode, out_buffer);
35:   len = sendto(sd, out_buffer, msglen, 0,
36:                (struct sockaddr*) addr,
37:                sizeof(*addr)
38:   );
39:   if (len != msglen){
40:     LOG(LOG_ERR, "Error sending RRQ: len (%d) != msglen (%d)", len, msglen);
41:     perror("Error");
42:     return 1;
43:   }
44:
45:   free(out_buffer);
46:   return 0;
47: }
48:
49:
50: int tftp_send_wrq(char* filename, char *mode, int sd, struct sockaddr_in *addr){
51:   int msglen, len;
52:   char *out_buffer;
53:
54:   msglen = tftp_msg_get_size_wrq(filename, mode);
55:   out_buffer = malloc(msglen);
56:
57:   tftp_msg_build_wrq(filename, mode, out_buffer);
58:   len = sendto(sd, out_buffer, msglen, 0,
59:                (struct sockaddr*) addr,
60:                sizeof(*addr)
61:   );
62:   if (len != msglen){
63:     LOG(LOG_ERR, "Error sending WRQ: len (%d) != msglen (%d)", len, msglen);
64:     perror("Error");
65:     return 1;
66:   }
67:
68:   free(out_buffer);
69:   return 0;
```

```
 70: }
 71:
 72:
 73: int tftp_send_error(int error_code, char* error_msg, int sd,
 74:                     struct sockaddr_in *addr){
 75:   int msglen, len;
 76:   char *out_buffer;
 77:
 78:   msglen = tftp_msg_get_size_error(error_msg);
 79:   out_buffer = malloc(msglen);
 80:
 81:   tftp_msg_build_error(error_code, error_msg, out_buffer);
 82:   len = sendto(sd, out_buffer, msglen, 0,
 83:                (struct sockaddr*) addr,
 84:                sizeof(*addr)
 85:   );
 86:   if (len != msglen){
 87:     LOG(LOG_ERR, "Error sending ERROR: len (%d) != msglen (%d)", len, msglen);
 88:     perror("Error");
 89:     return 1;
 90:   }
 91:
 92:   free(out_buffer);
 93:   return 0;
 94: }
 95:
 96:
 97: int tftp_send_ack(int block_n, char* out_buffer, int sd,
 98:                   struct sockaddr_in *addr){
 99:   int msglen, len;
100:
101:   msglen = tftp_msg_get_size_ack();
102:   tftp_msg_build_ack(block_n, out_buffer);
103:   len = sendto(sd, out_buffer, msglen, 0,
104:                (struct sockaddr*) addr,
105:                sizeof(*addr)
106:   );
107:
108:  if (len != msglen){
109:     LOG(LOG_ERR, "Error sending ACK: len (%d) != msglen (%d)", len, msglen);
110:     perror("Error");
111:     return 1;
112:   }
113:
114:   return 0;
115: }
116:
117:
118: int tftp_receive_file(struct fblock *m_fblock, int sd,
119:                       struct sockaddr_in *addr){
120:   char in_buffer[TFTP_MAX_DATA_MSG_SIZE], data[TFTP_DATA_BLOCK], out_buffer[4];
121:   int exp_block_n, rcv_block_n;
122:   int len, data_size, ret, type;
123:   unsigned int addrlen;
124:   struct sockaddr_in cl_addr, orig_cl_addr;
125:
126:   // init expected block number
127:   exp_block_n = 1;
128:
129:   addrlen = sizeof(cl_addr);
130:
131:   do{
132:     LOG(LOG_DEBUG, "Waiting for part %d", exp_block_n);
133:
134:     len = recvfrom(sd, in_buffer, tftp_msg_get_size_data(TFTP_DATA_BLOCK), 0,
135:                    (struct sockaddr*)&cl_addr,
136:                    &addrlen
137:     );
138:
```

