

Servicios Telemáticos

Práctica Final

Entrega Anticipada

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

Índice

1. Introducción	2
2. Servidor Web HTTP	2
2.1. Introducción	2
2.2. Usos de C++	2
2.3. Inicio del servidor	2
2.4. Intercambio de mensajes	4
2.5. Persistencia	5
2.5.1. Tiempos de timeout en la lectura de cabecera	6
2.5.2. Tiempos de timeout en la lectura del cuerpo del mensaje	6
2.5.3. Manejo de las cabeceras CONNECTION y KEEP-ALIVE	7
2.6. Métodos GET y POST	7
2.7. Mensajes HTTP Response	9
2.8. Otras anotaciones sobre el código fuente	10
2.9. Mejoras	11
2.10. Ejemplos de funcionamiento y capturas del tráfico de red	11
3. Despliegue de Servicios Telemáticos	12
3.1. Escenario Desarrollado y Versiones del Software	12
3.2. Servidor DNS	12
3.2.1. Configuración	12
3.2.2. Capturas Wireshark	14
3.3. Correo SMTP/POP	14
3.3.1. Usuarios de correo	15
3.3.2. Capturas Wireshark	15
3.4. Servidor HTTP y HTTPs basado en APACHE	16
3.4.1. Configuración	16
3.5. Certificación HTTPs a través de OPENSSL	17
3.5.1. Capturas Wireshark	17
3.6. IPsec con STRONGSWAN	17
3.6.1. Configuración	18
3.6.2. Capturas Wireshark	20
4. Horas de Trabajo	20

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

```
1 Code: log << "alberto , tenemos un problema de tipo " << type << endl;
2 Output: INFO(socket ---): alberto , tenemos un problema de tipo ---
3 Code: log << endl;
4 Output:
5 Code: log << "mensaje de log 1" << endl;
6       log << "mensaje de log 2\n";
7 Output: INFO(socket ---): mensaje de log 1
8       INFO(socket ---): mensaje de log 2
9 Code: logerr << "falló una llamada al sistema" << endl;
10 Output: ERROR: errno=--- exiting pid=---: falló una llamada al sistema
11 Code: logerr << "problemas Mike!" << endl << panic();
12 Output: ERROR: errno=--- exiting pid=---: problemas Mike!
13 -> 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 SIGHUP¹. El proceso padre finaliza automáticamente para devolver el control a la terminal.

Código 2: Fichero src/main.cpp

```

1 // Behave as a daemon.
2 switch (fork()) {
3     case -1:
4         logerr << "fork() fail" << endl << panic();
5     case 0:
6         signal(SIGCHLD, SIG_IGN); // Ignore children
7         signal(SIGHUP, SIG_IGN); // Ignore Hang up from terminal
8         log << "web server starting on port " << port << "...\\n";
9         break;
10    default:
11        return 0; // Return control to the user instantly.
12 }

```

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

```

1 // Set up the network socket
2 if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
3     logerr << "Coudln't set up the network socket" << endl << panic();
4
5
6 // Create an structure for the socket (IP address and port) where the server listens.
7 serv_addr.sin_family = AF_INET;
8 serv_addr.sin_addr.s_addr = htonl(INADDR_ANY); // Listen to any possible IP
9 serv_addr.sin_port = htons(port); // on port 'port'.
10
11 if (bind(listenfd, (struct sockaddr*) &serv_addr, sizeof(serv_addr)) < 0)
12     logerr << "bind fail" << endl << panic();
13 if (listen(listenfd, 64) < 0)
14     logerr << "listen fail" << endl << panic();
15
16 while (true) {
17     socklen_t length = sizeof(cli_addr);
18     if ((socket_fd = accept(listenfd, (struct sockaddr *) &cli_addr, &length)) < 0)
19         logerr << "accept fail" << endl << panic();
20     switch (fork()) {
21         case -1:
22             logerr << "fork() fail" << endl << panic();
23         case 0:
24             close(listenfd);
25             client_fd = socket_fd;
26             deal_with_client(); // El hijo termina tras llamar a esta función
27             logerr << "This shouldn't be printed." << endl << panic(0);
28         default:
29             close(socket_fd); //TODO error handling?
30     }
31 }

```

¹La 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 mensaje². 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                    << "successfully." << endl << panic(0);
30            }

```

²Enviar 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 unsupported 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 5s 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`

```
1  while (true) {
2      // Wait (up to SERVER_TIMEOUT) for data to read
3      log << "bufLen: " << bufLen << endl;
4      if (bufLen == 0) {
5          FD_ZERO(&readFds);
6          FD_SET(client_fd, &readFds);
7          timeout.tv_sec = SERVER_TIMEOUT;
8          timeout.tv_usec = 0;
9          if (!select(client_fd+1, &readFds, NULL, NULL, &timeout)) {
10             log << "SERVER_TIMEOUT" << endl << panic(0);
11         }
12     }
```

Código 8: Fichero `src/message_processing.cpp`

```
1      if (!header_complete) {
2          FD_ZERO(&readFds);
3          FD_SET(client_fd, &readFds);
4          timeout.tv_sec = LATENCY_TIMEOUT;
5          timeout.tv_usec = 0;
6          if (!select(client_fd+1, &readFds, NULL, NULL, &timeout)) {
7              log << "LATENCY_TIMEOUT" << endl << panic(PRECOND_ERR);
8              break;
9          }
10     }
```

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 hará³. 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`

Las 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

```

³Aunque 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.


