December 2011
Geoff Huston

Dual-Stack Esotropia

The introduction of a second IP protocol into the Internet presents many technical issues, and in previous columns we've explored many of the issues related to network engineering and infrastructure. In this column I'd like to head upward in the protocol stack to the rarefied air way up there at the level of the application, and look at how applications are managing to cope with the issue of two IP stacks.

There is a proposal doing the rounds in the IETF for standardising dual stack client connection management called "**Happy Eyeballs**" [draft-wing-v6ops-happy-eyeballs-ipv6-01]. It proposes taking the essentially sequential behaviour of current implementations, where they probe for connectivity in one protocol and then the other, and attempting to perform the initial connection (DNS name resolution and TCP connection handshake) in each protocol in parallel, and then proceeding with the fastest connection. In this article I'll look at how browsers on dual stack client systems access dual stack servers, and examine the corner cases when one protocol or the other isn't quite working properly, and how the browser attempts to recover from the error. There is an excellent article on this topic by Emile Aben that was published on the RIPE LABs site ("Hampering Eyeballs" - https://labs.ripe.net/Members/emileaben/hampered-eyeballs), and here I'd like to build on his work by taking a slightly deeper investigation of the issue and look at how a number of popular web browsers and operating systems cope with accessing dual stack servers, and look in particular at what happens when things don't go as smoothly as one would hope.

"Conventional" Dual Stack Connectivity

A conventional approach to the dual stack environment can be seen if I dust off the cobwebs of my old Windows XP implementation and turn on IPv6.

If I use a simple custom TCP application (there are a number of good resources for coding up a TCP connection, including http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html#getaddrinfo) that performs a conventional connect using the operating system interface then I see a normal Windows XP behaviour of protocol preference in a dual stack environment.

```
OS: Windows XP 5.1.2600 Service Pack 3
Connection: tcpopen foo.rd.td.h.labs.apnic.net

     Time     Packet Activity

        0     → DNS Query for AAAA record foo.rd.td.h.labs.apnic.net
      581       ← AAAA response 2a01:4f8:140:50c5::69:72
        4     → DNS Query for A record for foo.rd.td.h.labs.apnic.net
      299       ← A response 88.198.69.81
        3     → SYN to 2a01:4f8:140:50c5::69:72
      280       ← SYN + ACK response from 2a01:4f8:140:50c5::69:72
        1     → ACK to 2a01:4f8:140:50c5::69:72
   ------
     1168
```

The time is measured in milliseconds, and the above log shows the elapsed time between each packet event that was generated in attempting to complete the connection. What this shows is a 581ms delay between issuing the query to the DNS for the `AAAA` resource record for the domain name `foo.rd.td.h.labs.apnic.net`, and a 4 millisecond delay after receiving the response before issuing a DNS query for the `A` resource record for the same domain name. This second query takes 299ms to obtain a response. After a further 3ms the application has generated an initial TCP connection packet (a SYN packet) to the IPv6 address. After one Round Trip Time (RTT) interval of 280ms there is a SYN+ACK packet received in response. My local application sends an ACK packet after a further millisecond and the connection has been established, with a total elapsed time of 1.168 seconds. All these packet traces are performed at the client side of the connection.

What I am interested in here is the time taken from the start of the connection operation, which is when the DNS queries are fired off to the time when, from the client's perspective, TCP is able to complete the initial three-way handshake. In the above example this connection takes some 1.2 seconds, where some 0.9 seconds is spent performing name resolution using the DNS and 0.3 seconds (or one round trip time) is spent performing the TCP connection setup.

The TCP three-way handshake is a protocol exchange that precedes every TCP connection. The initiator sends an initial TCP packet that has the SYN flag set. The "other end" will respond with a TCP packet that has the SYN and ACK flags set. When the initiator receives the SYN+ACK response it will send an ACK to the "other end" and set the local status of the connection to *connected*. At this point it is possible for the initiator to send data. The "other end" will enter the *connected* state when it receives the ACK packet.

Deliberately, with all these examples in this article the server has been placed some distance away from the client, with an RTT of around 296ms for IPv4 and a slightly faster 280ms RTT for IPv6.

What is going on here is that the operating system has established there are "native" IPv4 and IPv6 interfaces on the clients device, and when it attempts to perform a connection using a domain name, it will query the DNS for both IPv6 and IPv4 addresses .

Even in this first step of the connection, it's pretty clear that this could be slightly faster if Windows XP generated the two DNS queries in parallel. As it stands, Windows XP appears to wait for the completion of the first DNS query before generating the second query.

