The ISP Column
A monthly column on things Internet

January 2009

Geoff Huston

A Tale of Two Protocols:

IPv4, IPv6, MTUs and Fragmentation

I have seen a number of commentaries and presentations in recent times that claim that IPv6 is identical to IPv4 in every respect except one: namely more addresses. Of course that's not just "more" addresses in the sense that 128 addresses are "more" than 32 addresses, but that's 2 to the power 96 times "more" addresses. Here we're talking massively, unimaginably massively, "more" addresses in IPv6! I must admit to some sympathy for such a claim given that I find the assertions that IPv6 provides superior QoS capability, better, security, improved mobility support or better anything else, as compared to IPv4, to be an expression of largely wishful thinking. There have been some minor tweaks in IPv6 in this respect, but nothing very major.

But there is one rather critical difference, and that is the deliberate change in the IPv6 with respect to MTU handling and packet fragmentation, and this relatively minor change in IPv6 has some really quite critical implications. In this article I'd like to illustrate some of the implications of this change with respect to the IPv6 treatment of packet fragmentation by taking an in-depth look at the IPv6 packet flows and why and how this change to packet fragmentation management can cause service-level disruption.

But first lets start with what has triggered my attention to this topic of IPv6 packet fragmentation.

On my laptop running Mac OS X I have two web browsers, the Apple-provided Safari browser, and Firefox. A situation I encountered recently was that I could enter precisely the same URL into both browsers at the same time. The Firefox browser would correctly display the document. The Safari screen would display a status message saying that it is waiting for the server to provide the requested data, and the screen remains blank indefinitely. Well maybe not indefinitely, but I ran out of patience after about 2 minutes, given that the other browser was able to display the page immediately. After restarting the browser and trying a few more times it was clear that whatever the problem was, it was not a transient condition, and it could be exercised on demand.

What I had encountered was one browser working as expected, while the other did not, with both browsers sitting on the same host, retrieving the same URL, at the same time. Is this a case of a bug being exercised in Safari? Or some special feature of Firefox that allows the page to be displayed? Or something completely random? Not at all! As we'll see, Safari is working perfectly. In fact every component on my Mac is working correctly. And every component of the remote server is also working correctly. So if there no nothing wrong at either end, then why does one browser correctly display the web page and the other display nothing, particularly when it looks like there is nothing out of the ordinary happening with my local configuration, the remote server or with the file being retrieved.

I suspect that this kind of behaviour is the very behaviour that keeps sites such as http://www.google.com from operating in dual stack mode and why, so far, they've put Google's IPv6 support at a different URL, namely http://ipv6.google.com.

The first step along the diagnostic path to look at the URL being retrieved: http://www.rfc-editor.org/authors/rfc5398.txt.

> *This is a transient URL, and the link will disappear once the document has been published as an RFC. But any large document will do, and if you are wanting to see if you can replicate the problem in your corner of the net, try http://www.rfc-editor.org/rfc-style-guide/rfc-style-manual-08.txt.*
>
> *As we'll see, the only real precondition here is that the document is larger than 1420 bytes.*

The URL object is a plain text object, so there's nothing special in the retried object that would trigger a different response between the two browsers. What about the server itself?

The Environment

Dual Stack Web Server

It appears that the remote server is configured as a dual stack system. A couple of DNS queries can confirm this:

```
$ dig +short www.rfc-editor.org A
128.9.160.27
```

```
$ dig +short www.rfc-editor.org AAAA
2001:1878:400:1:214:4fff:fe67:9351
```

This server, www.rfc-editor.org is a dual homed site, supporting both IPv4 and IPv6. This is generally considered the right thing to do if you want to get all those cute little green ticks in the IPv6 readiness tables, so its reasonable to conclude that the RFC Editor folk are really trying to do the right thing here, which is highly laudable.

So what is going wrong with my system when it tries to access this web site using the Safari browser?

Dual Stack Client

My Mac is also a dual stack system.

```
$ ifconfig en0
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        inet6 fe80::217:f2ff:fec9:1b10%en0 prefixlen 64 scopeid 0x4
        inet 203.10.60.24 netmask 0xffffff00 broadcast 203.10.60.255
        inet6 2001:dc0:2001:10:217:f2ff:fec9:1b10 prefixlen 64 autoconf
        ether 00:17:f2:c9:1b:10
        media: autoselect (1000baseT <full-duplex,flow-control>) status: active
```

In IPv4 the Ethernet interface my system has the address 203.10.60.24, and in IPv6 this interface uses the address 2001:dc0:2001:10:217:f2ff:fec9:1b10.

Dual Stack Web Browser

The typical behaviour of a web browser on a dual-stack host is to first perform an IPv4 and an IPv6 address query for the remote site. If there is an IPv6 address record returned, then the browser will first attempt an IPv6 connection to the server, and will fall back to IPv4 in the connection attempt fails after three attempts. If the initial TCP SYN packet exchange succeeds then the association of the AAA record (and implicitly the use of IPv6 as the connection protocol) is locally cached with the server name, and repeated connections to the same remote server name will invoke an IPv6 TCP connection without any delay.

To illustrate this behaviour, here's a packet dump of a Safari connection request to http://www.ripe.net, where the first set of packets are the DNS queries and responses.

```
** concurrent DNS queries for A and AAA A resource records
09:51:52.878779 IP (tos 0x0, ttl 64, id 1595, offset 0, flags [none], proto UDP (17),
                  length 58) dhcp24.potaroo.net.55123 > uneeda.telstra.net.domain:
                  [udp sum ok] 31014+ A? www.ripe.net. (30)
```

These are *tcpdump* reports. The dump shows the time that the packet was captured, followed by some details of the packet.

09:51:52.878779
This packet was captured at 9:51 am.

IP
The protocol type, in this case IPv4.

(tos 0x0, ttl 64, if 1595, offset 0, flags [none], proto UDP (17), length 58)
The IP packet header. The TOS field value of zero. The TTL field value is 64, and the packet identification value is 1595. The packet has not been fragmented, as the offset value of zero and the More Fragments bit flag is clear. Fragmentation is permitted, as the Don't Fragment flag is clear. The packet is a UDP packet.

dhcp24.potaroo.net.55123 > uneeda.telstra.net.domain:
The IP packet is addressed to 139.130.4.4 (uneeda.telstra.net) from 203.10.60.24 (dhcp24.potaroo.net). The destination UDP port is 53 ("domain"), and the source port is 55123.

[udp sum ok]
The UDP pseudo header checksum is ok.

31014+ A? www.ripe.net.
The query has the identifier value 3104, and is a DNS query for the A Resource Record for the DNS name "www.ripe.net".

```
09:51:52.879015 IP (tos 0x0, ttl 64, id 24361, offset 0, flags [none], proto UDP (17),
                  length 58) dhcp24.potaroo.net.60459 > uneeda.telstra.net.domain:
                  [udp sum ok] 55689+ AAAA? www.ripe.net. (30)

** DNS responses for A and AAAA resource records
09:51:53.058304 IP (tos 0x0, ttl 59, id 27655, offset 0, flags [none], proto UDP (17),
                  length 99) uneeda.telstra.net.domain > dhcp24.potaroo.net.55123:
                  [udp sum ok] 31014 q: A? www.ripe.net. 2/0/0 www.ripe.net. CNAME
                  aquila-www.ripe.net., aquila-www.ripe.net. A 193.0.19.25 (71)

09:51:53.061607 IP (tos 0x0, ttl 59, id 27663, offset 0, flags [none], proto UDP (17),
                  length 111) uneeda.telstra.net.domain > dhcp24.potaroo.net.60459:
                  [udp sum ok] 55689 q: AAAA? www.ripe.net. 2/0/0 www.ripe.net. CNAME
```

```
                aquila-www.ripe.net., aquila-www.ripe.net. AAAA
                2001:610:240:11::c100:1319 (83)
```

The packet dump of the initial DNS query shows that the browser learns both IPv6 and IPv4 addresses for the host name www.ripe.net. The actual server is aquila-www.ripe.net, and the server's IPv4 address is 193.0.19.25, and its IPv6 address is 2001:610:240:11::c100:1319. The browser appears to wait for both DNS queries to complete before proceeding.

As there is an IPv6 address for this server, the browser will then attempt a connection request using TCP over IPv6, and will send an initial SYN packet. Here's the packet dump of the initial TCP handshake.

```
** IPv6 TCP connection handshake
09:51:53.063157 IP6 (hlim 64, next-header: TCP (6), length: 44)
                2001:dc0:2001:10:217:f2ff:fec9:1b10.50680 > aquila.ripe.net.http: S,
                cksum 0x1a63 (correct), 2958875177:2958875177(0) win 65535 <mss
                1440,nop,wscale 5,nop,nop,timestamp 122186584 0,sackOK,eol>

09:51:53.454359 IP6 (hlim 53, next-header: TCP (6), length: 40) aquila.ripe.net.http
                > 2001:dc0:2001:10:217:f2ff:fec9:1b10.50680: S, cksum 0xc9e1
                (correct), 2390113939:2390113939(0) ack 2958875178 win 5712 <mss
                1300,sackOK,timestamp 2432757156 122186584,nop,wscale 7>

09:51:53.454476 IP6 (hlim 64, next-header: TCP (6), length: 32)
                2001:dc0:2001:10:217:f2ff:fec9:1b10.50680 > aquila.ripe.net.http: .,
                cksum 0x8e36 (correct), 1:1(0) ack 1 win 32803 <nop,nop,timestamp
                122186588 2432757156>
```

If the TCP three-way handshake completes, then the browser will "lock on" to using IPv6 for this server name, and will then send the HTTP request, and the server will send its response.

```
** HTTP request and document response, over IPv6
09:51:53.455089 IP6 (hlim 64, next-header: TCP (6), length: 400)
                2001:dc0:2001:10:217:f2ff:fec9:1b10.50680 > aquila.ripe.net.http: P,
                cksum 0x6093 (correct), 1:369(368) ack 1 win 32803
                <nop,nop,timestamp 122186588 2432757156>

                GET / HTTP/1.1
                User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_5_5; en-us)
                AppleWebKit/525.27.1 (KHTML, like Gecko) Version/3.2.1
                  Safari/525.27.1
                Accept:
                text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/
                plain;q=0.8,image/png,*/*;q=0.5
                Accept-Language: en-us
                Accept-Encoding: gzip, deflate
                Connection: keep-alive
                Host: www.ripe.net

** Server's ACK of the request
09:51:53.865482 IP6 (hlim 53, next-header: TCP (6), length: 32) aquila.ripe.net.http
                > 2001:dc0:2001:10:217:f2ff:fec9:1b10.50680: ., cksum 0x0b19
                (correct), 1:1(0) ack 369 win 53 <nop,nop,timestamp 2432757568
                122186588>

** Server commences download of the requested document
09:51:53.890642 IP6 (hlim 53, next-header: TCP (6), length: 1320)
                aquila.ripe.net.http > 2001:dc0:2001:10:217:f2ff:fec9:1b10.50680: .,
                cksum 0x41e3 (correct), 1:1289(1288) ack 369 win 53
                <nop,nop,timestamp 2432757582 122186588>

                HTTP/1.1 200 OK
                Date: Wed, 10 Dec 2008 22:51:53 GMT
                Server: Apache/2
                Accept-Ranges: bytes
                Keep-Alive: timeout=8, max=100
                Connection: Keep-Alive
                Transfer-Encoding: chunked
                Content-Type: text/html

[data download...]
```

