Servicios Telemáticos

Práctica Final

Entrega Anticipada

Autor:	Emilio Domínguez Sánchez
Convocatoria:	Junio 2020
Grupo:	1.4

Índice

1. Introducción

La asignatura de Servicios Telemáticos del grado en Ingeniería Informática consta de una parte práctica que consiste en configurar y poner en marcha un conjunto de servicios telemáticos. El enunciado de la práctica describe los objetivos. En esta memoria se recoge la implementación del enunciado así como el proceso de toma de decisiones.

2. Servidor Web HTTP

2.1. Introducción

Implementar y desplegar un servidor HTTP es la tarea principal de estas prácticas. De aquí en adelante, cuando hablemos del servidor nos referiremos a la implementación en lenguaje C que hemos llevado a cabo.

2.2. Usos de C++

Como comentaba, este apartado de prácticas debía realizarse en C. Sin embargo, comenté con mi profesor de prácticas la posibilidad de utilizar C++utilizando las mismas llamadas al sistema. Sin embargo, para cumplir con los requisitos de la entrega he mantenido el código como si fuese C. En ese sentido, el grueso del programa es código C. En concreto, he utilizado C++para

- Una clase para imprimir los registros. Permite escribir al registro utilizando la sintaxis que muestro en el código 1.
- Y en algunas funciones para pasar algunos valores por referencia. Cuando se quiere pasar por referencia un puntero en C, la sintaxis no es cómoda.

Código 1: Uso de las clases log y logerr

```
Code:   log << "alberto, tenemos un problema de tipo " << type << endl;
Output: INFO(socket ---): alberto, tenemos un problema de tipo ---
Code:   log << endl;
Output:
Code:   log << "mensaje de log 1" << endl;
        log << "mensaje de log 2\n";
Output: INFO(socket ---): mensaje de log 1
        INFO(socket ---): mensaje de log 2
Code:   logerr << "falló una llamada al sistema" << endl;
Output: ERROR: errno=--- exiting pid=---: falló una llamada al sistema
Code:   logerr << "problemas Mike!" << endl << panic();
Output: ERROR: errno=--- exiting pid=---: problemas Mike!
-> Program finishes execution with code -1
```

2.3. Inicio del servidor

El objetivo básico de un servidor web http es recibir peticiones, que vendrán en forma de un mensaje `HTTP request`, y tratarlas. Normalmente, tratarlas significa devolver un recurso al que se intenta acceder.

La función main del servidor se encuentra dentro del fichero main.cpp. Al inicio del programa (código 2), el objetivo es que el servidor sea ejecutado desde la terminal y funcione como un demonio (es decir, en segundo plano). Para ello, creamos un hijo e ignoramos la señal `SIGHUP`[1]. El proceso padre finaliza automáticamente para devolver el control a la terminal.

Código 2: Fichero `src/main.cpp`

```
    // Behave as a daemon.
    switch (fork()) {
        case -1:
            logerr << "fork() fail" << endl << panic();
        case 0:
            signal(SIGCHLD, SIG_IGN);  // Ignore children
            signal(SIGHUP,  SIG_IGN);  // Ignore Hang up from terminal
            log << "web server starting on port " << port <<  "...\n";
            break;
        default:
            return 0;              // Return control to the user instantly.
    }
```

Cuando el servidor recibe una nueva conexión TCP de un cliente, crea un proceso hijo que se encarga de atenderlo y espera hasta recibir otra conexión (código 3).

Código 3: Fichero `src/main.cpp`

```
    // Set up the network socket
    if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        logerr << "Coudln't set up the network socket" << endl << panic();


    // Create an structure for the socket (IP address and port) where the server listens.
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);  // Listen to any possible IP
    serv_addr.sin_port = htons(port);               // on port `port`.

    if (bind(listenfd, (struct sockaddr*) &serv_addr, sizeof(serv_addr)) < 0)
        logerr << "bind fail" << endl << panic();
    if (listen(listenfd, 64) < 0)
        logerr << "listen fail" << endl << panic();

    while (true) {
        socklen_t length = sizeof(cli_addr);
        if ((socket_fd = accept(listenfd, (struct sockaddr *) &cli_addr, &length)) < 0)
            logerr << "accept fail" << endl << panic();
        switch (fork()) {
            case -1:
                logerr << "fork() fail" << endl << panic();
            case  0:
                close(listenfd);
                client_fd = socket_fd;
                deal_with_client(); // El hijo termina tras llamar a esta función
                logerr << "This shouldn't be printed." << endl << panic(0);
            default:
                close(socket_fd); //TODO error handling?
        }
    }
```

[1]La señal `SIGHUP` es una señal que se envía a un proceso cuando la terminal que lo controla finaliza.

