COSC 264 Assignment

Jesse Sheehan (Student ID: 53366509)

August 21, 2018

I decided to use C in this assignment so I could practice it for ENCE260. Also I think it's a bit more fun to program in C.

Towards the end of the assignment I became aware of C's struct and enum language features. If I were to redo this project I would use structs to represent the DT-Request and DT-Response packets and enums to represent things like language codes, packet request type, etc.

I have been very liberal in the use of comments and have used Javadoc-style comments for providing more information about functions. This, unfortunately, makes the source code quite a long read (table 1).

Language	Files	Blank	Comment	Code
C	5	240	387	775
C Header	4	22	10	60
Total	9	262	397	835

Table 1: A lines-of-code breakdown of the C source.

Contents

Plagiarism Declaration

This form needs to accompany your COSC 264 assignment submission.

I understand that plagiarism means taking someone else's work (text, program code, ideas, concepts) and presenting them as my own, without proper attribution. Taking someone else's work can include verbatim copying of text, figures/images, or program code, or it can refer to the extensive use of someone else's original ideas, algorithms or concepts.

I hereby declare that:

- My assignment is my own original work. I have not reproduced or modified code, figures/images, or writings of others without proper attribution. I have not used original ideas and concepts of others and presented them as my own.
- I have not allowed others to copy or modify my own code, figures/images, or writings. I have not allowed others to use original ideas and concepts of mine and present them as their own.
- I accept that plagiarism can lead to consequences, which can include partial or total loss of marks, no grade being awarded and other serious consequences, including notification of the University Proctor.

Name: Jesse Sheehan

Student ID: 53366509

Signature: 

Date: 14/08/18

2 Source Code Listings

2.1 Utilities

Contains several general helper functions.

```c
// utils.h

#ifndef UTILS_H
#define UTILS_H

void fail(char funcname[], char condition[]);
void error(char message[], int code);
void printCurrentDateTimeString();
int max(int nums[], int n);

#endif
```

```c
// utils.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

/**
 * Prints an error message.
 *
 * @param funcname The name of the function where the error occurred.
 * @param condition A description of what failed.
 * */
void fail(char function_name[], char condition[])
{
    fprintf(stderr, "Failure: %s - %s\n", function_name, condition);
}

/**
 * Prints an error message then exits.
 *
 * @param message The message to print.
 * @param code The exit code.
 * */
void error(char message[], int code)
{
    fprintf(stderr, "*** Error: %s ***\n", message);
    exit(code);
}

/**
 * Returns the current time string without a newline character
 * @return A pointer to the datetime string.
 * */
void printCurrentDateTimeString()
{
    // used to store the time
    time_t rawtime;
```

```
39      struct tm* info;
40
41      // used to store the string
42      char str[32] = {0};
43
44      // get the local time
45      time(&rawtime);
46      info = localtime(&rawtime);
47
48      // format and print the date time string
49      strftime(str, 32, "%F %H:%M:%S", info);
50      printf("%s", str);
51  }
52
53  /**
54   * Returns the largest value in the array.
55   *
56   * @param nums The array to iterate through.
57   * @param n The length of the array
58   * */
59  int max(int nums[], int n)
60  {
61      int largest;
62      for (int i = 0; i < n; i++) {
63          if (i == 0 || nums[i] > largest) {
64              largest = nums[i];
65          }
66      }
67      return largest;
68  }
```

2.2 Protocol

Contains functions and definitions that are relevant to both the client and the server, and more specifically, the protocol.

```c
// protocol.h

#ifndef PROTOCOL_H
#define PROTOCOL_H

#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>

// Protocol definitions
#define MAGIC_NO 0x497E
#define PACKET_REQ 0x0001
#define PACKET_RES 0x0002

#define MIN_PORT_NO 1024
#define MAX_PORT_NO 64000

#define REQ_DATE 0x0001
#define REQ_TIME 0x0002
#define REQ_PKT_LEN 6

#define RES_TEXT_LEN 255
#define RES_PKT_LEN (13 + RES_TEXT_LEN)

// Language code definitions
#define LANG_ENG 0x0001
#define LANG_MAO 0x0002
#define LANG_GER 0x0003

// Helper functions
bool validLangCode(uint16_t langCode);
bool validReqType(uint16_t reqType);
char* getLangName(uint16_t langCode);
char* getRequestTypeString(uint16_t reqType);

// General packet functions
uint16_t dtPktMagicNo(uint8_t pkt[], size_t n);
uint16_t dtPktType(uint8_t pkt[], size_t n);
void dtPktDump(uint8_t pkt[]);
size_t dtPktLength(uint8_t pkt[]);

// DT Request functions
size_t dtReq(uint8_t pkt[], size_t n, uint16_t reqType);
uint16_t dtReqType(uint8_t pkt[], size_t n);
bool dtReqValid(uint8_t pkt[], size_t n);

// DT Response functions
size_t dtRes(uint8_t pkt[], size_t n, uint16_t reqType, uint16_t langCode, uint16_t
↪  year, uint8_t month, uint8_t day, uint8_t hour, uint8_t minute);
size_t dtResNow(uint8_t pkt[], size_t n, uint16_t reqType, uint16_t langCode);
bool dtResValid(uint8_t pkt[], size_t n);
```

```
uint16_t dtResLangCode(uint8_t pkt[], size_t n);
uint16_t dtResYear(uint8_t pkt[], size_t n);
uint8_t dtResMonth(uint8_t pkt[], size_t n);
uint8_t dtResDay(uint8_t pkt[], size_t n);
uint8_t dtResHour(uint8_t pkt[], size_t n);
uint8_t dtResMinute(uint8_t pkt[], size_t n);
uint8_t dtResLength(uint8_t pkt[], size_t n);
void dtResText(uint8_t pkt[], size_t n, char text[], size_t* textLen);

#endif
```

```
// protocol.c

#include <stdio.h>
#include <ctype.h>
#include <time.h>

#include "protocol.h"

// The phrases to send as a response. Written as templates to be filled with sprintf.
const char* PHRASES[3][2] = {
    { "Today's date is %s %02u, %04u", "The current time is %02u:%02u" },
    { "Ko te ra o tenei ra ko %s %02u, %04u", "Ko te wa o tenei wa %02u:%02u" },
    { "Heute ist der %02u. %s %04u", "Die Uhrzeit ist %02u:%02u" }
};

// The names of the months as strings. Some UTF codes are required for Maori and
↪  German.
const char* MONTHS[3][12] = {
    { "January", "February", "March", "April", "May", "June",
        "July", "August", "September", "October", "November", "December"},
    { "Kohit\u0101tea", "Hui-tanguru", "Pout\u016B-te-rangi",
    ↪  "Paenga-wh\u0101wh\u0101", "Haratua", "Pipiri",
        "H\u014Dngongoi", "Here-turi-k\u014Dk\u0101", "Mahuru", "Whiringa-\u0101-nuku",
        ↪  "Whiringa-\u0101-rangi", "Hakihea" },
    { "Januar", "Februar", "M\u00E4rz", "April", "Mai", "Juni",
        "Juli", "August", "September", "Oktober", "November", "Dezember" }
};

/**
 * Creates a DT Request packet and puts it into a uint8_t array.
 *
 * The packet array must be REQ_PKT_LEN long otherwise
 * you risk a buffer overflow.
 *
 * @param pkt A pointer to the packet.
 * @param n The size of the array. Must be REQ_PKT_LEN.
 * @param reqType Must be REQ_DATE or REQ_TIME.
 * @return The length of the packet.
 * */
size_t dtReq(uint8_t pkt[], size_t n, uint16_t reqType)
{

    if (validReqType(reqType) && n == REQ_PKT_LEN) {