Actually, as with many things in the DNS, its not so clear that serializing the DNS requests would be all that much faster. To illustrate this, in the example above to resolve the DNS name `foo.rd.td.h.labs.apnic.net`, the local DNS agent asks its DNS forwarding resolver, who in turn, if this was a clean state start of the forwarding resolver would first ask the root name servers for the `NS` records for `.net`, then it ask a `.net` name server for the name servers for `.apnic`, and so on. It is more likely that the first few steps are not required as these NS records are cached by the local DNS forwarding resolver, and wherever possible, the local cache is used rather than performing a new query on an authoritative name server. Now when there are two back-to-back queries, such as in this case where there is a second query for a different resource record associated with the same DNS name, then the local resolver will not repeat the set of `NS` record queries, as these are now held in the local cache, and it will simply ask the final authoritative server for the resource record being queried.

This is why the first query took 581ms, while the second query was performed in a single round trip time of 299ms. If the two queries were to be performed in parallel the second query would not be able to take advantage of the entire cached state that is collected from the first query, and both queries would likely take a comparable time. In this case I estimate that parallelising the requests would shave off some 0.3 seconds from the total connection time.

When it has collected the DNS responses, the client then consults its local preference table to determine which protocol to use for the initial connection attempt. As the Windows XP system uses the default set of protocol preferences, it prefers to use IPv6 first, and the connection proceeds using IPv6.

Broken Dual Stack Connectivity

The question I'm interested in here is what happens when the preferred protocol is broken? To test this I'll use a slightly different domain name. This domain name is also a dual protocol domain name with both IPv6 and IPv4 addresses, but now the IPv6 address is deliberately unreachable. Lets see how Windows XP and this simple `tcpopen` application handles this case:

```
OS: Windows XP 5.1.2600 Service Pack 3
Connection: tcpopen foo.rx.td.h.labs.apnic.net

     Time     Activity

        0     → DNS AAAA? foo.rx.td.h.labs.apnic.net
      581       ← AAAA 2a01:4f8:140:50c5::69:72
        4     → DNS A? foo.rx.td.h.labs.apnic.net
      299         ← A 88.198.69.81
        3     → SYN 2a01:4f8:140:50c5::69:dead
     3000     → SYN 2a01:4f8:140:50c5::69:dead
     6000     → SYN 2a01:4f8:140:50c5::69:dead
    12000     → SYN 88.198.69.81
      298       ← SYN+ACK 88.198.69.81
        0     → ACK 88.198.69.81
 --------
    22185
```

What the end user experiences here is a 22 second pause while the local system tries three times to connect using IPv6 by resending the initial TCP SYN packet and waiting. The connection attempts are spaced using an exponential increasing series of backoff timers of 3, 6 and 12 seconds. At the expiration of the third backoff interval the system then switches over to the other protocol and undertakes a connection attempt using IPv4.

In many complex systems having two ways to achieve the objective can be leveraged to make the system more robust and faster, but in this case the serialisation works against the user, and when the preferred connection path is not working, the additional 21 second connection delay makes the user experience a far worse service than if the server was just an IPv4-only service. Windows XP use of a serial view of managing connectivity in a dual stack world is not exactly working to produce a better outcome for the user in such cases.

I also have a Mac OS X system and performed the same connection test. In this case I used a domain name that is a variant of the broken IPv6 test, where the domain name now has three non-responsive IPv6 addresses and a single working IPv4 address. The profile of the connection attempt is as follows:

```
OS: Mac OS X 10.7.2
Connection: tcpopen foo.rxxx.td.h.labs.apnic.net


     Time     Activity

        0     → DNS AAAA? foo.rxxx.td.h.labs.apnic.net
        4     → DNS A? foo.rxxx.td.h.labs.apnic.net
        230        ← DNS AAAA   2a01:4f8:140:50c5::69:dead
                                2a01:4f8:140:50c5::69:deae
                                2a01:4f8:140:50c5::69:deaf
         20        ← A response   88.198.69.81
          3       → SYN 2a01:4f8:140:50c5::69:dead (1)
        980       → SYN 2a01:4f8:140:50c5::69:dead (2)
       1013       → SYN 2a01:4f8:140:50c5::69:dead (3)
       1002       → SYN 2a01:4f8:140:50c5::69:dead (4)
       1008       → SYN 2a01:4f8:140:50c5::69:dead (5)
       1103       → SYN 2a01:4f8:140:50c5::69:dead (6)
       2013       → SYN 2a01:4f8:140:50c5::69:dead (7)
       4038       → SYN 2a01:4f8:140:50c5::69:dead (8)
       8062       → SYN 2a01:4f8:140:50c5::69:dead (9)
      16091       → SYN 2a01:4f8:140:50c5::69:dead (10)
      32203       → SYN 2a01:4f8:140:50c5::69:dead (11)
       8031       → SYN 2a01:4f8:140:50c5::69:deae (repeat sequence of 11 SYNs)
      75124       → SYN 2a01:4f8:140:50c5::69:deaf (repeat sequence of 11 SYNs)
      75213       → SYN 88.198.69.81
        297        ← SYN+ACK 88.198.69.81
          0       → ACK 88.198.69.81
   --------
     226435
```

This is admittedly a pathologically broken example, but it illustrates two major differences in the default behaviour of Mac OS X and Windows XP. The first is that Mac OS X has parallelized the DNS resolution behaviour, rather than performing the DNS queries sequentially. The second difference is that Windows XP uses three connection probes over 21 seconds to conclude that an address is unreachable, while Mac OS X uses 11 connection probes over 75 seconds to reach the same conclusion.

