# The ISP Column

*A column on things Internet*

|  |
| --- |
| Geoff Huston |

October 2017

# Raw Sockets in IPv6

Operating Systems contain more than a task scheduler, memory manager and drivers for peripheral devices. Among many other functions, they also contain an interface to a local network protocol engine. This means that applications that run within the operating system’s environment do not need to re-implement a network protocol, but can make use of a shared common interface to the underlying network via a simple interface. In Unix, the commonly used interface to the underlying communications system is via the *socket* interface. This interface creates an abstract model of the underlying network by mimicking a simple peripheral device, and once the application connects to a network socket it is can use the network through invocation of *read* and *write* commands on that socket. In this manner, all of the details about managing the TCP or UDP packet headers and the common IP layer is managed by the operating system’s protocol drivers, and is largely hidden from the application. Obviously, this makes it simple for applications to access the Internet.

As part of a measurement experiment we wanted an implementation of an IPv6 UDP server and a TCP server that generated fragmented IPv6 packets where the application controls the packet fragmentation. The conventional socket interface masks any visibility to the underlying packet transactions, and therefore cannot be used in the conventional manner. There is a specialised socket option that allows an application to interact directly with the underlying communications driver and read and write IP datagrams without having the packets processed by the operating system’s IP protocol drivers. This is the *Raw Socket* option in IPv4, and an example of opening such a socket is shown in this code snippet:

/\*

 \* open\_raw\_socket

 \*

 \* open a raw socket interface into the kernel

 \*/

void

open\_raw\_socket()

{

 const int on = 1 ;

 /\* create the raw socket via the socket call\*/

 if ((sock\_fd = socket(AF\_INET, SOCK\_RAW, IPPROTO\_TCP)) < 0) {

 perror("socket() error");

 exit(EXIT\_FAILURE);

 }

 /\* inform the kernel the IP header is already attached via a socket option \*/

 if (setsockopt(sock\_fd, IPPROTO\_IP, IP\_HDRINCL, &on, sizeof(on)) < 0) {

 perror("setsockopt() error");

 exit(EXIT\_FAILURE);

 }

}

When writing code that supports IPv6 then in most aspects of the socket interface to the network stack it's a case of changing the protocol specifier from AF\_INET to AF\_INET6, expanding the IP address data structures to 128 bites and aside from explicit inclusion of the IPv6-specific network include files, there is not much else to do. However, in the case of raw sockets, Ipv6 is indeed very different. As noted in RFC3542:

 [The] difference from IPv4 raw sockets is that complete packets

 (that is, IPv6 packets with extension headers) cannot be sent or

 received using the IPv6 raw sockets API.

 …

 When writing to a raw socket the kernel will automatically fragment

 the packet if its size exceeds the path MTU, inserting the required

 fragment headers.

 …

 Most IPv4 implementations give special treatment to a raw socket

 created with a third argument to socket() of IPPROTO\_RAW, whose value

 is normally 255, to have it mean that the application will send down

 complete packets including the IPv4 header. (Note: This feature was

 added to IPv4 in 1988 by Van Jacobson to support traceroute, allowing

 a complete IP header to be passed by the application, before the

 IP\_HDRINCL socket option was added.) We note that IPPROTO\_RAW has no

 special meaning to an IPv6 raw socket (and the IANA currently

 reserves the value of 255 when used as a next-header field).

RFC 3542, “Advanced Sockets Application Program Interface (API) for IPv6”, W. Stevens et al, May 2003.

What we want to do, namely use an IPv6 raw socket that allows the application to have direct control of the entirety of the IPv6 packet header, including fragmentation handling, is not supported using the IPv6 raw socket interface in the contest of the code fragment above.

But there is still one more approach that can be used. Sockets can extend down one further level in the protocol stack, and connect directly to the network interface rather than to a network protocol engine. This form of raw socket requires the application to write complete packets to the socket interface, including the media layer framing (such as the Ethernet headers), the IP level packet header, the transport protocol header, as well as any payload. If we want to perform explicit control over the generation of IPv6 Extension Headers to support IPv6 packet fragmentation control at the application level, then this looks like a viable approach.

