

COSC 264 Assignment

Jesse Sheehan (Student ID: 53366509)

August 16, 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

1	Plagiarism Declaration	2
2	Source Code Listings	3
2.1	Utilities	3
2.2	Protocol	5
2.3	Server	17
2.4	Client	23
2.5	Protocol Testing	29
2.6	Makefile	38

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.

```
1 // utils.h
2
3 #ifndef UTILS_H
4 #define UTILS_H
5
6 void fail(char funcname[], char condition[]);
7 void error(char message[], int code);
8 void printCurrentDateTimeString();
9 int max(int nums[], int n);
10
11 #endif
```

```
1 // utils.c
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <time.h>
7
8 /**
9  * Prints an error message.
10  *
11  * @param funcname The name of the function where the error occurred.
12  * @param condition A description of what failed.
13  * */
14 void fail(char function_name[], char condition[])
15 {
16     fprintf(stderr, "Failure: %s - %s\n", function_name, condition);
17 }
18
19 /**
20  * Prints an error message then exits.
21  *
22  * @param message The message to print.
23  * @param code The exit code.
24  * */
25 void error(char message[], int code)
26 {
27     fprintf(stderr, "*** Error: %s ***\n", message);
28     exit(code);
29 }
30
31 /**
32  * Returns the current time string without a newline character
33  * @return A pointer to the datetime string.
34  * */
35 void printCurrentDateTimeString()
36 {
37     // used to store the time
38     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:%I", 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.

```

1  // protocol.h
2
3  #ifndef PROTOCOL_H
4  #define PROTOCOL_H
5
6  #include <stdint.h>
7  #include <stddef.h>
8  #include <stdbool.h>
9
10 // Protocol definitions
11 #define MAGIC_NO 0x497E
12 #define PACKET_REQ 0x0001
13 #define PACKET_RES 0x0002
14
15 #define MIN_PORT_NO 1024
16 #define MAX_PORT_NO 64000
17
18 #define REQ_DATE 0x0001
19 #define REQ_TIME 0x0002
20 #define REQ_PKT_LEN 6
21
22 #define RES_TEXT_LEN 255
23 #define RES_PKT_LEN (13 + RES_TEXT_LEN)
24
25 // Language code definitions
26 #define LANG_ENG 0x0001
27 #define LANG_MAO 0x0002
28 #define LANG_GER 0x0003
29
30 // Helper functions
31 bool validLangCode(uint16_t langCode);
32 bool validReqType(uint16_t reqType);
33 char* getLangName(uint16_t langCode);
34 char* getRequestTypeString(uint16_t reqType);
35
36 // General packet functions
37 uint16_t dtPktMagicNo(uint8_t pkt[], size_t n);
38 uint16_t dtPktType(uint8_t pkt[], size_t n);
39 void dtPktDump(uint8_t pkt[]);
40 size_t dtPktLength(uint8_t pkt[]);
41
42 // DT Request functions
43 size_t dtReq(uint8_t pkt[], size_t n, uint16_t reqType);
44 uint16_t dtReqType(uint8_t pkt[], size_t n);
45 bool dtReqValid(uint8_t pkt[], size_t n);
46
47 // DT Response functions
48 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);
49 size_t dtResNow(uint8_t pkt[], size_t n, uint16_t reqType, uint16_t langCode);
50 bool dtResValid(uint8_t pkt[], size_t n);

```

```

51 uint16_t dtResLangCode(uint8_t pkt[], size_t n);
52 uint16_t dtResYear(uint8_t pkt[], size_t n);
53 uint8_t dtResMonth(uint8_t pkt[], size_t n);
54 uint8_t dtResDay(uint8_t pkt[], size_t n);
55 uint8_t dtResHour(uint8_t pkt[], size_t n);
56 uint8_t dtResMinute(uint8_t pkt[], size_t n);
57 uint8_t dtResLength(uint8_t pkt[], size_t n);
58 void dtResText(uint8_t pkt[], size_t n, char text[], size_t* textLen);
59
60 #endif

```

