Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros de Telecomunicación

IPv6 Y PROTOCOLOS DE TRANSPORTE DE NUEVA GENERACIÓN

TRABAJO FIN DE MÁSTER

Ernesto García Muñoz

2014

Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros de Telecomunicación

Máster Universitario en
Ingeniería de Redes y Servicios Telemáticos

TRABAJO FIN DE MÁSTER

IPv6 Y PROTOCOLOS DE TRANSPORTE DE NUEVA GENERACIÓN

Autor
Ernesto García Muñoz

Director
David Fernández Cambronero

Departamento de Ingeniería de Sistemas Telemáticos

2014

Resumen

El objtivo inicial con el que se planteó este Trabajo Fin de Máster era estudiar, a nivel lo más práctico posible, nuevos protocolos de nivel de transporte en el contexto de la actual transición del nivel de red. Para ello se tomó como punto de partida una aplicación de tipo cliente/servidor con funcionalidad de shell remoto, multiplataforma, (mal) escrita en 2010 y diseñada para trabajar con TCP y/o UDP sobre IPv4, y se ha migrado para adaptarse a los cambios que están teniendo actualmente en Internet. Usar unos protocolos de nivel de transporte nuevos en una aplicación sobre una versión de IP que se ha quedado obsoleta es contradictorio, por lo que se decidió aumentar el ámbito del trabajo para incluir IPv6 e ICMPv6. En concreto se ha hecho una reestructuración completa de la aplicación para poder usar los protocolos de transporte SCTP y MPTCP tanto sobre IPv4, como dual-stack o IPv6. El camino seguido a lo largo de la elaboración del presente trabajo ha permitido estudiar en mayor profundidad IPv6 y la migración de IPv4 a IPv6, conocer las técnicas y recomendaciones para la transición de IPv4 a IPv6, tener un primer contacto con protocolos de transporte de nueva generación como son SCTP y MPTCP, volver a evidenciar las complejidades y limitaciones de la API Socket y seguir puliendo técnicas de programación en general. Durante la migración del código y gracias a los conocimientos adquiridos se han escrito tres librerías: una para facilitar el uso de la API Socket, otra para construir datagramas IPv4 o IPv6 y mensajes ICMPv4 o ICMPv6, y la tercera para facilitar la portabilidad multiplataforma de una aplicación que es uno de los requisitos del software producido. Esta memoria, junto con el código escrito y publicado en GitHub es el resultado del trabajo realizado.

TCP y UDP han funcionado, y siguen funcionando, muy bien. Entonces, ¿por qué habría que cambiarlos? Ésta es una de las preguntas que se intentan contestar. El contexto en el que se ubica este trabajo es Internet, y su modelo de referencia TPC/IP. Según el modelo de referencia de TCP/IP una red se descompone en los niveles de enlace, red, transporte y aplicación. De ellos, todos, excepto el nivel de transporte, han evolucionado. Por ejemplo, la tecnología inalámbrica ha irrumpido, lo cual ha hecho que el nivel de enlace cambie, el nivel de red está en un complejo proceso de actualización debido al agotamiento de direcciones. Por su parte, el nivel de aplicación ha sufrido una auténtica revolución debido a la complejidad de las aplicaciones actuales y a la evolución en el hardware y en los terminales, y esta complejidad hace que el número de líneas de código de un programa sea, hoy en día, muy elevado. El aumento de interfaces de un terminal, además de la evolución de Internet, hace que los programas que usan Internet requieran técnicas específicas para facilitar el establecimiento de la comunicación y el mantenimiento de ésta. Una manera de disminuir el número de líneas de un programa para simplificarlo, o facilitar la implementación de nuevo software, es mover funcionalidad del nivel de aplicación al nivel de transporte, lo cual además facilitaría su estandarización por parte del IETF. Esto está ocurriendo en la actualidad con MPTCP, que es una extensión de TCP para soportar múltiples caminos entre dos nodos de la red, y con SCTP, que es un protocolo relativamente nuevo que combina lo mejor de TCP y UDP. Debido a la complejidad de Internet y a lo proliferación de dispositivos *middlebox*, introducir nuevos protocolos en cualquiera de los niveles del modelo de referencia TCP/IP no es una empresa sencilla. En concreto la actual migración del nivel de red se está dilatando en el tiempo además de estar impactando en todos los demás niveles, lo cual contradice los principios de división en capas y encapsulación. Sin embargo, la actual migración de IPv4 a IPv6 podría usarse para actualizar, por ejemplo, el nivel de transporte, y este es el motivo por el cual, en este momento concreto, me haya interesado, no solo por la migración hacia IPv6, sino también por alternativas y evoluciones en el nivel de transporte.

Abstract

The original goal of this piece of work was very clear: to study, following a practical approach, new transport layer protocol within the current context of network layer migration. In order to accomplish that, the starting point was a client/server multiplatform application with remote terminal functionality, (badly) written in 2010 and designed to work with TCP and/or UDP over IPv4, and it has been migrated to adapt to current reality of the Internet. To study new transport layer protocols over a deprecated version of the IP protocol makes no sense, therefore IPv6 and ICMPv6 were included in the scope of this work. The application has been completely refactored to ease the addition of new transport protocols such as SCTP and MPTCP, and to be able to transparently work over IPv4, dual-stack, or IPv6. Doing that allowed to gain deeper understanding of IPv6 and its migration process, to have a first contact with new transport protocols, to arise once more the complexities associated with Socket API and to polish up programming skills. The result of the migrations and experimentation done are three libraries: one to handle in the easiest possible way the Socket API, another to craft IPv4 and IPv6 datagrams and ICMPv4 and ICMPv6 messages, and the last once to help writing multiplatform code. The present document, together with the code written and published in GitHub is the result of this work.

TCP and UPD protocols have been working very well. Then, why would we consider updating or changing them? This is one of the key questions that this work will try to answer. The context of this work is Internet and its reference model, TCP/IP. TCP/IP reference model decomposes the network in layers: link, network, transport and application. Except transport layer, all of them have evolved. For instance, wireless technology break out forced link layer to adapt to it, network layer is in the middle of a complex migration process nowadays due to the IPv4 addresses exhaustion. In the case of the application layer there has taken place a revolution in software functionality that has overcomplicated and has increased the number of lines of code. The increment of functionality in software has the side effect of over complicating it, which jeopardizes development, maintenance and quality in general. When it makes sense, moving functionality from application layer to transport layer would help to simplify applications development and maintenance, at the same time that will facilitate standardization of the new transport layer functionality thanks of the bodies such as IETF. Although migration from IPv4 to IPv6 is taking longer than expected, the depth in the involved changes could set the ground for further updates and additions in different layers, such as transport, if there exists the need. This is how, from my point of view, network, transport and application layers are linked and related in current scenario of the growing and mutating Internet.

Índice

Índice de ilustraciones

1. Introducción

1.1 Breve repaso histórico

Imaginar un mundo sin Internet[1] es hoy en día imposible. La red de redes llegó de forma rápida y actualmente es parte de la sociedad. Desde un punto de vista práctico es justo afirmar que Internet ha sido todo un éxito. La fortaleza de su arquitectura fue, en pasado, su dinamismo, que se tradujo en capacidad de escalar a nivel global y en capacidad de adaptación. Además su diseño tuvo lugar en un corto periodo ya que la gestión de sus protocolos y modelo de referencia fue rápida permitiendo implementar y desplegar sobre la marcha. El modelo de referencia de la arquitectura de Internet nació a partir de unos protocolos que ya estaban diseñados e inspirándose en otro ya existente de origen militar: ARPANET[2].

Internet no solo ha conseguido sobrevivir durante años, sino que además es un fenómeno sociocultural a nivel planetario digno de estudio, análisis y reflexión. Este paradigma ágil de diseño, implementación y despliegue usado para diseñar TCP/IP[3] es el opuesto al que se usó en aquella época con otro modelo de referencia: OSI[4]. OSI nació de forma muy ortodoxa, tal vez demasiado, y de manera opuesta como se hizo con TCP/IP. Con OSI lo primero que se hizo fue estandarizar el modelo de referencia, esto es, cómo una serie de protocolos de comunicaciones iban a relacionarse entre sí. Probablemente se sobre-diseñó, sin tener en cuenta si los fabricantes, que son los que tienen la última palabra, lo adoptarían. Mientras OSI se discutía y estaba en sus inicios, Internet ya estaba desplegándose y en funcionamiento: nacieron los protocolos y se desarrollaron, los fabricantes los adoptaron y por último se escribió el modelo de referencia TCP/IP, probablemente por pura formalidad técnica. Una vez que los fabricantes se dieron cuenta de que funcionaba ya no hubo marcha atrás y nadie se arriesgó a continuar por el camino que OSI ofrecía. El legado de OSI ha quedado relegado a explicar a alumnos universitarios cómo modelar una arquitectura de red mediante capas o niveles y protocolos. OSI también es útil para describir otros protocolos ya que fue vislumbrado antes de que los protocolos en sí fueran inventados. Según [TANNENBAUM] OSI establece:

- Una capa debe crearse cada vez que existe la necesidad de una abstracción diferente.
- Cada capa debe realizar una función bien definida.
- La función de cada capa debe ser elegida en vistas a definir protocolos estandarizados.
- Los límites entre capas deben elegirse de tal manera que se minimice la información que fluye a través de los interfaces.
- El número de capas debe ser suficiente para que cada capa tenga una función bien definida sin que la arquitectura sea inmanejable.

Se hace mención a OSI ya que en el desarrollo del presente trabajo se han tenido muy en cuenta los mencionados puntos a la hora de diseñar las librerías implementadas, haciendo especial hincapié en la abstracción entre capas y el flujo de información entre ellas. Esto es así porque, como se describirá posteriormente, en el trabajo se ha usado TCP/IP, pero aunque TCP/IP se describe mediante capas bien definidas su implementación, el interfaz de programación (Berkeley Socket API[5]) que ofrece es obsoleto, complejo, sin una estructura clara de la información que fluye entre capas, incluso sin una diferenciación interna clara de capas.

[1] Internet: red para interconectar redes
2 ARPANET: es un modelo de referencia creado por DARPA (proyectos avanzados de defensa) para crear una red resistente a un ataque nuclear.
3 TCP/IP: suite de protocolos de Internet que tiene sus orígenes en ARPANET.
4 OSI: Open Systems Interconnection.
5 API: interfaz de programación

1.2 El apocalipsis de los dos elefantes

El dinamismo de Internet, que es la clave de su éxito, es a su vez su mayor debilidad: se buscó una solución rápida y sencilla que simplemente funcionara. El mantenimiento de Internet durante los últimos años ha consistido en reestructurar sobre lo edificado no teniendo en cuenta en su diseño cómo actualizar y evolucionar a largo plazo. Esto indica que nadie previó en su día la expansión y potencial que tenía, siendo, además, fruto del paradigma de trabajo que se usó: una vez que los protocolos estaban implementados y diseñados, y los fabricantes empezaron a usarlos, fue cuando se escribió su modelo de referencia sobre la marcha

La teoría *"Apocalypse of the two elephants"* [CLARK] [TANNENBAUM] explica muy bien lo ocurrido con OSI y TCP/IP. Según esta teoría, los estándares deben escribirse entre el pico de investigación teórica y la inversión monetaria para su implementación. Si la estandarización se hace muy pronto significará que la tecnología no se ha asimilado correctamente. Si la estandarización se hace muy tarde, cada compañía que invierta en la tecnología lo habrá hecho a su manera desviándose del estándar. En TCP/IP primero se hizo la inversión y luego se escribió el estándar. Con OSI se tardó tanto tiempo en tener un modelo de referencia que al final nadie lo usó (ya estaba TCP/IP *up-and-running*). TCP/IP fue una solución sencilla y elegante, a su manera, que tuvo éxito, pero en la actualidad hay una serie de retos que están resultando un auténtico quebradero de cabeza, que es lo que se describe a continuación.

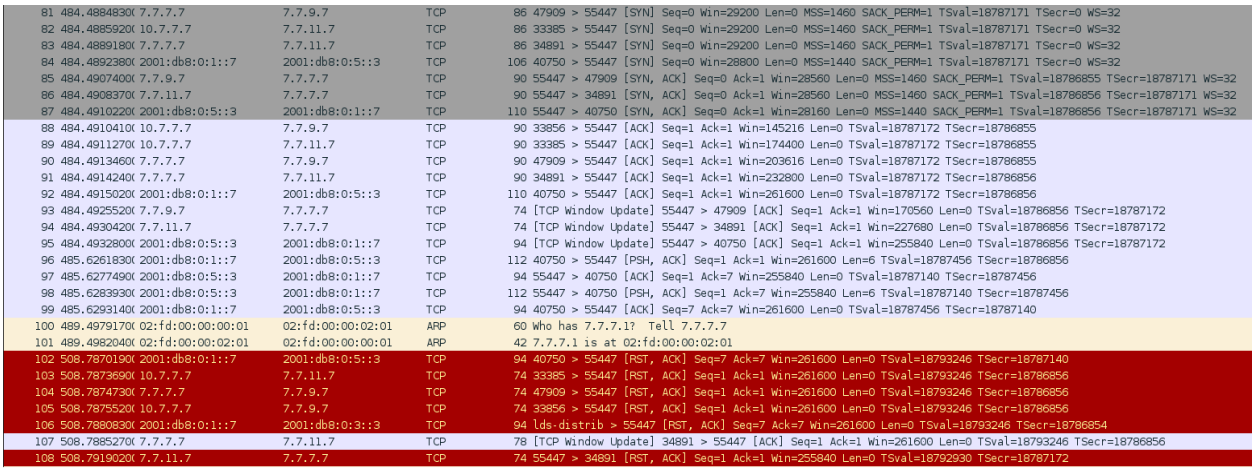


Ilustración 1: protocolos de TCP/IP en acción (captura de herramienta Wireshark)

Aplicación: HTTP, HTTPS, DNS, Skype
Transporte: TCP, UDP, SCTP, MPTCP
Red: IPv4, IPv6, ICMPv4, ICMPv6
Enlace: Ethernet, WiFi
Físico: Fibra óptica

Tabla 1: torre de TCP/IP con algunos protocolos

Internet no ha cambiado desde su nacimiento. Los protocolos de nivel de red y de transporte siguen siendo los mismos: IP[6], TCP[7] y UDP[8]. El funcionamiento es simple: los ordenadores y dispositivos en la red se identifican y localizan mediante lo que se llama direcciones IP (similar

6 IP: protocolo de Internet (nivel de red).
7 TCP: protocolo de control de transmisión (nivel de transporte).
8 UDP: protocolo de intercambio de datagramas (nivel de transporte).

a los números de teléfono de toda la vida). Hay dispositivos finales y dispositivos enrutadores y servidores que ayudan a que los paquetes con los datos viajen a través de la red. Los principales quebraderos de cabeza y problemas vienen a raíz de cómo se identifican los dispositivos en la red: un ordenador o dispositivo se identifica con un número llamado dirección IP. Sólo se reservaron 32 bits (2^32 direcciones IP) para las direcciones IP, y esto hace que actualmente haya más dispositivos y ordenares que direcciones disponibles, por lo que la escalabilidad de Internet, no es que peligre, es que ya ha desaparecido. Y aquí es donde hace su aparición estelar NAT[9]: el "hack" que se cuajó en la industria para solucionar los problemas de agotamiento de direcciones IP, y a su vez violar de forma flagrante varios de los principios sobre los que se basaba la arquitectura de Internet [RFC1958].

De los dos elefantes, OSI fue tan lento en el diseño de su modelo de referencia que acabó antes de empezar, mientras que TCP/IP se desarrolló tan rápido que ahora está resultando muy difícil actualizar uno de sus niveles. ¿Con OSI estaríamos ahora ante estos problemas?

1.3 RFC 1958 y breve reseña biográfica sobre el NAT

El dispositivo NAT, en combinación con un software cortafuegos sencillo y un router[10] es lo que casi todos tenemos en nuestras casas para conectar nuestros ordenadores y dispositivos a la red. Gracias al NAT los dispositivos conectados a la red (doméstica o corporativa) comparten una única dirección IP mediante el uso de una subred privada con su propio espacio de direcciones privado y reservado. NAT no es más que una tabla que mapea direcciones de red y puertos internos a direcciones IP y puertos externos. Esto, que por un lado soluciona el problema crítico del agotamiento de direcciones crea otra nueva generación de retos, además de añadir un nivel de complejidad extraordinario a las aplicaciones que utilizan Internet para comunicarse. En este punto Internet se les escapó de las manos a sus "creadores" y pasó a estar bajo control total de los fabricantes de dispositivos. Al IETF[11] no le quedó otra alternativa que escribir las correspondientes RFCs[12] para intentar armonizar el despliegue y uso de los traductores de direcciones. Y empiezan a aflorar las deficiencias de Internet: lo que hasta entonces había sido dinamismo, escalabilidad y éxito se convierte en algo muy complejo de mantener y actualizar, cosa que se ha ido haciendo últimamente a base de parches y soluciones muy poco elegantes (como por ejemplo NAT en cualquiera de sus sabores).

En *"Architectural Principles of the Internet"* [RFC1958] se establecieron algunos puntos básicos como que:

- El único principio de Internet es el continuo cambio.
- El objetivo de *ésta RFC* [RFC1958] no es dogmatizar sobre cómo debe desarrollarse Internet.
- ¿Hay realmente una arquitectura en Internet?
- Lo más importante de Internet es la conectividad E2E[13].
- Como el nivel de red es el caballo de batalla de Internet, es sentimiento general que un único protocolo de red hará las cosas más sencillas aunque se deja la puerta abierta a varios
- La inteligencia se delega a los extremos de la red.
- Entre otros…

La [RFC1958] -la última versión data del año 1996- comienza dejando claro que el único principio absoluto aplicable a Internet es el cambio y además se cuestiona si se puede hablar de arquitectura de Internet. Esta ambigüedad es probable que se deba a que en 1996 ya advirtieron

[9] NAT: traductor de direcciones de red.
[10] Router: encaminador de nivel 3.
[11] IETF: Internet Engineering Task Force es el organismo encargado de escribir los estándares (RFCs) de Internet.
[12] RFC: Request For Comments, o documento que describe formalmente protocolos o componentes de Internet.
[13] E2E: End-two-End o extremo a extremo.

las dificultades inmediatas que se avecinaban. Sin embargo se establece de forma clara que la parte crítica es la interconexión extremo a extremo entre aplicaciones, por lo que la capa más importante es la de red. La conectividad E2E es, dentro de la ambigüedad del documento, algo importante. Pues bien, precisamente es esa conectividad E2E de la que se habla en [RFC1958] la que se viola con los dispositivos NAT. Los dispositivos NAT dificultan, por no decir imposibilitan, la conectividad directa y natural entre dispositivos finales debido a cómo funcionan:

Como se ha mencionado anteriormente, un dispositivo NAT se implementa mediante una tabla. En esta tabla se hace un mapeo entre direcciones IP y puertos internos (usados por ordenadores de la subred privada) y direcciones IP y puertos externos. Para que esta tabla tenga una entrada es necesario que un dispositivo de su red privada, es decir interno, mande tráfico hacia Internet (*por favor, dejen salir antes de entrar*). Si un dispositivo permanece inactivo o a la espera en la subred privada no tendrá entrada en la tabla del NAT, por lo que no hay manera alguna de que otro dispositivo en cualquier otra parte de Internet pueda conectar con él, es decir, no hay principio E2E. Cuando desde casa con mi portátil conectado al router mediante, por ejemplo, WiFi[14] abro el navegador y pongo una página en la barra de direcciones, de una forma muy simplificada lo que ocurre es que mi portátil, con una dirección IP privada como puede ser 192.168.0.17, manda un mensaje HTTP a un servidor que está en alguna parte. Mi ordenador manda el mensaje a través del interfaz inalámbrico WiFi, y cuando éste llega al router, la funcionalidad NAT mira la dirección destino (del servidor Web) y traduce la dirección del dispositivo origen a la dirección pública que el proveedor de servicio de Internet ha suministrado[15] (la que resulta accesible al resto de equipos de Internet). Así cuando el servidor Web contesta, lo hace, no a la dirección IP privada del peticionario, sino a la IP pública que tiene asignado el router en su interfaz externa. Cuando la repuesta llega al router, éste mira en su tabla NAT y traduce la dirección destino a la dirección IP privada del ordenador que envió la petición. NAT significa traductor de direcciones de red, y precisamente traducir es su trabajo.

Si en teoría parece complejo, en la práctica se complica aún más por el hecho de que hay varios tipos de NATs, y de que la nomenclatura usada en su descripción es ambigua y errónea. El objetivo de este Trabajo Fin de Máster no es profundizar más en NATs, sino analizar y experimentar de forma práctica con las otras soluciones a largo plazo. Más información sobre NAT en [TCP/IP-1-12].

1.4 Evolución del nivel de red de IPv4 a IPv6

La solución a largo plazo no es NAT (se va a procurar no hablar más sobre NAT en lo que resta de memoria), sino IPv6[16]. *"Internet Protocol, Version 6 (IPv6) Specification"*, [RFC2460], la especificación del IETF para IPv6, data de 1998 (no muy distante de la ya mencionada [RFC1958]…). IPv6 es una evolución de IPv4, en la que los principios básicos de funcionamiento no se tocan. Se modifica la cabecera y se pulen algunos aspectos como la seguridad, pero la idea original es la misma (es la misma porque el punto de partida es el correcto). Sigue pasando el tiempo pero IPv6 no se adopta, porque los cambios en la infraestructura y sobre todo en las aplicaciones son muy profundos.

[14] WiFi: Wireless Fidelity

[15] Es un ejemplo para que mi abuela entienda como funciona el Internet obviando DHCP, ARP, DNS y el hecho de que el número de puerto de transporte también se usa en la mayoría de los casos. Además hay diferentes tipos de NAT que hacen diferentes tipos de traducciones, para hacer las cosas aún más entretenidas

[16] IPv6: protocolo de nivel 3 Internet, version 6

1.5 Revolución del nivel de aplicación

Estructurar un sistema tan complejo como una red de comunicaciones en capas o niveles fue un gran acierto, y el hacerlo de una manera tan sencilla como en el modelo de referencia TCP/IP supuso gran parte del éxito que ha tenido Internet. Sobre el papel, Internet está formado de dispositivos finales, dispositivos enrutadores, y servidores que ayudan a la transferencia de datos entre diferentes máquinas en la red. Los dispositivos enrutadores deberían implementar únicamente hasta el nivel de red, los servidores deberían ofrecer sus servicios mediante protocolos estandarizados (el ejemplo más obvio es DNS) y los dispositivos finales implementarían todas las capas de la torre de protocolos.

En los primeros años del *boom* de Internet, la mayor complejidad del código estaba en los niveles inferiores, ya que la complejidad de las primeras aplicaciones en aquel momento era limitada, en parte por el hardware del que se disponía. Pero gracias a la encapsulación o a la división en capas o niveles, las aplicaciones de usuario que hacían uso de Internet no tenían que implementar todos los niveles de la torre de protocolos, porque los niveles de transporte y de red (que eran los que más líneas de código requerían) estaban implementados en el kernel del sistema operativo y eran accesibles mediante la API Sockets. Y por aquellos primero años el número de líneas de código de una aplicación, en comparación con el número de líneas de código de los niveles de red y de transporte, era pequeño. Esto permitió escribir tantas aplicaciones que hicieran uso de TCP/IP como se desease, y así nació HTTP y la Web, el correo electrónico, IRC y los servicios de mensajería instantáneos, la seguridad en las comunicaciones, los servicios multimedia, etc. Tenía sentido que el código común usado por el nivel de aplicación se colocase en el sistema operativo para minimizar el tamaño de las aplicaciones.

Ha pasado el tiempo, la red ha crecido como nadie había imaginado, las direcciones IPv4 se agotan debido a la popularidad de Internet, y además, la complejidad del nivel de aplicación, esto es, de las aplicaciones y servicios que están sobre TCP/IP según el modelo de referencia, ha crecido considerablemente. Esto hace que aplicaciones que necesitan comunicarse en red, o lo que es mismo, que hacen uso del Internet y de TCP/IP, tengan un tamaño comparable al de TCP e IP juntos, si no mayor incluso. El que el número de líneas de código de la parte del kernel de un sistema operativo que implementa TCP/IP sea de millones de líneas de código no es un gran problema, ya que es un servicio centralizado, en el sentido de que no hay más alternativas que TCP sobre IP, o UDP sobre IP y a que gracias al trabajo y esfuerzo del IETF, de la industria, de la gente que contribuye a Internet, y gracias a que no existe una fragmentación en los niveles de red y de transporte se han podido mantener y desplegar en todos los dispositivos que existen en el planeta con éxito. Los niveles de red y transporte, donde todo es homogéneo y el código es abierto, donde no existe fragmentación ni varios protocolos con diferente nombre pero que hacen lo mismo, gracias al IETF, que se encarga de intentar estandarizar de forma rigurosa los protocolos y poner un poco de orden, y gracias a una API, que puede ser mejor o puede ser peor, pero es más o menos homogénea y está bien documentada, los actuales niveles de red (IP independientemente de la versión) y de transporte (TCP y UDP) son longevos y todo apunta a que van a seguir siéndolo.

En contraste, el nivel de aplicación es un auténtico caos. El nivel de aplicación se ha convertido en una auténtica marabunta de diferentes aplicaciones, muchas de las cuales hacen lo mismo, algunas peor que otras pero muchas veces ninguna de forma completamente correcta, diferentes protocolos de nivel de aplicación, algunos estandarizados más mal que bien y otros directamente no estandarizados o propietarios u ofuscados, y en general un ecosistema de aplicaciones cuya tendencia es complicarse y crecer considerablemente en tamaño y en número. En resumen, no existe ningún orden, o si existe yo no lo he visto, a nivel de aplicación, donde cada fabricante intenta

ganar cuota de mercado con alguna aplicación o con algún protocolo propietario y cerrado que fuerce el uso de su plataforma.

Cierto es que, en parte, el aumento de la complejidad de las aplicaciones es debido a cómo ha mutado Internet desde el tradicional esquema *end-to-end* a lo que hay en la actualidad, donde el incesante incremento de dispositivos de tipo *middlebox* violan el principio de conectividad extremo a extremo tradicional. En este escenario nuevo que se planteó hace unos años con la introducción de dispositivos adicionales en la red, con el fin del principio extremo a extremo (si alguna vez existió de forma pura y genuina) y con el desarrollo de algoritmos y técnicas que mejoran las prestaciones de los protocolos de nivel de transporte, las aplicaciones han engordado. El aumento de tamaño de las aplicaciones hace que la calidad del servicio que ofrecen disminuya ya que su mantenimiento se complica en exceso y su desarrollo o evolución se ralentiza, si no es que queda paralizado completamente. Lo peor de todo es que muy probablemente varias aplicaciones estén resolviendo el mismo problema de forma simultánea, lo que hace el código se duplique, y esta falta de coordinación hace, además, que la interoperabilidad sea inexistente, añadiendo una dimensión a la ecuación.

Adaptar los niveles de transporte a la realidad de Internet y adoptar y estandarizar algunas técnicas que hasta ahora se han implementado a nivel de aplicación, puede aumentar las prestaciones, disminuir la complejidad, homogeneizar ligeramente el ecosistema y probablemente facilitar la interoperabilidad entre aplicaciones. En Internet y en TCP/IP si no existiera la figura del IETF como organismo estandarizado las grandes multinacionales aumentarían aún más su posición de dominio, lo cual conduciría a un pseudo-monopolio (aún mayor del que ya hay). Internet no puede estar controlado por multinacionales siendo la figura del IETF imprescindible para coordinar el trabajo. Y esta marabunta de aplicaciones, de tira-y-afloja entre compañías multinacionales, entre el agotamiento de direcciones IPv4 y la desviación de la idea original de Internet, desde el principio de conectividad *extremo-a-extremo* a lo que haya hoy en día, que nadie es capaz de definir, nadie sepa en su conjunto, es la ubicación y el contexto del presente trabajo fin de máster, y la razón por la que se decidió echar un vistazo y jugar con los “nuevos” protocolos de nivel de transporte.

Tal vez haya llegado el momento de hacerle una liposucción al nivel de aplicación y mover funcionalidad, soluciones y algoritmos que se adaptan a los nuevos terminales y la nueva realidad de Internet del nivel de aplicación al nivel de transporte en aras a simplificar las aplicaciones, reducir su número de líneas de código, estandarizar mediante el IETF y en general permitir que se siga evolucionando de manera más armoniosa y coordinada, si no es ya demasiado tarde.

Se puede argumentar, por ejemplo en el caso de MPTCP (o en la potencial futura capacidad multicamino de SCTP) que el nivel de transporte no es el apropiado sino alguno superior, pero el modelo de referencia TCP/IP no tiene nivel de sesión, ni nivel de presentación. Probablemente desde un punto de vista purista el nivel de conexión de MPTCP esté ubicado en el nivel de sesión de OSI, no estoy seguro, pero lo que si tengo claro es que de seguir aumentando el tamaño de las aplicaciones, la interoperabilidad y en general el futuro de Internet como fue ideado peligra seriamente.

Comenzando por la migración de IPv4 a IPv6, ya que no tiene sentido usar protocolos de transporte nuevos sobre una versión antigua de IP, se procede a estudiar las últimas tendencias en protocolos de nivel transporte en la actualidad.

2. Objetivos

IPv6 es el caballo de batalla y la base sobre la que parte este trabajo. Migrar de IPv4 a IPv6 está siendo un camino largo y difícil en el que no solo hay que adaptar la red en sí, sino que también hay que adaptar las aplicaciones. Esto echa por tierra los principios de encapsulación de la torre de protocolos, ya que cambios en el nivel de red tienen su impacto en los niveles superiores. No pretendo argumentar en contra del modelo de referencia de Internet y su arquitectura (hicieron un buen trabajo en su época). El objetivo de este Trabajo Fin de Máster es analizar siempre desde el punto de vista práctico la migración de IPv4 a IPv6 y las oportunidades que esta migración puede, potencialmente, abrir a nuevos protocolos de transporte modernos como SCTP[17] y MPTCP[18]. TCP y UDP siguen siendo los protocolos usados por la gran mayoría de aplicaciones y servicios hoy en día y es difícil pensar en que esto vaya a cambiar, pero existen alternativas interesantes. Los contendientes son SCTP, un protocolo completamente nuevo, y MPTCP, que es una evolución continuista de TCP. Y éste es el contexto sobre el que gira el presente trabajo:

1. La problemática asociada al agotamiento de direcciones IPv4 como introducción.
2. IPv6 como solución a largo plazo, elegante y ortodoxa.
3. Migración de una aplicación de usuario de IPv4 a IPv6 (¿es complicado?).
4. Aprovechar los cambios que hay que hacer en las aplicaciones al migrarlas a IPv6 para experimentar con nuevos protocolos de transporte. Los equipos de red y los dispositivos han evolucionado por lo que tal vez sea hora de que también lo hagan los protocolos.
5. Evaluar el funcionamiento y utilidad de SCTP y MPTCP.

Actualmente el uso de SCTP está casi en su totalidad limitado al transporte de señalización IP en el núcleo de red de los operadores de Internet. SCTP es un protocolo completamente nuevo que combina todas las ventajas de TCP y UDP mejorando, además, sus características funcionales y no funcionales y resolviendo sus deficiencias. Al ser un protocolo nuevo requiere adaptar la actual infraestructura para su uso, y esto es su principal debilidad. MPTCP es un protocolo actualmente en fase experimental que aspira a ser una versión mejorada, pero continuista de TCP. MPTCP no requiere cambios en la red, ya que es un versión *multi-path* de TCP, lo cual juega a su favor. La debilidad de MPTCP es que las aplicaciones que hacen uso de UDP (el otro protocolo popular del nivel de Transporte) no pueden beneficiarse de él.

Éste es el contexto sobre el que va a girar el presente trabajo: IPv6, SCTP y MPTCP.

2.1 Metodología

Internet es un pozo sin fondo fascinante desde el punto de vista técnico: cuanto más lo estudias más profundo caes y más se da uno cuenta de lo complejo que es. Los retos actuales para modernizar Internet están al mismo nivel de los éxitos e hitos que ha logrado. Es por eso que la metodología que he seguido a lo largo de todo este trabajo es la misma que se siguió en su día con TCP/IP: esta memoria ha sido, como en su día ocurrió con el modelo de referencia TCP/IP, el último de los pasos (y también al que menos tiempo se le ha dedicado). Esta memoria es la última página que he escrito desde que hace ya varios años tuviera curiosidad por entender, de la forma más precisa posible, el funcionamiento de los protocolos de comunicaciones de Internet, lo que me llevó a dedicar mucho tiempo a estudiar lo que durante Ingeniería de Telecomunicación no estudié, me llevó de vuelta a la Universidad para estudiar el Máster (y redimir pecados anteriores), e incluso me ha llevado a irme a vivir al extranjero. Aunque parezca exagerado, así ha sido.

[17] SCTP: protocolo de nivel 4, Stream Control Transmission Protocol.
[18] MPTCP: protocolo de nivel 4 Multi Path Transmission Control Protocol.

El resultado del trabajo realizado durante estos meses ha sido una mejor compresión de la complejidad de Internet, que se ha materializado en tres librerías (dos de la cuales tienen como objetivo trabajar con sockets para comunicar aplicaciones) y dos aplicaciones. Aproximadamente el 70% del trabajo ha sido programando o depurando el código, un 20% probando en entornos multiplataforma (FreeBSD, Windows, Linux, JUNOS), a veces virtualizados mediante VNX, y el 10% restante ha sido el necesario estudio teórico. La metodología empleada en general ha sido "ortodoxa": primero analizar el problema desde un punto de vista teórico para posteriormente sintetizar un producto práctico, en este caso código. Debido a la complejidad de los escenarios, VNX ha sido, otra vez, de gran ayuda para validar las librerías y aplicaciones en escenarios prácticos.

La presente memoria tiene como objetivo poner en palabras y plasmar en un documento de texto el trabajo que se ha hecho, presentado unos ejemplos genuinos, el código desarrollado, los escenarios diseñados y las conclusiones obtenidas. El trabajo que se presenta en este documento no pretende ser de corte teórico, sino todo contrario, lo más practico posible (aunque cuando sea imprescindible, por coherencia y cohesión, se describen aspectos teóricos sacados de las referencias manejadas). No espere el lector otra síntesis de conceptos teóricos sobre protocolos y lenguajes de programación, no es ésta la meta perseguida (aunque habrá momentos en los que no quedará más remedio que hacerlo), entre otras cosas porque poner por escrito todos los conceptos teóricos estudiados sobre TCP/IP, Sockets, protocolos, lenguajes de programación, patrones de diseño, etc., no aportaría nada que no esté ya documentado y no cabría en una memoria de trabajo fin de máster. Este documento es una esencia concentrada, que no completa, del trabajo realizado. Se ha prestado atención en elaborar una lista de referencias (algunas son meramente informativas) para que el lector que guste sepa dónde encontrar más información sobre RFCs manejadas (unas más, otras menos), textos y herramientas que están, directa o indirectamente, relacionadas con lo que se va describir a continuación.

La mejor y única manera de realmente entender lo que aquí se ha hecho es visitar los proyectos en GitHub [ERNESTO81], compilarlos, ejecutarlos, y leer el código, porque por mucho que aquí se escriban párrafos y párrafos de texto, la verdadera memoria de este trabajo está escrita en C++, Perl y XML, y sus diagramas, tablas e ilustraciones son trazas de analizadores de protocolos, ficheros de trazas, configuraciones de cortafuegos, sistemas operativos (con las tareas de configuración y administración, especialmente FreeBSD, tal vez porque es la plataforma en la me he encontrado menos cómodo) y multitud de horas de diversión delante de un teclado.

El código es la documentación, calculando de forma aproximada hay más de 500 páginas de código.

2.2 Fases

El punto de partida con el que inicio este trabajo es una aplicación que escribí en 2010 para practicar con TCP/IP y con lenguajes de programación. Esa aplicación la usé en la asignatura de TARO para presentar un artículo y practicar con la herramienta VNX [VNX].

El resultado de este trabajo son tres librerías estáticas escritas en C++ [NETLAYER], [NETCRAFT], [NETTOOLS], una aplicación de ejemplo [NETEXAMPLE] sobre cómo usar las librerías y la aplicación NetRat [NETRAT] (que significa Network Remote Access Trojan, sí, Trojan) completamente reescrita de forma coherente y limpia. Todo esto está publicado en diferentes repositorios en GitHub [GITHUB] de forma pública en [ERNESTO81], véase ilustración 2:

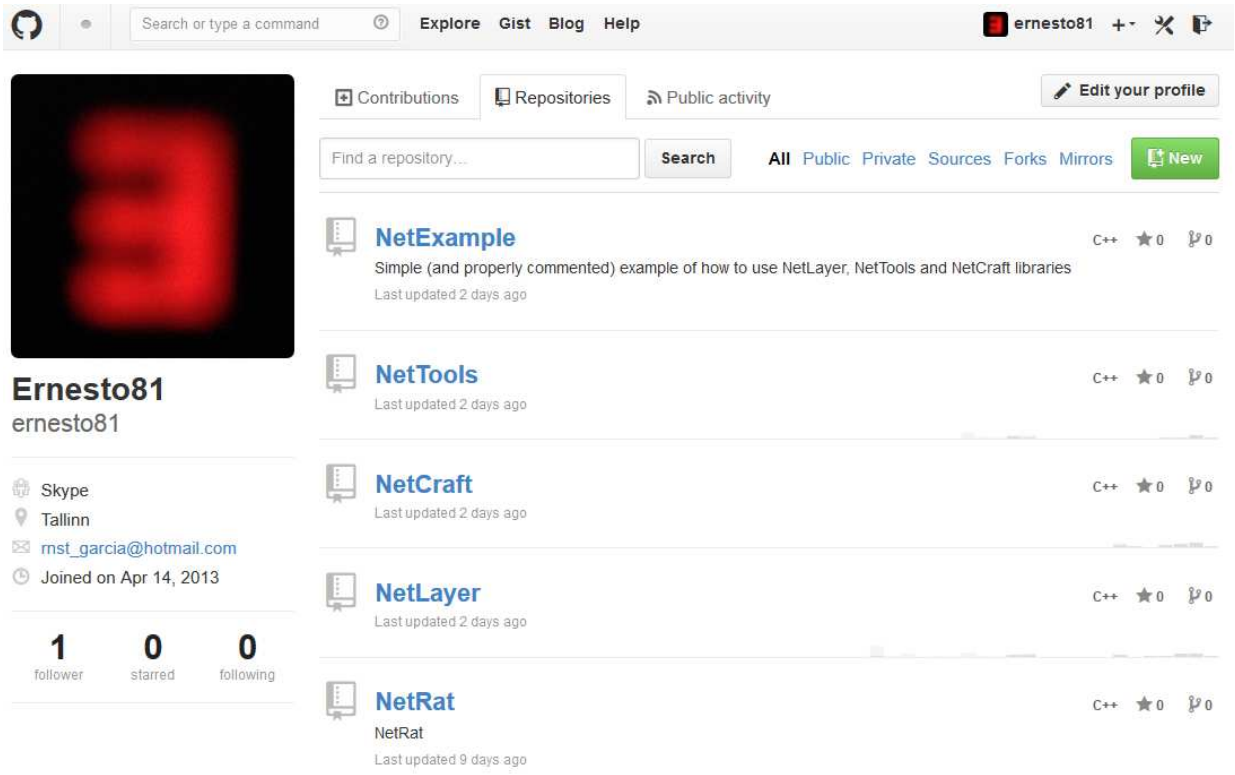


Ilustración 2: https://github.com/ernesto81 [ERNESTO81]

Las etapas que se han seguido son:

1. Estudio a nivel teórico de IPv6 continuando lo aprendido en TARO.
2. Análisis sobre cómo migrar una aplicación, en concreto mi creación NetRat [NETRAT], de IPv4 a IPv6. Cuando se comenzó a trabajar estaba mal escrita, por lo que se ha aprovechado para reescribirla casi desde cero de manera correcta. La aplicación NetRat es independiente de la plataforma funcionando correctamente para Linux, Windows y FreeBSD.
3. Implementación de tres librerías estáticas (NetLayer [NETLAYER], NetCraft [NETCRAFT], NetTools [NETTOOLS]) para encapsular de forma independiente distintas funcionalidades. Las librerías son independientes de la plataforma funcionando correctamente en Linux, Windows y FreeBSD.
   a. NetLayer: es un envoltorio sobre la interfaz Berkeley Socket que ofrece una interfaz homogénea para todas las llamadas de sistema, haciendo especial énfasis en los conceptos de encapsulación y flujo de datos entre clases. Se han usado patrones de diseño tomados de [GOF] como la cadena de responsabilidad, y se han diseñado otros, que si actualmente existen no soy consciente (combinando polimorfismo y platillas).
   b. NetCraft: es una librería que permite ensamblar de forma “artesanal” datagramas IPv4 e IPv6, y mensajes ICMPv4 e ICMPv6.
   c. NetTools: es una librería miscelánea (logger para depuración, y gestión inteligente threads, de forma individual o mediante pools).
4. Migración de IPv4 a IPv6. Inicialmente se estimó que para migrar NetRat de IPv4 a IPv6 se necesitaría unos dos 2 meses de trabajo. La realidad fue que se necesitaron 4 meses de muy duro trabajo. Estos cuatro meses incluyen la escritura de las tres librerías y el *refactoring* del código.
5. Aplicación de SCTP [RFC4960]. Una vez que la aplicación de partida es correcta y las librerías son operativas, se incluye el protocolo SCTP, lo cual resultó ser muy fácil gracias a la modularidad del código.

6. Aplicación de MPTCP [MPTCP]. Se habilita en el kernel MPTCP, se diseña un escenario y se prueba la funcionalidad junto con IPv4, IPv6 y dual-stack.
7. Creación de entornos virtualizados mediante VNX [VNX] para probar la aplicación.
8. Depuración y resolución de errores (mediante GDB, Valgrind, Visual Studio).
9. Todo esto se hizo empleando la herramienta de control de versiones Git [GIT]. Una vez que todo estaba mediamente "presentable" se publicó el código de forma abierta en GitHub [GITHUB] [ERNESTO81].
10. Auditoría legal sobre el código para verificar que no ha violado ninguna norma interna de Microsoft. El resultado de la auditoría fue favorable.
11. Redacción de la memoria.

El primer paso ha sido comprender el porqué de la necesidad de migrar de IPv6 a IPv4 y el problema técnico asociado, siendo imprescindible para ello un conocimiento profundo de IPv4. Una vez asimilados los fundamentos teóricos de los protocolos IPv4 e IPv6 el siguiente nivel es transporte (TCP, UDP, SCTP y MPTCP). El enfoque que se la ha dado al trabajo es eminentemente práctico, cuyo resultado ha sido una aplicación cliente-servidor, tres librerías, otra aplicación cliente-servidor de ejemplo (tutorial sobre cómo usar las librerías) y entornos complejos virtualizados [VNX] para verificar y validar el producto y los resultados. Poner en práctica lo anterior requiere dominar la API[19] Berkeley Socket [UNIX-1]. El software desarrollado se encuentra alojado en los servidores de GitHub usando su sistema de control de versiones y siendo accesible de manera pública, ya que para su publicación inicialmente se eligió la licencia GPL. Como resultado de la auditoría que Microsoft realizó al código de este trabajo y a sugerencia del abogado del departamento legal, finalmente se optó por doble licencia: MIT y GPL.

El ritmo de trabajo seguido a la hora de escribir la aplicación NetRat se muestra en la ilustración 3. La estructura de directorios y ficheros de las tres librarías, así como una de las aplicaciones que usa las tres librerías, se muestra en la ilustración 4.

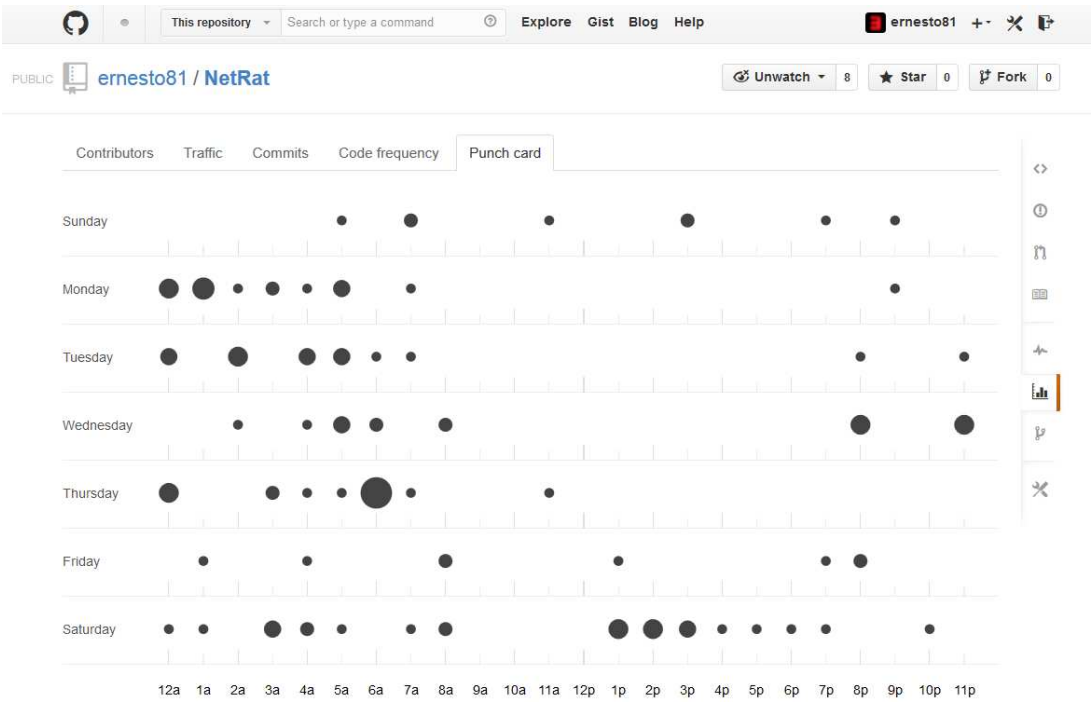


Ilustración 3: punch-card

[19] API: Application Program Interface o, en este contexto, interfaz que ofrece un protocolo para interactuar con él.