2.4. Intercambio de mensajes

La complejidad del servidor está en el trabajo que realiza el proceso para atender al cliente. La función C que maneja la conversación es deal_with_client .

- Para la lectura se utiliza un buffer de 8 KiB. El servidor lee (código 4) datos de la conexión TCP hasta completar la cabecera de un mensaje HTTP (indicada por la secuencia "\r\n\r\n").
- Tras completar la cabecera de un mensaje, se interpreta la primera línea, llamada `status-line` en el protocolo HTTP, para determinar el tipo de mensaje y llamar a una función que lo procese (código 5). Esto implica que la cabecera ha de caber en el buffer. Es decir, hay un límite de 8 KiB para el tamaño de la cabecera; y el servidor enviará un mensaje de error en caso de que se supere este tamaño. En la práctica es más que suficiente. Ningún mensaje, aún usando todos los campos disponibles en el protocolo HTTP, utilizaría tanto espacio.
- El servidor acepta mensajes de tipo GET y POST. Del campo de cabecera `content-length` obtiene el tamaño del cuerpo del mensaje, que incorpora al buffer.
- El servidor no conoce el tamaño de la cabecera de antemano, lo que significa que es posible leer más bytes que los que ocupa un mensaje[2]. Las funciones process_get y process_post devuelven un referencia al primer byte que no formaba parte de su mensaje. La función deal_with_client lo utiliza para completar la cabecera del siguiente mensaje. De esta manera, la función deal_with_client no necesita inspeccionar la cabecera (código 6).

Código 4: Fichero `src/message_processing.cpp`

```
1  // Read HTTP Header (maximum length of BUFFER_CAP)
2  bool aux = strstr(buf, "\r\n\r\n") != NULL;
3  for (bool header_complete = aux; !header_complete; buf[bufLen] = '\0') {
4      // Perform read operation
5      int r = read(client_fd, buf+bufLen, BUFFER_CAP-bufLen);
6      if (r < 0) {
7          logerr << "Error while reading HTTP header." << endl << panic(SERVER_ERR);
8      }
9
10     // Check for the end of the header ("\r\n\r\n")
11     header_complete = strstr(buf+max(bufLen-3, 0), "\r\n\r\n") != NULL;
12     bufLen += r;
13
14     // Error Control
15     if (!header_complete) {
16         if (bufLen == BUFFER_CAP) {
17             send_request_entity_too_large();
18             log << "The client filled the buffer with the header"
19                 << endl << panic(PRECOND_ERR);
20         }
21         if (r == 0) {
22             if (bufLen > 0) {
23                 send_bad_request();
24                 log << buf << endl;
25                 log << "The client stopped the connection in the middle "
26                     << "of the header" << endl << panic(PRECOND_ERR);
27             } else {
28                 log << "No new message. The connection stopped "
29                     << "succesfully." << endl << panic(0);
30             }
```

[2]Enviar dos mensajes antes de recibir la respuesta del primero se llama `pipelining`. Según nuestras pruebas, la versión de firefox que utilizamos en la máquina virtual no implementa `pipelining`.

```
31                  }
32                  FD_ZERO(&readFds);
33                  FD_SET(client_fd, &readFds);
34                  timeout.tv_sec = LATENCY_TIME_OUT;
35                  timeout.tv_usec = 0;
36                  if (!select(client_fd+1, &readFds, NULL, NULL, &timeout)) {
37                      log << "LATENCY_TIME_OUT" << endl << panic(PRECOND_ERR);
38                      break;
39                  }
40              }
41          }
```

Código 5: Fichero `src/message_processing.cpp`

```
1           Request_Line rl;
2           int rl_size = parse_request_line(rl, buf);
3           if (rl_size < 0) {
4               send_bad_request();
5               log << "The client sent an invalid request line" << endl << panic(PRECOND_ERR);
6           }
7
8           bool persistent = true;
9           if (strcmp(rl.version, "HTTP/1.0") == 0) {
10              persistent = false;
11          } else if (strcmp(rl.version, "HTTP/1.1") != 0){
12              send_version_not_supported();
13              log << "The client sent an unsuported version" << endl << panic(PRECOND_ERR);
14          }
```