```

1  // protocol.c
2
3  #include <stdio.h>
4  #include <ctype.h>
5  #include <time.h>
6
7  #include "protocol.h"
8
9  // The phrases to send as a response. Written as templates to be filled with sprintf.
10 const char* PHRASES[3][2] = {
11     { "Today's date is %s %02u, %04u", "The current time is %02u:%02u" },
12     { "Ko te ra o tenei ra ko %s %02u, %04u", "Ko te wa o tenei wa %02u:%02u" },
13     { "Heute ist der %02u. %s %04u", "Die Uhrzeit ist %02u:%02u" }
14 };
15
16 // The names of the months as strings. Some UTF codes are required for Maori and
17   ↪ German.
18 const char* MONTHS[3][12] = {
19     { "January", "February", "March", "April", "May", "June",
20       "July", "August", "September", "October", "November", "December"},
21     { "Kohit\u0101tea", "Hui-tanguru", "Pout\u016B-te-rangi",
22       ↪ "Paenga-wh\u0101wh\u0101", "Haratua", "Pipiri",
23       ↪ "H\u014Dngongoi", "Here-turi-k\u014Dk\u0101", "Mahuru", "Whiringa-\u0101-nuku",
24       ↪ "Whiringa-\u0101-rangi", "Hakihea" },
25     { "Januar", "Februar", "M\u00E4rz", "April", "Mai", "Juni",
26       "Juli", "August", "September", "Oktober", "November", "Dezember" }
27 };
28
29 /**
30  * Creates a DT Request packet and puts it into a uint8_t array.
31  *
32  * The packet array must be REQ_PKT_LEN long otherwise
33  * you risk a buffer overflow.
34  *
35  * @param pkt A pointer to the packet.
36  * @param n The size of the array. Must be REQ_PKT_LEN.
37  * @param reqType Must be REQ_DATE or REQ_TIME.
38  * @return The length of the packet.
39  * */
40 size_t dtReq(uint8_t pkt[], size_t n, uint16_t reqType)
41 {
42     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 /**

```

```

98  * Returns the magic number from the packet.
99  * No checking is done beforehand.
100 *
101 * @param pkt The packet.
102 * @param n The size of the packet.
103 * @return The magic number of the packet.
104 * */
105 uint16_t dtPktMagicNo(uint8_t pkt[], size_t n)
106 {
107     return ((pkt[0] << 8) | pkt[1]);
108 }
109
110 /**
111  * Returns the type of packet.
112  * No checking is done beforehand.
113  *
114  * @param pkt The packet.
115  * @param n The size of the packet.
116  * @return The type of the packet.
117  * */
118 uint16_t dtPktType(uint8_t pkt[], size_t n)
119 {
120     return ((pkt[2] << 8) | pkt[3]);
121 }
122
123 /**
124  * Returns true if reqType is a valid type of request.
125  *
126  * @param reqType The request type.
127  * @return True if reqType is either REQ_DATE or REQ_TIME.
128  * */
129 bool validReqType(uint16_t reqType)
130 {
131     return (reqType == REQ_DATE || reqType == REQ_TIME);
132 }
133
134 /**
135  * Returns true if langCode denotes a valid language.
136  * Valid languages codes are LANG_ENG for English,
137  * LANG_MAO for Maori or LANG_GER for German.
138  *
139  * @param langCode The language code.
140  * @return True if langCode is valid.
141  * */
142 bool validLangCode(uint16_t langCode)
143 {
144     return (langCode == LANG_ENG || langCode == LANG_GER || langCode == LANG_MAO);
145 }
146
147 /**
148  * Constructs a DT Response packet.
149  *
150  * @param pkt A pointer to the packet.
151  * @param n The size of the packet. Must be equal to RES_PKT_LEN.
152  * @param reqType The type of request. Must be either REQ_DATE or REQ_TIME.
153  * @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,
237         ↪ now->tm_mday, now->tm_hour, now->tm_min);
238 }
239
240 /**
241  * Returns true if the DT Response packet is valid.
242  *
243  * @param pkt The packet.
244  * @param n The size of the packet.
245  * @return True if the packet is valid.
246  * */
247 bool dtResValid(uint8_t pkt[], size_t n)
248 {
249     if (n < 13) {
250         return false;
251     }
252
253     if (dtPktMagicNo(pkt, n) != MAGIC_NO) {
254         return false;
255     }
256
257     if (dtPktType(pkt, n) != PACKET_RES) {
258         return false;
259     }
260
261     if (dtResYear(pkt, n) >= 2100) {
262         return false;
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.

```