```

```
42          pkt[0] = (uint8_t)(MAGIC_NO >> 8);
43          pkt[1] = (uint8_t)(MAGIC_NO & 0xFF);
44          pkt[2] = (uint8_t)(PACKET_REQ >> 8);
45          pkt[3] = (uint8_t)(PACKET_REQ & 0xFF);
46          pkt[4] = (uint8_t)(reqType >> 8);
47          pkt[5] = (uint8_t)(reqType & 0xFF);
48
49          return n;
50
51      }
52
53      return 0;
54  }
55
56  /**
57   * Returns the request type of a DT Request packet.
58   * No checking is performed beforehand.
59   *
60   * @param pkt An array of uint8 values making up the packet.
61   * @param n The size of the array. Must be REQ_PKT_LEN.
62   * @return The request type of the packet.
63   * */
64  uint16_t dtReqType(uint8_t pkt[], size_t n)
65  {
66      return ((pkt[4] << 8) | pkt[5]);
67  }
68
69  /**
70   * Returns true if the packet is a valid DT Request packet.
71   *
72   * @param pkt A pointer to the packet.
73   * @param n The number of items in the packet.
74   * @return True if the packet is valid. False otherwise.
75   * */
76  bool dtReqValid(uint8_t pkt[], size_t n)
77  {
78      if (n != REQ_PKT_LEN) {
79          return false;
80      }
81
82      if (dtPktMagicNo(pkt, n) != MAGIC_NO) {
83          return false;
84      }
85
86      if (dtPktType(pkt, n) != PACKET_REQ) {
87          return false;
88      }
89
90      if (!validReqType(dtReqType(pkt, n))) {
91          return false;
92      }
93
94      return true;
95  }
96
97  /**
```

```
 * Returns the magic number from the packet.
 * No checking is done beforehand.
 *
 * @param pkt The packet.
 * @param n The size of the packet.
 * @return The magic number of the packet.
 * */
uint16_t dtPktMagicNo(uint8_t pkt[], size_t n)
{
    return ((pkt[0] << 8) | pkt[1]);
}

/**
 * Returns the type of packet.
 * No checking is done beforehand.
 *
 * @param pkt The packet.
 * @param n The size of the packet.
 * @return The type of the packet.
 * */
uint16_t dtPktType(uint8_t pkt[], size_t n)
{
    return ((pkt[2] << 8) | pkt[3]);
}

/**
 * Returns true if reqType is a valid type of request.
 *
 * @param reqType The request type.
 * @return True if reqType is either REQ_DATE or REQ_TIME.
 * */
bool validReqType(uint16_t reqType)
{
    return (reqType == REQ_DATE || reqType == REQ_TIME);
}

/**
 * Returns true if langCode denotes a valid language.
 * Valid languages codes are LANG_ENG for English,
 * LANG_MAO for Maori or LANG_GER for German.
 *
 * @param langCode The language code.
 * @return True if langCode is valid.
 * */
bool validLangCode(uint16_t langCode)
{
    return (langCode == LANG_ENG || langCode == LANG_GER || langCode == LANG_MAO);
}

/**
 * Constructs a DT Response packet.
 *
 * @param pkt A pointer to the packet.
 * @param n The size of the packet. Must be equal to RES_PKT_LEN.
 * @param reqType The type of request. Must be either REQ_DATE or REQ_TIME.
 * @param langCode The language to respond in. Must be valid.
```

```
154   * @param year The year to return.
155   * @param month The month to return.
156   * @param day The day to return.
157   * @param hour The hour to return.
158   * @param minute The minute to return.
159   * @return The length of the packet.
160   * */
161  size_t dtRes(uint8_t pkt[], size_t n, uint16_t reqType, uint16_t langCode, uint16_t
      ↪  year, uint8_t month, uint8_t day, uint8_t hour, uint8_t minute)
162  {
163
164      // check reqType, langCode and n
165      if (!validReqType(reqType) || !validLangCode(langCode) || n != RES_PKT_LEN) {
166          return 0;
167      }
168
169      // write most of the data to the packet
170      pkt[0] = (uint8_t)(MAGIC_NO >> 8);
171      pkt[1] = (uint8_t)(MAGIC_NO & 0xFF);
172      pkt[2] = (uint8_t)(PACKET_RES >> 8);
173      pkt[3] = (uint8_t)(PACKET_RES & 0xFF);
174      pkt[4] = (uint8_t)(langCode >> 8);
175      pkt[5] = (uint8_t)(langCode & 0xFF);
176      pkt[6] = (uint8_t)(year >> 8);
177      pkt[7] = (uint8_t)(year & 0xFF);
178      pkt[8] = month;
179      pkt[9] = day;
180      pkt[10] = hour;
181      pkt[11] = minute;
182
183      // get the template phrase and the month as a string
184      char* phrase = (char*)PHRASES[langCode - 1][reqType - 1];
185
186      char* monthStr = (char*)MONTHS[langCode - 1][month - 1];
187
188      char text[RES_TEXT_LEN] = {0};
189      int length = 0;
190
191      if (reqType == REQ_DATE) {
192          // german has its values out of order, so handle it seperately
193          if (langCode == LANG_GER) {
194              length = sprintf(text, phrase, day, monthStr, year);
195          } else {
196              length = sprintf(text, phrase, monthStr, day, year);
197          }
198      } else {
199          length = sprintf(text, phrase, hour, minute);
200      }
201
202      // an error occurred during sprintf
203      if (length < 0) {
204          return 0;
205      }
206
207      // write the length to the packet
208      pkt[12] = (uint8_t)length;
```

```
209
210     // fill the rest of the packet with the text
211     for (int i = 0; i < length; i++) {
212         pkt[13 + i] = text[i];
213     }
214
215     return 13 + length;
216
217 }
218
219 /**