Ilustración 4: librerías y aplicaciones

2.3 Herramientas

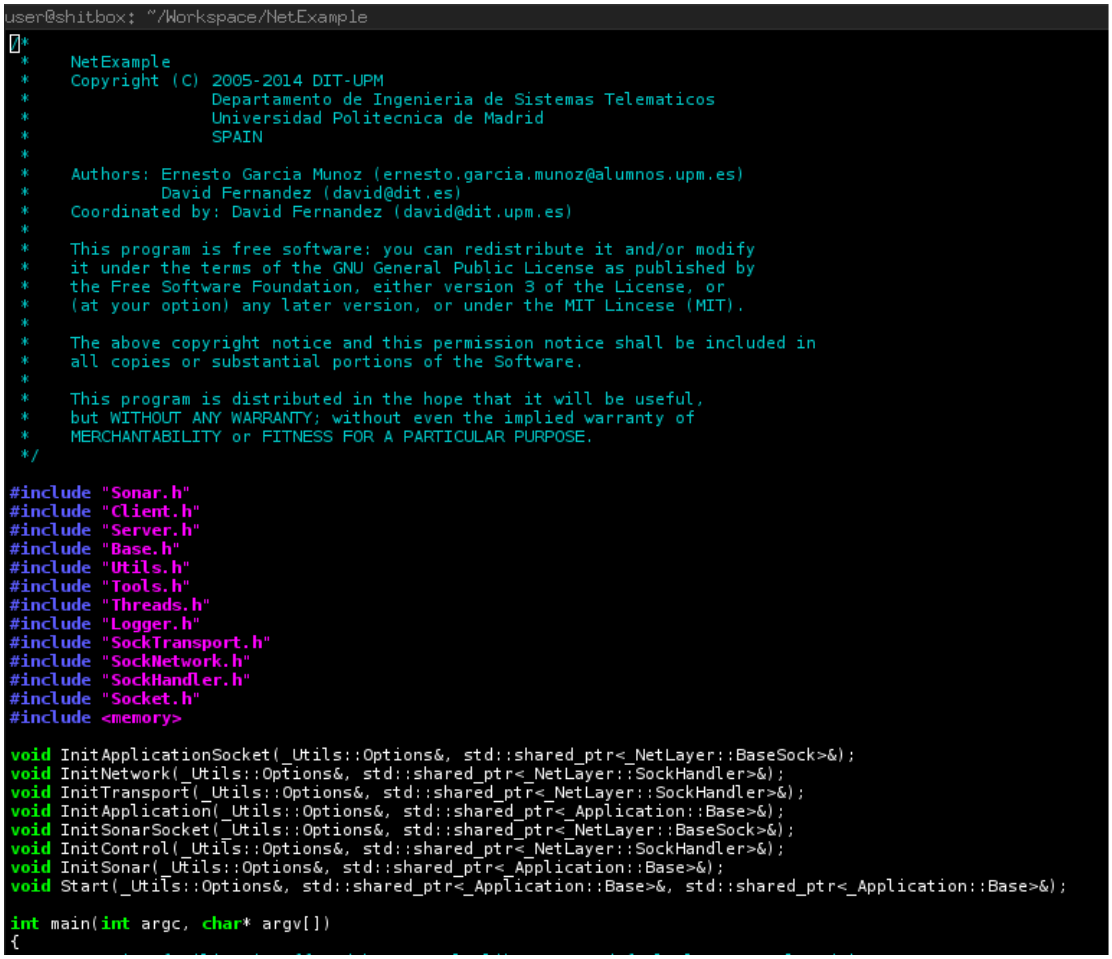
A continuación se describen las herramientas usadas en la elaboración del presente trabajo:

1. Debian Jessie [JESSIE]: el 75% del tiempo de desarrollo se ha realizado sobre la distribución Debian de Linux. En concreto Jessie (reléase de pruebas), ya que incluye de forma nativa C++11 (que fue la versión de C++ elegida debido a las novedades introducidas: Lambdas, Smart-Pointers).
2. FreeBSD 9.6 y 10 [FREEBSD]: la implementación para TCP/IP se escribió en su día para BSD, y su API se llama Berkeley socket, por ello es obligado experimentar (y familiarizarse) con la última versión disponible de FreeBSD cuando se empezó a trabajar. Un 20% del tiempo se ha invertido en esta plataforma.
3. Debian Wheezy [WHEEZY]: las máquinas virtuales del entorno de pruebas sobre las que se ejecuta la aplicación son Linux Debian (reléase Wheezy estable con MPTCP habilitado).
4. Git [GIT]: sistema de control de versiones que se ha usado para almacenar y versionar todo el código.
5. GitHub [GITHUB]: Servicio Web para acceder al código Git a través del navegador y además poder compartirlo públicamente. La ilustración 5 muestra, a modo de ejemplo, la interfaz Git mediante línea de comandos (en concreto para pedir información sobre la actividad en un determinado proyecto).
6. Vim [VIM]: mi editor favorito, sencillo y fácil de usar.
7. Wireshark [WIRESHARK]: analizador de tráfico.

```
user@shitbox: ~/Workspace/NetExample
user@shitbox:~/Workspace/NetExample$ git log
commit 3ef2b0b2bedffa356993689b6f72e40f54ba5b5b
Author: ernesto81 <rnst_garcia@hotmail.com>
Date:   Sun Apr 27 10:23:21 2014 +0300

    .

commit bec41e4248ea52c7adfa2579ebb2aef5da52bd90
Merge: 1b6f961 d93742a
Author: ernesto81 <rnst_garcia@hotmail.com>
Date:   Sat Apr 26 20:51:40 2014 +0300

    Merge branch 'master' of https://github.com/ernesto81/NetExample

commit 1b6f961841f6e98297ffc11d8d4d5a9ded5e6a31
Author: ernesto81 <rnst_garcia@hotmail.com>
Date:   Sat Apr 26 20:51:19 2014 +0300

    fuck it

commit d93742a17467328e9badc7893034aa43b58804ea
Author: Ernesto81 <rnst_garcia@hotmail.com>
Date:   Sat Apr 26 20:41:07 2014 +0300

    Update Readme.txt

commit f4b5b74e23dc6b3da551e751fd80f0ed9d24d5e1
Author: ernesto81 <rnst_garcia@hotmail.com>
Date:   Sat Apr 26 20:32:11 2014 +0300

    .

commit e66e40b5699719388362223e74c2623d164444d0
Author: ernesto81 <rnst_garcia@hotmail.com>
Date:   Sat Apr 26 20:14:31 2014 +0300

    NetExample: how-to use NetLayer, NetCraft and NetTools, code well documented
user@shitbox:~/Workspace/NetExample$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .obj/

nothing added to commit but untracked files present (use "git add" to track)
user@shitbox:~/Workspace/NetExample$ git branch -av
* master                3ef2b0b .
  remotes/origin/master 3ef2b0b .
user@shitbox:~/Workspace/NetExample$
```

Ilustración 5: Git

8. Tcpdump [TCPDUMP]: otro analizador de tráfico
9. Windows 8.1 [WIN8.1]: sistema operativo sobre el que se ha trabajado el 5% restante para garantizar que todo el código producido era compatible entre las principales familias de sistemas operativos.
10. Visual Studio [VS]: edición y depuración en entorno Windows.
11. Message Analyzer [MANALYZER]: analizar de tráfico de Microsoft.
12. VirtualBox [VBOX]: otra herramienta de virtualización.
13. C++ [CPP]: la ilustración 6 muestra un fragmento de código en este lenguaje.
14. Perl [CAMEL]: y la ilustración 7 otro fragmento de código, pero esta vez en Perl.
15. Patrones de diseño [GOF]: patrones de diseño de *"la pandilla de los cuatro"*.
16. VNX [VNX]: Virtual Network over LinuX es la herramienta de virtualización empleada para validar los resultados y desarrollar varios entornos relativamente complejos. El 90% del trabajo realizado sobre entorno virtualizado ha sido usando VNX.
17. JUNOS [JUNOS]: sistema operativo desarrollado por Juniper Networks sobre FreeBSD para sus productos encaminadores. En concreto se ha usado virtualizado, lo que en el argot se conoce como *aceituna* u *olive* en inglés [OLIVE].



```
user@shitbox: ~/Workspace/NetExample
/*
 *    NetExample
 *    Copyright (C) 2005-2014 DIT-UPM
 *                  Departamento de Ingenieria de Sistemas Telematicos
 *                  Universidad Politecnica de Madrid
 *                  SPAIN
 *
 *    Authors: Ernesto Garcia Munoz (ernesto.garcia.munoz@alumnos.upm.es)
 *             David Fernandez (david@dit.es)
 *    Coordinated by: David Fernandez (david@dit.upm.es)
 *
 *    This program is free software: you can redistribute it and/or modify
 *    it under the terms of the GNU General Public License as published by
 *    the Free Software Foundation, either version 3 of the License, or
 *    (at your option) any later version, or under the MIT Lincese (MIT).
 *
 *    The above copyright notice and this permission notice shall be included in
 *    all copies or substantial portions of the Software.
 *
 *    This program is distributed in the hope that it will be useful,
 *    but WITHOUT ANY WARRANTY; without even the implied warranty of
 *    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 */

#include "Sonar.h"
#include "Client.h"
#include "Server.h"
#include "Base.h"
#include "Utils.h"
#include "Tools.h"
#include "Threads.h"
#include "Logger.h"
#include "SockTransport.h"
#include "SockNetwork.h"
#include "SockHandler.h"
#include "Socket.h"
#include <memory>

void InitApplicationSocket(_Utils::Options&, std::shared_ptr<_NetLayer::BaseSock>&);
void InitNetwork(_Utils::Options&, std::shared_ptr<_NetLayer::SockHandler>&);
void InitTransport(_Utils::Options&, std::shared_ptr<_NetLayer::SockHandler>&);
void InitApplication(_Utils::Options&, std::shared_ptr<_Application::Base>&);
void InitSonarSocket(_Utils::Options&, std::shared_ptr<_NetLayer::BaseSock>&);
void InitControl(_Utils::Options&, std::shared_ptr<_NetLayer::SockHandler>&);
void InitSonar(_Utils::Options&, std::shared_ptr<_Application::Base>&);
void Start(_Utils::Options&, std::shared_ptr<_Application::Base>&, std::shared_ptr<_Application::Base>&);

int main(int argc, char* argv[])
{
```

Ilustración 6: C++ std11: lenguaje de programación principal

```
user@shitbox: ~/Workspace/NetRat/NetRatClient
#!/usr/bin/perl
#
#    NetRat
#    Copyright (C) 2014 rnst_garcia@hotmail.com
#
#    This program is free software: you can redistribute it and/or modify
#    it under the terms of the GNU General Public License as published by
#    the Free Software Foundation, either version 3 of the License, or
#    (at your option) any later version, or under the MIT Lincese (MIT).
#
#    The above copyright notice and this permission notice shall be included in
#    all copies or substantial portions of the Software.
#
#    This program is distributed in the hope that it will be useful,
#    but WITHOUT ANY WARRANTY; without even the implied warranty of
#    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
#

use strict;
use warnings;
use NetRatConf;
use Utils;
use SockOps;
use Data::Dumper;
use POSIX qw(strftime);

my $config = NetRatConf->instance();
$config->NetRatConf();
print STDOUT strftime("### %Y/%m/%e %H:%M:%S", localtime), " Log : Parameters : \n", Dumper($config) if $config->verb();
my $util = Utils->new();
eval {$util->Stun();}; die "TRAP: $@\n" if $@;
eval {$util->PortScanner();}; die "TRAP: $@\n" if $@;
eval {$util->RampUp();}; die "TRAP: $@\n" if $@;
eval {$util->Connect();}; die "TRAP: $@\n" if $@;
1;
```

Ilustración 7: lenguaje de programación para la parte cliente de NetRat.

```
---------------------------------------------------------------------------
Virtual Networks over LinuX (VNX) -- http://www.dit.upm.es/vnx - vnx@dit.upm.es
Version: 2.0b.3243 (built on 03/02/2014_23:58)
---------------------------------------------------------------------------
```

Ilustración 8: Virtual Networks over LinuX

3. Protocolos

3.1 El nivel de red

El protocolo de nivel de red, IP, es un protocolo no orientado a conexión y de tipo *"best-effort"*. Best-effort significa que IP no realiza ningún tipo de gestión de errores ni garantiza que el datagrama no sea descartado durante el trayecto desde el origen al destino [TCP/IP-1-12]. No orientado a conexión significa que IP no mantiene información sobre el flujo de datagramas.

En este trabajo se han usado IPv4 e IPv6, IPv4 como protocolo de partida e IPv6 como destino. También se han usado intensivamente ICMPv4 e ICMPv6 por dos motivos: primero para ejercitarse con los sockets de tipo SOCK_RAW (los más oscuros de usar), y segundo porque IPv6 se apoya fuertemente sobre ICMPv6 (tanto que IPv6 no puede funcionar sin ICMPv6). El resultado práctico han sido dos librerías (NetLayer [NETLAYER] y NetCraft [NETCRAFT]) y dos aplicaciones que pueden funcionar con IPv4, IPv6 o dual-stack. En el apartado en el que se expone el código escrito para este trabajo se profundizará con mayor detalle en cada uno de los componentes software desarrollados, pero a modo de introducción cabe comentar el objetivo de dos de las librerías NetLayer y NetCraft:

1. El objetivo de NetLayer es proporcionar una interfaz simple construida sobre la API de Sockets que permita al usuario desarrollar aplicaciones de comunicaciones que hagan uso de protocolos de nivel de red (IPv4, IPv6, ICMPv4 e ICMPv6) y protocolos de nivel de transporte (TCP, UDP, SCTP[20], MPTCP).
2. El objetivo de NetCraft es proporcionar al usuario una herramienta que permita, de la manera menos complicada posible, ensamblar datagramas IPv4 e IPv6 o mensajes ICMPv4 e ICMPv6 al trabajar con sockets de tipo SOCK_RAW (cabecera, cuerpo, cálculo de checksum[21]).

Los objetivos de IPv6 [RFC2460] son: "solucionar los problemas de direccionamiento y encaminamiento que limitan el crecimiento de Internet, pero además: mejora de prestaciones, soporte de trafico multimedia, seguridad, autoconfiguración, encaminamiento, etc.", como se estudió en la asignatura TARO.

3.1.1 Transición de IPv4 a IPv6 en la teoría

Los proveedores de Internet no pueden usar más direcciones IPv4 de las que tienen asignadas por RIPE, y ya se han agotado las que todavía estaban sin asignar. IPv6, la solución a largo plazo al agotamiento de direcciones IPv4, no se despliega porque no ha existido voluntad de atajar el problema de raíz y se ha optado por soluciones a corto plazo (malas desde un punto de vista técnico, pero tal vez rentables desde el económico). Ni siquiera las recomendaciones del IETF para comenzar la transición, de manera gradual y poco traumática, como la dual-stack, se pusieron en marcha antes de la extinción de las direcciones IPv4. A estas alturas la dual stack por sí misma ya no es viable porque no hay direcciones IPv4 suficientes para que funcione.

Dual-stack [UNIX-1] consiste en tener tanto IPv4 como IPv6 de forma nativa en el sistema operativo y es una recomendación del IETF para realizar la migración de IPv4 a IPv6 de forma poco traumática. Durante la transición de IPv4 a IPv6 los servicios y aplicaciones deben ser capaces de trabajar y comunicarse con otros servicios y aplicaciones usando tanto IPv4 como IPv6. En el extremo de la red, esto es, en un equipo conectado a Internet, en lugar de tener la misma aplicación por

[20] SCPT ofrece dos tipos de funcionamiento (*one-to-one* y *one-to-many*). El objetivo base de este trabajo es el estudio de protocolos, pero también el desarrollo de y librerías aplicaciones multiplataforma (FreeBSD, Windows, Linux). El hecho de que SCTP es un nuevo –y poco usado- protocolo hace que su ubicuidad sea limitada. Con el objetivo de mantener la independencia multiplataforma este trabajo se ha centrado en SCTP tipo *one-to-one*, ya que el *one-to-many* no está soportado en las plataformas usadas de forma nativa y además la funciones usadas no están completamente estandarizadas (y esto ensuciaba considerablemente el código de la librería NetLayer introduciendo muchos casos especiales).
[21] Checksum: termino anglosajón que se refiere a la suma de verificación de integridad del mensaje.

duplicado (una instancia trabajando en IPv4 y otra en IPv6), es mejor una sola trabajando en ambas versiones simultáneamente gracias al uso del dual-stack [UNIX-1]. Por ejemplo, un servidor que use dual-stack pero que esté escrito únicamente para IPv6 puede dar servicio tanto a clientes IPv6 como a clientes IPv4 gracias a que existe una dirección especial IPv6 ::ffff:0:0/96 (*IPv4 mapped address*) y al campo "*type*" en la cabecera ethernet que diferencia tramas que transportan un datagrama IPv4 de tramas con datagramas IPv6. Quiere esto decir que a la hora de desarrollar nuevas aplicaciones la mejor manera de asegura compatibilidad IPv4 e IPv6 es escribiéndolas completamente en IPv6 (siempre que se disponga de dual-stack) [UNIX-1]. El caso de aplicaciones que fueron escritas en IPv4 es diferente, ya que requieren ser adaptadas específicamente para usar IPv6 debido a las carencias de la API Socket.

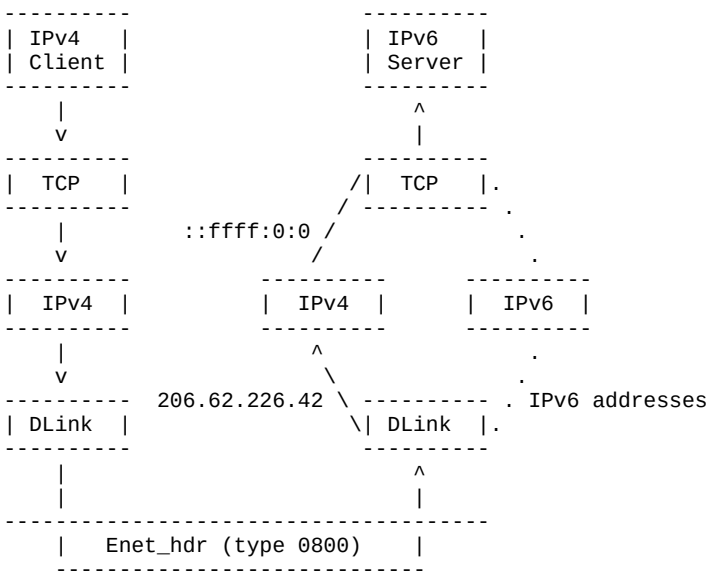


Ilustración 9: Cliente IPv4 comunicándose con Servidor IPv6 gracias a dual-stack ([UNIX-1])

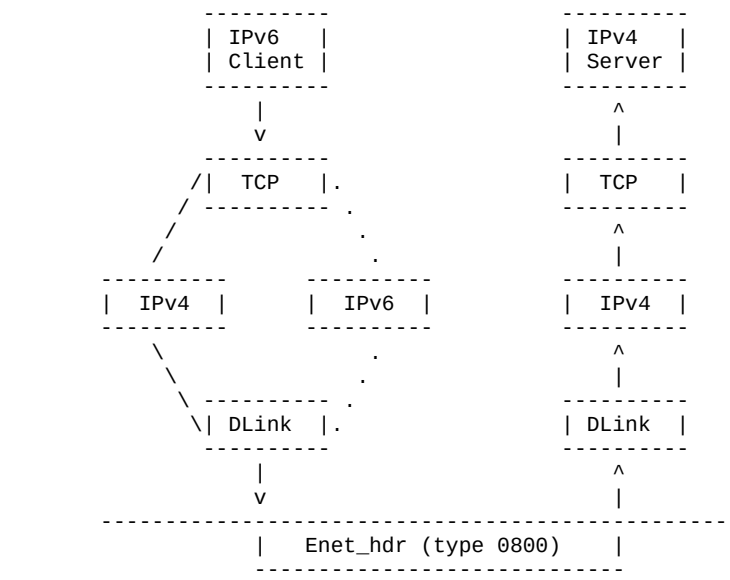


Ilustración 10: Cliente IPv6 comunicándose con Servidor IPv4 gracias a dual-stack ([UNIX-1])

Más información sobre interoperabilidad de aplicaciones escritas en IPv4 y/o IPv6 en [UNIX-1]. La conclusión es que la mejor manera de escribir aplicaciones compatibles tanto con IPv4 como con IPv6 cuando se dispone de dual-stack es escribiéndolas únicamente para IPv6.

3.1.2 Transición de IPv4 a IPv6 en la práctica

Dual-stack, como ya se ha comentado, es una recomendación del IETF para que en el sistema operativo de un dispositivo convivan tanto IPv4 como IPv6. Durante la transición de IPv4 a IPv6 las redes deberían soportar ambos protocolos, ya que es imposible migrar todos los servicios sin un paso intermedio. El uso de un nivel de red u otro estaría determinado por la red en la que se encuentra el dispositivo cliente, o por la red en la que se encuentra el servidor del servicio pretendido. El problema es que ya se han agotado las direcciones de IPv4 públicas, y dual-stack es un mecanismo para facilitar la transición, no la solución en sí al agotamiento de direcciones IPv4.

Idealmente la transición sería de IPv4 a IPv6 pasando por una fase intermedia IPv4/IPv6, pero esto no se ha adoptado a tiempo por los operadores de red y ya no quedan suficientes direcciones de red para ponerlo en marcha. Es necesaria otra solución en la que no sea necesaria la asignación de direcciones IPv4 públicas. En teoría y en el escenario propuesto por dual-stack, un cliente podría elegir usar IPv4 o IPv6 ayudado de servicios externos, como puede ser DNS (usado como conmutador). DNS, además de resolver direcciones IP, realizaría funciones de conmutación entre IPv4 e IPv6. La línea de separación entre qué hace quién empieza a difuminarse. Con dual-stack y DNS(64)[22] no es suficiente, porque no hay más direcciones IPv4 públicas y, ni dual-stack ni DNS(64) resuelven el problema.

No quedan direcciones IPv4, dual-stack es necesario para migrar de forma gradual y DNS puede ejercer el rol de conmutador entre IPv4 e IPv6. ¿Qué más es necesario?: CG-NAT[23]. Como no es posible seguir asignado direcciones públicas a los clientes es necesario añadir un nivel más a la jerarquía de equipos NAT. Mediante CG-NAT una IPv4 pública es compartida simultáneamente por varios usuarios finales, con sus redes domésticas correspondientes cada uno. El uso de un CG-NAT hace que se tenga doble NAT, lo que rompe el funcionamiento de algunas aplicaciones, un poco más. Esta solución gusta mucho a operadores de red, ya que se podrá simultanear CG-NAT con CG-NAT64 lo cual permite reutilizar equipamiento, para recauchutar un poco más la red (y ahorrar unos eurillos). En mi humilde opinión, soluciones como ésta van a hacer que Internet se colapse porque el nivel de complejidad que está alcanzando es antagonista de la semilla original y convierte a la red en algo totalmente fuera de control e inmanejable. Esta complejidad asociada traerá agujeros y brechas de seguridad, que aunque difíciles de detectar, son un potencial peligro para Internet. Es curioso que mientras toda la sociedad se está convirtiendo más Internet-dependiente, Internet por su parte se está convirtiendo en una cosa-monstruo inmanejable y completamente fuera de control.

De las múltiples combinaciones de posibilidades, una plausible es que los equipos de usuario solo funcionen con IPv6 pero el núcleo de la red y/o los servicios, sólo estén disponibles en IPv4, alternando el uso de NAT64 y DNS64 para permitir que dispositivos IPv6 tengan acceso a servicios IPv4. Y aquí es donde operadores de telecomunicaciones podrían reutilizar el CG-NAT para reconvertirlo en NAT64. Idealmente, los ISP[24] deberían utilizar la migración de IPv4 a IPv6 para simplificar y actualizar los equipos y arquitectura de sus redes. La simplificación vendría en la forma de una jerarquía de routers aplanada. En la migración gradual que los ISP harán se priorizaría redes que usan direcciones IP públicas de manera intensiva. Los cambios en la jerarquía de routers y NATs no es el único cambio, por ejemplo también sería necesario añadir registros AAAA al servicio DNS.

[22] DNS(64): servicio DNS con capacidad de traducción de dirección IPv4 e IPv6.
23 CG-NAT: Carrier Grade NAT es otro nivel NAT lo cual introduce jerarquía de dispositivos NAT en la red.
[24] ISP: Internet Service Provider o proveedor de servicio de Internet.

3.2 El nivel de transporte

Según [TANENBAUM], el nivel de transporte es el más importante en la torre de protocolos (según [TCP/IP-1-94] es el nivel de red). El objetivo de esta sección no es entrar en discusión sobre qué autor tiene la razón, si no describir el nivel de transporte y analizar las posibilidades que la migración actual en Internet puede ofrecer. El servicio que ofrece el nivel de transporte es el flujo de datos entre aplicaciones en diferentes máquinas *independientemente* del nivel de red que se use [TANENBAUM] [TCP/IP-1-94]. En un mundo ideal Internet estaría formado por clientes, servidores y enrutadores. Idealmente el nivel de transporte únicamente es necesario en los extremos de la red (en los clientes y en los servidores), mientras que los dispositivos que enrutan el tráfico en Internet solo necesitan hasta el nivel de red para permitir que el tráfico fluya entre los extremos. En la práctica esto no es así, ya que cada vez proliferan más *middleboxes*[25] que complican la conexión extremo a extremo.

Que el servicio ofrecido por el nivel de transporte es *independiente* del nivel de red, es una afirmación ingenua e incorrecta, como también lo es el decir que sólo está implementado en los extremos de la red. El nivel de transporte debería abstraer completamente del nivel de red, esto es, idealmente, los cambios que ocurran en el nivel de red son transparentes al nivel aplicación gracias al nivel de transporte. Si esto fuera cierto, gracias a que la implementación de la torre de protocolos sigue el diseño en capas del modelo de referencia TCP/IP, la migración de IPv4 a IPv6 pasaría desapercibida para las aplicaciones. En el *"mundo-ideal-de-arcoíris"* en el que viven los *ponis*, Internet IPv4 se podría apagar el domingo por la noche y el lunes por la mañana encenderlo con Internet IPv6, y ningún poni se habría inmutado. Esto está lejos de la realidad gracias en parte al interfaz de programación de TCP/IP: *The Socket API*[26].La sustentación de la afirmación anterior es muy fácil, no hay más que remitirse a la referencia [TCP/IP-2] y echar un vistazo rápido: si uno no tiene nada mejor que hacer en los próximos meses (tal vez año), la lectura del código que implementa TCP/IP puede ser un pasatiempo de gran complejidad y divertido. En la API Socket se entremezclan elementos de diferentes capas no existiendo ninguna encapsulación. No existe separación entre nivel de red o nivel de transporte. Por ejemplo, se entremezclan elementos del nivel de red (tipo de familia de direcciones, direcciones de nivel de red, etc.) con elementos del nivel de transporte (protocolo, número de puerto, etc.). En el mundo real, la migración del protocolo de una de las capas tiene impacto en todas y esto no es ideal, ni lo que se vendió sobre el papel. En los extremos de la red no es suficiente con actualizar el kernel del sistema operativo para actualizarse a IPv6. Las aplicaciones también tienen que actualizarse de manera profunda. Esta *no-idealidad* es la que justifica este trabajo: como la migración del protocolo IP va a afectar profundamente a todos los elementos de la red, y los protocolos de nivel de transporte siguen siendo los mismos que cuando se desplegó Internet, es una buena oportunidad para tener en cuenta mejoras, evoluciones y nuevos protocolos que puedan incrementar las prestaciones en Internet. Los protocolos de transporte de nueva generación que se van a usar en este trabajo han sido SCTP[27] y MPTCP[28]. Por otro lado, debido a la carencia de encapsulación y estructuración en capas en la API Sockets se escribió la librería NetLayer [NETLAYER] como *wrapper* sobre la API Socket, con objetivo de ofrecer una estructuración, similar a la del modelo de referencia TCP/IP en capas, que simplifique la tarea de escribir aplicaciones de comunicaciones en el que los cambios en el nivel de red afecten lo mínimo posible al nivel de transporte evitando tener que reescribir el código.

[25] Middlebox es un dispositivo que se añade a la red y que implementa toda la torre de protocolos TCP/IP. Ejemplos: NATs, analizadores de tráfico en tiempo real. Algunos de estas máquinas tienen objetivos oscuros como el análisis y filtrado del tráfico de clientes.
[26] Cierto es que el código que implementa TCP/IP ha sido usado durante decádas y ha funcionado bien, pero no está escrito siguiendo los patrones de diseño en capas que define el modelo de referencia que implementa, lo cual es contradictorio: sobre el papel el diseño es estructurado en niveles o capas, la implementación es una maraña de código.
[27] SCTP Stream Control Transmission Protocol es un protocolo nuevo de de nivel de transporte que combina lo mejor de TCP y UDP.
[28] MPTCP Multipath TCP es una evolución continuista de TCP que incluye multi-homing con la posibilidad de usar diferentes rutas entre las diferentes partes.

3.2.1 Nuevos protocolos de nivel de transporte

Los protocolos de nivel de transporte más usados, con diferencia, en la actualidad son TCP[29] [RFC793] y UDP[30] [RFC768]. UDP es un protocolo no orientado a conexión y no confiable, mientras que TCP es orientado a conexión y confiable [UNIX-1]. TCP es un protocolo extraordinariamente complejo mientras que UDP es simple. Únicamente hay que echar un vistazo a las cabeceras para apreciar esta diferencia (ilustraciones 11 y 12). Tanto UDP como TCP se explican en detalle en [TCP/IP-1-94], [TCP/IP-1-12], [UNIX-1] [TANENBAUM] y en sus correspondientes RFCs [RFC768] [RFC793].

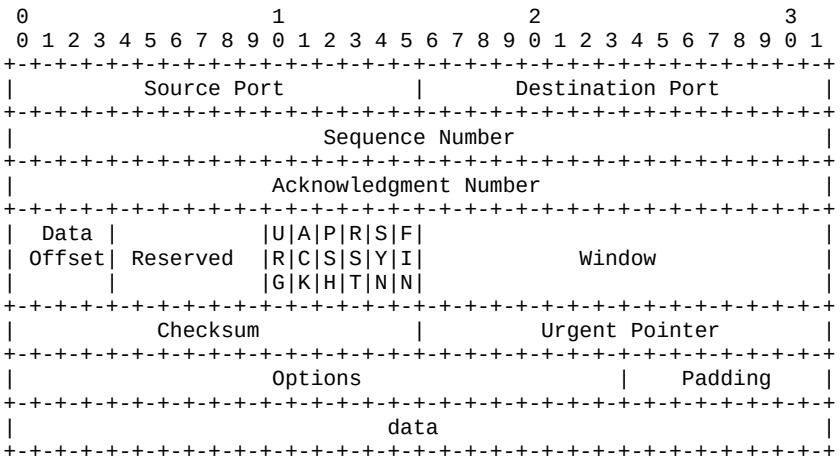


Ilustración 11: cabecera de un segmento TCP.

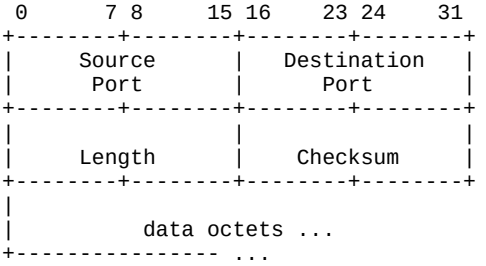


Ilustración 12: cabecera de un datagrama UDP.

Cuando estaba pensando el tipo de trabajo fin de Máster que iba a realizar, me decidí por protocolos de nivel de transporte, principalmente porque la programación de aplicaciones de comunicaciones es lo que más me gusta. Si hay un buen momento para *tocar* protocolos de nivel de transporte es justo ahora que está teniendo lugar la migración de IPv4 a IPv6. El objetivo último de este trabajo es diseñar librerías que permitan escribir aplicaciones de comunicaciones de manera rápida y ordenada siguiendo patrones de diseño y que permitan incluir o excluir protocolos, tanto de nivel de red como de transporte, de manera eficiente. Para lograr el objetivo final el punto de partida de este trabajo fue una aplicación (NetRat), que escribí a finales de 2010, para practicar con *Networking*. De forma muy resumida es una aplicación cliente servidor que hace uso de TCP/IP, en concreto, la versión de partida usaba IPv4 para transportar datagramas UDP (cuando trabaja en modo UDP) o segmentos TCP (cuando trabaja en modo TCP). Como el objetivo de este trabajo es el estudio a nivel práctico de nuevos protocolos de nivel de transporte, lo primero que se hizo fue seleccionar los candidatos: SCTP y MPTCP. Una vez que se sabía los protocolos de transporte que se iban a añadir se realizó un estudio sobre cómo añadirlos de la forma más limpia posible. Además, se observó que no es coherente usar unos protocolos de nivel

[29] TCP Transmission Control Protocol
[30] UDP User Datagram Protocol

de transporte de última generación sobre un nivel de red obsoleto, por lo que aumentó el ámbito de trabajo a IPv6. Así pues, los requisitos básicos para el *producto* final fueron:

1. Migrar aplicación de IPv4 a IPv6 (pasando por *dual-stack*). La aplicación es necesario que trabajen IPv4 (para máquinas sin IPv6), IPv6 o en modo *dual-stack*.
2. Añadir nuevos protocolos de nivel de transporte. Partiendo de que la aplicación podía trabajar con TCP y/o UDP, añadirle MPTCP y SCTP. Esta adición además tiene que ser lo más sencilla posible.

Además todo esto había que hacerlo de manera ordenada, por lo que se decidió escribir tres librerías:

1. *NetLayer:* es un *wrapper* sobre la API Socket que usa una cadena de responsabilidad para mimetizar el patrón de diseño en capas del modelo de referencia TCP/IP
2. *NetCraft:* elaboración manual de datagramas IP y mensajes ICMP (porque trabajar con el nivel de red directamente es algo más complejo).
3. *NetTools:* librería general que ayuda a abstraer las diferencias entre diferentes sistemas operativos en temas de threading.

El resultado ha sido que ahora NetRat es capaz de trabajar con IPv4, con IPv6, o con IPv4 e IPv6 simultáneamente. Además, añadirle la capacidad de operar sobre nuevos protocolos de nivel de transporte es muy sencillo gracias a la librería NetLayer. Añadir SCTP fue casi como cortar y pegar y cambiar una palabra. Como MPTCP es una evolución continuista de TCP, no hubo que cambiar nada a nivel de aplicación (únicamente activar la funcionalidad en el kernel de Linux y configurar la tabla de rutas, y elaborar escenarios con la herramienta VNX [VNX]). En la sección correspondiente al código (y en los repositorios públicos en GitHub [ERNESTO81]) de este trabajo se presentaran las aplicaciones y librerías elaboradas (que son el núcleo de este trabajo), pero antes se procede a presentar a los protocolos MPTCP y SCTP.

3.3 Ejemplo: dual-stack y MPTCP

Se va a presentar a continuación un ejemplo que involucra IPv4, IPv6, dual-stack y MPTCP (aunque todavía no se ha hablado del nivel de transporte). Aunque se intenta que los conceptos y ejemplos de la memoria sigan un proceso deductivo, este apartado es un inciso inductivo en el que se observará un ejemplo final y relativamente complejo que involucra a todos los componentes (aplicaciones, protocolos, escenarios virtualizados, configuraciones). El escenario propuesto en el ejemplo usa una de las aplicaciones y todas las librerías espacialmente desarrolladas para el trabajo fin de master. Está compuesto de un cliente con MPTCP, un servidor también con MPTCP y un router (JUNOS Olive[31]). El servidor tiene varios interfaces (multi-homing) y la tabla de rutas permite diferentes alternativas para llegar desde el cliente al servidor. Tanto el cliente como el servidor usan dual stack. El servidor escucha en la dirección IPv6 in6addr_any :: y el puerto (MP)TCP 55447.Aunque el servidor está escuchando en la dirección in6addr_any (::) la tupla correspondiente al socket conectado en el servidor es (7.7.7.7, 6543, 7.7.9.7, 55447, TCP), según muestra el comando “lsof -n –i”. Realmente lo que ocurre internamente es que la dirección IPv4 del cliente se mapea a la dirección especial IPv6 (::ffff:0:0/96) gracias a dual-stack.

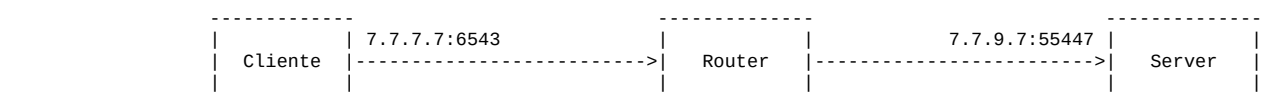


Ilustración 13: diagrama simplificado de componentes que conforman el ejemplo

[31] Olive es el nombre en clave para designar JUNOS en entorno virtualizado. JUNOS es el sistema operativo de los routers de Juniper, que está basado en FreeBSD.

Usando un analizador de protocolos en el router podemos ver en detalle la dual-stack y MPTCP en acción: en el ejemplo (ver ilustración 15 y 16) se observa cómo el cliente establece la conexión con el servidor y se intercambian algunos datos entre ambos. Lo espectacular del ejemplo no es únicamente como dual-stack y MPTPC interactúan con total naturalidad, sino cómo MPTCP es capaz de establecer múltiples conexiones, incluso por los recovecos más improbables, de manera correcta. En la traza mostrada en las ilustraciones 15 y 16 se observa cómo la conexión TCP se establece con normalidad, pero en el segundo 52 (cuando la transferencia de datos comienza) se observa cómo MPTCP entra en acción y el cliente intenta conectar con el servidor a través del interfaz con dirección IPv4 10.7.7.7 (curiosamente no es la ruta que se cabría esperar, ya que es la que utilizo para conectar con ssh[32] con las máquinas virtuales). En un primer momento el sniffer[33] se lanzó en el router, ya que lo normal sería que las múltiples conexiones se establecieran a través de éste. Lo que más me llamó la atención fue comprobar que también se establecen conexiones a través del host que alberga este entorno virtualizado (esto es posible porque está configurado como proxy[34]). Además, aunque el cliente inicia el 3-way-handshake[35] usando una dirección local IPv4, gracias al dual-stack, y la configuración de direcciones IPv6 en los interfaces, se han observado caminos que usan IPv6, como se muestra en las ilustraciones 14, 15 y 16. En la traza de la ilustración 15 se muestra cómo MPTCP es capaz de establecer múltiples caminos usando todos los interfaces y protocolos de red disponibles, e iniciando las conexiones tanto desde el cliente, lo que cabría esperar, como también desde el servidor.

La conexión inicial realizada fue 7.7.7.7:6543 -> 0::0:55447. Los caminos observados en el ejemplo (basado en segmentos TCP con flag SYN activado) se muestran a continuación en la ilustración 14, donde directo significa que es el cliente quien comienza el establecimiento de la conexión, mientras que inverso significa que ha sido el servidor quien lo ha hecho:

```
Caminos IPv4
------------
        7.7.7.7:6543     ->   7.7.9.7:55447 (directo)
        10.7.7.7:41233   ->   7.7.9.7:55447 (directo)
        10.7.9.7:55447   ->   7.7.7.7:50045 (inverso)
        10.7.7.7:46240   ->   7.7.11.7:55447 (directo)
        7.7.11.7:55447   ->   7.7.7.7:33215 (inverso)
        10.7.7.7:45841   ->   10.7.9.7:55447 (directo)
        7.7.9.7:55447    ->   10.7.7.7:40329 (inverso)
        7.7.7.7:43973    ->   10.7.9.7:55447 (directo)
Caminos IPv6
------------
        2001:db8:0:1::7:56449   ->   2001:db8:0:3::3:55447 (directo)
        2001:db8:0:1::7:59562   ->   2001:db8:0:5::3:55447 (directo)
        2001:db8:0:3::3:55447   ->   2001:db8:0:1::7:56447 (inverso)
```

Ilustración 14: múltiples caminos establecidos en el ejemplo

Aunque la mayoría de las conexiones se establecieron (se completó de forma correcta el 3-way-handshake), debido a que la cantidad de datos transferidos entre cliente y servidor no fue muy elevada, no todos los caminos se han usado. Se ha observado cierta tendencia a usar conexiones IPv6. En este sencillo ejemplo se han enviado segmentos con flag SYN activo por 11 caminos diferentes.

En la ilustración 15 se muestra en detalle la traza obtenida en el router, mientras que en la ilustración 16 se muestra la traza obtenida en la máquina host (recordar que MPTCP también fue capaz de establecer conexiones por ese camino):

[32] Ssh: secure shell.
[33] Sniffer es la palabra anglosajona para referirse a un analizador de protocolos.
[34] El host es una máquina Linux Debian (Jessie) que tiene la opción de enrutar IPv4 e IPv6 habilitada, lo que le permite ejercer como proxy.
[35] 3-way-handshake: establecimiento de conexión TCP.

Comandos ejecutados en el router del ejemplo para habilitar función sniffer y traza obtenida:

```
root@r1% cli
root@r1> monitor traffic
verbose output suppressed, use <detail> or <extensive> for full protocol decode
Address resolution is ON. Use <no-resolve> to avoid any reverse lookup delay.
Address resolution timeout is 4s.
Listening on fxp0, capture size 96 bytes

Reverse lookup for 7.7.9.7 failed (check DNS reachability).
Other reverse lookup failures will not be reported.
Use <no-resolve> to avoid reverse lookups on IP addresses.
```

```
 19:18:34.165914  In 7.7.7.7.6543   > 7.7.9.7.55447: S 4118140023:4118140023(0) win 29200 <mss 1460,sackOK,timestamp 3810471 0,nop,wscale 5,[|tcp]>
 19:18:34.167369 Out 7.7.9.7.55447  > 7.7.7.7.6543: S 1676928794:1676928794(0) ack 4118140024 win 28560 <mss 1460,sackOK,timestamp 3810058 3810471,nop,wscale 5,[|tcp]>
 19:18:34.168041  In 7.7.7.7.6543   > 7.7.9.7.55447: . ack 1 win 1825 <nop,nop,timestamp 3810471 3810058,opt-30:0081e8f8faba00[|tcp]>
 19:18:52.667848  In 7.7.7.7.6543   > 7.7.9.7.55447: P 1:7(6) ack 1 win 1825 <nop,nop,timestamp 3815096 3810058,opt-30:34010a070707,[|tcp]>
 19:18:52.669725 Out 7.7.9.7.55447  > 7.7.7.7.6543: . ack 7 win 1785 <nop,nop,timestamp 3814683 3815096,opt-30:34010a070907,[|tcp]>
 19:18:52.670913  In 10.7.7.7.41233 > 7.7.9.7.55447: S 2495658648:2495658648(0) win 29200 <mss 1460,sackOK,timestamp 3815097 0,nop,wscale 5,[|tcp]>
 19:18:52.672445 Out 10.7.9.7.55447 > 7.7.7.7.50045: S 609368363:609368363(0) ack 2196324273 win 28560 <mss 1460,sackOK,timestamp 3814684 3815097,nop,wscale 5,[|tcp]>
 19:18:52.674758  In 10.7.7.7.46240 > 7.7.11.7.55447: S 3346888714:3346888714(0) win 29200 <mss 1460,sackOK,timestamp 3815098 0,nop,wscale 5,[|tcp]>
 19:18:52.674769  In 7.7.7.7.33215  > 7.7.11.7.55447: S 7058333:7058333(0) win 29200 <mss 1460,sackOK,timestamp 3815098 0,nop,wscale 5,[|tcp]>
 19:18:52.675236 Out 10.7.9.7.55447 > 7.7.7.7.50045: . ack 1 win 3570 <nop,nop,timestamp 3814684 3815098,opt-30:360920010db826[|tcp]>
 19:18:52.676356  In 2001:db8:0:1::7.56449 > 2001:db8:0:3::3.55447: S 3239243036:3239243036(0) win 28800 <[|tcp]>
 19:18:52.676721  In 10.7.7.7.41233 > 7.7.9.7.55447: . ack 435314772 win 4563 <nop,nop,timestamp 3815098 3814684,opt-30:10001d6a613c04[|tcp]>
 19:18:52.676924 Out 7.7.11.7.55447 > 7.7.7.7.33215: S 3720748709:3720748709(0) ack 7058334 win 28560 <mss 1460,sackOK,timestamp 3814685 3815098,nop,wscale 5,[|tcp]>
 19:18:52.677777  In 10.7.7.7.46240 > 7.7.11.7.55447: . ack 3665705095 win 5475 <nop,nop,timestamp 3815099 3814685,opt-30:1000cbf81e2d24[|tcp]>
 19:18:52.678427  In 7.7.7.7.33215  > 7.7.11.7.55447: . ack 1 win 6388 <nop,nop,timestamp 3815099 3814685,opt-30:100020a735aee6[|tcp]>
 19:18:52.679427 Out 7.7.11.7.55447 > 7.7.7.7.33215: . ack 1 win 6248 <nop,nop,timestamp 3814686 3815099,opt-30:20018909ef48>
 19:18:52.680081  In 2001:db8:0:1::7.59562 > 2001:db8:0:5::3.55447: S 826285353:826285353(0) win 28800 <[|tcp]>
 19:18:52.680916 Out 2001:db8:0:3::3.55447 > 2001:db8:0:1::7.56449: S 2617781957:2617781957(0) ack 3239243037 win 28160 <[|tcp]>
 19:18:52.681838  In 2001:db8:0:1::7.56449 > 2001:db8:0:3::3.55447: . ack 1 win 7288 <[|tcp]>
 19:18:52.682049 Out 2001:db8:0:5::3.55447 > 2001:db8:0:1::7.59562: S 774127613:774127613(0) ack 826285354 win 28160 <[|tcp]>
 19:18:52.682736  In 2001:db8:0:1::7.59562 > 2001:db8:0:5::3.55447: . ack 1 win 8188 <[|tcp]>
 19:18:52.682819 Out 2001:db8:0:3::3.55447 > 2001:db8:0:1::7.56449: . ack 1 win 7128 <[|tcp]>
 19:18:52.683493 Out 2001:db8:0:5::3.55447 > 2001:db8:0:1::7.59562: . ack 1 win 8008 <[|tcp]>
 19:18:52.748229 Out 2001:db8:0:5::3.55447 > 2001:db8:0:1::7.59562: P 1:46(45) ack 1 win 8008 <[|tcp]>
 19:18:52.748720  In 2001:db8:0:1::7.59562 > 2001:db8:0:5::3.55447: . ack 46 win 8188 <[|tcp]>
 19:18:57.688940  In fe80::fd:ff:fe00:1 > 2001:db8:0:1::1: ICMP6, neighbor solicitation[|icmp6]
 19:18:57.689112 Out e80::2fd:ff:fe00:201 > fe80::fd:ff:fe00:1: ICMP6, neighbor advertisment[|icmp6]
 19:19:10.981473  In 2001:db8:0:1::7.59562 > 2001:db8:0:5::3.55447: P 1:8(7) ack 46 win 8188 <[|tcp]>
 19:19:10.985591 Out 2001:db8:0:5::3.55447 > 2001:db8:0:1::7.59562: . ack 8 win 8008 <[|tcp]>
 19:19:10.985711 Out 2001:db8:0:5::3.55447 > 2001:db8:0:1::7.59562: P 46:47(1) ack 8 win 8008 <[|tcp]>
 19:19:10.986460  In 2001:db8:0:1::7.59562 > 2001:db8:0:5::3.55447: . ack 47 win 8188 <[|tcp]>
 19:19:10.987898 Out 2001:db8:0:5::3.55447 > 2001:db8:0:1::7.59562: P 47:53(6) ack 8 win 8008 <[|tcp]>
 19:19:10.988572  In 2001:db8:0:1::7.59562 > 2001:db8:0:5::3.55447: . ack 53 win 8188 <[|tcp]>
 19:19:10.989489 Out 2001:db8:0:5::3.55447 > 2001:db8:0:1::7.59562: . 53:1461(1408) ack 8 win 8008 <[|tcp]>
 19:19:10.989610 Out 2001:db8:0:5::3.55447 > 2001:db8:0:1::7.59562: P 1461:1481(20) ack 8 win 8008 <[|tcp]>
 19:19:10.990118  In 2001:db8:0:1::7.59562 > 2001:db8:0:5::3.55447: . ack 1461 win 8279 <[|tcp]>
 19:19:10.990610  In 2001:db8:0:1::7.59562 > 2001:db8:0:5::3.55447: . ack 1481 win 8279 <[|tcp]>
 19:19:10.991863 Out 2001:db8:0:5::3.55447 > 2001:db8:0:1::7.59562: P 1481:1562(81) ack 8 win 8008 <[|tcp]>
 19:19:10.992525  In 2001:db8:0:1::7.59562 > 2001:db8:0:5::3.55447: . ack 1562 win 8279 <[|tcp]>
```

```
19:19:51.359228  In 2001:db8:0:1::7.59562 > 2001:db8:0:5::3.55447: P 8:46(38) ack 1562 win 8279 <[|tcp]>
19:19:51.360911 Out 2001:db8:0:5::3.55447 > 2001:db8:0:1::7.59562: P 1562:1563(1) ack 46 win 8008 <[|tcp]>
19:19:51.361575  In 2001:db8:0:1::7.59562 > 2001:db8:0:5::3.55447: . ack 1563 win 8279 <[|tcp]>
19:19:51.362708 Out 2001:db8:0:5::3.55447 > 2001:db8:0:1::7.59562: P 1563:1590(27) ack 46 win 8008 <[|tcp]>
19:19:51.363918  In 2001:db8:0:1::7.59562 > 2001:db8:0:5::3.55447: . ack 1590 win 8279 <[|tcp]>
19:19:51.364681 Out 2001:db8:0:5::3.55447 > 2001:db8:0:1::7.59562: P 1590:1600(10) ack 46 win 8008 <[|tcp]>
19:19:51.365802  In 2001:db8:0:1::7.59562 > 2001:db8:0:5::3.55447: . ack 1600 win 8279 <[|tcp]>
19:19:51.369359 Out 2001:db8:0:5::3.55447 > 2001:db8:0:1::7.59562: . 1600:3008(1408) ack 46 win 8008 <[|tcp]>
19:19:51.369447 Out 2001:db8:0:5::3.55447 > 2001:db8:0:1::7.59562: P 3008:3028(20) ack 46 win 8008 <[|tcp]>
19:19:51.370489 Out 2001:db8:0:5::3.55447 > 2001:db8:0:1::7.59562: . 3028:4436(1408) ack 46 win 8008 <[|tcp]>
19:19:51.370557 Out 2001:db8:0:5::3.55447 > 2001:db8:0:1::7.59562: P 4436:4456(20) ack 46 win 8008 <[|tcp]>
19:19:51.372847 Out 2001:db8:0:5::3.55447 > 2001:db8:0:1::7.59562: . 4456:5864(1408) ack 46 win 8008 <[|tcp]>
19:19:51.375056 Out 2001:db8:0:5::3.55447 > 2001:db8:0:1::7.59562: P 5864:5884(20) ack 46 win 8008 <[|tcp]>
19:19:51.375149  In 2001:db8:0:1::7.59562 > 2001:db8:0:5::3.55447: . ack 3008 win 8370 <[|tcp]>
19:19:51.375156  In 2001:db8:0:1::7.59562 > 2001:db8:0:5::3.55447: . ack 3028 win 8370 <[|tcp]>
19:19:51.375159  In 2001:db8:0:1::7.59562 > 2001:db8:0:5::3.55447: . ack 4436 win 8462 <[|tcp]>
19:19:51.375161  In 2001:db8:0:1::7.59562 > 2001:db8:0:5::3.55447: . ack 4456 win 8462 <[|tcp]>
19:19:51.375162  In 2001:db8:0:1::7.59562 > 2001:db8:0:5::3.55447: . ack 5864 win 8553 <[|tcp]>
19:19:51.376463 Out 2001:db8:0:5::3.55447 > 2001:db8:0:1::7.59562: . 5884:7292(1408) ack 46 win 8008 <[|tcp]>
19:19:51.376634 Out 2001:db8:0:5::3.55447 > 2001:db8:0:1::7.59562: P 7292:7312(20) ack 46 win 8008 <[|tcp]>
19:19:51.377680  In 2001:db8:0:1::7.59562 > 2001:db8:0:5::3.55447: . ack 5884 win 8553 <[|tcp]>
19:19:51.377688  In 2001:db8:0:1::7.59562 > 2001:db8:0:5::3.55447: . ack 7292 win 8644 <[|tcp]>
19:19:51.378019  In 2001:db8:0:1::7.59562 > 2001:db8:0:5::3.55447: . ack 7312 win 8644 <[|tcp]>
19:19:51.381585 Out 2001:db8:0:3::3.55447 > 2001:db8:0:1::7.56449: P 1:90(89) ack 1 win 8008 <[|tcp]>
19:19:51.382402  In 2001:db8:0:1::7.56449 > 2001:db8:0:3::3.55447: . ack 90 win 8644 <[|tcp]>
19:20:12.921700  In 2001:db8:0:1::7.59562 > 2001:db8:0:5::3.55447: P 46:51(5) ack 7312 win 8644 <[|tcp]>
19:20:12.922465  In 2001:db8:0:1::7.59562 > 2001:db8:0:5::3.55447: R 51:51(0) ack 7312 win 8644 <[|tcp]>
19:20:12.922949  In 2001:db8:0:1::7.56449 > 2001:db8:0:3::3.55447: R 1:1(0) ack 90 win 8644 <[|tcp]>
19:20:12.922957  In 10.7.7.7.46240 > 7.7.11.7.55447: R 0:0(0) ack 1 win 8644 <nop,nop,timestamp 3835160 3814686,opt-30:2001b2ee6d2a>
19:20:12.923302 Out 2001:db8:0:3::3.55447 > 2001:db8:0:1::7.56449: P 90:91(1) ack 1 win 8008 <[|tcp]>
19:20:12.923827  In 10.7.7.7.41233 > 7.7.9.7.55447: R 0:0(0) ack 1 win 8644 <nop,nop,timestamp 3835160 3814685,opt-30:2001b2ee6d2a>
19:20:12.923834  In 7.7.7.7.6543   > 7.7.9.7.55447: R 7:7(0) ack 1 win 8644 <nop,nop,timestamp 3835160 3814683,opt-30:2001b2ee6d2a>
19:20:12.924370  In 2001:db8:0:1::7.56449 > 2001:db8:0:3::3.55447: R 3239243037:3239243037(0) win 0
19:20:12.924992  In 7.7.7.7.33215  > 7.7.11.7.55447: . ack 1 win 8644 <nop,nop,timestamp 3835161 3814686,opt-30:7000f7992e0704[|tcp]>
19:20:12.926111 Out 7.7.11.7.55447 > 7.7.7.7.33215: R 1:1(0) ack 1 win 8008 <nop,nop,timestamp 3834748 3815099,opt-30:20018909ef7a>
```

Ilustración 15: traza obtenida en el router

A continuación se muestra la traza en la maquina que alberga el entorno virtualizado. Aunque el eje de tiempos no se corresponde con el de la traza del router R1, es el mismo ejemplo pero ejecutado dos veces:

```
user@shitbox:~$ sudo tcpdump -ni Lan4 tcp and not port 22
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on Lan4, link-type EN10MB (Ethernet), capture size 65535 bytes
```

```
01:22:08.292753 IP 10.7.7.7.45841 > 10.7.9.7.55447: Flags [S], seq 1969438484, win 29200, options [mss 1460,TS val 6563959 ecr 0,nop,wscale 5]
01:22:08.292914 IP 7.7.9.7.55447  > 10.7.7.7.40329: Flags [S.], seq 3205731971, ack 4651885, win 28560, options [mss 1460TS val 6563545 ecr 6563958,nop,wscale 5]
01:22:08.293203 IP 10.7.9.7.55447 > 10.7.7.7.45841: Flags [S.], seq 2708369543, ack 1969438485, win 28560, options [mss 1460,TS val 6563546 ecr 6563959,wscale 5]
01:22:08.293737 IP 10.7.7.7.45841 > 10.7.9.7.55447: Flags [.], ack 1, win 3650, options [nop,nop,TS val 6563959 ecr 6563546,mptcp join]
01:22:08.294119 IP 10.7.9.7.55447 > 10.7.7.7.45841: Flags [.], ack 1, win 2678, options [nop,nop,TS val 6563546 ecr 6563959,mptcp add-addr id 9 2001:db8:0:3::3]
01:22:08.295425 IP 7.7.7.7.43973  > 10.7.9.7.55447: Flags [S], seq 136306215, win 29200, options [mss 1460, TS val 6563959 ecr 0,nop,wscale 5]
01:22:08.295772 IP 7.7.9.7.55447  > 10.7.7.7.40329: Flags [.], ack 1, win 3570, options [nop,nop,TS val 6563546 ecr 6563959,mptcp add-addr id 10 2001:db8:0:5::3,]
01:22:08.301613 IP 7.7.7.7.43973  > 10.7.9.7.55447: Flags [.], ack 707187968, win 4563, options [nop,nop,TS val 6563961 ecr 6563546]
01:22:08.301989 IP 7.7.11.7.55447 > 10.7.7.7.37364: Flags [S.], seq 4273423756, ack 3313291987, win 28560, options [mss 1460, TS val 6563548 ecr 6563960,wscale 5]
01:22:08.303668 IP 7.7.11.7.55447 > 10.7.7.7.37364: Flags [.], ack 1, win 5355, options [nop,nop,TS val 6563548 ecr 6563961,mptcp dss ack 226904824]
01:22:33.612904 IP 7.7.7.7.43973  > 10.7.9.7.55447: Flags [R.], seq 0, ack 1, win 8279, options [nop,nop,TS val 6570289 ecr 6563548,mptcp dss ack 1731846726]
01:22:33.613248 IP 10.7.7.7.45841 > 10.7.9.7.55447: Flags [R.], seq 1, ack 1, win 8279, options [nop,nop,TS val 6570289 ecr 6563546,mptcp dss ack 1731846726]
```

Ilustración 16: traza obtenida en el host del entorno virtualizado

Para facilitar al lector la lectura y compresión del ejemplo, se muestra a continuación la configuración de los interfaces:

[1]: Configuración de los interface en el cliente

```
eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 7.7.7.7  netmask 255.255.255.0  broadcast 7.7.7.255
        inet6 2001:db8:0:1::7  prefixlen 64  scopeid 0x0<global>
        inet6 fe80::fd:ff:fe00:1  prefixlen 64  scopeid 0x20<link>
        ether 02:fd:00:00:00:01  txqueuelen 1000  (Ethernet)


eth2: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.7.7.7  netmask 255.0.0.0  broadcast 10.255.255.255
        inet6 fe80::fd:ff:fe00:2  prefixlen 64  scopeid 0x20<link>
        ether 02:fd:00:00:00:02  txqueuelen 1000  (Ethernet)


lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 0  (Local Loopback)
```

[2]: Configuración de los interfaces en el servidor:

```
eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 7.7.9.7  netmask 255.255.255.0  broadcast 7.7.9.255
        inet6 2001:db8:0:3::3  prefixlen 64  scopeid 0x0<global>
        inet6 fe80::fd:ff:fe00:101  prefixlen 64  scopeid 0x20<link>
        ether 02:fd:00:00:01:01  txqueuelen 1000  (Ethernet)


eth2: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 7.7.11.7  netmask 255.255.255.0  broadcast 7.7.11.255
        inet6 2001:db8:0:5::3  prefixlen 64  scopeid 0x0<global>
        inet6 fe80::fd:ff:fe00:102  prefixlen 64  scopeid 0x20<link>
        ether 02:fd:00:00:01:02  txqueuelen 1000  (Ethernet)


eth3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.7.9.7  netmask 255.0.0.0  broadcast 10.255.255.255
        inet6 fe80::fd:ff:fe00:103  prefixlen 64  scopeid 0x20<link>
        ether 02:fd:00:00:01:03  txqueuelen 1000  (Ethernet)


lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
```

[3]: Configuración de los interfaces en el router:

```
fxp0.0: flags=0x8000 <UP|MULTICAST>
        inet primary mtu 1500 local=7.7.7.1 dest=7.7.7.0/24 bcast=7.7.7.255
        inet6 primary mtu 1500 local=2001:db8:0:1::1 dest=2001:db8:0:1::/64
            local=fe80::2fd:ff:fe00:201 dest=fe80::/64


fxp1.0: flags=0x8000 <UP|MULTICAST>
        inet mtu 1500 local=7.7.9.1 dest=7.7.9.0/24 bcast=7.7.9.255
        inet6 mtu 1500 local=2001:db8:0:3::1 dest=2001:db8:0:3::/64
            local=fe80::2fd:ff:fe00:202 dest=fe80::/64


fxp2.0: flags=0x8000 <UP|MULTICAST>
        inet mtu 1500 local=7.7.11.1 dest=7.7.11.0/24 bcast=7.7.11.255
        inet6 mtu 1500 local=2001:db8:0:5::1 dest=2001:db8:0:5::/64
            local=fe80::2fd:ff:fe00:203 dest=fe80::/64
```

[4]: Configuración de los interface en el host de las máquinas virtuales:

```
Lan1    Link encap:Ethernet  HWaddr 02:00:00:92:c1:f5
        inet6 addr: fe80::ff:fe92:c1f5/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1


Lan2    Link encap:Ethernet  HWaddr 02:00:00:9c:d3:48
        inet6 addr: fe80::ff:fe9c:d348/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1


Lan3    Link encap:Ethernet  HWaddr 02:00:00:5c:17:2c
        inet6 addr: fe80::ff:fe5c:172c/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1


Lan4    Link encap:Ethernet  HWaddr 02:00:00:8f:69:2e
        inet addr:10.0.0.254  Bcast:0.0.0.0  Mask:255.0.0.0
        inet6 addr: fe80::ff:fe8f:692e/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
```

3.3.1 Reflexión sobre el ejemplo expuesto

En el ejemplo anterior se ha mostrado como MPTCP establece correctamente múltiples caminos entre un cliente y un servidor. El servidor está escuchando en la dirección in6addr_any IPv6 pero el cliente comienza el establecimiento de dirección hacia la dirección pública IPv4 del servidor. Esto nos ha permitido verificar que dual-stack en el lado servidor ha funcionado como se describe en la teoría [UNIX-1], siendo, además, ésta la forma aconsejada para escribir y desplegar aplicaciones IPv4-IPv6 interoperables. Cuando la transferencia de datos comenzó, MPTCP entró en acción, y en conjunción con IPv4 e IPv6 (dual-stack) se enviaron segmentos TCP con flag SYN activo por 11 caminos diferentes.

