Capítulo 5

Impacto de la migración en las capas superiores

5.1. Introducción

Ya que la arquitectura TCP/IP no está perfectamente dividida en capas, el cambio del protocolo IPv4 al protocolo IPv6 no sólo va a afectar a la capa de red, sino que también afectará a la capa de transporte y a la capa de aplicación.

Aunque en la capa de transporte se sigan utilizando los mismos protocolos, es necesario actualizar aquellos mecanismos que utilicen valores de la capa de red cuyo formato se haya modificado. De igual forma, es necesario realizar ciertos cambios en las aplicaciones para adecuarlas tanto a las nuevas direcciones como a los nuevos mecanismos que se definen en IPv6.

5.2. Impacto en la capa de transporte

En el diseño de IPv6 se consideró que, aunque se variara el protocolo de nivel de red, se deberían seguir utilizando los protocolos de transporte ya definidos. Algunos de ellos son:

- UDP (*User Datagram Protocol*), definido en la RFC 768 [1].
- TCP (*Transmission Control Protocol*), definido en la RFC 793 [2].

El único cambio que afecta a todos los protocolos de la capa de transporte es el algoritmo de cálculo de la suma de comprobación (*checksum*), ya que las direcciones IPv6 son diferentes a las direcciones IPv4 y los campos de la cabecera IPv4 son diferentes a los campos de la cabecera IPv6. A continuación se va a analizar cómo se realiza la suma de comprobación en cada uno de los protocolos para mostrar las modificaciones que se tienen que realizar como consecuencia del cambio de protocolo de nivel de red.

5.2.1. Suma de comprobación en IPv4

La cabecera de un paquete IPv4 contiene un campo, denominado *Header Checksum* (16 bits de extensión), que realiza la suma de comprobación de toda la cabecera del paquete. En función del protocolo de transporte que se utilice, tendremos:

- En el caso de que se utilice el protocolo TCP, se añade una suma de comprobación que incluye tanto los datos que se van a enviar como carga de un paquete IPv4 así como una "pseudo-cabecera" derivada de la cabecera completa del paquete IPv4, tal y como se muestra en la Figura 5.1.

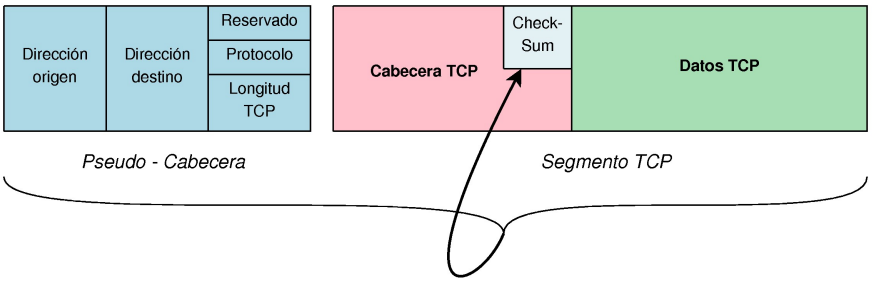

Figura 5.1: Suma de comprobación de TCP

- En el caso de que se utilice el protocolo UDP, la realización de otra suma de comprobación es opcional.

Por lo tanto, como la suma de comprobación de nivel de transporte incluye valores que corresponden al nivel de red, es necesario realizar algunas modificaciones en los protocolos de nivel de transporte.

5.2.2. Suma de comprobación en IPv6

En IPv6, la cabecera de un paquete no contiene ningún campo que almacene su suma de comprobación. Así, a su paso por cada uno de los encaminadores no es necesario actualizar el valor de la suma de comprobación y por tanto se disminuye el tiempo de procesamiento de cada paquete.

Solamente se realiza la suma de comprobación en la capa de transporte. Esta suma incluirá los datos de la capa de transporte y una "pseudo-cabecera" derivada de la cabecera del paquete IPv6. Esta "pseudo-cabecera" (Figura 5.2) está compuesta por los siguientes campos:

- Dirección IPv6 origen.
- Dirección IPv6 destino:
  - En el caso de que se utilice la cabecera de encaminamiento, cuando la dirección de destino no es la dirección de destino final, la suma de comprobación se realizará con la dirección de destino final.
- Longitud de los datos de la carga correspondiente a la capa de transporte.
- *Next-Header* (Siguiente cabecera):
  - En el caso de que el paquete IPv6 no tenga cabeceras de extensión, este campo almacenará el identificador del protocolo de nivel de transporte.
  - En el caso de que el paquete IPv6 tenga cabeceras de extensión, este valor se toma de la última cabecera de extensión del paquete.