220  * Constructs the DT Request Packet from the current time and date.
221  *
222  * @param pkt A pointer to the packet.
223  * @param n The size of the packet. Must be equal to RES_PKT_LEN.
224  * @param reqType The type of request. Must be either REQ_DATE or REQ_TIME.
225  * @param langCode The language to respond in. Must be valid.
226  * @return The length of the packet.
227  * */
228 size_t dtResNow(uint8_t pkt[], size_t n, uint16_t reqType, uint16_t langCode)
229 {
230     struct tm* now;
231     time_t raw_time;
232
233     time(&raw_time);
234     now = localtime(&raw_time);
235
236     return dtRes(pkt, n, reqType, langCode, now->tm_year + 1900, now->tm_mon + 1,
      ↪  now->tm_mday, now->tm_hour, now->tm_min);
237 }
238
239 /**
240  * Returns true if the DT Response packet is valid.
241  *
242  * @param pkt The packet.
243  * @param n The size of the packet.
244  * @return True if the packet is valid.
245  * */
246 bool dtResValid(uint8_t pkt[], size_t n)
247 {
248     if (n < 13) {
249         return false;
250     }
251
252     if (dtPktMagicNo(pkt, n) != MAGIC_NO) {
253         return false;
254     }
255
256     if (dtPktType(pkt, n) != PACKET_RES) {
257         return false;
258     }
259
260     if (dtResYear(pkt, n) >= 2100) {
261         return false;
262     }
263
```

```
264     if (dtResMonth(pkt, n) < 1 ||
265         dtResMonth(pkt, n) > 12) {
266         return false;
267     }
268 
269     if (dtResDay(pkt, n) < 1 ||
270         dtResDay(pkt, n) > 31) {
271         return false;
272     }
273 
274     if (dtResHour(pkt, n) < 0 ||
275         dtResHour(pkt, n) > 23) {
276         return false;
277     }
278 
279     if (dtResMinute(pkt, n) < 0 ||
280         dtResMinute(pkt, n) > 59) {
281         return false;
282     }
283 
284     if (dtResLength(pkt, n) + 13 != n) {
285         return false;
286     }
287 
288     return true;
289 }
290 
291 /**
292  * Returns the language code in the packet.
293  * No checking is done beforehand.
294  *
295  * @param pkt The packet.
296  * @param n The length of the packet.
297  * @return The language code.
298  * */
299 uint16_t dtResLangCode(uint8_t pkt[], size_t n)
300 {
301     return ((pkt[4] << 8) | pkt[5]);
302 }
303 
304 /**
305  * Returns the year defined in the packet.
306  * No checking is done beforehand.
307  *
308  * @param pkt The packet.
309  * @param n The length of the packet.
310  * @return The year.
311  * */
312 uint16_t dtResYear(uint8_t pkt[], size_t n)
313 {
314     return ((pkt[6] << 8) | pkt[7]);
315 }
316 
317 /**
318  * Returns the month defined in the packet.
319  * No checking is done beforehand.
```

```
 *
 * @param The packet.
 * @param The length of the packet.
 * @return The month.
 * */
uint8_t dtResMonth(uint8_t pkt[], size_t n)
{
    return (pkt[8]);
}

/**
 * Returns the day defined in the packet.
 * No checking is done beforehand.
 *
 * @param pkt The packet.
 * @param n The length of the packet.
 * @return The day.
 * */
uint8_t dtResDay(uint8_t pkt[], size_t n)
{
    return (pkt[9]);
}

/**
 * Returns the hour defined in the packet.
 * No checking is done beforehand.
 *
 * @param pkt The packet.
 * @param n The length of the packet.
 * @return The hour.
 * */
uint8_t dtResHour(uint8_t pkt[], size_t n)
{
    return (pkt[10]);
}

/**
 * Returns the minute defined in the packet.
 * No checking is done beforehand.
 *
 * @param pkt The packet.
 * @param n The length of the packet.
 * @return The minute.
 * */
uint8_t dtResMinute(uint8_t pkt[], size_t n)
{
    return (pkt[11]);
}

/**
 * Returns the length of the text in the packet.
 * No checking is done beforehand.
 *
 * @param pkt The packet.
 * @param n The length of the packet.
 * @return The length of the text.
```

```
 * */
uint8_t dtResLength(uint8_t pkt[], size_t n)
{
    return (pkt[12]);
}

/**
 * Copies the text from the packet and puts it into the char array text.
 * No checking is done beforehand.
 *
 * @param pkt The packet.
 * @param n The length of the packet.
 * @param text The char array to store the text in.
 * @param textLen A pointer where the length of the text is to be stored.
 * */
void dtResText(uint8_t pkt[], size_t n, char text[], size_t* textLen)
{
    *textLen = dtResLength(pkt, n);
    for (int i = 0; i < *textLen; i++) {
        text[i] = pkt[13 + i];
    }
    text[*textLen] = 0;
}

/**
 * Dumps the packet data to stdout.
 *
 * @param pkt The packet.
 * @param n The length of the packet.
 * */
void dtPktDump(uint8_t pkt[])
{
    size_t n = dtPktLength(pkt);
    for (int i = 0; i < n; i++) {

        // print the character
        printf("%02X ", pkt[i]);

        // if we are at the end of the line, also print the ascii values
        if ((i + 1) % 8 == 0) {

            // draw the partition
            printf("| ");

            // draw the ascii value or a period
            for (int j = i - 7; j <= i; j++) {
                if (isprint(pkt[j])) {
                    putchar(pkt[j]);
                } else {
                    putchar('.');
                }
            }
            putchar('\n');
        }
    }