```

22 while (bufoff[0] != '\r' || bufoff[1] != '\n') {
23     Header_Field hf;
24     int hf_size = parse_header_field(hf, bufoff);
25     if (hf_size < 0) {
26         send_bad_request();
27         logerr << "Incorrect header field format" << endl << panic(PRECOND_ERR);
28     }
29     if (strcmp(hf.field, "host") == 0) {
30         contains_host = true;
31     } else if (strcmp(hf.field, "connection") == 0) {
32         if (strstr(hf.value, "close") != NULL) {
33             persistent = false;
34         }
35     } else if (strcmp(hf.field, "content-length") == 0) {
36         content_length = strtol(hf.value, NULL, 10);
37     }
38     bufoff += hf_size;
39     // log << hf.field << ' ' << hf.value << endl;
40 }
41 bufoff += 2; // Discard last "\r\n".
42 bufLen -= bufoff - buf;
43
44 if (!contains_host) {
45     send_bad_request();
46     log << "The client didn't send a host field!" << endl << panic(PRECOND_ERR);
47 }
48
49 if (content_length != 0) {
50     send_not_implemented();
51     log << "The client sent content inside GET, which isn't supported."
52         << endl << panic(PRECOND_ERR);
53 }
54
55 /**
56  * Now that we've interpreted the complete header, we must
57  * evaluate the type of file that the client is asking for
58  * and return the file if it's supported or the appropriate
59  * error.
60  */
61
62 // Make the route relative.
63 char* rel_uri = rl.request_uri;
64 while (*rel_uri == '/')
65     rel_uri++;
66 const char* pathname = *rel_uri == '\0'? "index.html" : rel_uri;
67
68 log << "GET process complete. Send: " << pathname << endl;
69 send_static_file(STATUS_OK, pathname, persistent);
70 return bufoff;
71 }

```

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`

```

1  if (strcmp(bufoff, "email=emilio.dominguezs%40um.es") == 0) {
2      log << "POST Success!" << endl;
3      send_static_file(STATUS_OK, "success.html", persistent, true);
4  } else {
5      log << "POST Fail!" << endl;
6      send_static_file(STATUS_OK, "failure.html", persistent);
7  }

```

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 2·8 KiB, siempre hay espacio 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/unauthorized.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_TIMEOUT);
64 }
65 bufLen += sprintf(buf+bufLen, "Content-Type: %s\r\n", ct);
66 bufLen += sprintf(buf+bufLen, "Content-Length: %d\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

En el archivo tests/capture-global.pcapng se muestra una captura del funcionamiento del servidor tomada con Wireshark. En las fotos que presentamos hemos aplicado el filtro

```
http || tcp.flags.syn==1 || tcp.flags.fin==1 || tcp.flags.reset==1
```

para mostrar solo los mensajes HTTP y la apertura y el cierre de la conexión. A continuación detallamos el contenido.

fig. 1 Una prueba de los mensajes intercambiados para cargar la página de inicio e introducir un correo electrónico. Puede servir para verificar la persistencia, viendo como el servidor mantiene la conexión abierta unos segundos esperando a la segunda acción.

1	0.000000000	127.0.0.1	127.0.0.1	TCP	74	35082 → 8000 [SYN, Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=797954062 TSecr=0 WS=128
2	0.000027296	127.0.0.1	127.0.0.1	TCP	74	8000 → 35082 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=797954062 TSecr=797954062 WS=128
4	0.022038435	127.0.0.1	127.0.0.1	HTTP	407	GET / HTTP/1.1
6	0.022519033	127.0.0.1	127.0.0.1	HTTP	724	HTTP/1.1 200 OK (text/html)
8	0.042872422	127.0.0.1	127.0.0.1	HTTP	361	GET /logo-um.jpg HTTP/1.1
10	0.043181453	127.0.0.1	127.0.0.1	HTTP	171	HTTP/1.1 200 OK (JPEG JFIF image)
12	3.361018689	127.0.0.1	127.0.0.1	HTTP	588	POST /accion_form.html HTTP/1.1 (application/x-www-form-urlencoded)
13	3.361553425	127.0.0.1	127.0.0.1	HTTP	421	HTTP/1.1 200 OK (text/html)
15	3.405201338	127.0.0.1	127.0.0.1	HTTP	378	GET /success.jpeg HTTP/1.1
35	3.405865697	127.0.0.1	127.0.0.1	HTTP	14184	HTTP/1.1 200 OK (JPEG JFIF image)
37	8.408723370	127.0.0.1	127.0.0.1	TCP	66	8000 → 35082 [FIN, ACK] Seq=213293 Ack=1471 Win=65536 Len=0 TSval=797962471 TSecr=797957468
38	8.408931682	127.0.0.1	127.0.0.1	TCP	66	35082 → 8000 [FIN, ACK] Seq=1471 Ack=213294 Win=65536 Len=0 TSval=797962471 TSecr=797962471

Figura 1: Intercambio de mensajes sin errores.

fig. 2 Una prueba de algunos mensajes de error. Muestra un 404 NOT FOUND y un 415 UNSUPPORTED MEDIA TYPE. Además, también muestra la robustez del servidor. Ante un POST de más de 8 KiB, que hemos simulado introduciendo 9000 caracteres en el campo del correo, devuelve un mensaje 413 REQUEST ENTITY TOO LARGE y corta la conexión⁴.

fig. 3 Una captura probando que el servidor acepta PIPELINING. Como el navegador no hacía PIPELINING, lo hemos probado utilizando el script del código 12. En un solo paquete TCP se reciben 4 mensajes GET que el servidor responde en orden. Mediante los mensajes de log se vio que los 4 mensajes se leyeron en la misma operación de lectura.

⁴Aunque 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.

40	14.099707229	127.0.0.1	127.0.0.1	TCP	74	35086 → 8000 [SYN, Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=797968162 TSecr=0 WS=128
41	14.099733929	127.0.0.1	127.0.0.1	TCP	74	8000 → 35086 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=797968162 TSecr=797968162 WS=128
43	14.115520938	127.0.0.1	127.0.0.1	HTTP	415	GET /aaa.html HTTP/1.1
45	14.116187988	127.0.0.1	127.0.0.1	HTTP	1782	HTTP/1.1 404 Not Found (text/html)
47	14.154258428	127.0.0.1	127.0.0.1	HTTP	369	GET /logo-um.jpg HTTP/1.1
49	14.154526317	127.0.0.1	127.0.0.1	HTTP	1619	HTTP/1.1 200 OK (JPEG JFIF image)
51	19.059183877	127.0.0.1	127.0.0.1	HTTP	417	GET /aaa.random HTTP/1.1
52	19.059701487	127.0.0.1	127.0.0.1	HTTP	1758	HTTP/1.1 415 Unsupported Media Type (text/html)
53	19.088804058	127.0.0.1	127.0.0.1	HTTP	371	GET /logo-um.jpg HTTP/1.1
55	19.089173116	127.0.0.1	127.0.0.1	HTTP	171	HTTP/1.1 200 OK (JPEG JFIF image)
57	23.979305308	127.0.0.1	127.0.0.1	HTTP	407	GET / HTTP/1.1
58	23.979786220	127.0.0.1	127.0.0.1	HTTP	724	HTTP/1.1 200 OK (text/html)
59	24.007189463	127.0.0.1	127.0.0.1	HTTP	361	GET /logo-um.jpg HTTP/1.1
61	24.007438675	127.0.0.1	127.0.0.1	HTTP	171	HTTP/1.1 200 OK (JPEG JFIF image)
63	26.273953182	127.0.0.1	127.0.0.1	HTTP	9565	POST /accion.form.html HTTP/1.1 (application/x-www-form-urlencoded)
64	26.274808595	127.0.0.1	127.0.0.1	HTTP	1800	HTTP/1.1 413 Request Entity Too Large (text/html)
65	26.274868898	127.0.0.1	127.0.0.1	TCP	66	8000 → 35086 [FIN, ACK] Seq=35036 Ack=11444 Win=65536 Len=0 TSval=797980338 TSecr=797980337
66	26.275420511	127.0.0.1	127.0.0.1	TCP	66	35086 → 8000 [FIN, ACK] Seq=11444 Ack=35037 Win=65536 Len=0 TSval=797980339 TSecr=797980338

Figura 2: Intercambio de mensajes de error.

Código 12: Fichero tests/pipelining.sh