If there is no AAAA record for the server name, then the browser will use the IPv4 address and attempt the connection using TCP over IPv4. Here's a connection trace to the IPv4-only server www.google.com:

```
** concurrent DNS queries for A and AAA A resource records
09:52:07.661336 IP (tos 0x0, ttl 64, id 46622, offset 0, flags [none], proto UDP (17),
                length 60) dhcp24.potaroo.net.63484 > uneeda.telstra.net.domain:
                [udp sum ok] 13014+ A? www.google.com. (32)

09:52:07.661434 IP (tos 0x0, ttl 64, id 47662, offset 0, flags [none], proto UDP (17),
                length 60) dhcp24.potaroo.net.53534 > uneeda.telstra.net.domain:
                [udp sum ok] 20578+ AAAA? www.google.com. (32)

** DNS responses for A resource record and no AAAA resource record
09:52:07.746621 IP (tos 0x0, ttl 59, id 5416, offset 0, flags [none], proto UDP (17),
                length 144) uneeda.telstra.net.domain > dhcp24.potaroo.net.63484:
                [udp sum ok] 13014 q: A? www.google.com. 5/0/0 www.google.com. CNAME
                www.l.google.com., www.l.google.com. A 209.85.173.99,
                www.l.google.com. A 209.85.173.103, www.l.google.com. A
                209.85.173.104, www.l.google.com. A 209.85.173.147 (116)

09:52:07.750023 IP (tos 0x0, ttl 59, id 5426, offset 0, flags [none], proto UDP (17),
                length 128) uneeda.telstra.net.domain > dhcp24.potaroo.net.53534:
                [udp sum ok] 20578 q: AAAA? www.google.com. 1/1/0 www.google.com.
                CNAME www.l.google.com. ns: l.google.com. SOA f.l.google.com. dns-
                admin.google.com. 1365498 900 900 1800 60 (100)

** IPv4 TCP connection handshake
09:52:07.751608 IP (tos 0x0, ttl 64, id 7987, offset 0, flags [DF], proto TCP (6),
                length 64) dhcp24.potaroo.net.50684 > mh-in-f99.google.com.http: S,
                cksum 0x6127 (correct), 699722949:699722949(0) win 65535 <mss
                1460,nop,wscale 5,nop,nop,timestamp 122186731 0,sackOK,eol>

09:52:07.945646 IP (tos 0x0, ttl 48, id 56566, offset 0, flags [none], proto TCP (6),
                length 60) mh-in-f99.google.com.http > dhcp24.potaroo.net.50684: S,
                cksum 0xc284 (correct), 1111088365:1111088365(0) ack 699722950 win
                5672 <mss 1430,sackOK,timestamp 566450337 122186731,nop,wscale 6>

09:52:07.945760 IP (tos 0x0, ttl 64, id 39278, offset 0, flags [DF], proto TCP (6),
                length 52) dhcp24.potaroo.net.50684 > mh-in-f99.google.com.http: .,
                cksum 0x0758 (correct), 1:1(0) ack 1 win 65535 <nop,nop,timestamp
                122186733 566450337>


** HTTP request and document response, over IPv4
09:52:07.946356 IP (tos 0x0, ttl 64, id 1629, offset 0, flags [DF], proto TCP (6),
                length 898) dhcp24.potaroo.net.50684 > mh-in-f99.google.com.http: P,
                cksum 0x185b (correct), 1:847(846) ack 1 win 65535
                <nop,nop,timestamp 122186733 566450337>

                GET / HTTP/1.1
                User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_5_5; en-us)
                AppleWebKit/525.27.1 (KHTML, like Gecko) Version/3.2.1
                Safari/525.27.1
                Accept:
                text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/
                plain;q=0.8,image/png,*/*;q=0.5
                Accept-Language: en-us
                Accept-Encoding: gzip, deflate
                Cookie: rememberme=true; __utmx=173272373.; __utmxx=173272373.;
                SID=DQAAAHgAAACE_3ogvcIiR6ultKmbZ5_Y-AkYVLkGp55EL1-
                4cUqvHEmh8Kjm5oUWqWe_fST6hXBA57WrIpr-
                seELG7gSIuK3LNYbCynJ_RIat_weZZn5DfLuG8CvQ3iLQJYdLKTd3Ak5dZEkrKqtqdps
                krCnQD1pcVFO6k1tfgN7euV3yWPNQA; NID=17=Y-
                iTGpiaWGhSMMeczPXYLvN7mnOgNSD6RniNpe-
                IFvR1RjqgPz1_ApuLJ2aJEl973kIrcS3MkbeoZPqRoaCIaVgs5VbbPRjnW7mOOMm3YUT
                Km2RZEUYW_lwLi5h7pz9K; TZ=-660;
                PREF=ID=707b2eb5047c4408:TM=1226974755:LM=1228161958:GM=1:S=VLSk7abZ
                NZuXVtHX
                Connection: keep-alive
                Host: www.google.com

09:52:08.170216 IP (tos 0x0, ttl 48, id 56567, offset 0, flags [none], proto TCP (6),
                length 52) mh-in-f99.google.com.http > dhcp24.potaroo.net.50684: .,
                cksum 0x02b5 (correct), 1:1(0) ack 847 win 116 <nop,nop,timestamp
                566450562 122186733>
```

```
09:52:08.177759 IP (tos 0x0, ttl 48, id 56568, offset 0, flags [none], proto TCP (6),
                length 645) mh-in-f99.google.com.http > dhcp24.potaroo.net.50684: P,
                cksum 0xee85 (correct), 1:594(593) ack 847 win 116
                <nop,nop,timestamp 566450565 122186733>

                HTTP/1.1 302 Found
                Location: http://www.google.com.au/
                Cache-Control: private
                Content-Type: text/html; charset=UTF-8
                Set-Cookie: __utmx=; expires=Mon, 01-Jan-1990 00:00:00 GMT; path=/;
                domain=.google.com
                Set-Cookie: __utmxx=; expires=Mon, 01-Jan-1990 00:00:00 GMT; path=/;
                domain=.google.com
                Date: Wed, 10 Dec 2008 22:52:08 GMT
                Server: gws
                Content-Length: 222

[data download...]
```

The other connection case is where there is both IPv4 and IPv6 addresses and the IPv6 address does not respond. To look at the way in which various systems behave I've set up a domain name, impossible.rand.apnic.net that has both IPv6 and IPv4 addresses. The IPv6 address is 5000::1, which is, deliberately, an unconnected network. The Ipv4 address points to a valid web server at 203.119.0.116. The behaviours of various browsers on various host OS platforms is shown here:

Safari on Mac OSX

```
** IPV6 TCP SYN #1
14:23:03.837528 IP6 (hlim 64, next-header: TCP (6), length: 44)
                2001:dc0:2001:10:217:f2ff:fec9:1b10.52942 > 5000::1.http: S, cksum
                0x70c0 (correct), 3375473401:3375473401(0) win 65535 <mss
                1440,nop,wscale 5,nop,nop,timestamp 285658428 0,sackOK,eol>

** +1 seconds: IPv6 TCP SYN #2
14:23:04.788755 IP6 (hlim 64, next-header: TCP (6), length: 44)
                2001:dc0:2001:10:217:f2ff:fec9:1b10.52942 > 5000::1.http: S, cksum
                0x70b7 (correct), 3375473401:3375473401(0) win 65535 <mss
                1440,nop,wscale 5,nop,nop,timestamp 285658437 0,sackOK,eol>

** +1 seconds: IPv6 TCP SYN #3
14:23:05.789683 IP6 (hlim 64, next-header: TCP (6), length: 44)
                2001:dc0:2001:10:217:f2ff:fec9:1b10.52942 > 5000::1.http: S, cksum
                0x70ad (correct), 3375473401:3375473401(0) win 65535 <mss
                1440,nop,wscale 5,nop,nop,timestamp 285658447 0,sackOK,eol>

** +1 seconds: IPv6 TCP SYN #4 - drop TCP options
14:23:06.790876 IP6 (hlim 64, next-header: TCP (6), length: 28)
                2001:dc0:2001:10:217:f2ff:fec9:1b10.52942 > 5000::1.http: S, cksum
                0x9c26 (correct), 3375473401:3375473401(0) win 65535 <mss
                1440,sackOK,eol>

** +1 seconds: IPv6 TCP SYN #5
14:23:07.792074 IP6 (hlim 64, next-header: TCP (6), length: 28)
                2001:dc0:2001:10:217:f2ff:fec9:1b10.52942 > 5000::1.http: S, cksum
                0x9c26 (correct), 3375473401:3375473401(0) win 65535 <mss
                1440,sackOK,eol>

** +1 seconds: IPv6 TCP SYN #6
14:23:08.793307 IP6 (hlim 64, next-header: TCP (6), length: 28)
                2001:dc0:2001:10:217:f2ff:fec9:1b10.52942 > 5000::1.http: S, cksum
                0x9c26 (correct), 3375473401:3375473401(0) win 65535 <mss
                1440,sackOK,eol>

** +2 seconds: IPv6 TCP SYN #7
14:23:10.795789 IP6 (hlim 64, next-header: TCP (6), length: 28)
                2001:dc0:2001:10:217:f2ff:fec9:1b10.52942 > 5000::1.http: S, cksum
                0x9c26 (correct), 3375473401:3375473401(0) win 65535 <mss
                1440,sackOK,eol>

** +4 seconds: IPv6 TCP SYN #8
14:23:14.800180 IP6 (hlim 64, next-header: TCP (6), length: 28)
                2001:dc0:2001:10:217:f2ff:fec9:1b10.52942 > 5000::1.http: S, cksum
                0x9c26 (correct), 3375473401:3375473401(0) win 65535 <mss
                1440,sackOK,eol>
```

```
** +8 seconds: IPv6 TCP SYN #9
14:23:22.810059 IP6 (hlim 64, next-header: TCP (6), length: 28)
                 2001:dc0:2001:10:217:f2ff:fec9:1b10.52942 > 5000::1.http: S, cksum
                 0x9c26 (correct), 3375473401:3375473401(0) win 65535 <mss
                 1440,sackOK,eol>

** +16 seconds: IPv6 TCP SYN #10
14:23:38.829149 IP6 (hlim 64, next-header: TCP (6), length: 28)
                 2001:dc0:2001:10:217:f2ff:fec9:1b10.52942 > 5000::1.http: S, cksum
                 0x9c26 (correct), 3375473401:3375473401(0) win 65535 <mss
                 1440,sackOK,eol>

** +32.0 seconds: IPv6 TCP SYN #11
14:24:10.868094 IP6 (hlim 64, next-header: TCP (6), length: 28)
                 2001:dc0:2001:10:217:f2ff:fec9:1b10.52942 > 5000::1.http: S, cksum
                 0x9c26 (correct), 3375473401:3375473401(0) win 65535 <mss
                 1440,sackOK,eol>

** After 75 seconds and 11 IPv6 connection attempts, Safari now attempts an IPv4
                 connection
14:24:18.402774 IP (tos 0x0, ttl 64, id 3691, offset 0, flags [DF], proto TCP (6),
                 length 64) dhcp24.potaroo.net.52946 > wattle.apnic.net.http: S,
                 cksum 0x9348 (correct), 1099161300:1099161300(0) win 65535 <mss
                 1460,nop,wscale 5,nop,nop,timestamp 285659173 0,sackOK,eol>

** remote system TCP response
14:24:18.435474 IP (tos 0x0, ttl 56, id 7273, offset 0, flags [DF], proto TCP (6),
                 length 64) wattle.apnic.net.http > dhcp24.potaroo.net.52946: S,
                 cksum 0x0a60 (correct), 3703888493:3703888493(0) ack 1099161301 win
                 65535 <mss 1360,nop,wscale 6,nop,nop,timestamp 499692607
                 285659173,sackOK,eol>

** Completion of TCP 3 way handshake
14:24:18.435571 IP (tos 0x0, ttl 64, id 17232, offset 0, flags [DF], proto TCP (6),
                 length 52) dhcp24.potaroo.net.52946 > wattle.apnic.net.http: .,
                 cksum 0x49cc (correct), 1:1(0) ack 1 win 65535 <nop,nop,timestamp
                 285659173 499692607>

** Client to Server: HTTP Get command
14:24:18.436202 IP (tos 0x0, ttl 64, id 64549, offset 0, flags [DF], proto TCP (6),
                 length 586) dhcp24.potaroo.net.52946 > wattle.apnic.net.http: P,
                 cksum 0x5925 (correct), 1:535(534) ack 1 win 65535
                 <nop,nop,timestamp 285659173 499692607>

** Server to Client: ACK
14:24:18.470446 IP (tos 0x0, ttl 56, id 7276, offset 0, flags [DF], proto TCP (6),
                 length 52) wattle.apnic.net.http > dhcp24.potaroo.net.52946: .,
                 cksum 0x852f (correct), 1:1(0) ack 1 win 50297 <nop,nop,timestamp
                 499692642 285659173>

** Server to Client: First data packet
14:24:18.506189 IP (tos 0x0, ttl 56, id 7277, offset 0, flags [DF], proto TCP (6),
                 length 1400) wattle.apnic.net.http > dhcp24.potaroo.net.52946: .,
                 cksum 0x83ec (correct), 1:1349(1348) ack 535 win 50297
                 <nop,nop,timestamp 499692668 285659173>
```

In this case Safari took 75 seconds and 11 IPv6 connection attempts before flipping over to IPv4.