```

```
432         // fill the remaining space on the left hand side
433         for (int k = (n % 8); k < 8; k++) {
434             printf("   ");
435         }
436 
437         // draw the partition
438         printf("| ");
439 
440         // print the character if it is printable else a fullstop
441         for (int l = (n / 8) * 8; l < n; l++) {
442             if (isprint(pkt[l])) {
443                 putchar(pkt[l]);
444             } else {
445                 putchar('.');
446             }
447         }
448 
449         printf("\n");
450     }
451 
452     /**
453      * Returns the length of the packet.
454      *
455      * @param pkt The packet.
456      * @return The length of the packet.
457      * */
458     size_t dtPktLength(uint8_t pkt[])
459     {
460         uint16_t pktType = dtPktType(pkt, RES_PKT_LEN);
461 
462         if (pktType == PACKET_REQ) {
463             return REQ_PKT_LEN;
464         } else if (pktType == PACKET_RES) {
465             return (13 + dtResLength(pkt, RES_PKT_LEN));
466         } else {
467             return 0;
468         }
469     }
470 
471     char* getLangName(uint16_t langCode)
472     {
473         switch (langCode) {
474             case LANG_ENG: return "English";
475             case LANG_GER: return "German";
476             case LANG_MAO: return "Te Reo M\u0101ori";
477             default: return "";
478         }
479     }
480 
481     char* getRequestTypeString(uint16_t reqType)
482     {
483         switch (reqType) {
484             case REQ_DATE: return "date";
485             case REQ_TIME: return "time";
486             default: return "";
487         }
```

```
488 }
```