Como en la cabecera del paquete IPv6 no se almacena ninguna suma de comprobación, es obligatorio que los protocolos de nivel de transporte implementen una suma de comprobación que incluya algunos campos de la cabecera IPv6, incluyendo UDP. Esto provoca, de nuevo, una ruptura del modelo de capas ya que una capa superior utiliza los datos de la capa inferior para realizar una suma de comprobación.

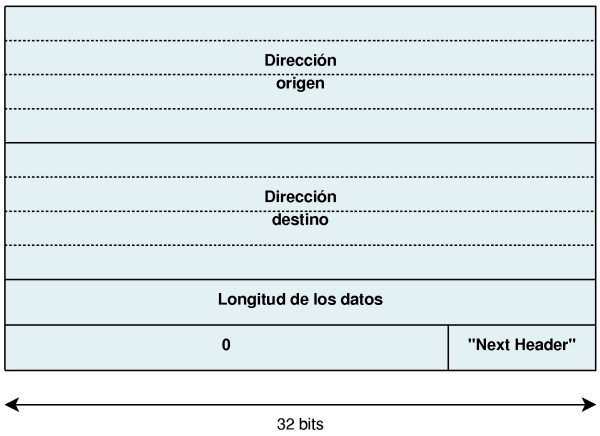

Figura 5.2: "Pseudo-cabecera" para la suma de comprobación en IPv6

5.3. Impacto en la capa de aplicación

Las aplicaciones que realizan transmisión de datos a través de una red IP, identifican a cada uno de los nodos mediante una dirección IP, obtenida, por ejemplo, mediante una consulta a un servidor DNS. Dado que el formato de una dirección IPv4 es diferente al formato de una dirección IPv6 y que la capa de transporte, por tanto, ha variado su interfaz, es necesario realizar algunas modificaciones en el código de la aplicación que está relacionado con la comunicación por red. Para realizar estas modificaciones es necesario poseer el código fuente de la aplicación, por lo que si no se tiene acceso a él hay que recurrir a los mecanismos de traducción vistos anteriormente.

5.4. Sockets en IPv6

Como ejemplo de los cambios que hay que realizar en las aplicaciones, a continuación se van a analizar los cambios realizados en la interfaz de la biblioteca de sockets en el lenguaje de programación C.

5.4.1. Introducción

Un programa, escrito en el lenguaje de programación C, utiliza la API correspondiente a los sockets para acceder a la red con el fin de la transmisión de datos. Dado el cambio de versión del protocolo, es necesario actualizar dicha interfaz para adecuarla a las direcciones IPv6 (de 128 bits de longitud por los 32 bits de longitud de las direcciones IPv4) y a los nuevos mecanismos (clases de tráfico, etiquetas de flujo...).

La interfaz correspondiente al protocolo IPv4 se definió inicialmente en Unix y, con el paso de los años, se convirtió en estándar de facto. Sin embargo, la interfaz correspondiente al protocolo IPv6 se define en la RFC 3493 [16] (básica) y RFC 3542 [15] (avanzada). Además, esta interfaz también se recoge en IEEE 1003.1 [22], dotándola de portabilidad e independencia de la plataforma utilizada.

5.4.2. Objetivos

Los objetivos que se han de cumplir en el diseño e implementación de la API para el protocolo IPv6, partiendo de la API del protocolo IPv4, son los siguientes:

- La nueva interfaz debe ser compatible con la API anterior, por lo que un programa diseñado con la API de IPv4 no debe fallar con la nueva API.

- Los cambios que se realicen deben ser los mínimos posibles para simplificar el proceso de adaptación de los programas ya implementados y que utilizan la interfaz de IPv4.
- La nueva interfaz debe proporcionar soporte tanto a IPv4 como a IPv6, pudiéndose establecer comunicaciones con otro ordenador utilizando tanto el protocolo IPv4 como el protocolo IPv6.
- La nueva interfaz ha de proporcionar los mecanismos necesarios para realizar un programa independiente de la versión del protocolo IP que se utilice. Así, el programador puede decidir si realizar un programa para utilizar sólo el protocolo IPv4 o sólo el protocolo IPv6 o ser independiente del protocolo que se utilice.