```
139:        // first block -> I need to save servers TID (aka its "original" sockaddr)
140:        if (exp_block_n == 1){
141:          char addr_str[MAX_SOCKADDR_STR_LEN];
142:          sockaddr_in_to_string(cl_addr, addr_str);
143:
144:          if (addr->sin_addr.s_addr != cl_addr.sin_addr.s_addr){
145:            LOG(LOG_WARN, "Received message from unexpected source: %s", addr_str);
146:            continue;
147:          } else{
148:            LOG(LOG_INFO, "Receiving packets from %s", addr_str);
149:            orig_cl_addr = cl_addr;
150:          }
151:        } else{
152:          if (sockaddr_in_cmp(orig_cl_addr, cl_addr) != 0){
153:            char addr_str[MAX_SOCKADDR_STR_LEN];
154:            sockaddr_in_to_string(cl_addr, addr_str);
155:            LOG(LOG_WARN, "Received message from unexpected source: %s", addr_str);
156:            continue;
157:          } else{
158:            LOG(LOG_DEBUG, "Sender is the same!");
159:          }
160:        }
161:
162:        type = tftp_msg_type(in_buffer);
163:        if (type == TFTP_TYPE_ERROR){
164:          int error_code;
165:          char error_msg[TFTP_MAX_ERROR_LEN];
166:
167:          ret = tftp_msg_unpack_error(in_buffer, len, &error_code, error_msg);
168:          if (ret != 0){
169:            LOG(LOG_ERR, "Error unpacking error msg");
170:            return 5;
171:          }
172:
173:          if (error_code == 1){
174:            LOG(LOG_INFO, "File not found");
175:            return 1;
176:          } else{
177:            LOG(LOG_ERR, "Received error %d: %s", error_code, error_msg);
178:            return 7;
179:          }
180:
181:        } else if (type != TFTP_TYPE_DATA){
182:          LOG(LOG_ERR, "Received packet of type %d, expecting DATA or ERROR.",type);
183:          return 8;
184:        }
185:
186:        ret = tftp_msg_unpack_data(in_buffer, len, &rcv_block_n, data, &data_size);
187:
188:        if (ret != 0){
189:          LOG(LOG_ERR, "Error unpacking data: %d", ret);
190:          return 4;
191:        }
192:
193:        if (rcv_block_n != exp_block_n){
194:          LOG(LOG_ERR,
195:              "Received unexpected block_n: rcv_block_n = %d != %d = exp_block_n",
196:              rcv_block_n,
197:              exp_block_n
198:          );
199:          return 3;
200:        }
201:
202:        exp_block_n++;
203:
204:        LOG(LOG_DEBUG, "Part %d has size %d", rcv_block_n, data_size);
205:
206:        if (data_size != 0){
207:          if (fblock_write(m_fblock, data, data_size))
```

```
208:          return 6;
209:      }
210:
211:      LOG(LOG_DEBUG, "Sending ack");
212:
213:      if (tftp_send_ack(rcv_block_n, out_buffer, sd, &cl_addr))
214:        return 2;
215:
216:    } while(data_size == TFTP_DATA_BLOCK);
217:    return 0;
218: }
219:
220:
221: int tftp_receive_ack(int *block_n, char* in_buffer, int sd,
222:                      struct sockaddr_in *addr){
223:    int msglen, len, ret;
224:    unsigned int addrlen;
225:    struct sockaddr_in cl_addr;
226:
227:    msglen = tftp_msg_get_size_ack();
228:    addrlen = sizeof(cl_addr);
229:
230:    len = recvfrom(sd, in_buffer, msglen, 0,
231:                   (struct sockaddr*)&cl_addr,
232:                   &addrlen
233:    );
234:
235:    if (sockaddr_in_cmp(*addr, cl_addr) != 0){
236:      char str_addr[MAX_SOCKADDR_STR_LEN];
237:      sockaddr_in_to_string(cl_addr, str_addr);
238:      LOG(LOG_WARN, "Message is coming from unexpected source: %s", str_addr);
239:      return 2;
240:    }
241:
242:    if (len != msglen){
243:      LOG(LOG_ERR, "Error receiving ACK: len (%d) != msglen (%d)", len, msglen);
244:      return 1;
245:    }
246:
247:    ret = tftp_msg_unpack_ack(in_buffer, len, block_n);
248:    if (ret != 0){
249:      LOG(LOG_ERR, "Error unpacking ack: %d", ret);
250:      return 8+ret;
251:    }
252:
253:    return 0;
254: }
255:
256:
257: int tftp_send_file(struct fblock *m_fblock, int sd, struct sockaddr_in *addr){
258:    char in_buffer[4], data[TFTP_DATA_BLOCK], out_buffer[TFTP_MAX_DATA_MSG_SIZE];
259:    int block_n, rcv_block_n;
260:    int len, data_size, msglen, ret;
261:
262:    if (m_fblock->remaining > TFTP_MAX_FILE_SIZE){
263:      LOG(LOG_ERR, "File is too big: %d", m_fblock->remaining);
264:      tftp_send_error(0, "File is too big.", sd, addr);
265:      return 4;
266:    }
267:
268:    // init sequence number
269:    block_n = 1;
270:
271:    do{
272:      LOG(LOG_DEBUG, "Sending part %d", block_n);
273:
274:      if (m_fblock->remaining > TFTP_DATA_BLOCK)
275:        data_size = TFTP_DATA_BLOCK;
276:      else
```