There is a third difference, which is not explicitly illustrated is the comparison of these two tests, namely that when there are multiple IPv6 addresses, Windows XP will try to connect using only one IPv6 address before failing over to IPv4, while Mac OS X will probe every IPv6 address using a 75 second probe sequence before flipping over protocol stacks to try IPv4. In the case where IPv6 is non-functional this can cause excessive delay in connection attempts, such as the 226 seconds encountered here.

Here Max OS X Lion does not seem to do any better than Windows XP, and the 75 seconds taken for each TCP connection attempt, as compared to the Windows XP setting of 21 seconds, makes the user experience far worse when all is not working perfectly in the underlying network.

This was a very simple TCP application, and it has been observed that web browsers often customise their approach in order to strike a good balance between robustness and responsiveness. So lets look at some popular browsers and examine the way in which they handle these situations where the remote dual stack server is impaired in some manner.

Dual Stack Behaviour on Mac OS X

Here I'm using a system running the latest version of the MAC OS X operating system, Lion, which is at version 10.7.2. I'll test the most popular web browsers for the Mac, namely Safari, Firefox, Opera, and Chrome, and look at how they handle connections in the Dual Stack world where one protocol is just not responding.

Safari on Mac OS X

The first browser case I'll look at here is the Safari browser on Mac OS X. Firstly, we connect to a URL where there is a dual stack server with responsive IPv4 and IPv6 addresses.

```
OS: Mac OS X 10.7.2
Browser: Safari: 5.1.1

   URL: www.rd.td.h.labs.apnic.net
```

```
 Time     Activity

          IPv4                          IPv6

   0      → DNS A? www.rd.td.h.labs.apnic.net
   1                                    → DNS  AAAA? www.rd.td.h.labs.apnic.net
 333                                      ← AAAA 2a01:4f8:140:50c5::69:72
   5        ← A 88.198.69.81
   1      → SYN 88.198.69.81
 270                                    → SYN 2a01:4f8:140:50c5::69:72
  28      ← SYN+ACK 88.198.69.81
   0      → ACK 88.198.69.81
   1      → [start HTTP session]
 251                                      ← SYN+ACK 2a01:4f8:140:50c5::69:72
   0                                    → RST 2a01:4f8:140:50c5::69:72
-----
 638      (time to connect)
```

Safari appears to be setting off two parallel connectivity sequences, where there is an IPv4 DNS query and TCP connection, and a parallel IPv6 DNS query and a TCP connection.

It is evident that the first session to provide a completed TCP handshake is used by the browser, while the other connection is terminated with a TCP RST (reset) as soon as it responds with a SYN+ACK.

What is not so clear from this trace is why the Safari system chose to fire off the first SYN in IPv4, even though the DNS response with the IPv6 address arrived first. What is also unclear is why Safari chose to wait for some 270ms between receiving the DNS response to the AAAA query and starting the IPv6 TCP connection. It appears that the system is using a table of RTT intervals associated with each address it has connected to in the recent past and also a "default" RTT for other addresses it has not encountered previously, and it fires off the first connection to the address that has the lowest RTT value. The failover time may also be managed from this RTT estimate, and in this case it may be that the system has an internal estimate that the expected connection time for this IPv4 address is 270ms, and when this time expires the system attempts to connect to the next address, which happens to be the IPv6 address, while leaving the first connection attempt open in case it responds.

Once it has a working connection it politely resets the other connection.