Los niveles de red IPv4 e IPv6, y la tecnología dual-stack han operado sin sorpresas en este ejemplo. Lo que ha sido sorprendente es el comportamiento de MPTCP, ya que no solo ha conseguido establecer múltiples rutas, haciendo honor a su nombre, sino que además ha funcionado con armonía junto a dual-stack. Como se verá en apartados posteriores, MPTCP no requiere ningún cambio en el API socket, hecho que facilita su adopción. El aparente buen funcionamiento de este nuevo protocolo de transporte junto con la, en teoría, inexistente necesidad de reescribir aplicaciones para usarlo, hacen que MPTCP sea un protocolo de nivel de transporte viable y a tener en cuenta a la hora de diseñar aplicaciones.

Más información sobre MPTCP y el uso de este en las aplicaciones en los apartados sobre protocolos de nivel de transporte y nivel de aplicación.

4. Internet Protocol

Las aplicaciones NetRat y NetExample ([NETRAT] [NETEXAMPLE]), a través de las librerías NetLayer y NetCraft ([NETLAYER] [NETCRAFT]) manejan datagramas IPv4 e IPv6 directamente mediante sockets de tipo SOCK_RAW. IPv4 e IPv6 tienen como objetivo permitir que un paquete fluya desde un origen a un destino de la red. Más información sobre IPv4 e IPv6 en [RFC791], [RFC2460], [TCP/IP-1-94], [TCP/IP-1-12] y [TANNENBAUM]. El código desarrollado en este trabajo ha tenido, desde el comienzo, como requerimiento funcional importante la compatibilidad entre diferentes plataformas (Windows, FreeBSD por motivos históricos y Linux). Como curiosidad comentar que me ha resultado sorprendente que a estas alturas del siglo XXI en el que se está poniendo en marcha un protocolo nuevo, como es IPv6, la constante que identifica la familia IPv6 (AF_INET6) tiene un valor diferente en cada una de las plataformas usadas: 10 en Linux, 23 en Windows y 28 en BSD. Parece una contradicción que AF_INET, que designa la familia IPv4 tenga una valor homogéneo de 2 en todas las plataformas mientras que AF_INET6 no.

En las ilustraciones 17 y 18 se muestra el aspecto de los datagramas IPv4 e IPv6. En las ilustraciones 19 y 20 se muestra el aspecto de las direcciones IPv4 e IPv6 como aperitivo a lo que está por llegar. Fragmentos de código tomados del kernel de FreeBSD [FreeBSD].

```
    struct ip {
        unsigned int ip_hl:4;           /* header length */
        unsigned int ip_v:4;            /* version */
        uint8_t ip_tos;                 /* type of service */
        uint16_t ip_len;                /* total length */
        uint16_t ip_id;                 /* identification */
        uint16_t ip_off;                /* fragment offset field */
#define IP_RF 0x8000                    /* reserved fragment flag */
#define IP_DF 0x4000                    /* dont fragment flag */
#define IP_MF 0x2000                    /* more fragments flag */
#define IP_OFFMASK 0x1fff               /* mask for fragmenting bits */
        uint8_t ip_ttl;                 /* time to live */
        uint8_t ip_p;                   /* protocol */
        uint16_t ip_sum;                /* checksum */
        struct in_addr ip_src, ip_dst;  /* source and dest address */
    };
```

Ilustración 17: cabecera de datagrama IPv4

```
    struct ip6 {
        union {
            struct ip6_hdrctl {
                uint32_t ip6_un1_flow;  /* 20 bits of flow-ID */
                uint16_t ip6_un1_plen;  /* payload length */
                uint8_t ip6_un1_nxt;    /* next header */
                uint8_t ip6_un1_hlim;   /* hop limit */
            } ip6_un1;
            uint8_t ip6_un2_vfc;        /* 4 bits version, top 4 bits class */
        } ip6_ctlun;
        struct in6_addr ip6_src;        /* source address */
        struct in6_addr ip6_dst;        /* destination address */
    };
```

Ilustración 18: cabecera de datagrama IPv6

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

Ilustración 19: dirección IPv4

```
     struct in6_addr {
         union {
             uint8_t __u6_addr8[16];
             uint16_t __u6_addr16[8];
             uint32_t __u6_addr32[4];
         } __u6_addr;                    /* 128-bit IP6 address */
     };
```

Ilustración 20: dirección IPv6

4.1 Cabeceras IPv4 e IPv6

Un datagrama es un paquete en el que la información que identifica el origen y el destino reside en el paquete en sí mismo [TANNENBAUM]. La cabecera de los datagramas IPv6 está alineada a 64 bits, en lugar de 32, la longitud de los campos de direcciones ha aumentado a 128 bits, y en general es más sencilla que su predecesora de IPv4. Las opciones en IPv6 se codifican en sus propias cabeceras.

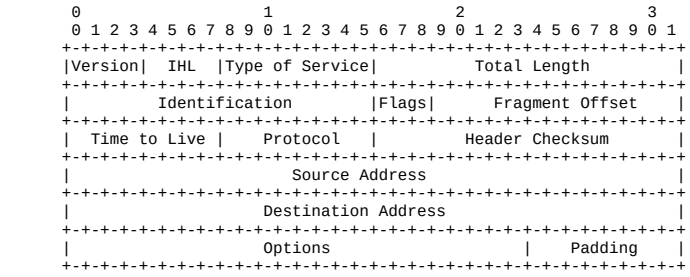


Ilustración 21: cabecera IPv4 alienada a 32 bits [RFC791]

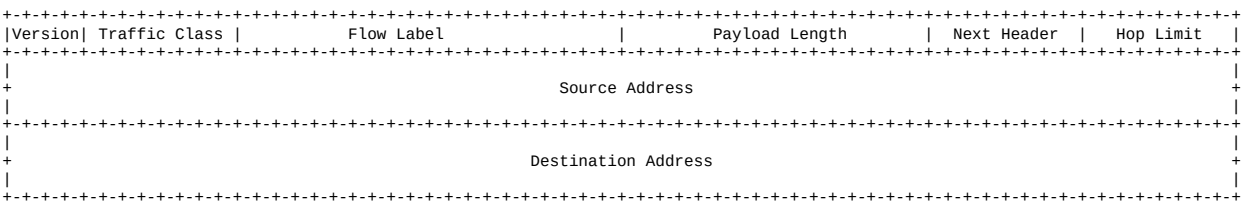


Ilustración 22: cabecera IPv6 alineada a 64 bits [RFC2460]

A continuación se muestra las funciones que se han usado para confeccionar los datagramas IPv4 y IPv6, tanto mediante la librería NetCraft [NETCRAFT] o directamente en Perl en el cliente de la aplicación NetRat [NETRAT]. En la librería NetCraft se utilizan las estructuras definidas en el kernel de FreeBSD como referencia para la cabecera (ilustraciones 17 y 18) y un puntero a una zona de memoria para la zona de carga. El usuario de la librería es responsable de reservar tanta memoria como considere necesaria para la zona de datos.

```
    void Ip::SetHeader(ip *_h)
    {
#ifdef DEBUG
        std::cout << "[Ip::SetHeader]" << std::endl;
#endif
        Packet<ip>::SetHeader(_h);
        memcpy(header, _h, _h->ip_hl<<2);
        hlength = _h->ip_hl<<2;
        payload += hlength;
    }
```

Ilustración 23: función IP:SetHeader en librería NetCraft

```
    void Ip::SetPayload(char *_p, uint16_t _l)
    {
#ifdef DEBUG
        std::cout << "[Ip::SetPayload]" << std::endl;
#endif
        Packet<ip>::SetPayload(_p, _l);
        plength = _l;
    }
```

Ilustración 24: función IP::SetPayload en librería NetCraft

```
    void Ip6::SetHeader(ip6 *_h)
    {
#ifdef DEBUG
        std::cout << "[Ip6::SetHeader]" << std::endl;
#endif
        Packet<ip6>::SetHeader(_h);
        memcpy(header, _h, IP6_HEADER_LENGTH);
        hlength = IP6_HEADER_LENGTH;
        payload += hlength;
    }
```

Ilustración 25: función IP6::SetHeader en librería NetCraft

```
void Ip6::SetPayload(char *_p, uint16_t _l)
{
#ifdef DEBUG
        std::cout << "[Ip6::SetPayload]" << std::endl;
#endif
        Packet<ip6>::SetPayload(_p, _l);
        plength = _l;
}
```

Ilustración 26: función IP6::SetPayload en librería NetCraft

```
template<class pheader>
void Packet<pheader>::SetHeader(pheader *_h)
{
#ifdef DEBUG
        std::cout << "[Packet::SetHeader]: doing nothing here" << std::endl;
#endif
}
```

Ilustración 27: función Packet<pheader>::SetHeader base en librería NetCraft

```
template<class pheader>
void Packet<pheader>::SetPayload(char *_p, uint16_t _l)
{
#ifdef DEBUG
        std::cout << "[Packet::SetPayload]" << std::endl;
#endif
        memcpy(payload, _p, _l);
}
```

Ilustración 28: función Packet<pheader>::SetPayload base en librería NetCraft

En las ilustración 23 a 28 se muestran las funciones de la librería NetCraft elaborada para este trabajo que se usan para construir los datagramas IPv4 e IPv6, lo cual se hace mediante las estructuras de datos mostradas en las ilustraciones 17 y 18 y mediante el uso internamente de punteros.

La librería NetCraft se puede usar tanto en un cliente como en un servidor, pero al estar escrito en Perl el cliente NetRat, se utilizaron otras funciones (escritas en Perl) para la codificación y decodificación de datagramas IP. En las ilustraciones a continuación se muestran las funciones del cliente en Perl para construir mensajes IPv4 e IPv6 en Perl, que al no disponer de estructuras como las mostradas en las ilustraciones 17 y 18 hace necesario codificar a binario de forma manual mediante las funciones pack (codificación) y unpack (decodificación), lo cual complica el código considerablemente:

```
sub _EncodeIP
{
    my $self=shift;
    my $ip=shift;
    $ip->{offset} = (substr(unpack('B8', pack('C', $ip->{flags})), 5, 3).substr(unpack('B16', pack('n', $ip->{offset})), 3, 13));
    $ip->{headerLength} = (5+length($ip->{options}));
    no warnings;
    my $pkt=pack(
            'C
             C          n
             n          B16
             C          C
             n          a4
             a4         a*',
        ($ip->{version}<<4)|$ip->{headerLength},
        $ip->{tos},
        $ip->{totalLength},
        $ip->{id},
        $ip->{offset},
        $ip->{ttl},
        $ip->{protocol},
        $ip->{checkSum},
        inet_pton(&NetRatConf::INET, $ip->{sourceIP}),
        inet_pton(&NetRatConf::INET, $ip->{destIP}),
        $ip->{options}
    );
    return $pkt;
}
```

Ilustración 29: codificación de un datagrama IPv4 en Perl

```
sub _DecodeIP
{
    my $self=shift;
    my $pkt=shift;
    my ($byte, $sword);
    my $ip=SockOps::IP->new();
    ($byte,
     $ip->{tos},
     $ip->{totalLength},
     $ip->{id},
     $sword,
     $ip->{ttl},
     $ip->{protocol},
     $ip->{checkSum},
     $ip->{sourceIP},
     $ip->{destIP},
     $ip->{options}) = unpack(
             'C        C
              n        n
              B16      C
              C        n
              a4       a4
              a*',
        $pkt
    );
    $ip->{version} = ($byte&0xf0)>>4;
    $ip->{headerLength} = $byte&0x0f;
    $ip->{flags} = substr($sword, 0, 3, '');
    $ip->{flags} = sprintf("%08d", $ip->{flags});
    $ip->{flags} = unpack('C', pack('B8', $ip->{flags}));
    $ip->{offset} = sprintf("%016d", $sword);
    $ip->{offset} = unpack('n', pack('B16', $ip->{offset}));
    my $olen=$ip->{headerLength}-5;
    $olen = 0 if ($olen < 0);
    ($ip->{options}, $ip->{data}) = unpack('a'.$olen.'a*', $ip->{options});
    $ip->{sourceIP} = inet_ntoa($ip->{sourceIP});
    $ip->{destIP} = inet_ntoa($ip->{destIP});
    return $ip;
}
```

Ilustración 30: decodificación de un datagrama IPv4 en Perl

```
sub _EncodeIP6
{
    my $self=shift;
    my $ip6=shift;
    $ip6->{length} = length($ip6->{data});
    my $pkt=pack(
           'N           n
            c           C
            a16         a16',
        ($ip6->{flow}&0x000fffff)|(($ip6->{ecn}<<20)&0x00300000)|(($ip6->{ds}<<22)&0x0fc00000)|(($ip6->{version}<<28)&0xf0000000),
        $ip6->{length},
        $ip6->{next},
        $ip6->{hop},
        inet_pton(&NetRatConf::INET6, $ip6->{sourceIP}),
        inet_pton(&NetRatConf::INET6, $ip6->{destIP})
    );
    return $pkt;
}
```

Ilustración 31: codificación de un datagrama IPv6 en Perl

```
sub _DecodeIP6
{
    my $self=shift;
    my $pkt=shift;
    my ($word, $src, $dst);
    my $ip6=SockOps::IP6->new();
    ($word,
     $ip6->{length},
     $ip6->{next},
     $ip6->{hop},
     $src,
     $dst,
     $ip6->{data}) = unpack(
            'N          n
             c          C
             a16        a16
             a*',
        $pkt
    );
    $ip6->{version} = ($word&0xf0000000)>>28;
    $ip6->{ds} = ($word&0x0fc00000)>>22;
    $ip6->{ecn} = ($word&0x00300000)>>20;
    $ip6->{flow} = $word&0x000fffff;
    $ip6->{sourceIP} = inet_ntop(&NetRatConf::INET6, $src);
    $ip6->{destIP} = inet_ntop(&NetRatConf::INET6, $dst);
    return $ip6;
}
```

Ilustración 32: decodificación de un datagrama IPv6 en Perl

Y así es como se ha trabajo a nivel de SOCK_RAW con los datagramas IPv4 e IPv6, tanto en C++ mediante la librería NetCraft, como en Perl.

La librería NetCraft se escribió precisamente para esta función: simplificar la tarea de construir datagramas IPv4 e IPv6 y mensajes ICMPv4 e ICMPv6, ya que trabajar a nivel de red es más complicado. Entrar en detalles de implementación de las librerías se sale fuera del ámbito de esta memoria ya que, nada más en código hay miles de líneas de código, y describirlo en detalle aumentaría el tamaño de esta memoria de forma factorial, por ello se invita al lector a descargarse el código, compilarlo y verlo por sí mismo, en el caso de que la descripción de los fragmentos que se han seleccionado para la memoria no sepan a poco.

4.2 Direcciones especiales IPv4 e IPv6

De los campos existentes en las cabeceras IPv4 e IPv6, los más usados, con diferencia, durante la elaboración de este trabajo han sido los relacionados con las direcciones origen y destino, es por ello que a continuación se va a exponer detalladamente lo que se ha observado. Tanto en IPv4 como en IPv6 se definen una serie de direcciones especiales [TCP/IP-1-12]:

```
0.0.0.0/8           host on the local network
10.0.0.0/8          private networks
127.0.0.1/8         loopback
169.254.0.0/16      link local addresses
172.16.0.0/12       (1100) private networks
192.0.0.0/24        IANA reserved
192.0.2.0/24        documentation
198.88.99.0/24      6to4 relays
192.168.0.0/16      private networks
198.18.0.0/15       benchmarcks
198.51.100.0/24     documentation
203.0.113.0/24      documentation
224.0.0.0/4         multicast
240.0.0.0/4         reserved
255.255.255.255/32  broadcast
::/0                default route entry
::/128              unspecified address
::1/128             loopback
::ffff:0:0/96       IPv4 mapped addresses
::{IPv4}/96         IPv4 compatible addr
2001::/32           Teredo address
2001:10::/28        Overlay routable crypto
2001:db8::/32       documentation
2002::/16           6to4 tunnel relays
3ffe::/16           6bone experiments
5f00::/16           6bone experiments
fc00::/7            Unique, local unicast
fe80::/10           Link-local unicast
ff00::/8             Multicast address
```

Ilustración 33: direcciones especiales IPv4 e IPv6

En este trabajo se han utilizado las direcciones 0.0.0.0/0 y ::/0 como ruta por defecto, 127.0.0.1 y ::1/128 localhost, 10.0.0.0/8, 172.16.0.0/12 y 192.168.0.0/16 para subredes privadas, ::ffff:0:0/96 dirección IPv4 mapeada a IPv6 y fe80::/10 para direcciones de ámbito local.

4.3 Direcciones IPv6

Las direcciones IPv6 tienen una longitud de 128 bits (cuatro veces más que IPv4). Su estructura también es diferente. Según el prefijo de la dirección IPv6 el ámbito es diferente. El ámbito de la dirección IPv6 es la porción de la red en la que puede ser usada:

Address type	Binary prefix	IPv6 notation
Unspecified	00...0 (128 bits)	::/128
Loopback	00...1 (128 bits)	::1/128
Multicast	11111111	FF00::/8
Link-Local unicast	1111111010	FE80::/10
Global Unicast	(resto Internet)	

```
Node-local:     comunicaciones en el mismo ordenador
Link-local:     para máquinas en el mismo enlace de red
Multicast:      para un conjunto de máquinas
Global:         en todo Internet
```

En IPv6 no hay direcciones broadcast y es habitual que un mismo dispositivo tenga varias direcciones IPv6 de varios tipos.

Direcciones Unicast:

De forma similar a como ocurre con las direcciones IPv4 CIDR[36], las direcciones Unicast IPv6 son "agregables"[37]con prefijos [TCP/IP-1-12]. Existen varios tipos de direcciones IPv6 Unicast: Link-Local Unicast y Global Unicast. Las direcciones Link-Local Unicast identifican interfaces de forma local mientras que las direcciones Global Unicast identifican interfaces en todo Internet.

Direcciones Link-Local Unicast:

Éstas se utilizan para identificar interfaces en un enlace, en contraste con las globales que identifican dispositivos de forma única a nivel global. Este tipo de direcciones tienen prefijo (fe80::/10) y hay dos maneras en las que el dispositivo puede crear esta dirección: de forma aleatoria (por ejemplos maquinas con Windows lo hacen de esta manera por defecto), o bien, generarla a partir de la dirección MAC.A continuación se muestra una dirección IPv6 de tipo Link-Local Unicast obtenida a partir de la dirección MAC de la tarjeta de red:

```
user@shitbox:~$ /sbin/ifconfig
eth0      Link encap:Ethernet  HWaddr[38] 70:71:bc:12:a4:40
          inet addr:192.168.1.251  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::7271:bcff:fe12:a440/64 Scope:Link

OUI[39] (Organizationally Unique Identifier), EUI[40] (Extended Unique Identifier)
Los 24 bits del OUI ocupan los tres primeros bytes de EUI-48 o EUI-64

                    |0       0 0       1 1       2|
                    |0       7 8       5 6       3|
                    +----+----+----+----+----+----+
                    |cccc|ccug|cccc|cccc|cccc|cccc|
                    +----+----+----+----+----+----+
```

Ilustración 34: ejemplo de dirección IPv6 de tipo link-local

u/g-bits: "u" significa que la dirección es administrada localmente. "g" significa que la dirección es de tipo multicast, c-bit: company-bits.

Las direcciones Link-Local Unicast en IPv6 pueden ser usadas en varios interfaces simultáneamente en una misma máquina siempre y cuando los interfaces en los que se usa estén conectados a diferentes enlaces. Esto introduce el concepto de *scope-zone* [RFC4007]. El *scope-zone* es una zona interconectada en una topología, por ejemplo, el conjunto de enlaces conectados a un router. El hecho de que existan diferentes zonas permite que una misma dirección link-local se pueda reutilizar en diferentes zonas que no estén conectadas. En el ejemplo anterior expresamos la zona de la dirección IP local fe80::7271:bcff:fe12:a440%eth0. Las direcciones IPv6 globales no tienen scope ya que no se reutilizan en diferentes interfaces.

A continuación se muestra un sencillo script en Perl [CAMEL] que imprime las direcciones IPv6 en existen en los diferentes interfaces de una máquina y el scope asociado a las que son de tipo Link-Local (extracto tomado del cliente NetRat [NETRAT]):

[36] CIDR: Classless Inter Domain Routing: las direcciones IPv4 sin clases (A, B, C, D y E) desaparecieron para mitigar el agotamiento de direcciones de clase B y la longitud y complejidad de las tablas de rutas.
[37] Address Aggregation: enrutado jerarquico para minizar tablas de rutas.
[38] IID: Interface IDentifiers, es la dirección MAC de un dispositivo de red
[39] OUI: Organizationally Unique Identifier, son los tres primero bytes del EUI (48 o 64).
[40] EUI: Extended Unique Identifier.

```
my @ipBuffer=`/sbin/ifconfig`;
my @ipList=();
my $scope;
foreach my $ipLine (@ipBuffer) {
    if ($ipLine=~/^(\S+?)\s/) {
        $scope = $1;
    }
    if ($ipLine=~/inet6 addr|inet6 /) {
        (my $ipTmp)=$ipLine=~/($ip6RE)/;
        if ($ipTmp=~/^fe.+?$/) {
            push @ipList, $ipTmp."%".$scope;
        } else {
            push @ipList, $ipTmp;
        }
    }
}
my $i=0;
foreach my $ipTmp (@ipList) {
    print STDOUT "[", $i, "]: ", $ipTmp, "\n";
    $i++;
}
```

Ilustración 35: Script en Perl en muestra por pantalla las direcciones IPv6

4.4 Ejemplo: direcciones IPv6 Link-Local en la práctica

En las secciones anteriores se ha presentado de forma resumida algunos de los conceptos más importantes relacionados con las direcciones IPv6 que se han manejado en este proyecto. Ahora, se va a mostrar con un ejemplo propio y concreto, y que ameniza la lectura de este trabajo, el uso de direcciones IPv6 de tipo Link-Local con el cliente y el servidor NetRat [NETRAT] [ERNESTO81].

En la ilustración 36 se muestra la fase de inicialización del servidor NetRat cuando está configurado para trabajar en IPv6 (dirección in6addr_any) y TCP (puerto de escucha 55447). En la imagen se ve cómo se invocan los correspondientes constructores y como se queda a la espera de recibir peticiones de conexión (cosa que hace de manera no bloqueante, gracias al uso de la función select y opciones Socket, como se describirá más adelante).A modo de introducción a las aplicaciones producidas, comentar que el servidor NetRat no se configura mediante línea de comandos, sino que lo hace mediante un fichero de configuración (NetRat.conf), lo cual no necesita compilación. También se puede configurar en tiempo de ejecución sobre la marcha desde el cliente mediante comando que el servidor interpreta, o se puede hacer en tiempo de compilación (en caso de que únicamente se disponga del ejecutable y no sea posible usar el fichero de configuración). En la traza mostrada en la ilustración 36 se observa que está configurado para trabajar con TCP sobre IPv6:

```
user@shitbox:~/Workspace/NetRat/NetRatServer$ ./NetRat
### 05/28/14 17:47:38 Log : [Logger::Logger]: logger file @ /tmp/Logger.txt
### 05/28/14 17:47:38 Log : [ServiceConfigurator::ServiceConfigurator]
### 05/28/14 17:47:38 Log : [ProtocolConfigurator::ProtocolConfigurator]
### 05/28/14 17:47:38 Log : [SecurityConfigurator::SecurityConfigurator]
### 05/28/14 17:47:38 Log : [ConfiguratorManager::ConfiguratorManager]
### 05/28/14 17:47:38 Log : [Main]: *Nix
### 05/28/14 17:47:38 Log : [ConfiguratorManager::StaticConfigure]: Configuration file: NetRat.conf
### 05/28/14 17:47:38 Log : [ServiceConfigurator::Config]: Number of Threads: 5
### 05/28/14 17:47:38 Log : [ServiceConfigurator::Config]: daemonize: false
### 05/28/14 17:47:38 Log : [ServiceConfigurator::Config]: IcmpHolePunch: false
### 05/28/14 17:47:38 Log : [ServiceConfigurator::Config]: ICMP enabled: false
### 05/28/14 17:47:38 Log : [ServiceConfigurator::Config]: DisFirewall: false
### 05/28/14 17:47:38 Log : [ServiceConfigurator::Config]: DoExec: true
### 05/28/14 17:47:38 Log : [ServiceConfigurator::Config]: MagicWord: _sid_
### 05/28/14 17:47:38 Log : [ServiceConfigurator::Config]: SafeGuard: 1000
### 05/28/14 17:47:38 Log : [ServiceConfigurator::Config]: MaxBuffer: 4096
### 05/28/14 17:47:38 Log : [ProtocolConfigurator::Config]: IPv6: true
### 05/28/14 17:47:38 Log : [ProtocolConfigurator::Config]: Tcp: true
### 05/28/14 17:47:38 Log : [ProtocolConfigurator::Config]: Udp: false
### 05/28/14 17:47:38 Log : [ProtocolConfigurator::Config]: Sctp: false
### 05/28/14 17:47:38 Log : [ProtocolConfigurator::Config]: Mptcp: false
### 05/28/14 17:47:38 Log : [ProtocolConfigurator::Config]: Host: ::
### 05/28/14 17:47:38 Log : [ProtocolConfigurator::Config]: Port: 55447
### 05/28/14 17:47:38 Log : [SecurityConfigurator::Config]: Ssl: false
### 05/28/14 17:47:38 Log : [SecurityConfigurator::Config]: IPsec: false
### 05/28/14 17:47:38 Log : [Svc::Svc]
### 05/28/14 17:47:38 Log : [SvcNix::SvcNix]
### 05/28/14 17:47:38 Log : [SvcNix::SvcMain]
### 05/28/14 17:47:38 Log : [Svc::SvcDynamicSetUp]
### 05/28/14 17:47:38 Log : [SvcNix::SvcMain]: preparing protocol threads
### 05/28/14 17:47:38 Log : [SvcNix::SvcMain]: tcpMode enabled, creating its protocol thread
### 05/28/14 17:47:38 Log : [SvcNix::SvcMain]: blocking in Select
### 05/28/14 17:47:38 Log : [Svc::SvcSockSetUp]
### 05/28/14 17:47:38 Log : [SockImpTcp::SockImpTcp]
### 05/28/14 17:47:38 Log : [SockImp::SockInit]
### 05/28/14 17:47:38 Log : [Svc::SvcTcpMode]
### 05/28/14 17:47:38 Log : [SockImpTcp::SockConnect]: SockTcp
### 05/28/14 17:47:38 Log : [Svc::SvcTcpMode]: main loop
### 05/28/14 17:47:38 Log : [Svc::SvcTcpMode]: pool size 1
### 05/28/14 17:47:38 Log : [SockImpTcp::SockListen]: SockTcp
### 05/28/14 17:47:38 Log : [SockImpTcp::SockListen]: blocked before Accept
### 05/28/14 17:47:38 Log : [SockImpTcp::SockListen]: SockTcp blocking in Select
### 05/28/14 17:47:42 Log : [SockImpTcp::SockListen]: SockTcp blocking in Select
### 05/28/14 17:47:44 Log : [SvcNix::SvcMain]: checking comm threads status ...
### 05/28/14 17:47:44 Log : [SvcNix::SvcMain]: cycling #thread 0
### 05/28/14 17:47:44 Log : [SvcNix::SvcMain]: blocking in Select
### 05/28/14 17:47:45 Log : [SockImpTcp::SockListen]: SockTcp blocking in Select
```

Ilustración 36: Servidor NetRat arrancando: escuchando en 0::0:55447 (TCP)

```
user@shitbox:~/Workspace/NetRat/NetRatClient$ ./NetRat.pl -h
NetRat client options :
----------------------
  {--dest or -d IPaddr}       Destination IPv4 address
  {--dest6 or -6 IP6addr}     Destination IPv6 address
  [--port or -p servPort]     Destination port (default: 10 UDP || 10 TCP)
  [--tcp]                     TCP transport protocol
  [--udp]                     UDP transport protocol
  [--sctp]                    SCTP transport protocol
  [--auto]                    Automatic reconnect to remote server ?
  [--hole]                    ICMPv4 or ICMPv6 Hole Punching ? (requires to be super user)
  [--tool or -t]              NetRatClient behavies as a tool set (inspired in NetCat)
  [--stun]                    STUN request IntraNAT ?
    [--server uri]            STUN server
  [--scan                     Scan :
    [--proc #Process]         number of processes to use in the scan
    [--range Min-MaxPort]     range of ports to scan
    [--sleep Seconds] ]       sleep seconds between scan
  [--lsrr IPhop]              Loose Source and Route Record ?
  [--fire]                    Firewalking ?
  [--verb or -v]              verbose ?

Examples :
----------
  TCP Mode
    ./NetRat.pl --dest 192.168.0.237 --verb
    ./NetRat.pl -d 192.168.0.237 -v
  UDP Mode
    ./NetRat.pl --dest 192.168.0.237 --udp --verb
    ./NetRat.pl -d 192.168.0.237 -u -n -v
  ICMP Mode
    sudo ./NetRat.pl --dest 192.168.0.237 --udp --verb
    sudo ./NetRat.pl -d 192.168.0.237 -u -v

          /--------------\
 (oO) ... |  Bye Bye !!! |
 /||\     \--------------/
```

Ilustración 37: Cliente NetRat (menú de ayuda)

En la ilustración 37 se muestra el menú de ayuda con las diferentes opciones que soporta el programa cliente. En este ejemplo concreto, nos vamos a conectar mediante IPv6 y TCP al servidor. El servidor está escuchando en la dirección ::, y como se está ejecutando el cliente en la misma máquina se puede usar la dirección Link-Local como dirección de destino al servidor y como origen para el cliente.

En la ilustración 38 se muestra el comando usado para conectarnos al servidor donde le indicamos que la dirección de destino es fe80::7271:bcff:fe12:a440 (fe80::/10 con la EUI-64 a partir de la IID), el puerto de destino es 55447 y el protocolo TCP. Antes de realizar la conexión al servidor, la aplicación cliente pide elegir la dirección IPv6 que deseamos usar con dirección de origen (se fijará usando bind). El código mostrado en la ilustración 35 es el encargado de comprobar los interfaces y las direcciones disponibles a usar en el cliente (esto se añadió para poder seleccionar manualmente la dirección IPv4 o IPv6 a usar en tiempo de ejecución). En este caso hay dos opciones:

```
[0]: fe80::7271:bcff:fe12:a440%eth0
[1]: ::1
```

La opción [0] es la dirección Link-Local IPv6 junto con el *scope-zone* de ésta, que es eth0 (quiere esto decir que esta dirección se está usando en el interfaz eth0). El *scope-zone* se añade como un sufijo separado por el carácter % a la dirección IPv6 y puede ser numérico (el identificador numérico del interfaz) o una cadena de caracteres (el nombre el interfaz). En este ejemplo el identificador del interfaz es eth0. En caso de tener más interfaces conectados a diferentes enlaces sería posible reutilizar esa misma dirección de tipo local en más de un lugar simultáneamente gracias al *scope-zone,* que hace que sea única.

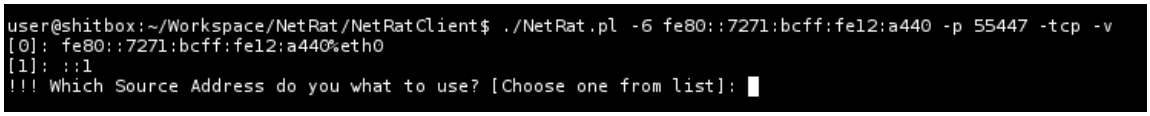


Ilustración 38: cliente NetRat preguntado a qué dirección IPv6 nos queremos "bindear[41]"

[41] Usando comando Bind ofrecido por la API de Sockets.

```
user@shitbox:~/Workspace/NetRat/NetRatClient$ ./NetRat.pl -6 fe80::7271:bcff:fe12:a440 -p 55447 -tcp -v
[0]: fe80::7271:bcff:fe12:a440%eth0
[1]: ::1
!!! Which Source Address do you what to use? [Choose one from list]: 0
### 2014/05/28 17:48:22 Log : Source Address: fe80::7271:bcff:fe12:a440%eth0
### 2014/05/28 17:48:22 Log : Parameters :
$VAR1 = bless( {
                 'range' => '1024',
                 'public' => '127.0.0.1',
                 'hole' => 0,
                 'source' => 'fe80::7271:bcff:fe12:a440%eth0',
                 'sstun' => 'stun1.voiceeclipse.net',
                 'fire' => 0,
                 'sleep' => 0,
                 'dest' => 'fe80::7271:bcff:fe12:a440',
                 'scope' => '',
                 'lsrr' => '',
                 'scan' => 0,
                 'proc' => 5,
                 'family' => 10,
                 'six' => 1,
                 '_user' => 'user',
                 '_cwd' => '~Workspace/NetRat/NetRatClient',
                 'osname' => 'linux',
                 'verb' => 1,
                 'recx' => 0,
                 'shellcmd' => '(?:type|cat)',
                 '_alrm' => 0,
                 'port' => '55447',
                 'tool' => 0,
                 '_cpphase1' => 0,
                 '_hostname' => 'shitbox',
                 'stun' => 0,
                 'pport' => '00006543',
                 'proto' => 'tcp',
                 '_kidpid' => 0,
                 '_cpphase2' => 0,
                 'shell' => 0,
                 'ttl' => 0
               }, 'NetRatConf' );
### 2014/05/28 17:48:22 Log : Skipping STUN request (in IPv6 ideal world there are no NATs)...
### 2014/05/28 17:48:22 Log : Public IP:Port fe80::7271:bcff:fe12:a440%eth0:00006543
### 2014/05/28 17:48:22 Log : Skipping Scanning...
### 2014/05/28 17:48:22 Log : Skipping Firewalking...
### 2014/05/28 17:48:22 Log : Connecting to fe80::7271:bcff:fe12:a440
### 2014/05/28 17:48:22 Log : Resolving TCP
### 2014/05/28 17:48:22 Log : GetAddrInfo for TCP
### 2014/05/28 17:48:22 Log : Resolved to fe80::7271:bcff:fe12:a440:55447
### 2014/05/28 17:48:22 Log : Resolved to fe80::7271:bcff:fe12:a440%eth0:6543
### 2014/05/28 17:48:22 Log : Creating TCP socket...
### 2014/05/28 17:48:22 Log : Skipping route IP option...
### 2014/05/28 17:48:22 Log : OK !!!
```

Ilustración 39: cliente NetRat ha establecido conexión TCP usando dirección IPv6 Link-Local

En la ilustración 39 se muestra cómo, una vez que se selecciona la dirección IPv6 (en este caso concreto la de tipo Link-Local con *scope-zone* eth0), se establece la conexión con el servidor (que es lo relevante en este punto). Las estructuras, tanto origen como destino, de tipo sockaddr se calculan mediante la función GetAddrInfo (que es independiente del protocolo de red usado, IPv4 o IPv6) [UNIX-1].

En la ilustración 40, correspondiente a la parte servidor, se muestra que efectivamente se ha establecido una conexión y se imprimen por pantalla los valores de los descriptores de fichero del socket TCP (tanto el pasivo a la escucha, como el activo conectado). En este punto del ejemplo hemos establecido una conexión TCP sobre IPv6 entre un cliente y un servidor que se ejecutan en la misma máquina, y en el cliente hemos usado la dirección de tipo Link-Local para especificar tanto origen como destino.

```
### 05/28/14 17:48:20 Log : [SockImpTcp::SockListen]: SockTcp blocking in Select
### 05/28/14 17:48:22 Log : [SockImpTcp::SockShell]
### 05/28/14 17:48:22 Log : [SockImpTcp::SockShell]: listenfd 4
### 05/28/14 17:48:22 Log : [SockImpTcp::SockShell]: writefd 5
### 05/28/14 17:48:22 Log : [SockImpTcp::SockShell]: SockTcp blocking in Select
### 05/28/14 17:48:22 Log : [SvcNix::SvcMain]: checking comm threads status ...
### 05/28/14 17:48:22 Log : [SvcNix::SvcMain]: cycling #thread 0
### 05/28/14 17:48:22 Log : [SvcNix::SvcMain]: blocking in Select
```

Ilustración 40: descriptores de fichero

```
user@shitbox:~$ sudo tcpdump -ni lo tcp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on lo, link-type EN10MB (Ethernet), capture size 65535 bytes
17:48:22.047411 IP6 fe80::7271:bcff:fe12:a440.6543 > fe80::7271:bcff:fe12:a440.55447: Flags [S], seq 1980231711, win 4
90, options [mss 65476,sackOK,TS val 16390290 ecr 0,nop,wscale 7], length 0
17:48:22.047439 IP6 fe80::7271:bcff:fe12:a440.55447 > fe80::7271:bcff:fe12:a440.6543: Flags [S.], seq 40388719, ack 19
231712, win 43690, options [mss 65476,sackOK,TS val 16390290 ecr 16390290,nop,wscale 7], length 0
17:48:22.047465 IP6 fe80::7271:bcff:fe12:a440.6543 > fe80::7271:bcff:fe12:a440.55447: Flags [.], ack 1, win 342, optio
 [nop,nop,TS val 16390290 ecr 16390290], length 0
```

Ilustración 41: TCP "3-way-hanshake" entre el cliente y el servidor

En la traza de la ilustración 41 se muestra lo que Tcpdump [TCPDUMP] ha capturado: es el establecimiento de conexión TCP en el que, tanto la dirección origen como la de destino de todos los segmentos es la misma (al estar en la misma máquina la diferencia, en este caso, es el número de puerto). Aunque la dirección que se usa es la de tipo link-local fe80::/10 y se ha especificado el *scope-zone* a "eth0", el interfaz usado es el de loopback[42] [TCP/IP-2], ya que tanto el cliente como el servidor se encuentran en la misma máquina.

Las ilustraciones 42, 43 y 44 muestran las funciones que en el cliente obtienen la estructura sockaddr (para las llamadas a la API Socket) para los tres protocolos de transporte empleados: TCP, UDP y SCPT independiente del nivel de red que se use. Como GetAddrInfo es independiente de IPv4 o IPv6, la diferencia fundamental entre los diferentes protocolos de nivel de transporte es el *hint*[43] que se le pasa GetAddrInfo para acotar la lista de direcciones que devolverá como respuesta.

En el caso de dirección de tipo Link-Local (la del ejemplo que nos ocupa), si no se adjunta el scope junto la dirección IPv6, la llamada a GetAddrInfo devuelve una error.

```
sub _ResolveTcp
{
    my $self=shift;
    my $rdest=shift;
    my $rlocal=shift;
    print STDOUT strftime("### %Y/%m/%e %H:%M:%S", localtime), " Log : Resolving TCP\n" if $self->config()->verb();
    print STDOUT strftime("### %Y/%m/%e %H:%M:%S", localtime), " Log : GetAddrInfo for TCP\n";
    push(@{$rdest}, $self->_GetAddr($self->config()->dest(), $self->config()->port(), {socktype => SOCK_STREAM}));
    push(@{$rlocal}, $self->_GetAddr($self->config()->source(), &NetRatConf::LPo, {socktype => SOCK_STREAM}));
    print STDOUT strftime("### %Y/%m/%e %H:%M:%S", localtime), " Log : Creating TCP socket...\n" if $self->config()->verb();
    socket(my $tsd, $rdest->[0]->{family}, $rdest->[0]->{socktype}, &NetRatConf::TCP) or die "socket: $!";
    setsockopt($tsd, SOL_SOCKET, SO_REUSEADDR, 1) or die "setsockopt: $!";
    setsockopt($tsd, SOL_SOCKET, SO_LINGER, pack('ii', 1, 0)) or die "setsockopt: $!";
    bind($tsd, $rlocal->[0]->{addr}) or die "bind : $!";
    $self->config()->set_tcp($tsd);
}
```

Ilustración 42: GetAddrInfo (TCP)

```
sub _ResolveUdp
{
    my $self=shift;
    my $rdest=shift;
    my $rlocal=shift;
    my $usd;
    my $rsd;
    print STDOUT strftime("### %Y/%m/%e %H:%M:%S", localtime), " Log : Resolving UDP\n" if $self->config()->verb();
    print STDOUT strftime("### %Y/%m/%e %H:%M:%S", localtime), " Log : GetAddrInfo for UDP\n";
    push(@{$rdest}, $self->_GetAddr($self->config()->dest(), $self->config()->port(), {socktype => SOCK_DGRAM}));
    push(@{$rlocal}, $self->_GetAddr($self->config()->source(), &NetRatConf::LPo, {socktype => SOCK_DGRAM}));
    push(@{$rdest}, $self->_GetAddr($self->config()->dest(), $self->config()->port(), {socktype => SOCK_RAW})) if ($self->config()->hole());
    print STDOUT strftime("### %Y/%m/%e %H:%M:%S", localtime), " Log : Creating UDP socket...\n" if $self->config()->verb();
    socket($usd, $rdest->[0]->{"family"}, $rdest->[0]->{socktype}, &NetRatConf::UDP) or die "socket: $!";
    if ($self->config()->hole()) {
        print STDOUT strftime("### %Y/%m/%e %H:%M:%S", localtime), " Log : Creating RAW socket...\n" if $self->config()->verb();
        if ($self->config()->six()) {
            socket($rsd, &NetRatConf::INET6, SOCK_RAW, &NetRatConf::RAW) or die "socket: $!";
        } else {
            socket($rsd, &NetRatConf::INET, SOCK_RAW, &NetRatConf::ICMP) or die "socket: $!";
            #setsockopt($rsd, &NetRatConf::IP, IP_HDRINCL, 1) or die "setsockopt: $!";
        }
    }
    bind($usd, $rlocal->[0]->{addr}) or die "bind: $!";
    $self->config()->set_icmp($rsd) if $self->config()->hole();
    $self->config()->set_udp($usd);
}
```

Ilustración 43: GetAddrInfo (UDP)

[42] Loopback: es un interfaz especial en el que los paquetes enviados son inmediatamente puestos en la cola de entrada. Este interfaz está implementado por completo en software [TCP/IP-2].
[43] Hint: es un argumento de la función GetAddrInfo.

```perl
sub _ResolveSctp
{
    my $self=shift;
    my $rdest=shift;
    my $rlocal=shift;
    print STDOUT strftime("### %Y/%m/%e %H:%M:%S", localtime), " Log : Resolving SCTP\n" if $self->config()->verb();
    print STDOUT strftime("### %Y/%m/%e %H:%M:%S", localtime), " Log : GetAddrInfo for SCTP\n";
    push(@{$rdest}, $self->_GetAddr($self->config()->dest(), $self->config()->port(), {socktype => SOCK_STREAM}));
    push(@{$rlocal}, $self->_GetAddr($self->config()->source(), &NetRatConf::LPo, {socktype => SOCK_STREAM}));
    print STDOUT strftime("### %Y/%m/%e %H:%M:%S", localtime), " Log : Creating SCTP socket...\n" if $self->config()->verb();
    socket(my $ssd, $rdest->[0]->{family}, $rdest->[0]->{socktype}, &NetRatConf::SCTP) or die "socket: $!";
    setsockopt($ssd, SOL_SOCKET, SO_REUSEADDR, 1) or die "setsockopt: $!";
    setsockopt($ssd, SOL_SOCKET, SO_LINGER, pack('ii', 1, 0)) or die "setsockopt: $!";
    bind($ssd, $rlocal->[0]->{addr}) or die "bind : $!";
    $self->config()->set_sctp($ssd);
}
```

Ilustración 44: GetAddrInfo (SCTP)

Mientras que las ilustraciones 42, 43 y 44 se muestran las funciones envoltorio a GetAddrInfo, en la ilustración 45 se muestra la llamada a GetAddrInfo en sí. La llamada es tal y como se describe en [UNIX-1], con la diferencia de que el lenguaje que se usa en esta referencia es C y el que se muestra la ilustración es Perl. Destacar que para sockets de tipo SOCK_RAW (que ofrecen acceso directo al nivel de red), no hay que pasar número de puerto, ya que no tiene sentido. La ausencia de número de puerto de transporte se muestra en el ejemplo a continuación, para el caso de que el tipo de socket sea SOCK_RAW (en caso de usar C, sería un nullptr[44]):

```perl
sub _GetAddr
{
    my $self=shift;
    my $host=shift;
    my $port=shift;
    my $hint=shift;
    my ($err, @res);
    if ($hint->{socktype} == SOCK_RAW) {
        print STDOUT strftime("### %Y/%m/%e %H:%M:%S", localtime), " Log : Getaddrinfo for SOCK_RAW\n" if $self->config()->verb();
        ($err, @res) = getaddrinfo($host); die "### $err\n" if $err;
    } else {
        ($err, @res) = getaddrinfo($host, $port, $hint); die "### $err\n" if $err;
    }
    if ($#res == 0) {
        ($err, $host, $port)=getnameinfo($res[0]->{addr}, NI_NUMERICHOST|NI_NUMERICSERV); die "### $err" if $err;
        print STDOUT strftime("### %Y/%m/%e %H:%M:%S", localtime), " Log : Resolved to $host:$port\n" if $self->config()->verb();
        return $res[0];
    } elsif ($#res > 0) {
        my $i=0;
        foreach my $ai (@res) {
            ($err, $host, $port)=getnameinfo($ai->{addr}, NI_NUMERICHOST|NI_NUMERICSERV); die "### $err" if $err;
            print STDOUT "[$i]: $host:$port\n" if $self->config()->verb();
            $i++;
        }
        print STDOUT "!!! Which Source Address do you what to use? [Choose one from list]: ";
        return $res[<STDIN>];
    } else {
        die "### Error resolving host ", $self->config()->dest
    }
}
```

Ilustración 45: Uso de GetAddrInfo en cliente NetRat

4.5 Más direcciones IPv6

```
|         n bits         |   m bits  |       128-n-m bits         |
+------------------------+-----------+----------------------------+
| global routing prefix  | subnet ID |       interface ID         |
+------------------------+-----------+----------------------------+
```

Ilustración 46: dirección IPv6 global unicast

```
|               80 bits               | 16 |      32 bits        |
+-------------------------------------+----+---------------------+
|0000.........................0000|0000|    IPv4 address     |
+-------------------------------------+----+---------------------+
```

Ilustración 47: dirección IPv6 IP4-compatible

```
|               80 bits               | 16 |      32 bits        |
+-------------------------------------+----+---------------------+
|0000.........................0000|FFFF|    IPv4 address     |
+-------------------------------------+----+---------------------+
```

Ilustración 48: dirección IPv6 IPv4-mapped

[44] Nullptr: puntero nulo (con valor 0)

4.6 Endianness[45] y direcciones IPv6

El término endiannes se refiere a cómo los datos de más de un byte se organizan en memoria. Hay dos tipos: little-endian y big-endian. Si en IPv4 había que tener cuidado con tipo de *endianness* a la hora de trabajar con protocolos de comunicaciones, con IPv6 aún más. Los datos, en las plataformas habituales (Windows, FreeBSD, Linux), se representan internamente con little-endianness, ya que es más eficiente para realizar cálculos aritméticos (primero unidades, luego decenas, centenas, etc.). Los datos que fluyen en la red viajan con formato big-endian, esto es, el primer byte que aparece en memoria es el primer byte que viaja por la red.