Internet Explorer on Windows XP

```
** IPV6 TCP SYN #1
15:27:13.294675 IP6 (hlim 64, next-header: TCP (6), length: 24)
                 2001:DC0:2001:10:7151:ADF3:D2EA:6018.1030 > 5000::1.80: S, cksum
                 0xde0c (correct), 3602820861:3602820861(0) win 16384 <mss 1440>

** +3 seconds: IPv6 TCP SYN #2
15:27:16.566773 IP6 (hlim 64, next-header: TCP (6), length: 24)
                 2001:DC0:2001:10:7151:ADF3:D2EA:6018.1030 > 5000::1.80: S, cksum
                 0xde0c (correct), 3602820861:3602820861(0) win 16384 <mss 1440>

** +6 seconds: IPv6 TCP SYN #3
15:27:23.129986 IP6 (hlim 64, next-header: TCP (6), length: 24)
                 2001:DC0:2001:10:7151:ADF3:D2EA:6018.1030 > 5000::1.80: S, cksum
                 0xde0c (correct), 3602820861:3602820861(0) win 16384 <mss 1440>

** After 23 seconds and 3 IPv6 connection attempts, Safari now attempts an IPv4
                 connection
```

```
15:27:36.255871 IP (tos 0x0, ttl 128, id 1537, offset 0, flags [DF], proto: TCP (6),
                length: 48) xp12.telstra.net.1093 > wattle.apnic.net.80: S, cksum
                0x31d9 (correct), 3032925425:3032925425(0) win 65535 <mss
                1460,nop,nop,sackOK>

** remote system TCP response
15:27:36.291706 IP (tos 0x0, ttl  56, id 8329, offset 0, flags [DF], proto: TCP (6),
                length: 48) wattle.apnic.net.80 > xp12.telstra.net.1093: S, cksum
                0x5600 (correct), 1260360205:1260360205(0) ack 3032925426 win 65535
                <mss 1360,sackOK,eol>

** Completion of TCP 3 way handshake
15:27:36.291799 IP (tos 0x0, ttl 128, id 1539, offset 0, flags [DF], proto: TCP (6),
                length: 40) xp12.telstra.net.1093 > wattle.apnic.net.80: ., cksum
                0x815f (correct), 1:1(0) ack 1 win 65535

** Server to Client: HTTP Get Command
15:27:36.292347 IP (tos 0x0, ttl 128, id 1540, offset 0, flags [DF], proto: TCP (6),
                length: 481) xp12.telstra.net.1093 > wattle.apnic.net.80: P, cksum
                0xd6e2 (correct), 1:442(441) ack 1 win 65535

** Server to Client: First data packet
15:27:36.357766 IP (tos 0x0, ttl  56, id 8331, offset 0, flags [DF], proto: TCP (6),
                length: 1400) wattle.apnic.net.80 > xp12.telstra.net.1093: ., cksum
                0x68a4 (correct), 1:1361(1360) ack 442 win 65535
```

In this case Explorer took 23 seconds and 3 IPv6 connection attempts before flipping over to IPv4.

Firefox on FreeBSD

```
** IPV6 TCP SYN #1
15:03:15.510137 IP6 (flowlabel 0x92322, hlim 64, next-header: TCP (6), length: 40)
                2001:dc0:2001:10:20e:7fff:feac:d687.53162 > 5000::1.http: S,
                284152944:284152944(0) win 65535 <mss 1440,nop,wscale
                1,sackOK,timestamp 2734433859 0>


** +3 seconds: IPv6 TCP SYN #2
15:03:18.509278 IP6 (flowlabel 0x92322, hlim 64, next-header: TCP (6), length: 40)
                2001:dc0:2001:10:20e:7fff:feac:d687.53162 > 5000::1.http: S,
                284152944:284152944(0) win 65535 <mss 1440,nop,wscale
                1,sackOK,timestamp 2734436859 0>


** +3 seconds: IPv6 TCP SYN #3
15:03:21.709264 IP6 (flowlabel 0x92322, hlim 64, next-header: TCP (6), length: 40)
                2001:dc0:2001:10:20e:7fff:feac:d687.53162 > 5000::1.http: S,
                284152944:284152944(0) win 65535 <mss 1440,nop,wscale
                1,sackOK,timestamp 2734440059 0>


** +6 seconds: IPv6 TCP SYN #4 - note the change of TCP options
15:03:28.109243 IP6 (flowlabel 0x92322, hlim 64, next-header: TCP (6), length: 28)
                2001:dc0:2001:10:20e:7fff:feac:d687.53162 > 5000::1.http: S,
                284152944:284152944(0) win 65535 <mss 1440,sackOK,eol>


** +3 seconds: IPv6 TCP SYN #5
15:03:31.309229 IP6 (flowlabel 0x92322, hlim 64, next-header: TCP (6), length: 28)
                2001:dc0:2001:10:20e:7fff:feac:d687.53162 > 5000::1.http: S,
                284152944:284152944(0) win 65535 <mss 1440,sackOK,eol>


** +6 seconds: IPv6 TCP SYN #6
15:03:37.509202 IP6 (flowlabel 0x92322, hlim 64, next-header: TCP (6), length: 28)
                2001:dc0:2001:10:20e:7fff:feac:d687.53162 > 5000::1.http: S,
                284152944:284152944(0) win 65535 <mss 1440,sackOK,eol>


** +12 seconds: IPv6 TCP SYN #7
15:03:49.709159 IP6 (flowlabel 0x92322, hlim 64, next-header: TCP (6), length: 28)
                2001:dc0:2001:10:20e:7fff:feac:d687.53162 > 5000::1.http: S,
                284152944:284152944(0) win 65535 <mss 1440,sackOK,eol>


** +24 seconds: IPv6 TCP SYN #8
```

```
15:04:13.909056 IP6 (flowlabel 0x92322, hlim 64, next-header: TCP (6), length: 28)
                2001:dc0:2001:10:20e:7fff:feac:d687.53162 > 5000::1.http: S,
                284152944:284152944(0) win 65535 <mss 1440,sackOK,eol>

** After 75 seconds and 8 IPv6 connection attempts, Safari now attempts an IPv4
                connection
15:04:30.509305 IP (tos 0x0, ttl  64, id 4878, offset 0, flags [DF], proto: TCP (6),
                length: 60) workdog.potaroo.net.64421 > wattle.apnic.net.http: S,
                3881266848:3881266848(0) win 65535 <mss 1460,nop,wscale
                3,sackOK,timestamp 2734508859 0>

** remote system TCP response
15:04:30.542713 IP (tos 0x0, ttl  56, id 47536, offset 0, flags [DF], proto: TCP (6),
                length: 64) wattle.apnic.net.http > workdog.potaroo.net.64421: S,
                597175425:597175425(0) ack 3881266849 win 65535 <mss 1360,nop,wscale
                6,nop,nop,timestamp 502160966 2734508859,sackOK,eol>

** Completion of TCP 3 way handshake
15:04:30.542788 IP (tos 0x0, ttl  64, id 4879, offset 0, flags [DF], proto: TCP (6),
                length: 52) workdog.potaroo.net.64421 > wattle.apnic.net.http: .,
                1:1(0) ack 1 win 8256 <nop,nop,timestamp 2734508892 502160966>

** Server to Client: HTTP Get Command
15:04:30.542935 IP (tos 0x0, ttl  64, id 4880, offset 0, flags [DF], proto: TCP (6),
                length: 465) workdog.potaroo.net.64421 > wattle.apnic.net.http: P,
                1:414(413) ack 1 win 8256 <nop,nop,timestamp 2734508892 502160966>

** Server to Client: ACK
15:04:30.579270 IP (tos 0x0, ttl  56, id 47537, offset 0, flags [DF], proto: TCP (6),
                length: 52) wattle.apnic.net.http > workdog.potaroo.net.64421: .,
                1:1(0) ack 1 win 50297 <nop,nop,timestamp 502161003 2734508892>

** Server to Client: First data packet
15:04:30.606468 IP (tos 0x0, ttl  56, id 47539, offset 0, flags [DF], proto: TCP (6),
                length: 1400) wattle.apnic.net.http > workdog.potaroo.net.64421: .,
                1:1349(1348) ack 414 win 50297 <nop,nop,timestamp 502161019
                2734508892>
```

In this case Firefox took 75 seconds and 8 IPv6 connection attempts before flipping over to IPv4.

The Observed Problem

Back to the problem. What I was seeing on my system was the Firefox would reliably display the document, yet, not matter how I tried I could not get Safari to display the same document. All Safari displayed was a blank screen and a status message saying that Safari was "connecting".