2.3 Server

Contains functions that pertain only to the server.

```h
// server.h

#ifndef SERVER_H
#define SERVER_H

bool readPorts(char** argv, uint16_t* ports);
void serve(uint16_t ports[]);
void handleSignal(int sig);

#endif
```

```c
// server.c

#include <arpa/inet.h>
#include <signal.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <unistd.h>

#include "protocol.h"
#include "server.h"
#include "utils.h"

// the socket descriptors
// these must be global in order to safely close them on SIGINT
int socket_fds[3];

/**
 * Usage: server <english port> <te reo maori port> <german port>
 * */
int main(int argc, char** argv)
{
    uint16_t ports[3] = {0};

    // validate the arguments
    if (argc != 4) {
        error("server must receive exactly 3 arguments", 1);
    }

    // read the ports into the ports array
    if (!readPorts(argv, ports)) {
        char msg[52] = {0};
        sprintf(msg, "ports must be between %u and %u (inclusive)", MIN_PORT_NO,
        ↪  MAX_PORT_NO);
        error(msg, 1);
    }
```

```
41
42      // check that the ports are unique
43      if (ports[0] == ports[1] || ports[0] == ports[2] || ports[1] == ports[2]) {
44          error("port numbers must be unique", 1);
45      }
46
47      // handle some signals so that the sockets can shutdown gracefully
48      signal(SIGINT, handleSignal);
49
50      // serve on the specified ports
51      serve(ports);
52
53      return EXIT_SUCCESS;
54  }
55
56  /**
57   * Gracefully shutdown the server by closing the sockets.
58   *
59   * @param sig The signal sent to the program.
60   * */
61  void handleSignal(int sig)
62  {
63      printf("Closing sockets...\n");
64
65      // close the sockets one at a time
66      for (int i = 0; i < 3; i++) {
67          close(socket_fds[i]);
68      }
69
70      exit(0);
71  }
72
73  /**
74   * Serves on all three ports.
75   *
76   * @param The list of ports to serve on.
77   * */
78  void serve(uint16_t ports[])
79  {
80
81      // holds the server address information
82      struct sockaddr_in server_addr[3];
83
84      // create three sockets for the three ports
85      for (int i = 0; i < 3; i++) {
86
87          // required for setsockopt(), set it to 1 to allow us to reuse local addresses
88          int option_value = 1;
89
90          socket_fds[i] = socket(AF_INET, SOCK_DGRAM, 0);
91
92          if (socket_fds[i] < 0) {
93              error("could not create a socket", 2);
94          }
95
96          // lets us reuse the port after killing the server.
```

```
97          setsockopt(socket_fds[i], SOL_SOCKET, SO_REUSEADDR,
98              (const void *) &option_value, sizeof(int));
99
100         // fill out the s_addr struct with information about how we want to serve data
101         memset((char *) &server_addr[i], 0, sizeof(server_addr[i]));
102         server_addr[i].sin_family = AF_INET;
103         server_addr[i].sin_addr.s_addr = INADDR_ANY;
104         server_addr[i].sin_port = htons(ports[i]);
105
106         // attempt to bind to the port number
107         if (bind(socket_fds[i], (struct sockaddr *) &server_addr[i],
            ↪  sizeof(server_addr[i])) < 0) {
108             error("could not bind to socket", 2);
109         }
110
111         // print some information and listen
112         printf("Listening on port %u for %s requests...\n", ports[i], getLangName(i +
            ↪  1));
113
114         listen(socket_fds[i], 5);
115
116     }
117
118     // loop forever
119     while (true) {
120
121         // holds the client address information
122         struct sockaddr_in client_addr;
123
124         // the length of the client address data struct
125         socklen_t client_addr_len = sizeof(client_addr);
126
127         // holds the IP address of the client
128         char client_ip_address_string[INET_ADDRSTRLEN];
129
130         // the active socket that received the request
131         // used to determine the port number and to send a response
132         int active_socket_fd = -1;
133
134         // the type of request we are handling, read from the request packet
135         uint16_t request_type = 0;
136
137         // the language the user requested the response in
138         // this is based on the port they connect to
139         uint16_t language_code = 0;
140
141         // holds the number of bytes received by the server for a request
142         int bytes_received;
143
144         // the buffer to place the received data
145         uint8_t buffer[256];
146
147         // the buffer to hold the response data
148         uint8_t response[RES_PKT_LEN];
149
150         // holds information on which sockets to wait for while selecting
```

```
151         fd_set socket_set;
152
153         // reset the socket_set struct
154         FD_ZERO(&socket_set);
155         for (int i = 0; i < 3; i++) {
156             FD_SET(socket_fds[i], &socket_set);
157         }
158
159         // perform the select
160         int selectResult = select(max(socket_fds, 3) + 1, &socket_set, NULL, NULL,
            ↪  NULL);
161
162         // if there was an error selecting
163         if (selectResult == -1) {
164             error("select failed", 4);
165         }
166
167         // iterate through the pollfd values until one is ready to receive data
168         // store the socket descriptor and the language code
169         for (int i = 0; i < 3; i++) {
170
171             if (FD_ISSET(socket_fds[i], &socket_set)) {
172
173                 active_socket_fd = socket_fds[i];
174                 language_code = i + 1;
175
176                 // break when we find a readable socket descriptor
177                 // of course this means that English will have priority over
178                 // Maori will have priority over German
179                 break;
180             }
181         }
182
183         // receive data from the client
184         bytes_received = recvfrom(active_socket_fd, buffer, sizeof(buffer), 0,
185             (struct sockaddr *) &client_addr, &client_addr_len);
186
187         // get the IP address of the client as a string
188         inet_ntop(AF_INET, &client_addr.sin_addr, client_ip_address_string,
            ↪  INET_ADDRSTRLEN);
189
190         // print the date, time and the ip address of the client
191         printCurrentDateTimeString();
192         printf(" - %s - ", client_ip_address_string);
193
194         // if an error occurred during reading the information, print an error
195         if (bytes_received < 0) {
196             printf("network error - packet discarded\n");
197             continue;
198         }
199
200         // handle the data
201         if (!dtReqValid(buffer, bytes_received)) {
202
203             printf("invalid request - packet discarded\n");
204
```

```
205         } else {
206
207             // print some more information
208             request_type = dtReqType(buffer, bytes_received);
209
210             printf("%s %s requested - ", getLangName(language_code),
                ↪  getRequestTypeString(request_type));
211
212             // zero the response packet buffer
213             memset(response, 0, RES_PKT_LEN);
214
215             // construct the response packet
216             size_t b = dtResNow(response, RES_PKT_LEN, request_type, language_code);
217
218             // attempt to sent the response packet
219             if (sendto(active_socket_fd, response, b, 0, (struct sockaddr *)
                ↪  &client_addr, client_addr_len) < 0) {
220                 printf("response failed to send\n");
221             } else {
222                 printf("response sent\n");
223             }
224
225         }
226
227     }
228
229 }
230
231 /**
232  * Reads the ports from argv and puts them into the ports array.
233  *
234  * @param arv The arguments passed into main.
235  * @param ports The array to populate with ports.
236  * @return True if all ports were valid.
237  * */
238 bool readPorts(char** argv, uint16_t* ports)
239 {
240     for (int i = 0; i < 3; i++) {
241         ports[i] = atoi(argv[i+1]);
242         if (ports[i] < MIN_PORT_NO ||
243             ports[i] > MAX_PORT_NO) {
244             return false;
245         }
246     }
247     return true;
248 }
```