```
277:        data_size = m_fblock->remaining;
278:
279:      if (data_size != 0)
280:        fblock_read(m_fblock, data);
281:
282:      LOG(LOG_DEBUG, "Part %d has size %d", block_n, data_size);
283:
284:      msglen = tftp_msg_get_size_data(data_size);
285:      tftp_msg_build_data(block_n, data, data_size, out_buffer);
286:
287:      // dump_buffer_hex(out_buffer, msglen);
288:
289:      len = sendto(sd, out_buffer, msglen, 0,
290:                   (struct sockaddr*)addr,
291:                   sizeof(*addr)
292:      );
293:
294:      if (len != msglen){
295:        return 1;
296:      }
297:
298:      LOG(LOG_DEBUG, "Waiting for ack");
299:
300:      ret = tftp_receive_ack(&rcv_block_n, in_buffer, sd, addr);
301:
302:      if (ret == 2){  //unexpected source
303:        continue;
304:      } else if (ret != 0){
305:        LOG(LOG_ERR, "Error receiving ack: %d", ret);
306:        return 2;
307:      }
308:
309:      if (rcv_block_n != block_n){
310:        LOG(LOG_ERR, "Received wrong block n: received %d != expected %d",
311:            rcv_block_n,
312:            block_n
313:        );
314:        return 3;
315:      }
316:
317:      block_n++;
318:
319:    } while(data_size == TFTP_DATA_BLOCK);
320:    return 0;
321: }
```

```
 1: /**
 2:  * @file
 3:  * @author Riccardo Mancini
 4:  *
 5:  * @brief Conversion functions from netascii to Unix standard ASCII.
 6:  *
 7:  * This library provides two functions to convert a file from netascii to Unix
 8:  * standard ASCII and viceversa.
 9:  * In particular, there are only two differences:
10:  * - 'LF' in Unix becomes 'CRLF' in netascii
11:  * - 'CR' in Unix becomes 'CRNUL' in netascii
12:  *
13:  * @see https://tools.ietf.org/html/rfc764
14:  */
15:
16:
17: #ifndef NETASCII
18: #define NETASCII
19:
20:
21: /**
22:  * Unix to netascii conversion.
23:  *
24:  * @param unix_filename    the filename of the input Unix file
25:  * @param netascii_filename the filename of the output netascii file
26:  * @return
27:  * - 0 in case of success
28:  * - 1 in case of an error opening unix_filename file
29:  * - 2 in case of an error opening netascii_filename file
30:  * - 3 in case of an error writing to netascii_filename file
31:  */
32: int unix2netascii(char *unix_filename, char* netascii_filename);
33:
34: /**
35:  * Netascii to Unix conversion.
36:  *
37:  * @param netascii_filename the filename of the input netascii file
38:  * @param unix_filename    the filename of the output Unix file
39:  * @return
40:  * - 0 in case of success
41:  * - 1 in case of an error opening unix_filename file
42:  * - 2 in case of an error opening netascii_filename file
43:  * - 3 in case of an error writing to unix_filename file
44:  * - 3 in case of bad formatted netascii
45:  */
46: int netascii2unix(char* netascii_filename, char *unix_filename);
47:
48:
49: #endif
```

```
1: /**
2:  * @file
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Implementation of netascii.h.
6:  *
7:  * @see netascii.h
8:  */
9:
10:
11: #include "include/netascii.h"
12: #include "include/logging.h"
13: #include <stdio.h>
14:
15:
16: /** LOG_LEVEL will be defined in another file */
17: extern const int LOG_LEVEL;
18:
19:
20: int unix2netascii(char *unix_filename, char* netascii_filename){
21:   FILE *unixf, *netasciif;
22:   char prev, tmp;
23:   int ret, result;
24:
25:   unixf = fopen(unix_filename, "r");
26:
27:   if (unixf == NULL){
28:     LOG(LOG_ERR, "Error opening file %s", unix_filename);
29:     return 1;
30:   }
31:
32:   netasciif = fopen(netascii_filename, "w");
33:
34:   if (unixf == NULL){
35:     LOG(LOG_ERR, "Error opening file %s", netascii_filename);
36:     return 2;
37:   }
38:
39:   prev = EOF;
40:
41:   while ((tmp = (char) fgetc(unixf)) != EOF){
42:     if (tmp == '\n' && prev != '\r'){ // LF -> CRLF
43:       ret = putc('\r', netasciif);
44:       if (ret == EOF)
45:         break;
46:
47:       ret = putc('\n', netasciif);
48:       if (ret == EOF)
49:         break;
50:
51:     } else if (tmp == '\r'){  // CR -> CRNUL
52:       char next = (char) fgetc(unixf);
53:       if (next != '\0')
54:         ungetc(next, unixf);
55:
56:       ret = putc('\r', netasciif);
57:       if (ret == EOF)
58:         break;
59:
60:       ret = putc('\0', netasciif);
61:       if (ret == EOF)
62:         break;
63:     } else{
64:       ret = putc(tmp, netasciif);
65:       if (ret == EOF)
66:         break;
67:     }
68:
69:     prev = tmp;
```