In this article, I will describe how we used this raw socket interface in IPv6 to generate a UDP-based DNS server and a TCP-based HTTP(S) server that allowed the server to exercise direct control over packet fragmentation.

### A “Raw Socket” UDP DNS Server

The aim of this server was to provide a font end to a conventional DNS server. Incoming IPv6 UDP DNS queries are passed to a conventional “back end” DNS resolver. UDP responses from the “back end” DNS resolver are fragmented into at least two IPv6 packets, ensuring that no packet is larger than 512 octets, and passed back to the original sender.



Let’s now look at the details of the code for this packet handler. The first part of the packet processor is a conventional UDP server listening port that listens on IPv6 to incoming packets addressed to the local port 53.

 /\* Create a IPv6 datagram socket and associate it with the variable sockfd\*/

 sockfd = socket(AF\_INET6,SOCK\_DGRAM,17);

 if (sockfd < 0) {

 perror("Socket") ;

 exit(EXIT\_FAILURE) ;

 }

 /\* The local IPv6 address used to listen is stored in the variable host

 This is converted to an internal representation of the IPv6 address \*/

 if (((status = inet\_pton(AF\_INET6,host,&listen.sin6\_addr))) <= 0) {

 if (!status)

 fprintf(stderr, "Not in presentation format");

 else

 perror("inet\_pton");

 exit(EXIT\_FAILURE);

 }

 listen.sin6\_family = AF\_INET6 ;

 listen.sin6\_port = htons(53);

 /\* now bind the socket to the local IPv6 address and port 53 \*/

 if (bind(sockfd,(const struct sockaddr \*) &listen,(sizeof listen))) {

 perror("bind") ;

 exit(EXIT\_FAILURE) ;

 }

 /\* we can now set up the listener – the source address and source port of

 the incoming UDP packet is stored in the cliaddr struct \*/

 addrlen= sizeof \*cliaddr ;

 cliaddr=malloc(addrlen);

 len=addrlen;

 for ( ;; ) { /\* do forever \*/

 if ((rc = recvfrom(sockfd, buf, MAXBUF, 0, (struct sockaddr \*) cliaddr, &len)) < 0 ) {

 printf("server error: errno %d\n",errno);

 perror("reading datagram");

 exit(1);

 }

 /\* at this stage the client’s address and port is stored in cliaddr, and the DNS query

 is stored in buf, with length rc – we can pass this DNS query to the back end \*/

 }

The next part of the code is also quite conventional. It takes the original DNS query, which is the payload returned by the recvfrom() call, and passes it to a DNS resolver without alteration. This example code is synchronous for simplicity, so the routine to query the back end DNS resolver will wait for the DNS resolver’s response before returning. The code also uses IPv4 for the connection to the DNS server.

 /\* open a UDP socket to the DNS server at address serveraddr4 \*/

 if ((dns\_sockfd = socket(AF\_INET, SOCK\_DGRAM, 0)) < 0) {

 perror("UDP slave socket() error") ;

 return 1;

 }

 if (connect(dns\_sockfd, (struct sockaddr \*) &serveraddr4, sizeof(serveraddr4)) < 0) {

 perror("UDP V4 connect error") ;

 return 1;

 }

 /\* send the DNS query to the server \*/

 i = write(sockfd, query, data\_size) ;

 if (i < 0) {

 perror("ERROR writing to socket");

 return 1;

 }

 /\* collect the server's reply \*/

 dns\_response.len = recvfrom(dns\_sockfd, dns\_response.buf, MAXBUF, 0,

 (struct sockaddr \*) &serveraddr4, &serverlen);

 if (dns\_response.len < 0) {

 perror("UDP recvfrom");

 return 1;

 }

 close(dns\_sockfd) ;

This is the simplest approach, but obviously not the most efficient. The code could be made asynchronous by using a raw socket interface that generates queries to the back end DNS resolver and immediately returns. A second process would bind itself to listen for responses from the back end DNS resolver and associate this response with the original client address and client source port, and each time a packet is received on this channel it would be passed to the responder to respond to the original client.