```

1  #!/bin/sh
2
3
4  printf "Testing pipelining.\n";
5  (
6  message="GET / HTTP/1.1\nHost: Pepito\n\n"
7  printf "$message$message$message$message"
8  sleep 7
9  ) | telnet 127.0.0.1 8000

```

3. Despliegue de Servicios Telemáticos

3.1. Escenario Desarrollado y Versiones del Software

El escenario objetivo está compuesto por dos equipos distintos conectados a través de una red local. Para simularlo, se ha utilizado el gestor de máquinas virtuales VIRTUAL BOX 6.1.

En un equipo residen los servidores principales de los servicios. Se trata de una instancia de UBUNTU 16.04 SERVER. En el otro equipo se encuentra instalado UBUNTU 16.04 DESKTOP y cumple el papel de cliente en los servicios. Las dos máquinas virtuales disponen de una tarjeta de red virtual que simula que estuviesen conectadas a la misma red.

3.2. Servidor DNS

Uno de los servicios a desplegar era un servidor DNS raíz. El resto de los servicios se apoyan en la resolución de nombres que aporta. Por ejemplo, para que funcione el servidor mail existen dos registros: pop.sstt7628.org y smtp.sstt7628.org.

Se eligió la implementación BIND9, que es el servidor DNS más comúnmente usado en internet y el estándar de facto en sistema UNIX.

3.2.1. Configuración

La instalación se llevó a cabo con el gestor de paquetes de UBUNTU. La configuración depende de varios archivos de texto plano.

86	43.577563780	127.0.0.1	127.0.0.1	TCP	74	35098	-	8000	[SYN]	Seq=0	Win=65495	Len=0	MSS=65495	SACK_PERM=1	TSval=797997642	TSecr=0	WS=128				
87	43.577590588	127.0.0.1	127.0.0.1	TCP	74	8000	-	35098	[SYN, ACK]	Seq=0	Ack=1	Win=65483	Len=0	MSS=65495	SACK_PERM=1	TSval=797997642	TSecr=797997642	WS=128			
89	43.577807746	127.0.0.1	127.0.0.1	HTTP	194	GET / HTTP/1.1	GET / HTTP/1.1	GET / HTTP/1.1	GET / HTTP/1.1	GET / HTTP/1.1	GET / HTTP/1.1	GET / HTTP/1.1	GET / HTTP/1.1	GET / HTTP/1.1	GET / HTTP/1.1	GET / HTTP/1.1	GET / HTTP/1.1	GET / HTTP/1.1			
91	43.579093061	127.0.0.1	127.0.0.1	HTTP	1382	HTTP/1.1	200	OK	(text/html)	HTTP/1.1	200	OK	(text/html)	HTTP/1.1	200	OK	(text/html)	HTTP/1.1	200	OK	(text/html)
93	43.579146679	127.0.0.1	127.0.0.1	HTTP	724	HTTP/1.1	200	OK	(text/html)	HTTP/1.1	200	OK	(text/html)	HTTP/1.1	200	OK	(text/html)	HTTP/1.1	200	OK	(text/html)
95	43.579236457	127.0.0.1	127.0.0.1	HTTP	724	HTTP/1.1	200	OK	(text/html)	HTTP/1.1	200	OK	(text/html)	HTTP/1.1	200	OK	(text/html)	HTTP/1.1	200	OK	(text/html)
1..	48.583254444	127.0.0.1	127.0.0.1	TCP	66	8000	-	35098	[FIN, ACK]	Seq=2633	Ack=129	Win=65536	Len=0	TSval=798002647	TSecr=797997643	WS=0					
1..	48.583381436	127.0.0.1	127.0.0.1	TCP	66	35098	-	8000	[FIN, ACK]	Seq=129	Ack=2634	Win=65536	Len=0	TSval=798002648	TSecr=798002647	WS=0					

Figura 3: Prueba de PIPELINING.

En el fichero `/etc/bind/named.conf.options` se definen las opciones globales. Se ha configurado que se acepten peticiones de hosts en la red local y que se redirijan las peticiones que no se sepan resolver al servidor DNS de la Universidad de Murcia.

Fichero `/etc/bind/named.conf.options`

```

1 // Para leer todo acerca a la configuración:
2 // https://www.zytrax.com/books/dns/ch7/
3 options {
4     directory "/var/cache/bind";
5
6     allow-query {           // Hosts que tienen permitido realizar consultas
7         192.168.56.0/24; // Red local de la práctica
8     };
9
10    forwarders {            // Direcciones IP de servidores dns para consultas desconocidas.
11        155.54.1.1; // Dirección IP de los servidores DNS de la universidad.
12        155.54.1.2; // Obtenida con el comando dig -t ns um.es
13    };
14
15    recursion yes; // Habilita la recursión para resolver consultas anidadas.
16                  // Por ejemplo: www.sstt7628.org (org, sstt7628.org, www.sstt7628.org)
17
18    dnssec-validation auto;
19
20    auth-nxdomain no;      # conform to RFC1035
21    listen-on-v6 { any; };
22 };

```

El fichero `/etc/bind/named.conf.local` sirve para configurar las zonas que conoce el servidor. Basta con poner que se trata de una zona manejada por nuestro servidor (TYPE MASTER) y definir un fichero de zona que debemos localizar en la carpeta `/etc/bind`. Es habitual, si hay pocos ficheros de zona, que el nombre de los ficheros de zona debe comenzar por DB y no se incluyan dentro de ninguna subcarpeta, pero no es obligatorio.

Fichero `/etc/bind/named.conf.local`

