The ISP Column
A monthly column on things Internet

May 2010

Geoff Huston

Two Simple Hints for Dual Stack Servers

It seems that the imminent prospect of IPv4 address exhaustion has managed to generate a renewed interest in IPv6. A number of the conversations I have had lately have been about setting up dual stack servers, and there is a widespread concern that if you convert a server from single stack IPv4 to dual stack then some clients will have problems in accessing your site. The same concern has been voiced with converting a mail server from single stack to dual stack.

Here are two very simple hints may be of assistance to you:

1. Drop the interface MTU of the server. 1280 is a prudent value.

2. Run a local 6to4 interface on the server, and use it to route 2002::/16 for outbound packets.

That's all. If you are happy to go off and do just that then you've read as much as you need to. If you want to understand why these two additional adjustments are helpful, then read on.

Setting the Interface MTU

If you look on most servers you will see an interface Maximum Transmission Unit (MTU) size of 1500 octets. This particular number is derived from the old Ethernet specification and has remained a persistent feature of the networking environment for many years. The more general observation is that hosts set their MTU to be the MTU size of the local subnet to which they are attaching. In other words, the local host's MTU is set to the maximal size of the interface.

There is a lower bound to this number. In IPv4 it was set to 68 octets. The IPv6 specification says that all intermediate systems in a IPv6 network should be able to pass a packet of a total size that is no smaller than 1280 octets.

> In IPv6 the minimum MTU is 1280 bytes, and every IPv6 destination must be able to reassemble a fragmented IPv6 datagram of up to 1500 bytes in length.
>
> RFC2460:
> IPv6 requires that every link in the internet have an MTU of 1280 octets or greater. On any link that cannot convey a 1280-octet packet in one piece, link-specific fragmentation and reassembly must be provided at a layer below IPv6.
>
> …
>
> A node must be able to accept a fragmented packet that, after reassembly, is as large as 1500 octets. A node is permitted to

> accept fragmented packets that reassemble to more than 1500 octets. An upper-layer protocol or application that depends on IPv6 fragmentation to send packets larger than the MTU of a path should not send packets larger than 1500 octets unless it has assurance that the destination is capable of reassembling packets of that larger size.

So this interface MTU setting of 1500 octets should be just fine, shouldn't it?

Unfortunately that's not the case. These days it appears that up to one half of the end systems that are capable of running IPv6 do so via the auto-tunnelling technologies of 6to4 and Teredo. (http://www.potaroo.net/bgp/stats/1x1/v6types.png).

Tunnel Behaviour

Tunnels add to the potential for encountering trouble in an end-to-end IPv6 connection.

- A tunnel "inflates" the packet. A tunnel adds an additional overhead of 20 bytes for a basic IPv4 header, 24 bytes for a GRE tunnel header, or 40 bytes for a UDP-based IPv4 header. There are other forms of tunnels as well, such as the 8 byte PPPoE header, or the overheads of the AH and ESP headers of IPSEC tunnels. This additional header overhead implies that the tunnel's MTU is smaller than the "raw" interface MTU.

- A tunnel is not aware of its own "path". There may be further tunnels "inside" the tunnel, so that the tunnel ingress MTU is not necessarily aware of the tunnel path MTU.

- The routing of the interior of the tunnel may change, so that the tunnel path MTU may be variable.

However, the default behaviour of IPv4 tunnels should be benign, or so you would think. The outer IPv4 "wrapper" will have DF bit (the IPv4 "Don't Fragment") on the tunnel header cleared, so that a MTU mismatch within the tunnel will cause the tunnel packet to be fragmented. The fragmentation is not visible on an end-to-end basis as the tunnel egress has the responsibility to assemble all the original IPv4 packet fragments. If you added the IPv4 tunnel wrapper to the packet before attempting to pass it into the tunnel your assumption would be fine. But that's not the way its done. The tunnel is regarded as the same as any other interface, and if the original packet can't fit into the ingress point of the tunnel then it will get fragmented before having the tunnel encapsulation added. So if the original IPv6 packet is too large to fit into the tunnel interface without fragmentation, then the original packet is discarded and an ICMPv6 message is generated to flag the MTU mismatch.