A repeat of the test a few minutes after the one above shows a similar behaviour, but with the other protocol being selected for the initial connection attempt:

```
 URL: www.rd.td.h.labs.apnic.net

 Time     Activity

          IPv4                          IPv6

   0      → DNS A? www.rd.td.h.labs.apnic.net
   0                                    → DNS  AAAA? www.rd.td.h.labs.apnic.net
 330                                      ← AAAA 2a01:4f8:140:50c5::69:72
   0        ← A 88.198.69.81
   1                                    → SYN 2a01:4f8:140:50c5::69:72
 120      → SYN 88.198.69.81
 160                                      ← SYN+ACK 2a01:4f8:140:50c5::69:72
   0                                    → ACK 2a01:4f8:140:50c5::69:72
   1                                    → [start HTTP session]
 136        ← SYN+ACK 88.198.69.81
   0      → RST 88.198.69.81
-----
 611      (time to connect)
```

In this case the two DNS responses were received in the same millisecond interval, and once again the IPv6 AAAA record response arrived slightly ahead of the A record response. In this case Safari selected the IPv6 address. Again I am guessing here, but it may be that the system now has a cached value of the IPv4 RTT time of 299ms, and it may have a cached value of the IPv6 RTT of 280ms, or it could be using some lower default estimate of the IPv6 RTT. It appears that Safari is using a sort function on the returned addresses based on some estimated RTT, and selecting the lowest RTT address to try first.

Here there was a delay of 120ms between the two SYN packets, as compared to a wait of 270ms in the previous experiment. This appears to be derived from some revised internal estimate of the expected RTT to the IPv6 destination address, which in this case is unduly optimistic as the actual RTT is 280ms.

I have to comment that, overall, this algorithm is fast! The RTT between the client and the server is 280ms in IPv6 and 299ms in IPv4 and the absolute minimum possible connection time is 579ms assuming a single IPv4 DNS RTT and a single IPv6 connection RTT. Safari's 611ms connection time is extremely efficient.

So that's what happens when everything is working. What happens when Safari is presented with a URL that has a some unresponsive addresses? This second test described here attempts to connect to `www.rxxx.td.h.labs.apnic.net`, which is configured with three unresponsive IPv6 addresses and a single responsive IPv4 address.

```
URL: www.rxxx.td.h.labs.apnic.net

Time     Activity

         IPv4                          IPv6

   0     → DNS A? www.rxxx.td.h.labs.apnic.net
   0                                   → DNS  AAAA? www.rxxx.td.h.labs.apnic.net
 299                                     ← AAAA  2a01:4f8:140:50c5::69:dead
                                                 2a01:4f8:140:50c5::69:deae
                                                 2a01:4f8:140:50c5::69:deaf
   2                                   → SYN 2a01:4f8:140:50c5::69:dead
   0       ← A 88.198.69.81
 270                                   → SYN 2a01:4f8:140:50c5::69:deae
 120                                   → SYN 2a01:4f8:140:50c5::69:deaf
 305     → SYN 88.198.69.81
 300       ← SYN+ACK 88.198.69.81
   0     → ACK 88.198.69.81
   1     → [start HTTP session]
-----
1297
```

In this test the `AAAA` query returned first, and Safari immediately attempted to open a session using the first IPv6 address. Safari now appears to have a default estimate of 270ms for this connection attempt, and upon its expiry it then tries the second IPv6 address, even though the A record response has been received from the DNS in the intervening period. Some 120ms later Safari attempts to connect using the third unresponsive address. A further 300ms later it then tries to connect using the IPv4 address. The total effort takes 1.3 seconds, and each address is tried only once in Safari.

The behaviour of Safari in OS X Lion is described in a post from an Apple engineer in July of this year (http://lists.apple.com/archives/ipv6-dev/2011/Jul/msg00009.html). That note refers to the command `nettop -n -m route` to show the information that is maintained by OS X for each destination route, including the current RTT estimates that are maintained by the system.

Chrome on Mac OS X

Chrome has also been equipped with a version of the "happy eyeballs" Dual Stack connection algorithm, which uses short timers and a relatively fast failover from one protocol to the other.

```
OS: Mac OS X 10.7.2
Browser: Chrome 16.0.912.36
```

The first test is again a simple dual stack URL, where both IPv4 and IPv6 are responsive.

```
URL: www.rd.td.h.labs.apnic.net
```

```
  Time     Activity

           IPv4                         IPv6

    0      → DNS A? www.rd.td.h.labs.apnic.net
    0                                   → DNS  AAAA? www.rd.td.h.labs.apnic.net
  299        ← A 88.198.69.81
    1                                     ← AAAA  2a01:4f8:140:50c5::69:72
    1      → SYN 88.198.69.81 (port a)
    1      → SYN 88.198.69.81 (port b)
  250      → SYN 88.198.69.81 (port c)
   48        ← SYN+ACK 88.198.69.81 (port a)
    0      → ACK 88.198.69.81 (port a)
    0      → [start HTTP session (port a)]
-----
  600
```