```
 70:    }
 71:
 72:    // Error writing to netasciif
 73:    if (ret == EOF){
 74:      LOG(LOG_ERR, "Error writing to file %s", netascii_filename);
 75:      result = 3;
 76:    } else{
 77:      LOG(LOG_INFO, "Unix file %s converted to netascii file %s",
 78:          unix_filename,
 79:          netascii_filename
 80:      );
 81:      result = 0;
 82:    }
 83:
 84:    fclose(unixf);
 85:    fclose(netasciif);
 86:
 87:    return result;
 88: }
 89:
 90: int netascii2unix(char* netascii_filename, char *unix_filename){
 91:    FILE *unixf, *netasciif;
 92:    char tmp;
 93:    int ret;
 94:    int result = 0;
 95:
 96:    unixf = fopen(unix_filename, "w");
 97:
 98:    if (unixf == NULL){
 99:      LOG(LOG_ERR, "Error opening file %s", unix_filename);
100:      return 1;
101:    }
102:
103:    netasciif = fopen(netascii_filename, "r");
104:
105:    if (unixf == NULL){
106:      LOG(LOG_ERR, "Error opening file %s", netascii_filename);
107:      return 2;
108:    }
109:
110:    while ((tmp = (char) fgetc(netasciif)) != EOF){
111:      if (tmp == '\r'){  // CRLF -> LF ; CRNUL -> CR
112:        char next = (char) fgetc(netasciif);
113:        if (next == '\0'){  // CRNUL -> CR
114:         ret = putc('\r', unixf);
115:         if (ret == EOF)
116:          break;
117:        } else if (next == '\n'){  // CRLF -> LF
118:         ret = putc('\n', unixf);
119:         if (ret == EOF)
120:          break;
121:        } else if (next == EOF) { // bad format
122:          LOG(LOG_ERR, "Bad formatted netascii: unexpected EOF after CR");
123:          result = 4;
124:          break;
125:        } else{                   // bad format
126:          LOG(LOG_ERR, "Bad formatted netascii: unexpected 0x%x after CR", next);
127:          result = 4;
128:          break;
129:        }
130:      } else{
131:
132:        // nothing else needs to be done!
133:
134:        ret = putc(tmp, unixf);
135:        if (ret == EOF)
136:          break;
137:      }
138:    }
```

```
139:
140:    if (result == 0){
141:      // Error writing to unixf
142:      if (ret == EOF){
143:        LOG(LOG_ERR, "Error writing to file %s", unix_filename);
144:        result = 3;
145:      } else{
146:        LOG(LOG_INFO, "Netascii file %s converted to Unix file %s",
147:            netascii_filename,
148:            unix_filename
149:        );
150:        result = 0;
151:      }
152:    } // otherwise there was an error (4 or 5) and result was already set
153:
154:    fclose(unixf);
155:    fclose(netasciif);
156:
157:    return result;
158: }
```

```
 1: /**
 2:  * @file
 3:  * @author Riccardo Mancini
 4:  *
 5:  * @brief Implementation of the TFTP client that can only make read requests.
 6:  */
 7:
 8:
 9: #include "include/logging.h"
10: #include "include/tftp_msgs.h"
11: #include "include/tftp.h"
12: #include "include/fblock.h"
13: #include "include/inet_utils.h"
14: #include "include/debug_utils.h"
15: #include "include/netascii.h"
16: #include <arpa/inet.h>
17: #include <sys/types.h>
18: #include <sys/socket.h>
19: #include <netinet/in.h>
20: #include <string.h>
21: #include <stdio.h>
22: #include <stdlib.h>
23: #include <time.h>
24:
25: /** Defining LOG_LEVEL for tftp_client executable */
26: const int LOG_LEVEL = LOG_WARN;
27:
28:
29: /** max stdin line length */
30: #define READ_BUFFER_SIZE 80
31:
32: /** Maximum number of arguments for commands */
33: #define MAX_ARGS 3
34:
35: /** String for txt */
36: #define MODE_TXT "txt"
37:
38: /** String for bin*/
39: #define MODE_BIN "bin"
40:
41:
42: /**
43:  * Global transfer_mode variable for storing user chosen transfer mode string.
44:  *
45:  * @see MODE_TXT
46:  * @see MODE_BIN
47:  */
48: char* transfer_mode;
49:
50:
51: /**
52:  * Splits a string at each delim.
53:  *
54:  * Trailing LF will be removed. Consecutive delimiters will be considered as
55:  * one.
56:  *
57:  * @param line [in]     the string to split
58:  * @param delim [in]    the delimiter
59:  * @param max_argc [in] maximum number of parts to split the line into
60:  * @param argc [out]    counts of the parts the line is split into
61:  * @param argv [out]    array of parts the line is split into
62:  */
63: void split_string(char* line, char* delim, int max_argc, int *argc,
64:                   char **argv){
65:   char *ptr;
66:   int len;/**
67:  * Prints command usage information.
68:  */
69:   char *pos;
```