2.4 Client

Contains functions that pertain only to the client.

```c
// client.h

#ifndef CLIENT_H
#define CLIENT_H

#include <stdint.h>

int main(int argc, char** argv);
void request(uint16_t reqType, char* ip_addr, char* port);

#endif
```

```c
// client.c

#include <arpa/inet.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>

#include "client.h"
#include "protocol.h"
#include "utils.h"

/**
 * Usage: client <time|date> <ip address> <port>
 * */
int main(int argc, char** argv)
{
    uint16_t request_type, port;

    // validate the number of arguments passed in
    if (argc != 4) {
        error("client expects exactly 4 arguments", 1);
    }

    // set the request_type based on the first argument
    if (strcmp(argv[1], "date") == 0) {
        request_type = REQ_DATE;
    } else if (strcmp(argv[1], "time") == 0) {
        request_type = REQ_TIME;
    } else {
        error("first argument should be either \"date\" or \"time\"" ,1);
    }

```

```
41      // set the port based on the third argument
42      port = atoi(argv[3]);
43      if (port < MIN_PORT_NO || port > MAX_PORT_NO) {
44          char msg[55] = {0};
45          sprintf(msg, "the port must be between %u and %u (inclusive)", MIN_PORT_NO,
            ↪  MAX_PORT_NO);
46          error(msg, 1);
47      }
48
49      // send a request
50      request(request_type, argv[2], argv[3]);
51
52      return 0;
53  }
54
55  /**
56   * Sends a request to the server.
57   *
58   * @param request_type The type of request, either REQ_DATE or REQ_TIME.
59   * @param ip_address_string The ip address of the server as a string.
60   * @param port The port the server is listening on.
61   * */
62  void request(uint16_t request_type, char* ip_address_string, char* port_string)
63  {
64
65      // the address information of the server
66      // struct sockaddr_in server_address;
67      socklen_t server_address_len;
68
69      // the socket descriptor of the client
70      int client_socket;
71
72      // the buffers to hold the raw request and response packets
73      uint8_t req[REQ_PKT_LEN] = {0};
74      uint8_t buffer[RES_PKT_LEN] = {0};
75
76      // stores the amount of time for select() to wait before returning
77      struct timeval timeout;
78
79      // this is required for select() to work
80      fd_set socket_set;
81
82      // this is used to test what is returned by select()
83      int select_result;
84
85      // set aside some space for the text from the incoming data to be placed
86      char text[RES_TEXT_LEN] = {0};
87
88      // denotes the length of the text received.
89      size_t text_len = 0;
90
91      // holds the server address hints
92      struct addrinfo hints;
93
94      // a pointer to all the addresses returned by getaddrinfo
95      struct addrinfo *addresses;
```

```
96
97      // a pointer to the current address used as the server address
98      struct addrinfo *server_address;
99
100     // setup the hints data structure
101     memset(&hints, 0, sizeof(struct addrinfo));
102     hints.ai_family = AF_INET;
103     hints.ai_socktype = SOCK_DGRAM;
104
105     // get the address info
106     if (getaddrinfo(ip_address_string, port_string, &hints, &addresses) != 0) {
107         error("bad hostname or ip address", 1);
108     }
109
110     // iterate over every possible server address returned by getaddrinfo
111     // and select the first one that we can connect to
112     for (server_address = addresses; server_address != NULL; server_address =
        ↪  server_address->ai_next) {
113
114         // create a socket
115         client_socket = socket(server_address->ai_family,
116             server_address->ai_socktype, server_address->ai_protocol);
117
118         // if it is no good, get another one
119         if (client_socket < 0) {
120             continue;
121         }
122
123         // attempt to connect to the server, if this works, break
124         if (connect(client_socket, server_address->ai_addr,
125             server_address->ai_addrlen) == 0) {
126             break;
127         }
128
129         // close this socket and get another one
130         close(client_socket);
131     }
132
133     // display an error if we could not connect to the server
134     if (server_address == NULL) {
135         error("could not connect", 1);
136     }
137
138     server_address_len = sizeof(server_address);
139
140     // create the packet
141     if (dtReq(req, REQ_PKT_LEN, request_type) == 0) {
142         error("could not create packet", 3);
143     }
144
145     // set the timeout to be one second
146     timeout.tv_sec = 1;
147     timeout.tv_usec = 0;
148
149     // set the socket_set
150     FD_ZERO(&socket_set);
```

```
151     FD_SET(client_socket, &socket_set);
152
153     // wait for the socket to be writable
154     select_result = select(client_socket + 1, NULL, &socket_set, NULL, &timeout);
155
156     // print an error if something went wrong while selecting
157     if (select_result < 0) {
158         error("could not select", 4);
159     }
160
161     // print an error if a timeout occurred
162     if (select_result == 0) {
163         error("select timed out", 4);
164     }
165
166     // attempt to send the packet
167     if (sendto(client_socket, req, REQ_PKT_LEN, 0, (struct sockaddr *)
        ↪  server_address->ai_addr, server_address->ai_addrlen) < 0) {
168         error("could not send packet", 2);
169     }
170
171     // set the timeout to be one second, this must be set again because select()
        ↪  modifies the timeout
172     timeout.tv_sec = 1;
173     timeout.tv_usec = 0;
174
175     // set the socket_set, this must be set again because select() modifies socket_set
176     FD_ZERO(&socket_set);
177     FD_SET(client_socket, &socket_set);
178
179     // Wait for the socket to be readable
180     select_result = select(client_socket + 1, &socket_set, NULL, NULL, &timeout);
181
182     // print an error if something went wrong while selecting
183     if (select_result < 0) {
184         error("could not select", 4);
185     }
186
187     // print an error if a timeout occurred
188     if (select_result == 0) {
189         error("select timed out", 4);
190     }
191
192     // attempt to receive the response
193     if (recvfrom(client_socket, buffer, RES_PKT_LEN, 0, (struct sockaddr *)
        ↪  &server_address, &server_address_len) < 0) {
194         error("could not recieve packet", 2);
195     }
196
197     // free the memory used by getaddrinfo
198     freeaddrinfo(addresses);
199
200     // close the socket
201     close(client_socket);
202
203     // extract the text, storing it in text and the length in text_len
```

```
204     dtResText(buffer, dtPktLength(buffer), text, &text_len);
205 
206     // print the other information
207     printf("MagicNo:\t0x%04X\n", dtPktMagicNo(buffer, RES_PKT_LEN));
208     printf("PacketType:\t%u\n", dtPktType(buffer, RES_PKT_LEN));
209     printf("LanguageCode:\t%u\n", dtResLangCode(buffer, RES_PKT_LEN));
210     printf("Year:\t\t%u\n", dtResYear(buffer, RES_PKT_LEN));
211     printf("Month:\t\t%u\n", dtResMonth(buffer, RES_PKT_LEN));
212     printf("Day:\t\t%u\n", dtResDay(buffer, RES_PKT_LEN));
213     printf("Hour:\t\t%u\n", dtResHour(buffer, RES_PKT_LEN));
214     printf("Minute:\t\t%u\n", dtResMinute(buffer, RES_PKT_LEN));
215     printf("Length:\t\t%u\n", dtResLength(buffer, RES_PKT_LEN));
216 
217     // print the text response
218     printf("Text:\t\t%s\n", text);
219 
220 }
```