Let's try that a second time using a subtly different domain name, but with the same dual stack behaviour:

```
  URL: xxx.rd.td.h.labs.apnic.net

  Time     Activity

           IPv4                         IPv6

    0      → DNS A? xxx.rd.td.h.labs.apnic.net
    0                                   → DNS  AAAA? xxx.rd.td.h.labs.apnic.net
  298                                     ← AAAA  2a01:4f8:140:50c5::69:72
    1        ← A 88.198.69.81
   10                                   → SYN 2a01:4f8:140:50c5::69:72 (a)
    0                                   → SYN 2a01:4f8:140:50c5::69:72 (b)
  250                                   → SYN 2a01:4f8:140:50c5::69:72 (c)
   28                                     ← SYN+ACK 2a01:4f8:140:50c5::69:72 (a)
    0                                   → ACK 2a01:4f8:140:50c5::69:72 (a)
    0                                   → [start HTTP session (a)]
-----
  587
```

What appears to be happening in Chrome is that whichever DNS response arrives first, Chrome appears to select as the preferred protocol and attempts to connect by sending TCP SYN packets. Unlike Safari, Chrome attempts to connect using 2 ports in parallel, and after a 250ms delay, if there has been no SYN ACK in response, it will send a further SYN on a third port. Compared to the ideal connection time of 579ms, a connection of 587ms for IPv6 is extremely efficient, and is the fastest I've seen. It's very quick!

Now lets look at what happens when the IPv6 address is unresponsive:

```
  URL: xxx.rx.td.h.labs.apnic.net

  Time     Activity

           IPv4                         IPv6

    0      → DNS A? xxx.rx.td.h.labs.apnic.net
    0                                   → DNS  AAAA? xxx.rx.td.h.labs.apnic.net
  298                                     ← AAAA  2a01:4f8:140:50c5::69:dead
    0        ← A 88.198.69.81
   11                                   → SYN 2a01:4f8:140:50c5::69:dead (a)
    0                                   → SYN 2a01:4f8:140:50c5::69:dead (b)
  250                                   → SYN 2a01:4f8:140:50c5::69:dead (c)
   51      → SYN 88.198.69.81 (d)
    1      → SYN 88.198.69.81 (e)
  250      → SYN 88.198.69.81 (f)
   48        ← SYN+ACK 88.198.69.81 (d)
    0      → ACK 88.198.69.81 (d)
    0      → [start HTTP session (d)]
-----
  909
```

There is a 300ms connection timer that is evidently started upon sending the first SYN, and when this timer expires Chrome then repeats the sequence of port opening in the other protocol if the first set of open calls have been unresponsive. This way, within 300ms Chrome is able to switch back from an unresponsive IPv6 address to IPv4 address and hopefully complete the connection quickly.

When the URL has two or more IPv6 addresses Chrome appears to test only one of the IPv6 addresses, again using 3 SYNs, and then when the 300ms timer expires it flips protocols and tests the IPv4 address. Will it flip back after a further 300ms if there are other untested IPv6 addresses? The following test uses Chrome with a subtle variant of the test, where there are two IPv6 addresses, one responsive and one unresponsive, and just one unresponsive IPv4 address. I also constrained the order of the DNS responses such that the unresponsive IPv6 is first in the list of AAAA addresses returned by the DNS:

```
      URL: www.rxz6.td.h.labs.apnic.net

      Time     Activity

               IPv4                        IPv6

        0      → DNS A? www.rxz6.td.h.labs.apnic.net
        0                                  → DNS  AAAA? www.rxz6.td.h.labs.apnic.net
      298                                    ← AAAA  2a01:4f8:140:50c5::69:dead
                                                     2a01:4f8:140:50c5::69:72
        0        ← A 203.133.248.95
       11                                  → SYN 2a01:4f8:140:50c5::69:dead (a)
        0                                  → SYN 2a01:4f8:140:50c5::69:dead (b)
      250                                  → SYN 2a01:4f8:140:50c5::69:dead (c)
       51      → SYN 203.133.248.95 (d)
        1      → SYN 203.133.248.95 (e)
      250      → SYN 203.133.248.95 (f)

       +0.5s   → SYN 203.133.248.95 (d,e,f)
                                           → SYN 2a01:4f8:140:50c5::69:dead (a,b,c)

       +1s     repeat of 6 syn packets (3)
       +1s     repeat of 6 syn packets (4)
       +1s     repeat of 6 syn packets (5)
       +1s     repeat of 6 syn packets (6)
       +2s     repeat of 6 syn packets (7)
       +4s     repeat of 6 syn packets (8)
       +8s     repeat of 6 syn packets (9)
      +16s     repeat of 6 syn packets (10)
      +32s     repeat of 6 syn packets (11)

       +7s                                 → SYN 2a01:4f8:140:50c5::69:72 (g)
        0                                  → SYN 2a01:4f8:140:50c5::69:72 (h)
      200                                  → SYN 2a01:4f8:140:50c5::69:72 (i)
       80                                    ← SYN+ACK 2a01:4f8:140:50c5::69:72 (g)
        0                                  → ACK 2a01:4f8:140:50c5::69:72 (g)
        0                                  → [start HTTP session (g)]
    -----
       76.5 seconds
```

In this case the total delay to establish the connection is 76.5 seconds, and it included sending 69 SYN packets! A similar 76 second delay is evident when the browser is presented with unresponsive IPv4 and IPv6 addresses coupled with a responding IPv4 address.