So what happens when an IPv6 packet encounters a simple protocol 41 IP tunnel? The tunnel will add 20 octets of IP packet header to the original IPv6 packet. This would imply that a MTU of 1500 at the interface level translates to a tunnel MTU of 1480. If the IPv6 packet is 1480 octets or smaller it will be accepted by the tunnel. If the packet is larger than 1480 octets then an ICMPv6 "packet too big" message will be directed back to the IPv6 source address and the IPv6 packet will be discarded.

Oddly enough, once the packet has been passed into the tunnel then the default case is that further fragmentation can be performed on the tunnel packet, as the common default option is to use an IPv4 tunnel header with the DF bit cleared. For such fragmentation conditions, packet reassembly is performed at the tunnel endpoint, rather than at the inner packet's ultimate destination. For low speed tunnels this may probably be benign behaviour. For higher speed situations the reassembly packet load at the tunnel egress may be unacceptably high.

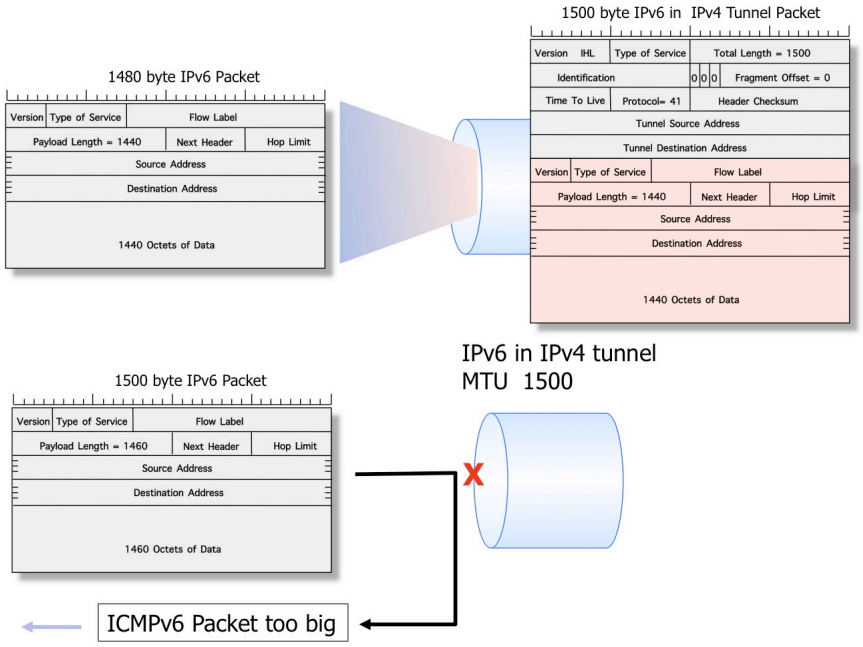


Figure 1 – IPv6 tunnels

To avoid this, the tunnel can be configured to map the encapsulated packet's DF bit to the outer wrapper IP packet. As long as the tunnel ingress point is prepared to perform ICMP relay functions and remap the reverse tunnel ICMP message into a message that has the tunnel headers stripped out and the original source and destination addresses placed into the IMCP message header then there is the possibility of allowing the end-to-end Path-MTU discovery to take account of the additional tunnel overhead.

It should not be surprising that all of this just gets too complex to maintain operationally, and the pragmatic result of all of these considerations is that most host systems use an MTU of 1500 bytes and most interior routers use an MTU of around 9000 bytes or larger on point-to-point links, and generally avoid going less than 1500 octets on any interior link. As long as tunnels are generally avoided then there is no real path MTU discovery taking place on the Internet, and the firewall and tunnel issues and the NAT treatment of ICMP messages are largely irrelevant in such a uniform MTU environment, and most of the Internet appears to work acceptably well for most of the Internet's users.

But IPv6 is different, particularly in its current use of auto-tunnelling and its different treatment of packet fragmentation.

Packet Fragmentation in IPv6

IPv6 takes a much more stringent approach to packet fragmentation than IPv4. IPv6 assumes that all TCP sessions in IPv6 have Path MTU discovery capability, and also assumes that all UDP applications can also perform some equivalent form of path MTU discovery.