```
320  *
321  * @param The packet.
322  * @param The length of the packet.
323  * @return The month.
324  * */
325  uint8_t dtResMonth(uint8_t pkt[], size_t n)
326  {
327      return (pkt[8]);
328  }
329
330  /**
331   * Returns the day defined in the packet.
332   * No checking is done beforehand.
333   *
334   * @param pkt The packet.
335   * @param n The length of the packet.
336   * @return The day.
337   * */
338  uint8_t dtResDay(uint8_t pkt[], size_t n)
339  {
340      return (pkt[9]);
341  }
342
343  /**
344   * Returns the hour defined in the packet.
345   * No checking is done beforehand.
346   *
347   * @param pkt The packet.
348   * @param n The length of the packet.
349   * @return The hour.
350   * */
351  uint8_t dtResHour(uint8_t pkt[], size_t n)
352  {
353      return (pkt[10]);
354  }
355
356  /**
357   * Returns the minute defined in the packet.
358   * No checking is done beforehand.
359   *
360   * @param pkt The packet.
361   * @param n The length of the packet.
362   * @return The minute.
363   * */
364  uint8_t dtResMinute(uint8_t pkt[], size_t n)
365  {
366      return (pkt[11]);
367  }
368
369  /**
370   * Returns the length of the text in the packet.
371   * No checking is done beforehand.
372   *
373   * @param pkt The packet.
374   * @param n The length of the packet.
375   * @return The length of the text.
```

```
376  * */
377  uint8_t dtResLength(uint8_t pkt[], size_t n)
378  {
379      return (pkt[12]);
380  }
381
382  /**
383   * Copies the text from the packet and puts it into the char array text.
384   * No checking is done beforehand.
385   *
386   * @param pkt The packet.
387   * @param n The length of the packet.
388   * @param text The char array to store the text in.
389   * @param textLen A pointer where the length of the text is to be stored.
390   * */
391  void dtResText(uint8_t pkt[], size_t n, char text[], size_t* textLen)
392  {
393      *textLen = dtResLength(pkt, n);
394      for (int i = 0; i < *textLen; i++) {
395          text[i] = pkt[13 + i];
396      }
397      text[*textLen] = 0;
398  }
399
400  /**
401   * Dumps the packet data to stdout.
402   *
403   * @param pkt The packet.
404   * @param n The length of the packet.
405   * */
406  void dtPktDump(uint8_t pkt[])
407  {
408      size_t n = dtPktLength(pkt);
409      for (int i = 0; i < n; i++) {
410
411          // print the character
412          printf("%02X ", pkt[i]);
413
414          // if we are at the end of the line, also print the ascii values
415          if ((i + 1) % 8 == 0) {
416
417              // draw the partition
418              printf("| ");
419
420              // draw the ascii value or a period
421              for (int j = i - 7; j <= i; j++) {
422                  if (isprint(pkt[j])) {
423                      putchar(pkt[j]);
424                  } else {
425                      putchar('.');
426                  }
427              }
428              putchar('\n');
429          }
430      }
431  }
```

```

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.

```

1 // server.h
2
3 #ifndef SERVER_H
4 #define SERVER_H
5
6 bool readPorts(char** argv, uint16_t* ports);
7 void serve(uint16_t ports[]);
8 void handleSignal(int sig);
9
10 #endif