Obviously this is an extensive delay, and it would be useful to understand where this delay is coming from.

There is a TCP control variable, `net.inet.tcp.keepinit` whose value on my system is set to `75000`. In the source code of the TCP driver for Mac OS X in the code in definition module `tcp_timer.h` there is the declaration of the structure of :

```
  int tcp_syn_backoff[TCP_MAXRXTSHIFT + 1]={1, 1, 1, 1, 1, 2, 4, 8, 16, 32, 64, 64, 64};
```

What appears to be the case is that there is a standard construct in the Mac OS X operating system (and in the FreeBSD operating system, upon which Max OS X has been constructed) that paces out the SYN retransmits according to the `tcp_syn_backoff` array, to a maximum interval of the value of `net.inet.tcp.keepinit` milliseconds.

When Chrome initiates a TCP connection, it leaves it open for the full length of time defined as the operating system default. The implementation appears to be:

1. Initiate a DNS query for A and AAAA values
2. For the first protocol to respond, attempt to open two ports
3. If no response in 250ms, attempt to open a third port
4. After a further 50ms open two connections in the other protocol
5. If no response in a further 250ms, open a third port in the other protocol
6. If the connection attempt fails (using the host operating system's SYN retry defaults), use alternate addresses and retry the connection procedure from step 2

Chrome is fast when there is a problem with one protocol, and will take no more than an additional 300ms to connect in the case where its initial connection attempt uses the unresponsive protocol. However, in more complex cases where there are unresponsive addresses in both protocols Chrome's use of the host operating system defaults when performing the connection implies that the connection time for these complex cases can take more than 75 seconds on MAC OS X.

Firefox on Mac OS X

How does the latest version of Firefox fare under similar conditions?

```
OS: Mac OS X 10.7.2
Browser: Firefox 8.0

   URL: www.rxxx.td.h.labs.apnic.net

   Time     Activity

            IPv4                        IPv6

      0     → DNS A? www.rxxx.td.h.labs.apnic.net
      0                                 → DNS  AAAA? www.rxxx.td.h.labs.apnic.net
    298                                   ← AAAA  2a01:4f8:140:50c5::69:dead
                                                  2a01:4f8:140:50c5::69:deae
                                                  2a01:4f8:140:50c5::69:deaf
      0       ← A 88.198.69.81
      1                                 → SYN 2a01:4f8:140:50c5::69:dead (a)
      0                                 → SYN 2a01:4f8:140:50c5::69:dead (b)
   1120                                 → SYN 2a01:4f8:140:50c5::69:dead (a)
      0                                 → SYN 2a01:4f8:140:50c5::69:dead (b)
    +1s     repeat of 2 syn packets (3)
    +1s     repeat of 2 syn packets (4)
    +1s     repeat of 2 syn packets (5)
    +1s     repeat of 2 syn packets (6)
    +2s     repeat of 2 syn packets (7)
    +4s     repeat of 2 syn packets (8)
    +8s     repeat of 2 syn packets (9)
   +16s     repeat of 2 syn packets (10)
   +32s     repeat of 2 syn packets (11)
    +7s                                 → SYN 2a01:4f8:140:50c5::69:deae (c)
      0                                 → SYN 2a01:4f8:140:50c5::69:deae (d)
   +75s     10 repeats of 2 x SYNs to 2a01:4f8:140:50c5::69:deae
   +75s     11 repeats of 2 x SYNs to 2a01:4f8:140:50c5::69:deaf (e,f)
            → SYN 88.198.69.81 (g)
            → SYN 88.198.69.81 (h)
    298       ← SYN+ACK 88.198.69.81 (g)
      0     → ACK 88.198.69.81 (g)
      0     → [start HTTP session (g)]
  -----
    226.5 seconds
```

Admittedly, this is a deliberately perverse example, where there are three unresponsive IPv6 addresses and one responsive IPv4 address behind this particular URL, but it shows the sequential nature of address processing in Firefox 8.0, where all the IPv6 addresses are probed for `net.inet.tcp.keepinit` milliseconds (which is set to 75 seconds in my case) before shifting over to probe the IPv4 address. In this case the connection time is an impressive 226 seconds.

The other difference between Firefox and Chrome is that Firefox starts the connection with two parallel connections, and does not open up a third connection.