Código 6: Fichero `src/message_processing.cpp`

```
1           char* consumed;
2           if (strcmp(rl.method, "GET") == 0) {
3               consumed = process_get(rl, buf, bufLen, buf+rl_size, persistent);
4           } else if (strcmp(rl.method, "POST") == 0) {
5               consumed = process_post(rl, buf, bufLen, buf+rl_size, persistent);
6           } else {
7               send_not_implemented();
8               log << "Method " << rl.method << " not allowed" << endl << panic(PRECOND_ERR);
9           }
10
11          /* Move the excessive data to the beginning of the buffer */
12          for (int i = 0; i < bufLen; i++) {
13              buf[i] = consumed[i];
14          }
15
16          if (not persistent) {
17              log << "No persistency. The connection finished succesfully." << endl << panic(0);
18          }
```

2.5. Persistencia

Como comentábamos en la sección anterior, el servidor está preparado para aceptar más de un mensaje en la misma conexión. Es lo que se conoce como persistencia en el protocolo HTTP. Las funciones que inspeccionan la cabecera comprueban si el cliente busca mantener la conexión según el valor de la línea de cabecera `Connection` y la versión HTTP (el protocolo `HTTP/1.0` no se diseñó enfocado a persistencia).

El servidor se ha configurado con un timeout de 5 s que anuncia en la cabecera `Keep-Alive`. Se trata del máximo tiempo que el servidor mantendrá la conexión abierta antes de recibir el primer

byte del siguiente mensaje (como especifica el protocolo). No obstante, incurrimos en un riesgo de bloqueo si no incluimos otro timeout cuando se espera para completar un mensaje. Un cliente malintencionado podría dejar una conexión abierta indefinidamente, lo que se traduce en un riesgo de negación de servicio (`DoS`). Por tanto, hemos incluido un timeout de 1 s.

2.5.1. Tiempos de timeout en la lectura de cabecera

Para el control de los tiempos de timeout hemos utilizado la llamada al sistema select de linux. Es una directiva de sincronización que permite bloquearse a la espera de datos de un `socket` durante un tiempo determinado.

La implementación del timeout entre mensajes se puede ver en el código 7. Es importante ver que hay una comprobación antes sobre el tamaño del buffer de lectura. El motivo es el que ya hemos comentado. Puede ser que hayamos leído parte del mensaje siguiente (incluso entero), en cuyo caso no tenemos por qué esperar.

La implementación del timeout para completar la cabecera se puede ver el código 8. En ambos casos, si el cliente no envía datos en el tiempo establecido se aborta la conexión. En el caso de que no haya comenzado un mensaje, se trata de un corte normal. En cambio, si ya había comenzado el mensaje y no nos ha llegado el resto del mensaje puede ser que la conexión funcione mal o que sea el cliente el que funcione incorrectamente. En cualquier caso, cortamos la conexión y terminamos con un código de error.

Código 7: Fichero `src/message_processing.cpp`

```
    while (true) {
        // Wait (up to SERVER_TIME_OUT) for data to read
        log << "bufLen: " << bufLen << endl;
        if (bufLen == 0) {
            FD_ZERO(&readFds);
            FD_SET(client_fd, &readFds);
            timeout.tv_sec = SERVER_TIME_OUT;
            timeout.tv_usec = 0;
            if (!select(client_fd+1, &readFds, NULL, NULL, &timeout)) {
                log << "SERVER_TIME_OUT" << endl << panic(0);
            }
        }
```

Código 8: Fichero `src/message_processing.cpp`

```
            if (!header_complete) {
                FD_ZERO(&readFds);
                FD_SET(client_fd, &readFds);
                timeout.tv_sec = LATENCY_TIME_OUT;
                timeout.tv_usec = 0;
                if (!select(client_fd+1, &readFds, NULL, NULL, &timeout)) {
                    log << "LATENCY_TIME_OUT" << endl << panic(PRECOND_ERR);
                    break;
                }
            }
```

2.5.2. Tiempos de timeout en la lectura del cuerpo del mensaje

Para aquellos mensajes que contienen un cuerpo (mensajes POST principalmente) también tenemos que controlar el timeout si intentamos leer más datos. Si no, corremos el riesgo de mantener

un hilo infinitamente abierto ante un error o un cliente malintencionado. Se hace de forma similar al código 8 pero dentro de las funciones process_get y process_post.