Ejemplo: representación de una dato de 2 bytes en máquinas big y little endian.

0x0102, en una máquina big endian se representa en memoria como [A=0x01, A+1=0x02] (donde A posición de memoria)

0x0102, en una máquina little endian se representa en memoria como [A=0x02, A+1=0x01] (donde A es posición de memoria)

En [UNIX-1] hay un fragmento de código que determina si una máquina es big o little endian, ilustración 49. Básicamente, en una variable de tipo unión con dos tipos de 2 bytes (uint16_t y un array de dos bytes), asignamos al entero un valor de 2 bytes, y vemos, mediante el array cómo se ha almacenado en memoria:

```
union {
    uint16_t s;
    char c[sizeof(uint16_t)];
} type16;

type16.s = 0x0102;

¿Cómo se representan internamente?

type16.c[0]=?
type16.c[1]=?

Para determinar si es big-endian o little-endian comprobar

If type16.c[0] = 2  && type16.c[1] = 1 then Little endian
If type16.c[1] = 2  && type16.c[0] = 1 then Big endian
```

Ilustración 49: endianness de una máquina.

Por qué es relevante en IPv6:

Las direcciones IPv4 tienes 4 bytes de longitud, por lo que se almacenan internamente en variables enteras de 32 bits, concretamente en in_addr_t, definido en <netinet/in.h>, que es un alias de uint32_t. El hecho de que la dirección IPv4 tiene sólo cuatro bytes simplifica las cosas, ya de forma natural se usa el tipo *"unsigned integer"* de 4 bytes para almacenarla en memoria:

```
struct in_addr {
    in_addr_t s_addr;    /* 32-bit IPv4 address */
}
```

Ilustración 50: representación interna de direcciones IPv4

Con IPv6 las direcciones crecen en longitud de 32 bits a 128 bits. No hay ningún tipo de dato fundamental, entendiendo por fundamental como unidad básica de almacenamiento de una computadora, que sea entero con 128 bits, por lo que hay que representar las direcciones IPv6 mediante algún tipo compuesto. La representación interna de una dirección IPv6 se hace mediante un array, pero, ¿qué tipo de array? La dirección IPv6 tiene 128 bits, ¿entonces?, ¿se representa como un array de 16 bytes?, ¿un array de 8 palabras de 2 bytes?, o ¿un array de 4 palabras de 4 bytes? La

[45] Endianess o byte-ordering, son los términos anglosajones que se refieren a como datos de más de un byte se almacenan en memoria

respuesta es que cualquier representación es posible. Es más, internamente la estructura que almacena la dirección IPv6 en binario es una unión de todos estos tipos juntos, tal y como se muestra en la ilustración 51:

```
struct in6_addr {
    union {
        uint8_t  u6_addr8[16];    /* 128-bit IPv6 address */
        uint16_t u6_addr16[8];    /* 128-bit IPv6 address */
        uint32_t u6_addr32[4];    /* 128-bit IPv6 address */
    } s6_addr;
}
```

Ilustración 51: posibles representaciones internas de direcciones IPv6"

Generalmente la API Socket abstrae al usuario de tener que manejar datos binarios de tan bajo nivel, pero en el caso en el que sea inevitable manejar estructuras IPv6 binarias es importante tener clara la representación de las tres posibles que se usa ya que las direcciones IP deben estar almacenadas con big-endianness. *¿Por qué es relevante la "endianness" en IPv6?* Un array de bytes no tiene endiannes, porque está compuesto de datos fundamentales de longitud 1 byte, pero una array de palabras de 2 o 4 bytes sí tiene endiannes. Si se quiere inicializar a mano una dirección IPv6 se puede hacer

```
const struct in6_addr naddr6 = { {
    0x3f, 0xfe, 0x05, 0x01,
    0x00, 0x08, 0x00, 0x00,
    0x02, 0x60, 0x97, 0xff,
    0xfe, 0x40, 0xef, 0xab
}};
```

ó

```
const struct in6_addr naddr6 = { {
    u6_addr32 = {
        0x3ffe0501,
        0x00080000,
        0x026097ff,
        0xfe40efab
    }
}};
```

Ilustración 52: dirección IPv6 inicializada a mano

En el ejemplo de la ilustración 52, la dirección IPv6 primero se inicializa como una array de bytes, y después se inicializa como un array de palabras de 4 bytes. Si se inicializa como array de palabras de 4 bytes hay que prestar atención a que la codificación sea big-endian (como en el ejemplo). En caso de inicializar una dirección IPv6 como un array de palabras de 2 ó 4 bytes con codificación little-endian (lo cual no tiene mucho sentido), habrá que usar las funciones htons ó htonl (dependiente de la longitud de los elementos) para traducir la codificación de la estructura binario de little-endian a big-endian.

Lo habitual es ceñirse a la API Socket (como por ejemplo getaddreinfo, que es una función nueva e independiente del protocolo de nivel de red), no siendo necesario manipular las direcciones IP a nivel de bytes.

```
struct sockaddr_in {
    uint8_t            sin_len;
    sa_family_t        sin_family;
    in_port_t          sin_port;
    struct in_addr     sin_addr;
    char               sin_zero[8];
};
```

Ilustración 53: estructura sockadd_in (IPv4).

```
struct sockaddr {
```

```
    uint8_t            sa_len;
    sa_family_t        sa_family;
    char               sa_data[14];
};
```

Ilustración 54: estructura genérica sockaddr

```
struct sockaddr_in6 {
    uint8_t            sin6_len;
    sa_family_t        sin6_family;
    in_port_t          sin6_port;
    uint32_t           sin6_flowinfo;
    struct in6_addr    sin6_addr;
    uint32_t           sin6_scope_id;
};
```

Ilustración 55: estructura sockaddr_in6 (IPv6)

```
struct sockaddr_storage {
    uint8_t            ss_len;
    sa_family_t        ss_family;
/* Implementation dependant */
};
```

Ilustración 56: estructura genérica (aumentada) sockaddr_storage

Idealmente, la forma correcta de usar la API Sockets, en lo concerniente a direcciones de máquinas remotas, es mediante DNS[46] (gethostbyname, gethostbyaddr o getaddrinfo, esta última independiente del protocolo de nivel de red usado) [UNIX-1]. Usando las funciones DNS la respuesta ya está codificada en binario y además, con las nuevas funciones, es independiente del nivel de red que se use simplificando las cosas.

```
#include <netdb.h>

int getaddrinfo(const char *hostname, const char *service,
                const struc addrinfo *hints, struct addrinfo **result);
```

Ilustración 57: función getaddrinfo [UNIX-1]

En la figura 45 se muestra la función getaddrinfo. La respuesta de la resolución DNS se devuelve mediante un puntero a un array de tipo addrinfo. En addrinfo, el campo ai_addr es un puntero a una estructura sockaddr genérica (figura 42), que la estructura binaria a usar con las funciones de la API Socket.

Cuando se trabaje directamente con literales hay que usar las funciones inet_aton, inet_addr, inet_ntoa, o mejor aún, inet_pton o inet_ntop (para que el código sea independiente del protocolo de nivel de red que se use). La función inet_pton, que traduce un literal IPv4 o IPv6 a la representación binaria correspondiente, se encarga también de que la codificación sea la correcta, esto es, la dirección IP está codificada con big-endianness. Inet_ntop hace la traducción inversa, de una estructura binaria a un literal IPv4 o IPv6.

[46] DNS: Domain Name System (DNS) es una base de datos distribuida usada por TCP/IP para mapear entre nombre de máquinas y direcciones IPv4 o IPv6

4.7 Constantes y variables globales en IPv6

A continuación se muestran algunas variables globales y constantes que se han usado en la parte práctica de este trabajo. Hay definidas algunas constantes que resultan útiles a la hora de trabajar con direcciones IPv4 o IPv6:

```
INADDR_ANY
INADDR_LOOPBACK
IN6ADDR_ANY_INIT
IN6ADDR_LOOPBACK_INIT
```

Ilustración 58: constantes para inicializar variables sockaddr_in y sockaddr_in6.

```
in6addr_any
in6addr_loopback
```

Ilustración 59: variables globales para sin6.sin6_addr

IN6ADDR_ANY_INIT es una constante que se usa para inicializar estructuras in6_addr. in6addr_any es una variable global para que sea el sistema el que seleccione la dirección IPv6 origen.

Estas constantes están codificadas en el mismo esquema que use la máquina, que generalmente será little-endianness, por lo que será necesario aplicar las funciones htonl y/o htons.

```
struct sockaddr_in6 sin6;
sin6.sin6_len = sizeof(sin6);
sin6.sin6_family = AF_INET6;
sin6.sin6_flowinfo = 0;
sin6.sin6_port = htons(23455);
sin6.sin6_addr = in6addr_any;
```

Ilustración 60: usando in6addr_any

```
struct in6_addr anyaddr = IN6ADDR_ANY_INIT;
```

Ilustración 61: usando IN6ADDR_ANY_INIT

```
#define INADDR_ANY              ((in_addr_t) 0x00000000)
#define INADDR_BROADCAST        ((in_addr_t) 0xffffffff)
#define INADDR_NONE             ((in_addr_t) 0xffffffff)

extern const struct in6_addr in6addr_any;        /* :: */
extern const struct in6_addr in6addr_loopback;   /* ::1 */

#define IN6ADDR_ANY_INIT { { { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 } } }
#define IN6ADDR_LOOPBACK_INIT { { { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1 } } }

#define INET_ADDRSTRLEN 16
#define INET6_ADDRSTRLEN 46
```

Ilustración 62: constantes in variables globales IPv4 e IPv6 en el kernel

Y, obviamente, todo esto también aplica a Windows: WinSockets (ws2ipdef.h):

```
#define IN6ADDR_ANY_INIT {0}
#define IN6ADDR_LOOPBACK_INIT {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1}
extern CONST IN6_ADDR in6addr_any;
extern CONST IN6_ADDR in6addr_loopback;
extern CONST IN6_ADDR in6addr_allnodesonnode;
extern CONST IN6_ADDR in6addr_allnodesonlink;
extern CONST IN6_ADDR in6addr_allroutersonlink;
extern CONST IN6_ADDR in6addr_allmldv2routersonlink;
extern CONST IN6_ADDR in6addr_teredoinitiallinklocaladdress;
extern CONST IN6_ADDR in6addr_linklocalprefix;
extern CONST IN6_ADDR in6addr_multicastprefix;
extern CONST IN6_ADDR in6addr_solicitednodemulticastprefix;
extern CONST IN6_ADDR in6addr_v4mappedprefix;
extern CONST IN6_ADDR in6addr_6to4prefix;
extern CONST IN6_ADDR in6addr_teredoprefix;
extern CONST IN6_ADDR in6addr_teredoprefix_old;
```

Ilustración 63: c:\Program Files (x86)\Windows Kits\8.1\Include\shared\ws2ipdef.h

4.8 Ejemplo: estructuras Sockets

A continuación se muestra, con fragmentos de código tomados de [NETLAYER], las estructuras de direcciones Socket que más se han usado a lo largo del trabajo.

```
typedef sockaddr_storage SAS;

    template<class IAX, class SAIX>
    class ActiveSock : public BaseSock {
    private:
        std::vector<SAS> sock;
        socklen_t socklen;
    public:
        int SetAddr(u_short, std::string);
        const SAIX *GetSock(u_short) const;
        int CopySock(u_short, SAIX&);
        int SetPort(u_short, uint16_t);
        socklen_t GetLength() const;
        ActiveSock(int _f=IPv4);
        ActiveSock(const ActiveSock&);
        ActiveSock& operator=(const ActiveSock&);
        virtual ~ActiveSock();
    };
```

Ilustración 64: almacenamiento de estructuras sockets en clase _NetLayer::ActiveSock

En el código desarrollado las estructuras de las direcciones del socket se almacenan en un array de elementos de tipo sockaddr_storage, porque es la estructura que mayor espacio ofrece. Esto se ha hecho así para poder usar IPv4 (sockaddr_in 16 bytes), o IPv6 (28 bytes) de forma transparente. Para curarnos en salud, la elección es sockaddr_storage que ofrece suficiente espacio para almacenar cualquiera.

En la librería NetLayer [NETLAYER] hay dos funciones para configurar la dirección (IPv4 o IPv6) y el número de puerto en el socket (pasivo o activo):

```
    template<class IAX, class SAIX>
    int ActiveSock<IAX, SAIX>::SetAddr(u_short end, std::string _addr)
    {
#ifdef DEBUG
        std::cout << "[ActiveSock::SetAddr]: " <<  _addr.c_str() << std::endl;
#endif
        addr[end] = _addr;
        memset(&sock[end], 0, sizeof(SAS));
        switch(family) {
        case AF_INET: {
            auto tmp(reinterpret_cast<sockaddr_in*>(&sock[end]));
            if (inet_pton(AF_INET, _addr.c_str(), static_cast<void*>(&tmp->sin_addr)) == 1) {
                tmp->sin_family = AF_INET;
                socklen = sizeof(sockaddr_in);
                return 0;
            } else
                return 1;
        }
        case AF_INET6: {
            auto tmp6(reinterpret_cast<sockaddr_in6*>(&sock[end]));
            if (inet_pton(AF_INET6, _addr.c_str(), static_cast<void*>(&tmp6->sin6_addr)) == 1) {
                tmp6->sin6_family = AF_INET6;
                socklen = sizeof(sockaddr_in6);
                return 0;
            } else
                return 1;
        }
        default:
            return 1;
        }
    }
```

Ilustración 65: configurando la dirección IP (IPv4 o IPv6) en ActiveSock [NETLAYER].

En la ilustración 65 se muestra la función _NetLayer::ActiveSock<IAX, SAIX>::SetAddr en la que se configura la dirección IP. Internamente las estructuras de direcciones están representadas como un array de elementos de tipo sockaddr_storage, por ello el primer paso es el correspondiente casting a sockaddr_in (en caso de trabajar con AF_INET) o sockaddr_in6 (AF_INET6), luego la función inet_pton (independiente de IPv4 o IPv6, a diferencia de sus predecesoras) nos realiza el resto del trabajo.

```
    template<class IAX, class SAIX>
    int ActiveSock<IAX, SAIX>::SetPort(u_short end, uint16_t _port)
    {
#ifdef DEBUG
        std::cout << "[ActiveSock::SetPort]: " << _port << std::endl;
#endif
        port[end] = _port;
        switch(family) {
        case AF_INET: {
            auto tmp(reinterpret_cast<sockaddr_in*>(&sock[end]));
            tmp->sin_port = htons(_port);
            return 0;
        }
        case AF_INET6: {
            auto tmp6(reinterpret_cast<sockaddr_in6*>(&sock[end]));
            tmp6->sin6_port = htons(_port);
            return 0;
        }
        default:
            return 1;
        }
    }
```

Ilustración 66: configurando el puerto en ActiveSock [NETLAYER].

En la ilustración 66 se muestra cómo se configura el número de puerto en la estructura de direcciones (_NetLayer::ActiveSock<IAX, SAIX>::SetPort). Como se explica en [UNIX-1], el número de puerto hay que almacenarlo en la codificación usada en la red (big-endiannes), por ello se usa la función htons (conversión de host a network, 2 bytes).

4.9 Checksum

A continuación se muestra como se ha usado el checksum en el trabajo elaborado mediante fragmentos de código concretos:

IPv4

[RFC1071] define como calcular de forma eficiente el checksum para IP, UDP y TCP. A continuación se muestra como se ha implementado el algoritmo para calcular el checksum en las librerías y en las aplicaciones desarrolladas.

```
sub _ChecksumIP
{
    my $self=shift;
    my $ip=shift;
    my $pkt=shift;
    my $pseudo=pack(
            'C
             C      n
             n      B16
             C      C
             n      a4
             a4',
        ($ip->{version}<<4)|$ip->{headerLength},
        $ip->{tos},
        $ip->{totalLength},
        $ip->{id},
        $ip->{offset},
        $ip->{ttl},
        $ip->{protocol},
        $ip->{checkSum},
        inet_pton(&NetRatConf::INET, $ip->{sourceIP}),
        inet_pton(&NetRatConf::INET, $ip->{destIP})
    );
    my $cksum = $self->_CheckSum($pseudo);
    substr($$pkt, 10, 2) = pack('n', $cksum);
    $ip->{checkSum} = $cksum;
}
```

Ilustración 67: función que realiza cálculo de checksum en IPv4 (cliente NetRat [NETRAT])

En la ilustración 67 se muestra cómo se calcula el checksum para paquetes IPv4 en el cliente NetRat. Es una función escrita en Perl [CAMEL] en la que se calcula el checksum de la cabecera IP. Esta función hace uso de _Checksum (función genérica que implementa el algoritmo propuesto en [RFC1071]), y que se muestra en la ilustración 68:

```
sub _CheckSum
{
    my $self=shift;
    my $pkt=shift;
    my $len_msg=length($pkt);
    my $num_short=$len_msg/2;
    my $chk=0;
    foreach my $short (unpack("S$num_short", $pkt)) {
        $chk += $short;
    }
    if ($len_msg % 2) {
        $chk += unpack("C", substr($pkt, $len_msg-1, 1));
    }
    $chk = ($chk>>16)+($chk&0xffff);
    #Note: returns data in network byte order
    return unpack("n", scalar reverse pack("n", (~(($chk>>16)+$chk)&0xffff)));
}
```

Ilustración 68: implementación en Perl del algoritmo propuesto en [RFC1071] para el cálculo del checksum

En las ilustraciones 69 y 70 se muestra cómo se ha calculado el checksum de la cabecera IPv4 en la librería [NETCRAFT]. El algoritmo es el propuesto en [RFC1071] y se muestra en la figura 58.

```
uint16_t Ip::CheckSum()
{
    header->ip_sum = 0;
    header->ip_sum = Packet<ip>::CheckSum((uint16_t*)header, header->ip_hl<<2);
    return header->ip_sum;
}
```

Ilustración 69: cálculo del checksum de la cabecera IP en la librería NetCraft [NETCRAFT]

```
template<class pheader>
uint16_t Packet<pheader>::CheckSum(uint16_t *addr, uint16_t len)
{
    int nleft=len;
    uint32_t sum=0;
    uint16_t *w=addr;
    uint16_t answer=0;
    while (nleft > 1) {
        sum += *w++;
        nleft -= 2;
    }
    if (nleft == 1) {
        *reinterpret_cast<unsigned char*>(&answer)=*reinterpret_cast<unsigned char*>(w);
        sum += answer;
    }
    sum = (sum>>16)+(sum&0xffff);
    sum += (sum>>16);
    answer = ~sum;
    return answer;
}
```

Ilustración 70: algoritmo para el cálculo del checksum [RFC1071]

IPv6

La cabecera de un datagrama IPv6 no tiene checksum. Son las capas superiores (las que hacen uso de IPv6), las responsables de calcular el checksum, incluyendo campos propios de la cabecera IPv6 en lo que se conoce como pseudo-cabecera. Esta decisión es acertada y se hizo en aras a simplificar el nivel de red y es resultado del aumento de fiabilidad en las redes actuales.

5. Internet Control Message Protocol

ICMP significa Internet Control Message Protocol [RFC792] y es un protocolo que trabaja junto a IP y lo complementa ofreciendo información sobre errores en la red y diagnosis, por lo que se ubica en el nivel de red en la torre de protocolos de TCP/IP. En las imágenes 71 y 72 se muestra el aspecto habitual de las cabeceras usadas.

```
    struct icmp {
        uint8_t icmp_type;             /* type of message, see below */
        uint8_t icmp_code;             /* type sub code */
        uint16_t icmp_cksum;           /* ones complement checksum of struct */
        union {
            unsigned char ih_pptr;              /* ICMP_PARAMPROB */
            struct in_addr ih_gwaddr;   /* gateway address */
            struct ih_idseq {           /* echo datagram */
                uint16_t icd_id;
                uint16_t icd_seq;
            } ih_idseq;
            uint32_t ih_void;
            /* ICMP_UNREACH_NEEDFRAG -- Path MTU Discovery (RFC1191) */
            struct ih_pmtu {
                uint16_t ipm_void;
                uint16_t ipm_nextmtu;
            } ih_pmtu;
            struct ih_rtradv {
                uint8_t irt_num_addrs;
                uint8_t irt_wpa;
                uint16_t irt_lifetime;
            } ih_rtradv;
        } icmp_hun;
#define icmp_pptr           icmp_hun.ih_pptr
#define icmp_gwaddr         icmp_hun.ih_gwaddr
#define icmp_id             icmp_hun.ih_idseq.icd_id
#define icmp_seq            icmp_hun.ih_idseq.icd_seq
#define icmp_void           icmp_hun.ih_void
#define icmp_pmvoid         icmp_hun.ih_pmtu.ipm_void
#define icmp_nextmtu        icmp_hun.ih_pmtu.ipm_nextmtu
#define icmp_num_addrs      icmp_hun.ih_rtradv.irt_num_addrs
#define icmp_wpa            icmp_hun.ih_rtradv.irt_wpa
#define icmp_lifetime       icmp_hun.ih_rtradv.irt_lifetime
        union {
            struct {
                uint32_t its_otime;
                uint32_t its_rtime;
                uint32_t its_ttime;
            } id_ts;
            uint32_t id_mask;
            char *id_data;
        } icmp_dun;
#define icmp_otime          icmp_dun.id_ts.its_otime
#define icmp_rtime          icmp_dun.id_ts.its_rtime
#define icmp_ttime          icmp_dun.id_ts.its_ttime
#define icmp_mask           icmp_dun.id_mask
#define icmp_data           icmp_dun.id_data
    };
```

Ilustración 71: cabecera de mensaje ICMPv4

```
    struct icmp6 {
        uint8_t icmp6_type;               /* type field */
        uint8_t icmp6_code;               /* code field */
        uint16_t icmp6_cksum;             /* checksum field */
        union {
            uint32_t icmp6_un_data32[1];  /* type-specific field */
            uint16_t icmp6_un_data16[2];  /* type-specific field */
            uint8_t icmp6_un_data8[4];    /* type-specific field */
        } icmp6_dataun;
    };
```

Ilustración 72: cabecera de mensaje ICMPv6

El protocolo IP, en cualquiera de sus versiones, es de tipo "best-effort[47]" y no-orientado a conexión. A diferencia de la red telefónica conmutada (conmutación de circuitos), en la que se establece una conexión con un ancho de banda determinado durante la llamada, Internet es una red de conmutación de paquetes en la que la información sobre origen y destino viaja en el paquete en sí, en lugar de residir en los conmutadores [TCP/IP-1-12]. La unidad básica de transporte en Internet (desde el punto de vista de extremo a extremo, esto es, en el nivel de red) es lo que se conoce como datagrama. Un datagrama es un paquete en el que la información que identifica tanto al origen como al destino viaja encapsulada en la cabecera [TCP/IP-1-12]. El protocol IP de TCP/IP es no-orientado a conexión y no confiable y lo que suele ocurrir, si hay error durante la transferencia de datos, es que los datagrams erróneos son descartados. Como bien se explica en el capítulo 8 de [TCP/IP-1-12] ("*ICMPv4 and ICMPv6: Internet Control Message Protocol*), el protocolo IP por sí mismo no ofrece la posibilidad de informar sobre si un datagrama enviado ha alcanzado satisfactoriamente el destino. IP tampoco ofrece herramientas para el diagnóstico de la red. ICMP, *Internet Control Message Protocol*, proporciona la información de errores y diagnosis que IP no es capaz. El objetivo de ICMP es ser usado junto IP para proveer información sobre errores y diagnosis en la red. ICMP no proporciona confiabilidad, esto es responsabilidad de las capas superiores al nivel de Red. Existen dos protocolos ICMP: ICMPv4 [RFC792] e ICMPv6 [RFC4443]. Además de las RFCs correspondientes, las referencias [TCP/IP-1-94], [TCP/IP-1-12], [UNIX-1] y [TANENBAUM] ofrecen información detallada sobre este protocolo.

Hay varios motivos por los que se ha prestado especial atención a ICMP. IPv4 usa ICMPv4 para el informe de errores y reportar información sobre el estado de la red. ICMPv4 es útil para, por ejemplo, actualizar la tabla de rutas en una máquina (ICMP tipo 5 "*Redirect*") o estimar la latencia entre nodos.

Mensaje de tipo 0 (Echo Reply)	Aplicación ping (informativo)
Mensaje de tipo 3 (Destination Unreachable)	Host/Protocolo (error)
Mensaje de tipo 4 (Source Quench)	Congestión (obsoleto) (error)
Mensaje de tipo 5 (Redirect)	Actualizar ruta (error)
Mensaje de tipo 8 (Echo)	Aplicación ping (informativo)
Mensaje de tipo 9 (Router Advertisement)	Alternativa a DHCPv4 (informativo)
Mensaje de tipo 10 (Router Solicitation)	Petición de *Router Advertisement* (informativo)
Mensaje de tipo 11 (Time Exceeded)	TTL excedido (error)
Mensaje de tipo 12 (Parameter Problem)	Error en cabecera o paquete (error)

Tabla 2: tipos de mensajes ICMPv4 [TCP/IP-1-12]

El caso de ICMPv6 es diferente. En IPv6, ICMPv6 no solo proporciona información de tipo "*nice-to-have*", IPv6 no puede funcionar sin ICMPv6:

1. *Autoconfiguración:* en IPv6 se prestó especial atención a la autoconfiguración de un nodo, y para ello se emplea ICMPv6 de forma intensiva. La autoconfiguración se puede llevar a cabo de dos maneras diferentes: mediante DHCPv6, como se ocurre con IPv4 (DHCPv4), lo que se conoce como configuración con estado, o bien mediante mensajes ICMPv6 provenientes del router o de otros nodos que permiten al nodo configurar su tabla de rutas y direcciones IPv6 (configuración sin estado).
2. *Neighbor discovery:* en IPv6 no se usa ARP o RARP si no que se emplea el protocolo Neighbor Discovery. El protocolo Neighbor Discovery en IPv6 [RFC4861] permite a los nodos de un enlace determinar si tienen conectividad y la resolución de direcciones (como la hace ARP en IPv4).

[47] Best-effort significa que no se garantiza que el datagrama IP alcance el destino final. Además no se reserva de antemano ningún tipo de recursos en la red (como ocurria antiguamente con las viejas redes de conmutación de circuitos o en la red telefónica).

En la ilustración 73 se muestra una traza con mensajes ICMPv6 de tipo 135 y 136 del protocolo Neighbor discovery que son el equivalente en IPv6 a los mensajes ARP en IPv4.

Realmente la capacidad de autoconfiguración y el protocolo *Neighbor Discovery* no son conceptos independientes, si no que trabajan en conjunto y se complementan.

En las tablas 2 y 3 se muestran los mensajes ICMP (en el caso de ICMPv6 no se muestran todos). Casi todos estos tipos de mensajes se han usado a lo largo de la elaboración de este trabajo, y se detallarán en secciones posteriores. En concreto, los tipos 0, 3, 8 en ICMPv4, y 1, 3, 128, 129 en ICMPv6, se han usado a nivel de código (esto es, elaborando manualmente la cabecera y checksum del paquete). Además se ha observado en la práctica cómo funciona la autoconfiguración de nodos así como el protocolo Neighbor Discovery [RFC4861] in IPv6.

Mensaje de tipo 1 (Destination Unreachable)	Host/Puerto/Protocolo (error)
Mensaje de tipo 2 (Packet too big)	Necesario fragmenter (error)
Mensaje de tipo 3 (Time Exceeded)	TTL excedido (error)
Mensaje de tipo 4 (Parameter Problem)	Error en cabecera o paquete (error)
Mensaje de tipo 128 (Echo Request)	Aplicación ping (informativo)
Mensaje de tipo 129 (Echo Reply)	Aplicación ping (informativo)
Mensaje de tipo 133 (Router Solicitation)	Autoconfiguración sin estado, ND [RFC4861]
Mensaje de tipo 134 (Router Advertisement)	Autoconfiguración sin estado, ND [RFC4861]
Mensaje de tipo 135 (Neighbor Solicitation)	ND [RFC4861]
Mensaje de tipo 136 (Neighbor Advertisement)	ND [RFC4861]
Mensaje de tipo 137 (Redirect Message)	Actualizar ruta

Tabla 3: tipos de mensajes ICMPv6 [TCP/IP-1-12]

Otro de los motivos por los que se ha trabajo intensivamente con ICMP (en cualquiera de sus sabores: ICMPv4 o ICMPv6) es que, a nivel de código, la única manera de trabajar a nivel del protocolo ICMP (*aka* nivel de red) es mediante sockets de tipo SOCK_RAW [UNIX-1], lo que permite ejercitarse a nivel práctico con ellos.

El enfoque de este trabajo siempre ha sido práctico, por lo que no se va a proceder a elaborar una síntesis de lo que ya está escrito sobre ICMP, sino que se va a presentar el trabajo y los resultados obtenidos. Cuando sea oportuno, y coherente para facilitar la lectura, se introducirán apuntes teóricos tomados de las referencias manejadas.

5.1 Ejemplo: Neighbor Discovery

La traza de la ilustración 73 se obtuvo con el objetivo de mostrar tráfico MPTCP sobre IPv6 (más sobre MPTCP en la sección de protocolos de nivel de transporte) pero se colaron los mensajes ICMPv6 del protocolo ND[48]. En relación con el protocolo ND únicamente se han capturado (en cualquiera de las trazas obtenidas durante el desarrollo de este trabajo, no sólo en la mostrada en la ilustración 73) mensajes ICMPv6 de tipo 135 y 136, pero ningún par ICMPv6 tipo 133 y 134 (Router Solicitation/Router Advertisement). El motivo es que los routers que se han usado en los escenarios de pruebas no se han configurado para asistir a los nodos en la tarea de la autoconfiguración por lo que no envían mensajes periódicos ICMPv6 de tipo 134, esto es, no se usa la capacidad de autoconfiguración sin estado. Además de no usar la autoconfiguración sin estado, en los escenarios usados en VNX, no se ha desplegado ningún servidor DHCPv6 (autoconfiguración con estado). La autoconfiguración con estado en IPv6 se lleva a cabo mediante DHCPv6 y es útil para configuración direcciones IPv6, tabla de rutas y servidor DNS a usar.

[48] ND Neighbor Discovery

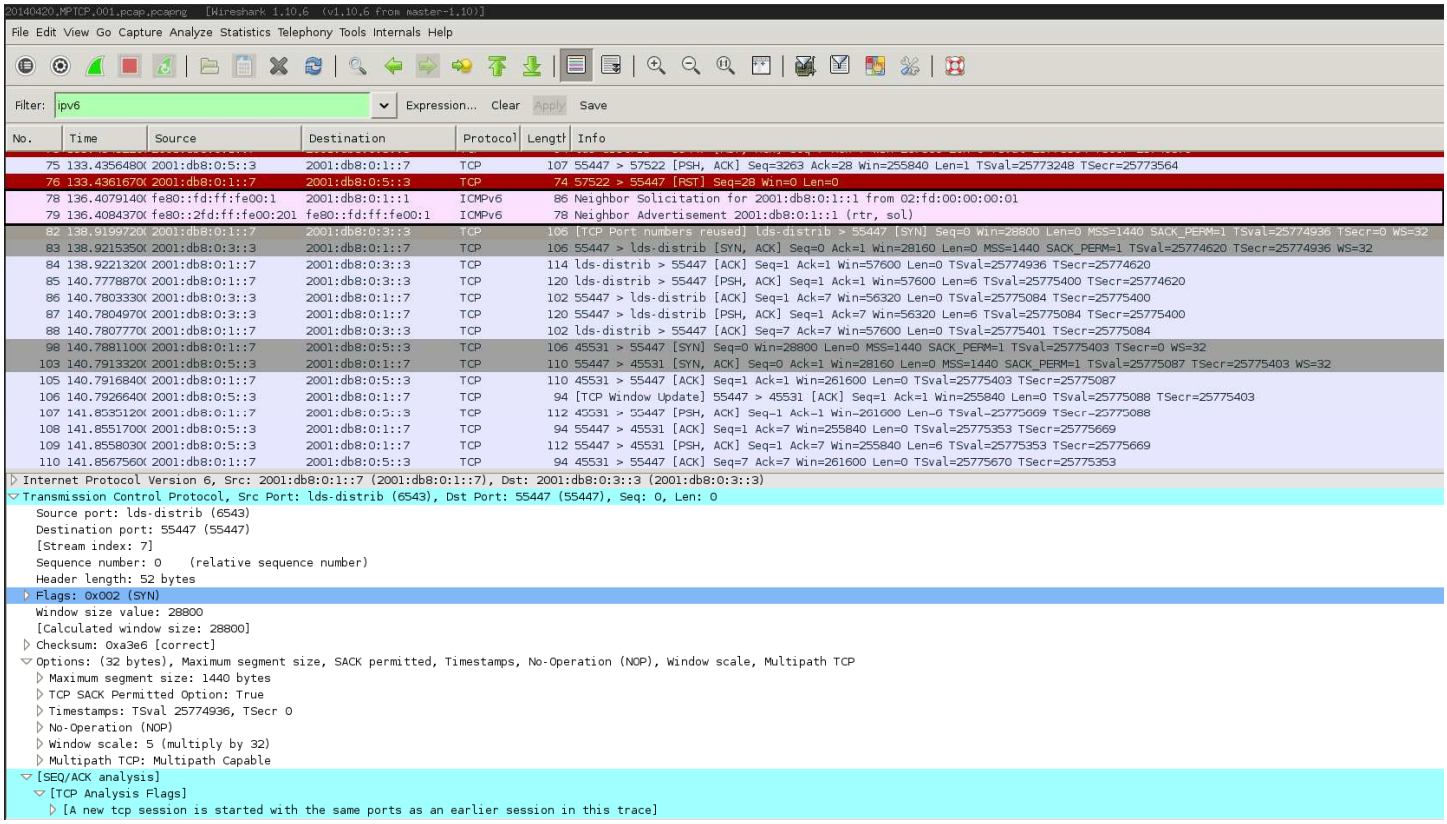


Ilustración 73: Neighbor Solicitation (ICMPv6 tipo 135) y Neighbor Advertisement (ICMPv6 tipo 136)

No sólo no se han configurado los routers para ofrecer capacidad de ayuda a la autoconfiguración de los nodos, sino que además los nodos que conforman los escenarios propuestos en este trabajo no hacen uso de la capacidad de autoconfiguración porque se han configurado de manera estática (tanto direcciones IPv6, como tabla de rutas y servidores DNS). Al configurar estáticamente las máquinas de los escenarios se evita que los nodos envíen mensajes ICMPv6 de tipo 133 (Router Solicitation) o se active el servicio DHCP-cliente. El objetivo de este trabajo es estudiar protocolos de nivel de transporte de nueva generación sobre IPv6, por lo que los escenarios se ha procurado sean lo más concretos posibles minimizando el ruido que pueden introducir mensajes de otros protocolos que están fuera del ámbito de este trabajo. Sin resolución de direcciones Ethernet a partir de la direcciones IPv6 es imposible que haya comunicación, por lo que siempre habrá mensajes ICMPv6 tipo 135 y 136. Resumiendo: el objetivo no era capturar mensajes ICMP del protocolo ND, pero IPv6 necesita ICMPv6 para funcionar, por lo que siempre habrá mensajes ICMPv6 entre los datagramas de IPv6.

Aunque el estudio de una red completa IPv6 se sale del ámbito de este trabajo, no está de más añadir un pequeño inciso. Actualmente, una red IPv6 pura completamente funcional e integrada en Internet tendría los siguientes elementos:

1. *Routers:* habría uno o más routers que, además de las tareas propias de encaminamiento, ayudarían a los nodos a configurar sus direcciones IPv6 y la tabla de rutas gracias al protocolo ND y la autoconfiguración sin estado. Los routers envían el mensaje ICMPv6 de tipo 133 a la dirección *all-host* multicas de tipo *link-local*.
2. *Servidor DHCPv6:* DHCPv6 se usa para autoconfiguración con estado (al igual que en DHCPv4 en redes IPv4) para que los clientes DHCPv6 configuren las direcciones IPv6. En un escenario en el que los routers publican periódicamente mensajes Router Advertisement con los prefijos IPv6 (por ejemplo 2003:123::/64) que los nodos deben usar, la autoconfiguración con estado no es necesaria. Aun así, es necesario que los nodos configuren la dirección IPv6 del

servidor DNS al que deben direccionar sus peticiones de resolución de direcciones. En un escenario completamente dinámico, esto es, en el que el usuario no tenga que configurar nada de forma manual, la información sobre servidor DNS a usar es proporcionada en mensajes DHCPv6 que el servidor envía a la dirección multicas de tipo link-local.

3. *DNS*: es necesario que haya un servidor DNS para servir las peticiones de los nodos. Para que la red IPv6 se integre en Internet (donde todavía la mayoría de los servicios se proporcionan sobre IPv4), es necesario usar un servidor DNS64 en conjunción con un dispositivo NAT64 (para traducir las direcciones IPv6 a IPv4, traducir IPv4 a IPv6 mapped-IPv4, y viceversa).

Actualmente los operadores de telecomunicaciones empiezan a desplegar CgNAT, en vista a mutarlos a dispositivos NAT64 cuando sea necesario.

5.2 Mensajes ICMPv4 e ICMPv6

ICMP define varios tipos de mensajes. Cada tipo de mensajes puede tener variantes conocidas mediante un código. En la ilustración 74 se muestra es formato de cabecera genérico de un paquete ICMP. El formato concreto dependerá del tipo de mensaje ICMP que se envíe (los primeros 4 bytes tienen el mismo formato para todos los mensajes). La cabera genérica mostrada en la ilustración 62 es común para ICMPv4 e ICMPv6. La cabecera ICMP, tanto v4 como v6, es de 8 bytes. La longitud del mensaje ICMP es mayor de 8 bytes, ya que además de la cabecera se suele incluir información sobre datagramas erróneos o sobre diagnosis en la zona de carga del mensaje.

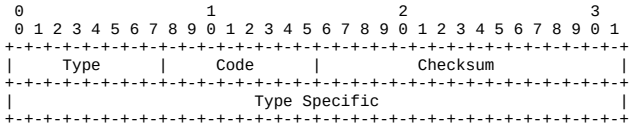


Ilustración 74: formato de mensaje ICMP

Los mensajes ICMP viajan encapsulados dentro de datagramas IP. En IPv4 el valor del campo protocolo que indica que el datagrama transporta un mensaje ICMPv4 es 1. En IPv6 el valor del campo protocolo que indica que el datagrama transporta un mensaje ICMPv6 es 58.

Tipo	*Código*	*Uso*
(0) echo reply	(0) -	Ping (NetRat/NetExample)
(3) destination unreachable	(3) port unreachable	Traceroute (NetRat)
(8) echo request	(0) -	Ping (NetRat/NetExample)
(11) time exceeded	(0) in transit	ICMP hole punching (NetRat)

Tabla 4: mensajes ICMPv4 usados a nivel de código en el código del trabajo ([NETRAT], [NETEXAMPLE])

Tipo	*Código*	*Uso*
(1) destination unreachable	(4) port unreachable	Traceroute (NetRat)
(3) time exceeded	(0) in transit	ICMP hole punching (NetRat)
(128) echo request	(0) -	Ping (NetRat/NetExample)
(129) eche reply	(0) -	Ping (NetRat/NetExample)

Tabla 5: mensajes ICMPv6 usados a nivel de programación en el código del trabajo ([NETRAT], [NETEXAMPLE])

Se procede ahora a describir en mayor detalle los mensajes de las tablas 4 y 5, puesto que en las secciones en las se presenta el código, se entrará en detalle en la codificación de estos mensajes, por lo que es necesario describirlos:

Echo request/Echo reply:

Éste es el par de mensajes ICMP más usado. El formato del mensaje echo request y echo reply es el mismo tanto en ICMPv4 como en ICMPv6, siendo las diferencias la manera en la que se calcula el checksum en primer lugar, y, los valores numéricos del campo *type* que para ICMPv4 son 8 y 0 respectivamente, y para ICMPv6 128 y 129 [TCP/IP-1-12] en segundo.

El campo *identifier* (identificador) de la cabecera es común que sea el PID[49] del proceso que envía el mensaje, mientras que el número de secuencia (*sequence number*) se incrementa en cada nuevo mensaje [UNIX-1], generalmente comenzando en cero. Como a nivel de red no existe el concepto de número de puerto, se usa el identificador para demultiplexar las respuestas en caso de que exista más de una instancia de, por ejemplo, ping enviando mensajes echo request [TCP/IP-1-12].

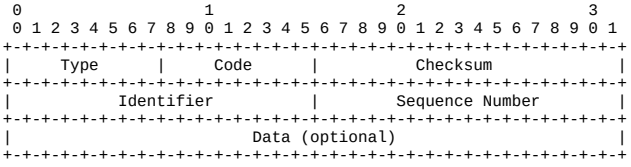


Ilustración 75: formato mensaje Echo Request o Echo Replay

Este tipo de mensajes son útiles para determinar si un equipo está operativo en la red y son usados, por ejemplo, por el programa ping [PING], existente en cualquier sistema operativo, y en las aplicaciones NetRat y NetExample [NETRAT] [NETEXAMPLE] de este trabajo.

Los mensajes ICMP *echo request* como *echo reply* se encapsulan en datagramas IP y opcionalmente transportan datos en el campo *data* del mensaje, como puede ser una marca de tiempo para estimar el RTT.[50] (Ver ilustración 76).

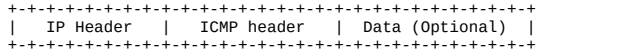


Ilustración 76: Encapsulación de mensajes ICMP echo request y replay

El uso que se hace de los mensajes ICMP *Echo request* y *reply* en el software desarrollado depende de la aplicación en sí. En NetExample [NETEXAMPLE] el uso es meramente ilustrativo, ya que el objetivo de la aplicación es mostrar cómo usar las librerías desarrolladas, y mimetiza de manera simplificada a la aplicación ping: el cliente envía un mensaje ICMPv4 o ICMPv6 *echo request* y el servidor contesta con el correspondiente *echo reply*. En el caso de NetRat [NETRAT], se emplea el mensaje ICMP *echo request* pero no su par *echo reply*. En NetRat se implementa la técnica ICMP Hole Punch [ICMPHOLE] y uno de los mensajes involucrados es el ICMP *echo request* enviado a una dirección IP que no existe, por lo que los routers no lo encaminan y nunca se obtiene el *reply*.

NOTA: La técnica ICMP Hole Punching se implementó en NetRat a finales de 2010. Esta técnica no es funcional en la práctica en IPv4, aunque la idea es original y elegante (exposición de esta técnica de NAT-traversal en [ICMPHOLE]). NetRat se ha reutilizado en este trabajo porque es una buena aplicación didáctica para ejercitarse con Sockets y TCP/IP. IPv6 abre nuevas posibilidades, y la ausencia, teórica, de dispositivos NAT hace que ICMP Hole Punch tenga más posibilidades de ser realmente funcional que actualmente en IPv4, como técnica para abrir *pin-holes* en dispositivos cortafuegos de forma autónoma. Las conclusiones sobre ICMP Hole Punching se expusieron en el trabajo de la asignatura *Temas Avanzadas de Redes de Ordenadores*, por lo que no se van a repetir en este trabajo.

[49] PID: Process IDentifier, o identificador número del proceso en el sistema operativo.
[50] RTT: Round Trip Time o tiempo que un mensaje necesita para viajar de ida y vuelta entre dos puntos en Internet.

Destination Unreachable:

Este es uno de los tipos de mensajes ICMP de informe de errores más comunes. En ICMPv4 el campo tipo tiene valor 3, mientras que en ICMPv6 el valor es 1. De las diferentes variantes de este tipo de mensaje se ha usado *Port Unreachable*, que tiene código 3 en ICMPv4 y código 4 en ICMPv6. Este mensaje se genera para informar de la no existencia de un aplicación escuchando en el puerto de destino (UDP generalmente [TCP/IP-1-12], y en el caso que aquí ocupa) al que se envía un datagrama. Esto es, UDP genera este tipo de mensajes si se recibe un datagrama para un puerto que no está a la escucha.

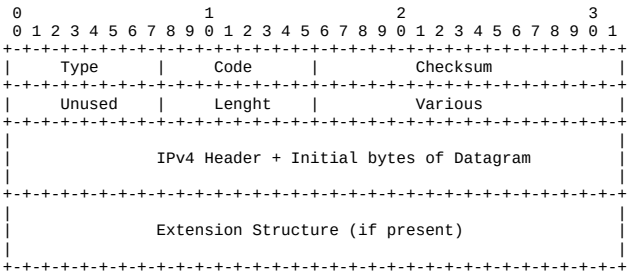


Ilustración 77: ICMPv4 Destination Unreachable

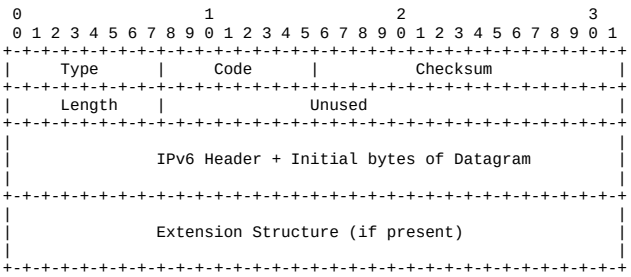


Ilustración 78: ICMPv6 Destination Unreachable

En las ilustraciones 77 y 78 se muestran las cabeceras de los mensajes ICMPv4 e ICMPv6 respectivamente de tipo *Destination Unreachable*. En el caso de ICMPv4 el campo *lenght* indica la longitud del mensaje ICMP en palabras de 4 bytes mientras que en ICMPv6 es en palabras de 8 bytes (ya que IPv4 está alineado a 32 bits mientras que IPv6 lo está a 64 bits).

En la ilustración 79 se muestra cómo se encapsula el mensaje ICMPv4 *Destination Unreachable* en un datagrama IPv4 y en la 80 un ICMPv6 en un datagrama IPv6.

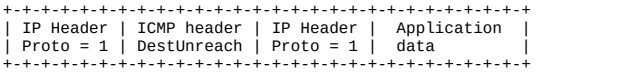


Ilustración 79: Encapsulación de mensajes ICMPv4 echo request y replay

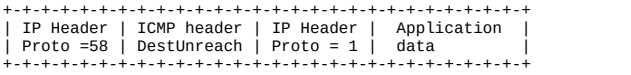


Ilustración 80: Encapsulación de mensajes ICMPv6 echo request y replay

En general la longitud de un mensaje como el de la ilustración 79 será 20 bytes de cabecera IPv4, 8 bytes de cabecera ICMPv4, 20 bytes de cabecera IPv4 más los bytes que se añadan relativos a la aplicación (con cabecera IPv4 sin opciones). En el caso de IPv6, la diferencia es que la cabecera es de 40 bytes (sin cabeceras adicionales de extensión).

La aplicación NetRat [NETRAT] implementa una función en la parte cliente que realiza el traceroute para determinar el número de saltos hasta el servidor. Es ahí donde se hace uso de este mensaje ICMP. La versión usada es un traceroute de tipo UDP. Más información [NETRAT] o [TRACEROUTE].

Time Exceeded:

En ICMPv4 el campo type tiene el valor 11 mientras que en ICMPv6 tiene el valor 3. La variante de este mensaje que se ha usado es la que tiene código 0 (tanto en ICMPv4 como en ICMPv6), *Time Exceeded In Transit*. Este mensaje se envía cuando el valor del campo *TTL* en la cabecera IPv4, o el valor del campo *Hop Limit* en la cabecera IPv6, es 0 ó 1 y es enviado por un router intermedio. Quiere esto decir que el valor del *TTL* o *Hop Limit* se ha agotado y el datagrama todavía no ha llegado al destino final.

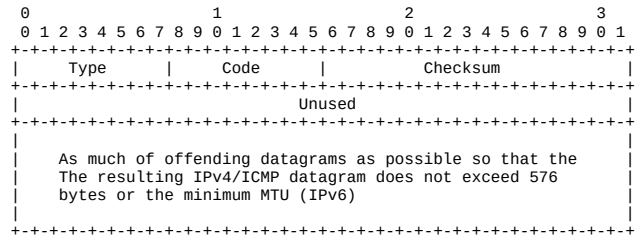


Ilustración 81: ICMP Time Exceeded in Transit ([TCP/IP-1-12])

Estos mensajes se han usado en la aplicación NetRat. En el lado del cliente para en la función que calcula la distancia hasta el servidor en número de saltos (al estilo Traceroute, en versión UDP). Además se usa en la parte del servidor para implementar la técnica ICMP Hole Punch [ICMPHOLE].

5.3 Checksum

A diferencia de lo que ocurre con el campo checksum de la cabecera IPv4 que únicamente cubre la cabecera, el campo checksum de ICMPv4 incluye tanto la cabecera ICMPv4 como la parte de datos del mensaje. En IPv6 no existe el campo checksum en la cabecera del datagrama, por lo que la forma de calcular el checksum en ICMPv6 difiere de la usada en ICMPv4, ya que en la primera se calcula a partir de una *pseudo-cabecera* que incluye los campos más relevantes de IPv6, mientras que en ICMPv4 únicamente se cubre el mensaje ICMPv4 en su totalidad.

```
struct pip6 {
    uint8_t icmp6_type;                /* type field */
    uint8_t icmp6_code;                /* code field */
    uint16_t icmp6_cksum;              /* checksum field */
    union {
        uint32_t icmp6_un_data32[1];   /* type-specific field */
        uint16_t icmp6_un_data16[2];   /* type-specific field */
        uint8_t icmp6_un_data8[4];     /* type-specific field */
    } icmp6_dataun;
};

struct picmp6 {
    pip6 pseudo;
    icmp6 header;
    char *payload;
    uint16_t hl;
    uint16_t pl;
};
```

Ilustración 82: Pseudocabecera usada en el cálculo del checksum de ICMPv6 (en C) [NETCRAFT]

En la ilustración 73 se muestra un fragmento de código del cliente NetRat en el que se prepara la pseudo-cabecera necesaria para el cómputo del checksum de un mensaje ICMPv6. En este caso el lenguaje usado es Perl.

```
sub _ChecksumICMP6
{
    my $self=shift;
    my $ip6=shift;
    my $icmp=shift;
    my $pkt=shift;
    my $length=length($$pkt);
    my $pseudo=pack(
           'a16     a16
            N       N
            a*',
        inet_pton(&NetRatConf::INET6, $ip6->{sourceIP}),
        inet_pton(&NetRatConf::INET6, $ip6->{destIP}),
        $length,
        $ip6->{next},
        $$pkt
    );
    my $cksum=$self->_CheckSum($pseudo);
    substr($$pkt, 2, 2) = pack('n', $cksum);
    $icmp->{checkSum} = $cksum;
}
```

Ilustración 83: Pseudo-cabecera para el cómputo del checksum en ICMPv6 ([NETRAT])

El algoritmo usado para el cómputo del checksum es el que se mostró en las ilustraciones 68 (escrito en Perl) ó 70 (escrito en C++) y se define en la [RFC1071].

Fe de errata: en la librería NetCraft [NETCRAFT], que se utiliza específicamente para construir datagramas IP e ICMP, el cálculo del campo checksum de la cabecera ICMPv6 está mal, ya que no se utiliza la pseudo-cabecera, si no que se hizo igual que en ICMPv4. Lo curioso de todo es que Wireshark muestra el valor del campo checksum como correcto (se ha asegurado que no es la tarjeta de red la que calcula el campo checksum, por esto se deshabilito esa optimización en el kernel), y además el paquete atraviesa cortafuegos.