Firstly, here's a dump of the Firefox packets, showing a working connection:

```
** IPv4 TCP connection handshake
09:14:13.807595 IP (tos 0x0, ttl 64, id 884, offset 0, flags [DF], proto TCP (6),
                length 64) dhcp20.potaroo.net.49971 > www.rfc-editor.org.http: S,
                cksum 0x7a82 (correct), 707719249:707719249(0) win 65535 <mss
                1460,nop,wscale 5,nop,nop,timestamp 372220164 0,sackOK,eol>

09:14:14.022077 IP (tos 0x0, ttl 49, id 24131, offset 0, flags [DF], proto TCP (6),
                length 64) www.rfc-editor.org.http > dhcp20.potaroo.net.49971: S,
                cksum 0x388f (correct), 3785256116:3785256116(0) ack 707719250 win
                49232 <nop,nop,timestamp 1649135606 372220164,mss 1460,nop,wscale
                0,nop,nop,sackOK>

09:14:14.022191 IP (tos 0x0, ttl 64, id 24364, offset 0, flags [DF], proto TCP (6),
                length 52) dhcp20.potaroo.net.49971 > www.rfc-editor.org.http: .,
                cksum 0x39a9 (correct), 1:1(0) ack 1 win 65535 <nop,nop,timestamp
                372220166 1649135606>

**Client to Server: HTTP GET request for document
09:14:14.022318 IP (tos 0x0, ttl 64, id 19744, offset 0, flags [DF], proto TCP (6),
                length 565) dhcp20.potaroo.net.49971 > www.rfc-editor.org.http: P,
                cksum 0xe0ff (correct), 1:514(513) ack 1 win 65535
                <nop,nop,timestamp 372220166 1649135606>

                GET /authors/rfc5398.txt HTTP/1.1
                Host: www.rfc-editor.org
```

```
                User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US;
                rv:1.9.0.4) Gecko/2008102920 Firefox/3.0.4
                Accept:
                text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
                Accept-Language: en-us,en;q=0.5
                Accept-Encoding: gzip,deflate
                Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
                Keep-Alive: 300
                Connection: keep-alive
                If-Modified-Since: Fri, 05 Dec 2008 23:42:40 GMT
                If-None-Match: "962f-1544-3b93fc00"
                Cache-Control: max-age=0

**Server to Client: ACK of the HTTP GET Request
09:14:14.245804 IP (tos 0x0, ttl 49, id 24132, offset 0, flags [DF], proto TCP (6),
                length 52) www.rfc-editor.org.http > dhcp20.potaroo.net.49971: .,
                cksum 0x7942 (correct), 1:1(0) ack 514 win 48719 <nop,nop,timestamp
                1649135628 372220166>

**Server to Client: HTTP Server OK response
09:14:14.274626 IP (tos 0x0, ttl 49, id 24133, offset 0, flags [DF], proto TCP (6),
                length 372) www.rfc-editor.org.http > dhcp20.potaroo.net.49971: P,
                cksum 0x04ac (correct), 1:321(320) ack 514 win 48719
                <nop,nop,timestamp 1649135629 372220166>

                HTTP/1.1 200 OK
                Date: Mon, 08 Dec 2008 22:14:14 GMT
                Server: Apache/2.2.4 (Unix) mod_ssl/2.2.4 OpenSSL/0.9.8e DAV/2
                Last-Modified: Mon, 08 Dec 2008 22:10:23 GMT
                ETag: "962f-14fd-4b1231c0"
                Accept-Ranges: bytes
                Content-Length: 5373
                Keep-Alive: timeout=5, max=100
                Connection: Keep-Alive
                Content-Type: text/plain

**Client to Server: ACK of first packet
09:14:14.274728 IP (tos 0x0, ttl 64, id 36637, offset 0, flags [DF], proto TCP (6),
                length 52) dhcp20.potaroo.net.49971 > www.rfc-editor.org.http: .,
                cksum 0x364e (correct), 514:514(0) ack 321 win 65535
                <nop,nop,timestamp 372220169 1649135629>


**Server to Client: First of four back-to-back packets (sender is using an initial
                window size of 4)
09:14:14.284709 IP (tos 0x0, ttl 49, id 24134, offset 0, flags [DF], proto TCP (6),
                length 1500) www.rfc-editor.org.http > dhcp20.potaroo.net.49971: .,
                cksum 0x7b84 (correct), 321:1769(1448) ack 514 win 48719
                <nop,nop,timestamp 1649135629 372220166>

                [data]

**Server to Client: Second data packet
09:14:14.293156 IP (tos 0x0, ttl 49, id 24135, offset 0, flags [DF], proto TCP (6),
                length 1500) www.rfc-editor.org.http > dhcp20.potaroo.net.49971: P,
                cksum 0xd372 (correct), 1769:3217(1448) ack 514 win 48719
                <nop,nop,timestamp 1649135629 372220166>

                [data]

**Client to Server: ACK generated for the first two packets
09:14:14.293227 IP (tos 0x0, ttl 64, id 23065, offset 0, flags [DF], proto TCP (6),
                length 52) dhcp20.potaroo.net.49971 > www.rfc-editor.org.http: .,
                cksum 0x2b0b (correct), 514:514(0) ack 3217 win 65522
                <nop,nop,timestamp 372220169 1649135629>


** Server to Client: Third data packet
09:14:14.497735 IP (tos 0x0, ttl 49, id 24136, offset 0, flags [DF], proto TCP (6),
                length 1500) www.rfc-editor.org.http > dhcp20.potaroo.net.49971: .,
                cksum 0x8d33 (correct), 3217:4665(1448) ack 514 win 48719
                <nop,nop,timestamp 1649135653 372220169>

                [data]

** Server to Client: final data packet
09:14:14.503974 IP (tos 0x0, ttl 49, id 24137, offset 0, flags [DF], proto TCP (6),
                length 1081) www.rfc-editor.org.http > dhcp20.potaroo.net.49971: P,
```

```
                cksum 0x2819 (correct), 4665:5694(1029) ack 514 win 48719
                <nop,nop,timestamp 1649135653 372220169>


                [data]

**Client to Server: ACK of data
09:14:14.504066 IP (tos 0x0, ttl 64, id 48896, offset 0, flags [DF], proto TCP (6),
                length 52) dhcp20.potaroo.net.49971 > www.rfc-editor.org.http: .,
                cksum 0x2137 (correct), 514:514(0) ack 5694 win 65535
                <nop,nop,timestamp 372220171 1649135653>
```

Clearly Firefox is using IPv4 for the retrieval. This is not surprising, as when I look at *about:config* in the Firefox browser, I can see that the local variable *network.dns.disableIPv6* to True. The document is 5373 bytes long, and is downloaded in its entirety in four IPv4 packets, three of 1500 bytes and the final packet of 1081 bytes. Three of these packets have a payload of 1448 bytes, coupled with 20 byte TCP header and a 10 byte TCP timestamp option header, a 2 byte padding field and a 20 byte IPv4 packet header, while the fourth packet contains the remaining 1029 bytes of the document. The IPv4 packet headers have the DF fragment bit set, preventing fragmentation of the packet in transit, as the Apache server driver uses Path MTU discovery and explicitly prevents intermediate systems performing Ipv4 packet fragmentation.

What happens when Safari is used to download the same page? Safari is operating in dual stack mode and the DNS query has resulted in both AAAA and A records, so, as described above, Safari will attempt to undertake the page retrieval using IPv6. Here's the packet dump.

```
**Start TCP session with three-way handshake.
09:14:47.239483 IP6 (hlim 64, next-header: TCP (6), length: 44)
                2001:dc0:2001:10:217:f2ff:fec9:1b10.49978 > www.rfc-editor.org.http:
                S, cksum 0x3306 (correct), 1585949805:1585949805(0) win 65535 <mss
                1440,nop,wscale 5,nop,nop,timestamp 372220498 0,sackOK,eol>
09:14:47.523248 IP6 (hlim 53, next-header: TCP (6), length: 44) www.rfc-
                editor.org.http > 2001:dc0:2001:10:217:f2ff:fec9:1b10.49978: S,
                cksum 0x8667 (correct), 994602419:994602419(0) ack 1585949806 win
                49980 <nop,nop,timestamp 1649138957 372220498,mss 1440,nop,wscale
                0,nop,nop,sackOK>
09:14:47.523340 IP6 (hlim 64, next-header: TCP (6), length: 32)
                2001:dc0:2001:10:217:f2ff:fec9:1b10.49978 > www.rfc-
                editor.org.http: ., cksum 0x0a39 (correct), 1:1(0) ack 1 win 32799
                <nop,nop,timestamp 372220501 1649138957>

**Client to Server: HTTP GET request for document
09:14:47.523780 IP6 (hlim 64, next-header: TCP (6), length: 451)
                2001:dc0:2001:10:217:f2ff:fec9:1b10.49978 > www.rfc-editor.org.http:
                P, cksum 0x7b3b (correct), 1:420(419) ack 1 win 32799
                <nop,nop,timestamp 372220501 1649138957>

                GET /authors/rfc5398.txt HTTP/1.1
                User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_5_5; en-us)
                AppleWebKit/525.27.1 (KHTML, like Gecko) Version/3.2.1
                Safari/525.27.1
                Cache-Control: max-age=0
                Accept:
                text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/
                plain;q=0.8,image/png,*/*;q=0.5
                Accept-Language: en-us
                Accept-Encoding: gzip, deflate
                Connection: keep-alive
                Host: www.rfc-editor.org

**Server to Client: ACK of the HTTP GET Request
09:14:47.825010 IP6 (hlim 53, next-header: TCP (6), length: 32) www.rfc-
                editor.org.http > 2001:dc0:2001:10:217:f2ff:fec9:1b10.49978: .,
                cksum 0xc6ff (correct), 1:1(0) ack 420 win 49561 <nop,nop,timestamp
                1649138985 372220501>

**Server to Client: HTTP Server OK response
09:14:47.825813 IP6 (hlim 53, next-header: TCP (6), length: 352) www.rfc-
                editor.org.http > 2001:dc0:2001:10:217:f2ff:fec9:1b10.49978: P,
                cksum 0x4f67 (correct), 1:321(320) ack 420 win 49561
                <nop,nop,timestamp 1649138985 372220501>

                HTTP/1.1 200 OK
                Date: Mon, 08 Dec 2008 22:14:47 GMT
```

```
                  Server: Apache/2.2.4 (Unix) mod_ssl/2.2.4 OpenSSL/0.9.8e DAV/2
                  Last-Modified: Mon, 08 Dec 2008 22:10:23 GMT
                  ETag: "962f-14fd-4b1231c0"
                  Accept-Ranges: bytes
                  Content-Length: 5373
                  Keep-Alive: timeout=5, max=100
                  Connection: Keep-Alive
                  Content-Type: text/plain

**Client to Server: ACK of the first part of the response
09:14:47.825898 IP6 (hlim 64, next-header: TCP (6), length: 32)
                  2001:dc0:2001:10:217:f2ff:fec9:1b10.49978 > www.rfc-
                  editor.org.http: ., cksum 0x0741 (correct), 420:420(0) ack 321 win
                  32789 <nop,nop,timestamp 372220504 1649138985>

**[The Server sends 4 back-to-back data packets as part of its TCP initial window.
                  The Server's 3 full-size packets are dropped within the network -
                  note the stream offset sequence number at the client end jump from
                  321 to 4605, indicating a drop of three TCP packets each of 1428
                  bytes of TCP payload]

**Server to Client: last part of the document
09:14:48.102778 IP6 (hlim 53, next-header: TCP (6), length: 1121) www.rfc-
                  editor.org.http > 2001:dc0:2001:10:217:f2ff:fec9:1b10.49978: P,
                  cksum 0x65b0 (correct), 4605:5694(1089) ack 420 win 49561
                  <nop,nop,timestamp 1649139014 372220504>

                  [data]

**Client to Server: ACK of previous good data - signalling the data drop
09:14:48.102877 IP6 (hlim 64, next-header: TCP (6), length: 44)
                  2001:dc0:2001:10:217:f2ff:fec9:1b10.49978 > www.rfc-
                  editor.org.http: ., cksum 0x56eb (correct), 420:420(0) ack 321 win
                  32799 <nop,nop,timestamp 372220506 1649138985,nop,nop,sack 1
                  {4605:5694}>

**[Server attempts to resend the data using three single full sized data packet -
                  which gets dropped within the network]

**Server to client: data transmission has finished - send a FIN TCP packet
09:14:52.820267 IP6 (hlim 53, next-header: TCP (6), length: 32) www.rfc-
                  editor.org.http > 2001:dc0:2001:10:217:f2ff:fec9:1b10.49978: F,
                  cksum 0xaec7 (correct), 5694:5694(0) ack 420 win 49561
                  <nop,nop,timestamp 1649139486 372220506>

**Client to Server: ACK of previous good data - signalling the data drop
09:14:52.820373 IP6 (hlim 64, next-header: TCP (6), length: 44)
                  2001:dc0:2001:10:217:f2ff:fec9:1b10.49978 > www.rfc-
                  editor.org.http: ., cksum 0x56bb (correct), 420:420(0) ack 321 win
                  32799 <nop,nop,timestamp 372220554 1649138985,nop,nop,sack 1
                  {4605:5694}>

**[Server attempts to resend the data using a restart, attempting to send the first
                  full sized data packets - which gets dropped within the network]

**[Client aborts the connection]
```

What the Safari browser shows is a blank screen, with the status text: "Contacting "www.rfc-editor.org". The browser appears to be hung. What is confusing to the user is that while the browser will not respond to this particular URL, it will however respond normally to requests for non-existent documents with the correct error display. Here's a dump of a request to the same server for a non-existent document.

```
**Client to Server: HTTP GET request of a non-existent document
09:15:04.376570 IP6 (hlim 64, next-header: TCP (6), length: 425)
                  2001:dc0:2001:10:217:f2ff:fec9:1b10.49980 > www.rfc-editor.org.http:
                  P, cksum 0x31f1 (correct), 394:787(393) ack 467 win 32799
                  <nop,nop,timestamp 372220669 1649140303>

                  GET /authors/rfc5399.txt HTTP/1.1
                  User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_5_5; en-us)
                  AppleWebKit/525.27.1 (KHTML, like Gecko) Version/3.2.1
                  Safari/525.27.1
                  Accept:
                  text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/
                  plain;q=0.8,image/png,*/*;q=0.5
```

```
                Accept-Language: en-us
                Accept-Encoding: gzip, deflate
                Connection: keep-alive
                Host: www.rfc-editor.org

**Server to Client: ACK of the HTTP GET Request
09:15:04.652396 IP6 (hlim 53, next-header: TCP (6), length: 32) www.rfc-
                editor.org.http > 2001:dc0:2001:10:217:f2ff:fec9:1b10.49980: .,
                cksum 0x8067 (correct), 467:467(0) ack 787 win 49194
                <nop,nop,timestamp 1649140670 372220669>
**Server to Client: Response: No such Document
09:15:04.668977 IP6 (hlim 53, next-header: TCP (6), length: 497) www.rfc-
                editor.org.http > 2001:dc0:2001:10:217:f2ff:fec9:1b10.49980: P,
                cksum 0xd42e (correct), 467:932(465) ack 787 win 49194
                <nop,nop,timestamp 1649140670 372220669>

                HTTP/1.1 404 Not Found
                Date: Mon, 08 Dec 2008 22:15:04 GMT
                Server: Apache/2.2.4 (Unix) mod_ssl/2.2.4 OpenSSL/0.9.8e DAV/2
                Content-Length: 217
                Keep-Alive: timeout=5, max=99
                Connection: Keep-Alive
                Content-Type: text/html; charset=iso-8859-1

                <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
                <html><head>
                <title>404 Not Found</title>
                </head><body>
                <h1>Not Found</h1>
                <p>The requested URL /authors/rfc5399.txt was not found on this
                server.</p>
                </body></html>

**Client to Server: ACK of the Server Response
09:15:04.669083 IP6 (hlim 64, next-header: TCP (6), length: 32)
                2001:dc0:2001:10:217:f2ff:fec9:1b10.49980 > www.rfc-
                editor.org.http: ., cksum 0xbead (correct), 787:787(0) ack 932 win
                32784 <nop,nop,timestamp 372220672 1649140670>
```

Diagnosis

The browser using IPv4 appears to be working without any issues at all, while the IPv6 connection is unable to receive some documents.