2.5.3. Manejo de las cabeceras `Connection` y `Keep-Alive`

El protocolo HTTP exige al servidor que informe al cliente si va a mantener la conexión abierta y cuánto tiempo asegura que lo hará[3] . Lo primero se hace a través de la cabecera `Connection`, que puede tomar los valores `close` o `keep-alive`. El tiempo se especifica en la cabecera `keep-alive`. En nuestra implementación, las funciones que envían ficheros reciben un parámetro que indica si la conexión se mantendrá abierta.

2.6. Métodos GET y POST

La funciones process_get y process_post inspeccionan del buffer de lectura un mensaje (tipo GET o POST), leen datos del socket hasta completar el cuerpo del mensaje y devuelven el recurso que pide el cliente en forma de un mensaje `HTTP response`. En caso de que el mensaje no cumpla el formato o se produzca un error, se devuelve el mensaje de error que indique el protocolo y dependiendo del caso se aborta la ejecución.

Hemos aplicado el siguiente criterio para el corte de la conexión. Si el error es de formato, se considera que no se puede determinar dónde acabaría el mensaje y empezaría el siguiente, y por tanto se aborta la conexión. Lo mismo sucede si se produce un error interno; por ejemplo, por un fallo de una llamada al sistema. En cambio, un error como intentar acceder a un recurso que no existe se considera un error leve. Se devolvería un mensaje con el famoso código HTTP, `404 Not Found`, y se procesaría el siguiente mensaje. Se puede ver la implementación de la función process_get en el código 9.

Código 9: Fichero `src/message_processing.cpp`

```
1  /**
2   * @brief Processes a GET request
3   *
4   * This function may read extra data into the read buffer to complete
5   * its message. Therefore, any reference to a fragment of the message
6   * (like the request line) MUST be considered invalidated.
7   * The function may update the persistent behaviour according the the
8   * header connection field.
9   */
10 char* process_get(const Request_Line& rl, char* buf, int& bufLen,
11                   char* bufoff, bool& persistent) {
12     log << "New GET Request for " << rl.request_uri << " " << rl.version << endl;
13
14     if (!valid_uri(rl.request_uri)) {
15         send_forbidden();
16         logerr << rl.request_uri << " is not a valid uri" << endl << panic(PRECOND_ERR);
17     }
18
19     bool contains_host = false;
20     int content_length = 0;
21     // Read header fields
```

[3]Aunque el servidor intente mantener siempre la conexión abierta, puede no poder hacerlo por dos motivos. Porque el cliente pida mantenerla cerrada o porque haya que abortar la conexión, por ejemplo tras un error de formato.

```
    while (bufoff[0] != '\r' || bufoff[1] != '\n') {
        Header_Field hf;
        int hf_size = parse_header_field(hf, bufoff);
        if (hf_size < 0) {
            send_bad_request();
            logerr << "Incorrect header field format" << endl << panic(PRECOND_ERR);
        }
        if (strcmp(hf.field, "host") == 0) {
            contains_host = true;
        } else if (strcmp(hf.field, "connection") == 0) {
            if (strstr(hf.value, "close") != NULL) {
                persistent = false;
            }
        } else if (strcmp(hf.field, "content-length") == 0) {
            content_length = strtol(hf.value, NULL, 10);
        }
        bufoff += hf_size;
        // log << hf.field << ' ' << hf.value << endl;
    }
    bufoff += 2; // Discard last "\r\n".
    bufLen -= bufoff-buf;

    if (!contains_host) {
        send_bad_request();
        log << "The client didn't send a host field!" << endl << panic(PRECOND_ERR);
    }

    if (content_length != 0) {
        send_not_implemented();
        log << "The client sent content inside GET, which isn't supported."
            << endl << panic(PRECOND_ERR);
    }

    /**
     * Now that we've interpreted the complete header, we must
     * evaluate the type of file that the client is asking for
     * and return the file if it's supported or the appropiate
     * error.
     */

    // Make the route relative.
    char* rel_uri = rl.request_uri;
    while (*rel_uri == '/')
        rel_uri++;
    const char* pathname = *rel_uri == '\0'? "index.html" : rel_uri;

    log << "GET process complete. Send: " << pathname << endl;
    send_static_file(STATUS_OK, pathname, persistent);
    return bufoff;
}
```

La función process_post se comporta de manera similar a la hora de inspeccionar la cabecera, pero la funcionalidad es limitada. Al tratarse de un proyecto de prácticas, el contenido de prueba del servidor solo contiene un formulario HTML. Si se recibe la cadena "email=emilio.dominguezs%40um.es" devuelve una página de felicitación (código 10).