5.4 Ejemplo: usando ICMPv4 e ICMPv6 en la práctica

En la asignatura del máster *Temas Avanzadas de Redes de Ordenadores,* el trabajo final se realizó sobre el artículo [ICMPHOLE], en el que se presentaba una técnica NAT-traversal autónoma[51], lo cual la hace interesante. La estrategia propuesta en [ICMPHOLE] hace uso intensivo de mensajes ICMP por lo que, aunque la técnica en sí mismo no siempre funciona, es muy buena oportunidad para trabajar con mensajes ICMPv4 e ICMPv6. En esta sección se muestra el *ICMP hole punching* una vez más, pero esta vez se muestra tanto sobre IPv4 (como ya se hizo en la asignatura de TARO), como sobre IPv6, para ilustrar el uso de mensajes ICMP confeccionados mediante la librería NetCraft o en el cliente NetRat mediante funciones que se mostrarán a continuación. La aplicación de ejemplo [NETEXAMPLE] elaborada para mostrar cómo usar las librerías también usa NetCraft para confeccionar mensajes ICMPv4 e ICMPv6, tanto en el cliente como en el servidor.

En el artículo en el que se describe esta técnica *NAT-Traversal* [ICMPHOLE], los autores hacen uso de ICMPv4. Bien es sabido que se está planteando la cuestión de eliminar los dispositivos NAT en IPv6 (cosa que aún está por ver), debido a que el balance entre las ventajas e inconvenientes de su uso es negativo. Aun así, se decidió implementar *ICMP hole punching* sobre IPv6 ya que existe la posibilidad de que sí funcione como técnica para abrir *pin-holes* en cortafuegos, y ésta es una de las novedades de este trabajo: implementar *ICMP hole punching* sobre IPv6. En concreto, se muestra esta técnica implementada en una de las aplicaciones usadas para la elaboración de este trabajo, [NETRAT]. El interfaz con la API Socket se realiza mediante la librería NetLayer [NETLAYER] y la construcción de los mensajes ICMPv4 e ICMPv6 se realiza con la librería NetCraft [NETCRAFT] en el caso del servidor. En el cliente se hace mediante funciones correspondientes.

[51] NAT-T autónomo significa que únicamente se requieren a los nodos finales como actores, a diferencia de STUN (Session Traversal Utilities for NAT) o TURN (Traversal Using Relays around NAT), que requieren el uso de servidores o nodos repetidores en el caso de TURN.

5.4.1 ICMPv4 Hole Punching

ICMP hole punching usa conceptos de la aplicación *traceroute* [TRACEROUTE], para atravesar un cortafuegos[52]. En las dos trazas que se muestran en las ilustraciones 84 a continuación, correspondientes al servidor NetRat cuando ICMP hole punching está habilitado, se aprecia el funcionamiento de *ICMP hole punching* en acción.

El servidor NetRat, cuando tiene la funcionalidad ICMP hole punching habilitada, tiene un socket de tipo SOCK_RAW enviando mensaje ICMP ECHO REQUEST y escuchando. En la ilustración 84 se muestra la traza del servidor NetRat y se aprecia que además de estar enviando mensajes ICMPv6 de tipo 0x80 con código 0x00 (128 en decimal: ECHO REQUEST), se están recibiendo mensajes de tipo 0x88 con código 0x00 (136 en decimal: Neighbor Advertisement).

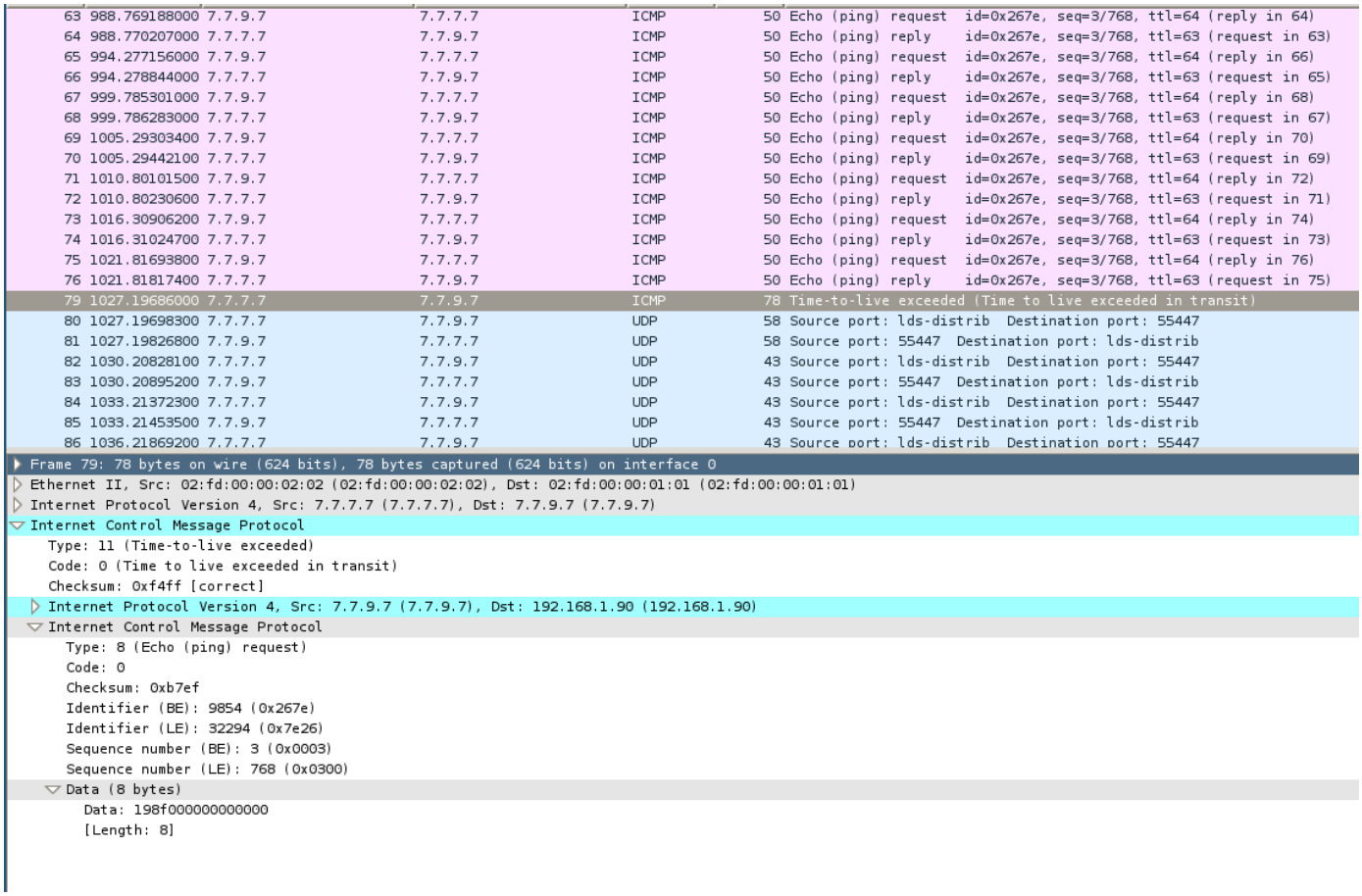


Ilustración 84: ICMP hole punch sobre ICMPv4 [ICMPHOLE]

El fragmento de código de la ilustración 85 es la implementación de la técnica NAT-T *ICMP hole punching* en el lado servidor de la aplicación NetRat [ICMPHOLE] [NETRAT] [ERNESTO81]. Esta función es un muy buen ejemplo de cómo usar las librerías NetLayer y NetCraft [NETLAYER] [NETCRAFT] para codificar y decodificar mensajes ICMP y para usar la API Socket de la manera más sencilla posible.

[52] La idea que desarrollaron los autores del *ICMP hole punching* es muy inteligente: cuando ejecutamos traceroute llegan numerosos mensajes ICMP de máquinas hacia las que no se ha enviado ningún datagrama IP pero sin embargo son capaces de atravesar un cortafuegos.

```
int SockTrav::SockIcmp4HolePunch(std::shared_ptr<_NetLayer::BaseSock> _socket)
{
    using _Configurator::ServiceConfigurator;
    int NotRecvFrom=1, n;
    char icmpBuff[MAX_BUFF];
    char pl[ICMP_PAYLOAD]={0x06,0x05,0x04,0x03};
    size_t len1, len2;
    uint16_t remPort;
    int maxfdp1;
    fd_set rset;
    timeval tv;
    logh->Log("[SockTrav::SockIcmp4HolePunch]");
    echo4.reset(new _NetCraft::Icmp);
    echo4->SetEcho(ICMP_ECHO, ICMP_ECHO_CODE, htons(ICMP_ECHO_ID), htons(ICMP_ECHO_SEQ));
    echo4->SetPayload(pl, ICMP_PAYLOAD);
    echo4->CheckSum();
    while (NotRecvFrom && !ServiceConfigurator::GetReset() && !ServiceConfigurator::GetConfig()) {
        logh->Log( "[SockTrav::SockIcmp4HolePunch]: Blocking in Select" );
        maxfdp1 = _socket->GetSd(0)+1;
        FD_ZERO(&rset);
        FD_SET(_socket->GetSd(0), &rset);
        tv.tv_sec = _socket->GetTv()*ECHO_FACTOR;
        tv.tv_usec = TV_SEL_U;
        select(maxfdp1, &rset, 0, 0, &tv);
        if (FD_ISSET(_socket->GetSd(0), &rset)){
            memset(icmpBuff, 0, MAX_BUFF);
            _socket->SetBuff(icmpBuff);
            _socket->SetN(MAX_BUFF);
            while ((n=Recvfrom(_socket))>=0) {
                logh->Hex(icmpBuff, n);
                ip4.reset(new _NetCraft::Ip(icmpBuff, n));
                len1 = n-(ip4->GetHeader()->ip_hl<<2);
                icmp4.reset(new _NetCraft::Icmp(ip4->GetPayload(), len1));
                if (icmp4->GetHeader()->icmp_type==ICMP_TIMXCEED && icmp4->GetHeader()->icmp_code==ICMP_TIMXCEED_INTRANS) {
                    ip4.reset(new _NetCraft::Ip(icmp4->GetPayload(), len1-8));
                    len2 = (len1-8)-(ip4->GetHeader()->ip_hl<<2);
                    icmp4.reset(new _NetCraft::Icmp(ip4->GetPayload(), len2));
                    if (icmp4->GetHeader()->icmp_type==ICMP_ECHO && icmp4->GetHeader()->icmp_code==ICMP_ECHO_CODE) {
                        _NetCraft::payload _tmp = icmp4->GetPayload();
                        remPort = ((*((uint16_t*)(_tmp)))<<8);
                        remPort |= (*((uint16_t*)(++_tmp)));
                        remPort &= 0xffff;
                        logh->Log("[SockTrav::SockIcmp4HolePunch]: !!! ", remPort);
                        NotRecvFrom = 0;
                        break;
                    } else break;
                } else break;
            }
        } else {
            auto fake=reinterpret_cast<char*>(echo4->GetHeader());
            logh->Log("[SockTrav::SockIcmp4HolePunch]: Sending fake echo request");
            logh->Hex(fake, ICMP_LENGTH);
            _socket->SetBuff(fake);
            _socket->SetN(ICMP_LENGTH);
            Sendto(_socket);
        }
    }
    echo4.reset(nullptr);
    logh->Log("[SockTrav::SockIcmp4HolePunch]: Exiting");
    return _socket->GetError()->Get();
}
```

Ilustración 85: ICMPv4 hole punching

La construcción del mensaje ICMPv4 Echo-Request que se usa en la función SockIcmp4HolePunch se realiza mediante la función Icmp::SetEcho que se muestra a continuación, perteneciente a la librería NetCraft [NETCRAFT]:

```
void Icmp::SetEcho(uint8_t _t, uint8_t _c, uint16_t _i, uint16_t _s)
{
#ifdef DEBUG
        std::cout << "[Icmp::SetEcho]" << std::endl;
#endif
        header->icmp_type = _t;
        header->icmp_code = _c;
        header->icmp_cksum = 0;
        header->icmp_id = _i;
        header->icmp_seq = _s;
        hlength = ICMP_HEADER_LENGTH;
        payload += hlength;
}
```

Ilustración 86: construcción del mensajes ICMPv4 Echo-Request

Por parte del cliente, para hacer que el *ICMP hole punching* funcione hay que construir un mensaje ICMPv4 Time Exceeded in Transit de tal manera que el NAT o cortafuegos del servidor piense que es enviado desde un router intermedio, y esto se hace en la función PrePacket en Perl para ICMPv4:

```
sub PrePacket
{
    my $self=shift;
    my $echo=SockOps::ICMP->new(
        type => &NetRatConf::ICMP_ECHO,
        code => &NetRatConf::ICMP_ECHO_CODE,
        checkSum => 0,
        rest => unpack('N', pack('n n', 0x267e, 0x0003)),
    );
    my $udp=SockOps::UDP->new(
        sourcePort => 10,
        destPort => 10,
        length => 0,
        checkSum => 0,
    );
    my $ipFake=SockOps::IP->new(
        version => 4,
        headerLength => 5,
        tos => 0,
        totalLength => 0,
        id => 0x7437,
        flags => 0,
        offset => 0,
        ttl => 1,
        protocol => &NetRatConf::ICMP, # &NetRatConf::UDP,
        sourceIP => $self->config()->dest(),
        destIP => &NetRatConf::FIP,
        checkSum => 0,
    );
    my $icmp = SockOps::ICMP->new(
        type => &NetRatConf::ICMP_TIMXCEED,
        code => &NetRatConf::ICMP_TIMXCEED_INTRANS,
        checkSum => 0,
        rest => 0,
    );
    my $icmpECHO=$self->_EncodeICMP($echo);
    my $udpDGRAM=$self->_EncodeUDP($udp);
    my $ipFAKE = $self->_EncodeIP($ipFake);
    my $icmpMSG=$self->_EncodeICMP($icmp);
    $echo->{data} = pack('n4', $self->config()->pport());
    my $echoh=$icmpECHO.$echo->{data};
    $self->_ChecksumICMP($echo, \$echoh);
    $udp->{data} = pack('n4', $self->config()->pport());
    my $udph=$udpDGRAM.$udp->{data};
    $self->_LengthUDP($udp, \$udph);
    $self->_ChecksumUDP($ipFake, $udp, \$udph);
    $ipFake->{data} = $echoh;
    my $ipfh=$ipFAKE.$ipFake->{data};
    $self->_LengthIP($ipFake, \$ipfh);
    $self->_ChecksumIP($ipFake, \$ipfh);
    $icmp->{data} = $ipfh;
    my $ttleith=$icmpMSG.$icmp->{data};
    $self->_ChecksumICMP($icmp, \$ttleith);
    my $outputIcmp=$ttleith;
    my $outputUdp=$udph;
    return $outputIcmp, $outputUdp;
}
```

Ilustración 87: confección artesanal del mensaje ICMPv4 Time Exceeded in Transit

```
sub _EncodeIP
{
    my $self=shift;
    my $ip=shift;
    $ip->{offset} = (substr(unpack('B8', pack('C', $ip->{flags})), 5, 3).substr(unpack('B16', pack('n', $ip->{offset})), 3, 13));
    $ip->{headerLength} = (5+length($ip->{options}));
    no warnings;
    my $pkt=pack(
            'C
             C           n
             n           B16
             C           C
             n           a4
             a4          a*',
        ($ip->{version}<<4)|$ip->{headerLength},
        $ip->{tos},
        $ip->{totalLength},
        $ip->{id},
        $ip->{offset},
        $ip->{ttl},
        $ip->{protocol},
        $ip->{checkSum},
        inet_pton(&NetRatConf::INET, $ip->{sourceIP}),
        inet_pton(&NetRatConf::INET, $ip->{destIP}),
        $ip->{options}
    );
    return $pkt;
}
```

Ilustración 88: codificación de un datagrama Ipv4 en Perl

```
sub _EncodeICMP
{
    my $self=shift;
    my $icmp=shift;
    no warnings;
    my $pkt=pack(
            'C          C
             n          N',
        $icmp->{type},
        $icmp->{code},
        $icmp->{checkSum},
        $icmp->{rest}
    );
    return $pkt;
}
```

Ilustración 89: codificación de un mensaje ICMPv4 en Perl

Las ilustraciones 88 y 89 muestran las funciones que se han escrito en Perl en el lado cliente de la aplicación NetRat para codificar datagramas IPv4 y mensajes ICMPv4.

5.4.2 ICMPv6 Hole Punching

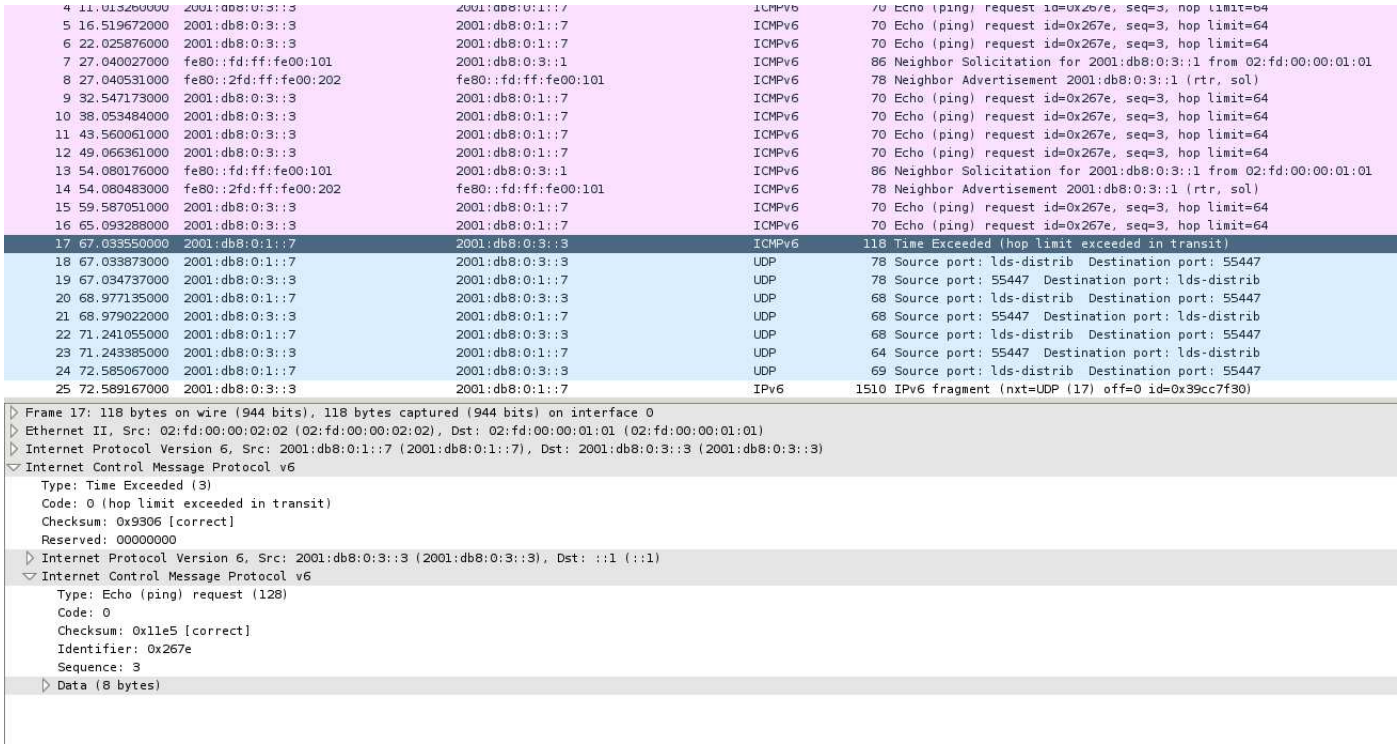


Ilustración 90: ICMP hole punch sobre ICMPv6 [ICMPHOLE]

En la aplicación NetRat se implementó *ICMP hole punching* sobre IPv6 tal y como se muestra en la función SockIcmp6HolePunch (ilustración 91). La ilustración 90 muestra esta técnica en acción sobre IPv6 y las funciones necesarias para codificar y decodificar datagramas IPv6 y mensajes ICMPv6 son las mostradas 92, 93 y 94.

He considerado importante mostrar fragmentos del código en este trabajo, ya que escribirlo ha llevado la mayor parte del tiempo empleado, pero por mucho que aquí se comente la única manera de tener una visión completa es leyéndolos directamente.

```
int SockTrav::SockIcmp6HolePunch(std::shared_ptr<_NetLayer::BaseSock> _socket)
{
    using _Configurator::ServiceConfigurator;
    int NotRecvFrom=1, n;
    char icmpBuff[MAX_BUFF];
    char pl[ICMP_PAYLOAD]={0x06,0x05,0x04,0x03};
    size_t len1, len2;
    uint16_t remPort;
    int maxfdp1;
    fd_set rset;
    timeval tv;
    logh->Log("[SockTrav::SockIcmp6HolePunch]");
    echo6.reset(new _NetCraft::Icmp6);
    echo6->SetEcho(ICMP6_ECHO, ICMP6_ECHO_CODE, htons(ICMP_ECHO_ID), htons(ICMP_ECHO_SEQ));
    echo6->SetPayload(pl, ICMP_PAYLOAD);
    echo6->CheckSum();
    while (NotRecvFrom && !ServiceConfigurator::GetReset() && !ServiceConfigurator::GetConfig()) {
        logh->Log( "[SockTrav::SockIcmp6HolePunch]: Blocking in Select" );
        maxfdp1 = _socket->GetSd(0)+1;
        FD_ZERO(&rset);
        FD_SET(_socket->GetSd(0), &rset);
        tv.tv_sec = _socket->GetTv()*ECHO_FACTOR;
        tv.tv_usec = TV_SEL_U;
        select(maxfdp1, &rset, 0, 0, &tv);
        if (FD_ISSET(_socket->GetSd(0), &rset)){
            memset(icmpBuff, 0, MAX_BUFF);
            _socket->SetBuff(icmpBuff);
            _socket->SetN(MAX_BUFF);
            while ((n=Recvfrom(_socket))>=0) {
                logh->Hex(icmpBuff, n);
                len1 = n;
                icmp6.reset(new _NetCraft::Icmp6(icmpBuff, len1));
                if (icmp6->GetHeader()->icmp6_type==ICMP6_TIMXCEED && icmp6->GetHeader()->icmp6_code==ICMP6_TIMXCEED_INTRANS) {
                    ip6.reset(new _NetCraft::Ip6(icmp6->GetPayload(), len1-8));
                    len2 = len1-48;
                    icmp6.reset(new _NetCraft::Icmp6(ip6->GetPayload(), len2));
                    if (icmp6->GetHeader()->icmp6_type==ICMP6_ECHO && icmp6->GetHeader()->icmp6_code==ICMP6_ECHO_CODE) {
                        _NetCraft::payload _tmp = icmp6->GetPayload();
                        remPort = ((*((uint16_t*)(_tmp)))<<8);
                        remPort |= (*((uint16_t*)(++_tmp)));
                        remPort &= 0xffff;
                        logh->Log("[SockTrav::SockIcmp6HolePunch]: !!! ", remPort);
                        NotRecvFrom = 0;
                        break;
                    } else break;
                } else break;
            }
        } else {
            auto fake=reinterpret_cast<char*>(echo6->GetHeader());
            logh->Log("[SockTrav::SockIcmp6HolePunch]: Sending fake echo request");
            logh->Hex(fake, ICMP_LENGTH);
            _socket->SetBuff(fake);
            _socket->SetN(ICMP_LENGTH);
            Sendto(_socket);
        }
    }
    echo6.reset(nullptr);
    logh->Log("[SockTrav::SockIcmp6HolePunch]: Exiting");
    return _socket->GetError()->Get();
}
```

Ilustración 91: ICMPv6 hole punching

```
void Icmp6::SetEcho(uint8_t _t, uint8_t _c, uint16_t _i, uint16_t _s)
{
#ifdef DEBUG
        std::cout << "[Icmp6::SetEcho]" << std::endl;
#endif
        header->icmp6_type = _t;
        header->icmp6_code = _c;
        header->icmp6_cksum = 0;
        header->icmp6_dataun.icmp6_un_data16[0] = _i;
        header->icmp6_dataun.icmp6_un_data16[1] = _s;
        hlength = ICMP6_HEADER_LENGTH;
        payload += hlength;
}
```

Ilustración 92: construcción de mensajes ICMPv6 Echo-Request

```
sub PrePacket6
{
    my $self=shift;
    my $echo6=SockOps::ICMP->new(
        type => &NetRatConf::ICMP6_ECHO,
        code => &NetRatConf::ICMP_ECHO_CODE,
        checkSum => 0,
        rest => unpack('N', pack('n n', 0x267e, 0x0003)),
    );
    my $udp6=SockOps::UDP->new(
        sourcePort => 10,
        destPort => 10,
        length => 0,
        checkSum => 0,
    );
    my $ip6Fake=SockOps::IP6->new(
        version => 6,
        ds => 0,
        ecn => 0,
        flow => 0x7437,
        length => 0,
        next => &NetRatConf::ICMP6,
        hop => 1,
        sourceIP => $self->config()->dest(),
        destIP => &NetRatConf::FIP6
    );
    my $icmp6 = SockOps::ICMP->new(
        type => &NetRatConf::ICMP6_TIMXCEED,
        code => &NetRatConf::ICMP_TIMXCEED_INTRANS,
        checkSum => 0,
        rest => 0,
    );
    my $ip6=SockOps::IP6->new(
        next => &NetRatConf::ICMP6,
        sourceIP => $self->config()->source(),
        destIP => $self->config()->dest()
    );
    my $icmp6ECHO=$self->_EncodeICMP($echo6);
    my $udp6DGRAM=$self->_EncodeUDP($udp6);
    my $ip6FAKE=$self->_EncodeIP6($ip6Fake);
    my $icmp6MSG=$self->_EncodeICMP($icmp6);
    $echo6->{data} = pack('n4', $self->config()->pport());
    my $echo6h=$icmp6ECHO.$echo6->{data};
    $self->_ChecksumICMP6($ip6Fake, $echo6, \$echo6h);
    $udp6->{data} = pack('n4', $self->config()->pport());
    my $udp6h=$udp6DGRAM.$udp6->{data};
    $self->_LengthUDP($udp6, \$udp6h);
    $self->_ChecksumUDP6($ip6Fake, $udp6, \$udp6h);
    $ip6Fake->{data} = $echo6h;
    my $ip6fh=$ip6FAKE.$ip6Fake->{data};
    $self->_LengthIP6($ip6Fake, \$ip6fh);
    $icmp6->{data} = $ip6fh;
    my $ttleit6h=$icmp6MSG.$icmp6->{data};
    $self->_ChecksumICMP6($ip6, $icmp6, \$ttleit6h);
    my $outputIcmp6=$ttleit6h;
    my $outputUdp6=$udp6h;
    return $outputIcmp6, $outputUdp6;
}
```

Ilustración 93: confección de mensaje ICMPv6 Time Exceeded in Transit

```
sub _EncodeIP6
{
    my $self=shift;
    my $ip6=shift;
    $ip6->{length} = length($ip6->{data});
    my $pkt=pack(
           'N           n
            c           C
            a16         a16',
        ($ip6->{flow}&0x000fffff)|(($ip6->{ecn}<<20)&0x00300000)|(($ip6->{ds}<<22)&0x0fc00000)|(($ip6->{version}<<28)&0xf0000000),
        $ip6->{length},
        $ip6->{next},
        $ip6->{hop},
        inet_pton(&NetRatConf::INET6, $ip6->{sourceIP}),
        inet_pton(&NetRatConf::INET6, $ip6->{destIP})
    );
    return $pkt;
}
```

Ilustración 94: codificación de datagrama IPv6 en Perl

5.4.3 SOCK_RAW

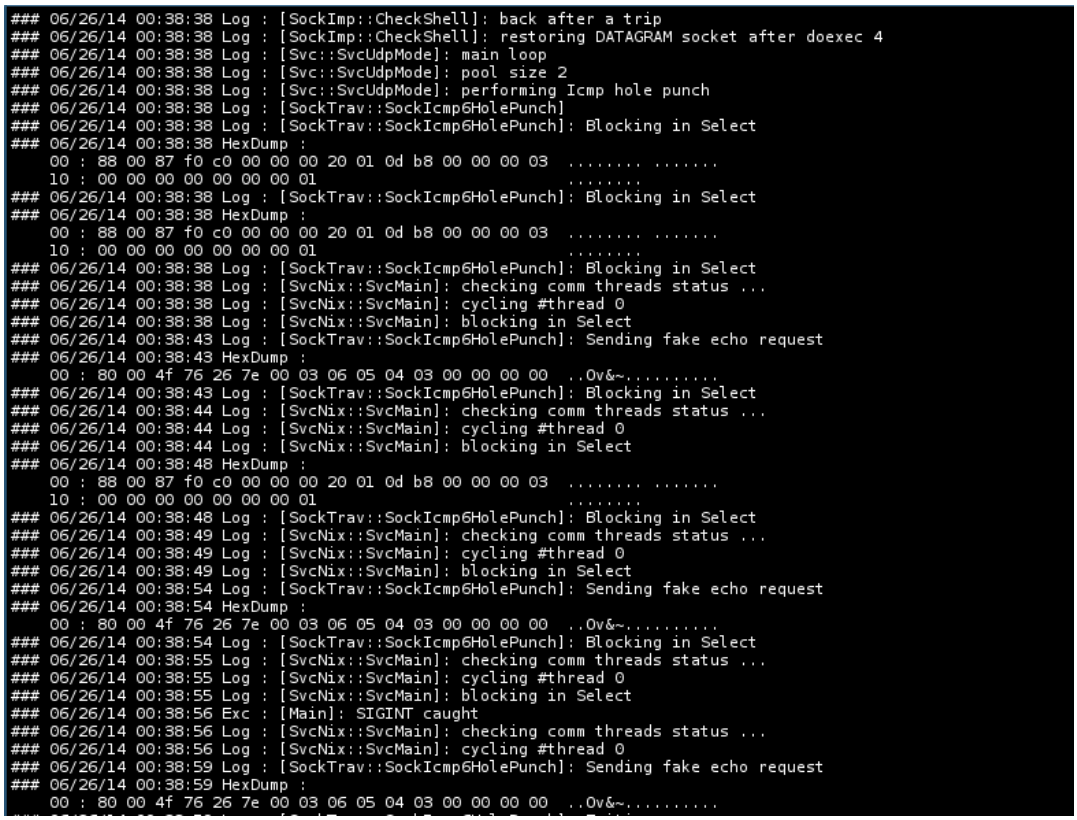


Ilustración 95: mensajes ICMPv6 capturados por servidor NetRat en socket de tipo SOCK_RAW

La única forma de programar una técnica como la mostrada aquí, en la que se hace uso de datagramas IP y mensajes ICMP, es mediante sockets de tipo SOCK_RAW. A la hora de enviar los mensajes ICMP Echo no ha sido necesario construir la cabecera IP mediante la opción IP_HDRINCL, pero es nuestra responsabilidad el calcular correctamente el checksum del mensaje ICMP. De forma análoga, a la hora de construir el mensaje ICMP Time Exceeded in Transit, es necesario el calcular correctamente el checksum de los mensajes ICMP Time Exceeded y los que conforman la zona de datos del mensajes, que son el datagrama IP y el mensaje Echo. Como se muestra en las funciones SockIcmp4HolePunch y SockIcmp6HolePunch el envío del mensaje ICMP Echo y la recepción se multiplexa comediante la función select [UNIX-1]. Para la recepción se usa une socket de tipo SOCK_RAW en la función Recvfrom de la librería NetLayer [NETLAYER].

Un socket de tipo SOCK_RAW a la escucha tiene un comportamiento que varía de una plataforma a otra, pero hay una serie de puntos que son aplicables a todas. Por ejemplo, como el socket de tipo RAW se creó con un campo *protocol* igual a ICMP, y se invocó *bind* con una dirección local, todos los mensajes ICMP que lleguen al kernel dirigidos a la dirección local usada en el socket se van a pasar al nivel de aplicación [UNIX-1], y eso es lo que efectivamente se aprecia en la ilustración 95. En dicha ilustración se aprecian mensajes con campo tipo igual a 0x88 (Neighbor Advertisement) que se pasan a la aplicación.

5.5 Direcciones IPv6, mensajes ICMPv6 y el protocolo Neighbor Discovery

Durante la elaboración del ejemplo de la sección anterior se observó cómo interactúan un ordenador personal y un router ambos con IPv6 habilitados, lo cual queda recogido en la ilustración 96. En concreto el ordenador hace uso de la dirección multicast especial de *Solicited-node address range* ff02::1:ffxx:xxxx (en concreto ff02::1:ff9a::bcaa). La RFC del protocolo Neighbor Discovery [RFC4861] pasa superficialmente sobre la utilidad de este tipo de direcciones multicast diciendo simplemente que minimiza el número de direcciones en nodos que tienen diferentes interfaces con diferentes proveedores. En la traza que se ha obtenido se observa que se entremezclan mensajes ICMPv6 de tipo 0x87 con direcciones de link-local con mensajes con la dirección multicast: cuando el router recibe el mensaje ICMPv6 0x87 dirigido a la dirección *solicited-node address* contesta con su dirección link-local:

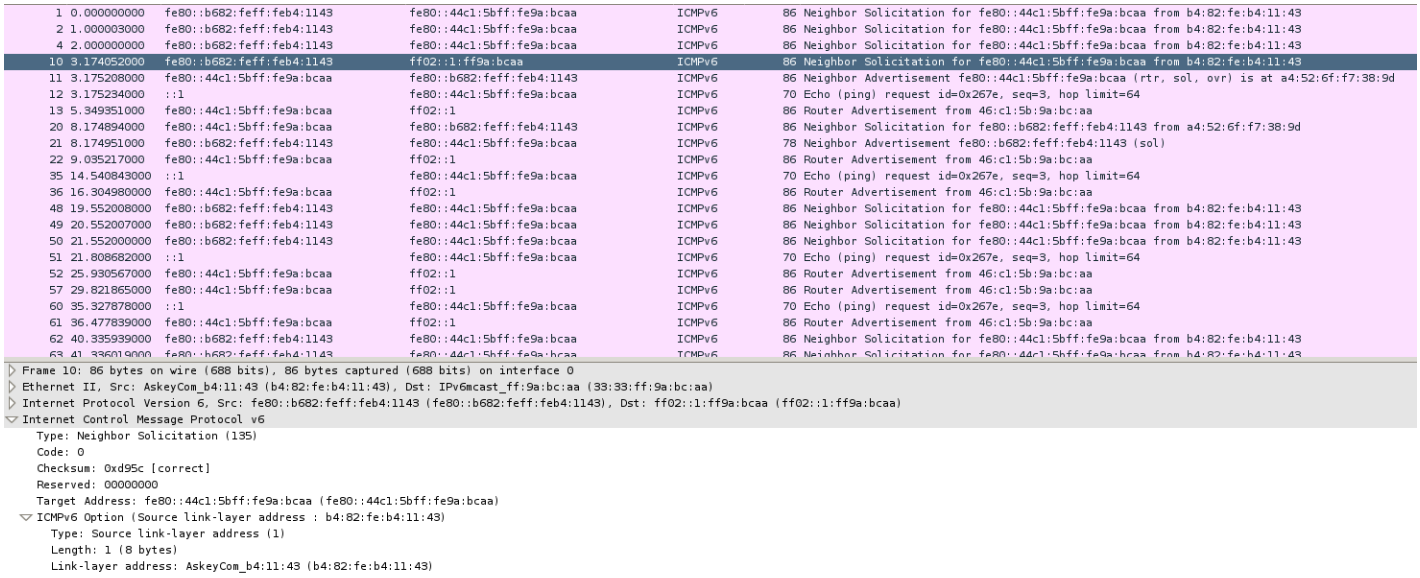


Ilustración 96: protocolo ND y dirección IPv6 “solicited node address”

6. Multipath TCP

El objetivo original de este trabajo era estudiar un poco más el nivel de transporte de la torre de protocolos TCP/IP. TCP y UDP han sido dos protocolos que han hecho, hacen y seguirán haciendo un buen trabajo. TCP como protocolo de gran complejidad, confiable, orientado a conexión, y UDP como lo contrario, un protocolo para el intercambio de datagramas sin ningún tipo de fiabilidad y no orientado a conexión. No existe un motivo flagrante, como en el caso del agotamiento de direcciones IPv4, que obligue a forzar un cambio de algo que ha funcionado correctamente desde que se introdujo, pero los dispositivos que usan Internet han evolucionado mientras que la red de redes simplemente se ha ido manteniendo y parchando a lo largo de estos años. Si en su día, probablemente, la mayor complejidad técnica residía en los niveles de red y transporte del modelo de referencia TCP/IP, hoy en día gracias a, o por culpa de, la revolución de la Sociedad de la Información, la complejidad del nivel de aplicación del software es, de largo, mucho mayor que la del resto de los niveles de la torre de protocolos. Coger las técnicas y algoritmos de bajo nivel, esto es, que están *pegando* al nivel de transporte, y moverlos al nivel de transporte para aligerar el código del nivel de aplicación y estandarizarlos conveniente, en aras a incrementar interoperabilidad, puede ser una buena idea. En esta dirección se va a experimentar con MPTCP como evolución de TCP añade la funcionalidad multi-camino en lo que resta de sección.

Los cambios necesarios para hacer que una aplicación TCP bien escrita haga uso de MPTCP son nulos. En el caso concreto de este trabajo, una vez que la aplicación estaba en el punto de partida deseado (más o menos correctamente escrita y soportando IPv4 y/o IPv6) MPTCP trabajo de forma natural una vez que se habilitó en el kernel del sistema operativo. El único cambio necesario es habilitar la funcionalidad MPTCP en el sistema operativo correspondiente y probablemente modificar la tabla de rutas. Esto es así porque la API para programar una aplicación que use MPTCP es la misma que se usaría para escribirla en TCP. En la ilustración 97 se muestra una traza de MPTCP sobre IPv6. El establecimiento de conexión es el típico de TCP (*three-way-handshake*), que tiene lugar múltiples veces: en primer lugar cuando el cliente invoca la función *connect* (para conectarse al servidor) y luego cuando MPTCP considera que ha llegado el momento de intentar establecer caminos alternativos, cosa que suele ocurrir casi inmediatamente a continuación del establecimiento de la primera conexión y cuando comienza a fluir datos.

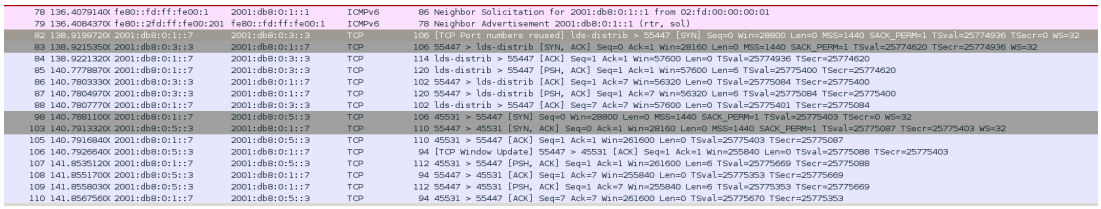


Ilustración 97: captura de protocolo MPTCP sobre IPv6

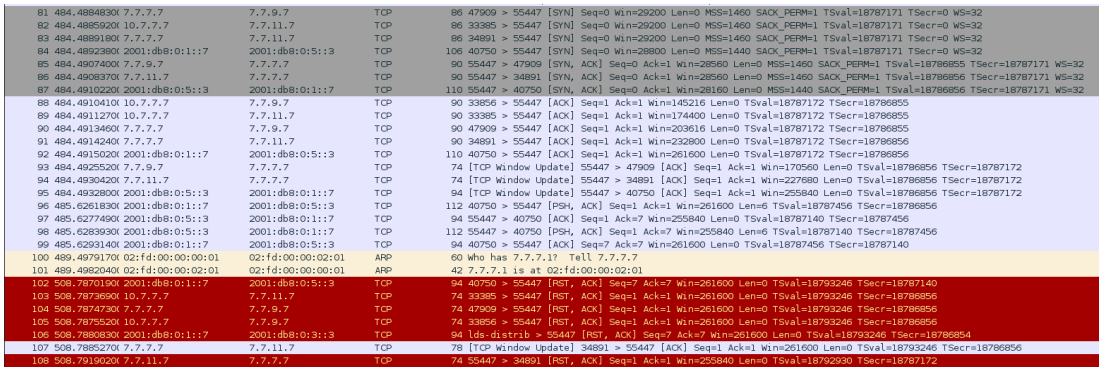


Ilustración 98: traza MPTCP sobre dual-stack

En la traza de la figura 97 se muestra otra captura de MPTCP en acción usando la aplicación NetRat. En esta traza se observa cómo funciona MPTCP sobre *dual-stack*, estableciéndose conexiones por múltiples rutas (tanto IPv4 como IPv6) de manera inmediata. En este caso la conexiones se inician siempre desde el lado cliente hacia el lado servidor, pero puede ocurrir, como se explicó en el ejemplo de la sección 3.2, que la conexión de un *path* de establezca de forma inversa (desde el lado servidor hacia el lado del cliente). Hay una pequeña fase de transferencia de datos para finalmente abortar las conexiones.

El diagrama de estados para las conexiones de los múltiples caminos que MPTCP puede escoger es el mismo que el de una conexión TCP. Como bien se explica en cualquier libro sobre TCP/IP, cuando una aplicación realiza un cierre activo (*activo-close*) de la conexión TCP, ésta debe permanecer en estado TIME_WAIT por un periodo de tiempo igual al doble de MSL[53], en caso de que sea necesario retransmitir el último ACK [TCP/IP-1-94]. La mejor explicación sobre el porqué de este estado la leí en el libro *Computer Networks* [TANNENBAUM] en la sección 6.3.2 (*Connection Release*). Por defecto, mientras que la conexión TCP permanece en estado TIME_WAIT el puerto que se ha usado queda fuera de servicio. Esto se puede *hackear* usando la opción SO_REUSEADDR [TCP/IP-1-94], la cual permite que el mismo puerto se vuelva a usar (en la función *bind*), pero aun así, no es posible establecer una conexión (usando la función *connect*). La manera en la que en TCP se evita el estado TIME_WAIT es mediante un cierre abrupto (*abruptive release*): en lugar de la secuencia de 4 paquetes FIN/ACK, se envía un segmente con el flag RST (reset) activado, por lo que la conexión pasa a estado CLOSED inmediatamente, siendo posible reutilizar el mismo puerto inmediatamente. En la traza de la ilustración 79 se muestra como todos los caminos MPTCP se cierran mediante segmentos TCP con el flag RST activado. Esto es así porque en el cliente (escrito en Perl) se ha seleccionado la opción SO_LINGER (1,0) para evitar el estado TIME_WAIT, y MPTCP se comporta como cabría esperar con TCP. En la ilustración 80 se muestra el fragmento de código extraído del cliente NetRat responsable de que esto ocurra así:

```
setsockopt($tsd, SOL_SOCKET, SO_REUSEADDR, 1) or die "setsockopt: $!";
setsockopt($tsd, SOL_SOCKET, SO_LINGER, pack('ii', 1, 0)) or die "setsockopt: $!";
```

Ilustración 99: opción SO_REUSEADDR y SO_LINGER

En la ilustración 80 también se ha observado un segmento ACK de tipo *window update* enviado desde el cliente hacia el servidor. En TCP se usan estos mensajes para actualizar el valor de la *ventana disponible*, y es parte del mecanismo de ventana deslizante en TCP. Simplemente quiere esto decir que los múltiples caminos que MPTCP establece se comportan de la misma manera que cabría esperar con TCP.

6.1 Ejemplo: MPTCP en acción

En la sección 3.2 de este documento se hizo un adelanto sobre MPTCP con un ejemplo, dentro del contexto de MPTCP e IPv6 (*dual-stack* concretamente). En esta sección se va describir en más detalle el escenario y lo que ha sido necesario modificar para poder usar MPTCP. A nivel de código, para poder usar MPTCP no ha habido que realizar ningún cambio. MPTCP usa exactamente el mismo API que TCP, por lo que cualquier aplicación que esté bien escrita para TCP puede usar MPTCP, lo cual simplifica enormemente el trabajo. De la misma forma que migrar la aplicación NetRat para soportar IPv4 y/o IPv6 de forma coherente requirió 4 meses de trabajo (y por este motivo la sección de protocolos de red, IP e ICMP, es extensa acorde al tiempo invertido), usar MPTCP fue inmediato. Pero para poder ver MPTCP en acción es necesario disponer de varias rutas por las que los segmentos puedan alegremente, y usando el interfaz localhost esto no ocurre. Pero en la UPM, sus profesores aparte de escribir artículos se dedican a realizar *trabajo* de *campo,* como crear *start-ups* o elaborar herramientas que sean útiles en el día a día, y gracias a eso se dispone de *Virtual Network over LinuX* [VNX]. VNX ha resultado clave para poder elaborar

[53] MSL: Maximum Segment Lifetime

un escenario realista que esté compuesto de elementos finales y elementos enrutadores. En la sección 5, en la que se describen los escenarios virtualizados usados a lo largo del trabajo, se volverá a hablar de VNX. Para lo que nos ocupa ahora, basta con mencionar que el escenario está compuesto de un cliente con un interfaz de red con IPv4 e IPv6 (y otro interfaz *secreto mediante el cual me conecto con SSH*), un servidor con dos interfaces con IPv4 e IPv6 (más otro secreto para SSH) y un router con tres interfaces todos con IPv4 e IPv6: uno hacia el cliente y dos hacia el servidor (más el correspondiente interno de gestión para SSH).

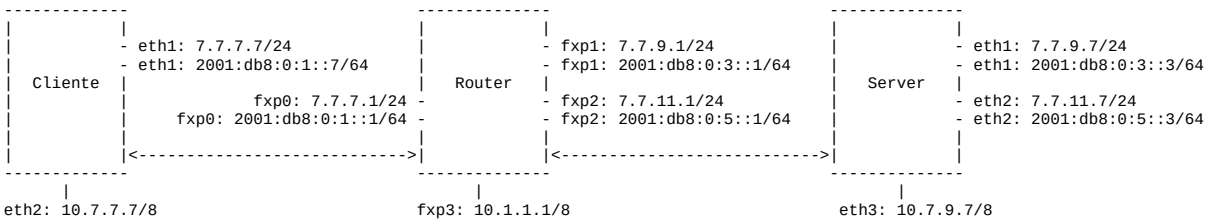


Ilustración 100: diagrama de los componentes del escenario

La ilustración 81 muestra el diagrama de los componentes que forman parte del ejemplo. El cliente tiene dos interfaces (uno de los cuales, eth2, se usa para conectarse mediante SSH desde la máquina host). El cliente y el servidor se comunican a través de un router que tiene cuatro interfaces: uno hacia el cliente, fxp0.0, dos hacia el servidor, fxp1.0 y fxp2.0, y otro de gestión para conectarse mediante SSH, fxp3.0. El servidor tiene tres interfaces: eth1, eth2 y eth3 (este último para SSH). Ambos, cliente y servidor, usan un kernel Linux [WHEEZY] con MPTCP habilitado. El servidor usa OS de Juniper [JUNOS] virtualizado, lo que se conoce como “Aceituna” [OLIVE]. ¿Cuántas rutas posibles hay desde el cliente hasta el servidor? ¿Una?, ¿dos?, ¿tres?, ¿mil?, ¿un millón?, ¿infinitas? Las posibles rutas desde el cliente hacia el servidor son seis bidireccionales tanto para IPv4 e IPv6 en este escenario, lo que quiere decir que, como los hosts usan *dual-stack* se podrían ver hasta 24 segmentos TCP con flag SYN activado (por cada sentido del camino y por cada protocolo). Obviamente la mitad de las rutas son redundante, ya que TCP (y MPTCP por ende) es bidireccional, por lo que no todas las peticiones de establecimiento de conexión se completarán. En el ejemplo de la sección 3.2, en concreto en la ilustración 12, ya se hizo un adelanto detallado sobre MPTCP y se observaron 11 segmentos TCP con flag SYN activo de los cuales no todas se completaron, y de las que lo hicieron, no todas fueron usadas debido a que la carga de datos transferida no era excesivamente elevada y no había congestión ni errores en la red. A continuación se procede a describir detalladamente todos los pasos y eventos que han tenido lugar en el establecimiento de conexión y transferencia de datos MPTCP.

```
sub _ResolveTcp
{
    my $self=shift;
    my $rdest=shift;
    my $rlocal=shift;
    print STDOUT strftime("### %Y/%m/%e %H:%M:%S", localtime), " Log : Resolving TCP\n" if $self->config()->verb();
    print STDOUT strftime("### %Y/%m/%e %H:%M:%S", localtime), " Log : GetAddrInfo for TCP\n";
    push(@{$rdest}, $self->_GetAddr($self->config()->dest(), $self->config()->port(), {socktype => SOCK_STREAM}));
    push(@{$rlocal}, $self->_GetAddr($self->config()->source(), &NetRatConf::LPo, {socktype => SOCK_STREAM}));
    print STDOUT strftime("### %Y/%m/%e %H:%M:%S", localtime), " Log : Creating TCP socket...\n" if $self->config()->verb();
    socket(my $tsd, $rdest->[0]->{family}, $rdest->[0]->{socktype}, &NetRatConf::TCP) or die "socket: $!";
    setsockopt($tsd, SOL_SOCKET, SO_REUSEADDR, 1) or die "setsockopt: $!";
    setsockopt($tsd, SOL_SOCKET, SO_LINGER, pack('ii', 1, 0)) or die "setsockopt: $!";
    bind($tsd, $rlocal->[0]->{addr}) or die "bind : $!";
    $self->config()->set_tcp($tsd);
}
```

Ilustración 101: resolución de dirección en MPTCP (exactamente igual que en TCP). Cliente NetRat.

Cuando un cliente quiere conectarse, mediante cualquier protocolo de nivel de transporte o de red, con un servidor, la manera recomendada de hacerlo es mediante las funciones de resolución de direcciones de nueva generación (*getaddrinfo* y su complementaria *getnameinfo*) ya que son independientes del protocolo de nivel de red que se use. Para este ejemplo se ha usado la aplicación NetRat [NETRAT]. En la ilustración 82 se muestra cómo el cliente obtiene la estructura

sockaddr_in (IPv4) o sockaddr_in6 (IPv6) necesaria para conectarse con el servidor, y para MPTCP se hace exactamente de la misma manera que para TCP. En concreto, la función *_ResolveTcp* crea dos arrays de estructuras sockets: una con la estructura con la información local (a usar con *bind*) y otra con la estructura con la información del host remoto (a usar con *connect*):

```
push(@{$rdest}, $self->_GetAddr($self->config()->dest(), $self->config()->port(), {socktype => SOCK_STREAM}));
push(@{$rlocal}, $self->_GetAddr($self->config()->source(), &NetRatConf::LPo, {socktype => SOCK_STREAM}));
```

Esta función usa otra función, _GetAddr, que es la que invoca getaddrinfo y getnameinfo. En Perl, getaddrinfo devuelve una lista de estructuras socket. Getaddrinfo lanza el resolver DNS y devuelve una lista de estructuras socket con las entradas DNS que haya obtenido:

```
($err, @res) = getaddrinfo($host, $port, $hint);
```

A modo informativo comentar que la función complementaria de getaddrinfo es getnameinfo, que se una para, a partir de una dirección IP o un número de puerto obtener el FQDN de la máquina el nombre del servido asociado a puerto en cuestión. En el cliente de NetRat se usa para realizar una petición inversa DNS e imprimir por pantalla los resultados. Esta técnica es un mecanismo básico de seguridad para tener una mínima certeza de que el resultado devuelto por la petición DNS es legítima. A continuación se muestra por pantalla el fragmento de código tomado del cliente que realiza esta operación:

```
my $i=0;
foreach my $ai (@res) {
    ($err, $host, $port)=getnameinfo($ai->{addr}, NI_NUMERICHOST|NI_NUMERICSERV); die "### $err" if $err;
    print STDOUT "[$i]: $host:$port\n" if $self->config()->verb();
    $i++;
}
```

En el otro extremo tenemos al servidor NetRat a la escucha. En concreto está escuchando en la dirección IPv6 :: y en el puerto 55447, pero como el kernel dispone de dual-stack aceptará cualquier tipo de petición de conexión dirigida a ese puerto, ya sea sobre IPv4 o sobre IPv6. La máquina en la se ejecuta el servidor tiene varios interfaces y el kernel tiene el protocolo MPTCP habilitado. En la ilustración 83 se observa el resultado de ejecutar el comando *lsof –i -n* (que muestra por pantalla los descriptores de ficheros asociados a Internet en el host) y efectivamente se verifica que tenemos una aplicación el [::]:55447.