2.5 Protocol Testing

Contains functions that test the integrity of the protocol functions.

```c
// protocol.test.c

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#include "../protocol.h"
#include "../utils.h"

int main(void)
{
    uint16_t failures = 0;

    // ** dtReq **
    // create a request packet with a size too small
    size_t smallReqPktLen = REQ_PKT_LEN - 1;
    uint8_t smallReqPkt[smallReqPktLen];

    if (dtReq(smallReqPkt, smallReqPktLen, REQ_DATE) != 0) {
        failures++;
        fail("dtReq", "n is too small");
    }

    // create a request packet with a size too large
    size_t largeReqPktLen = REQ_PKT_LEN + 1;
    uint8_t largeReqPkt[largeReqPktLen];

    if (dtReq(largeReqPkt, largeReqPktLen, REQ_DATE) != 0) {
        failures++;
        fail("dtReq", "n is too large");
    }

    // create a request packet with an invalid reqType
    uint8_t reqPktDate[REQ_PKT_LEN] = {0};

    if (dtReq(reqPktDate, REQ_PKT_LEN, 99) != 0) {
        failures++;
        fail("dtReq", "invalid reqType");
    }

    // create a valid request packet
    if (dtReq(reqPktDate, REQ_PKT_LEN, REQ_DATE) != REQ_PKT_LEN) {
        failures++;
        fail("dtReq", "reqType is valid");
    }

    uint8_t reqPktTime[REQ_PKT_LEN] = {0};

    if (dtReq(reqPktTime, REQ_PKT_LEN, REQ_TIME) != REQ_PKT_LEN) {
        failures++;
        fail("dtReq", "reqType is valid");
    }
```

```
53
54      // ** dtReqType **
55      // check that the dtReqType is returned
56      if (dtReqType(reqPktDate, REQ_PKT_LEN) != REQ_DATE) {
57          failures++;
58          fail("dtReqType", "REQ_DATE not returned");
59      }
60
61      if (dtReqType(reqPktTime, REQ_PKT_LEN) != REQ_TIME) {
62          failures++;
63          fail("dtReqType", "REQ_TIME not returned");
64      }
65
66      // ** dtReqValid **
67      // check with a packet with a size too small
68      if (dtReqValid(smallReqPkt, smallReqPktLen)) {
69          failures++;
70          fail("dtReqValid", "n should be too small");
71      }
72
73      // check with a packet with a size too large
74      if (dtReqValid(largeReqPkt, largeReqPktLen)) {
75          failures++;
76          fail("dtReqValid", "n should be too large");
77      }
78
79      // check with an invalid magic number
80      uint8_t badMagicNoReqPkt[REQ_PKT_LEN] = {0xDE, 0xAD, 0x00, 0x01, 0x00, 0x01};
81      if (dtReqValid(badMagicNoReqPkt, REQ_PKT_LEN)) {
82          failures++;
83          fail("dtReqValid", "magic no should be incorrect");
84      }
85
86      // check with an invalid packet type
87      uint8_t badPktTypeReqPkt[REQ_PKT_LEN] = {0x49, 0x7E, 0x99, 0x88, 0x00, 0x01};
88      if (dtReqValid(badPktTypeReqPkt, REQ_PKT_LEN)) {
89          failures++;
90          fail("dtReqValid", "pktType should be incorrect");
91      }
92
93      // check with an invalid request type
94      uint8_t badReqTypeReqPkt[REQ_PKT_LEN] = {0x49, 0x7E, 0x000, 0x01, 0xDE, 0xAD};
95      if (dtReqValid(badReqTypeReqPkt, REQ_PKT_LEN)) {
96          failures++;
97          fail("dtReqValid", "reqType should be incorrect");
98      }
99
100     // check with a valid packet
101     if (!dtReqValid(reqPktTime, REQ_PKT_LEN)) {
102         failures++;
103         fail("dtReqValid", "time packet should be correct");
104     }
105
106     if (!dtReqValid(reqPktDate, REQ_PKT_LEN)) {
107         failures++;
108         fail("dtReqValid", "date packet should be correct");
```

```
109     }
110
111     // ** dtPktMagicNo **
112     // check that the magic number is extracted
113     if (dtPktMagicNo(reqPktDate, REQ_PKT_LEN) != MAGIC_NO) {
114         failures++;
115         fail("dtPktMagicNo", "magic no should be correct");
116     }
117
118     // ** dtPktType **
119     // check that the packet type is extracted
120     if (dtPktType(reqPktDate, REQ_PKT_LEN) != PACKET_REQ) {
121         failures++;
122         fail("dtPktType", "packet type should be correct");
123     }
124
125     // ** validReqType **
126     // check with a valid request type
127     if (!validReqType(REQ_DATE)) {
128         failures++;
129         fail("validReqType", "REQ_DATE should be correct");
130     }
131
132     if (!validReqType(REQ_TIME)) {
133         failures++;
134         fail("validReqType", "REQ_TIME should be correct");
135     }
136
137     // check with an invalid request type
138     if (validReqType(0xBEEF)) {
139         failures++;
140         fail("validReqType", "0xBEEF should be incorrect");
141     }
142
143     // ** validLangCode **
144     // check with valid lang codes
145     if (!validLangCode(LANG_ENG)) {
146         failures++;
147         fail("validLangCode", "LANG_ENG should be correct");
148     }
149
150     if (!validLangCode(LANG_GER)) {
151         failures++;
152         fail("validLangCode", "LANG_GER should be correct");
153     }
154
155     if (!validLangCode(LANG_MAO)) {
156         failures++;
157         fail("validLangCode", "LANG_MAO should be correct");
158     }
159
160     // check with an invalid lang code
161     if (validLangCode(0xBEEF)) {
162         failures++;
163         fail("validLangCode", "0xBEEF should be incorrect");
164     }
```

```
165
166     // ** dtRes **
167     // create a packet with a size too small
168     size_t smallResPktLen = RES_PKT_LEN - 1;
169     uint8_t smallResPkt[smallResPktLen];
170     if (dtRes(smallResPkt, smallResPktLen, REQ_DATE, LANG_ENG, 2018, 6, 10, 12, 45) ==
        ↪  smallResPktLen) {
171         failures++;
172         fail("dtRes", "n should be too small");
173     }
174
175     // create a packet with a size too large
176     size_t largeResPktLen = RES_PKT_LEN + 1;
177     uint8_t largeResPkt[largeResPktLen];
178     if (dtRes(largeResPkt, largeResPktLen, REQ_DATE, LANG_ENG, 2018, 6, 10, 12, 45) ==
        ↪  largeResPktLen) {
179         failures++;
180         fail("dtRes", "n should be too large");
181     }
182
183     // create a packet with an invalid request type
184     uint8_t badReqTypeResPkt[RES_PKT_LEN] = {0};
185     if (dtRes(badReqTypeResPkt, RES_PKT_LEN, 0xBEEF, LANG_ENG, 2018, 6, 10, 12, 45) ==
        ↪  RES_PKT_LEN) {
186         failures++;
187         fail("dtRes", "reqType should be invalid");
188     }
189
190     // create a packet with an invalid language code
191     uint8_t badLangCodeResPkt[RES_PKT_LEN] = {0};
192     if (dtRes(badLangCodeResPkt, RES_PKT_LEN, REQ_TIME, 0xBEEF, 2018, 6, 10, 12, 45) ==
        ↪  RES_PKT_LEN) {
193         failures++;
194         fail("dtRes", "langCode should be invalid");
195     }
196
197     // create valid packets and check that the size returned is correct
198     uint8_t dateEngResPkt[RES_PKT_LEN] = {0};
199     if (dtRes(dateEngResPkt, RES_PKT_LEN, REQ_DATE, LANG_ENG, 2018, 6, 10, 12, 45) !=
        ↪  13 + 29) {
200         failures++;