The result of this design assumption is that all fragmentation control fields are removed from the base IPv6 packet header. All IPv6 routers, or any other intermediate system, must not attempt to perform packet fragmentation on an IPv6 packet. If an IPv6 packet is too large for the next hop interface, then the router must discard the IPv6 packet and generate an ICMPv6 "Packet too big" message and send this back to the IPv6 source. In the case of TCP the IPv6 host system should perform Path MTU discovery based on these ICMPv6 messages, and avoid performing packet fragmentation at the source. In other cases, such as UDP, the upper level protocol driver may be able to reformat the original payload data into multiple IPv6 packets. IPv6 allows the source of

the packet to perform payload fragmentation, and generate a number of IPv6 packets, each with a fragmentation control header that plays a similar role to the IPv4 fragmentation control fields.

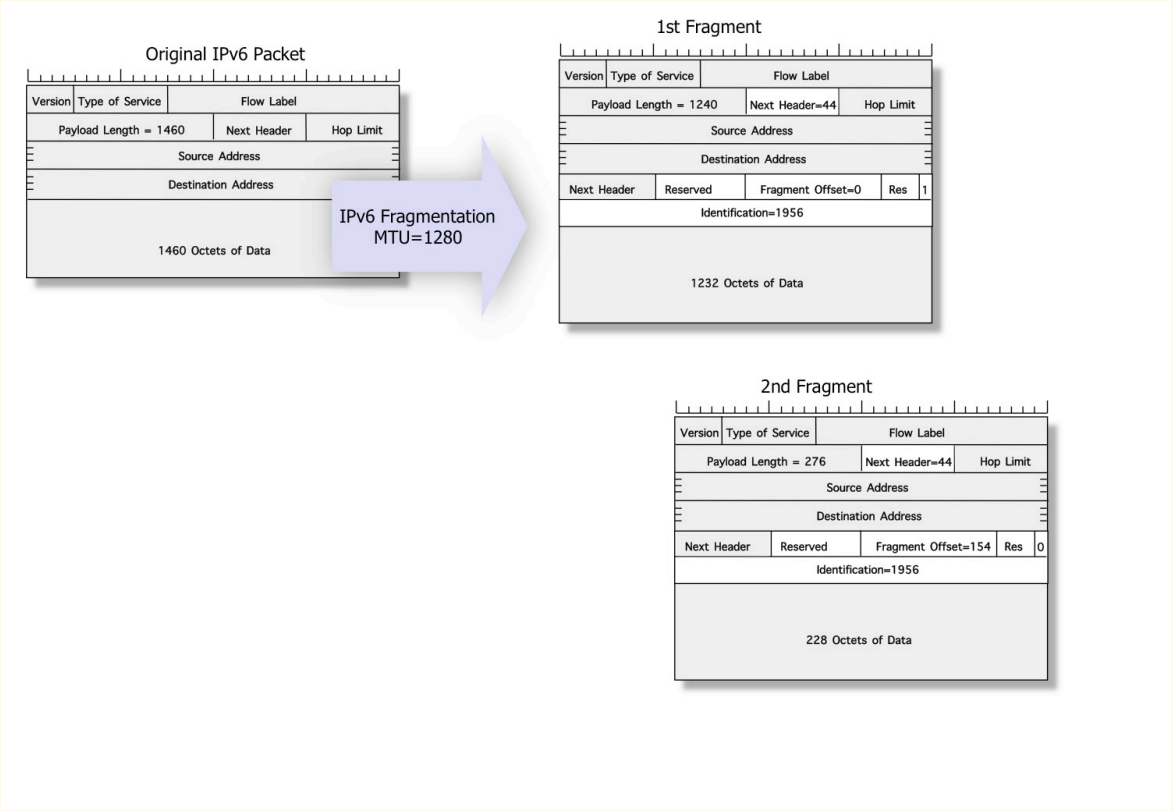
IPv6 Fragmentation Control

The IPv6 packet header has a 16 bit unsigned Payload Length field, indicating the length of the packet, less the 40 byte IPv6 packet header. This allows for a maximum "normal" IPv6 packet size of 65575 octets. IPv6 may include a *jumbogram* header that permits larger packets of up to 4G bytes in size, although router support for such large packets is optional.

All IPv6 hosts and routers must pass packets up to 1280 octets in length.

Fragmentation of a IPv6 packet may only be performed at the source point of the packet. No further fragmentation, nor any form of fragment reassembly, is attempted by any intermediate device. Once fragmented at the source, an IPv6 packet is reassembled only at the point of the final destination, as per the IPv6 packet's destination address.