Código 10: Fichero `src/message_processing.cpp`

```
    if (strcmp(bufoff, "email=emilio.dominguezs%40um.es") == 0) {
        log << "POST Success!" << endl;
        send_static_file(STATUS_OK, "success.html", persistent, true);
    } else {
        log << "POST Fail!" << endl;
        send_static_file(STATUS_OK, "failure.html", persistent);
    }
```

2.7. Mensajes HTTP Response

La función send_static_file envía un fichero del sistema a través de un mensaje HTTP. Es una función genérica a la que llaman process_get, process_post y otras funciones que envían mensajes de error (como send_bad_request).

La función utiliza un buffer de escritura de 2·8 KiB para poder hacer lecturas (de disco) y escrituras de 8 KiB y minimizar las llamadas al sistema.

- Mientras el buffer no contenga más de 8 KiB y no se haya leído el fichero al completo, se hace una operación de lectura de 8 KiB del fichero (la operación puede leer menos datos). Como el buffer es de $2 \cdot 8$ KiB, siempre hay espcacio para los datos.
- Mientras el buffer contenga más de 8 KiB o el fichero se haya leído completamente y queden datos por enviar, se ejecuta una operación de escritura de 8 KiB. A continuación se desplaza el exceso de datos al comienzo del buffer para mantener todo el espacio libre a continuación del espacio usado.

Código 11: Fichero `src/message_processing.cpp`

```
1  int send_static_file(int code, const char* pathname,
2                       bool keepAlive = true, bool isInternalFile = false);
3
4  int send_not_found(bool keepAlive = true) { return send_static_file(STATUS_NOT_FOUND,
5                                                 "data/not_found.html", keepAlive, true); }
6
7  int send_forbidden(bool keepAlive = true) { return send_static_file(STATUS_FORBIDDEN,
8                                                 "data/forbidden.html", keepAlive, true); }
9
10 int send_bad_request(bool keepAlive = false) { return send_static_file(STATUS_BAD_REQUEST,
11                                                    "data/bad_request.html", keepAlive, true); }
12
13 int send_unauthorized(bool keepAlive = true) { return send_static_file(STATUS_UNAUTHORIZED,
14                                                    "data/unathorized.html", keepAlive, true); }
15
16 int send_internal_server_error(bool keepAlive = false) {
17     if (ct == NULL) {
18         if (isInternalFile) return send_status_line(code);
19         else return send_unsupported_media_type();
20     }
21     if (access(pathname, F_OK)) {
22         if (isInternalFile) return send_status_line(code);
23         else return send_not_found();
24     }
25     if (access(pathname, R_OK)) {
26         if (isInternalFile) return send_status_line(code);
27         else return send_forbidden();
28     }
29
30     int resource_fd = open(pathname, O_RDONLY);
31     if (resource_fd < 0) {
32         if (isInternalFile) return send_status_line(code);
33         else {
34             send_internal_server_error();
35             logerr << "fstat fail" << endl << panic();
36         }
37     }
38     struct stat statbuf;
39     if (fstat(resource_fd, &statbuf) < 0) {
40         if (isInternalFile) return send_status_line(code);
41         else {
42             send_internal_server_error();
43             logerr << "fstat fail" << endl << panic();
```

```
44          }
45      }
46
47      // Composing the header
48      char buf[2*BUFFER_CAP];
49      int bufLen = sprintf(buf, "%s %d %s\r\n", HTTP_VERSION, code, to_reason_phrase(code));
50      bufLen += sprintf(buf+bufLen, "Server: %s\r\n", SERVER_NAME);
51      time_t raw_time;
52      char* time_string;
53      if (time(&raw_time) < 0 || (time_string = ctime(&raw_time)) == NULL) {
54          logerr << "error getting time!" << endl;
55      } else {
56          bufLen += sprintf(buf+bufLen, "Date: %s", time_string);
57          buf[bufLen-1] = '\r';
58          buf[bufLen] = '\n';
59          bufLen++;
60      }
61      bufLen += sprintf(buf+bufLen, "Connection: %s\r\n", keepAlive? "keep-alive" : "close");
62      if (keepAlive) {
63          bufLen += sprintf(buf+bufLen, "Keep-Alive: timeout=%d\r\n", SERVER_TIME_OUT);
64      }
65      bufLen += sprintf(buf+bufLen, "Content-Type: %s\r\n", ct);
66      bufLen += sprintf(buf+bufLen, "Content-Length: %ld\r\n", statbuf.st_size);
67
68      bufLen += sprintf(buf+bufLen, "\r\n");
69
70      // Message writting.
71      int r;
72      char* off = buf;
73      do {
74          r = read(resource_fd, off+bufLen, BUFFER_CAP);
75          if (r < 0) {
76              logerr << "error de lectura wey " << endl << panic(-1);
77          }
78          bufLen += r;
79          // Write in blocks of BUFFER_CAP (except possibly the last one)
80          while (bufLen > BUFFER_CAP || (bufLen > 0 && r == 0)) {
81              int w = write(client_fd, off, min(BUFFER_CAP, bufLen));
82              if (w < 0) {
83                  logerr << "error de escritura wey" << endl;
84              }
85              off += w;
86              bufLen -= w;
87          }
88
89          if (off + bufLen > buf+BUFFER_CAP) {
90              for (int i = 0; i < bufLen; i++) {
91                  buf[i] = off[i];
92              }
93              off = buf;
94          }
95      } while (r != 0);
96      return code;
97  }
```