```

1 zone "sstt7628.org." IN {
2     type master; // El servidor lee la información de un fichero de zona local
3                 // y responde de forma autoritativa.
4     file "/etc/bind/db.sstt7628.org.zone"; // El fichero de zona
5 }

```

En el fichero de zona (`/ETC/BIND/DB.SSTT7628.ORG.ZONE`) se configuraron los valores asociados a nuestro dominio, como el TTL (*time to live*). En mi caso he definido varios registros, entre ellos dos registros de correo electrónico, que se utilizan en el resto de apartados.

Fichero `/etc/bind/db.sstt7628.org.zone`

```

1 ; Toda la información de configuración de los ficheros de zona en
2 ; https://www.zytrax.com/books/dns/ch8/
3 $ORIGIN sstt7628.org. ; Especificamos que a los nombres que acaben sin un punto
4 ; se les añada esta ruta

```

```

5 $TTL      3600
6 @         IN      SOA      sstt7628.org. root.sstt7628.org. (
7                               1          ; Serial
8                               3600       ; Refresh
9                               1800       ; Retry
10                              604800      ; Expire
11                              3600)      ; Negative Cache TTL
12 ;
13 @         IN      NS       localhost.
14 cliente   IN      A        192.168.56.101
15 servidor  IN      A        192.168.56.104
16 www       IN      CNAME     servidor
17 web       IN      CNAME     servidor
18 mail      IN      A        192.168.56.104
19 @         IN      MX 10     mail
20 smtp      IN      CNAME     mail
21 pop       IN      CNAME     mail

```

Por otro lado, además de registrar el servidor DNS hay que configurar los hosts para que hagan peticiones a este servidor.

La configuración de dominios DNS se encuentra en el fichero `/etc/resolv.conf`. No obstante, los cambios que hagamos sobre este fichero se perderán al reiniciar el ordenador, porque es un fichero generado automáticamente. En su lugar, podemos editar el fichero `/etc/resolvconf/resolv.conf.d/head`, que se copia al principio durante la generación. Para añadir un servidor DNS basta con añadir la línea