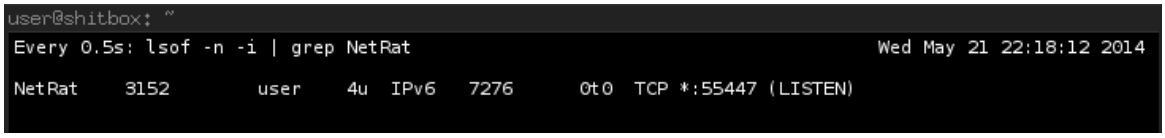


Ilustración 102: servidor NetRat a la escucha de peticiones TCP con MPTCP habilitado en el kernel.

Ilustración 103: el terminal de izquierda muestra el comando que se va a ejecutar en el cliente para conectarse con el servidor. El terminal de la derecha muestra el comando que se va a ejecutar en el router que separa el cliente del servidor para analizar el tráfico que circule por éste.

En el terminal de la izquierda en la ilustración 85 se muestra la salida por pantalla del cliente NetRat. El comando ejecutado establece como dirección remota 7.7.9.7, que es una de las dirección IPv4 de uno de los interfaces de la máquina en la que se encuentra el servidor. El puerto es el 55447 y el protocolo TCP. Lo primero que se pide el cliente es especificar cuál de las direcciones locales deseamos usar como dirección de origen: 7.7.7.7, 10.7.7.7 o 127.0.0.1. Y se procede a enviar una petición de conexión hacia el servidor que queda captura en el terminal de la derecha (en el que se está ejecutando la opción sniffer en el router JUNOS del escenario). Aunque se está usando TCP con un kernel que soporta MPTCP, en este punto del ejemplo únicamente se ha apreciado una petición de conexión con la tupla (7.7.7.7:6543, 7.7.9.7:55447).

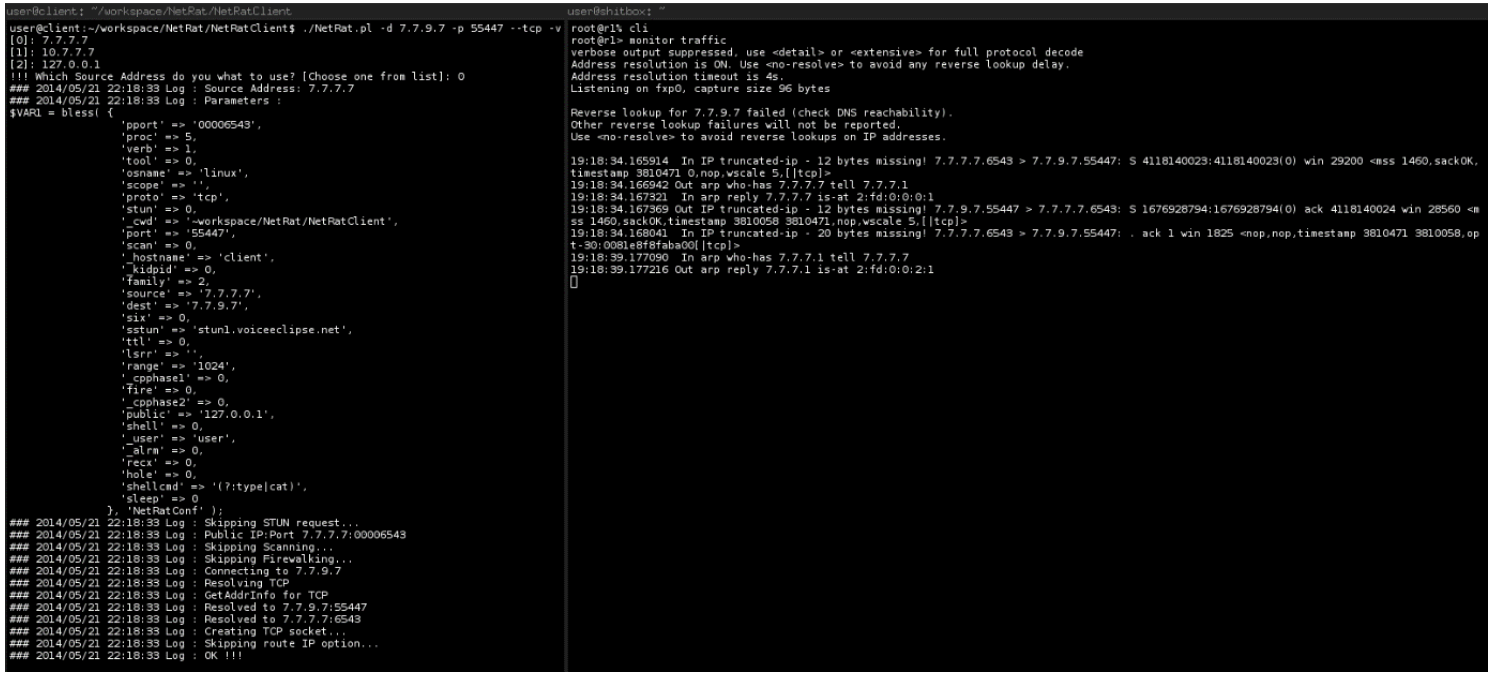


Ilustración 104: en el terminal de la izquierda se muestra la traza del cliente una vez que se ha conectado con el servidor NetRat. En el terminal de la derecha se muestra el tráfico que ha sido capturado en el router del escenario.

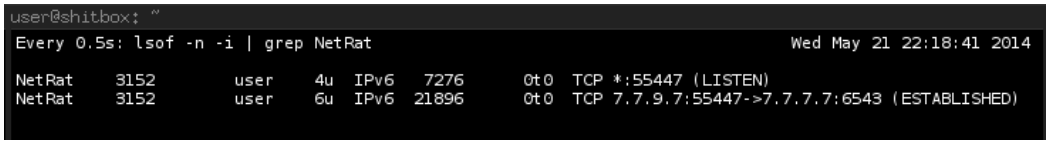


Ilustración 105: terminal en el servidor que muestra el descriptor de fichero del socket a la escucha y el descriptor de fichero del socket conectado con el cliente.

En la ilustración 86 se muestra la salida por pantalla del comando *lsof –i –n* cuando el cliente ha establecido la conexión con el servidor. Obsérvese que el comando muestra que el protocolo de nivel de red es IPv6 pero las direcciones que aparecen son IPv4. Esto es gracias al uso de *dual-stack* junto con la dirección IPv6 [::]. En este punto tenemos establecida una conexión TCP entre el cliente y el servidor por un único camino.

En la ilustración 88 se muestra en el terminal de la izquierda (correspondiente al cliente NetRat) lo que ocurre al teclear *"_sid_"*: devuelve un terminal remoto en el servidor, pero si encriptación TLS, en este punto, como se aprecia en el *shellbang* que se imprime por pantalla. Este intercambio de datos entre el cliente y el servidor (el cliente envía un segmento TCP con la cadena "_sid_", el servidor redirecciona el descriptor asociado al socket conectado a STDIN y STDOUT, y devuelve el *shellbang* al cliente) dispara a MPTCP, que comienza a establecer conexiones por múltiples caminos (tanto IPv4 como IPv6, como en ambas direcciones, esto es, tanto del cliente hacia el servidor como del servidor hacia el cliente), lo cual queda recogido en el terminal de la derecha en la ilustración 88, que es el correspondiente al router *sniffing* tráfico entre cliente y servidor. El ejemplo en sí es aparentemente sencillo, por no decir, un cliente y un servidor conectándose entre ellos en máquinas con MPTCP habilitado y realizando una transferencia de datos suficiente para que la funcionalidad MPTCP se dispare, pero las apariencias engañan:

Primero, los sockets se identifican mediante un número entero, lo que se conoce como descriptor de socket. En el caso de TCP (y por ende MPTCP), hay dos tipos de sockets: pasivos (o a la escucha) o activos (o conectados). En la ilustración 86, correspondiente a la salida por pantalla del comando *lsof –i –n* en el servidor, se observan ambos sockets. El socket con descriptor 4 es el pasivo (o a la escucha mediante *accept*), y el socket con descriptor 6 (el conectado al cliente). En TCP un descriptor de socket que esté conectado identifica de forma unívoca dos direcciones IP y dos números de puerto. El socket de descriptor 6 identifica a una tupla de ip-puertos diferente en las ilustraciones 86 y 87, y esto hace que nos plateemos la siguiente pregunta: ¿cuántos descriptores de fichero se usan en MPTCP? Aparentemente uno para el socket conectado, pero, ¿cómo se identifican internamente la tuplas ip-puerto? *Segundo,* como se comentó en el ejemplo de la sección 3.2, el tráfico entre cliente y servidor puede seguir caminos que involucran los interfaces de gestión de las máquinas virtuales que se suelen usa para conectarse mediante SSH. Esos caminos no aparecen completamente representados en las ilustraciones 88 y 89.

Parece ser que el comando *lsof* sólo muestra por pantalla la última de las conexiones que se ha establecido (en la traza mostrada en la ilustración 88, la identificada por la tupla [2001:db8:0:1::7]:56449, [2001:db8:0:3::3]:55447, que es la última conexión establecida por uno de los posibles caminos desde el cliente hasta el servidor en este ejemplo).

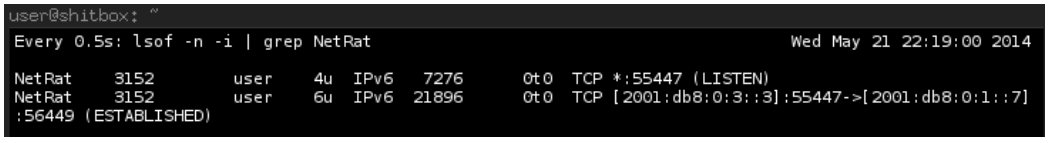


Ilustración 106: terminal en el servidor que muestra el descriptor de fichero del socket a la escucha y otro descriptor de fichero de otra conexión MPTCP, ésta IPv6.

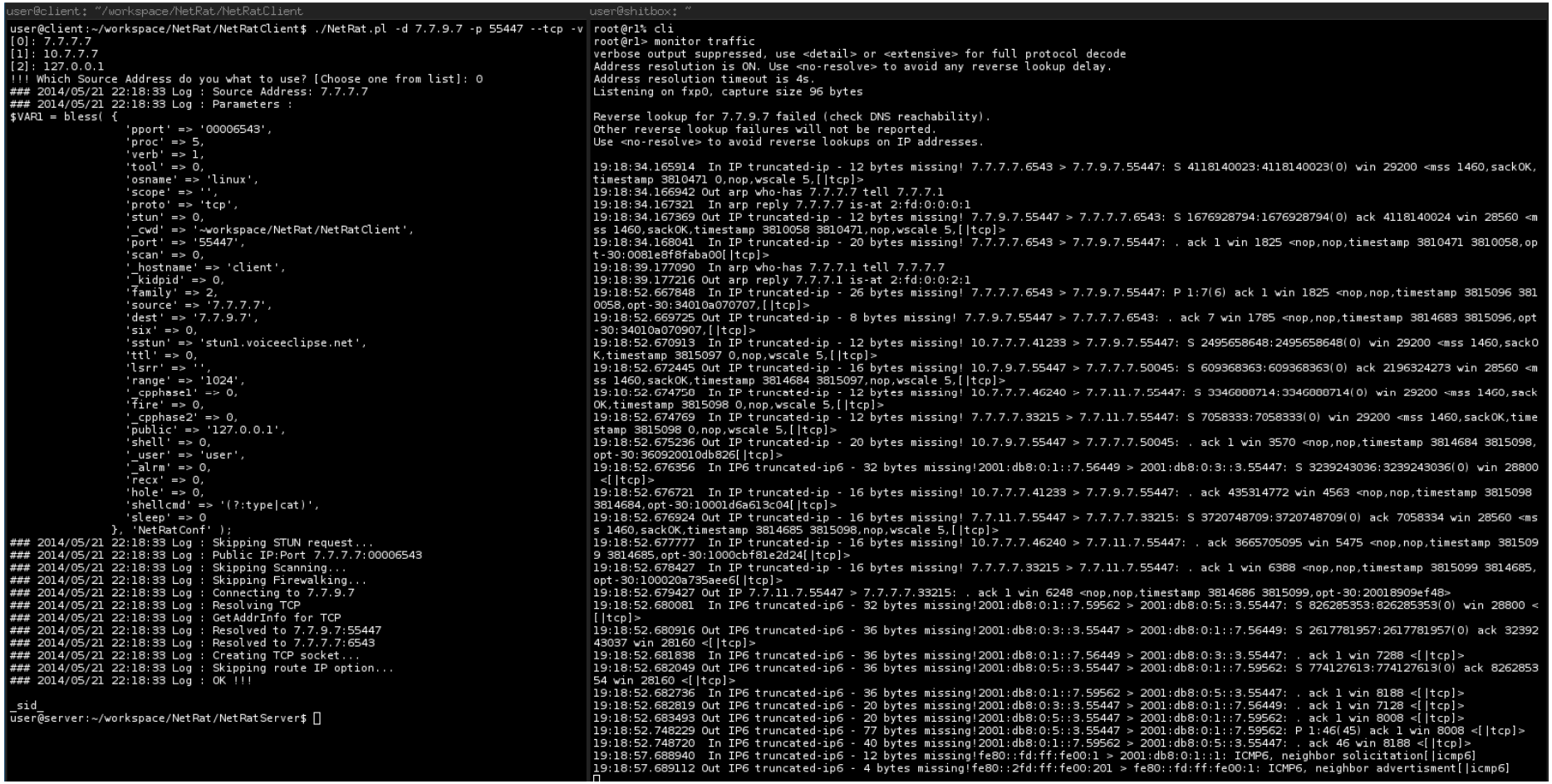


Ilustración 107: comando _sid_ ejecutado en el cliente (terminal de la izquierda), y captura en el router (terminal de la derecha)

En las ilustraciones 13 y 14, correspondientes al ejemplo presentado en la sección 3.2 se mostraba una tabla de texto con la salida por pantalla de la captura realiza en el router (ilustración 13) y la traza captura en la máquina que hace de host del entorno virtualiza (ilustración 14). En caso de que las imágenes con capturas “puras y duras” no sean del agrado del lector, se le invita gentilmente a revistar aquellas ilustraciones ya que lo mostrado es MPTCP sobre *dual-stack*.

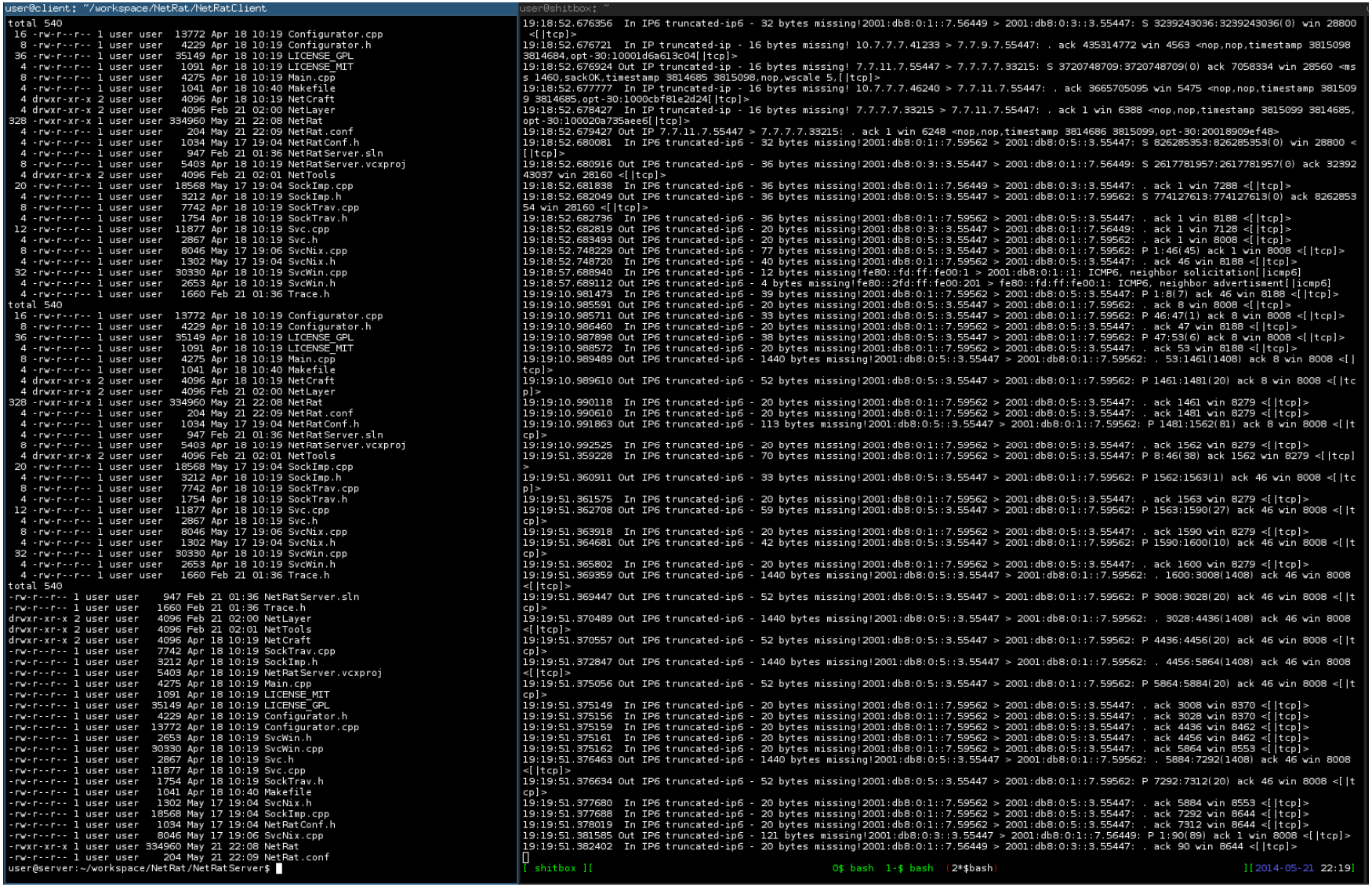


Ilustración 108: en el terminal de la izquierda se ejecutan comando para listar el directorio actual del servidor desde el cliente. En el terminal de la derecha se aprecia como los datos se transmites por múltiples caminos, con preferencia por aquellos sobre IPv6.

6.2 El nivel de conexión en MPTCP

¿Cómo hace MPTCP para, mediante el mismo API que TCP, gestionar simultáneamente una conexión formada por varios caminos? MPTCP es un protocolo que se ha adaptado a la realidad de Internet. Para poder gestionar la conexión MPTCP de una manera transparente a los dispositivos que conforman el Internet de hoy en día, léase *middleboxes*, se usa el campo opciones de la cabecera TCP, esto es, la información de control relativa a MPTCP no viaja en la zona de datos de los segmentos TCP, sino que lo hacen como opciones dentro de la cabecera. En el artículo *How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP* [MPTCP], los autores analizan el efecto de introducir nuevos campos de opciones con los dispositivos *middlebox* desplegados en Internet. La conclusión de dicho análisis es que la única manera de codificar el plano de control asociado a MPTCP es mediante el campo opciones de la cabecera TCP, descartando completamente el hacerlo en la zona de datos. Además, aunque en la mayoría de los casos las conexiones MPTCP trabajaron de manera transparente a los dispositivos *middlebox* de Internet, puede haber efectos colaterales indeseados, como por ejemplo que se eliminen de la cabecera las opciones nuevas. Por ello MPTCP implementa un mecanismo de *fallback*[54] a TCP en caso de que no sea posible establecer la conexión MPTCP entre dos puntos remotos.

Las referencias que describen MPTCP en más detalle [RFC6182] [RFC6824] introducen el concepto de conexión MPTCP como un conjunto de flujos distribuidos cada uno de ellos por un camino diferente, o lo que es lo mismo, una conexión MPTCP es un conjunto de conexiones TCP cada una de ellas identificada mediante una tupla IP:puerto diferente. Los flujos en MPTCP se manejan de forma interna y completamente transparente al nivel de aplicación, que únicamente interactúa con el nivel de conexión a través de un descriptor de socket, exactamente igual que se hace con TCP. El control de la conexión MPTCP se hace mediante el uso de opciones específicas

6.2.1 MP_CAPABLE

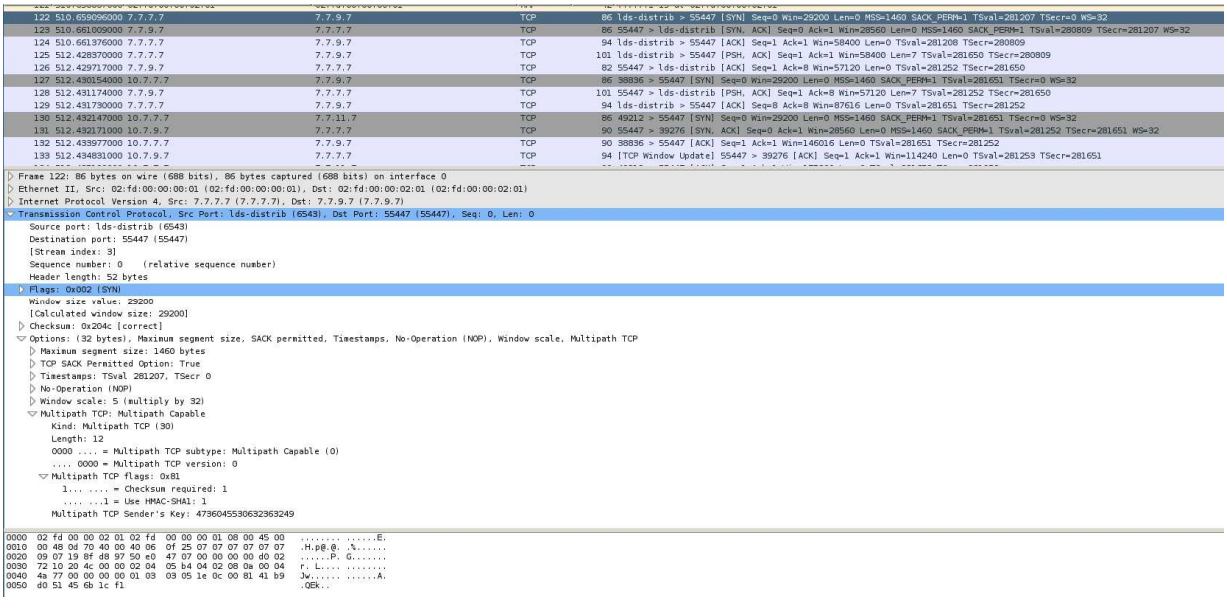


Ilustración 109: opción MP_CAPABLE

Al inicio del establecimiento de la conexión MPTCP se utiliza la opción MP_CAPABLE para negociar el uso de MPTCP en caso de que los nodos lo soporten.

[54] Fallback en la jerga es el término que se utiliza para describir un mecanismo redundante y auxiliar en el caso de fallo o error para garantizar que el servicio no quede interrumpido.

6.2.2 ADD_ADDRESS

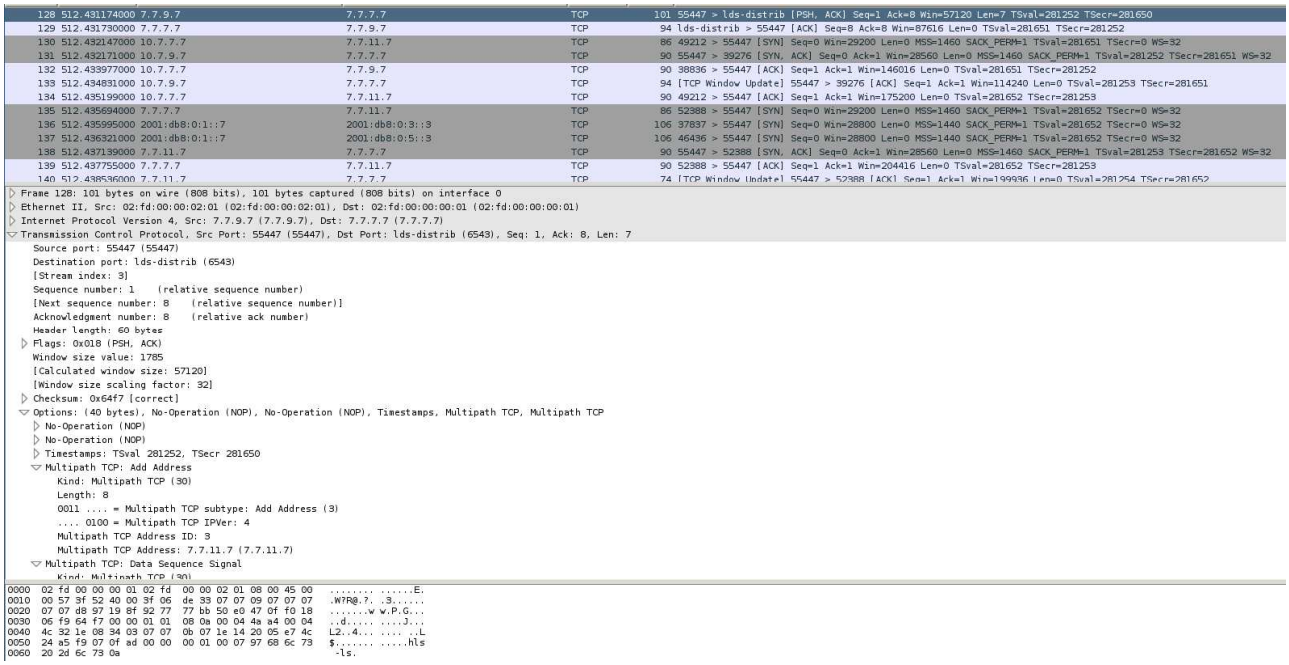


Ilustración 110: opción ADD ADDRESS

La opción add address se usa, para en el caso de que sí se soporte MPTCP y los nodos dispongan de más de un interfaz, puedan informar a la otra parte de estas direcciones, ya que los subsiguientes establecimientos de conexión TCP se pueden iniciar desde cualquier parte, no quedando limitado al nodo cliente.

6.2.3 MP_JOIN

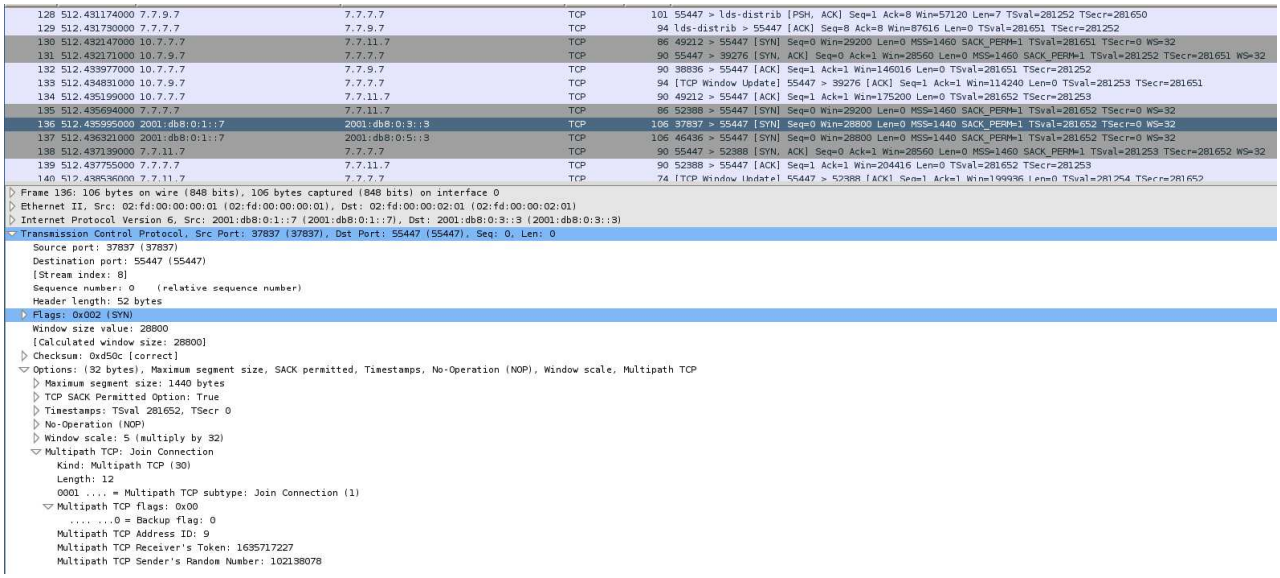


Ilustración 111: opción MP_JOIN

Los caminos alternativos de añaden a la conexión MPTCP mediante segmentos TCP con la opción MP_JOIN. A nivel de TCP esto se hace mediante el clásico *3-way-handshake*. Como se aprecia en la ilustración 111, el segmento TCP tiene el flag SYN activado además de la opción MPTCP.

6.2.4 Data Sequence Signal

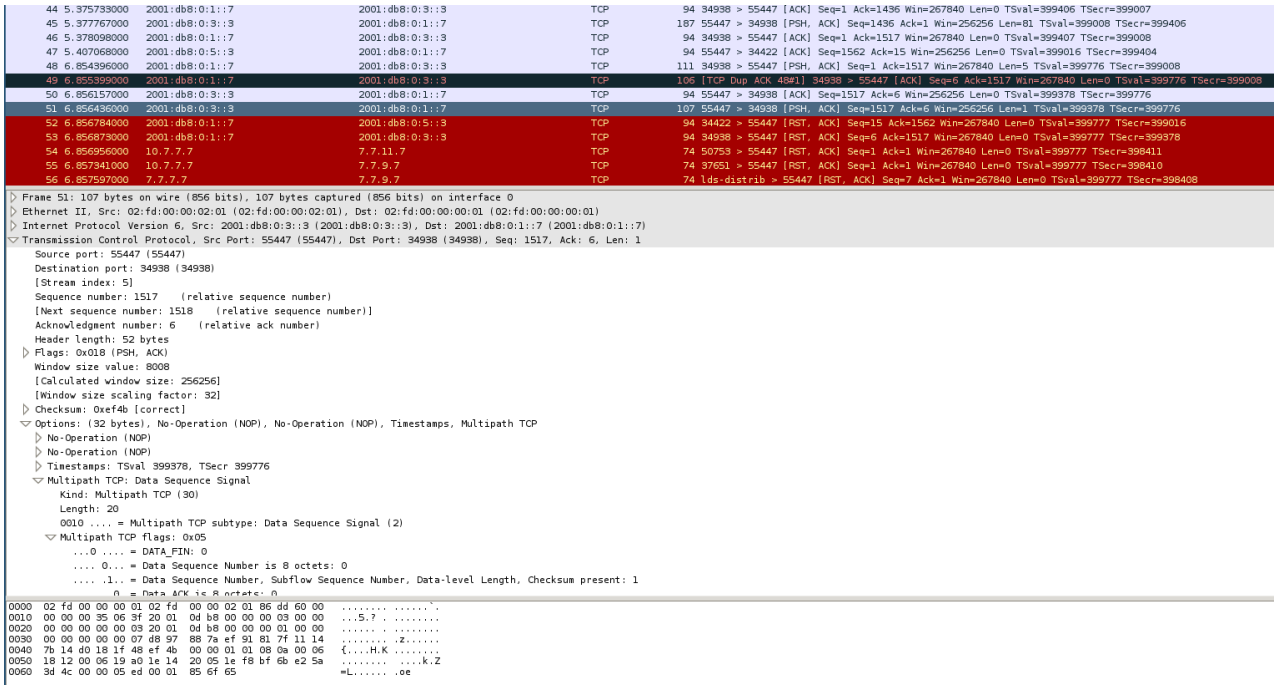


Ilustración 112: data sequence signal y cierre de conexión abrupto (SO_LINGER 1,0) en MPTCP.

La opción *data sequence signal* es usada para proveer fiabilidad a nivel de conexión MPTCP. Debido a que cada flujo se gestiona a nivel TCP (con su propio número de secuencia), el global de todos los caminos, esto es, la conexión MPTCP, se gestiona mediante el *data sequence signal*.

Cuando se quiere cerrar la conexión MPTCP se activa el flag FIN en la opción *data sequence signal*.

6.2.5 REMOVE_ADDR y cierre de conexión

La opción REMOVE_ADDR se puede usar para eliminar interfaces de los posibles candidatos a participar de la conexión MPTCP (esto se hizo para soportar movilidad). Un camino deja se de ser usado cuando se cierra la conexión TCP mediante la secuencia típica de terminación con segmentos con flag FIN activado. La conexión MPTCP se cierra activando el flag FIN en el *data sequence signal*. También es posible cerrar una conexión MPTCP de forma abrupta mediante el uso de la opción SO_LINGER, tal y como se muestra en la ilustración 112.

6.3 Comentario final sobre MPTCP

Como bien indica su nombre y ha sido posible evidenciar en los múltiples ejemplos presentados en este trabajo, MPTCP divide el flujo de datos TCP a través de múltiples caminos, con el fin aprovechar los múltiples interfaces de los que los modernos terminales disponen, evitar caminos con congestión o balancear la carga, y aumentar la resistencia a fallos de aplicaciones que dependan de TCP para su funcionamiento. Además, en el caso de los dispositivos *middlebox* interactúen de forma no deseada con MPTCP, se dispone de un mecanismo de *fallback* a TCP puro y duro, gracias a lo cual MPTCP debería, en teoría, ser capaz de funcionar al menos tan bien como lo hace TCP.

El ejemplo de la sección 6.1 permite observar en acción a este protocolo ejecutado en Linux (en concreto en Debian Jessie). Respecto a la elección de la plataforma para testear MPTCP, Linux es la única plataforma de las tres usadas que soporta MPTCP mediante una librería. Al comienzo de la elaboración de la sección correspondiente del trabajo fue necesario instalar la librería manualmente de los repositorios, una vez hecho esto el kernel se actualiza automáticamente como lo haría por defecto. En la elaboración del software se prestó especial atención a la compatibilidad multiplataforma, pero debido a que el uso de MPTCP no requiere cambios a nivel de código, por sencillez las pruebas se han limitado a la plataforma Linux. Esta falta de funcionalidad multiplataforma en lo relativo a MPTCP es un punto negativo a MPTCP.

A un nivel de usuario práctico no hay mucho más que comentar sobre MPTCP. MPTCP ofrece la funcionalidad de TCP añadiendo la capacidad multicamino, por lo que todas las técnicas para ofrecer un servicio orientado a conexión fiable usadas por TCP son aplicables a MPTCP (retransmisiones, controles de flujo y congestión con ventana deslizante y de congestión, algoritmos varios como Nagle, ACK retrasado, etc.). El enfoque del este trabajo fin de máster que en su día se adoptó fue práctico, significando práctico que la mayor parte del esfuerzo realizado se invirtió en el desarrollo de software y escenarios de pruebas para validar los resultados. Aunque se ha profundizado bastante en general, es imposible revisar a la vez todos los conceptos teóricos que involucra MPTCP: hubo que elegir entre centrarse en algún aspecto concreto de MPTCP o estudiarlo desde cierta altura que permitiera tener una perspectiva global. Y se optó por la segunda opción. Desde de un punto de vista más teórico, si TCP ya es de por sí complejo, MPTCP se complica aún más debido a la gestión de los múltiples caminos. Las referencias [MPTCP] y su equivalente IETF RFC ([RFC6182]), [RFC6356] y [RFC 6824] ofrecen más información sobre este protocolo.

En la referencia [MPTCP] los autores de MPTCP explican detalladamente el proceso y las decisiones de diseño que adoptaron, además de proporcionar una introducción al protocolo en sí. Esta referencia además de hacer un interesante análisis del Internet de hoy en día expone algunos detalles sobre MPTCP que se procede a resaltar:

1. MPTCP hace uso del campo opciones de TCP (en lugar de usar la zona de *carga* del segmento).
2. La negociación de la funcionalidad MPTCP comienza en el segmento con flag SYN activado mediante una opción especial llamada MP_CAPABLE. Como comentan los autores de MPTCP en su artículo [MPTCP], el hecho de que un segmento TCP contenga una nueva opción no provocó que *middleboxes* de Internet descartaran el segmento, pero si la quitaron en algunos casos. En el escenario propuesto en este trabajo en el que no hay ningún dispositivo de tipo *middlebox* MPTCP funcionó sin problemas.
3. Una conexión MPTCP se representa mediante un socket a nivel de aplicación.
4. Una vez que se ha establecido la conexión MPTCP se puede añadir caminos alternativos mediante el uso de la opción MP_JOIN. Aunque a nivel de transporte existen múltiples conexiones TCP (unívocamente identificadas mediante la correspondiente tupla IP:puerto), a nivel de

aplicación únicamente se maneja un único descriptor de fichero (como se puso de manifiesto en el ejemplo de la sección 6.1).

5. Los múltiples caminos son transparentes a nivel de aplicación ya que la API es la misma que la usada en el caso de TCP y el número de descriptores de ficheros no aumenta. Como se explica en detalle en [RFC6824] MPTCP mimetiza TCP a *nivel de conexión MPTCP* mediante el uso de opciones (MP_CAPABLE, MP_JOIN, ADD_ADDR, REMOVE_ADDR, DATA_SEQUENCE_SIGNAL).
6. El flujo de cada camino tiene su propia máquina de estados, identificador y número de secuencia asociado, debido a las pruebas y conclusiones que los autores presentan en [MPTCP], pero el control de flujo se realiza mediante una única ventana, por lo que es necesaria una nueva opción TCP para un número ACK adicional, esto es, el mecanismo de confirmación tiene *dos dimensiones*: uno por camino y otro en el global.
7. Un segmento por un camino con el flag FIN activo significa que ese camino va a dejar de usarse. En aras a soportar movilidad existe la opción REMOVE_ADDR. La conexión MPTCP se cierra activando el flag DATA_FIN en el campo *data sequence signal,* que es semánticamente equivalente a enviar un segmento con el flag FIN activado en el caso de TCP.

7. Stream Control Transmission Protocol

Como una imagen vale más que mil palabras (lo cual es demostrable matemáticamente), se presenta SCTP [RFC4960] mediante una traza en modo *charlatán* en la ilustración 113. En concreto, en la traza se muestra SCTP sobre IPv6, y viendo la traza hexadecimal no hay mucha diferencia respecto TCP o UDP.

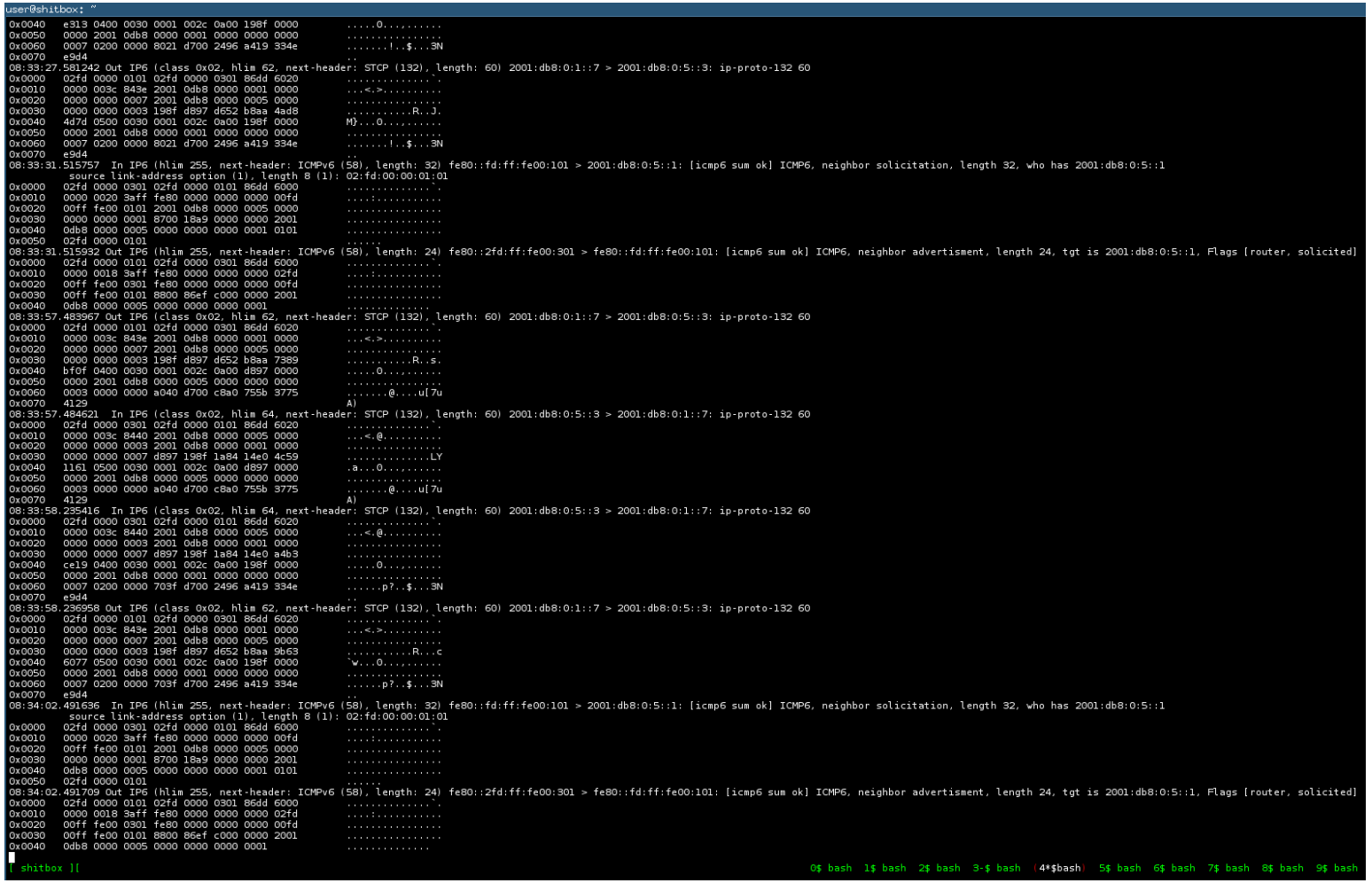


Ilustración 113: traza capturada mediante tcpdump de tráfico SCTP sobre IPv6

Desde que existe Internet los protocolos que implementan el núcleo de su modelo de referencia poco han cambiado, y esto se debe a que han funcionado muy bien a excepción del agotamiento de direcciones IPv4 que nadie fue capaz prever. Ahora se habla mucho de IPv6 porque no queda más remedio que actualizar IPv4 debido al agotamiento de direcciones IPv4, y debido a la magnitud de cambios que la transición a IPv6 involucra, si hay un momento apropiado para tocar algún nivel más de la torre de protocolos TCP/IP, en el caso de que realmente sea necesario, es ahora (o nunca). El objetivo inicial de este trabajo fue SCTP a secas: el estudio de un nuevo protocolo de nivel de transporte con el que ya estaba familiarizado ligeramente, porque lo había visto transportando tráfico Diameter. Antes de que algún lector despistado se ilusione con todo lo que promete SCTP, es preciso decir que muy probablemente SCTP nunca va a alcanzar la cuota de popularidad de la que actualmente disfrutan TCP y UDP. SCTP viene cargado con buenas ideas, pero no parece probable que le coma terreno a los protocolos clásicos de nivel transporte en el campo del desarrollo de aplicaciones por dos motivos: el primero porque los fabricantes no lo están adoptando en sus sistemas operativos, y segundo y principal, porque no resuelve ninguno de los problemas más acuciantes que amenazan a Internet hoy en día. ¿Y por qué entonces se ha decidido incluirlo en este trabajo? Porque SCTP es, a nivel técnico, un protocolo interesante que ya incorpora de forma nativa algunas técnicas que actualmente se implementan a nivel de aplicación como los *heartbeats*[55], el multi-homing, la eliminación del estado TIME_WAIT para los números de puerto o el incremento de seguridad en general. En un escenario ficticio en el que SCTP estuviera desplegado y soportado a nivel global muy probablemente las aplicaciones que hacen uso intensivo de Internet serían un poco más sencillas y manejables, más aún en el caso de que soportase la funcionalidad de multi-camino, cosa que aún no hace. Las transgresiones de los principios básicos y elementales sobre los que se cimentó Internet tampoco le allanan el camino a SCTP para que sea adoptado en masa, ya en el red hay infinidad de dispositivos de los denominados *middleboxes* husmeando y metiendo sus narices (e incluso modificando) en los niveles superiores al red, cuando esto no debería ser así.

En la RFC *"Architectural Guidelines for Multipath TCP Development"* [RFC 6182] se compara brevemente SCTP con MPTCP, ya que ambos protocolos tienen similitudes. De la misma manera que MPTCP, SCTP es de tipo *multi-homed* (puede usar varios interfaces de red en la misma máquina), pero a diferencia de MPTCP, esto lo hace por motivos de movilidad y para aumentar la fiabilidad, mientras que MPTCP además lo hace para aprovechar las cualidades *multicamino* [RFC6182], lo cual SCTP todavía no es capaz de hacer. MPTCP es una extensión sobre TCP para añadir la capacidad multicamino y aprovechar los múltiples interfaces de los que los actuales dispositivos disponen. De la misma manera que MPTCP hace con TCP, se está trabajando en versiones multicamino de SCTP en la actualidad [DRAFTSCTP]. En este capítulo se revisa el protocolo SCTP, en concreto la variante *one-to-one,* usado por la aplicación NetRat [NETRAT].

7.1 Adopción de SCTP en una aplicación

SCTP [RFC4960] nació con dos variantes *one-to-one* y *one-to-many*. La versión *one-to-one* también es conocida como SCTP en modo TCP, mientras que la versión *one-to-many* también se conoce como versión UDP. La versión *one-to-one* se diseñó para facilitar la migración de aplicaciones TCP a SCTP mediante el menor número de cambios necesarios. La versión *one-to-many* se diseñó para minimizar el número de descriptores de fichero que un servidor tiene que manejar a la hora de gestionar múltiples conexiones SCTP [UNIX-1] (igual que un servidor UDP puede recibir datagramas de cualquier cliente UDP en un único socket UDP, siempre y cuando no haya ejecutado la función connect en ese socket [UNIX-1]). El punto de partida sobre el que se comenzó con SCTP fue una

[55] Heartbeat: en la jerga de Internet y de los protocolos de comunicaciones *heartbeats* (latidos de corazón) son paquetes intercambiados para verficar que los extremos siguen operativos y para mantener los *pin-holes* en los cortafuegos y en los dispositivos NAT abiertos. Esta funcionalidad que viene *de-serie* en SCTP, es el equivalente al keepalive de TCP. UDP no tiene estas características.

aplicación que hacía uso de los protocolos TCP y UDP (basada sobre tres librerías, de las cuales una, NetLayer, es el *envoltorio* sobre la API Socket que también iba a ser usado con SCTP).

La aplicación NetRat [NETRAT] está implementada de tal manera que puede funcionar con cualquiera de los protocolos de nivel de transporte de forma individual o simultáneamente. En el caso que se el servidor se configure para escuchar en puerto TCP (o MPTCP), UDP o SCTP simultáneamente, la lógica asociada a cada protocolo se ejecuta en una hebra diferente[56]. Añadir SCTP *one-to-one* como crear una hebra específica para este protocolo y la lógica, que es la misma que para TCP. Aunque habría sido posible reutilizar la lógica de TCP en la aplicación, se crearon las funciones específicas para SCTP para separar un protocolo del otro para la posible ampliación a SCTP *one-to-many*, que no tuvo lugar, como se detallará a continuación.

```
void SvcNix::SvcMain()
{
    logh->Log("[SvcNix::SvcMain]");
    SvcDynamicSetUp();
    logh->Log("[SvcNix::SvcMain]: preparing protocol threads");
    timeval tv;
    while (!_Configurator::ServiceConfigurator::GetConfig() && !_Configurator::ServiceConfigurator::GetReset()) {
        tv.tv_sec = (*static_cast<const int*>(servh->Config(_Configurator::SAFE)))*THREAD_FACTOR/FACTOR;
        tv.tv_usec = TV_SEL_U;
        if (tcpFlag) {
            logh->Log("[SvcNix::SvcMain]: tcpMode enabled, creating its protocol thread");
            tcpThread = std::make_shared<_NetTools::SmartThread>([](void *_t)->void* {
                auto _x=static_cast<SvcNix*>(_t);
                _x->SvcTcpMode();
                _x->SetProtoFlag(_x->tcpFlag, true);
                return nullptr;
            }, (void*)this);
            SetProtoFlag(tcpFlag, false);
            status->Add(tcpThread);
        }
        if (udpFlag) {
            logh->Log("[SvcNix::SvcMain]: udpMode enabled, creating its protocol thread");
            udpThread = std::make_shared<_NetTools::SmartThread>([](void *_t)->void* {
                auto _x=static_cast<SvcNix*>(_t);
                _x->SvcUdpMode();
                _x->SetProtoFlag(_x->udpFlag, true);
                return nullptr;
            }, (void*)this);
            SetProtoFlag(udpFlag, false);
            status->Add(udpThread);
        }
        if (sctpFlag) {
            logh->Log("[SvcNix::SvcMain]: sctpMode enabled, creating its protocol thread");
            sctpThread = std::make_shared<_NetTools::SmartThread>([](void *_t)->void* {
                auto _x=static_cast<SvcNix*>(_t);
                _x->SvcSctpMode();
                _x->SetProtoFlag(_x->sctpFlag, true);
                return nullptr;
            }, (void*)this);
            SetProtoFlag(sctpFlag, false);
            status->Add(sctpThread);
        }
        logh->Log("[SvcNix::SvcMain]: blocking in Select");
        select(1, 0, 0, 0, &tv);
        logh->Log("[SvcNix::SvcMain]: checking comm threads status ...");
        if (!_Configurator::ServiceConfigurator::GetConfig())
            logh->Log("[SvcNix::SvcMain]: cycling #thread ", status->Cycle());
    }
    status->Join();
    SvcDynamicCleanUp();
    logh->Log("[SvcNix::SvcMain]: service thread ended");
    logh->Log("[SvcNix::SvcMain]: pool size ", pool->Size());
    if (!_Configurator::ServiceConfigurator::GetReset()) {
        _Configurator::ServiceConfigurator::SetConfig(false);
        logh->Log("[SvcNix::SvcMain]: recovering ...");
        SvcMain();
    } else
        logh->Log("[SvcNix::SvcMain]: tearing Down gently  ");
}
```

Ilustración 114: función principal de la aplicación NetRat que se encarga de crear los threads correspondientes a cada protocolo de nivel de transporte

[56] También habría sido posible multiplexar mediante la función select, pero se optó por implementar cada protocolo en su propio thread.

La incorporación de SCTP de tipo *one-to-one* a una aplicación que usa TCP, es inmediato. Crear un socket para usar SCTP *one-to-one* se hace especificando un tipo SOCK_STREAM (igual que TCP) y un protocolo IPPROTO_SCTP (ésta es la única diferencia). Como se ha comentado en varias ocasiones, NetRat en la parte servidor interactúa con la API Socket mediante la librería NetLayer. Para usar SCTP *one-to-one* a través de NetLayer [NETLAYER] no fue necesario realizar ningún cambio adicional.

... después de este inesperado éxito migrando a SCTP *one-to-one,* y cargado de optimismo, procedía intentar hacer lo mismo con SCTP *one-to-many,* y entonces comenzaron los problemas...