This brings us to the last part of the server, which is the processing of IPv6 outbound packets directed back to the original client, using a raw ethernet socket interface.

The first part of this code opens a raw socket to in the interface named by the variable interface. In order to construct Ethernet packets, we need to get the Ethernet MAC address of the interface, which is performed by an ioctl call on a raw socket interface. We also use the if\_nametoindex() call to get the local index value of the named interface. We can then open a raw Ethernet socket on the named interface.

 /\* Get a socket descriptor to look up interface \*/

 if ((sd = socket (PF\_PACKET, SOCK\_RAW, htons (ETH\_P\_ALL))) < 0) {

 perror("socket() failed to get socket descriptor for using ioctl() ");

 exit(EXIT\_FAILURE);

 }

 /\* Use an ioctl() to look up interface name and get its MAC address \*/

 /\* clear the ifr variable \*/

 memset(&ifr, 0, sizeof (ifr));

 /\* write in the name of the interface: held in variable interface \*/

 snprintf(ifr.ifr\_name, sizeof (ifr.ifr\_name), "%s", interface);

 if (ioctl(sd, SIOCGIFHWADDR, &ifr) < 0) {

 perror("ioctl() failed to get source MAC address ");

 exit (EXIT\_FAILURE);

 }

 /\* done! \*/

 close(sd);

 /\* Copy source MAC address into src\_mac \*/

 memcpy(src\_mac, ifr.ifr\_hwaddr.sa\_data, 6 \* sizeof (uint8\_t));

 /\* Find interface index from interface name using the call if\_nametoindex()

 and store the index value in struct sockaddr\_ll device (which will be

 used as an argument of sendto() of the subsequent output call \*/

 memset(&device, 0, sizeof (device));

 if ((device.sll\_ifindex = if\_nametoindex(interface)) == 0) {

 perror ("if\_nametoindex() failed to obtain interface index ");

 exit (EXIT\_FAILURE);

 }

 /\* now copy over the mac address \*/

 device.sll\_family = AF\_PACKET;

 memcpy(device.sll\_addr, src\_mac, 6 \* sizeof (uint8\_t));

 device.sll\_halen = 6;

 /\* now we are ready to submit a request for a raw socket descriptor \*/

 if ((sd = socket (PF\_PACKET, SOCK\_RAW, htons (ETH\_P\_ALL))) < 0) {

 perror ("socket() failed ");

 exit (EXIT\_FAILURE);

 }

There is still one critical piece of information that is missing here. When we generate the Ethernet packet, we need to add the destination MAC address of the local IPv6 gateway. One way is to use the ifconfig cli command and pull this information manually. However, we can also try to perform this automatically by listening for periodic IPv6 router advertisements on the interface. The MAC source address of these router advertisements packets is the destination address we are after. Here’s a code snippet that binds to the interface and looks for RA messages:

 /\* Request a socket descriptor sd \*/

 if ((sd = socket (AF\_INET6, SOCK\_RAW, IPPROTO\_ICMPV6)) < 0) {

 perror ("Failed to get socket descriptor ");

 exit (EXIT\_FAILURE);

 }

 /\* Set flag so we receive destination address from recvmsg \*/

 on = 1;

 if ((status = setsockopt (sd, IPPROTO\_IPV6, IPV6\_RECVPKTINFO, &on, sizeof (on))) < 0) {

 perror ("setsockopt to IPV6\_RECVPKTINFO failed ");

 exit (EXIT\_FAILURE);

 }

 /\* Obtain MAC address of this interface \*/

 memset (&ifr, 0, sizeof (ifr));

 snprintf (ifr.ifr\_name, sizeof (ifr.ifr\_name), "%s", interface);

 if (ioctl (sd, SIOCGIFHWADDR, &ifr) < 0) {

 perror ("ioctl() failed to get source MAC address ");

 exit (EXIT\_FAILURE);