```
nameserver 192.168.56.104.
```

Podemos evitar reiniciar el sistema para ver los cambios ejecutando la orden `resolvconf -u`.

3.2.2. Capturas Wireshark

A continuación (fig. 4) se muestra un trozo de la captura Wireshark adjunta donde se ve una consulta DNS.

1	0.000000000	192.168.56.101	192.168.56.104	DNS	132 Standard query 0xe2cf A www.um.es OPT
2	0.829131548	192.168.56.104	192.168.56.101	DNS	625 Standard query response 0xe2cf A www.um.es CNAME ...
3	0.829131548	192.168.56.104	192.168.56.101	DNS	573 Standard query response 0xe2cf A www.um.es CNAME ...
4	5.012687038	PcsCompu_96:c...	PcsCompu_25:2b...	ARP	42 Who has 192.168.56.104? Tell 192.168.56.101
5	5.013185140	PcsCompu_25:2...	PcsCompu_96:c5...	ARP	60 192.168.56.104 is at 08:00:27:25:2b:71
6	10.705260269	192.168.56.101	192.168.56.104	DNS	139 Standard query 0x387b A www.sstt7628.org OPT
7	10.706013910	192.168.56.104	192.168.56.101	DNS	245 Standard query response 0x387b A www.sstt7628.org...
8	10.706013910	192.168.56.104	192.168.56.101	DNS	193 Standard query response 0x387b A www.sstt7628.org...

Figura 4: Dos consultas DNS

3.3. Correo SMTP/POP

Se ha desplegado también un servicio de correo electrónico SMTP/POP mediante las implementaciones EXIM y DOVECOT. El procedimiento ha sido el explicado en las sesiones de prácticas.

3.3.1. Usuarios de correo

Se crearon dos usuarios de correo con nombres nombre1_49277628@sstt7628.org y nombre2_49277628@sstt7628.org como pedía la práctica. Con la configuración vista en clase, basta con registrar ambos usuarios en el sistema y el directorio de correo (MAILBOX) se crea automáticamente.

Para verificar el funcionamiento se se utilizó el gestor de correo THUNDERBIRD (en el cliente) y se enviaron varios correos de una a otra.

3.3.2. Capturas Wireshark

A continuación (fig. 5) se muestra un trozo de la captura Wireshark adjunta donde se muestran los mensajes intercambiados para el envío de un correo electrónico.

11	29.709742663	192.168.56.101	192.168.56.104	DNS	132 Standard query 0x9faf A live.thunderbird.net
12	29.709764706	192.168.56.101	192.168.56.104	DNS	132 Standard query 0x163e AAAA live.thunderbird.net
13	29.780039888	192.168.56.101	192.168.56.104	DNS	128 Standard query 0x039c A pop.sstt7628.org
14	29.780060758	192.168.56.101	192.168.56.104	DNS	128 Standard query 0xc412 AAAA pop.sstt7628.org
15	29.780611495	192.168.56.104	192.168.56.101	DNS	230 Standard query response 0x039c A pop.sstt7628.org...
17	29.780611495	192.168.56.104	192.168.56.101	DNS	178 Standard query response 0x039c A pop.sstt7628.org...
16	29.780628013	192.168.56.104	192.168.56.101	DNS	188 Standard query response 0xc412 AAAA pop.sstt7628...
18	29.780628013	192.168.56.104	192.168.56.101	DNS	136 Standard query response 0xc412 AAAA pop.sstt7628...
19	29.784723661	192.168.56.101	192.168.56.104	TCP	126 60186 → 110 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 ...
20	29.784927794	192.168.56.101	192.168.56.104	TCP	126 60188 → 110 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 ...
21	29.785072960	192.168.56.104	192.168.56.101	TCP	126 110 → 60186 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=...
22	29.785072960	192.168.56.104	192.168.56.101	TCP	74 [TCP Out-Of-Order] 110 → 60186 [SYN, ACK] Seq=0 A...
23	29.785108717	192.168.56.101	192.168.56.104	TCP	118 60186 → 110 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSv...
24	29.785539290	192.168.56.104	192.168.56.101	TCP	126 110 → 60188 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=...
25	29.785539290	192.168.56.104	192.168.56.101	TCP	74 [TCP Out-Of-Order] 110 → 60188 [SYN, ACK] Seq=0 A...
26	29.785572920	192.168.56.101	192.168.56.104	TCP	118 60188 → 110 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSv...
27	29.808015719	192.168.56.104	192.168.56.101	POP	138 S: +OK Dovecot ready.
28	29.808015719	192.168.56.104	192.168.56.101	TCP	86 [TCP Retransmission] 110 → 60186 [PSH, ACK] Seq=1...
29	29.808090884	192.168.56.101	192.168.56.104	TCP	118 60186 → 110 [ACK] Seq=1 Ack=21 Win=29312 Len=0 TS...
30	29.808116359	192.168.56.104	192.168.56.101	POP	138 S: +OK Dovecot ready.
31	29.808116359	192.168.56.104	192.168.56.101	TCP	86 [TCP Retransmission] 110 → 60188 [PSH, ACK] Seq=1...
32	29.808155303	192.168.56.101	192.168.56.104	TCP	118 60188 → 110 [ACK] Seq=1 Ack=21 Win=29312 Len=0 TS...
33	29.831265839	192.168.56.101	192.168.56.104	POP	124 C: AUTH
34	29.833280229	192.168.56.104	192.168.56.101	TCP	118 110 → 60186 [ACK] Seq=21 Ack=7 Win=29056 Len=0 TS...
36	29.833280229	192.168.56.104	192.168.56.101	TCP	66 110 → 60186 [ACK] Seq=21 Ack=7 Win=29056 Len=0 TS...
35	29.833301482	192.168.56.104	192.168.56.101	POP	133 S: +OK
37	29.833301482	192.168.56.104	192.168.56.101	TCP	81 [TCP Retransmission] 110 → 60186 [PSH, ACK] Seq=2...
38	29.836158297	192.168.56.101	192.168.56.104	POP	124 C: AUTH
39	29.837936006	192.168.56.104	192.168.56.101	TCP	118 110 → 60188 [ACK] Seq=21 Ack=7 Win=29056 Len=0 TS...
40	29.837936006	192.168.56.104	192.168.56.101	TCP	66 [TCP Dup ACK 39#1] 110 → 60188 [ACK] Seq=21 Ack=7...
41	29.838554086	192.168.56.104	192.168.56.101	POP	133 S: +OK
42	29.838554086	192.168.56.104	192.168.56.101	TCP	81 [TCP Retransmission] 110 → 60188 [PSH, ACK] Seq=2...
43	29.860682019	192.168.56.101	192.168.56.104	POP	124 C: CAPA
44	29.861138710	192.168.56.104	192.168.56.101	POP	201 S: +OK
45	29.861138710	192.168.56.104	192.168.56.101	TCP	149 [TCP Retransmission] 110 → 60186 [PSH, ACK] Seq=3...
46	29.870300297	192.168.56.101	192.168.56.104	POP	124 C: CAPA
47	29.870722008	192.168.56.104	192.168.56.101	POP	201 S: +OK
48	29.870722008	192.168.56.104	192.168.56.101	TCP	149 [TCP Retransmission] 110 → 60188 [PSH, ACK] Seq=3...
49	29.900931660	192.168.56.101	192.168.56.104	POP	130 C: AUTH PLAIN
50	29.904286306	192.168.56.104	192.168.56.101	POP	122 +
51	29.904286306	192.168.56.104	192.168.56.101	TCP	70 [TCP Retransmission] 110 → 60186 [PSH, ACK] Seq=1...
52	29.917047031	192.168.56.101	192.168.56.104	POP	148 C: AG5vbWJyZTFfNzYyOABhbHVtbn8=
53	29.917231201	192.168.56.101	192.168.56.104	TCP	118 60188 → 110 [ACK] Seq=13 Ack=119 Win=29312 Len=0 ...
54	29.920084530	192.168.56.101	192.168.56.104	POP	130 C: AUTH PLAIN
55	29.920595026	192.168.56.104	192.168.56.101	POP	122 +
56	29.920595026	192.168.56.104	192.168.56.101	TCP	70 [TCP Retransmission] 110 → 60188 [PSH, ACK] Seq=1...
57	29.920639649	192.168.56.101	192.168.56.104	TCP	118 60188 → 110 [ACK] Seq=25 Ack=123 Win=29312 Len=0 ...
58	29.925043350	192.168.56.101	192.168.56.104	POP	148 C: AG5vbWJyZTFfNzYyOABhbHVtbn8=
59	29.947122738	192.168.56.104	192.168.56.101	POP	134 S: +OK Logged in.
60	29.947122738	192.168.56.104	192.168.56.101	TCP	82 [TCP Retransmission] 110 → 60186 [PSH, ACK] Seq=1...
61	29.958389133	192.168.56.104	192.168.56.101	POP	134 S: +OK Logged in.
62	29.958389133	192.168.56.104	192.168.56.101	TCP	82 [TCP Retransmission] 110 → 60188 [PSH, ACK] Seq=1...
63	29.981584886	192.168.56.101	192.168.56.104	POP	124 C: STAT
64	29.981771799	192.168.56.101	192.168.56.104	POP	124 C: STAT
65	29.982040939	192.168.56.104	192.168.56.101	POP	130 S: +OK 4 5473
67	29.982040939	192.168.56.104	192.168.56.101	TCP	78 [TCP Retransmission] 110 → 60186 [PSH, ACK] Seq=1...
66	29.982056543	192.168.56.104	192.168.56.101	POP	131 S: +OK 7 10292

Figura 5: Tráfico generado por Thunderbird

3.4. Servidor HTTP y HTTPs basado en Apache

El grueso del trabajo era implementar el servidor HTTP descrito en la primera sección. No obstante, también hemos tenido que configurar un servidor APACHE que recibiese peticiones HTTP (en el puerto 80) y HTTPs (en el puerto 443).

3.4.1. Configuración

La configuración se lleva a cabo, una vez más, a través de ficheros de configuración del sistema. Para usar APACHE hay que configurar VIRTUAL HOSTS a través de los siguientes pasos.

1. Se crea el fichero del sitio `/etc/apache2/sites-available/sitio.conf` (Apache incluye ficheros de ejemplo que se pueden utilizar como base). Para la práctica se ha configurado ambos VIRTUAL HOSTS en el mismo fichero porque corresponden al mismo sitio.
2. Se activa el fichero con el comando `a2ensite`, que entre otras cosas crea un enlace dinámico a nuestro fichero en la carpeta `/etc/apache2/sites-available/`.
3. Y se recarga el servicio de apache (`service apache2 reload`), tras lo cual ya podemos acceder.

Si no disponemos de un servicio de DNS que resuelva los nombres de nuestros sitios, podemos añadir una entrada al fichero `/etc/hosts`, que sirve para registrar traducciones estáticas en LINUX. En mi caso, instalé APACHE antes de configurar BIND y ese fue el procedimiento inicial.

El servidor HTTP se protegió mediante usuario y contraseña siguiendo los siguientes pasos.

- Configuramos el virtual host HTTP de APACHE para restringir la entrada a un grupo de usuarios registrados en ficheros de configuración de APACHE. Es decir, para utilizar un login y contraseña.

Primera parte del fichero `/etc/apache2/sites-available/sstt7628.conf`