El principal cambio que se debe realizar en la API de IPv4 es el soporte de las direcciones IPv6. Las direcciones IPv4, al ser de 32 bits, se pueden almacenar en una variable de tipo entero; sin embargo, las direcciones IPv6, de 128 bits de longitud, necesitan una nueva estructura para almacenarla. Además, las direcciones IPv6 también necesitan que se especifique su ámbito (locales, restringidas al *site* o globales). Un ordenador puede disponer de múltiples direcciones IP, tanto IPv4 como IPv6, asignadas a un mismo nombre por lo que en el proceso de resolución de nombres también se debe incluir el tipo de direcciones que se solicitan y se devuelven. Por último, la representación de las direcciones IPv6 ha cambiado con respecto a las direcciones IPv4. En IPv4, el carácter ":" se utiliza para separar la dirección IPv4 del puerto; sin embargo, en IPv6, el carácter ":" se utiliza para separar cada uno de los grupos de 16 bits de la dirección, por lo que ahora se ha definido el siguiente formato: "[::]:", marcando la dirección IPv6 con los caracteres "[" y "]".

A modo de resumen, los cambios a realizar en la API de IPv4 son los siguientes:

- Nuevas estructuras para almacenar las direcciones IPv6.
- Nuevas funciones de traducción de nombre a dirección IP, independientemente del protocolo.
- Nuevas funciones de traducción de dirección IP a nombre, independientemente del protocolo.
- Nuevas funciones que sean capaces de convertir una cadena de caracteres a una estructura adecuada para almacenar una dirección IPv6.
- Nuevas funciones para permitir utilizar diferentes clases de tráfico, etiquetar flujos o establecer el valor máximo del número de saltos.

5.4.3. Estructuras

A continuación se van a mostrar cada una de las nuevas estructuras definidas en la interfaz y que van a dar el soporte adecuado para permitir que un programa que las utilice sea independiente de la versión del protocolo que utilice.

5.4.3.1. `in6_addr`

La estructura `in6_addr` (Implementación 32), definida en la librería `netinet/in.h`, va a contener una tabla de 16 elementos, cada uno de ellos de 8 bits, almacenando los 128 bits de una dirección IPv6. En IPv4, se utiliza la estructura `in_addr` (Implementación 33).

Implementación 32 Estructura `in6_addr`

```
struct in6_addr {
   uint8_t  s6_addr[16];
};
```

Implementación 33 Estructura `in_addr`

```
struct in_addr {
   uint32_t  s_addr;
};
```

5.4.3.2. `sockaddr_in6`

La estructura `sockaddr_in6` (Implementación 34), definida en la librería `netinet/in.h`, va a contener los siguientes elementos:

- `sin6_family`:
  Almacena el identificador de la familia de direcciones IP correspondiente. El valor para la familia de direcciones IPv6 es `AF_INET6`.
- `sin6_port`:
  Almacena el valor del puerto de la capa de transporte.
- `sin6_flowinfo`:
  Almacena tanto la clase de tráfico como la etiqueta de flujo.
- `sin6_addr`:
  Almacena la propia dirección IPv6. La constante `IN6ADDR_ANY` especifica cualquier dirección IPv6.
- `sin6_scope_id`:
  Almacena el identificador de la interfaz para el ámbito de la dirección:
  - Si el ámbito es "link-local", almacenará el índice de la interfaz.
  - Si el ámbito es "unique-local", almacenará el identificador del *site* correspondiente.

Implementación 34 Estructura `in6_addr`

```
struct sockaddr_in6 {
   sa_family_t      sin6_family;
   uint16_t         sin6_port;
   uint32_t         sin6_flowinfo;
   struct in6_addr  sin6_addr;
   uint32_t         sin6_scope_id;
};
```

En IPv4, se utiliza la estructura `in_addr` (Implementación 35).

Implementación 35 Estructura `in_addr`

```
struct sockaddr_in {
   sa_family_t     sin_family;
   uint16_t        sin_port;
   struct in_addr  sin_addr;
};
```

5.4.3.3. `sockaddr_storage`

La estructura `sockaddr_storage` (Implementación 36), definida en la librería `sys/socket.h`, es una estructura lo suficientemente grande como para contener una estructura `sockaddr_in` o `sockaddr_in6`, de forma que con una conversión directa (*cast*) se puedan ubicar correctamente. Así, se puede almacenar una dirección IP independientemente de la versión del protocolo que se utilice. En IPv4, se utiliza la estructura `sockaddr` (Implementación 37).