```
 70:
 71:    // remove trailing LF
 72:    if ((pos=strchr(line, '\n')) != NULL)
 73:      *pos = '\0';
 74:
 75:    // init argc
 76:    *argc = 0;
 77:
 78:    // tokenize string
 79:    ptr = strtok(line, delim);
 80:
 81:          while(ptr != NULL && *argc <= max_argc){
 82:      len = strlen(ptr);
 83:
 84:      if (len == 0)
 85:        continue;
 86:
 87:      LOG(LOG_DEBUG, "arg[%d] = '%s'", *argc, ptr);
 88:
 89:      argv[*argc] = malloc(strlen(ptr)+1);
 90:      strcpy(argv[*argc], ptr);
 91:
 92:                  ptr = strtok(NULL, delim);
 93:      (*argc)++;
 94:          }
 95: }
 96:
 97: /**
 98:  * Prints command usage information.
 99:  */
100: void print_help(){
101:    printf("Usage: ./tftp_client SERVER_IP SERVER_PORT\n");
102:    printf("Example: ./tftp_client 127.0.0.1 69");
103: }
104:
105: /**
106:  * Handles !help command, printing information about available commands.
107:  */
108: void cmd_help(){
109:    printf("Sono disponibili i seguenti comandi:\n");
110:    printf("!help --> mostra l'elenco dei comandi disponibili\n");
111:    printf("!mode {txt|bin} --> imposta il modo di trasferimento ");
112:    printf("dei file (testo o binario)\n");
113:    printf("!get filename nome_locale --> richiede al server il nome del file ");
114:    printf("<filename> e lo salva localmente con il nome <nome_locale>\n");
115:    printf("!quit --> termina il client\n");
116: }
117:
118: /**
119:  * Handles !mode command, changing transfer_mode to either bin or text.
120:  *
121:  * @see transfer_mode
122:  */
123: void cmd_mode(char* new_mode){
124:    if (strcmp(new_mode, MODE_TXT) == 0){
125:      transfer_mode = TFTP_STR_NETASCII;
126:      printf("Modo di trasferimento testo configurato\n");
127:    } else if (strcmp(new_mode, MODE_BIN) == 0){
128:      transfer_mode = TFTP_STR_OCTET;
129:      printf("Modo di trasferimento binario configurato\n");
130:    } else{
131:      printf("Modo di traferimento sconosciuto: %s. Modi disponibili: txt, bin\n",
132:              new_mode
133:      );
134:    }
135: }
136:
137: /**
138:  * Handles !get command, reading file from server.
```

```
139:  */
140:  int cmd_get(char* remote_filename, char* local_filename, char* sv_ip,
141:              int sv_port){
142:    struct sockaddr_in my_addr, sv_addr;
143:    int sd;
144:    int ret, tid, result;
145:    struct fblock m_fblock;
146:    char *tmp_filename;
147:
148:    LOG(LOG_INFO, "Initializing...\n");
149:
150:    sd = socket(AF_INET, SOCK_DGRAM, 0);
151:    if (strcmp(transfer_mode, TFTP_STR_OCTET) == 0)
152:      m_fblock = fblock_open(local_filename,
153:                             TFTP_DATA_BLOCK,
154:                             FBLOCK_WRITE|FBLOCK_MODE_BINARY
155:        );
156:    else if (strcmp(transfer_mode, TFTP_STR_NETASCII) == 0){
157:      tmp_filename = malloc(strlen(local_filename)+5);
158:      strcpy(tmp_filename, local_filename);
159:      strcat(tmp_filename, ".tmp");
160:      m_fblock = fblock_open(tmp_filename,
161:                             TFTP_DATA_BLOCK,
162:                             FBLOCK_WRITE|FBLOCK_MODE_TEXT
163:        );
164:    }else
165:      return 2;
166:
167:    LOG(LOG_INFO, "Opening socket...");
168:
169:    sv_addr = make_sv_sockaddr_in(sv_ip, sv_port);
170:    my_addr = make_my_sockaddr_in(0);
171:    tid = bind_random_port(sd, &my_addr);
172:    if (tid == 0){
173:      LOG(LOG_ERR, "Error while binding to random port");
174:      perror("Could not bind to random port:");
175:      fblock_close(&m_fblock);
176:      return 1;
177:    } else
178:      LOG(LOG_INFO, "Bound to port %d", tid);
179:
180:    printf("Richiesta file %s (%s) al server in corso.\n",
181:           remote_filename,
182:           transfer_mode
183:    );
184:
185:    ret = tftp_send_rrq(remote_filename, transfer_mode, sd, &sv_addr);
186:    if (ret != 0){
187:      fblock_close(&m_fblock);
188:      return 8+ret;
189:    }
190:
191:    printf("Trasferimento file in corso.\n");
192:
193:    ret = tftp_receive_file(&m_fblock, sd, &sv_addr);
194:
195:
196:    if (ret == 1){    // File not found
197:      printf("File non trovato.\n");
198:      result = 0;
199:    } else if (ret != 0){
200:      LOG(LOG_ERR, "Error while receiving file!");
201:      result = 16+ret;
202:    } else{
203:      int n_blocks = (m_fblock.written+m_fblock.block_size-1)/m_fblock.block_size;
204:      printf("Trasferimento completato (%d/%d blocchi)\n", n_blocks, n_blocks);
205:      printf("Salvataggio %s completato.\n", local_filename);
206:
207:      result = 0;
```