```
1 <VirtualHost *:80>
2   ServerName www.sstt7628.org
3
4   ServerAdmin admin@sstt7628.org
5   DocumentRoot /var/www/sstt7628
6
7   <Directory /var/www/sstt7628>
8     AllowOverride AuthConfig
9     AuthType Basic
10    AuthName "Restricted access to group sstt7628"
11    AuthBasicProvider file
12    AuthUserFile /etc/apache2/passwords
13    AuthGroupFile /etc/apache2/groups
14    Require Group sstt7628
15    Order allow,deny
16    allow from all
17  </Directory>
18
19  ErrorLog ${APACHE_LOG_DIR}/error.log
20  CustomLog ${APACHE_LOG_DIR}/access.log combined
21 </VirtualHost>
```

- Los usuarios se crean con el comando `htpasswd -c /etc/apache2/passwords sstt7628` y se registran en el grupo añadiendo un fichero `groups` en la carpeta donde se encuentre el fichero de usuarios con el siguiente formato

Fichero /etc/apache2/groups

```
1 sstt7628: sstt7628
```

- La configuración de usuarios requiere del módulo `authz_groupfile` de `APACHE`. No hace falta instalarlo en `UBUNTU`. Basta con ejecutar el comando `a2enmod authz_groupfile` para habilitarlo.

3.5. Certificación HTTPs a través de OpenSSL

La configuración del servidor HTTPs es más complicada y requiere que una entidad certificadora firme (en el sentido utilizado en informática para claves públicas y privadas) los certificados que utiliza el protocolo.

Aunque no se ha desplegado una entidad certificadora en el servidor (lo que tendría sentido sería que estuviese en un tercer ordenador), se ha simulado el proceso ejecutando los comandos de la herramienta `OPENSSL` tal cual vimos en las clases de prácticas y se encuentra documentado en las diapositivas.

Una vez generadas las claves para ambos dispositivos y configurado el navegador web del cliente, `FIREFOX` para que reconociese a la entidad certificadora, se configuró el servidor HTTPs en el fichero `/etc/apache2/sites-available/sstt7628.conf`.

Segunda parte del fichero `/etc/apache2/sites-available/sstt7628.conf`

```
1 <VirtualHost *:443>
2   ServerName www.sstt7628.org
3
4   ServerAdmin admin@sstt7628.org
5   DocumentRoot /var/www/sstt7628
6
7   # Authentication Configuration
8   SSLEngine on
9   # Los ficheros de claves han sido generados usando OpenSSL.
10  SSLCertificateFile /home/alumno/CAentity/servercert.pem
11  SSLCertificateKeyFile /home/alumno/CAentity/serverkey.pem
12  SSLCACertificateFile /home/alumno/CAentity/cacert.pem
13
14  SSLVerifyClient require
15  SSLVerifyDepth 10
16 </VirtualHost>
```

3.5.1. Capturas Wireshark

A continuación (figs. 6 and 7) se muestra un trozo de la captura Wireshark adjunta donde se muestra primero un acceso al servidor web vía `HTTP` y después vía `HTTPs`.

3.6. IPsec con Strongswan

Por último, se pedía implementar el protocolo `IPsec` para proteger los paquetes `IP` entre ambos dispositivos. Los requisitos eran utilizar `IKEv2` para el establecimiento del canal y operar en modo

276	72.571111216	192.168.56.101	192.168.56.104	DNS	128	Standard query 0xcd9a A www.sstt7628.org
277	72.571147460	192.168.56.101	192.168.56.104	DNS	128	Standard query 0x0a15 AAAA www.sstt7628.org
278	72.571635914	192.168.56.104	192.168.56.101	DNS	234	Standard query response 0xcd9a A www.sstt7628.org...
280	72.571635914	192.168.56.104	192.168.56.101	DNS	182	Standard query response 0xcd9a A www.sstt7628.org...
279	72.571651825	192.168.56.104	192.168.56.101	DNS	192	Standard query response 0x0a15 AAAA www.sstt7628...
281	72.571651825	192.168.56.104	192.168.56.101	DNS	140	Standard query response 0x0a15 AAAA www.sstt7628...
282	72.572001106	192.168.56.101	192.168.56.104	DNS	122	Standard query 0x016b A www.st1.um
283	72.572020072	192.168.56.101	192.168.56.104	DNS	122	Standard query 0xe6c4 AAAA www.st1.um
284	74.754104372	192.168.56.101	192.168.56.104	DNS	149	Standard query 0x9d7f A firefox.settings.services...
285	74.754146027	192.168.56.101	192.168.56.104	DNS	149	Standard query 0x4cba AAAA firefox.settings.servi...
286	74.813807618	192.168.56.101	192.168.56.104	TCP	126	41840 → 80 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 S...
287	74.814100910	192.168.56.104	192.168.56.101	TCP	126	80 → 41840 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0...
288	74.814100910	192.168.56.104	192.168.56.101	TCP	74	[TCP Out-Of-Order] 80 → 41840 [SYN, ACK] Seq=0 Ac...
289	74.814140706	192.168.56.101	192.168.56.104	TCP	118	41840 → 80 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSva...
290	74.814313401	192.168.56.101	192.168.56.104	HTTP	565	GET / HTTP/1.1
291	74.814886709	192.168.56.104	192.168.56.101	TCP	118	80 → 41840 [ACK] Seq=1 Ack=448 Win=30080 Len=0 TS...
293	74.814886709	192.168.56.104	192.168.56.101	TCP	66	80 → 41840 [ACK] Seq=1 Ack=448 Win=30080 Len=0 TS...
292	74.814898750	192.168.56.104	192.168.56.101	HTTP	872	HTTP/1.1 401 Unauthorized (text/html)
294	74.814898750	192.168.56.104	192.168.56.101	TCP	820	[TCP Retransmission] 80 → 41840 [PSH, ACK] Seq=1 ...
295	74.814930769	192.168.56.101	192.168.56.104	TCP	118	41840 → 80 [ACK] Seq=448 Ack=755 Win=30720 Len=0 ...

Figura 6: Tráfico generado por FIREFOX con la petición HTTP

túnel con autenticación a través de la cabecera AH, pero sin encriptación. Para ello, se podían utilizar los certificados de identidad que utilizamos para la conexión HTTPs.

Se ha hecho uso de STRONGSWAN, una implementación *Open Source* de IPsec. Como en el resto de casos, la instalación se ha llevado a cabo a través del gestor de paquetes de UBUNTU, apt.

3.6.1. Configuración

La configuración de los canales se puede escribir en el fichero `/etc/ipsec.conf` de cada host.

Fichero `/etc/ipsec.conf` del servidor