There is an additional tuning parameter in Firefox 8.0, where, using the `about:config` setting, the parameter `network.http.fast-fallback-to-IPv4` can be set to `true`.

```
URL: www.rxxx.td.h.labs.apnic.net

Time     Activity

         IPv4                         IPv6

   0     → DNS A? www.rxxx.td.h.labs.apnic.net
   1                                  → DNS  AAAA? www.rxxx.td.h.labs.apnic.net
 320       ← A 88.198.69.81
   1     → SYN 88.198.69.81 (a)
   2                                    ← AAAA  2a01:4f8:140:50c5::69:deaf
                                                2a01:4f8:140:50c5::69:dead
                                                2a01:4f8:140:50c5::69:deae
   1                                  → SYN 2a01:4f8:140:50c5::69:deaf (b)
 296       ← SYN+ACK 88.198.69.81 (a)
   0     → ACK 88.198.69.81 (a)
   0     → [start HTTP session (a)]
1200                                  → SYN 2a01:4f8:140:50c5::69:deaf (b)
1100                                  → SYN 2a01:4f8:140:50c5::69:deaf (b)
1100                                  → SYN 2a01:4f8:140:50c5::69:deaf (b)
1100                                  → SYN 2a01:4f8:140:50c5::69:deaf (b)
1100                                  → SYN 2a01:4f8:140:50c5::69:deaf (b)
2100                                  → SYN 2a01:4f8:140:50c5::69:deaf (b)
4200                                  → SYN 2a01:4f8:140:50c5::69:deaf (b)
4000     → [close HTTP session (a)]
4100                                  → SYN 2a01:4f8:140:50c5::69:deaf (b)
16100                                 → SYN 2a01:4f8:140:50c5::69:deaf (b)
32100                                 → SYN 2a01:4f8:140:50c5::69:deaf (b)
7000                                  → SYN 2a01:4f8:140:50c5::69:deaf (b)
 +75s                                 → 11 x SYN to 2a01:4f8:140:50c5::69:dead
 +75s                                 → 11 x SYN to 2a01:4f8:140:50c5::69:deaf

             Now something strange happens - Firefox opens an IPv4 session, even
             though the web page has been loaded and completed some 225 seconds
             ago!

   0     → SYN 88.198.69.81 (f)
 296       ← SYN+ACK 88.198.69.81 (f)
   0     → ACK 88.198.69.81 (f)

             And now something equally strange happens. The Linux server at the
             remote end responds to 3 seconds of inactivity by resending the
             SYN+ACK packet

3000       ← SYN+ACK 88.198.69.81 (f)
   0     → ACK 88.198.69.81 (f)

             And two seconds later, this gratuitous session is closed:

2000     → FIN+ACK 88.198.69.81 (f)
 298       ← FIN+ACK 88.198.69.81 (f)
   0     → ACK 88.198.69.81 (f)
------
231515
```

This setting causes Firefox to dispense with the efforts to open two ports in parallel, and instead it looks a lot like the "happy eyeballs" behaviour, where both TCP sessions are opened as soon as there is a DNS response within the context of each protocol. With this setting set to `true` Firefox is equally quick in a standard dual stack environment.

However there appears to be a bug in the implementation where even when one protocol completes the web transaction the other protocol continues its connection attempt to exhaustion, including the testing of all addresses in the case that the addresses are unresponsive. I was expecting to see the local client shut down the connection attempt on the outstanding connection and send a TCP reset (RST) if the connection has already elicited a SYN+ACK response.

There is a second bug visible here at the end of the connection attempts for an unresponsive protocol. As the packet trace illustrates, Firefox switches over and connects using the other protocol, even though there is no outstanding HTTP task. Once the connection is made, Firefox then appears to have a 5 second timeout and then closes the session and presumably completes the internal task of opening up the second protocol.

This packet trace example also highlights what I would interpret as another implementation bug, this time in the TCP implementation in the Linux server, where a TCP session that is opened but has no traffic will, after 3 seconds in this idle state, send a gratuitous SYN+ACK packet.

Opera on Mac OSX

The last of the browsers to be tested here is the Opera browser.

```
OS: Mac OS X 10.7.2
Browser: Opera 11.52
```

Opera appears to try the first IPv6 address with a single port, and if this is unresponsive after the default TCP SYN timeout (75 seconds in my case) then it will try the next IPv6 address, if there is another IPv6 address, or switch to IPv4 and try the first IPv4 address.