```

```

1 // server.c
2
3 #include <arpa/inet.h>
4 #include <signal.h>
5 #include <stdlib.h>
6 #include <stdint.h>
7 #include <stdio.h>
8 #include <stdbool.h>
9 #include <string.h>
10 #include <sys/select.h>
11 #include <sys/socket.h>
12 #include <sys/time.h>
13 #include <unistd.h>
14
15 #include "protocol.h"
16 #include "server.h"
17 #include "utils.h"
18
19 // the socket descriptors
20 // these must be global in order to safely close them on SIGINT
21 int socket_fds[3];
22
23 /**
24  * Usage: server <english port> <te reo maori port> <german port>
25  * */
26 int main(int argc, char** argv)
27 {
28     uint16_t ports[3] = {0};
29
30     // validate the arguments
31     if (argc != 4) {
32         error("server must receive exactly 3 arguments", 1);
33     }
34
35     // read the ports into the ports array
36     if (!readPorts(argv, ports)) {
37         char msg[52] = {0};
38         sprintf(msg, "ports must be between %u and %u (inclusive)", MIN_PORT_NO,
39             ↵ MAX_PORT_NO);
39         error(msg, 1);
40     }

```



```
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],
108         ↪ sizeof(server_addr[i])) < 0) {
109         error("could not bind to socket", 2);
110     }
111
112     // print some information and listen
113     printf("Listening on port %u for %s requests...\n", ports[i], getLangName(i +
114         ↪ 1));
115
116     listen(socket_fds[i], 5);
117
118 }
119
120 // loop forever
121 while (true) {
122
123     // holds the client address information
124     struct sockaddr_in client_addr;
125
126     // the length of the client address data struct
127     socklen_t client_addr_len = sizeof(client_addr);
128
129     // holds the IP address of the client
130     char client_ip_address_string[INET_ADDRSTRLEN];
131
132     // the active socket that received the request
133     // used to determine the port number and to send a response
134     int active_socket_fd = -1;
135
136     // the type of request we are handling, read from the request packet
137     uint16_t request_type = 0;
138
139     // the language the user requested the response in
140     // this is based on the port they connect to
141     uint16_t language_code = 0;
142
143     // holds the number of bytes received by the server for a request
144     int bytes_received;
145
146     // the buffer to place the received data
147     uint8_t buffer[256];
148
149     // the buffer to hold the response data
150     uint8_t response[RES_PKT_LEN];
151
152     // 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,
161                               ↪ NULL);
162
163     // if there was an error selecting
164     if (selectResult == -1) {
165         error("select failed", 4);
166     }
167
168     // iterate through the pollfd values until one is ready to receive data
169     // store the socket descriptor and the language code
170     for (int i = 0; i < 3; i++) {
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,
189               ↪ INET_ADDRSTRLEN);
190
191     // print the date, time and the ip address of the client
192     printCurrentDateTimeString();
193     printf(" - %s - ", client_ip_address_string);
194
195     // if an error occurred during reading the information, print an error
196     if (bytes_received < 0) {
197         printf("network error - packet discarded\n");
198         continue;
199     }
200
201     // handle the data
202     if (!dtReqValid(buffer, bytes_received)) {
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),
211             ↪ getReqTypeString(request_type));
212
213         // zero the response packet buffer
214         memset(response, 0, RES_PKT_LEN);
215
216         // construct the response packet
217         size_t b = dtResNow(response, RES_PKT_LEN, request_type, language_code);
218
219         // attempt to sent the response packet
220         if (sendto(active_socket_fd, response, b, 0, (struct sockaddr *)
221             ↪ &client_addr, client_addr_len) < 0) {
222             printf("response failed to send\n");
223         } else {
224             printf("response sent\n");
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.