 }

 /\* Retrieve interface index of this node \*/

 if ((ifindex = if\_nametoindex (interface)) == 0) {

 perror ("if\_nametoindex() failed to obtain interface index ");

 exit (EXIT\_FAILURE);

 }

 /\* Bind a device socket to this interface \*/

 if (setsockopt (sd, SOL\_SOCKET, SO\_BINDTODEVICE, (void \*) &ifr, sizeof (ifr)) < 0) {

 perror ("SO\_BINDTODEVICE failed");

 exit (EXIT\_FAILURE);

 }

 /\* Listen for incoming message from socket sd

 Keep at it until we get a router advertisement \*/

 ra = (struct nd\_router\_advert \*) inpack;

 while (ra->nd\_ra\_hdr.icmp6\_type != ND\_ROUTER\_ADVERT) {

 if ((len = recvmsg (sd, &msghdr, 0)) < 0) {

 perror ("recvmsg failed ");

 exit (EXIT\_FAILURE);

 }

 }

 pkt = (uint8\_t \*) inpack;

 for (i=2; i<=7; i++) {

 dst\_mac[i-2] = pkt[sizeof (struct nd\_router\_advert) + i];

 }

 close (sd);

We now have collected enough information to generate an IPv6 UDP packet and send it. We will fragment the packet in every case so the fragmentation header is always present:

 /\* IPv6 header \*/

 iphdr = (struct ip6\_hdr \*) &out\_packet\_buffer[0] ;

 /\* IPv6 version (4 bits), Traffic class (8 bits), Flow label (20 bits) \*/

 iphdr->ip6\_flow = htonl ((6 << 28) | (0 << 20) | 0);

 /\* Next header (8 bits): 44 for Frag \*/

 iphdr->ip6\_nxt = 44;

 /\* Hop limit (8 bits): default to maximum value \*/

 iphdr->ip6\_hops = 255;

 /\* src address \*/

 bcopy(&srcaddr->sin6\_addr,&(iphdr->ip6\_src), 16) ;

 /\* dst address \*/

 bcopy(&cliaddr->sin6\_addr,&(iphdr->ip6\_dst), 16);

 /\* set up the UDP packet \*/

 uhdr = (struct udphdr \*) &(payload[0]);

 uhdr->uh\_sport = htons(port) ;

 uhdr->uh\_dport = cliaddr->sin6\_port;

 uhdr->uh\_ulen = htons(dns\_response->len + 8);

 uhdr->uh\_sum = 0;

 /\* copy payload bytes from the dns response buffer to the payload buffer \*/

 bcopy(dns\_response->buf,&payload[8],dns\_response->len) ;

 /\* calculate the UDP checksum \*/

 uhdr->uh\_sum = udp\_checksum(uhdr,dns\_response->len + 8, &srcaddr->sin6\_addr, &cliaddr->sin6\_addr);

 /\* now fragment the output \*/

 /\* set up the frag header \*/

 fhdr = (struct ip6\_frag \*) &out\_packet\_buffer[40];

 fhdr->ip6f\_nxt = 17 ;

 fhdr->ip6f\_reserved = 0 ;

 fhdr->ip6f\_offlg = htons(1);

 fhdr->ip6f\_ident = rand() % 4294967296 ;

 /\* the total size to send is the payload size plus the UDP header \*/

 to\_send = dns\_response->len + 8 ;

 to\_buf = (char \*) uhdr ;

 /\* now carve up the UDP response into frags \*/

 /\* initial block of UDP payload size is at least 8 bytes less than the original dns response \*/

 units = dns\_response->len / 8 ;

 if (units > 16) datalen = 128 ;

 else datalen = (units - 1) \* 8 ;

 frag\_offset = 0 ;

 /\* add in the size of the UDP header into datalen in the first instance \*/

 datalen += 8 ;

 /\* Destination and Source MAC addresses \*/

 memcpy(ether\_frame, dst\_mac, 6 \* sizeof (uint8\_t));

 memcpy(ether\_frame + 6, src\_mac, 6 \* sizeof (uint8\_t));