```
208:    }
209:
210:    fblock_close(&m_fblock);
211:    if (strcmp(transfer_mode, TFTP_STR_NETASCII) == 0){
212:      netascii2unix(tmp_filename, local_filename);
213:      remove(tmp_filename);
214:      free(tmp_filename);
215:    }
216:
217:    return result;
218:
219: }
220:
221: /**
222:  * Handles !quit command.
223:  */
224: void cmd_quit(){
225:    printf("Client terminato con successo\n");
226:    exit(0);
227: }
228:
229: /** Main */
230: int main(int argc, char** argv){
231:    char* sv_ip;
232:    short int sv_port;
233:    int ret, i;
234:    char read_buffer[READ_BUFFER_SIZE];
235:    int cmd_argc;
236:    char *cmd_argv[MAX_ARGS];
237:
238:    //init random seed
239:    srand(time(NULL));
240:
241:    // default mode = bin
242:    transfer_mode = TFTP_STR_OCTET;
243:
244:    if (argc != 3){
245:      print_help();
246:      return 1;
247:    }
248:
249:    // TODO: check args
250:    sv_ip = argv[1];
251:    sv_port = atoi(argv[2]);
252:
253:    while(1){
254:      printf("> ");
255:      fflush(stdout); // flush stdout buffer
256:      fgets(read_buffer, READ_BUFFER_SIZE, stdin);
257:      split_string(read_buffer, " ", MAX_ARGS, &cmd_argc, cmd_argv);
258:
259:      if (cmd_argc == 0){
260:        printf("Comando non riconosciuto : ''\n");
261:        cmd_help();
262:      } else{
263:        if (strcmp(cmd_argv[0], "!mode") == 0){
264:          if (cmd_argc == 2)
265:            cmd_mode(cmd_argv[1]);
266:          else
267:            printf("Il comando richiede un solo argomento: bin o txt\n");
268:        } else if (strcmp(cmd_argv[0], "!get") == 0){
269:          if (cmd_argc == 3){
270:            ret = cmd_get(cmd_argv[1], cmd_argv[2], sv_ip, sv_port);
271:            LOG(LOG_DEBUG, "cmd_get returned value: %d", ret);
272:          } else{
273:            printf("Il comando richiede due argomenti:");
274:            printf(" <filename> e <nome_locale>\n");
275:          }
276:        } else if (strcmp(cmd_argv[0], "!quit") == 0){
```

```
277:              if (cmd_argc == 1){
278:                cmd_quit();
279:              } else{
280:                 printf("Il comando non richiede argomenti\n");
281:              }
282:          } else if (strcmp(cmd_argv[0], "!help") == 0){
283:              if (cmd_argc == 1){
284:                cmd_help();
285:              } else{
286:                 printf("Il comando non richiede argomenti\n");
287:              }
288:          } else {
289:             printf("Comando non riconosciuto : '%s'\n", cmd_argv[0]);
290:             cmd_help();
291:          }
292:        }
293:
294:       // Free malloc'ed strings
295:       for(i = 0; i < cmd_argc; i++)
296:         free(cmd_argv[i]);
297:     }
298:
299:     return 0;
300: }
```