```

1 // client.h
2
3 #ifndef CLIENT_H
4 #define CLIENT_H
5
6 #include <stdint.h>
7
8 int main(int argc, char** argv);
9 void request(uint16_t reqType, char* ip_addr, char* port);
10
11 #endif

```

```

1 // client.c
2
3 #include <arpa/inet.h>
4 #include <netdb.h>
5 #include <netinet/in.h>
6 #include <stdint.h>
7 #include <stdio.h>
8 #include <stdlib.h>
9 #include <string.h>
10 #include <sys/select.h>
11 #include <sys/socket.h>
12 #include <sys/types.h>
13 #include <time.h>
14 #include <unistd.h>
15
16 #include "client.h"
17 #include "protocol.h"
18 #include "utils.h"
19
20 /**
21  * Usage: client <time/date> <ip address> <port>
22  * */
23 int main(int argc, char** argv)
24 {
25     uint16_t request_type, port;
26
27     // validate the number of arguments passed in
28     if (argc != 4) {
29         error("client expects exactly 4 arguments", 1);
30     }
31
32     // set the request_type based on the first argument
33     if (strcmp(argv[1], "date") == 0) {
34         request_type = REQ_DATE;
35     } else if (strcmp(argv[1], "time") == 0) {
36         request_type = REQ_TIME;
37     } else {
38         error("first argument should be either \"date\" or \"time\"", 1);
39     }
40

```