 /\* Next is ethernet type code (ETH\_P\_IPV6 for IPv6) \*/

 ether\_frame[12] = ETH\_P\_IPV6 / 256;

 ether\_frame[13] = ETH\_P\_IPV6 % 256;

 /\* now send the payload in fragments \*/

 while (to\_send > 0) {

 /\* each time we send datalen bytes plus the 8 byte frag header \*/

 iphdr->ip6\_plen = htons(datalen + 8);

 /\* now assemble the ether frame using 2 x 6 octet MAC addresses, a 2 octet

 Ethernet frame type field, a 40 octet IPv6 header, a 8 octet Extension

 Header and the payload \*/

 frame\_length = 6 + 6 + 2 + 40 + 8 + datalen;

 /\* IPv6 header + frag header \*/

 memcpy (ether\_frame + ETH\_HDRLEN, iphdr, 48);

 /\* payload fragment \*/

 memcpy (ether\_frame + ETH\_HDRLEN + 48, to\_buf, datalen);

 /\* Send ethernet frame out using the raw socket \*/

 if ((bytes = sendto (sd, ether\_frame, frame\_length, 0,

 (struct sockaddr \*) &device, sizeof (device))) <= 0) {

 perror ("sendto() failed");

 exit (EXIT\_FAILURE);

 }

 to\_send -= datalen ;

 to\_buf += datalen ;

 if (to\_send > 0) {

 if (to\_send <= 512) {

 /\* last frag \*/

 frag\_offset += (datalen / 8) ;

 fhdr->ip6f\_offlg = htons(frag\_offset << 3) ;

 datalen = to\_send ;

 }

 else {

 frag\_offset += (datalen / 8) ;

 fhdr->ip6f\_offlg = htons((frag\_offset << 3) + 1);

 datalen = 512 ;

 }

 }

 }

### A Raw Socket TCP HTTP(S) Server

The second part of our experiment was to test the extent to which fragmented IPv6 packets were successfully passed to end users. Within the constraints of our experimental setup this implied that we need to place a packet fragmentation module into a TCP session. The adopted approach is similar to that used for UDP, namely using a ‘normal’ HTTP(S) server as a backend server, and implementing a packet handler to perform the fragmentation of outbound packets directed to the end client.

There are a number of possible implementation options. The chosen option was to use what was in effect a session-less packet handler along the lines of an IPv6 NAT. Incoming SYN packets from the client created a new NAT binding state and the packet headers were re-written and passed to the back end. Other incoming packets from the client were matched against an established NAT binding state, the packet headers were re-written and passed to the back end. Packets from the back end needed to be matched against an established NAT state, and the packet headers were translated and large payload responses were fragmented as they were passed to the client.

 The first task is to receive TCP packets from the client and get them passed to the application, bypassing the operating system’s conventional behaviour. Most Unix systems will recognise an incoming TCP packet and if there is no listening TCP port associated with the destination port number, the system will respond with a TCP RST packet. This needs to be stopped.

In FreeBSD systems there is a system parameter than can be set to prevent this automatic blackhole behaviour:

 sysctl -w net.tcp.blackhole=2.

Debian systems don’t appear to have this form of control over these kernel-generated TCP reset responses, so the alternative is to suppress the TCP reset packet before it leaves the system using an iptables filter entry:

 ip6tables -I OUTPUT 1 -o eth0 -p tcp --tcp-flags ALL RST -j DROP

We then need to pick up all incoming IPv6 TCP packets addressed to port 80 and port 443 and process them within the context of this application. The simplest way to achieve this is by using the packet capture library routines (libpcap).