As with IPv4, IPv6 uses three control fields, the *Packet Identification*, *More Fragments Flag*, and *Fragment Offset* fields. These fields are formatted into an 8-byte IPv6 *Fragmentation Header*, referenced using the IPv6 Next Header code of 44.



When a host fragments an IPv6 packet is adds a *Fragmentation Header* to the IPv6 packet. The header has three control fields: *Identification*, *More Fragments*, and *Fragmentation Offset*:

- *Identification.* A 32-bit value used to identify all the fragments of a packet, allowing the destination host to perform packet reassembly.

- *More Fragments Flag*. When a packet is fragmented, all packets except the final fragment have the *More Fragments* flag set. The fragmentation algorithm operates

such that only the final fragment of the original IP packet has this field clear (set to zero).

- *Fragmentation Offset Value*. This 13-bit value counts the offset of the start of this payload fragment from the start of the original packet. The unit used by this counter is octa-bytes, implying that fragmentation must align to 64-bit boundaries.

The fields altered by fragmentation are shown in the figure above, where a 1500-byte IP packet has been fragmented into a 1280-byte packet and one 316-byte packet. The IP packet length has been altered to reflect the fragment size, and the *Fragmentation Offset Value* field has been set to 154 respectively. The final fragment has the *More Fragments* flag cleared to show that it is the final fragment of the original packet.

IPv6 defines an explicit ordering of IPv6 packet headers:

- IPv6 packet header
- Hop-by-Hop Options header
- Destination Options headers (intermediate destinations)
- Routing header
- Fragment header
- Authentication header
- Encapsulating Security Payload header
- Destination Options header (final destination)
- Upper-layer header

The first four header types form the "un-fragmentable part" of the packet, and are reproduced in the headers of every fragment packet, while the final four header types are treated by the IPv6 fragmentation algorithm as a part of the payload, and are not reproduced in every fragment's header.

The basic behaviour for IPv6 TCP is to avoid fragmentation. IPv6 TCP uses the local MTU as the initial local MSS value (adjusting the MTU for the IP and TCP packet headers), and then use the minimum of this MSS value and the remote party's MSS value as the session MSS value, deriving an initial MTU value by adjusting to allow for the IP and TCP packet headers. This initial MTU is used as the initial path MTU estimate. One the TCP connection is established the sender will rely on incoming ICMPv6 "packet too big" messages to trigger the local TCP instance to use a smaller MTU, using the MTU indicated in the ICMPv6 packet as the new MTU. So as long as the sender is receiving ICMPv6 messages then TCP should adjust correctly when the initial Path MTU estimate is too high.

But if there is a condition that prevents the source from receiving packet-too-big ICMPv6 messages then the algorithm fails, and the application may hang when full-sized TCP packets are passed through the network. In some cases this may happen at a point well distanced from the two endpoints of the TCP session, so that the ICMPv6 filtering may be occurring at a point that is not under the control of the source or the destination.

Over the years there has been a fair amount of popular folklore that has cemented itself in the form of firewall "rules". One of the more pervasive forms of these networking myths is that all incoming ICMP has nothing useful to say to the local network, and can often become a source of DoS attack. Many firewalls place ICMP into the general category of "unsolicited traffic" and simply block all forms of ICMP, including this ICMPv6 "packet too big" message. The incidence of ICMP delivery failure remains in the "uncomfortably high" category and, surprisingly, ICMP filtering is still seen on some transit routes. However, tunnelling adds to this level of uncertainty. The location of the tunnel ingress and egress points are not readily determined because of the use of anycast addresses in the common auto-tunnelling mechanisms. The implication is that even when the transit path appears to allow ICMP messages, a tunnel ingress point may lie on the other side of an edge firewall that is blocking ICMP delivery.

Setting the Interface MTU

This risk of service failure for some service clients is the basic reason why so many web server systems are averse to configuring themselves as dual stack IPv4 and IPv6 servers. The problem is that through no fault of their own in the local configuration of the IPv6 server, and through no fault in the configuration of the IPv6 client, there are situations where the application fails, even though every part of the system appears to be functioning. These faults which happen despite everything looking to be working correctly are of course the most difficult, and hence the most expensive, to diagnose and correct.

How can this be fixed?