Implementación 36 Estructura `sockaddr_storage`

```
struct sockaddr_storage {
   sa_family_t  sin6_family;
};
```

Implementación 37 Estructura `sockaddr`

```
struct sockaddr {
   sa_family_t  sa_family;
   char         sa_data[];
};
```

5.4.3.4. `addrinfo`

La estructura `addrinfo` (Implementación 38), definida en la librería `netdb.h`, va a almacenar los resultados que se obtienen cuando se resuelve una dirección IP partir de una consulta a un servidor DNS, por ejemplo. Se utiliza en las funciones `getaddrinfo()` y `getnameinfo()`. Esta estructura va a contener los siguientes elementos:

- `ai_flags`:
  Almacena un valor determinado para limitar, en el caso que se necesite, la respuesta a obtener ante la petición de la resolución de un nombre. El valor para la función `bind()` es `AI_PASSIVE`.
- `ai_family`:
  Almacena el identificador de la familia de direcciones IP. Así, para la familia de direcciones IPv4 tomará el valor `AF_INET` y para la familia de direcciones IPv6 tomará el valor `AF_INET6`.
- `ai_socktype`:
  Almacena el tipo de socket. Así, para un socket TCP tomará el valor `SOCK_STREAM`, y para un socket UDP tomará el valor `SOCK_DGRAM`.
- `ai_protocol`:
  Almacena el identificador de protocolo del socket. Así, para IPv4 tomará el valor `IPPROTO_IPV4` y para IPv6 tomará el valor `IPPROTO_IPV6`.
- `ai_addrlen`:
  Almacena la longitud de la estructura que apunta `ai_addr`.
- `ai_addr`:
  Apunta a la estructura `sockaddr` que almacena la propia dirección IP.
- `ai_canonname`:
  Apunta a una cadena de caracteres que almacena el nombre correspondiente.

- `ai_next`:
  Apunta a la próxima estructura `addrinfo` dentro de la lista enlazada.

Implementación 38 Estructura `addrinfo`

```
struct addrinfo {
   int              ai_flags;
   int              ai_family;
   int              ai_socktype;
   int              ai_protocol;
   socklen_t        ai_addrlen;
   struct sockaddr  *ai_addr;
   char             *ai_canonname;
   struct addrinfo  *ai_net;
};
```

En IPv4, se utiliza las estructuras `hostent` (Implementación 39) y `servent` (Implementación 40).

Implementación 39 Estructura `hostent`

```
struct hostent {
   char  *h_name;
   char  **h_aliases;
   int   h_addrtype;
   int   h_length;
   int   **h_addr_list;
};
```

Implementación 40 Estructura `servent`

```
struct servent {
   char  *s_name;
   char  **s_aliases;
   int   s_port;
   char  *s_proto;
};
```

5.4.4. Funciones

A continuación se van a mostrar cada una de las nuevas funciones definidas en la interfaz y que van a dar el soporte adecuado para permitir que un programa que las utilice sea independiente de la versión del protocolo que utilice.

5.4.4.1. `getaddrinfo()`

La función `getaddrinfo()` (Implementación 41), definida en la librería `netdb.h`, se encarga de resolver el nombre de un ordenador a través de una petición a un servidor DNS o a través de un método similar. Esta función también se encarga de resolver el nombre de un servicio, obteniendo su número de puerto. Así, esta función sustituye a las funciones `gethostbyname()`

y `getservbyname()` utilizadas en IPv4. El resultado de la consulta es una lista de estructuras `addrinfo` donde se almacena toda la información. Para liberar la memoria ocupada por las estructuras `addrinfo` es necesario utilizar la función `freeaddrinfo()`.

Implementación 41 Prototipo de la función `getaddrinfo()`

```
int getaddrinfo (const char *nodename,
                 const char *servname,
                 const struct addrinfo *hints,
                 struct addrinfo **res);
```

Esta función permite los siguientes parámetros:

- `nodename`:
  Cadena de caracteres que almacena el nombre del ordenador.
- `servname`:
  Cadena de caracteres que almacena el nombre del servicio.
- `hints`:
  Estructura `addrinfo` que se utiliza para indicar las opciones de resolución del nombre a través de un servidor DNS o similar.
- `res`:
  Lista enlazada de estructuras `addrinfo` que almacena los resultados, almacenando cada elemento un único resultado.