One possible theory could be that there is some middleware or filter out there that is knocking out the RFC Editor's documents. While some RFCs are a little out of date, and others are of dubious value, I'm pretty sure that even the most incompetent of content filters wouldn't find them offensive!

So is it my browser, my operating system or something to do with my configuration? It does not appear to be so. Other web sites are dual stacked, such as http://www.potaroo.net and http://www.apnic.net work just fine for my local system, suing both Firefox and Safari in both IPv4 and IPv6. And IPv6-only sites, such as, ipv6.google.com also appear to work just fine.

So is it something about the rfc-editor's web server? I can test this out by attempting to connect using a different location. And when I do that, there is no problem at all. When I connect my Mac to the network at another location, both browsers work just fine.

So in looking as to why this is occurring, its likely that this has nothing to do with local configuration, the content of the document, the HTTP protocol or the behaviour of TCP. The major difference here is that of the difference in protocols. A likely hypothesis is that the IPv6 problem here is something about the path between the rfc-editor's web server and my location.

As always in the case of network problems, there are two trusty tools: traceroute and ping, or in the case of IPv6 traceroute6 and ping6. So let's use traceroute6 to check if there is an IPv6 network path between my system and the remote web server.

```
$ traceroute6 2001:1878:400:1:214:4fff:fe67:9351
traceroute6 to 2001:1878:400:1:214:4fff:fe67:9351 (2001:1878:400:1:214:4fff:fe67:9351)
                from 2001:dc0:2001:10:217:f2ff:fec9:1b10, 30 hops max, 12 byte
                packets
 1  2001:dc0:2001:10:2b0:c2ff:fe8d:c034  2.187 ms  2.125 ms  2.108 ms
 2  2001:dc0:2001:249::1  35.696 ms  36.683 ms  38.184 ms
 3  2001:dc0:2001:255::2  166.429 ms  167.06 ms  168.776 ms
 4  2001:200:0:fe00::9c1:1  168.784 ms  169.741 ms  169.496 ms
 5  2001:240:bb01:e::7d  164.829 ms 2001:240:bb01:f::7e  170.876 ms  168.719 ms
 6  tky009bf01.IIJ.Net  165.025 ms tky001bf01.IIJ.Net  166.906 ms tky009bf00.IIJ.Net
                177.922 ms
 7  tky001ix02.IIJ.Net  192.543 ms  166.019 ms  174.734 ms
 8  2001:7fa:7:1::2914:1  267.687 ms  285.419 ms  268.23 ms
 9  xe-5-0-0.a21.tokyjp01.jp.ra.gin.ntt.net  267.699 ms  303.905 ms  271.633 ms
10  xe-9-1.a15.tokyjp01.jp.ra.gin.ntt.net  256.395 ms xe-2-
                1.a15.tokyjp01.jp.ra.gin.ntt.net  267.366 ms  270.076 ms
11  xe-3-0-0.a21.osakjp01.jp.ra.gin.ntt.net  261.096 ms  257.71 ms  259.382 ms
12  ae-1.r20.osakjp01.jp.bb.gin.ntt.net  267.685 ms  263.266 ms  261.358 ms
13  as-1.r20.lsanca03.us.bb.gin.ntt.net  271.908 ms  266.593 ms  285.734 ms
14  po-1.r00.lsanca03.us.bb.gin.ntt.net  269.095 ms  274.537 ms  289.78 ms
15  ge-0.usc-los.lsanca03.us.bb.gin.ntt.net  272.253 ms  255.659 ms  272.387 ms
16  2001:1878:8::2  287.127 ms  273.509 ms  253.279 ms
17  www.rfc-editor.org  274.352 ms  256.764 ms  275.732 ms
```

It's a bit of a strange path - the outbound path appears to start in Australia and then head on a route to Japan, and then across a second system to the west coast of the US, where the server appears to be located. The essential observation is that there is consistent network connectivity here, so its not a broken network path that is causing the problem.

The packet trace reveals that all the packets from my system to the server appear to be getting through, but only some of the packets from the server to my system are making it. In particular, all the small packets are making it through, but the large packets appear to be getting dropped by the network consistently. We can test out this hypothesis with a simple ping test.

```
$ ping6 www.rfc-editor.org
PING6(56=40+8+8 bytes) 2001:dc0:2001:10:217:f2ff:fec9:1b10 -->
                2001:1878:400:1:214:4fff:fe67:9351
16 bytes from 2001:1878:400:1:214:4fff:fe67:9351, icmp_seq=0 hlim=248 time=272.253 ms
16 bytes from 2001:1878:400:1:214:4fff:fe67:9351, icmp_seq=1 hlim=248 time=258.123 ms
16 bytes from 2001:1878:400:1:214:4fff:fe67:9351, icmp_seq=2 hlim=248 time=283.547 ms
16 bytes from 2001:1878:400:1:214:4fff:fe67:9351, icmp_seq=3 hlim=248 time=252.508 ms
16 bytes from 2001:1878:400:1:214:4fff:fe67:9351, icmp_seq=4 hlim=248 time=272.222 ms
16 bytes from 2001:1878:400:1:214:4fff:fe67:9351, icmp_seq=5 hlim=248 time=278.751 ms
16 bytes from 2001:1878:400:1:214:4fff:fe67:9351, icmp_seq=6 hlim=248 time=256.72 ms
^C
--- www.rfc-editor.org ping6 statistics ---
7 packets transmitted, 7 packets received, 0% packet loss
round-trip min/avg/max = 252.508/267.732/283.547 ms
```

So far so good. But that used small packets. Lets try much larger packets:

```
$ ping6 -s 1452 www.rfc-editor.org
PING6(1500=40+8+1452 bytes) 2001:dc0:2001:10:217:f2ff:fec9:1b10 -->
                2001:1878:400:1:214:4fff:fe67:9351
^C
--- www.rfc-editor.org ping6 statistics ---
3 packets transmitted, 0 packets received, 100% packet loss
```

So packets which are set to the local maximum packet size of 1500 bytes are not making it there and back. How about packets that are far smaller that the local maximum size?

```
$ ping6 -s 1400 www.rfc-editor.org
PING6(1448=40+8+1400 bytes) 2001:dc0:2001:10:217:f2ff:fec9:1b10 -->
                2001:1878:400:1:214:4fff:fe67:9351
1408 bytes from 2001:1878:400:1:214:4fff:fe67:9351, icmp_seq=0 hlim=248 time=321.879
                ms
1408 bytes from 2001:1878:400:1:214:4fff:fe67:9351, icmp_seq=1 hlim=248 time=322.99
                ms
1408 bytes from 2001:1878:400:1:214:4fff:fe67:9351, icmp_seq=2 hlim=248 time=333.956
                ms
^C
--- www.rfc-editor.org ping6 statistics ---
```

```
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 321.879/326.275/333.956 ms
```

Now we are on to something. Lets see what the maximum packet size that can make it there and back is by looking at a few packet sizes in between.

```
$ ping6 -s 1431 www.rfc-editor.org
PING6(1479=40+8+1431 bytes) 2001:dc0:2001:10:217:f2ff:fec9:1b10 -->
                  2001:1878:400:1:214:4fff:fe67:9351
1439 bytes from 2001:1878:400:1:214:4fff:fe67:9351, icmp_seq=0 hlim=248 time=322.465
                  ms
1439 bytes from 2001:1878:400:1:214:4fff:fe67:9351, icmp_seq=1 hlim=248 time=339.634
                  ms
^C
--- www.rfc-editor.org ping6 statistics ---
3 packets transmitted, 2 packets received, 33% packet loss
round-trip min/avg/max = 322.465/331.049/339.634 ms
```

```
$ ping6 -s 1432 www.rfc-editor.org
PING6(1480=40+8+1432 bytes) 2001:dc0:2001:10:217:f2ff:fec9:1b10 -->
                  2001:1878:400:1:214:4fff:fe67:9351
1440 bytes from 2001:1878:400:1:214:4fff:fe67:9351, icmp_seq=0 hlim=248 time=357.386
                  ms
1440 bytes from 2001:1878:400:1:214:4fff:fe67:9351, icmp_seq=1 hlim=248 time=324.929
                  ms
^C
--- www.rfc-editor.org ping6 statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 324.929/341.158/357.386 ms
```

```
$ ping6 -s 1433 www.rfc-editor.org
PING6(1481=40+8+1433 bytes) 2001:dc0:2001:10:217:f2ff:fec9:1b10 -->
                  2001:1878:400:1:214:4fff:fe67:9351
^C
--- www.rfc-editor.org ping6 statistics ---
3 packets transmitted, 0 packets received, 100% packet loss
```

It appears that packets of size 1480 bytes are just fine, but packets of size 1481 bytes are getting dropped. Now an IPv4 packet header is 20 bytes in length, and when a conventional IPv6-in-IPv4 tunnel is set up, the packet is inflated by the size of a single IPv4 packet header, or 20 bytes. The other observation is that a very prevalent maximum packet size is 1500 octets, or the maximum size of a standard Ethernet frame.

Now we are getting closer to the cause of the problem. The IPv6 packet from the server to my system is being sent out from the server as a full size 1500 octet packet, with 1428 bytes of TCP payload, and 32 bytes of TCP header and 40 bytes of IPv6 header. It is highly probably that somewhere on the path the IPv6 packet is placed into an IPv4 tunnel, and the packet size is inflated to 1520 octets with the addition of the IPv4 packet header. At the tunnel ingress the tunnel has a 1500 octet maximum packet size interface and the tunnel is evidently not permitted to perform the IPv4 fragmentation, or the tunnel ingress has a larger MTU but some interior point in the tunnel has a 1500 octet MTU and the tunnel packet has the IPv4 don't fragment bit set. The oversized IPv6 packet is dropped.

So when we re-examine the initial IPv6 TCP handshake, the client offered the server a MSS of 1440 bytes, or an equivalent assertion that a 1500 octet packet (accounting for a 40 byte IPv6 packet header and a 20 byte TCP header) would be received by the client. Similarly, the server offered the client the same 1440 byte MSS, or the same equivalent assertion about a 1500 octet MTU. In TCP the lower of the two MSS values is used to determine the maximum-sized packet used by both sides in the connection, and this initial value is used as the starting Path MTU by both sides of the connection.

So the problem is that the remote server believes that the MTU of the path to the client is 1500 bytes, while the true MTU is 1480 bytes. So why has the server managed to get confused about the path MTU?

Why is this happening?

There are four factors that contribute to this problem: firstly, the desire to maximize packet size, secondly, the handling of packet fragmentation in IPv4, thirdly, the handling of tunnels, and lastly, the handling of packet fragmentation in IPv6.

Maximizing Packet Size

If fragmentation is causing such a problem then shouldn't we try to avoid the problem completely and just use the minimum packet size for all IP packets?