The "fix" answer is to locate and reprogram every single device on the Internet that is blocking ICMPv6 messages. That is of course a somewhat ambitious and totally impractical response. So are there local actions you can take that will simply help make IPv6 work for you? A more practical question is:

How can this be avoided?

The "avoid" answer is to use a local MTU setting that does not require fragmentation for tunnels. The client could be configured to use an IPv6 MTU that is at least 40 octets smaller than the MTU of the attached subnet. But what about multi-layer tunnels? How low should you go with this setting to ensure clear end-to-end un-fragmented connectivity? If you want to avoid all forms of path MTU packet fragmentation then the appropriate response is to use an interface MTU equal to the minimum IPv6 packet size, namely 1280 octets.

So if you are a dual stack server and you want to maximise the probability that all IPv6 clients will successfully connect to your service over IPv6, then the simple approach here is to drop the server's MTU to 1280.

```
# ifconfig en0 mtu 1280
```

This would be enough to get over most of these fragmentation issues. It would have the side effect of dropping the packet size for all packets, but the performance impact of such an alteration in the maximum packet size is relatively minor in all but the most demanding of environments.

The same advice applies to dual stack clients. If you are seeing your browser deliver a white screen and get stuck in "loading…" then try dropping your local MTU size.

Configuring a Local 6to4 Interface

The second piece of advice concerns making 6to4 more robust. As already noted, 6to4 these days appears to represent up to one half of all the potential IPv6 clients, so making 6to4 work well is extremely important.

6to4 is asymmetric, and perhaps it is this asymmetry that adds to the potential for breakage.

In the 6to4 model the IPv4 client has no local IPv6 connectivity. In order to send an IPv6 packet, it has to undertake a couple of additional actions. It uses a synthetic IPv6 address which is generated by joining the prefix 2002::/16 to its local IPv4 address. The outbound IPv6 packet is "wrapped" up with a 20 octet IPv4 header using IP protocol number 41. The destination address in the IPv4 packet is 192.88.99.1, which is the anycast address used by 6to4 relays. The IPv4 packet is routed over IPv4 to the "closest" 6to4 outbound relay ("close" in terms of anycast

routing of 192.88.99.0/24). The 6to4 outbound relay strips off the IPv4 header and sends the inner IPv6 packet towards the original IPv6 destination.

The IPv6 server side requires no change at all. It receives an IPv6 packet that has a source address drawn from the address block 2002::/16. It can simply reply to this using the source address of the incoming packet as its destination address. However that alone is insufficient to get that packet back to the client. The server is hoping that there are inbound 6to4 relay servers announcing a route to 2002::/16. If that's the case the IPv6 packet will get forwarded to the "closest" such relay server. The relay server will generate an IPv4 header, using the IPv4 destination address it pulled out of the IPv6 address and using its own IPv4 address as the source address.