201         fail("dtRes", "english date packet isn't the correct length");
202         dtPktDump(dateEngResPkt);
203     }
204
205     uint8_t timeEngResPkt[RES_PKT_LEN] = {0};
206     if (dtRes(timeEngResPkt, RES_PKT_LEN, REQ_TIME, LANG_ENG, 2018, 6, 10, 12, 45) !=
        ↪  13 + 25) {
207         failures++;
208         fail("dtRes", "english time packet isn't the correct length");
209         dtPktDump(timeEngResPkt);
210     }
211
212     // ** dtResValid **
213     // check with a packet that is too small
214     if (dtResValid(smallResPkt, smallResPktLen)) {
```

```
215         failures++;
216         fail("dtResValid", "n should be too small");
217     }
218 
219     // check with a packet that is too large
220     if (dtResValid(largeResPkt, largeResPktLen)) {
221         failures++;
222         fail("dtResValid", "n should be too large");
223     }
224 
225     // check with a packet with an invalid magic number
226     timeEngResPkt[0] = 0xBE;
227     timeEngResPkt[1] = 0xEF;
228     if (dtResValid(timeEngResPkt, RES_PKT_LEN)) {
229         failures++;
230         fail("dtResValid", "magic number should be invalid");
231     }
232     timeEngResPkt[0] = (uint8_t)(MAGIC_NO >> 8);
233     timeEngResPkt[1] = (uint8_t)(MAGIC_NO & 0xFF);
234 
235     // check with a packet with an invalid packet type
236     timeEngResPkt[2] = 0xBE;
237     timeEngResPkt[3] = 0xEF;
238     if (dtResValid(timeEngResPkt, RES_PKT_LEN)) {
239         failures++;
240         fail("dtResValid", "packet type should be invalid");
241     }
242     timeEngResPkt[2] = (uint8_t)(PACKET_RES >> 8);
243     timeEngResPkt[3] = (uint8_t)(PACKET_RES & 0xFF);
244 
245     // check with a packet with an invalid year
246     timeEngResPkt[6] = 0xBE;
247     timeEngResPkt[7] = 0xEF;
248     if (dtResValid(timeEngResPkt, RES_PKT_LEN)) {
249         failures++;
250         fail("dtResValid", "year should be invalid");
251     }
252     timeEngResPkt[6] = (uint8_t)(2018 >> 8);
253     timeEngResPkt[7] = (uint8_t)(2018 & 0xFF);
254 
255     // check with a packet with a month too small
256     timeEngResPkt[8] = 0;
257     if (dtResValid(timeEngResPkt, RES_PKT_LEN)) {
258         failures++;
259         fail("dtResValid", "month should be too small");
260     }
261 
262     // check with a packet with a month too large
263     timeEngResPkt[8] = 13;
264     if (dtResValid(timeEngResPkt, RES_PKT_LEN)) {
265         failures++;
266         fail("dtResValid", "month should be too large");
267     }
268     timeEngResPkt[8] = (uint8_t)(6);
269 
270     // check with a packet with a day too small
```

```
    timeEngResPkt[9] = 0;
    if (dtResValid(timeEngResPkt, RES_PKT_LEN)) {
        failures++;
        fail("dtResValid", "day should be too small");
    }

    // check with a packet with a day too large
    timeEngResPkt[9] = 32;
    if (dtResValid(timeEngResPkt, RES_PKT_LEN)) {
        failures++;
        fail("dtResValid", "day should be too large");
    }
    timeEngResPkt[9] = (uint8_t)(10);

    // check with a packet with an hour too large
    timeEngResPkt[10] = 24;
    if (dtResValid(timeEngResPkt, RES_PKT_LEN)) {
        failures++;
        fail("dtResValid", "hour should be too large");
    }
    timeEngResPkt[10] = (uint8_t)(12);

    // check with a packet with a minute too large
    timeEngResPkt[11] = 60;
    if (dtResValid(timeEngResPkt, RES_PKT_LEN)) {
        failures++;
        fail("dtResValid", "minute should be too large");
    }
    timeEngResPkt[11] = (uint8_t)(45);

    // check with valid packets
    if (!dtResValid(timeEngResPkt, dtPktLength(timeEngResPkt))) {
        failures++;
        fail("dtResValid", "english time packet should be valid");
        dtPktDump(timeEngResPkt);
    }

    if (!dtResValid(dateEngResPkt, dtPktLength(dateEngResPkt))) {
        failures++;
        fail("dtResValid", "english date packet should be valid");
        dtPktDump(dateEngResPkt);
    }

    // ** dtResLangCode **
    // check that the lang code is extracted
    if (dtResLangCode(dateEngResPkt, RES_PKT_LEN) != LANG_ENG) {
        failures++;
        fail("dtResLangCode", "lang code is not extracted");
    }

    // ** dtResYear **
    // check that the year is extracted
    if (dtResYear(dateEngResPkt, RES_PKT_LEN) != 2018) {
        failures++;
        fail("dtResYear", "year is not extracted");
    }
```

```
327
328     // ** dtResMonth **
329     // check that the month is extracted
330     if (dtResMonth(dateEngResPkt, RES_PKT_LEN) != 6) {
331         failures++;
332         fail("dtResMonth", "month is not extracted");
333     }
334
335     // ** dtResDay **
336     // check that the day is extracted
337     if (dtResDay(dateEngResPkt, RES_PKT_LEN) != 10) {
338         failures++;
339         fail("dtResDay", "day is not extracted");
340     }
341
342     // ** dtResHour **
343     // check that the hour is extracted
344     if (dtResHour(dateEngResPkt, RES_PKT_LEN) != 12) {
345         failures++;
346         fail("dtResHour", "hour is not extracted");
347     }
348
349     // ** dtResMinute **
350     // check that the minute is extracted
351     if (dtResMinute(dateEngResPkt, RES_PKT_LEN) != 45) {
352         failures++;
353         fail("dtResMinute", "minute is not extracted");
354     }
355
356     // ** dtResLength **
357     // check that the length is extracted
358     if (dtResLength(dateEngResPkt, RES_PKT_LEN) != 29) {
359         failures++;
360         fail("dtResLength", "length is not extracted");
361         dtPktDump(dateEngResPkt);
362     }
363
364     // ** dtResText **
365     // check that the text is extracted and that the length is ok
366     char text[RES_TEXT_LEN] = {0};
367     size_t textLen = 0;
368     dtResText(dateEngResPkt, RES_PKT_LEN, text, &textLen);
369
370     for (int i = 0; i < textLen - 1; i++) {
371         if (text[i] != dateEngResPkt[13 + i]) {
372             failures++;
373             fail("dtResText", "text is not extracted");
374         }
375     }
376
377     return failures;
378 }
```

2.6 Makefile

Used to build the aforementioned source listings.

```
# Makefile

CFLAGS = -std=gnu99 -Werror -Wall -I ./src/

all: libs server client

libs:
	gcc $(CFLAGS) -c -o obj/protocol.o src/protocol.c
	gcc $(CFLAGS) -c -o obj/utils.o src/utils.c

server: libs src/server.c
	gcc $(CFLAGS) -o bin/server obj/protocol.o obj/utils.o src/server.c

client: libs src/client.c
	gcc $(CFLAGS) -o bin/client obj/protocol.o obj/utils.o src/client.c

test: libs src/test/protocol.test.c
	gcc $(CFLAGS) -o bin/test/protocol.test obj/protocol.o obj/utils.o
	↪  src/test/protocol.test.c

pdf:
	cd report; pdflatex --shell-escape -undump=pdflatex report.tex

clean:
	rm -v obj/protocol.o obj/utils.o
	rm -v bin/server
	rm -v bin/client
	rm -v bin/test/*
	rm report/report.pdf
```