> Both IPv4 and IPv6 define a "minimum packet size". All IP hosts and routers must be able to pass a packet of this minimum size without resorting to IP fragmentation. The implication is that all IP interfaces must be able to support an MTU of this "minimum size". In IPv4 this is 68 bytes on a hop-by-hop basis, and every IPv4 destination must be able to reassemble a fragmented IPv4 datagram of up to 576 bytes in length.
>
> > RFC791:
> > Every internet module must be able to forward a datagram of 68 octets without further fragmentation. This is because an internet header may be up to 60 octets, and the minimum fragment is 8 octets.
> >
> > Every internet destination must be able to receive a datagram of 576 octets either in one piece or in fragments to be reassembled.
>
> 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 why not keep packet sizes small? After all fragmentation is problematical for firewalls and filters, because fragments do not contain the TCP or UDP port addresses that are a conventional

part of so many filtering roles, and fragmentation is a problem for the destination host in so far as each fragment that has a new IP identifier causes the destination to open up a new fragmentation reassembly context. One possible answer is carriage efficiency, where the desire is to maximize the ration of data payload to protocol header overhead. The larger the packet size the larger the carriage efficiency. But this is a somewhat esoteric argument in a world of high capacity fibre systems.

The most compelling argument is to maximise data performance.

Routers and switches have a limit to their switching capacity, and the limit is one of the number of packets per second that the router can switch. If you really want to see your shiny new room-sized massive router get heated, then fragment all your IPv4 packets down to 68 bytes and send it a 100Gpbs of IP traffic to switch. That's 1.47 billion packets per second! From a switch's perspective the most efficient performance point is when the packet size is maximized.

From an end to end performance perspective packet size is also a factor. TCP inflates its sending window by a Maximum Segment Size (MSS) unit for each received ACK in Slow Start and by one MSS per Round Trip Time in congestion avoidance mode. Smaller packet sizes makes TCP less "responsive" to speed increases, which in practical terms has some impact on end-to-end performance under situations of sustained high volume transfers over a network path that is experiencing some level of traffic contention. This is perhaps countered to some small extent by the observation that TCP is a feedback controlled system and the smaller packets increase the packet "density," which, in turn, increases the level of signalling information which is being passed to the sender. However the basic TCP performance algorithm (see The Appendix of the article on "Evolving TCP" <http://www.potaroo.net/ispcol/2004-08/tcp2.html> for details) says that the bandwidth of a TCP connection is directly proportional to the MSS size and inversely proportional to the Round Trip Time multiplied a factor which can be approximated to the square root of the packet loss rate. So if you can increase the MSS without altering the packet loss rate then you will probably see better TCP performance.

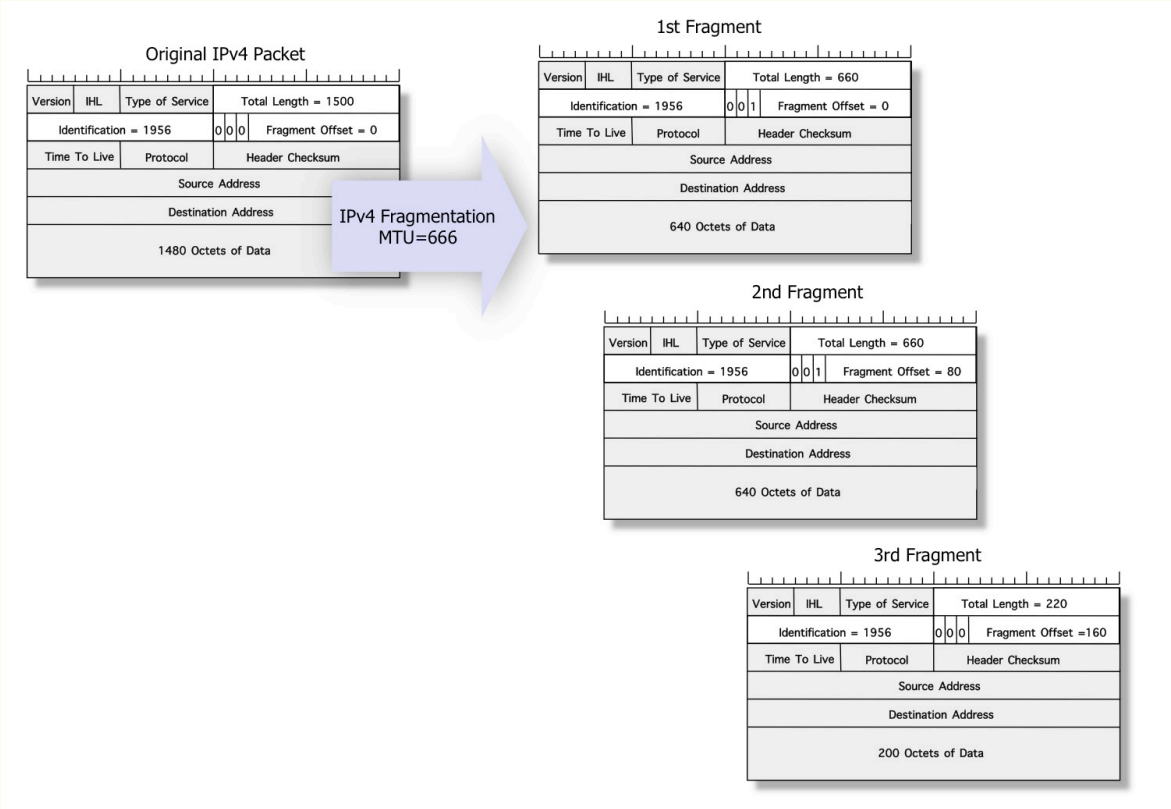
Packet Fragmentation in IPv4

According to the IPv4 technical specifications, all IPv4 hosts and routers must pass packets up to 576 bytes in length. Packets of a larger size may be fragmented by the host or the router. The *IPv4 Packet Identification*, *Fragmentation Flags*, and *Fragment Offset* fields comprise a 32-bit IPv4 packet header segment used to control packet fragmentation. The IP specification allows a router to fragment an oversized packet into smaller units that match the MTU of the next network hop.

Fragmentation may occur more than once within an end-to-end transit path, and an already fragmented packet may be further fragmented without change to the fragmentation functionality. To accommodate this, and to allow some degree of transit efficiency, no reassembly is attempted within the network, implying that all reassembly is performed at the destination point.

> **IPv4 Fragmentation Control**
>
> The IPv4 packet header field *Total Packet Length* is a 16-bit field that specifies the total length of the packet, including both the header and the payload. The value is in units of octets, and the field size of 16 bits allows the maximum IPv4 packet size to be 65,535 octets.
>
> All hosts and routers must pass packets up to 576 octets in length without needing to invoke fragmentation. Packets of a larger size may be fragmented by the host or the router.
>
> The *Packet Identification*, *Fragmentation Flags*, and *Fragment Offset* fields comprise a 32-bit header segment used to control packet fragmentation. Within the design of the IP protocol, every network has a maximum packet size, or maximum transmission unit (MTU),

and these sizes vary. The MTU is an outcome of the network's design and is a product of the network's bandwidth, maximal diameter, and desired imposed jitter. Because sender is not aware of the sequence of MTUs supported on the end-to-end sequence of networks to reach any particular destination, the IPv4 specification allows every router to fragment an oversized IP packet into smaller units that match the MTU of the next hop. Fragmentation may occur more than once within an end-to-end transit path, and an already fragmented packet may be further fragmented without change to the fragmentation functionality. To accommodate this, and to allow some degree of transit efficiency, no fragmentation reassembly is attempted within the network. Once fragmented, a packet is reassembled only at the point of the final destination, as per the IP packet's destination address.



Three IPv4 packet header fields are used in fragmentation control: *Packet Identifier*, *Fragmentation Flags*, and *Fragmentation Offset Value*.

- *Packet Identifier.* A 16-bit value used to identify all the fragments of a packet, allowing the destination host to perform packet reassembly. Note that the packet identifier value cannot be reused while fragments of a previous incarnation of this identifier value remain within the network. For low- to medium-speed networks, this constraint is not a problem, but at gigabit speeds, the wraparound limitation may prove to be a significant limitation. When using maximum-sized packets, fragmentation identifier wraparound occurs every 32 gigabits. When using the more common 576-byte packet size, the fragmentation identifier wraparound is every 256 megabits. As traffic flow speeds increase, minimum packet sizes will need to increase, while at the same time fragmentation capability may have to be dropped as an available option for IP routers, because the combination of the Packet Identifier field and packet size acts as a limit on the amount of data held in transit within the network.

- *Fragmentation Flags*. The three-bit Flag field has the first bit flag reserved. The second bit flag is the *Don't Fragment* flag. When a router attempts to fragment an IP packet with this flag set, no fragmentation occurs, and an ICMP error message is sent back to the sender to inform of the delivery error, and the packet is discarded.

The third bit flag is the *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). When a fragment is further fragmented, this definition of the final fragment refers to the original packet. Thus, fragmentation of an already fragmented packet where the *More Fragments* flag is already set results in a sequence of smaller packets where the final packet still has the *More Fragments* flag set. Fragmentation of an already fragmented packet that was the final fragmented packet, i.e. where the *More Fragments* flag is clear and the *Fragment Offset* is non-zero, will result in a sequence of smaller packets where the *More Fragments* flag is set for all but the final smaller fragment.

- *Fragmentation Offset Value*. This 13-bit value counts the offset of the start of this 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 two 660-byte packets and one 220-byte packet. The IP packet length has been altered to reflect the fragment size, and the *Fragmentation Offset Value* field has been set to 0, 80, and 160 respectively. The final fragment has the *More Fragments* flag cleared to show that it is the final fragment of the original packet.

IP packet fragmentation is not completely transparent in terms of performance, nor is it very secure. If one fragment is lost within the network, the entire packet is discarded by the receiver, as retransmitting just the lost fragment is not a viable option for the protocol. After all, the sender has no knowledge that the packet was fragmented, nor the manner of the fragmentation split. Fragmentation also has resource implications for the destination, where reception of a single fragment implies that the destination has to buffer the fragment and start a reassembly timer to await the reception of the other fragments before completing reassembly. This can be exploited into a resource exhaustion attack by flooding the destination with bogus packets with the *Fragment Offset Value* set to some large non-zero value.

Rather than rely on IP packet fragmentation to adapt to various per-hop packet contortions, TCP can use a path MTU discovery algorithm, as described in RFC1191, to allow the TCP session to proceed without the use of fragmentation. Applications that use Path MTU discovery send all packets with the *Don't Fragment* flag bit set, preventing any router from performing fragmentation on the packet flow. When a router cannot forward the packet because of an MTU mismatch the router discards the packet and generates a return ICMP Destination Unreachable message to the source address with an ICMP code of "fragmentation needed and DF set". The router should place the value of the next hop MTU into the ICMP message, if it supports the RFC1191 mechanism. Upon receipt of this IMCP message the TCP source can set its MTU down to the next hop MTU if that information is available in the ICMP message, or perform some other form of MTU reduction to adapt to the path MTU. If all else fails, the sender can drop down to the minimum size IPv4 packet that does not require fragmentation, namely 576 bytes.

This algorithm explicitly reverses the information flow of the fragmentation situation. IPv4 without TCP Path MTU discovery essentially signals the information about the fragmentation condition forwards, towards the destination. Path MTU reverses this, and makes the intermediate router send the fragmentation backwards, back towards the message source. This would normally be a pretty trivial semantic change were it not for two additional factors: ICMP filters and tunnels.