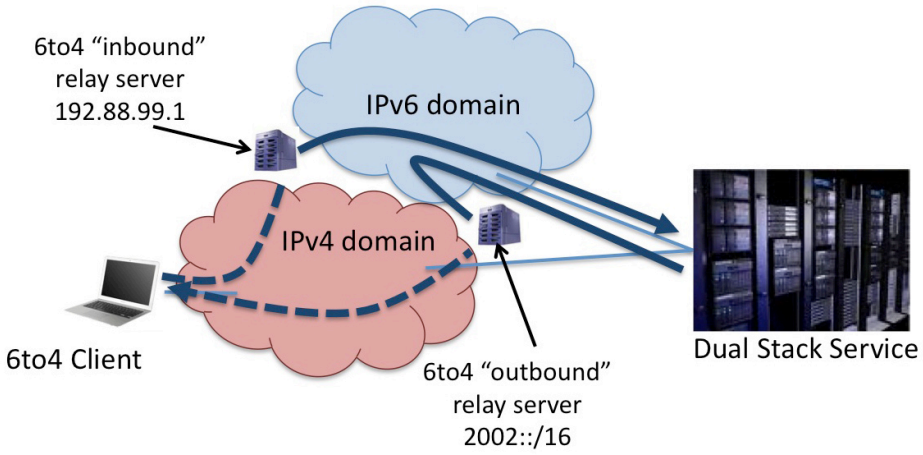


Figure 2 – 6to4 Data Paths

Of note here is that the two relay servers will probably be different servers, and the path taken by the outbound and inbound packets will probably differ. The outbound relay server that strips off the IPv4 wrapper and forwards the inner IPv6 packet across the IPv6 network is "close" to the client while the inbound relay server is "close" to the server. The IPv6 MTU of the inbound relay server is 20 octets smaller than the server's MTU, assuming the general use of the 1500 octet MTU, so there is a strong likelihood that the server should receive ICMPv6 "packet too big" responses if it attempts to send 1500 octet packets to a remote 6to4 client. However, if the server is in the "inside" of some firewall device that discards ICMP, then the potential for problems increases. The other problem here is the potential variability of the server's path, as it necessarily involves the participation of a 6to4 relay that is reachable via the IPv6 route to 2002::/16. As this is any anycast route, there is no assurance that the path is stable.

How can a dual stack server mitigate these issues with 6to4?

One simple approach is to run the outbound 6to4 relay on the dual stack server. The server is now able to perform the tunnel encapsulation locally, and generate outbound IPv4 protocol 41 packets that are destined to the client without the need for active relays. The server uses a host route to pass all packets destined to 2002::/16 to the local 6to4 interface, and routes the IPv6 default route as normal.

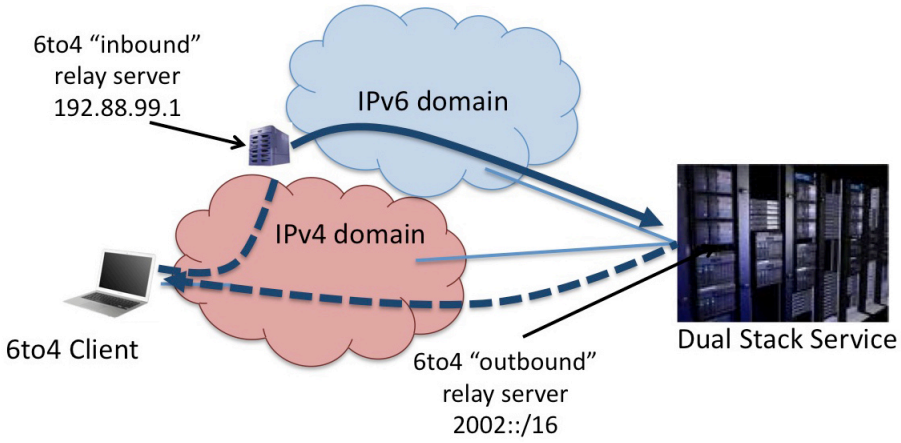


Figure 3 – Local 6to4 at the Server

Time to Get Moving!

I have heard that in terms of the server side of this transition converting an IPv4 server, whether it’s a HTTP server, a mail server, a DNS server, or any other form of service, the task is basically straightforward. It’s a case of turning on the IPv6 protocol support in the service platform, configuring the various forms of routing, switching, load sharing, and filtering middleware to act consistently across both IPv4 and IPv6, and of course obtaining an IPv6 packet transit service. And while there are details to work through in each part of this overall activity, its certainly not a great leap into the unknown. Yet the number of web sites that are reachable using IPv6 remains frustratingly low. (See http://speedlab01.cmc.co.ndcwest.comcast.net:8088/monitor/ for a report on the monitoring of IPv6 web site reachability in the top 1M site list maintained by Alexa.)

Maybe its because there are these lingering concerns over IPv6, where even if you have done everything by the book there will still be clients who will no longer be able to access your service. I'd like to think that these doubts are without foundation, and with a little additional care in locally managing your MTU and in assisting those clients out there who are using 6to4, you can set up a fully functional dual stack service environment that is as reliable and as accessible as a single protocol IPv4 service.

It really is time to stop just talking about conversion of these IPv4-only services to Dual Stack with IPv6, and start actually doing it!

Disclaimer

The above views do not necessarily represent the views or positions of the Asia Pacific Network Information Centre, nor those of the Internet Society.

Author

Geoff Huston B.Sc., M.Sc., is the Chief Scientist at APNIC, the Regional Internet Registry serving the Asia Pacific region. He has been closely involved with the development of the Internet for many years, particularly within Australia, where he was responsible for the initial build of the Internet within the Australian academic and research sector. He is author of a number of Internet-related books, and was a member of the Internet Architecture Board from 1999 until 2005, and served on the Board of Trustees of the Internet Society from 1992 until 2001.

www.potaroo.net