```
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,
46             ↵ MAX_PORT_NO);
47     error(msg, 1);
48 }
49
50 // send a request
51 request(request_type, argv[2], argv[3]);
52
53 return 0;
54 }
55
56 /**
57  * Sends a request to the server.
58  *
59  * @param request_type The type of request, either REQ_DATE or REQ_TIME.
60  * @param ip_address_string The ip address of the server as a string.
61  * @param port The port the server is listening on.
62  */
63 void request(uint16_t request_type, char* ip_address_string, char* port_string)
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 *)
168         ↪ server_address->ai_addr, server_address->ai_addrlen) < 0) {
169         error("could not send packet", 2);
170     }
171
172     // set the timeout to be one second, this must be set again because select()
173     ↪ modifies the timeout
174     timeout.tv_sec = 1;
175     timeout.tv_usec = 0;
176
177     // set the socket_set, this must be set again because select() modifies socket_set
178     FD_ZERO(&socket_set);
179     FD_SET(client_socket, &socket_set);
180
181     // Wait for the socket to be readable
182     select_result = select(client_socket + 1, &socket_set, NULL, NULL, &timeout);
183
184     // print an error if something went wrong while selecting
185     if (select_result < 0) {
186         error("could not select", 4);
187     }
188
189     // print an error if a timeout occurred
190     if (select_result == 0) {
191         error("select timed out", 4);
192     }
193
194     // attempt to receive the response
195     if (recvfrom(client_socket, buffer, RES_PKT_LEN, 0, (struct sockaddr *)
196         ↪ &server_address, &server_address_len) < 0) {
197         error("could not recieve packet", 2);
198     }
199
200     // free the memory used by getaddrinfo
201     freeaddrinfo(addresses);
202
203     // close the socket
204     close(client_socket);
205
206     // 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.

```
1  // protocol.test.c
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <stdint.h>
6
7  #include "../protocol.h"
8  #include "../utils.h"
9
10 int main(void)
11 {
12     uint16_t failures = 0;
13
14     // ** dtReq **
15     // create a request packet with a size too small
16     size_t smallReqPktLen = REQ_PKT_LEN - 1;
17     uint8_t smallReqPkt[smallReqPktLen];
18
19     if (dtReq(smallReqPkt, smallReqPktLen, REQ_DATE) != 0) {
20         failures++;
21         fail("dtReq", "n is too small");
22     }
23
24     // create a request packet with a size too large
25     size_t largeReqPktLen = REQ_PKT_LEN + 1;
26     uint8_t largeReqPkt[largeReqPktLen];
27
28     if (dtReq(largeReqPkt, largeReqPktLen, REQ_DATE) != 0) {
29         failures++;
30         fail("dtReq", "n is too large");
31     }
32
33     // create a request packet with an invalid reqType
34     uint8_t reqPktDate[REQ_PKT_LEN] = {0};
35
36     if (dtReq(reqPktDate, REQ_PKT_LEN, 99) != 0) {
37         failures++;
38         fail("dtReq", "invalid reqType");
39     }
40
41     // create a valid request packet
42     if (dtReq(reqPktDate, REQ_PKT_LEN, REQ_DATE) != REQ_PKT_LEN) {
43         failures++;
44         fail("dtReq", "reqType is valid");
45     }
46
47     uint8_t reqPktTime[REQ_PKT_LEN] = {0};
48
49     if (dtReq(reqPktTime, REQ_PKT_LEN, REQ_TIME) != REQ_PKT_LEN) {
50         failures++;
51         fail("dtReq", "reqType is valid");
52     }
53 }
```

```
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
```

```
271     timeEngResPkt[9] = 0;
272     if (dtResValid(timeEngResPkt, RES_PKT_LEN)) {
273         failures++;
274         fail("dtResValid", "day should be too small");
275     }
276
277     // check with a packet with a day too large
278     timeEngResPkt[9] = 32;
279     if (dtResValid(timeEngResPkt, RES_PKT_LEN)) {
280         failures++;
281         fail("dtResValid", "day should be too large");
282     }
283     timeEngResPkt[9] = (uint8_t)(10);
284
285     // check with a packet with an hour too large
286     timeEngResPkt[10] = 24;
287     if (dtResValid(timeEngResPkt, RES_PKT_LEN)) {
288         failures++;
289         fail("dtResValid", "hour should be too large");
290     }
291     timeEngResPkt[10] = (uint8_t)(12);
292
293     // check with a packet with a minute too large
294     timeEngResPkt[11] = 60;
295     if (dtResValid(timeEngResPkt, RES_PKT_LEN)) {
296         failures++;
297         fail("dtResValid", "minute should be too large");
298     }
299     timeEngResPkt[11] = (uint8_t)(45);
300
301     // check with valid packets
302     if (!dtResValid(timeEngResPkt, dtPktLength(timeEngResPkt))) {
303         failures++;
304         fail("dtResValid", "english time packet should be valid");
305         dtPktDump(timeEngResPkt);
306     }
307
308     if (!dtResValid(dateEngResPkt, dtPktLength(dateEngResPkt))) {
309         failures++;
310         fail("dtResValid", "english date packet should be valid");
311         dtPktDump(dateEngResPkt);
312     }
313
314     // ** dtResLangCode **
315     // check that the lang code is extracted
316     if (dtResLangCode(dateEngResPkt, RES_PKT_LEN) != LANG_ENG) {
317         failures++;
318         fail("dtResLangCode", "lang code is not extracted");
319     }
320
321     // ** dtResYear **
322     // check that the year is extracted
323     if (dtResYear(dateEngResPkt, RES_PKT_LEN) != 2018) {
324         failures++;
325         fail("dtResYear", "year is not extracted");
326     }
```

```
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.

```
1  # Makefile
2
3  CFLAGS = -std=gnu99 -Werror -Wall -I ./src/
4
5  all: libs server client
6
7  libs:
8      gcc $(CFLAGS) -c -o obj/protocol.o src/protocol.c
9      gcc $(CFLAGS) -c -o obj/utils.o src/utils.c
10
11  server: libs src/server.c
12      gcc $(CFLAGS) -o bin/server obj/protocol.o obj/utils.o src/server.c
13
14  client: libs src/client.c
15      gcc $(CFLAGS) -o bin/client obj/protocol.o obj/utils.o src/client.c
16
17  test: libs src/test/protocol.test.c
18      gcc $(CFLAGS) -o bin/test/protocol.test obj/protocol.o obj/utils.o
19      ↪ src/test/protocol.test.c
20
21  pdf:
22      cd report; pdflatex --shell-escape -undump=pdflatex report.tex
23
24  clean:
25      rm -v obj/protocol.o obj/utils.o
26      rm -v bin/server
27      rm -v bin/client
28      rm -v bin/test/*
29      rm report/report.pdf
```