 /\* the PCAP capture filter - http and https v6 traffic only\*/

 sprintf(filter\_exp,"dst host %s and (port 80 or port 443) and tcp and ip6",host);

 /\* open capture device \*/

 if ((handle = pcap\_open\_live(interface, SNAP\_LEN, 1, 1, errbuff)) == NULL) {

 fprintf(stderr, "Couldn't open device %s: %s\n", interface, errbuff);

 exit(EXIT\_FAILURE) ;

 }

 /\* compile the filter expression \*/

 if (pcap\_compile(handle, &fp, filter\_exp, 0, 0) == -1) {

 fprintf(stderr, "Couldn't parse filter %s: %s\n", filter\_exp, pcap\_geterr(handle));

 exit(EXIT\_FAILURE) ;

 }

 /\* install the filter \*/

 if (pcap\_setfilter(handle, &fp) == -1) {

 fprintf(stderr, "Couldn't install filter %s: %s\n", filter\_exp, pcap\_geterr(handle));

 exit(EXIT\_FAILURE) ;

 }

 /\* set up the packet capture in an infinite loop \*/

 if (debug) printf("Enter PCAP packet capture loop\n") ;

 pcap\_loop(handle, -1, got\_packet, NULL) ;

Each incoming packet will invoke the packet handler got\_packet(), which can be used to process both packets coming in from the client and packets coming in from the HTTP(S) back end.

However, before looking at this code there is one more aspect of many modern host systems that we need to disable. These days many Ethernet interfaces are “smart” and part of that additional functionality is to perform TCP segmentation offloading into the interface card. The argument in favour of this is one of relieving the kernel of extraneous I/O interrupts, allowing the kernel to pass large buffers of memory to the interface card and have the card perform TCP segmentation for outgoing packets. Similarly, the card may aggregate several arriving packets into a single pseudo-TCP segment that is passed to the operating system in a single interrupt transaction. Obviously, we need to disable this functionality in this context:

 ethtool –K eth0 generic-segmentation-offload off

 ethtool –K eth0 generic-receive-offload: off

We can now examine the major aspects of the packet handler within the got\_packet() routine. The first is the handling of packets that come from the end client. The source address and source port are used to lookup the local NAT translation table, and this is used to create an outbound packet destined to the back-end HTTP(S) server.

 /\* this is a packet from the client to the packet handler \*/

 if ((dport == 80) || (dport === 443)) {

 /\* search of a matching entry in the local NAT table using the

 source address and source port as the lookup key \*/

 bcopy(&(ip->ip6\_src), &(bdg.ip6\_src),16);

 bdg.sport = sport ;

 bdp = find\_binding(&bdg,dport) ;

 /\* if we cannot find a NAT table entry, and the packet contains a TCP

 SYN flag, then we can create a new NAT entry \*/

 if ((!bdp) && (tcp->th\_flags & TH\_SYN)) {

 if (dport == 80) {

 port = next\_port\_80(&(ip->ip6\_src),ntohs(tcp->th\_sport)) ;

 bdp = port\_80\_ptr[port]->entry;

 }

 else {

 port = next\_port\_443(&(ip->ip6\_src),ntohs(tcp->th\_sport)) ;

 bdp = port\_443\_ptr[port]->entry;

 }

 }

 else if (!bdp)

 return;

 /\* retain the current TCP session sequence number in the NAT table \*/

 bdp->seq = tcp->th\_seq ;

 /\* perform a header translation and pass the packet to the back-end

 http(s) server \*.

 send\_packet\_to\_http(packet,bdp) ;

 }

The packets received from the back end have the ‘reverse’ NAT header substitution applied in both the IPv6 and TCP packet headers. The TCP checksum on the TCP pseudo-header is computed and the response is ready for the final phase of sending.