```
  1: /**
  2:  * @file
  3:  * @author Riccardo Mancini
  4:  *
  5:  * @brief Implementation of the TFTP server that can only handle read requests.
  6:  *
  7:  * The server is multiprocessed, with each process handling one request.
  8:  */
  9:
 10:
 11: #define _GNU_SOURCE
 12: #include <stdlib.h>
 13:
 14: #include "include/tftp_msgs.h"
 15: #include "include/tftp.h"
 16: #include "include/fblock.h"
 17: #include "include/inet_utils.h"
 18: #include "include/debug_utils.h"
 19: #include "include/netascii.h"
 20: #include <arpa/inet.h>
 21: #include <sys/types.h>
 22: #include <sys/socket.h>
 23: #include <netinet/in.h>
 24: #include <string.h>
 25: #include <strings.h>
 26: #include <stdio.h>
 27: #include "include/logging.h"
 28: #include <sys/types.h>
 29: #include <unistd.h>
 30: #include <time.h>
 31: #include <linux/limits.h>
 32: #include <libgen.h>
 33:
 34:
 35: /** Defining LOG_LEVEL for tftp_server executable */
 36: const int LOG_LEVEL = LOG_INFO;
 37:
 38:
 39: /** Maximum length for a RRQ message */
 40: #define MAX_MSG_LEN TFTP_MAX_MODE_LEN+TFTP_MAX_FILENAME_LEN+4
 41:
 42:
 43: /** Finds longest common prefix length of strings str1 and str2 */
 44: int strlcpl(const char* str1, const char* str2){
 45:   int n;
 46:   for (n = 0; str1[n] != '\0' && str2[n] != '\0' && str1[n] == str2[n]; n++);
 47:   return n;
 48: }
 49:
 50: /**
 51:  * Check whether file is inside dir.
 52:  *
 53:  * @param path  file absolute path (can include .. and .  and multiple /)
 54:  * @param dir   directory real path (can't include .. and . and multiple /)
 55:  * @return      1 if true, 0 otherwise
 56:  *
 57:  * @see realpath
 58:  */
 59: int path_inside_dir(char* path, char* dir){
 60:   char *parent, *orig_parent, *ret_realpath;
 61:   char parent_realpath[PATH_MAX];
 62:   int result;
 63:
 64:   orig_parent = parent = malloc(strlen(path) + 1);
 65:   strcpy(parent, path);
 66:
 67:   do{
 68:     parent = dirname(parent);
 69:     ret_realpath = realpath(parent, parent_realpath);
```

```
 70:    } while (ret_realpath == NULL);
 71:
 72:    if (strlcpl(parent_realpath, dir) < strlen(dir))
 73:      result = 0;
 74:    else
 75:      result = 1;
 76:
 77:    free(orig_parent);
 78:    return result;
 79: }
 80:
 81: /**
 82:  * Prints command usage information.
 83:  */
 84: void print_help(){
 85:    printf("Usage: ./tftp_server LISTEN_PORT FILES_DIR\n");
 86:    printf("Example: ./tftp_server 69 .\n");
 87: }
 88:
 89: /**
 90:  * Sends file to a client.
 91:  */
 92: int send_file(char* filename, char* mode, struct sockaddr_in *cl_addr){
 93:    struct sockaddr_in my_addr;
 94:    int sd;
 95:    int ret, tid, result;
 96:    struct fblock m_fblock;
 97:    char *tmp_filename;
 98:
 99:    sd = socket(AF_INET, SOCK_DGRAM, 0);
100:    my_addr = make_my_sockaddr_in(0);
101:    tid = bind_random_port(sd, &my_addr);
102:    if (tid == 0){
103:      LOG(LOG_ERR, "Could not bind to random port");
104:      perror("Could not bind to random port:");
105:      fblock_close(&m_fblock);
106:      return 4;
107:    } else
108:      LOG(LOG_INFO, "Bound to port %d", tid);
109:
110:    if (strcasecmp(mode, TFTP_STR_OCTET) == 0){
111:      m_fblock = fblock_open(filename,
112:                            TFTP_DATA_BLOCK,
113:                            FBLOCK_READ|FBLOCK_MODE_BINARY
114:      );
115:    } else if (strcasecmp(mode, TFTP_STR_NETASCII) == 0){
116:      tmp_filename = malloc(strlen(filename)+5);
117:      strcpy(tmp_filename, filename);
118:      strcat(tmp_filename, ".tmp");
119:      ret = unix2netascii(filename, tmp_filename);
120:      if (ret != 0){
121:        LOG(LOG_ERR, "Error converting text file to netascii: %d", ret);
122:        return 3;
123:      }
124:      m_fblock = fblock_open(tmp_filename,
125:                            TFTP_DATA_BLOCK,
126:                            FBLOCK_READ|FBLOCK_MODE_TEXT
127:      );
128:    } else{
129:      LOG(LOG_ERR, "Unknown mode: %s", mode);
130:      return 2;
131:    }
132:
133:    if (m_fblock.file == NULL){
134:      LOG(LOG_WARN, "Error opening file. Not found?");
135:      tftp_send_error(1, "File not found.", sd, cl_addr);
136:      result = 1;
137:    } else{
138:      LOG(LOG_INFO, "Sending file...");
```