Cuando se escribieron las librerías siempre se tuvo en cuenta la compatibilidad multiplataforma. Esta compatibilidad ha hecho que algunas funciones interesantes de la API Socket no sean usadas al no estar soportadas en diferentes plataformas. La librería NetLayer tiene como objetivo ofrecer una interfaz simplificada hacia la API Socket. Desde mi punto de vista personal, eso se ha conseguido, funcionando correctamente en Linux, FreeBSD y Windows y logrando el requisito de compatibilidad multiplataforma. Añadir SCTP *one-to-many*, que necesita unas funciones específicas (a diferencia de *one-to-one* que usa la misma API que TCP) y no portadas a todas las plataformas contravenía el objetivo de compatibilidad marcado desde el origen. Alguien dijo una vez que *cuando el beneficio no es aparente las pérdidas están a la vuelta de la esquina*, pero en el caso de evaluar el coste de usar SCTP *one-to-many* y el beneficio que podría aportar, se decidió por sacrificar *one-to-many* en aras a mantener el código lo menos complicado (dentro de su complejidad) posible. Por ejemplo las funciones sctp_sendto y sctp_recvmsg son versiones 'simplificadas' de recvmsg y sendmsg, necesarias para enviar a través de SCTP *one-to-many* habrían requerido adaptar la librería NetLayer para un caso poco útil, pero complicando considerablemente el código. Ya de por sí, la API socket es complicada (siendo el objetivo de NetLayer [NETLAYER] mitigar en lo posible este nivel de complejidad) como para usar unas funciones extremadamente complejas. Por todo esto se descartó *one-to-many* como una opción para este trabajo. En lo que resta de apartado se va a presentar los resultados obtenidos con SCTP *one-to-one*.

7.2 Cabecera SCTP

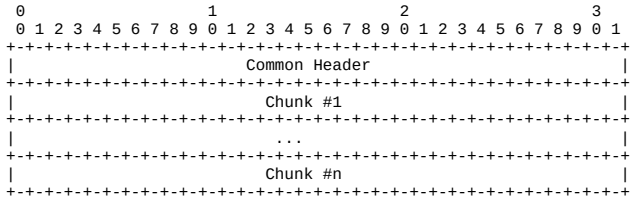


Ilustración 115: cabecera SCTP

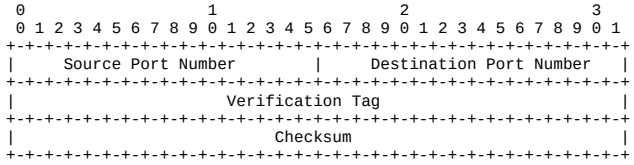


Ilustración 116: cabecera común de SCTP

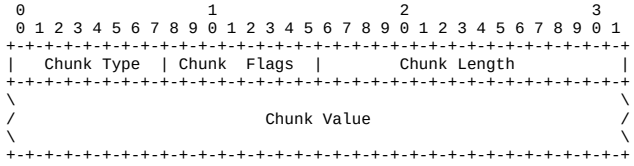


Ilustración 117: cabecera de chunk

7.3 Máquina de estados en SCTP

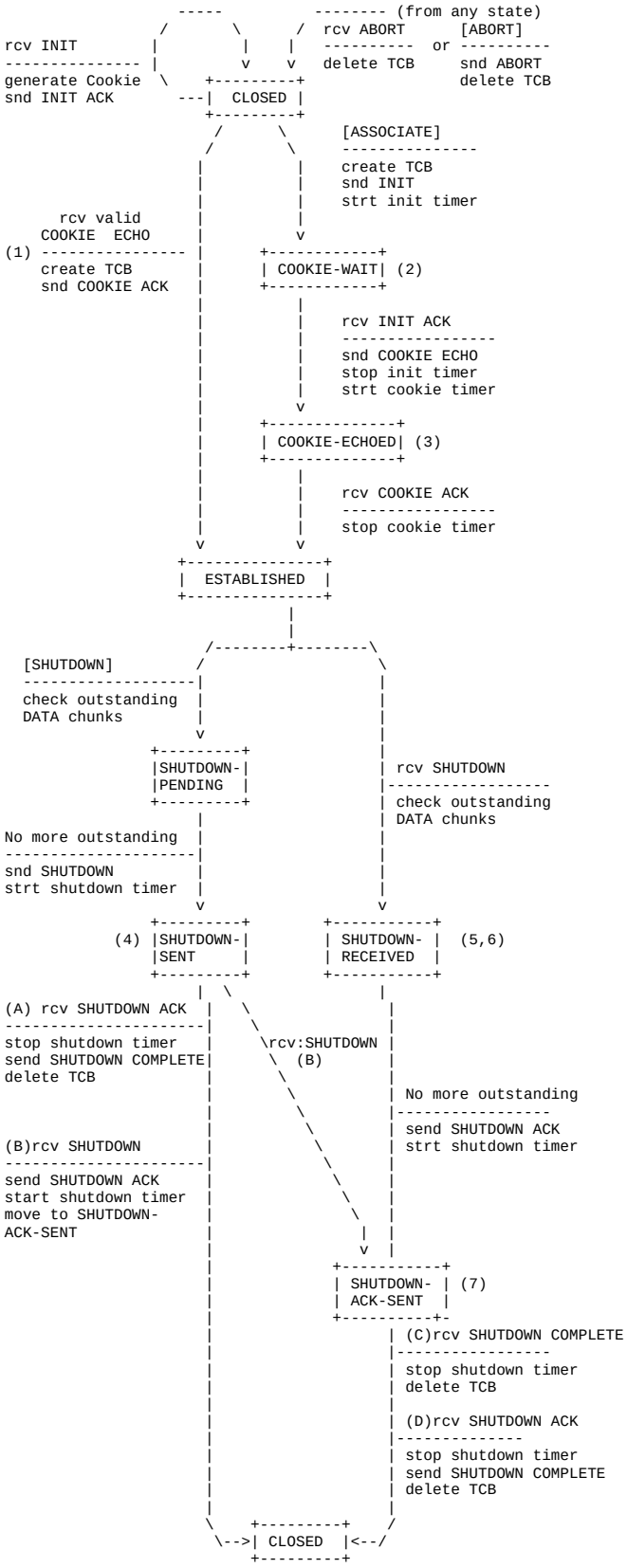


Ilustración 118: máquina de estados SCTP (diagrama sacado de la referencia [RFC4960])

7.3.1 Establecimiento de la asociación

En la bibliografía, una conexión SCTP se conoce como asociación para resaltar el que se pueden usar diferentes direcciones IP gracias a la capacidad multihoming del protocolo. La ilustración 115 muestra el diagrama de estados del protocolo SCTP. En SCTP el establecimiento de la conexión se realiza mediante el denominado *4-way-handshake*, evitando un ataque de tipo *denial of service* [UNIX-1]. Durante el establecimiento de la asociación, el cliente y el servidor intercambian unos identificadores que estarán presentes durante la asociación y que hacen que no exista el estado TIME_WAIT a nivel de puerto (existe pero a nivel de identificador de asociación).

7.3.2 Fase de transferencia de datos

SCTP es un protocolo de nivel de transporte fiable y orientado a mensaje donde la unidad de transferencia es conocida como *chunk*. En la ilustración 116 se muestra una captura de SCTP de tipo *one-to-one* sobre IPv6. Lo que se ha observado durante la experimentación con SCTP es que cada *chunk* de datos transporta lo que SCTP considera un mensaje de usuario, y en el caso de transferencias de texto se observó que SCTP divide los datos en mensajes cada vez que encuentra caracteres fin de línea, lo cual en algunos casos condujo a tener muchos *data-chunks* en cada mensaje SCTP, cada uno con muy pocos datos, con la consiguiente sobrecarga que las cabeceras de los chunks conllevan. En la ilustración 116 también se observan mensajes de tipo Heartbeat que serían útiles para mantener los pin-holes de los cortafuegos y de los dispositivos NAT.

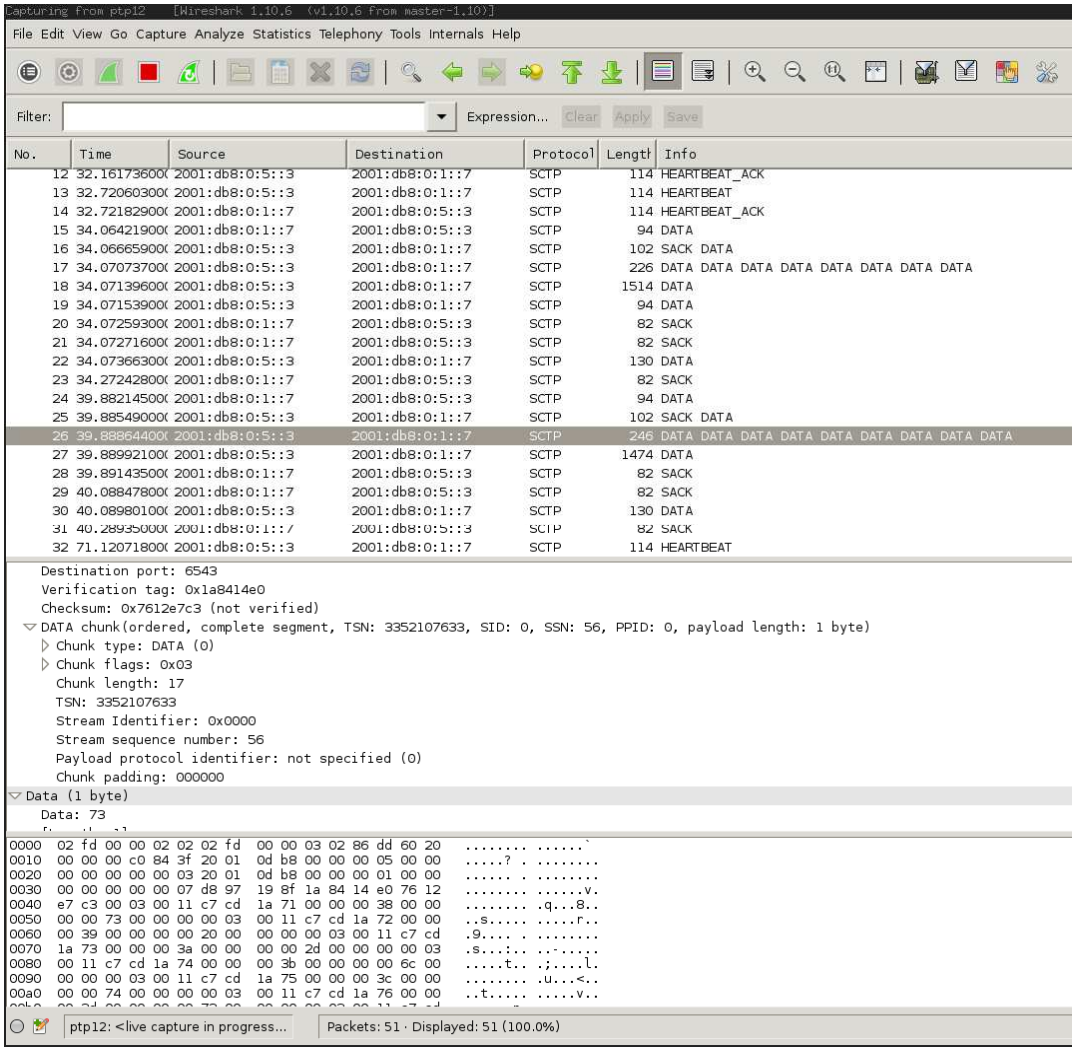


Ilustración 119: traza de tráfico SCTP one-to-one sobre IPv6

7.3.3 Cierre de la conexión

En el caso de SCTP, el cierre de la conexión se realiza mediante una secuencia de tres paquetes. Además, gracias al uso de la etiquetas lo puertos pueden ser usados inmediatamente después del cierre de una asociación gracias a la existencia del estado TIME_WAIT. Realmente el estado TIME_WAIT existe, pero a nivel de etiqueta, no a nivel de puerto.

43	155.256799000	7.7.9.7	7.7.7.7	SCTP	94 HEARTBEAT
44	155.257200000	7.7.7.7	7.7.9.7	SCTP	94 HEARTBEAT_ACK
45	155.453732000	7.7.7.7	7.7.9.7	SCTP	94 HEARTBEAT
46	155.454946000	7.7.9.7	7.7.7.7	SCTP	94 HEARTBEAT_ACK
47	171.645717000	7.7.7.7	7.7.11.7	SCTP	94 HEARTBEAT
48	171.646905000	7.7.9.7	7.7.7.7	SCTP	94 HEARTBEAT_ACK
49	184.584099000	7.7.7.7	7.7.9.7	SCTP	70 DATA
50	184.587344000	7.7.9.7	7.7.7.7	SCTP	82 SACK DATA
51	184.587786000	7.7.7.7	7.7.9.7	SCTP	54 SHUTDOWN
52	184.587863000	7.7.7.7	7.7.9.7	SCTP	70 SHUTDOWN SACK
53	184.589799000	7.7.9.7	7.7.7.7	SCTP	150 DATA DATA DATA DATA DATA
54	184.590618000	7.7.7.7	7.7.9.7	SCTP	70 SHUTDOWN SACK
55	184.591888000	7.7.9.7	7.7.7.7	SCTP	50 SHUTDOWN_ACK
56	184.592113000	7.7.7.7	7.7.9.7	SCTP	50 SHUTDOWN_COMPLETE

Ilustración 120: secuencia de cierre de asociación SCTP

La opción SO_LINGER, igual que con TCP, permite configurar la manera en la que se libera la asociación SCTP. En la ilustración 117 se muestra el cierre abrupto de una asociación SCPT mediante el intercambio de únicamente dos paquetes de tipo ABORT.

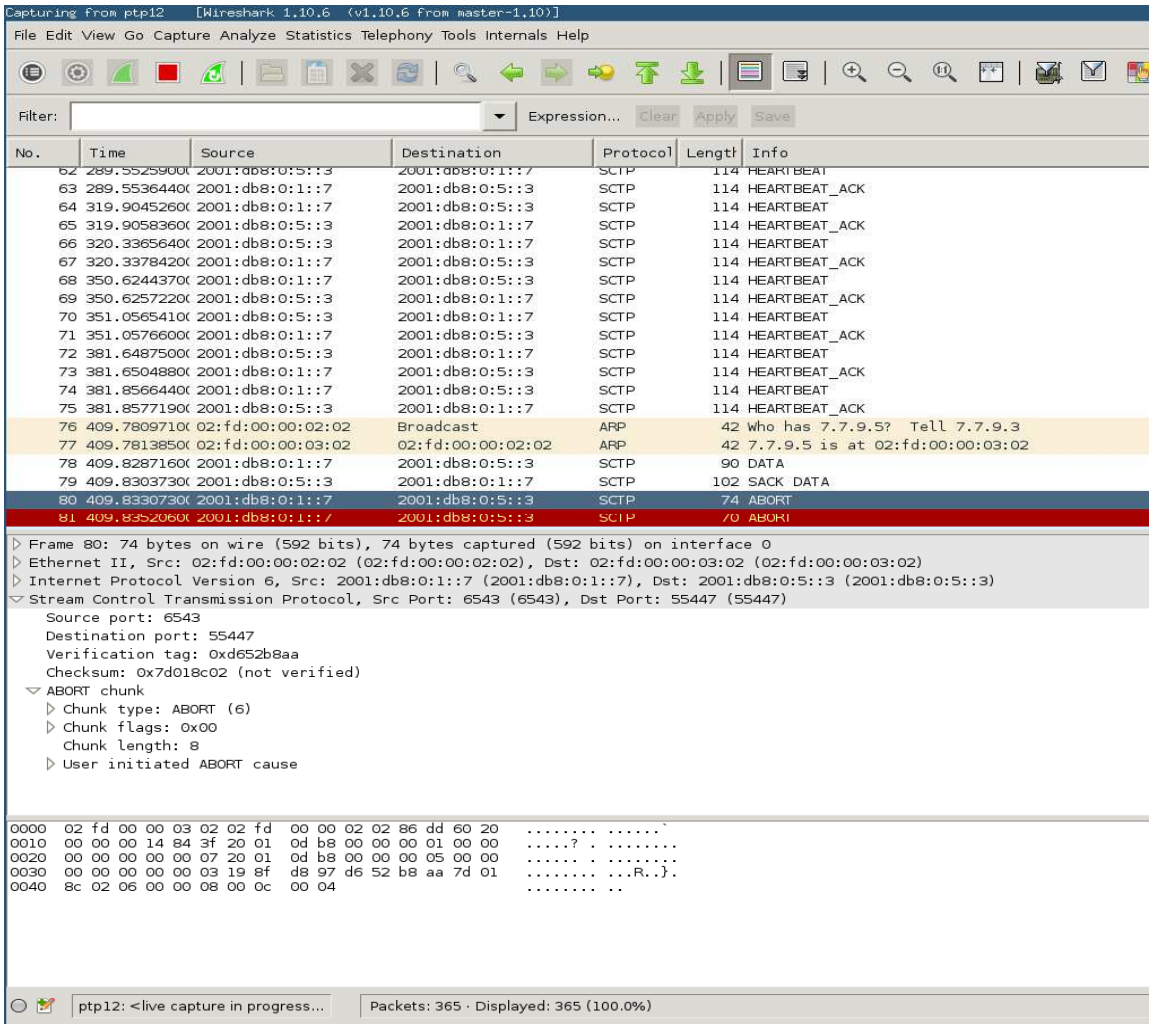


Ilustración 121: cierre de conexión SCTP abrupto

7.4 Usando SCTP en la práctica

```
sub _ResolveSctp
{
    my $self=shift;
    my $rdest=shift;
    my $rlocal=shift;
    print STDOUT strftime("### %Y/%m/%e %H:%M:%S", localtime), " Log : Resolving SCTP\n" if $self->config()->verb();
    print STDOUT strftime("### %Y/%m/%e %H:%M:%S", localtime), " Log : GetAddrInfo for SCTP\n";
    push(@{$rdest}, $self->_GetAddr($self->config()->dest(), $self->config()->port(), {socktype => SOCK_STREAM}));
    push(@{$rlocal}, $self->_GetAddr($self->config()->source(), &NetRatConf::LPo, {socktype => SOCK_STREAM}));
    print STDOUT strftime("### %Y/%m/%e %H:%M:%S", localtime), " Log : Creating SCTP socket...\n" if $self->config()->verb();
    socket(my $ssd, $rdest->[0]->{family}, $rdest->[0]->{socktype}, &NetRatConf::SCTP) or die "socket: $!";
    setsockopt($ssd, SOL_SOCKET, SO_REUSEADDR, 1) or die "setsockopt: $!";
    setsockopt($ssd, SOL_SOCKET, SO_LINGER, pack('ii', 1, 0)) or die "setsockopt: $!";
    bind($ssd, $rlocal->[0]->{addr}) or die "bind : $!";
    $self->config()->set_sctp($ssd);
}
```

Ilustración 122: resolución de direcciones en SCTP

El hecho de haberse limitado a la versión *one-to-one* de SCTP hace que alguna de sus funcionalidades más esotéricas, como multiplexar múltiples streams de datos en la misma asociación, únicamente disponibles para *one-to-many,* no se hayan visto. Aun así, trabajar con SCPT *one-to-one* ha sido igual que si se hubiera hecho con TCP. En la ilustración 119 se observa que para usar la función GetAddrInfo, la *pista* o *hint* que se suministra es la misma que con TCP (socktype => SOCK_STREAM). En las ilustraciones 123 a 125 muestra la aplicación NetRat funcionando como un shell remote en el que el cliente ejecuta de forma remota comando *ls –ltr* en el servidor teniendo lugar toda la transferencia de datos entre ambos.

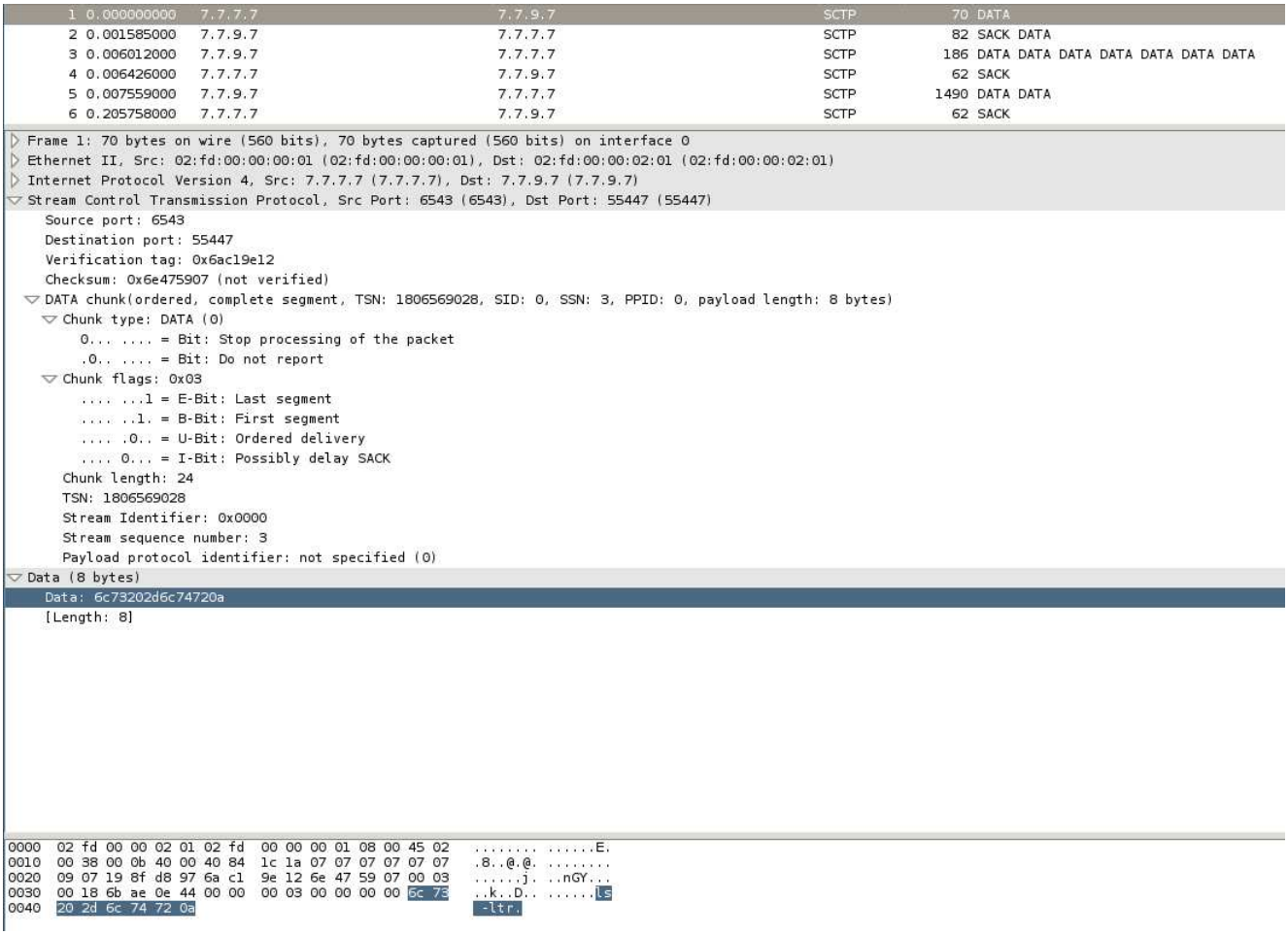


Ilustración 123: shell remoto sobre SCTP(ejecutando comando ls -ltr)

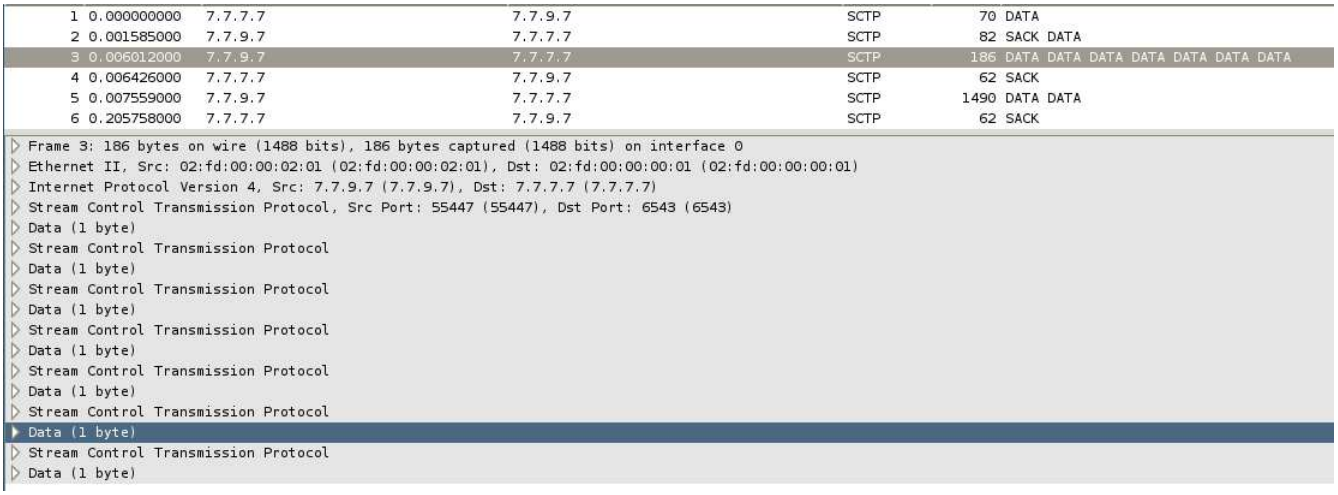


Ilustración 124: SCTP data chuncks y control chunks

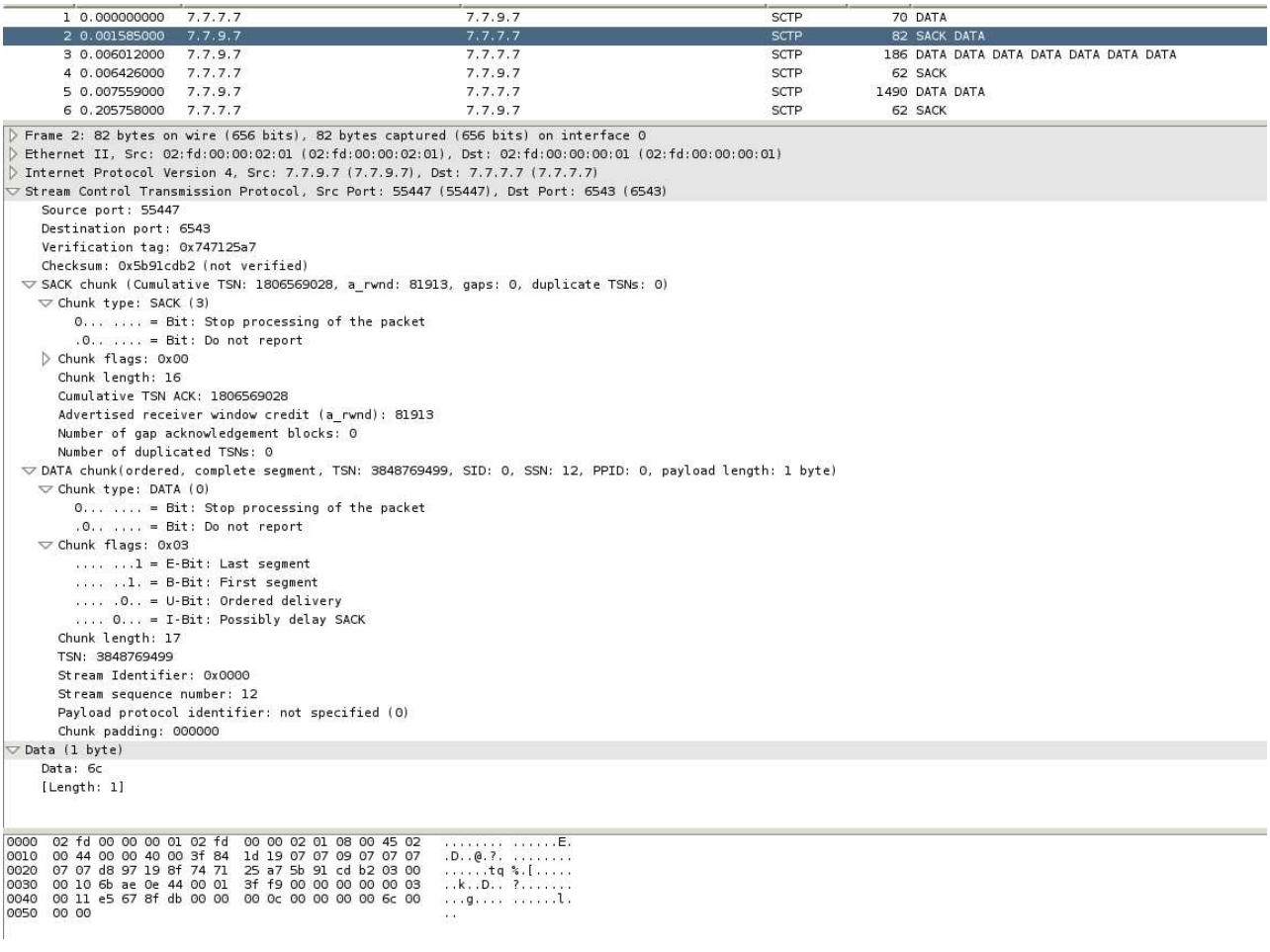


Ilustración 125: acuse de recibo selectivo en SCTP

8. Librerías y aplicaciones

La mejor manera de entender la parte práctica del trabajo (que ha sido en la que más se ha invertido) es descargándose el código de GitHub [ERNESTO81], compilándolo, ejecutándolo y leyéndolo. En esta sección se va a describir a grandes rasgos las aplicaciones y librerías entrando en detalle en los puntos que considere que merecen especial mención.

El principal objetivo del trabajo siempre ha sido estudiar en profundidad desde un punto de vista práctico los niveles de red y transporte, en concreto IPv6 e ICMPv6, SCTP y MPTCP. El resultado de este trabajo son tres librerías una aplicación de tipo cliente-servidor para ilustrar el uso de las librerías, y la actualización de otra aplicación escrita en 2010 de tipo cliente-servidor. Aunque no quiere decir nada, el número de líneas de código escritas (sin tener en cuenta ficheros de configuración, Makefile, Visual Studio o escenarios VNX) ha sido considerable tal y como se muestra en la tabla a continuación:

Proyecto	Líneas	Lenguaje
NetCraft	1100	C++
NetLayer	2600	C++
NetTools	1300	C++
NetExample	1900	C++
NetRat	5600	C++ Perl
Total	12500	C++ Perl

Tabla 6: número de líneas de código

8.1 NetRat

NetRat [NETRAT] es una aplicación de tipo cliente servidor que escribí en 2010 para Linux, Windows y FreeBSD, y que se ha actualizado y usado para probar las librerías especialmente desarrolladas para este trabajo. En honor a la verdad, la primera versión de NetRat estaba tan mal escrita, que gracias al *refactoring* realizado a lo largo de este trabajo han nacido las tres librerías para mitigar el caracter monolítico de la versión inicial. Si es cierto que NetRat estaba mal escrita, también es cierto que ha resultado muy útil para aprender. Hay margen de mejora aún.

NetRat se compone de tres partes: las clases de configuración, las clases de control y las clases de comunicaciones. Es además una aplicación con varios threads. La configuración se hace de forma estática a partir de un fichero de texto en el que se especifica que funcionalidades se quieren habilitar o deshabilitar. Esto se hace en tiempo de ejecución y no es necesario recompilar. También se puede cambiar la configuración del servidor desde el cliente mediante una interfaz específica. Para el caso en el que no es aconsejable usar el fichero de configuración también es posible configurar inicialmente el servidor en tiempo de compilación mediante el fichero cabecera NetRatConf.h, siendo posible modificar posteriormente la configuración desde el cliente en tiempo de ejecución (así únicamente es necesario el binario NetRat). La parte de control en el servidor se ejecuta en su propia hebra y es la encargada de gestionar las hebras de comunicaciones (TCP, UDP y SCTP se ejecutan en hebras diferentes) y las señales o eventos propios del sistema operativo. La parte de comunicaciones está compuesta por tantas hebras como protocolos estén habilitados, esto es, TCP tiene su propia hebra, al igual que UDP y SCTP. La gestión de las hebras es multiplataforma gracias al uso de NetTool [NETTOOLS]

La aplicación puede funcionar en sistemas con únicamente IPv4, únicamente IPv6 o dual-stack, gracias al uso de las librerías NetLayer [NETLAYER] y NetCraft [NETCRAFT]. El servidor de NetRat

se ha construido sobre las librerías NetLayer (para una interfaz con API socket simplificada y limpia), NetCraft (para construir datagramas IP o mensajes ICMP) y sobre NetTools (gestión y sincronización de hebras, y logging).

8.1.1 Estructura de ficheros y directorios de NetRat

```
NetEnviron ---> (NetEnviron es el escenario VNX)
   MPTCP.conf ---> (configuración de la tabla de rutas para el servidor)
   olive_mptcp_scenario.xml ---> (escenario VNX para MPTCP)
   olive_scenario.xml ---> (escenario VNX genérico)
NetRatClient ---> (aplicación cliente)
   Client.pm ---> (cliente STUN)
   LICENSE_GPL ---> (doble licencia MIT GPL)
   LICENSE_MIT ---> (doble licencia MIT GPL)
   NetRatConf.pm ---> (módulo de configuración del cliente)
   NetRat.pl ---> (punto de entrada del cliente)
   SockOps.pm ---> (módulo de operaciones de bajo nivel)
   Utils.pm ---> (módulo de operaciones de alto nivel)
NetRatServer ---> (aplicación servidor)
   Configurator.cpp ---> (clases para configuración estática y dinámica)
   Configurator.h ---> (módulo de configuración)
   LICENSE_GPL ---> (doble licencia MIT GPL)
   LICENSE_MIT ---> (doble licencia MIT GPL)
   Main.cpp ---> (main de la aplicación servidor)
   Makefile ---> (GNU makefile para Linux y FreeBSD)
   NetCraft ---> (links simbólicos a librería NetCraft)
      Crafter.h -> ../../../NetCraft/Crafter.h
      CraftIcmp6.h -> ../../../NetCraft/CraftIcmp6.h
      CraftIcmp.h -> ../../../NetCraft/CraftIcmp.h
      CraftIp6.h -> ../../../NetCraft/CraftIp6.h
      CraftIp.h -> ../../../NetCraft/CraftIp.h
      CraftRaw.h -> ../../../NetCraft/CraftRaw.h
      Headers.h -> ../../../NetCraft/Headers.h
      LibNetCraft.a -> ../../../NetCraft/LibNetCraft.a
      NetCraftConf.h -> ../../../NetCraft/NetCraftConf.h
   NetLayer ---> (links simbólicos a librería NetLayer)
      LibNetLayer.a -> ../../../NetLayer/LibNetLayer.a
      NetLayerConf.h -> ../../../NetLayer/NetLayerConf.h
      SockError.h -> ../../../NetLayer/SockError.h
      Socket.h -> ../../../NetLayer/Socket.h
      SockHandler.h -> ../../../NetLayer/SockHandler.h
      SockNetwork.h -> ../../../NetLayer/SockNetwork.h
      SockTransport.h -> ../../../NetLayer/SockTransport.h
   NetRat ---> (ejecutable)
   NetRat.conf ---> (fichero de configuración)
   NetRatConf.h ---> (fichero de configuración redundante)
   NetRatServer.sln ---> (Visual Studio)
   NetRatServer.vcxproj ---> (Visual Studio)
   NetRat-Valgrind.log ---> (Informe de uso correcto de la memoria dinámica)
   NetTools ---> (links simbólicos a librería NetTools)
      LibNetTools.a -> ../../../NetTools/LibNetTools.a
      Logger.h -> ../../../NetTools/Logger.h
      NetToolsConf.h -> ../../../NetTools/NetToolsConf.h
      Threads.h -> ../../../NetTools/Threads.h
      Tools.h -> ../../../NetTools/Tools.h
   SockImp.cpp ---> (logica de comunicaciones usando NetLayer)
   SockImp.h ---> (lógica de comunicaciones usando NetLayer)
   SockTrav.cpp ---> (NAT-traversal usando NetLayer y NetCraft)
   SockTrav.h ---> (NAT-traversal usando NetLayer y NetCraft)
   Svc.cpp ---> (lógica de los thread de comunicaciones)
   Svc.h ---> (lógica de los threads de comunicaciones)
   SvcNix.cpp ---> (thread de control, dependiente de plataforma)
   SvcNix.h ---> (thread de control, dependiente de plataforma)
   SvcWin.cpp ---> (thread de control, dependiente de plataforma)
   SvcWin.h ---> (thread de control, dependiente de plataforma)
   Trace.h
README.md
```

8.2 NetExample

NetExample [NETEXAMPLE] es una aplicación de tipo cliente servidor especialmente escrita para ilustrar como usar las librerías NetLayer, NetCraft y NetTools de forma conjunta. En su modo básico es un servidor de eco, pero con funcionalidad sniffer (SOCK_RAW capturando todo). Además tienen una funcionad sonar (como los murciélagos) en la que se usa NetCraft para construir mensajes ICMP ECHO REQUEST y REPLY.

8.2.1 Estructura de ficheros y directorios de NetExample

```
.
Base.cpp ---> (clase base tanto para cliente como para servidor)
Base.h ---> (clase base tanto para cliente como para servidor)
bin ---> (binarios)
   Linux
       Sonar
Client.cpp ---> (cliente)
Client.h ---> (cliente)
LICENSE_GPL ---> (licencia dual MIT GPL)
LICENSE_MIT ---> (licencia dual MIT GPL)
Main.cpp ---> (punto de entrada)
Makefile ---> (GNU makefile para Linux y FreeBSD)
NetCraftLib ---> (librería estática NetCraft)
   Crafter.h
   CraftIcmp6.h
   CraftIcmp.h
   CraftIp6.h
   CraftIp.h
   CraftRaw.h
   FreeBSD
      LibNetCraft.a
   Headers.h
   Linux
      LibNetCraft.a
   NetCraftConf.h
NetExample-Valgrind.log ---> (informe de uso eficiente de memoria dinámica)
NetLayerLib ---> (librería estática NetLayer)
   FreeBSD
      LibNetLayer.a
   Linux
      LibNetLayer.a
   NetLayerConf.h
   SockError.h
   Socket.h
   SockHandler.h
   SockNetwork.h
   SockTransport.h
NetToolsLib ---> (librería estática NetTools)
   FreeBSD
      LibNetTools.a
   Linux
      LibNetTools.a
   Logger.h
   NetToolsConf.h
   Threads.h
   Tools.h
Readme.txt
Server.cpp ---> (servidor)
Server.h ---> (servidor)
Sonar.cpp ---> (clase sonar para ilustrar uso de SOCK_RAW)
Sonar.h ---> (clase sonar para ilustrar uso de SOCK_RAW)
Utils.cpp ---> (misceláneo: GetOpts)
Utils.h
```

8.3 NetCraft

NetCraft [NETCRAFT] es una librería para construir datagramas IPv4 e IPv6, y mensajes ICMPv4 e ICMPv6. Es útil cuando se desea emplear SOCK_RAW (tanto como si la opción IP_HDRINCL está habilitada o como si no lo está).

8.3.1 Estructura de ficheros y directorios de NetCraft

```
.
 Crafter.h ---> (clase base)
 CraftIcmp6.cpp ---> (clase para ICMPv6)
 CraftIcmp6.h ---> (clase para ICMPv6)
 CraftIcmp.cpp ---> (clase para ICMPv4)
 CraftIcmp.h ---> (clase para ICMPv4)
 CraftIp6.cpp ---> (clase para IPv6)
 CraftIp6.h ---> (clase para IPv6)
 CraftIp.cpp ---> (clase para IPv4)
 CraftIp.h ---> (clase para IPv4)
 CraftRaw.cpp ---> (clase base SOCK_RAW)
 CraftRaw.h ---> (clase base SOCK_RAW)
 Headers.h ---> (Berkeley headers)
 LibNetCraft.a
 LICENSE_GPL ---> (licencia dual MIT GPL)
 LICENSE_MIT ---> (licencia dual MIT GPL)
 Makefile ---> (GNU makefile para Linux FreeBSD)
 NetCraftConf.h ---> (fichero de configuración)
 NetCraft.sln ---> (Visual Studio)
 NetCraft.vcxproj ---> (Visual Studio)
 NetCraft.vcxproj.filters ---> (Visual Studio)
 README.md
```

8.4 NetLayer

NetLayer [NETLAYER] es una librería estática para simplificar el uso de la API Berkeley Socket. Además está estructurada al estilo de la torre de protocolos TCP/IP como una cadena de responsabilidad en la que cada función de la API socket se resuelve en el nivel (aplicación, transporte o red) que le corresponde en función de las argumentos que necesite. Esta librería facilita la escritura de aplicaciones multiplataforma.

8.4.1 Estructura de ficheros y directorios de NetLayer

```
.
 LibNetLayer.a
 LICENSE_GPL
 LICENSE_MIT
 Makefile
 NetLayerConf.h
 NetLayer.sln
 NetLayer.vcxproj
 NetLayer.vcxproj.filters
 README.md
 SockError.cpp
 SockError.h
 Socket.h
 SockHandler.cpp
 SockHandler.h
 SockNetwork.cpp
 SockNetwork.h
 SockTransport.cpp
 SockTransport.h
```

8.5 NetTools

NetTools [NETTOOLS] es una librería estática para gestionar thread y sistema de trazas. Esta libría facilita la escritura de aplicaciones multiplataforma.

8.5.1 Estructura de ficheros y directorios de NetTools

```
.
 LibNetTools.a
 LICENSE_GPL
 LICENSE_MIT
 Logger.cpp
 Logger.h
 Makefile
 NetToolsConf.h
 NetTools.sln
 NetTools.vcxproj
 NetTools.vcxproj.filters
 README.md
 Threads.cpp
 Threads.h
 Tools.cpp
 Tools.h
```

9. Conclusiones y líneas futuras

Como bien se expone en [RFC6182] los nuevos dispositivos denominados *middleboxes* en la jerga (NATs, cortafuegos, Proxies, etc.) suponen un problema a la hora de usar protocolos de nivel de transporte que no sean los clásicos UDP o TCP. Si el diseño y la puesta en marcha de MPTCP son de una complejidad medio-elevada, hacer que SCTP funcione a gran escala se asemeja a una misión imposible. MPTCP presenta algunas ventajas sobre SCTP, como no requiere cambios significativos en la infraestructura existente o en las aplicaciones, ya que ha sido diseñado para adaptarse. Además MPTCP se beneficia de capacidad de enviar datos simultáneamente a través de múltiples caminos, mientras que SCTP, este mecanismo únicamente se usa como medio para aumentar la fiabilidad de la conexión. La funcionalidad multicamino de SCTP está, actualmente en fase de discusión.

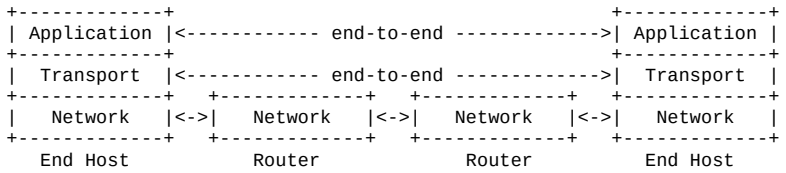


Ilustración 126: Internet en el mundo de los ponnies y de los arcoiris

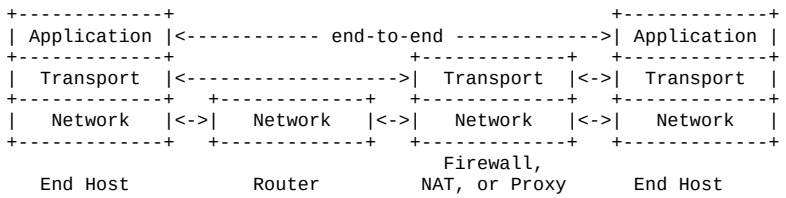


Ilustración 127: Internet en la realidad de forma medianamente aproximada

MPTCP, al ser una evolución continuista de TCP queda limitado a las aplicaciones que hagan uso de TCP (aquellas que hagan uso de UDP no pueden usar MPTCP sin, básicamente, reescribirlas completamente desde cero). El funcionamiento de MPTCP que se ha observado en este trabajo, aunque limitado y no demasiado extenso ha sido simplemente espectacular. Por su parte SCTP sobre el papel resulta un protocolo prometedor, pero en la práctica tiene que enfrentarse a dificultades aún mayores que MPTCP ya que requiere que la infraestructura sea modificada especialmente para su uso, y que las aplicaciones sean reescritas. De los dos modos de funcionamiento que ofrece SCTP en este trabajo se ha usado el sencillo por motivos de compatibilidad multiplataforma y para que la complejidad de las librerías y de las aplicaciones no fuera desproporcionada, esto es, a la hora de decidir si incluir o no en el trabajo final el modo de operación *one-to-many* de SCTP se evaluó los beneficios e inconvenientes siendo el resultado que no todas las plataformas lo soportan, la API de Socket para usarlo es diferente y complicada (y además algunas de las funciones de las que depende no están estandarizadas), la infraestructura de red no está adaptada y en general la balanza de beneficio/coste era negativa. El modo *one-to-one* funcionó realizando cambios básicos a las aplicaciones y librerías y en general la impresión fue positiva (ofrece funcionalidades interesantes como los *heartbeats*).

Sobre IPv6, lo único que queda comentar es que ojalá la migración de IPv4 a IPv6 se acelere y tenga lugar, y que esta migración traiga con sigo un aplanamiento de la red y una mejora de la conectividad *extremo-a-extremo*. Casi todos los dispositivos ya están adaptados a trabajar sobre IPv6, y en la práctica, no ha sido tan engorroso como me pensaba el trabajar con direcciones de 128 bits. IPv6 parece que ha aprendido de los errores de IPv4 (el hecho de que el campo hop limit de la cabecera solo tenga 8 bits no parece que vaya a ser un problema a largo plazo, ya que el número de saltos parece tiende a reducirse debido a internconexiones continentales).

Mover funcionalidad desde el nivel de aplicación al nivel de transporte podría facilitar la escritura de aplicaciones, pero desde un punto de vista purista y ortodoxo, no tengo muy claro que la funcionalidad multicamino haya que ubicarla en el nivel de transporte. La realidad es que el modelo de referencia es el que es y no hay capas intermedias entre transporte y aplicación.

Inicialmente, la idea que tenía para este trabajo era centrarme, únicamente, en SCPT para evaluar si podría representar o no una alternativa a TCP y UDP. No era coherente probar un protocolo de nueva generación sobre un nivel de transporte obsoleto, por lo que aumentó el espectro del trabajo para cubrir IPv6. IPv6 no puede trabajar sin ICMPv6 por lo que también hubo que incluir a este último. Por recomendación de mi tutor además se incluyó MPTCP, lo cual fue muy buena sugerencia. En este trabajo se ha migrado una aplicación de IPv4 a IPv6 y se ha probado con SCTP y MPTCP. Los resultados se ha validad en entornos virtualizados mediate VNX. SCTP no parece que vaya a comerle terreno a TCP y UDP. MPTCP tiene la desventaja de que únicamente cubre nivel de transporte orientado a conexión y fiable. La funcionalidad multicamino es en general buena idea para cierto tipo de aplicaciones.

Las líneas futuras podrían ser, intentar comunicar aplicaciones mediante MPTCP y SCTP en el Internet real, en el caso de SCTP mediante direcciones públicas debido a la interferencia con los NATs. Y sobre todo evaluar las prestanciones de SCTP y MPTCP frente a TCP y UDP, lo cual ha sido imposible en este trabajo. Este trabajo ha sido una primera toma de contacto con estos protocolos de nivel de trasnporte.

En la línea de IPv6, este trabajo ha asentado los conocimientos adquiridos en la asignatura *Temas Avanzados de Redes de Ordenadores*, y se ha tenido un primer contacto, tanto teórico como práctico, con NAT64 y DNS64. Se ha adivinado el camino que pretenden seguir los operadores de telecomunicación para migrar su red a IPv6. El rol de DNS64 por sí solo, como elemento conmutador en la transición dará mucho que hablar. Además el uso de CgNAT como antesala a NAT64 también parece que va a ser el camino a seguir. En este trabajo se ha pasado a gran altura sobre estos conceptos. Un estudio más meticuloso también podría suponer nuevas líneas futuras de trabajo.

En lo referente a las aplicaciones, se ha pulido, se han aprendido nuevos patrones de diseño y se han creado (es lo que creo) nuevos patrones de diseño, pero aún así las técnicas de programación evolucionan. Por ejemplo, está de moda hoy en día lo que se conoce como *Test-Driven-Development*, que consiste en usar un conjunto de casos de prueba unitarios como especificación técnica de la aplicación. A nivel de programación sería interesante experimentar con esta técnica de programación y verificar si realmente aumenta la productividad y mejora la calidad del código.

Los patrones de diseño son buenas ideas que están probadas y que funcionan. No siempre es necesario reinventar la rueda. Otra línea futura es lo que se conoce como *Pattern-Oriented-Design*, que en el fondo es lo que ha hecho con la libraría NetLayer: se usó un patrón conocido como la cadena de responsabilidad [GoF], ya que es el que más similitudes tenía con el modelo en torre de protocolos de TCP/IP. Aplicar el diseño orientado a patrones también puede resultar interesante para conocer si mejora productividad y aumenta la calidad del software.

Respecto a la conectividad extremo a extremo, solo hay una manera para siempre asegurar que es posible establecer una conexión entre dos extremos: el uso de nodos intermedios al estilo de TURN.