```

1 config setup
2
3 conn %default # Valores por defecto para las conexiones
4     ikelifetime=60m
5     keylife=20m
6     rekeymargin=3m
7     keyingtries=1
8     mobike=no
9     keyexchange=ikev2
10    authby=pubkey
11
12 conn host-host # Conexión host-host de la práctica
13     left=192.168.56.104 # IP del servidor
14     leftcert=/etc/ipsec.d/certs/servercert.pem # Clave pública generada con OpenSSL
15     leftid="C=ES, ST=Murcia, O=UMU, OU=sstt7628, CN=www.sstt7628.org"
16     right=192.168.56.101 # IP del cliente
17     rightid="C=ES, ST=Murcia, O=UMU, OU=sstt7628, CN=emilio49277628"
18     type=tunnel
19     ah=sha256
20     auto=start

```

Fichero `/etc/ipsec.conf` del cliente

```

1 config setup
2
3 conn %default # Valores por defecto para las conexiones
4     ikelifetime=60m
5     keylife=20m
6     rekeymargin=3m
7     keyingtries=1

```

329	78.684760757	192.168.56.104	192.168.56.101	HTTP	778 HTTP/1.1 200 OK (text/html)
330	78.684760757	192.168.56.104	192.168.56.101	TCP	726 [TCP Retransmission] 80 → 41840 [PSH, ACK] Seq=75...
331	78.684827157	192.168.56.101	192.168.56.104	TCP	118 41840 → 80 [ACK] Seq=942 Ack=1415 Win=32256 Len=0...
332	78.864727341	192.168.56.101	192.168.56.104	DNS	142 Standard query 0x35da A incoming.telemetry.mozill...
333	78.881210922	192.168.56.101	192.168.56.104	DNS	142 Standard query 0xb093 AAAA incoming.telemetry.moz...
334	79.188013036	192.168.56.101	192.168.56.104	HTTP	564 GET /logo-um.jpg HTTP/1.1
335	79.188899808	192.168.56.104	192.168.56.101	HTTP	299 HTTP/1.1 304 Not Modified
336	79.188899808	192.168.56.104	192.168.56.101	TCP	247 [TCP Retransmission] 80 → 41840 [PSH, ACK] Seq=14...
337	79.188964421	192.168.56.101	192.168.56.104	TCP	118 41840 → 80 [ACK] Seq=1388 Ack=1596 Win=33792 Len=...
338	80.393048537	192.168.56.101	192.168.56.104	DNS	149 Standard query 0x6160 A firefox.settings.services...
339	80.393075872	192.168.56.101	192.168.56.104	DNS	149 Standard query 0xd49b AAAA firefox.settings.servi...
340	81.251567931	192.168.56.101	192.168.56.104	DNS	126 Standard query 0x9d7c A www.google.com
341	81.251586916	192.168.56.101	192.168.56.104	DNS	126 Standard query 0x1c82 AAAA www.google.com
342	81.419783868	192.168.56.101	192.168.56.104	DNS	137 Standard query 0xe0e8 A push.services.mozilla.com
343	81.419812032	192.168.56.101	192.168.56.104	DNS	137 Standard query 0x4109 AAAA push.services.mozilla...
344	81.548932928	192.168.56.101	192.168.56.104	DNS	139 Standard query 0x5198 A safebrowsing.googleapis.c...
345	81.548975597	192.168.56.101	192.168.56.104	DNS	139 Standard query 0x3eff AAAA safebrowsing.googleapi...
346	82.572985463	192.168.56.104	192.168.56.101	DNS	122 Standard query response 0x016b Server failure A w...
347	82.572985463	192.168.56.104	192.168.56.101	DNS	70 Standard query response 0x016b Server failure A w...
348	82.573093859	192.168.56.104	192.168.56.101	DNS	122 Standard query response 0xe6c4 Server failure AAA...
349	82.573093859	192.168.56.104	192.168.56.101	DNS	70 Standard query response 0xe6c4 Server failure AAA...

▶ Frame 328: 612 bytes on wire (4896 bits), 612 bytes captured (4896 bits) on interface 0
 ▶ Ethernet II, Src: PcsCompu_96:c5:78 (08:00:27:96:c5:78), Dst: PcsCompu_25:2b:71 (08:00:27:25:2b:71)
 ▶ Internet Protocol Version 4, Src: 192.168.56.101, Dst: 192.168.56.104
 ▶ Authentication Header
 ▶ Internet Protocol Version 4, Src: 192.168.56.101, Dst: 192.168.56.104
 ▶ Transmission Control Protocol, Src Port: 41840, Dst Port: 80, Seq: 448, Ack: 755, Len: 494
 ▶ Hypertext Transfer Protocol
 ▶ GET / HTTP/1.1\r\n
 Host: www.sstt7628.org\r\n
 User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:76.0) Gecko/20100101 Firefox/76.0\r\n
 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8\r\n
 Accept-Language: es-ES,es;q=0.8,en-US;q=0.5,en;q=0.3\r\n
 Accept-Encoding: gzip, deflate\r\n
 Connection: keep-alive\r\n
 Upgrade-Insecure-Requests: 1\r\n
 If-Modified-Since: Thu, 14 May 2020 14:26:56 GMT\r\n
 If-None-Match: "1fa-5a59c7d2766d5-gzip"\r\n
 ▶ Authorization: Basic c3N0dDc2Mjg6cGFzc3dvcmQ=

\r\n
 [Full request URI: http://www.sstt7628.org/]
 [HTTP request 2/3]
 [Prev request in frame: 290]
 [Response in frame: 329]
 [Next request in frame: 334]

Figura 7: Tráfico generado por FIREFOX con la petición HTTPs