The problem with this approach is that many filters and firewalls treat ICMP packets as potentially hostile, and they often block the ICMP packet's traversal through the firewall. The second problem is that the use of NATs has the potential to further confound the matter because the NAT may not perform recognise the packet because the incoming packet has the source address of the intermediate system, and therefore may not be recognised as being part of the packet flow belonging to an already established binding. It may also be the case that the NAT may not

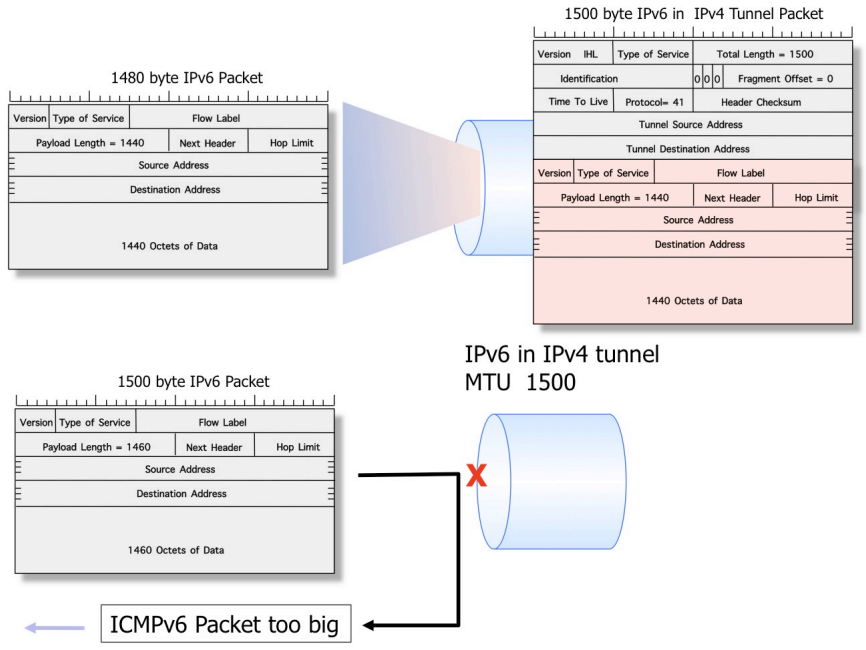
perform the appropriate address transform on the inner IP packet carried as a payload to the outer ICMP packet. The result is that because the addressed host may not be aware that it is behind a NAT, let alone be aware of the NAT's IP translation binding, the host therefore may not even recognise the ICMP packet as being directed to itself, even if it is passed through the firewalls and NATs.

Tunnels

Tunnels add to the complexity here. The first problem with a tunnel is that it 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. The second problem with a tunnel is that there may be further tunnels "inside" the tunnel, so that the tunnel ingress is not necessarily aware of the true tunnel MTU. The third problem is that the routing of the interior of the tunnel may change, so that the tunnel 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 on the tunnel header cleared, so that a MTU mismatch with the tunnel will cause the 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 IP packet fragments. And 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 packet has the DF bit set then if the packet is too large to fit into the tunnel interface without fragmentation, then the original packet is discarded and an ICMP message is generated to flag the MTU mismatch.



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.

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.

Of course VPN users can experience some very strange outcomes, but as long as the VPN tunnel points are carefully configured, most VPNs can be made to work relatively robustly.

But IPv6 can be different, particularly in its 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 or "raw" IP, upper level protocol driver may not be able to reformat the original payload data into multiple IPv6 packets, and prefer to have the remote upper level protocol instance receive a single packet payload. 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.

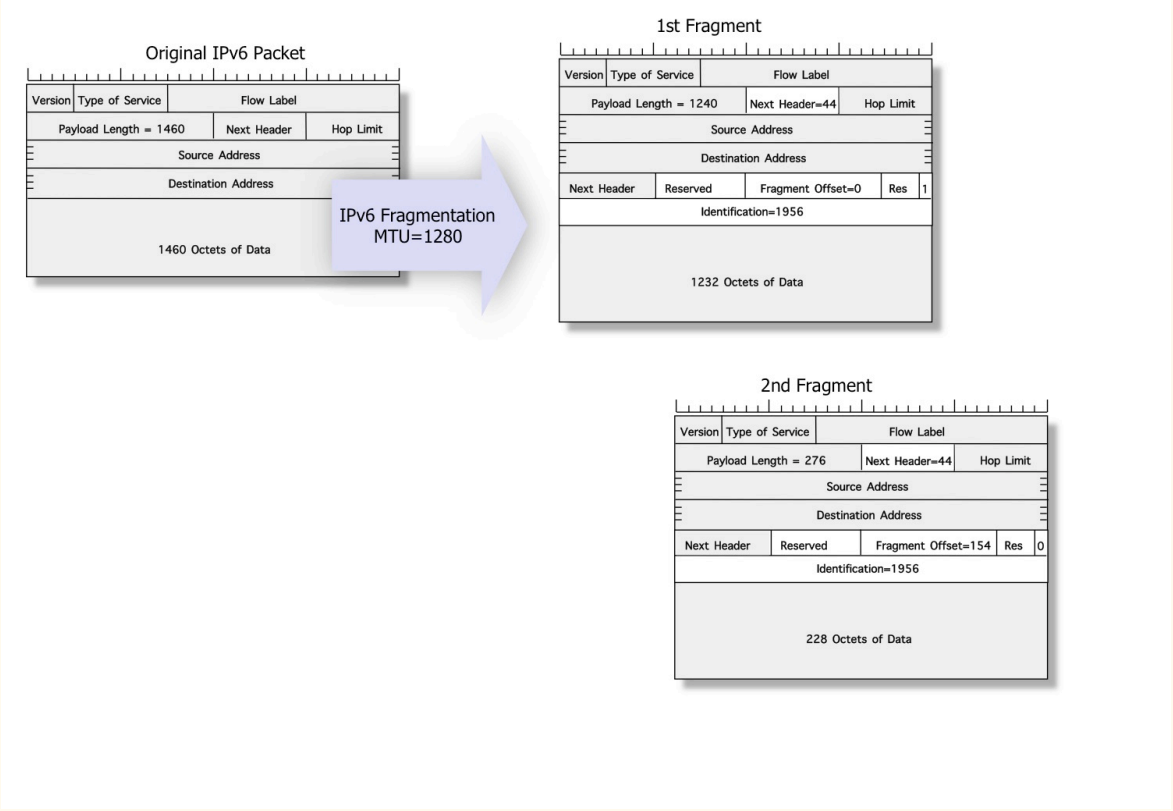
IPv6 Fragmentation Control

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 "unfragmentable 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 packet-too-big ICMPv6 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.

This 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.

What's the answer?

I suppose that there are a number of approaches, depending on what part of the network you want to alter. The following is by no means an exhaustive list, but looks at the most reasonable changes that could fix this problem.

Change the Network's Behaviour

The root cause of the problem here is the missing ICMPv6 packet, which has been swallowed up by either a filter or a poorly constructed NAT.

Filters should permit packet-too-big ICMP messages, and probably should allow a the following set of ICMP message in order to do no harm to IPv6 Path MTU discovery. Here's an example set of ICMP filters.

```
deny   icmp any any fragments
permit icmp any any echo
permit icmp any any echo-reply
permit icmp any any packet-too-big
permit icmp any any time-exceeded
permit icmp any any port-unreachable
permit icmp any any net-unreachable
deny   icmp any any
```

But its not easy to ensure that every possible filter on every possible path is updated to work correctly, and perhaps there are responses that the client or the server might also be able to undertake that would mitigate this "IPv6 hang" behaviour directly.

Change the Client Application Behaviour

The client's dual stack browser behaviour could be altered. The problem here is that the browser operates in serial mode and caches its results. In this case the browser has performed a DNS quad-A query for the server's name and received a IPv6 address in response, and has then successfully completed a TCP handshake. As far as the browser is concerned at the completion of this initial TCP handshake it can talk to this server using IPv6 and it caches this result, and will now not attempt to fall back to use IPv4 within the lifetime of this session. When the HTTP application fails to complete a GET operation using the IPv6 transport, the browser assumes a connectivity failure, and will not attempt to restart TCP using IPv4 and retry the GET operation using IPv4.

It is possible to consider an altered browser behaviour where it would perform the initial DNS name resolution in both IPv4 and IPv6 in parallel, and then perform the initial handshake in parallel as well, and then use the protocol that returns first to perform the HTTP operation, leaving the other protocol channel open, and fall back to use it in the event of a HTTP level timeout of a GET.

Of course this imposes greater levels of load on the dual stack server with all these persistent connections being held open just in case, and of course it really does not solve the underlying problem here that will exhibit itself in any TCP application that attempts to use full size TCP packets, not just HTTP fetches But if the basic catch cry of networking is "every application should fend for itself", then equipping a dual stack client's browser with greater levels of resilience in the face of such issues is a likely response, irrespective of the increase in server load that this may imply.

Change the Client Protocol Stack

The client could be configured to use an IPv6 MTU that is 40 octets smaller than the interface MTU, implicitly allowing for a UDP-based IPv4 tunnel to be present in the path. Using this approach the client would offer an MSS in the initial handshake that is 100 bytes less than the interface MTU. The two TCP endpoints will pick an initial mss that is the minimum of the two MSSs offered in the initial TCP handshake. By offering a lower MSS the client is able to alter the server's behaviour and in effect force the server to use a lower Path MTU, and therefore get around the problem of missing a packet-too-big ICMPv6 message.

So if you are a dual stack client and you are seeing some strange behaviour from servers, namely that they fail to deliver content, and you feel confident you know what you are doing, then the brute force

```
# ifconfig en0 mtu 1400
```

would be enough to get over most of these 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.

Change the Server Protocol Behaviour

And what of the server?

The server could change its application behaviour to interpret a TCP transmission timeout failure as a potential MTU failure after 3 retransmit intervals and try the fourth retransmission with a minimum size IPv6 packet of 1280 bytes.

But asking vendors to change their TCP behaviour to incorporate various patches and workarounds is always going to be a hard ask, and there is always a more direct solution for servers in any case. If you want to increase the reliability of the IPv6 part of your dial stack server, then pick a conservative MTU for your server. What works for the client will work for the server, and

```
# ifconfig <interface> mtu 1400
```

will ensure that the IPv6 service will probably not fail due to undetected Path MTU problems. What I've done for http://www.potaroo.net, which is a dual-stack server, is to drop the MTU as follows:

```
# ifconfig bge1 mtu 1400
$ ifconfig bge1
bge1: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1400
          options=1b<RXCSUM,TXCSUM,VLAN_MTU,VLAN_HWTAGGING>
          inet6 fe80::215:c5ff:fefc:5f07%bge1 prefixlen 64 scopeid 0x2
          inet 203.119.0.116 netmask 0xfffffff0 broadcast 203.119.0.127
          inet6 2001:dc0:2001:7:215:c5ff:fefc:5f07 prefixlen 64
          ether 00:15:c5:fc:5f:07
          media: Ethernet autoselect (100baseTX <full-duplex>)
          status: active
```

This seems to me to represent the most practical approach to making dual stack clients and servers more reliable in the face of these vagaries of network behaviour. Waiting for every filter in the Internet to do the right thing with ICMP messages may well be a fruitless task, and adding further complexity into applications or the TCP protocol behaviour seems to take the long way around the problem. The most effective approach appears to be the simplest one as well - whether you are a dual stack client, or a dual stack server, the best way to get more reliable service under these rather strange corner cases is to drop your MTU.

How to create a Reliable Dual Stack Environment

In listening to a number of presentations, most notably in recent times presentations relating to popular web sites that have avoided configuring a dual stack service, I am struck that there is a widespread perception that switching a server to dual stack creates all kinds of strange problems for end users. I hope the above analysis offers some reasonable explanation as to what is happening in the network to cause these particular problems, and provided some possible approaches as to how servers can avoid the issue.

It seems that if there is one practical lesson from all this its simple: if you are worried about the possible adverse consequences of converting your servers to support a dual stack configuration, and you're waiting for the ICMP filtering problem to be "fixed" across the Internet, then you'll be waiting for a very long time. You can take a very easy step as the system administrator of a dual stack server by simply dropping the MTU of the interfaces used for IPv6 traffic by a further 40 bytes or so, to allow for the possibility of "hidden" tunnels in the IPv6 path to your clients.