10. Anexo A: escenarios VNX de pruebas

10.1 Escenario básico

Escenario básico formado por un cliente, un servidor y dos routers Olive. Se soporta tanto IPv4 como IPv6

```
<?xml version="1.0" encoding="UTF-8"?>

<vnx xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="/usr/share/xml/vnx/vnx-2.00.xsd">
  <global>
    <version>2.0</version>
    <scenario_name>Olive</scenario_name>
    <automac offset="0"/>
    <vm_mgmt type="none" />
    <vm_defaults>
            <console id="0" display="no"/>
            <console id="1" display="yes"/>
    </vm_defaults>
  </global>

  <net name="Lan1"  mode="virtual_bridge" />
  <net name="Lan2"  mode="virtual_bridge" />
  <net name="Lan3"  mode="virtual_bridge" />
  <net name="ptp12" mode="virtual_bridge" />

  <!-- NODES -->
  <vm name="client" type="libvirt" subtype="kvm" os="linux" exec_mode="sdisk">
    <filesystem type="cow">/usr/share/vnx/filesystems/rootfs_client</filesystem>
    <mem>256M</mem>
    <if id="1" net="Lan1">
      <ipv4>7.7.7.7/24</ipv4>
      <ipv6>2001:db8:0:1::7/64</ipv6>
    </if>
    <if id="2" net="Lan3">
      <ipv4>10.7.7.7/8</ipv4>
    </if>
    <route type="ipv4" gw="10.0.0.254">default</route>
    <route type="ipv4" gw="7.7.7.3">7.7.11.0/24</route>
    <route type="ipv6" gw="2001:db8:0:1::1">2001:db8:0:5::/64</route>
  </vm>

  <vm name="server" type="libvirt" subtype="kvm" os="linux" exec_mode="sdisk">
    <filesystem type="cow">/usr/share/vnx/filesystems/rootfs_server</filesystem>
    <mem>256M</mem>
    <if id="1" net="Lan2">
      <ipv4>7.7.11.7/24</ipv4>
      <ipv6>2001:db8:0:5::3/64</ipv6>
    </if>
    <if id="2" net="Lan3">
      <ipv4>10.7.11.7/8</ipv4>
    </if>
    <route type="ipv4" gw="10.0.0.254">default</route>
    <route type="ipv4" gw="7.7.11.5">7.7.7.0/24</route>
    <route type="ipv6" gw="2001:db8:0:5::1">2001:db8:0:1::/64</route>
  </vm>

  <vm name="r1" type="libvirt" subtype="kvm" os="olive">
    <filesystem type="cow">/usr/share/vnx/filesystems/rootfs_olive</filesystem>
    <mem>256M</mem>
    <conf>conf/example_olive/r1.conf</conf>
    <if id="1" net="Lan1" name="fxp0">
      <ipv4>7.7.7.3/24</ipv4>
      <ipv6>2001:db8:0:1::1/64</ipv6>
    </if>
    <if id="2" net="ptp12" name="fxp1">
      <ipv4>7.7.9.3/24</ipv4>
      <ipv6>2001:db8:0:3::1/64</ipv6>
    </if>
    <if id="3" net="Lan3" name="fxp2">
      <ipv4>10.7.9.3/8</ipv4>
    </if>
    <route type="ipv4" gw="7.7.9.5">7.7.11.0/24</route>
    <route type="ipv6" gw="2001:db8:0:3::3">2001:db8:0:5::/64</route>
```

```
  </vm>

  <vm name="r2" type="libvirt" subtype="kvm" os="olive">
    <filesystem type="cow">/usr/share/vnx/filesystems/rootfs_olive</filesystem>
    <mem>256M</mem>
    <conf>conf/example_olive/r1.conf</conf>
    <if id="1" net="Lan2" name="fxp0">
      <ipv4>7.7.11.5/24</ipv4>
      <ipv6>2001:db8:0:5::1/64</ipv6>
    </if>
    <if id="2" net="ptp12" name="fxp1">
      <ipv4>7.7.9.5/24</ipv4>
      <ipv6>2001:db8:0:3::3/64</ipv6>
    </if>
    <if id="3" net="Lan3" name="fxp2">
      <ipv4>10.7.9.5/8</ipv4>
    </if>
    <route type="ipv4" gw="7.7.9.3">7.7.7.0/24</route>
    <route type="ipv6" gw="2001:db8:0:3::1">2001:db8:0:1::/64</route>
  </vm>

  <host>
    <hostif net="Lan3">
       <ipv4>10.0.0.254/8</ipv4>
    </hostif>
  </host>

</vnx>
```

10.2 Escenario MPTCP

Escenario de pruebas de MPTCP. La diferencia respecto al escenario propuesto en la sección anterior es que se ha prescindido de uno de los routers, ya que no hace falta, y se ha aumentado el número de interfaces en el lado del servidor, para aumentar el número de posibles rutas.

```
<?xml version="1.0" encoding="UTF-8"?>

<vnx xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="/usr/share/xml/vnx/vnx-2.00.xsd">
  <global>
    <version>2.0</version>
    <scenario_name>Olive</scenario_name>
    <automac offset="0"/>
    <vm_mgmt type="none" />
    <vm_defaults>
            <console id="0" display="no"/>
            <console id="1" display="no"/>
    </vm_defaults>
  </global>

  <net name="Lan1"  mode="virtual_bridge" />
  <net name="Lan2"  mode="virtual_bridge" />
  <net name="Lan3"  mode="virtual_bridge" />
  <net name="Lan4"  mode="virtual_bridge" />

  <!-- NODES -->
  <vm name="client" type="libvirt" subtype="kvm" os="linux" exec_mode="sdisk">
    <filesystem type="cow">/usr/share/vnx/filesystems/rootfs_client</filesystem>
    <mem>256M</mem>
    <if id="1" net="Lan1">
      <ipv4>7.7.7.7/24</ipv4>
      <ipv6>2001:db8:0:1::7/64</ipv6>
    </if>
    <if id="2" net="Lan4">
      <ipv4>10.7.7.7/8</ipv4>
    </if>
    <route type="ipv4" gw="10.0.0.254">default</route>
    <route type="ipv4" gw="7.7.7.1">7.7.9.0/24</route>
    <route type="ipv4" gw="7.7.7.1">7.7.11.0/24</route>
    <route type="ipv6" gw="2001:db8:0:1::1">2001:db8:0:3::/64</route>
    <route type="ipv6" gw="2001:db8:0:1::1">2001:db8:0:5::/64</route>
  </vm>

  <vm name="server" type="libvirt" subtype="kvm" os="linux" exec_mode="sdisk">
    <filesystem type="cow">/usr/share/vnx/filesystems/rootfs_server</filesystem>
    <mem>256M</mem>
    <if id="1" net="Lan2">
      <ipv4>7.7.9.7/24</ipv4>
      <ipv6>2001:db8:0:3::3/64</ipv6>
    </if>
    <if id="2" net="Lan3">
      <ipv4>7.7.11.7/24</ipv4>
      <ipv6>2001:db8:0:5::3/64</ipv6>
    </if>
    <if id="3" net="Lan4">
      <ipv4>10.7.9.7/8</ipv4>
    </if>
  </vm>

  <vm name="r1" type="libvirt" subtype="kvm" os="olive">
    <filesystem type="cow">/usr/share/vnx/filesystems/rootfs_olive</filesystem>
    <mem>256M</mem>
    <conf>conf/example_olive/r1.conf</conf>
    <if id="1" net="Lan1" name="fxp0">
      <ipv4>7.7.7.1/24</ipv4>
      <ipv6>2001:db8:0:1::1/64</ipv6>
    </if>
    <if id="2" net="Lan2" name="fxp1">
      <ipv4>7.7.9.1/24</ipv4>
      <ipv6>2001:db8:0:3::1/64</ipv6>
    </if>
    <if id="3" net="Lan3" name="fxp2">
      <ipv4>7.7.11.1/24</ipv4>
      <ipv6>2001:db8:0:5::1/64</ipv6>
```

```
      </if>
      <if id="4" net="Lan4" name="fxp3">
        <ipv4>10.1.1.1/8</ipv4>
      </if>
    </vm>

    <host>
      <hostif net="Lan4">
         <ipv4>10.0.0.254/8</ipv4>
      </hostif>
    </host>

  </vnx>
```

10.2.1 Configuración de la tabla de rutas en el servidor

A continuación se muestra la configuración usada en la tabla de rutas del servidor del escenario de pruebas de MPTCP:

```
ip rule add from 7.7.9.7 table 1
ip rule add from 7.7.11.7 table 2
ip route add 7.7.9.0/24 dev eth1 scope link table 1
ip route add default via 7.7.9.1 dev eth1 table 1
ip route add 7.7.11.0/24 dev eth2 scope link table 2
ip route add default via 7.7.11.1 dev eth2 table 2
ip route add default scope global nexthop via 7.7.9.1 dev eth1
ip -6 rule add from 2001:db8:0:3::3 table 1
ip -6 rule add from 2001:db8:0:5::3 table 2
ip -6 route add 2001:db8:0:3::/64 dev eth1 scope link table 1
ip -6 route add default via 2001:db8:0:3::1 dev eth1 table 1
ip -6 route add 2001:db8:0:5::/64 dev eth2 scope link table 2
ip -6 route add default via 2001:db8:0:5::1 dev eth2 table 2
ip -6 route add default scope global nexthop via 2001:db8:0:3::1 dev eth1
ip rule show
ip -6 rule show
ip route
ip -6 route
ip route show table 1
ip -6 route show table 1
ip route show table 2
ip -6 route show table 2
```

11. Anexo B: direcciones broadcast y multicast

A lo largo de la elaboración de trabajo se ha dedicado muchas horas trabajando con direcciones IP. En IPv6, las direcciones multicast se usan intensivamente, por ejemplo en el protocolo Neighbor Discovery, tal y como se ha observado en los ejemplos expuestos a lo largo de este trabajo. A continuación se describen en un poco más de detalle este tipo de direcciones.

Las direcciones de broadcast y multicast sirven para enviar mensajes a múltiples destinatarios y/o para el descubrimiento de servidores [TCP/IP-1-94] [TCP/IP-1-12] [UNIX-1].

```
Ethernet multicast address (01:00:00:00:00:00)
Ethernet broadcast address (ff:ff:ff:ff:ff:ff)
```

Ilustración 128: direcciones ethernet multicast y broadcast

IPv4

Tipos de direcciones de broadcast en IPv4:

- Limited broadcast address 255.255.255.255
- Net-direct broadcast address (por ejemplo, para= clase A netid.255.255.255)
- Subnet-direct broadcast address (por ejemplo, para class B 128.1, con una subred 128.1.2 -> 128.1.2.255)

Direcciones multicast en IPv4

```
                    28 bits
          ------------------------------
Class D: |1|1|1|0| multicast group ID | 224.0.0.0 - 239.255.255.255
          ------------------------------

224.0.0.0   - 224.0.0.255       Local network control; not forwarded
224.0.1.0   - 224.0.1.255       Internetwork control; forwarded normally
224.0.2.0   - 224.255.255       Ad hoc block 1
224.1.0.0   - 224.1.255.255     Reserved
224.2.0.0   - 224.2.255.255     SDP/SAP
224.3.0.0   - 224.4.255.255     Ad hoc block II
224.5.0.0   - 224.255.255.255   Reserved
225.0.0.0   - 224.255.255.255   Reserved
232.0.0.0   - 232.255.255.255   Source-specific-multicast (SSM)
233.0.0.0   - 233.251.255.255   GLOP
233.252.0.0 - 233.255.255.255   Ad hoc block III (233.252.0.0/24 reserved for documentation)
234.0.0.0   - 234.255.255.255   Unicast-prefix-based IPv4 multicast addresses
235.0.0.0   - 238.255.255.255   Reserved
239.0.0.0   - 239.255.255.255   Administrative scope
```

Ilustración 129: direcciones IPv4 multicast [TPC/IP-1-94] [TCP/IP-1-12]

IPv6

```
|   8    |  4 |  4 |                  112 bits                   |
+------ -+----+----+---------------------------------------------+
|11111111|flgs|scop|                  group ID                   |
+--------+----+----+---------------------------------------------+
```

```
    0  reserved
    1  Interface-Local scope
    2  Link-Local scope
    3  reserved
    4  Admin-Local scope
    5  Site-Local scope
    6  (unassigned)
    7  (unassigned)
    8  Organization-Local scope
    9  (unassigned)
    A  (unassigned)
    B  (unassigned)
    C  (unassigned)
    D  (unassigned)
```

```
E  Global scope
F  reserved
```

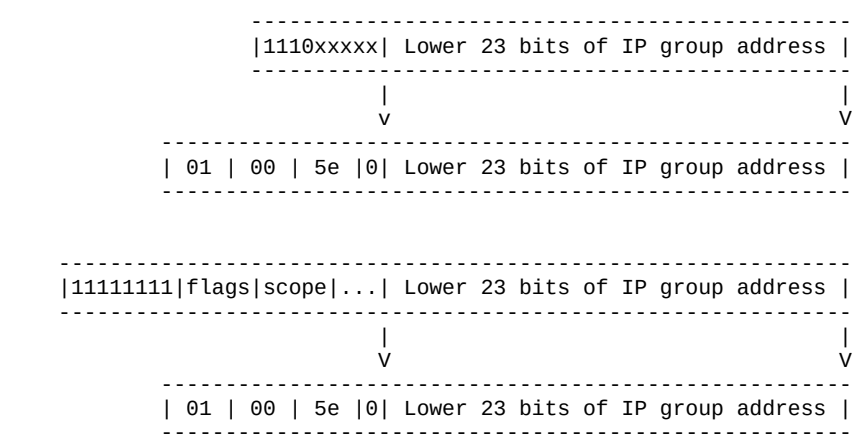
Ilustración 130: direcciones IPv6 multicast

Direcciones IPv6 multicast reservadas:

```
ff01::1     Node    All nodes
ff01::2     Node    All routers
ff01::fb    Node    mDNSv6
ff02::1     Link    All node
ff02::2     Link    All routers
ff02::fb    Link    mDNSv6
ff02::1:2   Link    All DHCP agents
ff05::2     Site    All routers
ff05::fb    Site    mDNSv6
ff05::1:3   Site    All DHCP servers
```

Ilustración 131: direcciones IPv6 multicast reservadas

Convirtiendo una dirección IP multicast a una dirección MAC/Ethernet 802 multicast:

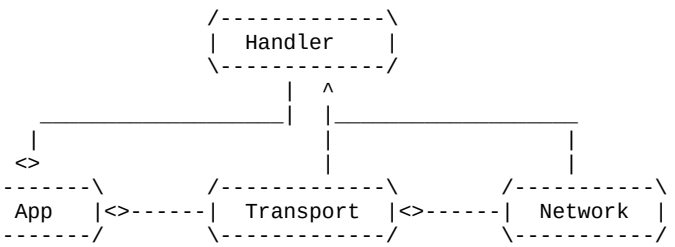


12. Anexo C: NetExample (documentación)

La aplicación de ejemplo NetExampe [NETEXAMPLE] se escribió con el fin de ilustrar, a modo de tutorial, la manera en la que las librerías desarrolladas para este trabajo ([NETLAYER], [NETCRAFT] y [NETTOOLS]) deben usarse. Todos los ficheros que componen el proyecto NetExample están documentados, pero en el fichero *main.cpp* en concreto se prestó especial esfuerzo en que esta documentación fuera precisa y completa, describiendo en detalle las librerías en sí, algunos patrones de diseño y cómo usarlas. En concreto, la documentación presente en *main.cpp* es especialmente útil para inicializar las estructuras necesarias para usar NetLayer, a la vez que muestra como gestionar las trazas con NetTools.

A continuación se reproduce el contenido de dicho fichero.

```
/* Socket API wrapper is offered by NetLayer library
   It is implemented as a Chain of Responsibility where its layer is a link in the chain
   This fits with the Stack of protocol layers (TCP/IP reference model):
   each layer logic is implemented a in a separate class, objects communicate through homogeneous API
   and pass the message to each other layer */

/* NetLayer:
   =========
```




```
   (No design constraint set at application layer: how the application uses NetLayer (Handler, Transport, Network)
    it is a design choice)

   The public interface is

   void Socket(std::shared_ptr<BaseSock>);
   void Connect(std::shared_ptr<BaseSock>);
   void Bind(std::shared_ptr<BaseSock>);
   void Listen(std::shared_ptr<BaseSock>);
   void Accept(std::shared_ptr<BaseSock>);
   void Close(std::shared_ptr<BaseSock>);
   void Setsockopt(std::shared_ptr<BaseSock>);
   void Getsockopt(std::shared_ptr<BaseSock>);
   void Getaddrinfo(std::shared_ptr<BaseSock>);
   int Recv(std::shared_ptr<BaseSock>);
   int Send(std::shared_ptr<BaseSock>);
   int Recvfrom(std::shared_ptr<BaseSock>);
   int Sendto(std::shared_ptr<BaseSock>);

   where std::shared_ptr was used to ease the memory management

   Where does each syscall gets managed in the chain of responsibilitity?
   This diagram shows how the Socket API syscalls get resolved internally


    SysCall          client             Transport             Network
    -------          ------             ---------             -------
    Socket             |--------------------0--------------------->X
    Bind               |-------------------------------------------->X
    Listen             |-------------------->X
    Accept             |-------------------------------------------->X
    Connect            |-------------------------------------------->X
    Recv               |-------------------->X
    Send               |-------------------->X
    Recvfrom           |-------------------------------------------->X
```

```
 Sendto             |------------------------------------------>X
 Setsockopt         |-------------------->X..................>X
 Getsockopt         |-------------------->X..................>X
 Getaddrinfo        |------------------------------------------>X
```

What's the WHY of this? Socket API has been there for a long while, and working very well
The purpose is not to reinvent the wheel. I wrote NetLayer library because AF_INET and
AF_INET6 compatible code is a tedious and complex task: lot of effort is spent dealing
with the socket API due to casting requirements different socket data structures and corner cases
so better to spend energy on that only once creating a "less user-hostile" API, minimizing
error probability.

Said that, what's the WHY of this? Let's use Bind syscall to illustrate the idea. Literally taken for
R. Stevens bible "The bind function assigns a local protocol address to a socket. With the Internet
protocols, the protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6
address, along with the 16-bit TCP or UDP (or SCTP) port number". Lot of dense and mixed information
in a single sentence. TCP/IP reference model defines a stack of protocols, each layer with well defined
purpose offering a specific service to the upper layer by means of a interface. And here comes one of
the problems, the Socket API intermix layers and protocol versions (and even protocol families.
In Bind syscall we have to pass a Network layer IPv4 or IPv6 address (Internet family, AF_INET or AF_INET6)
and a Transport layer protocol number. Now try to write a simple client server application that can
use IPv4 or IPv6, and TCP or UDP. Doing that, for non-experts (like me) it is error prone.

So what's the WHY of NetLayer? NetLayer focus on Internet protocol families AF_INET and AF_INET6 reducing scope.
It offers a Socket data structure (and a pool of them) that is the same of any kind of Internet socket
and only concern is to feed it with Network layer address, port numbers, options or whatever is needed,
so all the functions in the library have exactly the same, signature. So coming back to the Bind
example, we have instantiated a socket object, filled it with the family, type and protocol we want to
use, set the corresponding network address and transport port number and invoke syscall passing as
argument the socket object (and invoke Socket). Internally and transparently to the application using
the library, NetLayer is implemented as chain or responsibility where the socket object is the message
passed between links. When Bind is invoked from the Application layer, transport link tries to serve it,
but the transport link is only aware of Transport layer information, and since bind requires IP addr,
it is unable to manage it, so it delegates to the successor link, which it is the Network layer. And
this is the reasoning. What is achieved is a clear distintiction between layers, the way the TCP/IP
does, being transport link only aware of transport data (TCP, UDP or SCTP related) and network link
responsible of managing all IPv4 and IPv6 difference (releasing the user of having to do that manually)

This internal layout eases the write a cleaner API and tries to ressemble the same architecture than
TCP/IP reference model
*/

```
/* Network Layer handler declaration:
   network shared_ptr stores a SockIPvX object
   SockIPvX hides Socket API complexities related with and manages both, IPv4 and IPv6 */
std::shared_ptr<_NetLayer::SockHandler> network;

/* Transport Layer handler declaration:
   transport shared_ptr stores a SockTcp, SockUdp, SockSctp (even a SockIcmp) object
   SockTcp, SockUdp, SockSctp and SockIcmp manage Socket operations that are placed at transport layer
   If an operation is unable to be serverd by any of these objects, it is delegated to the next link of
   the chain (network object). This situation happens for example with Sendto, where we need destination
   IP address (and port for transport), but since IP address is L3 data, transport layer objects are unaware
   of if, hence, delegate it to next link. */
std::shared_ptr<_NetLayer::SockHandler> transport;

/* Although ICMP falls into Network Layer protocol category it has been placed
   under SockTransport.h within NetLayer (it is travelling over IPv4 or IPv6 and
   it is independent of of IPv4 or IPv6 Socket API data structures although
   ICMPv4 and ICMPv6 are to different protocols
   Placing ICMP in SockTransport also responds to the fact that ICMP doesn't require
   Any IP L3 data. Port number in Socket objects must be explicitly set to zero to
   avoid EINVAL error when trying to send a message */
std::shared_ptr<_NetLayer::SockHandler> control;

/* Application Layer handler declaration:
   application shared_ptr stores and Application layer object that makes use of Socket API
   The aim of this is to offer a simple API hiding low-level Berkeley socket API oddities
   This is not part of NetLayer library. This the application code that uses Socket API
   through NetLayer */
std::shared_ptr<_Application::Base> application;

/* The other application object, which will be working with ICMPv4 or ICMPv6
   in order to illustrate the usage of NetCraft library and how deal with
   raw sockets using NetLayer library */
std::shared_ptr<_Application::Base> sonar;
```

```
/* Socket declaration:
   socket shared_ptr stores an ActiveSock object using polymorphism through BaseSock. This is
   like this because ActiveSock is dependent on the IP version and BaseSock offer a clean
   interface L3 version indenpendent that, for most of the times, it is enought to handel the object.
   The socket is the "message" passed between different layers and contains all socket information */

/* socket object will be used to store TCP or UDP mode socket data
   This object will be used by the TCP and UDP client/server side */
std::shared_ptr<_NetLayer::BaseSock> socket;

/* sonarSocket will be used to store ICMPv4 or ICMPv6 socket data
   This object will be used by the ICMPv4 or ICMPv6 sonar side
   The name of sonar it is just a funny joke (it merely send PINGs or "STALK"
   using SOCK_RAW type sockets). Using SOCK_RAW type socket let us to illustrate
   how to manually craft IPv4, IPv6, ICMPv4 or ICMPv6 datagrams using NetCraft library */
std::shared_ptr<_NetLayer::BaseSock> sonarSocket;

/* Instantiate and initialize socket, application, transport and network objects:
   Socket needs to know IP addreses and port number
   Transport needs to know which protocol is to be used (UDP or TCP)
   Network needs to know with IP version is to be used (4 or 6) */
InitApplicationSocket(options, socket);
InitNetwork(options, network);
InitTransport(options, transport);
InitApplication(options, application);

/* Instantiate and initialize raw socket for ICMPv4 or ICMPv6
   Initialize ICMP link */
InitSonarSocket(options, sonarSocket);

/* InitApplicationSocket and InitSonarSocket are basically the same with the exception
   that both sockets are different: socket is SOCK_DGRAM or SOCK_STREAM while sonar is
   SOCK_RAW (so port number has to be explicitly set to zero), hence there are two
   different init functions for two different kinds of sockets */

/* The network layer is the same, as in TCP/IP reference model UDP, TCP and ICMP all share IP */
InitControl(options, control);
InitSonar(options, sonar);

/* At this point all the links of the chain have been instantiated and initialized
   The message passed between links is the socket object
   The order of succesor is application->transport->network */

/* Transport and Control rely over Network:

        Transport and Control use IPv4 or IPv6 Network protocols:

            /-------------\        /-----------\
            | Transp/Ctrl |<>------|  Network  |
            \-------------/        \-----------/
 */

/* If transport is not able to manage a Socket syscall, it delegates it to network,
   that it is the next in the chain of responsibility, so we set successor to network */
transport->SetSucc(network);

/* If ICMP control link it is not able to manage a Socket syscall, it delegates it to network */
control->SetSucc(network);

/* We have used aggregation in Application layer rather than inheritance because
   We are not creating an application protocol itself
   Our application makes use of TCP or UDP directly, so let's set the handler
   This how personally I've decided, but other designs are possible */

/* Application relies over Transport and Control relies over Network:

        Application uses TCP or UDP Transport protocols:

            /-------------\        /-----------\
            | Application |<>------| Transport |
            \-------------/        \-----------/

        Sonar uses ICMPv4 or ICMPv6 Control protocols:

            /-------------\        /-----------\
            |   Sonar     |<>------|  Control  |
            \-------------/        \-----------/
```

```
     */

    /* Set the transport layer SockHandler that will be used by the application */
    application->SetHandler(transport);

    /* The sonar sample application relays directly over network layer, concretely over ICMP */
    sonar->SetHandler(control);

    /* The message passed from one link of the chain to the successor link is the socket object */

    /* Set the socket object that will be used to store Sockets info and passed through links of the chain */
    application->SetSocket(socket);

    /* Set its corresponding socket object as well */
    sonar->SetSocket(sonarSocket);

    /* And this is how NetLayer is used. From here on, the design of Client, Server and Sonar classes
       is my personal taste for writing this example. NetLayer does not force any programming pattern
       in the application layer, it only offer a simplified API toward AF_INET and AF_INET6 socket families */

    /* Once everything is instantiated and initialize, run the main logic */
    Start(options, application, sonar);

    logger->Log("[Main]: cerrando el chiringuito, bye bye");

    /* Let's decrement reference counters of the shared_ptr */
    socket.reset();
    network.reset();
    transport.reset();
    application.reset();
    sonarSocket.reset();
    sonar.reset();
}

/*
 * Initialize the Socket object
 * It contains all socket relevant data (local and remote IP addresses, port numbers, etc.)
 * Internally in NetLayer the socket object is the link passed between different links in the chain
 * For example, when transport layer cannot perform an operation because it is dependent on
 * IPv4 or IPv6 data, it delegates responsibility to Network layers passing the message
 */
void InitApplicationSocket(_Utils::Options &options, std::shared_ptr<_NetLayer::BaseSock> &socket)
{
    auto logger=_NetTools::Singleton<_NetTools::Logger>::Instance();
    logger->Log("[InitApplicationSocket]: initializing socket object");

    /* Distiguish between IPv4 and IPv6 because structures are incompatible:
       in_addr or in6_addr, and sockaddr_in or sockaddr_in6 are completely different
       One of the goals of NetLayer library is to minimize the Socket API complexities due to IPv4 and IPv6 differences
*/
    if (options.Six()) {
        logger->Log("[InitApplicationSocket]: instantiating IPv6 socket object");
        auto tempSocket = std::make_shared<_NetLayer::ActiveSock<_NetLayer::IA6, _NetLayer::SAI6>>(IPv6);
        /* Set the local address and port part of the socket tuple
           We distinguish between Client (the active party) and Server (the passive party) */
        tempSocket->SetAddr(options.Client()?Active:Passive, options.Addr());
        tempSocket->SetPort(options.Client()?Active:Passive, options.Port());
        /* shared_ptr copy constructor in action */
        socket = tempSocket;
    } else {
        logger->Log("[InitApplicationSocket]: instantiating IPv4 socket object");
        auto tempSocket = std::make_shared<_NetLayer::ActiveSock<_NetLayer::IA, _NetLayer::SAI>>(IPv4);
        tempSocket->SetAddr(options.Client()?Active:Passive, options.Addr());
        tempSocket->SetPort(options.Client()?Active:Passive, options.Port());
        /* shared_ptr copy constructor in action */
        socket = tempSocket;
    }

    /* Socket will be used through BaseSock interface, but to set Addr and Port ActiveSock interface has been used
       This is done like this because the aim was to have a clean API IP protocol and oddities independent
       The socket is the "message" and its hierachy is:
       DataSock is the container of socket data, BaseSock offers and IP independent interface
       ActiveSocket takes into account differences between IPv4 and IPv6 but NetLayer handles them internally
       To use it the only requirement is to specify which IP version to use and pass the corresponding IP address and
       transport layer port */

    /* Do we have a passive socket that will listen for incoming connections or messages
```

```
       or do we have an active socket that will perform active open or send messages ???
       Yeah, this is a error prone and should be done in a smarter why because by default
       the socket object is configured as passive, so if in the client we forget to set its
       mode to active it crash when trying to find a socket descripor in the wrong place
       ModeClient sets socket as NOT passive and ModeServer sets socket as passive */
    socket->SetMode(options.Client()?ModeClient:ModeServer);
}

/*
 * Initialize the link in the chain of responsibility that corresponds with the Network layer in TCP/IP
 * In this example Network is the last link, so succesor will be a null pointer
 * Network can be IPv4 or IPv6
 */
void InitNetwork(_Utils::Options &options, std::shared_ptr<_NetLayer::SockHandler> &network)
{
    auto logger=_NetTools::Singleton<_NetTools::Logger>::Instance();
    logger->Log("[InitNetwork]: initializing network layer");

    /* Create a SockIPv4 or SockIPv6 instance */
    if (!options.Six()) {
        /* AF_INET */
        logger->Log("[InitNetwork]: instantiating IPv4 network layer object");
        network = std::make_shared<_NetLayer::SockIPvX<_NetLayer::IA, _NetLayer::SAI>>(IPv4);
    } else {
        /* AF_INET6 */
        logger->Log("[InitNetwork]: instantiating IPv6 network layer object");
        network = std::make_shared<_NetLayer::SockIPvX<_NetLayer::IA6, _NetLayer::SAI6>>(IPv6);
    }
}

/*
 * Initialize the link in the chain of responsibility that corresponds with the Transport layer in TCP/IP
 * Possibilities are: UDP, TCP, SCTP (and ICMP). ICMP is in Network layer withing TCP/IP, but it was placed here since
it uses IP
 * In this example possibilities have been limited to UDP and TCP
 */
void InitTransport(_Utils::Options &options, std::shared_ptr<_NetLayer::SockHandler> &transport)
{
    auto logger=_NetTools::Singleton<_NetTools::Logger>::Instance();
    logger->Log("[InitTransport]: initializing transport layer");

    if (options.Udp()) {
        /* Type SOCK_DATAGRAM, Protocol IPPROTO_UDP */
        logger->Log("[InitTransport]: instantiating UDP transport object");
        transport = std::make_shared<_NetLayer::SockUdp>(DATAGRAM, DATAGRAM_UDP);
    } else {
        /* Type SOCK_STREAM, Protocol IPPROTO_TCP */
        logger->Log("[InitTransport]: instantiating TCP transport object");
        transport = std::make_shared<_NetLayer::SockTcp>(STREAM, STREAM_TCP);
    }
}

/* Instantiate and initialize Client */
void InitClient(_Utils::Options &options, std::shared_ptr<_Application::Base> &application)
{
    auto logger=_NetTools::Singleton<_NetTools::Logger>::Instance();
    logger->Log("[InitClient]");

    if (options.Udp()){
        application = std::make_shared<_Application::ClientUdp>();
    } else {
        application = std::make_shared<_Application::ClientTcp>();
    }
}

/* Instantiate and initialize Server */
void InitServer(_Utils::Options &options, std::shared_ptr<_Application::Base> &application)
{
    auto logger=_NetTools::Singleton<_NetTools::Logger>::Instance();
    logger->Log("[InitServer]");

    if (options.Udp()){
        application = std::make_shared<_Application::ServerUdp>();
    } else {
        application = std::make_shared<_Application::ServerTcp>();
    }
}
```

```
/*
 * Initialize the application layer object that makes use of Socket API
 * NetLayer can be used through inheritance, composition or aggregation
 * The library design does not force the use of any concrete pattern
 * In this example aggregation has been used
 */
void InitApplication(_Utils::Options &options, std::shared_ptr<_Application::Base> &application)
{
    auto logger=_NetTools::Singleton<_NetTools::Logger>::Instance();
    logger->Log("[InitApplication]: initializing application layer");

    if (options.Client()) {
        InitClient(options, application);
    } else {
        InitServer(options, application);
    }
}

/*
 * To work with SOCK_RAW type sockets it is always funny.
 * Yes, the signature is weird, since we have to access API of ActiveSocket (not accessible from BaseSocket)
 */
void InitSonarSocket4(_Utils::Options &options, std::shared_ptr<_NetLayer::ActiveSock<_NetLayer::IA, _NetLayer::SAI>>
&socket)
{
    auto logger=_NetTools::Singleton<_NetTools::Logger>::Instance();
    logger->Log("[InitSonarSocket4]: initializing socket object");
    if (options.Client()) {
        /* Configure the IPv4 address of the remote machine we want to range */
        socket->SetAddr(Active, options.Addr());
        socket->SetPort(Active, 0);
        /* Configure local IPv4 address with 0.0.0.0 */
        socket->SetAddr(Passive, IN4_ADDR_ANY);
        socket->SetPort(Passive, 0);
    } else {
        /* Configure local IPv4 address with 0.0.0.0 */
        socket->SetAddr(Passive, options.Addr());
        socket->SetPort(Passive, 0);
    }
}

/*
 * To work with SOCK_RAW type sockets it is always funny.
 * Yes, the signature is weird, since we have to access API of ActiveSocket (not accessible from BaseSocket)
 */
void InitSonarSocket6(_Utils::Options &options, std::shared_ptr<_NetLayer::ActiveSock<_NetLayer::IA6, _NetLayer::SAI6>>
&socket)
{
    auto logger=_NetTools::Singleton<_NetTools::Logger>::Instance();
    logger->Log("[InitSonarSocket6]: initializing socket object");
    if (options.Client()) {
        /* Configure the IPv6 address of the remote machine we want to range */
        socket->SetAddr(Active, options.Addr());
        socket->SetPort(Active, 0);
        /* Configure local IPv6 address with :: */
        socket->SetAddr(Passive, IN6_ADDR_ANY);
        socket->SetPort(Passive, 0);
    } else {
        /* Configure local IPv6 address with :: */
        socket->SetAddr(Passive, options.Addr());
        socket->SetPort(Passive, 0);
    }
}

/*
 * Initialize the Socket object. Similar to InitApplicationSocket but with the difference
 * that now we are at network layer (SOCK_RAW type socket) so transport layer proto number
 * makes no sense (it has to be explicitly set to zero to avoid EINVAL error when calling sendto)
 */
void InitSonarSocket(_Utils::Options &options, std::shared_ptr<_NetLayer::BaseSock> &socket)
{
    auto logger=_NetTools::Singleton<_NetTools::Logger>::Instance();
    logger->Log("[InitSonarSocket]: initializing socket object");

    /* Distiguish between IPv4 and IPv6 because structures are incompatible */
    if (options.Six()) {
```

```
        logger->Log("[InitSonarSocket]: instantiating IPv6 socket object");
        auto tempSocket = std::make_shared<_NetLayer::ActiveSock<_NetLayer::IA6, _NetLayer::SAI6>>(IPv6);
        InitSonarSocket6(options, tempSocket);
        /* shared_ptr copy constructor in action */
        socket = tempSocket;
    } else {
        logger->Log("[InitSonarSocket]: instantiating IPv4 socket object");
        auto tempSocket = std::make_shared<_NetLayer::ActiveSock<_NetLayer::IA, _NetLayer::SAI>>(IPv4);
        InitSonarSocket4(options, tempSocket);
        /* shared_ptr copy constructor in action */
        socket = tempSocket;
    }

    socket->SetMode(options.Client()?ModeClient:ModeServer);
}

/*
 * Initialize the link in the chain of responsibility that corresponds with the Transport layer in TCP/IP
 * Possibilities are: UDP, TCP, SCTP (and ICMP). ICMP is in Network layer withing TCP/IP, but it was placed here since
it uses IP
 * In this example possibilities have been limited to UDP and TCP
 */
void InitControl(_Utils::Options &options, std::shared_ptr<_NetLayer::SockHandler> &control)
{
    auto logger=_NetTools::Singleton<_NetTools::Logger>::Instance();
    logger->Log("[InitControl]: initializing ICMP");

    if (options.Six()) {
        /* Type SOCK_RAW, Protocol IPPROTO_ICMP6 */
        logger->Log("[InitControl]: instantiating ICMPv6 control object");
        control = std::make_shared<_NetLayer::SockUdp>(CONTROL, CONTROL_ICMP6);
    } else {
        /* Type SOCK_RAw, Protocol IPPROTO_ICMP */
        logger->Log("[InitControl]: instantiating ICMPv4 control object");
        control = std::make_shared<_NetLayer::SockTcp>(CONTROL, CONTROL_ICMP);
    }
}

/*
 * Initialize the application layer object that makes use of Socket API (for the Sonar, Raw Sockets)
 * NetLayer can be used through inheritance, composition or aggregation
 * The library design does not force the use of any concrete pattern
 * In this example aggregation has been used
 */
void InitSonar(_Utils::Options &options, std::shared_ptr<_Application::Base> &sonar)
{
    auto logger=_NetTools::Singleton<_NetTools::Logger>::Instance();
    logger->Log("[InitSonar]: initializing sonar application layer");

    if (options.Client()) {
        sonar = std::make_shared<_Sonar::Sonar>(_Sonar::PING);
    } else {
        sonar = std::make_shared<_Sonar::Sonar>(_Sonar::STALK);
    }
}

std::shared_ptr<_NetTools::SmartThread> StartApplication(std::shared_ptr<_Application::Base> &application)
{
    auto logger=_NetTools::Singleton<_NetTools::Logger>::Instance();
    logger->Log("[StartApplication]");

    /* Lambdas simplify the task of creating the Thread object at the cost of making the code more obscure
       We have to fit to the POSIX thread interface, this is the WHY of this cast thingy
       typedef void*(*PF)(void*) where PF means pointer to a function */
    auto applicationThread=std::make_shared<_NetTools::SmartThread>([](void *_t)->void* {
        auto temp=static_cast<_Application::Base*>(_t);
        temp->Init();
        temp->Run();
        return nullptr;
    }, (void*)application.get());

    /* The usage of lambdas it's been a question of taste. IMHO I think they fit very well here
       but if you think they make the code look spaghetti-like, you have write a separate function
       and place the logic there */

    return applicationThread;
}
```

```
std::shared_ptr<_NetTools::SmartThread> StartSonar(std::shared_ptr<_Application::Base> &sonar)
{
    auto logger=_NetTools::Singleton<_NetTools::Logger>::Instance();
    logger->Log("[StartSonar]");

    /* Lambdas simplify the task of creating the Thread object at the cost of making the code more obscure
       We have to fit to the POSIX thread interface, this is the WHY of this cast thingy
       typedef void*(*PF)(void*) where PF means pointer to a function */
    auto sonarThread=std::make_shared<_NetTools::SmartThread>([](void *_t)->void* {
        auto temp=static_cast<_Application::Base*>(_t);
        temp->Init();
        temp->Run();
        return nullptr;
    }, (void*)sonar.get());

    /* The usage of lambdas it's been a question of taste. IMHO I think they fit very well here
       but if you think they make the code look spaghetti-like, you have write a separate function
       and place the logic there */

    return sonarThread;
}

void         Start(_Utils::Options&         options,         std::shared_ptr<_Application::Base>         &application,
std::shared_ptr<_Application::Base> &sonar)
{
    auto logger=_NetTools::Singleton<_NetTools::Logger>::Instance();
    logger->Log("[Start]");

    /* To ilustrate Threads classes implemented in NetTools library, there will be created two threads:
       one thread will run the TCP or UDP logic while another thread will run ICMP (raw socket) logic */

    /* Will store and manage both thread using a thread pool, which will simplify things
       since we'll have to create the corresponding threads and add them to the pool */
    _NetTools::ThreadPool threads;

    /* There are three kind of thread classes defined in NetTools library:
       Thread: POSIX and Windows threads compatible class with simple interface. Internally has a state machine.
       Thread class interface is simple: Start, Join (blocking), Halt (non-blocking) and Status (state machine).
       SmartThread class: it is smart because it switches state from RUNNING to HALTED when the thread has endend.
       This allows to asynchronously release thread resources. This is in the thread pool class.
       AutoThread class: it another iteration. This kind of thread release by itself resources once it is done.
       There is also a thread pool class to ease several threads simultaneously */

    /* Let's start TCP or UDP part in its own thread */
    threads.Add(StartApplication(application));

    if (options.Sonar()) {
        /* In the same way, ICMP logic will execute in its own thread */
        threads.Add(StartSonar(sonar));
    }

    /* Wait all threads conforming the pooling to complete their tasks
       This is a blocking operation because it waits untill all the threads in the pool have finished
       The non-blocking alternative is Cycle, which will check threads status and will only resources of
       those in status HALTED */
    threads.Join();

    /* In this the blocking Join option has been used. In NetRat application, where each protocol has its own
       thread, the non-blocking Cycle version was used instead. In that particular case the goal was different,
       because there might exist the case when we wish to restart a communication thread due to configuration change */

    logger->Log("[Start]: all thread reaped, exiting");
}
```

13. Referencias

[TCP/IP-1-94] W. Richard Stevens, “TCP/IP Illustrated”, Volumen 1, 1 Edición, 1994 Addison-Wesley Professional Computing Series

[TCP/IP-1-12] Kevin R. Fall, W. Richard Stevens, “TCP/IP Illustrated”, Volumen 1, 2 Edición, 2012 Addison-Wesley Professional Computing Series

[TCP/IP-2] Gary R. Wright, W. Richard Stevens, “TCP/IP Illustrated”, Volumen 2, 1 Edición, Addison-Wesley Professional Computing Series

[UNIX-1] W. Richard Stevens, B. Fenner, Andrew M. Rudoff, “UNIX Network Programming, The Sockets Networking API”, Volumen 1, 3 Edición, 2004 Addison-Wesley Professional Computing Series

[GOF] E. Gamma, R. Helm, R. Johnson, J. Vlissides, “Design Patterns, Elements of Reusable Object-Oriented Software”, 1995 Addison-Wesley Professional Computing Series. (*Gang Of Four*).

[TANNENBAUM] A. S. Tannenbaum, “Computers Network”, 4 edición, 2003 Pearson Education.

[CLARK] D. Clark, “Apocalypse”, M.I.T. disponible en http://groups.csail.mit.edu/ana/People/DDC/Apocalypse.html

[VNX] Virtual Networks over LinuX, disponible en: http://neweb.dit.upm.es/vnxwiki/index.php/Main_Page.

[GIT] “Sistema de control de versiones” disponible en http://git-scm.com/

[GITHUB] “Front-end sobre Git” disponible en www.github.com

[ERNESTO81] E. García Muñoz, “Proyectos publicados en GitHub”, 2014, disponible en https://github.com/ernesto81

[NETLAYER] E. García Muñoz, “Net Layer, Network Socket Layer Abstraction”, 2014, disponible en https://github.com/ernesto81/NetLayer

[NETTOOLS] E. García Muñoz, “NetTools, Network generic Tools”, 2014, disponible en https://github.com/ernesto81/NetTools

[NETCRAFT] E. García Muñoz, “NetCraft, Network IPv4, IPv6, ICMPv4 and ICMPv6 Crafter”, 2014, disponible en https://github.com/ernesto81/NetCraft

[NETRAT] E. García Muñoz, “NetRat, Network Remote Access Trojan”, 2014, disponible en https://github.com/ernesto81/NetRat

[PING] Mike Muuss, “Ping, the Internet Sonar”, Julio 1983, disponible en http://ftp.arl.army.mil/~mike/ping.html

[TRACEROUTE] Van Jacobson, “Traceroute”, disponible en http://linux.die.net/man/8/traceroute

[NETCAT] The Hobbit, “The Network Cat, swiss army knife of Internetworking”, 1.10, Marzo 1996, disponible en http://nc110.sourceforge.net/

[NETEXAMPLE] E. García Muñoz, “NetExample, Network simple Example”, 2014, disponible en: https://github.com/ernesto81/NetExample

[ICMPHOLE] A. Muller, N. Evans, C. Grothoff, S. Kamkar, “Autonomous NAT Traversal”, 2010 IEEE 10 Conferencia internacional sobre P2P, Agosto 2010

[CAMEL] L. Wall, T. Christiansen, J. Orwant, “Programming PERL”, 3 Edición, 2000 O’Reilly (*el libro del Camello*)

[CPP] B. Stroustrup, “The C++ Programming Language”, Edición Especial, 2000 Addison-Wesley

[IPTABLES] Gregor N. Purdy, “Linux iptables pocket reference, Firewall, NAT & Accounting”, 1 Edición, 2004 O’Reilly.

[MPTCP] C. Raiciu, C Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, M. Handley, “How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP”.

[JESSIE] “Debian Jessie Release Information” disponible en http://www.debian.org/releases/testing/

[WHEEZY] “Debian Wheezy Release Information” disponible en https://www.debian.org/releases/wheezy/

[FREEBSD] “Proyecto FreeBSD” disponible en http://www.freebsd.org/es/

[JUNOS] “Juniper Networks OS” disponible en http://www.juniper.net/es/es/products-services/software/junos-platform/junos-pulse/

[OLIVE] “JUNOS virtualized” disponible en http://juniper.cluepon.net/index.php/Olive

[GDB] “GDB: The GNU Project Debugger” disponible en http://www.gnu.org/software/gdb/

[VALGRIND] “Valgrind” disponible en http://valgrind.org/

[VIM] “VI improved” disponible en http://www.vim.org/

[WIRESHARK] “Wireshark” disponible en http://www.wireshark.org/

[TCPDUMP] “TCPDUMP & LIBPCAP” disponible en http://www.tcpdump.org/

[WIN8.1] “Microsoft Windows 8.1” disponible en http://windows.microsoft.com/es-es/windows-8/meet

[SERVER12] “Microsoft Windows Server 2012 R2” disponible en http://www.microsoft.com/es-es/server-cloud/products/windows-server-2012-r2/default.aspx#fbid=fuQR5JVIGg0

[VS] “Visual Studio” disponible en http://www.visualstudio.com/

[MANALYZER] “Microsoft Message Analyzer” disponible en http://www.microsoft.com/en-us/download/details.aspx?id=40308

[HYPERV] “Microsoft Hyper-V” disponible en http://www.microsoft.com/oem/en/products/servers/Pages/hyper_v_server.aspx#fbid=LxufnrQp-k6

[VBOX] “Oracle Virtual Box” disponible en https://www.virtualbox.org/

[RFC768] J. Postel, “User Datagram Protocol”, RFC 768, Agosto 1980, disponbile en http://www.ietf.org/rfc/rfc768.txt

[RFC791] "Internet Protocol, Darpa Internet Program Protocol Specification", RFC 791, Septiembre 1981, disponible en http://www.ietf.org/rfc/rfc791.txt

[RFC792] J. Postel, "Internet Control Message Protocol, Darpa Internet Program Protocol Specification", RFC 792, Septiembre 1981, disponible en http://www.ietf.org/rfc/rfc792.txt

[RFC793] "Transmission Control Procotol, Darpa Internet Program Protocol Specification", RFC 793, Septiembre 1981, disponible en http://www.ietf.org/rfc/rfc793.txt

[RFC1071] R. Braden, D. Borman, C. Partridge, "Computing the Internet Checksum", RFC 1071, Septiembre 1988, disponible en http://www.ietf.org/rfc/rfc1071.txt

[RFC1958] B. Carpenter, "Architectural Principles of the Internet", RFC 1958, Junio 1996, disponible en http://www.ietf.org/rfc/rfc1958.txt

[RFC2460] S. Deering, R. Hinden, "Internet Protocol Version 6 (IPv6) Specification", RFC 2460, Diciembre 1998, disponible en http://www.ietf.org/rfc/rfc2460.txt

[RFC2461] T. Narten, E. Nordmark, W. Simpson, "Neighbor Discovery for IP Version 6 (IPv6)", RFC 2461, Diciembre 1998, disponible en http://www.ietf.org/rfc/rfc2461.txt

[RFC4007] S. Deering, B Haberman, T. Jinmei, E. Nordmark, B. Zill, "IPv6 Scope Address Architecture", RFC 4007, disponible en http://www.ietf.org/rfc/rfc4007.txt

[RFC4038] M-K. Shin, Y-G Hong, J. Hagino, P. Savola, E. M. Castro, "Application Aspects of IPv6 Transition", RFC 4038, Marzo 2005, disponible en http://www.ietf.org/rfc/rfc4038.txt

[RFC4213] E. Nordmark, R. Gilligan, "Basic Transition Mechanisms for IPv6 Hosts and Routers", RFC 4213, Marzo 2005, disponible en http://www.ietf.org/rfc/rfc4213.txt

[RFC4291] R. Hinden, S. Deering, "Version 6 Addressing Architecture", RFC 4291, Febrero 2006, disponible en http://www.ietf.org/rfc/rfc4291.txt

[RFC4443] A. Conta, S. Deering, M. Gupta, "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specificacion", RFC 4443, Marzo 2006, disponible en http://www.ietf.org/rfc/rfc4443.txt

[RFC4787] F. Audet, C. Jennings, "Network Address Translation (NAT) Behavioral Requirements for Unicast UDP", RFC 4787, Enero 2007, disponible en http://www.ietf.org/rfc/rfc4787.txt

[RFC4861] T. Narten, E. Nordmark, W. Simpson, H. Soliman, "Neighbor Discovery for IP version 6 (IPv6)", RFC 4861, Septiembre 2007, disponible en http://www.ietf.org/rfc/rfc4861.txt

[RFC4890] E. Davies, J. Mohacsi, "Recommendations for Filtering ICMPv6 Messages in Firewalls", RFC 4890, Mayo 2007, disponible en http://www.ietf.org/rfc/rfc4890.txt

[RFC4960] Ed R. Stewart, "Stream Control Transmission Protocol", RFC 4960, Septiembre 2007, disponible en http://www.ietf.org/rfc/rfc4960.txt

[RFC5095] J. Abley7, P. Savola, G. Neville-Neil, “Deprecation of Type 0 Routing Headers in IPv6”, RFC 5095, Diciembre 2007, disponible en http://www.ietf.org/rfc/rfc5095.txt

[RFC5245] J. Rosenberg, “Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator”, RFC 5245, disponible en http://www.ietf.org/rfc/rfc5245.txt

[RFC5389] J. Rosenberg, R. Mahy, P. Matthews, D. Wing, “Session Travesal Utilities for NAT (STUN)”, RFC 5389, Octubre 2008, disponible en http://www.ietf.org/rfc/rfc5389.txt

[RFC5722] S. Krishnan, “Handling of Overlapping IPv6 Fragments”, RFC 5722, Diciembre 2009, disponible en http://www.ietf.org/rfc/rfc5722.txt

[RFC5871] J. Arkko, S. Bradner, “IANA Allocation Guidelines for the IPv6 Routing Header”, RFC 5871, Mayo 2010, disponible en http://www.ietf.org/rfc/rfc5871.txt

[RFC5952] S. Kawamura, M. Kawashima, “A Recommendation for IPv6 Address Text Representation”, RFC 5952, Agosto 2010, disponible en http://www.ietf.org/rfc/rfc5952.txt

[RFC6052] C. Bao, C. Huitema, M. Bagnulo, M. Boucadair, X. Li, “IPv6 Addressing of IPv4/IPv6 Translators”, RFC 6052, Octubre 2010, http://www.ietf.org/rfc/rfc6052.txt

[RFC6145] X. Li, C. Bao, F. Baker, “IP/ICMP Translation Algorithm”, RFC 6145, Abril 2011, disponible en http://www.ietf.org/rfc/rfc6145.txt

[RFC6146] M. Bagnulo, P. Matthews, I. van Beijnum, “Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers”, RFC 6146, Abril 2011, disponible en http://www.ietf.org/rfc/rfc6146.txt

[RFC6147] M Bagnulo, A. Sullivan, P. Matthews, I. van Beijnum, “DNS64: DNS Extensions for Network Address Translation from IPv6 Clients to IPv4 Servers”, RFC 6147, Abril 2011, disponible en http://www.ietf.org/rfc/rfc6147.txt

[RFC6177] T. Narten, G. Huston, L. Roberts, “IPv6 Address Assignment to End Sites”, RFC 6177, Marzo 2011, disponible en http://www.ietf.org/rfc/rfc6177.txt

[RFC6182] A. Ford, C. Raiciu, M Handley, S. Barre, J. Iyengar, “Architectural Guidelines for Multipath TCP Developement”, RFC 6182, Marzo 2011, disponible en http://www.ietf.org/rfc/rfc6182.txt

[RFC6314] C. Boulton, J. Rosenberg, G. Camarillo, F. Audet, “NAT Traversal Practices for Client-Server SIP”, RFC 6314, Julio 2011, disponible en http://www.ietf.org/rfc/rfc5314.txt

[RFC6356] C. Raiciu, M. Handly, D. Wischik, “Coupled Congestion Control for Multipath Transport Protocols”, RFC 6356, Octubre 2011, disponible en http://www.ietf.org/rfc/rfc6256.txt

[RFC6437] S. Amate, B. Carpenter, S. Jiang, J. Rajahalme, “IPv6 Flow Label Specification”, RFC 6437, Noviembre 2011, disponible en http://www.ietf.org/rfc/rfc6437.txt

[RFC6564] S. Krishnan, J. Woodyatt, E. Kline, J. Hoagland, M. Bhatia, “A Uniform Format for IPv6 Extension Headers”, RFC 6564, Abril 2012, disponible en http://www.ietf.org/rfc/rfc6564.txt

[RFC6824] A. Ford, C. Raiciu, M. Handley, O. Bonaventure, “TCP Extensions for Multipath Operation with Multiple Addresses”, RFC 6824, Enero 2013, disponible en http://www.ietf.org/rfc/rfc6824.txt

[RFC6877] M. Mawatari, M. Kawashima, C. Byrne, “464XLAT: Combination of Stateful and Stateless Translation”, RFC 6877, Abril 2013, disponible en http://www.ietf.org/rfc/rfc6877.txt

[RFC6935] M. Eubanks, P. Chimento, M. Westerlund, “IPv6 and UDP Checksums for Tunneled Packets”, RFC 6935, Abril 2013, disponible en http://www.ietf.org/rfc/rfc6935.txt

[RFC6946] F. Gont, “Processing of IPv6 "Atomic Fragments”, RFC 6946, Mayo 2013, disponible en http://www.ietf.org/rfc/rfc6946.txt

[DRAFTSCTP] P. Amer, M Becke, T. Dreibholz, N. Ekiz, J. Iyengar, P. Natarajan, R. Stewart, M. Tuexen, “Load Sharing for the Stream Control Transmission Protocol (SCTP)”, Marzo 2013, disponible en http://tools.ietf.org/html/draft-tuexen-tsvwg-sctp-multipath-06