 /\* IPv6 header \*/

 iphdr = (struct ip6\_hdr \*) &out\_packet\_buffer[0] ;

 o\_iphdr = (struct ip6\_hdr \*) (packet + ETH\_HDRLEN) ;

 /\* IPv6 version (4 bits), Traffic class (8 bits), Flow label (20 bits) \*/

 iphdr->ip6\_flow = o\_iphdr->ip6\_flow ;

 /\* payload length \*/

 len = ntohs(o\_iphdr->ip6\_plen);

 /\* Hop limit (8 bits): default to maximum value \*/

 iphdr->ip6\_hops = 255;

 /\* src address \*/

 bcopy(&local6\_addr,&(iphdr->ip6\_src), 16) ;

 /\* dst address \*/

 bcopy(&tp->ip6\_src,&(iphdr->ip6\_dst), 16);

 /\* TCP header \*/

 orig\_tcp = (struct tcphdr \*) (packet + ETH\_HDRLEN + 40);

 tcp = (struct tcphdr \*) &(out\_packet\_buffer[40]);

 tcp->th\_dport = htons(tp->sport) ;

 tcp->th\_sport = orig\_tcp->th\_sport;

 /\* copy payload bytes from the original packet to the payload buffer \*/

 memcpy(&out\_packet\_buffer[44],&packet[ETH\_HDRLEN + 44],len - 4) ;

 /\* Destination and Source MAC addresses \*/

 memcpy(ether\_frame, dst\_mac, 6 \* sizeof (uint8\_t));

 memcpy(ether\_frame + 6, src\_mac, 6 \* sizeof (uint8\_t));

 /\* Next is ethernet type code (ETH\_P\_IPV6 for IPv6) \*/

 ether\_frame[12] = ETH\_P\_IPV6 / 256;

 ether\_frame[13] = ETH\_P\_IPV6 % 256;

 /\* Copy the IPv6 header into the ether\_frame \*/

 memcpy(ether\_frame + ETH\_HDRLEN, &out\_packet\_buffer[0], 40);

 payload = ntohs(iphdr->ip6\_plen) - (tcp->th\_off \* 4) ;

 pptr = &out\_packet\_buffer[40 + (tcp->th\_off \* 4)];

 tcp\_hdr\_len = tcp->th\_off \* 4 ;

 tcp\_sequence = ntohl(tcp->th\_seq) ;

If the payload is small we’ll send it without additional fragmentation.

 if (payload <= 16) {

 /\* copy across the TCP header \*/

 memcpy(ether\_frame + ETH\_HDRLEN + 40, &out\_packet\_buffer[40], tcp\_hdr\_len);

 /\* copy across the payload \*/

 memcpy(ether\_frame + ETH\_HDRLEN + 40 + tcp\_hdr\_len, pptr, payload);

 /\* IPv6 payload length \*/

 e\_iphdr = (struct ip6\_hdr \*) (ether\_frame + ETH\_HDRLEN) ;

 e\_iphdr->ip6\_plen = htons(tcp\_hdr\_len + payload);

 /\* Next header (8 bits): 6 for TCP \*/

 e\_iphdr->ip6\_nxt = IPPROTO\_TCP;

 /\* put in the adjusted sequence number and the new checksum \*/

 e\_tcp = (struct tcphdr \*) (ether\_frame + ETH\_HDRLEN + 40) ;

 e\_tcp->th\_seq = htonl(tcp\_sequence) ;

 e\_tcp->th\_sum = 0 ;

 e\_tcp->th\_sum = tcp\_checksum(e\_tcp,tcp\_hdr\_len+payload,tcp\_hdr\_len+payload,

 &(e\_iphdr->ip6\_src),&(e\_iphdr->ip6\_dst)) ;

 /\* ethernet frame length \*/

 frame\_length = ETH\_HDRLEN + 40 + tcp\_hdr\_len + payload ;

 if ((bytes = sendto (sd, ether\_frame, frame\_length, 0,

 (struct sockaddr \*) &device, sizeof (device))) <= 0) {

 perror ("sendto() failed");

 exit (EXIT\_FAILURE);

 }

 return;

 }

http://www.pdbuchan.com/rawsock/rawsock.html


### Author

*Geoff Huston* B.Sc., M.Sc., is the Chief Scientist at APNIC, the Regional Internet Registry serving the Asia Pacific region.

[*www.potaroo.net*](http://www.potaroo.net)


### Disclaimer

The above views do not necessarily represent the views or positions of the Asia Pacific Network Information Centre.