```

8  mobike=no
9  keyexchange=ikev2
10 authby=pubkey
11
12 conn host-host # Conexión host-host de la práctica
13 left=192.168.56.101 # IP del cliente
14 leftcert=/etc/ipsec.d/certs/servercert.pem # Clave pública generada con OpenSSL
15 leftid="C=ES, ST=Murcia, O=UMU, OU=sstt7628, CN=emilio49277628"
16 right=192.168.56.104 # IP del servidor
17 rightid="C=ES, ST=Murcia, O=UMU, OU=sstt7628, CN=www.sstt7628.org"
18 type=tunnel
19 ah=sha256
20 auto=start

```

En la configuración se puede ver que se ha seleccionado el modo túnel y que se utiliza el protocolo sin encriptación. Concretamente, la línea `ah=sha256` especifica que no se utilice encriptación ESP, el modo predeterminado en STRONGSWAN. SHA256 es una función de hash que se utiliza para asegurar la integridad de los paquetes en la cabecera AH. Hay muchas opciones disponibles, y he elegido esta función por ser bastante común.

Por último, hay que modificar otro fichero, `/etc/ipsec.secrets`, que contiene la información delicada. Lo que hay que incluir en él es la ruta al fichero que contiene la clave privada.

Fichero /etc/ipsec.secrets del cliente

```
1 : RSA /etc/ipsec.d/private/clientkey.pem
```

3.6.2. Capturas Wireshark

A continuación (fig. 8) se muestra un trozo de la captura Wireshark adjuntada donde se ve que los mensajes originados por un ping de una máquina a otra incluyen la cabecera AH de IPsec.

596	104.474852805	192.168.56.101	192.168.56.104	ICMP	150	Echo (ping) request	id=0x30ca, seq=2/512, ttl=64...
597	104.476431950	192.168.56.104	192.168.56.101	ICMP	150	Echo (ping) reply	id=0x30ca, seq=2/512, ttl=64...
598	104.476431950	192.168.56.104	192.168.56.101	ICMP	98	Echo (ping) reply	id=0x30ca, seq=2/512, ttl=64...
599	105.476096989	192.168.56.101	192.168.56.104	ICMP	150	Echo (ping) request	id=0x30ca, seq=3/768, ttl=64...
600	105.476762829	192.168.56.104	192.168.56.101	ICMP	150	Echo (ping) reply	id=0x30ca, seq=3/768, ttl=64...
601	105.476762829	192.168.56.104	192.168.56.101	ICMP	98	Echo (ping) reply	id=0x30ca, seq=3/768, ttl=64...
602	106.477870368	192.168.56.101	192.168.56.104	ICMP	150	Echo (ping) request	id=0x30ca, seq=4/1024, ttl=6...
603	106.478317605	192.168.56.104	192.168.56.101	ICMP	150	Echo (ping) reply	id=0x30ca, seq=4/1024, ttl=6...
604	106.478317605	192.168.56.104	192.168.56.101	ICMP	98	Echo (ping) reply	id=0x30ca, seq=4/1024, ttl=64...

▶	Frame 599: 150 bytes on wire (1200 bits), 150 bytes captured (1200 bits) on interface 0
▶	Ethernet II, Src: PcsCompu_96:c5:78 (08:00:27:96:c5:78), Dst: PcsCompu_25:2b:71 (08:00:27:25:2b:71)
▶	Internet Protocol Version 4, Src: 192.168.56.101, Dst: 192.168.56.104
▼	Authentication Header
	Next header: IPIP (4)
	Length: 6 (32 bytes)
	Reserved: 0000
	AH SPI: 0xc2f2ab53
	AH Sequence: 241
	AH ICV: 83d4f000095b65c14c6949b99bfdaa9d0097b900
▶	Internet Protocol Version 4, Src: 192.168.56.101, Dst: 192.168.56.104
▼	Internet Control Message Protocol
	Type: 8 (Echo (ping) request)
	Code: 0
	Checksum: 0x5d84 [correct]
	[Checksum Status: Good]
	Identifier (BE): 12490 (0x30ca)
	Identifier (LE): 51760 (0xca30)
	Sequence number (BE): 3 (0x0003)
	Sequence number (LE): 768 (0x0300)
	[Response frame: 600]
	Timestamp from icmp data: May 16, 2020 17:59:17.000000000 CEST
	[Timestamp from icmp data (relative): 0.094004183 seconds]
▶	Data (48 bytes)

Figura 8: Tráfico generado por el comando ping. Se ve la cabecera AH.

4. 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 del servidor HTTP	7 h 39 min
Revisión antes de la primera entrega	35 min
Configuración de los servicios telemáticos	7 h 30 min
Documentación de los servicios telemáticos	2 h
Revisión final	40 min
TOTAL	25 h 38 min