```
139:       ret = tftp_send_file(&m_fblock, sd, cl_addr);
140:
141:       if (ret != 0){
142:         LOG(LOG_ERR, "Error sending file: %d", ret);
143:         result = 16+ret;
144:       } else{
145:         LOG(LOG_INFO, "File sent successfully");
146:         result = 0;
147:       }
148:     }
149:
150:     fblock_close(&m_fblock);
151:
152:     if (strcasecmp(mode, TFTP_STR_NETASCII) == 0){
153:       LOG(LOG_DEBUG, "Removing temp file %s", tmp_filename);
154:       remove(tmp_filename);
155:       free(tmp_filename);
156:     }
157:
158:     return result;
159: }
160:
161: /** Main */
162: int main(int argc, char** argv){
163:     short int my_port;
164:     char *dir_rel_path;
165:     char *ret_realpath;
166:     char dir_realpath[PATH_MAX];
167:     int ret, type, len;
168:     char in_buffer[MAX_MSG_LEN];
169:     unsigned int addrlen;
170:     int sd;
171:     struct sockaddr_in my_addr, cl_addr;
172:     int pid;
173:     char addr_str[MAX_SOCKADDR_STR_LEN];
174:
175:     if (argc != 3){
176:       print_help();
177:       return 1;
178:     }
179:
180:     my_port = atoi(argv[1]);
181:     dir_rel_path = argv[2];
182:
183:     ret_realpath = realpath(dir_rel_path, dir_realpath);
184:     if (ret_realpath == NULL){
185:       LOG(LOG_FATAL, "Directory not found: %s", dir_rel_path);
186:       return 1;
187:     }
188:
189:     addrlen = sizeof(cl_addr);
190:
191:     sd = socket(AF_INET, SOCK_DGRAM, 0);
192:     my_addr = make_my_sockaddr_in(my_port);
193:     ret = bind(sd, (struct sockaddr*) &my_addr, sizeof(my_addr));
194:     if (ret == -1){
195:       perror("Could not bind: ");
196:       LOG(LOG_FATAL, "Could not bind to port %d", my_port);
197:       return 1;
198:     }
199:
200:     LOG(LOG_INFO, "Server is running");
201:
202:     while (1){
203:       len = recvfrom(sd, in_buffer, MAX_MSG_LEN, 0,
204:                      (struct sockaddr*)&cl_addr,
205:                      &addrlen
206:       );
207:       type = tftp_msg_type(in_buffer);
```

```
208:        sockaddr_in_to_string(cl_addr, addr_str);
209:        LOG(LOG_INFO, "Received message with type %d from %s", type, addr_str);
210:        if (type == TFTP_TYPE_RRQ){
211:          pid = fork();
212:          if (pid == -1){ // error
213:            LOG(LOG_FATAL, "Fork error");
214:            perror("Fork error:");
215:            return 1;
216:          } else if (pid != 0 ){  // father
217:            LOG(LOG_INFO, "Received RRQ, spawned new process %d", (int) pid);
218:            continue; // father process continues loop
219:          } else{        // child
220:            char filename[TFTP_MAX_FILENAME_LEN], mode[TFTP_MAX_MODE_LEN];
221:            char file_path[PATH_MAX], file_realpath[PATH_MAX];
222:
223:            //init random seed
224:            srand(time(NULL));
225:
226:            ret = tftp_msg_unpack_rrq(in_buffer, len, filename, mode);
227:
228:            if (ret != 0){
229:              LOG(LOG_WARN, "Error unpacking RRQ");
230:              tftp_send_error(0, "Malformed RRQ packet.", sd, &cl_addr);
231:              break; // child process exits loop
232:            }
233:
234:            strcpy(file_path, dir_realpath);
235:            strcat(file_path, "/");
236:            strcat(file_path, filename);
237:
238:            // check if file is inside directory (or inside any of its subdirs)
239:            if (!path_inside_dir(file_path, dir_realpath)){
240:              // it is not! I caught you, Trudy!
241:              LOG(LOG_WARN, "User tried to access file %s outside set directory %s",
242:                  file_realpath,
243:                  dir_realpath
244:              );
245:
246:              tftp_send_error(4, "Access violation.", sd, &cl_addr);
247:              break; // child process exits loop
248:            }
249:
250:            ret_realpath = realpath(file_path, file_realpath);
251:
252:            // file not found
253:            if (ret_realpath == NULL){
254:              LOG(LOG_WARN, "File not found: %s", file_path);
255:              tftp_send_error(1, "File Not Found.", sd, &cl_addr);
256:              break; // child process exits loop
257:            }
258:
259:            LOG(LOG_INFO, "User wants to read file %s in mode %s", filename, mode);
260:
261:            ret = send_file(file_realpath, mode, &cl_addr);
262:            if (ret != 0)
263:              LOG(LOG_WARN, "Write terminated with an error: %d", ret);
264:            break;  // child process exits loop
265:          }
266:        } else{
267:          LOG(LOG_WARN, "Wrong op code: %d", type);
268:          tftp_send_error(4, "Illegal TFTP operation.", sd, &cl_addr);
269:          // main process continues loop
270:        }
271:    }
272:
273:    LOG(LOG_INFO, "Exiting process %d", (int) getpid());
274:    return 0;
275: }
```