5.4.4.2. `getnameinfo()`

La función `getnameinfo()` (Implementación 42), definida en la librería `netdb.h`, se encarga de obtener el nombre de un ordenador a partir de su nombre a través de una petición a un servidor DNS o a través de un método similar. Esta función también se encarga de obtener el nombre de un servicio a partir del número del puerto. Así, esta función sustituye a las funciones `gethostbyaddr()` y `getservbyport()` utilizadas en IPv4. El resultado de la consulta es un puntero a una cadena de caracteres que almacena el nombre del ordenador solicitado y un puntero a una cadena de caracteres que almacena el nombre del servicio solicitado.

Implementación 42 Prototipo de la función `getnameinfo()`

```
int getnameinfo (const struct sockaddr *sa,
                 const socklen_t salen,
                 const char *node,
                 const socklen_t nodelen,
                 const char *service,
                 const socklen_t servicelen,
                 int flags);
```

Esta función permite los siguientes parámetros:

- `sa`:
  Apunta a la estructura `sockaddr` que se desea resolver. En el caso de que contenga una dirección IPv6 con correspondencia IPv4 (*IPv4 mapped*) o traducida (*IPv4 translated*), la interfaz extraerá la dirección IPv4 correspondencia y la resolverá.

- `salen`:
  Longitud de la cadena de caracteres `sal`.
- `node`:
  Cadena de caracteres que almacena el nombre de la máquina que se ha obtenido tras la consulta.
- `nodelen`:
  Longitud máxima de la cadena de caracteres `node`.
- `service`:
  Cadena de caracteres que almacena el nombre del servicio que se ha obtenido tras la consulta.
- `servicelen`:
  Longitud de la cadena de caracteres `service`.
- `flags`:
  Almacena un valor que varía el comportamiento por defecto de esta función.

5.4.5. Comparativa

La Tabla 5.1 muestra una comparación entre las estructuras y funciones disponibles para cada uno de los protocolos.

Protocolos		*Independiente del protocolo*	*IPv6*	*IPv4*
Estructuras de datos		`sockaddr_storage` `addrinfo`	`in6_addr` `sockaddr_in6`	`in_addr` `sockaddr_in`
Resolución de nombres	Nombre a dirección	`getaddrinfo()`	`getipnodebyname()`	`gethostbyname()`
	Dirección a nombre	`getnameinfo()`	`getipnodebyaddr()`	`gethostbyaddr()`
Conversión de direcciones	Caracteres a dirección		`inet_pton()`	`inet_aton()`
	Dirección a caracteres		`inet_ntop()`	`inet_ntoa()`

Tabla 5.1: Comparación entre la API de IPv4 e IPv6

5.4.6. Ejemplos

El extracto de código, que se muestra en Implementación 43, implementa el proceso por el cual, a partir del nombre de la máquina a la que queremos conectarnos y el nombre el servicio que deseamos solicitar, obtenemos la dirección IPv6 del servidor y el número del puerto al que enviar la petición.

En el extracto de código, que se muestra en Implementación 44, se implementa el proceso por el cual, a partir del nombre del servicio que deseamos ofrecer, obtenemos el número de puerto donde ofrecerlo.

Implementación 43 Extracto de código de un cliente

```
/* Variables iniciales */
struct addrinfo *res;
struct addrinfo hints;
int s;

/* Rellenamos con ceros la estructura hints */
memset(&hints,0,sizeof(hints));

/* Protocolo TCP */
hints.ai_socktype = SOCK_STREAM;

/* Realizamos la resolución */
getaddrinfo(‘‘nombremaquina’’, ‘‘servicio’’, &hints, &res);

/* Creamos el socket de comunicaciones */
s = socket (res->ai_family, res->ai_socktype, res->ai_protocol);

/* Conectamos con el servidor */
connect (s, res->ai_addr, res->ai_addrlen);
```

Implementación 44 Extracto de código de un servidor

```
/* Variables iniciales */
struct addrinfo *res;
struct addrinfo hints;
int s;

/* Rellenamos con ceros la estructura hints */
memset(&hints,0,sizeof(hints));

/* Protocolo TCP */
hints.ai_family = AF_INET6;
hints.ai_flags = AI_PASSIVE;
hints.ai_socktype = SOCK_STREAM;

/* Realizamos la resolución */
getaddrinfo(NULL, ‘‘servicio’’, &hints, &res);

/* Creamos el socket de comunicaciones */
s = socket (res->ai_family, res->ai_socktype, res->ai_protocol);

/* Vinculamos el puerto */
bind (s, res->ai_addr, res->ai_addrlen);
```