At the start of this article I referenced ipv6.google.com. What MSS does this server offer on a TCP connection?

```
**  client to server  - MSS of 1440
05:12:35.661598 IP6 (hlim 64, next-header: TCP (6), length: 44)
                2001:dc0:2001:10:217:f2ff:fec9:1b10.51562 > 2001:4860:b002::68.http:
                S 77702533:77702533(0) win 65535 <mss 1440,nop,wscale
                5,nop,nop,timestamp 285328553 0,sackOK,[|tcp]>
```

```
** server to client - MSS of 1212,
05:12:35.989745 IP6 (hlim 243, next-header: TCP (6), length: 24)
                 2001:4860:b002::68.http > 2001:dc0:2001:10:217:f2ff:fec9:1b10.51562:
                 S, cksum 0xac69 (correct), 704559415:704559415(0) ack 77702534 win
                 8190 <mss 1212>

** completion of 3 way TCP handshake
05:12:35.989841 IP6 (hlim 64, next-header: TCP (6), length: 20)
                 2001:dc0:2001:10:217:f2ff:fec9:1b10.51562 >
                 2001:4860:b002::68.http: ., cksum 0xe32c (correct), 1:1(0) ack 1 win
                 65535

** HTTP GET request
05:12:35.990384 IP6 (hlim 64, next-header: TCP (6), length: 845)
                 2001:dc0:2001:10:217:f2ff:fec9:1b10.51562 > 2001:4860:b002::68.http:
                 P 1:826(825) ack 1 win 65535

** ACK of request
05:12:36.351407 IP6 (hlim 60, next-header: TCP (6), length: 20)
                 2001:4860:b002::68.http >
                 2001:dc0:2001:10:217:f2ff:fec9:1b10.51562: ., cksum 0xc4ab (correct),
                 1:1(0) ack 826 win 6984

** first data packet 1212 bytes of payload, 1272 byte packet
05:12:36.376562 IP6 (hlim 60, next-header: TCP (6), length: 1232)
                 2001:4860:b002::68.http >
                 2001:dc0:2001:10:217:f2ff:fec9:1b10.51562: . 1:1213(1212) ack 826
                 win 6984

** second data packet 1212 bytes of payload, 1272 byte packet
05:12:36.383811 IP6 (hlim 60, next-header: TCP (6), length: 1232)
                 2001:4860:b002::68.http >
                 2001:dc0:2001:10:217:f2ff:fec9:1b10.51562: . 1213:2425(1212) ack 826
                 win 6984

** ACK of first two data packets
05:12:36.383884 IP6 (hlim 64, next-header: TCP (6), length: 20)
                 2001:dc0:2001:10:217:f2ff:fec9:1b10.51562 >
                 2001:4860:b002::68.http: ., cksum 0xd67b (correct), 826:826(0) ack
                 2425 win 65535
```

It appears from this trace that this instance of the site is offering an MSS of 1212, corresponding to an MTU of 1272 if one allows for the conventional 60 bytes of IPv6 and TCP headers per packet. This is 8 bytes less than the minimum supported MTU for IPv6 of 1280 bytes. This is perhaps the most conservative position with respect to tunnelling fragmentation, and maximises to prospects of successfully connecting to the server over IPv6. Although why 1272 and not 1280 is used here remains an open question.

So, for servers, the conservative message if you want a robust dual stack operation is to set the server's MTU for IPv6 to at least 40 bytes less than the interface MTU size, or even to consider resetting this down to 1280 bytes to take the most conservative position.

And the same goes for clients. If you are experiencing strange "hangs" in your local dual stack environment, you might try dropping the local MTU size of your host's network interfaces by some 40 bytes or so before you take the more drastic step of turning off all IPv6 support on your platform.

Postscript

I started this article with a problem I was having with retrieving documents over IPv6 from http://www.rfc-editor.org. The problem has magically been fixed for me, but I didn't do a thing. While I was preparing this material my path to this server changed with the addition of a new tunnel. Here's the new path that I now see to this server.

```
$ traceroute6 www.rfc-editor.org
traceroute6 to www.rfc-editor.org (2001:1878:400:1:214:4fff:fe67:9351) from
                2001:dc0:2001:10:217:f2ff:fec9:1b10, 30 hops max, 12 byte packets
 1  2001:dc0:2001:10:2b0:c2ff:fe8d:c034  1.772 ms  1.233 ms  1.286 ms
 2  2001:dc0:2001:249::1  36.253 ms  35.054 ms  34.582 ms
 3  2001:dc0:8000::2:2  35.669 ms  37.18 ms  36.038 ms
 4  apnic-1.tunnel.tserv1.fmt.ipv6.he.net  231.558 ms  231.723 ms  233.099 ms
 5  1g-3-9.core1.fmt1.ipv6.he.net  239.552 ms  251.8 ms  240.694 ms
 6  10gigabitethernet1-1.core1.pao1.he.net  239.208 ms  234.838 ms  231.211 ms
 7  10gigabitethernet1-1.core1.lax1.he.net  240.625 ms  244.372 ms  246.882 ms
 8  10gigabitethernet1-3.core1.lax2.he.net  240.248 ms  241.717 ms  243.527 ms
 9  2001:504:13::9  244.329 ms  244.358 ms  247.989 ms
10  2001:1878:8::2  242.494 ms  240.967 ms  241.866 ms
11  www.rfc-editor.org  243.669 ms  240.794 ms  243.642 ms
```

Its pretty clear even from the DNS names of the intermediate routers here that there's at least one tunnel in the path, between hops 3 and 4, and the IPv6 Path MTU is no larger than 1480 bytes, so the TCP offered MSS should be 1420 bytes in the initial TCP handshake. However neither client nor server are initially aware of the existence of tunnels, so the conditions are much the same as the earlier condition that had the problem. But this time there is no problem. Safari is successfully pulling in the web pages from the server without a problem:

```
**Start TCP session with three-way handshake - note the offer of a 1440 MSS on both
                sides
21:32:52.200911 IP6 (flowlabel 0x24763, hlim 64, next-header: TCP (6), length: 40)
                2001:dc0:2001:10:20e:7fff:feac:d687.49587 > www.rfc-editor.org.http:
                S, cksum 0x6e41 (correct), 1009278231:1009278231(0) win 65535 <mss
                1440,nop,wscale 1,sackOK,timestamp 2585009958 0>
21:32:52.447775 IP6 (hlim 50, next-header: TCP (6), length: 44) www.rfc-
                editor.org.http > 2001:dc0:2001:10:20e:7fff:feac:d687.49587: S,
                cksum 0x3c26 (correct), 3310917243:3310917243(0) ack 1009278232 win
                49980 <nop,nop,timestamp 1705417549 2585009958,mss 1440,nop,wscale
                0,nop,nop,sackOK>
21:32:52.447852 IP6 (flowlabel 0x24763, hlim 64, next-header: TCP (6), length: 32)
                2001:dc0:2001:10:20e:7fff:feac:d687.49587 > www.rfc-
                editor.org.http: ., cksum 0xbed6 (correct), ack 1 win 32844
                <nop,nop,timestamp 2585010205 1705417549>

**Client to Server: HTTP GET request for document and Server to Client ACK
21:32:52.448050 IP6 (flowlabel 0x24763, hlim 64, next-header: TCP (6), length: 438)
                2001:dc0:2001:10:20e:7fff:feac:d687.49587 > www.rfc-editor.org.http:
                P, cksum 0xe3e2 (correct), 1:407(406) ack 1 win 32844
                <nop,nop,timestamp 2585010205 1705417549>

21:32:52.714093 IP6 (hlim 50, next-header: TCP (6), length: 32) www.rfc-
                editor.org.http > 2001:dc0:2001:10:20e:7fff:feac:d687.49587: .,
                cksum 0x7bcb (correct), ack 407 win 49574 <nop,nop,timestamp
                1705417576 2585010205>

** Server to Client: HTTP: Server OK response
21:32:52.717158 IP6 (hlim 50, next-header: TCP (6), length: 351) www.rfc-
                editor.org.http > 2001:dc0:2001:10:20e:7fff:feac:d687.49587: P,
                cksum 0x4276 (correct), 1:320(319) ack 407 win 49574
                <nop,nop,timestamp 1705417576 2585010205>

** Server to Client: First data packet - note the IPv6 packet size is now 1480 bytes,
                so the Server has performed Path MTU discovery and reduced the MTU
                by 20 bytes. Note also that this packet was received 21 milliseconds
                after the previous packet, indicating that the tunnel point on this
                path is within 10 milliseconds of the server
21:32:52.738526 IP6 (hlim 50, next-header: TCP (6), length: 1440) www.rfc-
                editor.org.http > 2001:dc0:2001:10:20e:7fff:feac:d687.49587: .,
```

```
                cksum 0xd515 (correct), 1:1409(1408) ack 407 win 49574
                <nop,nop,timestamp 1705417577 2585010205>

** Client to Server: ACK
21:32:52.738644 IP6 (flowlabel 0x24763, hlim 64, next-header: TCP (6), length: 32)
                2001:dc0:2001:10:20e:7fff:feac:d687.49587 > www.rfc-
                editor.org.http: ., cksum 0xb8a3 (correct), ack 1409 win 32299
                <nop,nop,timestamp 2585010496 1705417576>

** Server to Client: Second and third data packets and Server to Client ACK
21:32:52.747296 IP6 (hlim 50, next-header: TCP (6), length: 1440) www.rfc-
                editor.org.http > 2001:dc0:2001:10:20e:7fff:feac:d687.49587: .,
                cksum 0xacda (correct), 1409:2817(1408) ack 407 win 49574
                <nop,nop,timestamp 1705417577 2585010205>

21:32:52.749191 IP6 (hlim 50, next-header: TCP (6), length: 391) www.rfc-
                editor.org.http > 2001:dc0:2001:10:20e:7fff:feac:d687.49587: .,
                cksum 0x4b4e (correct), 2817:3176(359) ack 407 win 49574
                <nop,nop,timestamp 1705417577 2585010205>

21:32:52.749294 IP6 (flowlabel 0x24763, hlim 64, next-header: TCP (6), length: 32)
                2001:dc0:2001:10:20e:7fff:feac:d687.49587 > www.rfc-
                editor.org.http: ., cksum 0xb044 (correct), ack 3176 win 32664
                <nop,nop,timestamp 2585010506 1705417577>

** Client opens a new TCP session
21:32:52.771805 IP6 (flowlabel 0x087bc, hlim 64, next-header: TCP (6), length: 40)
                2001:dc0:2001:10:20e:7fff:feac:d687.49588 > www.rfc-editor.org.http:
                S, cksum 0x5ae5 (correct), 3472874335:3472874335(0) win 65535 <mss
                1440,nop,wscale 1,sackOK,timestamp 2585010529 0>

** Server to Client - TCP handshake response - note that the server has now cached
                the smaller Path MTU for this client, and the offered MSS is now
                1420 in the initial TCP handshake
21:32:53.012890 IP6 (hlim 50, next-header: TCP (6), length: 44) www.rfc-
                editor.org.http > 2001:dc0:2001:10:20e:7fff:feac:d687.49588: S,
                cksum 0xd216 (correct), 64991551:64991551(0) ack 3472874336 win
                49280 <nop,nop,timestamp 1705417606 2585010529,mss 1420,nop,wscale
                0,nop,nop,sackOK>

21:32:53.013007 IP6 (flowlabel 0x087bc, hlim 64, next-header: TCP (6), length: 32)
                2001:dc0:2001:10:20e:7fff:feac:d687.49588 > www.rfc-
                editor.org.http: ., cksum 0x5109 (correct), ack 1 win 33088
                <nop,nop,timestamp 2585010770 1705417606>

[and so on…]
```

The difference here is that on this new path there is an ICMPv6 packet-too-big error being generated in response to the server's first full size data packet, but this time it is being successfully passed back to the server site, as it is evident that the server has adjusted its path MTU value for the first large packet. The tentative conclusion of this observation is that the problem in the previous path that appeared to be blocking ICMPv6 packets was probably not necessarily a filter problem within server site. It is more likely that the ICMPv6 filter existed further along the original path, and was probably not connected with either the server's immediate network environment, nor that of the client.

In closing, I should stress that this is definitely not an issue with the RFC editor's web site per se, nor the network and the associated local filters that are used by the RFC Editor's local network environment. I am sure that they operate a professional service and the problem examined here was not related to the RFC Editor's service infrastructure in any way. The fault I was experiencing here appears to have been a fault in over-zealous ICMP filtering further along the network path between this web site and myself.

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.

About the Author

GEOFF HUSTON is the Chief Scientist at APNIC, the Regional Internet Registry serving the Asia Pacific region. he graduated from the Australian National University with a B.Sc, and M.Sc. in Computer Science. 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.

http://www.potaroo.net