2.8. Otras anotaciones sobre el código fuente

Esta sección está dedicada a partes del código que no forman parte de la lógica principal del programa.

- El fichero http_parsing.cpp contiene funciones que sirven para separar los campos de un mensaje HTTP. El mensaje original se modifica y se insertan caracteres nulos en el lugar de

algunos delimitadores y se devuelven referencias al caracter donde empezaba cada campo. Como se mantienen referencias al mensaje original, el código es muy eficiente, pero al leer información en el buffer encima de la información actual las referencias se vuelven inválidas. Las funciones también comprueban fallos en el formato de la cabecera.

- El fichero defs.hpp contiene definiciones generales para el servidor. Por ejemplo, permite a un usuario cambiar los tiempos de timeout o las asociaciones `MIME` que identifican el tipo de fichero según su extensión.

2.9. Mejoras

Esta sección está pendiente. En la entrega de final incluirá las mejoras que se hayan implementado.

2.10. Ejemplos de funcionamiento y capturas del tráfico de red

Como prueba del funcionamiento del servidor hemos realizado las siguientes capturas del tráfico.

- La captura `capture-home-page.pcapng` muestra los mensajes intercambiados para cargar la página de inicio.
- La captura `success-persistent.pcapng` contiene además los mensajes tras introducir el correo electrónico en el formulario. Puede servir para verificar la persistencia, viendo como el servidor mantiene la conexión abierta unos segundos esperando a la segunada acción.
- La captura `capture-not-found.pcapng` muestra una respuesta `404 Not Found`.
- La captura `capture-unsupported-media-type.pcapng` muestra una respuesta `415 Unsupported Media Type`.
- La captura `capture-request-entity-too-large.pcapng` nos muestra el comportamiento del servidor ante un mensaje POST de 9 kB. Aunque como hemos explicado antes el servidor está preparado para procesar mensajes de tamaño arbitrario leyéndolos en bloques de 8 KiB, hemos establecido el máximo en 8 KiB. En la captura podemos ver que el servidor devuelve el mensaje de error `413 Request Entity Too Large` y corta la conexión.
- La captura `capture-pipelining.pcapng` nos muestra la respuesta del servidor ante un cliente que lleva a cabo `pipelining` con un ejemplo en el que llegan 4 mensajes GET a la vez.

3. Horas de Trabajo

Para contabilizar las horas trabajadas en esta entrega hemos utilizado la herramienta `progesa-test` de la Facultad de Informática. A continuación se puede ver una tabla con el tiempo dedicado a cada apartado de la práctica.

Entrega Anticipada	
Actividad	Trabajo Autónomo
Implementación básica servidor HTTP	17 h 24 min
Documentación	5 h 45 min
Revisión	35 min
TOTAL	23 h 44 min