To expose this connection behaviour I'll use a URL that has an unresponsive IPv6 address and two IPv4 addresses, one unresponsive and one reachable address.

```
URL: xxx.rxz4.td.h.labs.apnic.net


Time     Activity

         IPv4                        IPv6

   0     → DNS A? xxx.rxz4.td.h.labs.apnic.net
   0                                 → DNS  AAAA? xxx.rxz4.td.h.labs.apnic.net
 298                                   ← AAAA  2a01:4f8:140:50c5::69:dead
   0       ← A 203.133.248.95
               88.198.69.81
  51                                 → SYN 2a01:4f8:140:50c5::69:dead (a)
1100                                 → SYN 2a01:4f8:140:50c5::69:dead (a)
1100                                 → SYN 2a01:4f8:140:50c5::69:dead (a)
1100                                 → SYN 2a01:4f8:140:50c5::69:dead (a)
1100                                 → SYN 2a01:4f8:140:50c5::69:dead (a)
1100                                 → SYN 2a01:4f8:140:50c5::69:dead (a)
2100                                 → SYN 2a01:4f8:140:50c5::69:dead (a)
4200                                 → SYN 2a01:4f8:140:50c5::69:dead (a)
8200                                 → SYN 2a01:4f8:140:50c5::69:dead (a)
16100                                → SYN 2a01:4f8:140:50c5::69:dead (a)
32100                                → SYN 2a01:4f8:140:50c5::69:dead (a)
6900     → SYN 203.133.248.95 (b)
1100     → SYN 203.133.248.95 (b)
1100     → SYN 203.133.248.95 (b)
1100     → SYN 203.133.248.95 (b)
1100     → SYN 203.133.248.95 (b)
1100     → SYN 203.133.248.95 (b)
2100     → SYN 203.133.248.95 (b)
4200     → SYN 203.133.248.95 (b)
8200     → SYN 203.133.248.95 (b)
16100    → SYN 203.133.248.95 (b)
32100    → SYN 203.133.248.95 (b)
6900     → SYN 203.133.248.95 (c)
1100     → SYN 203.133.248.95 (c)
```

```
  1100     → SYN 203.133.248.95 (c)
  1100     → SYN 203.133.248.95 (c)
 -----
 15549
```

What was unexpected to see here were the four final packets, which appear to be Opera initiating a third connection attempt, repeating the previous address, but truncating this after a further 3.3 seconds. What appears to be happening is that Opera has a connection regime that selects only one address from each protocol, and it invokes a conventional connection with these addresses, preferring IPv6 over IPv4. Interestingly, when the IPv4 connection attempt returns failure (after a total elapsed time of some 152 seconds since the start of this entire connection process) Opera then invokes a third connection attempt, reusing the failed IPv4 address! I am not sure why it does not try the second (and in this case working) IPv4 address at this point. Then Opera appears to abort the entire connection process after a further 3 seconds (and four SYN packets). I guess that Opera has a "master" timer of 155 seconds, and if no progress has been made in establishing a connection after 155 seconds it simply terminates the current connection and reports failure to the user.

Dual Stack Behaviour on Windows 7

In Windows 7 the operating system has a default bias to prefer IPv6 in its local preference tables.

For Windows 7 the preference between IPv4 and IPv6 is set using flag values in the registry entry:
`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\tcpip6\Parameters\DisabledComponents`

For Windows XP this preference table is set using the command: `netsh interface ipv6 set prefixpolicy`

Windows also uses a SYN retransmit timer, where a connection attempt takes up to 21 seconds, using 3 SYN probes, with a backoff timer of 3, 6 and 12 seconds following each SYN probe.

The options for browsers are evidently limited here – either the application can use the operating system default behaviour and experience a 20 second failover to IPv4 when the IPv6 address is unresponsive, or it can fork a second connection context in IPv4 without waiting for the initial connection to timeout. Almost all browsers tested use the first approach, including Explorer 9, Firefox 8, Safari 5 and Opera 11.5. The exceptions are Chrome, which performs a parallel failover connection 300ms after the primary connection, and Firefox 8 with fast failover enabled, which performs a fully parallel connection in both protocols.

I have also performed the same set of tests on Windows XP, and the browsers behave in the same manner. It seems that while many aspects of the system have changed on the evolutionary path from Windows XP through Windows Vista to Windows 7, one of the invariants is the behaviour of the protocol stack on these systems, and the dual stack recovery behaviour is identical, as far as I can tell by these tests.

Lets look at a few of these browsers' connection behaviour in Windows 7 in a bit more detail.

Explorer on Windows 7

```
OS: Windows 7
Browser: Explorer 9.0.8.112.16421

   URL: xxx.rxxx.td.h.labs.apnic.net

  Time     Activity
```

```
             IPv4                           IPv6

     0       → DNS A? xxx.rxxx.td.h.labs.apnic.net
   335         ← A 88.198.69.81
     1                                      → DNS  AAAA? xxx.rxxx.td.h.labs.apnic.net
   325                                        ← AAAA  2a01:4f8:140:50c5::69:dead
     1                                      → SYN 2a01:4f8:140:50c5::69:dead (a)
     0                                      → SYN 2a01:4f8:140:50c5::69:dead (b)
  3000                                      → SYN 2a01:4f8:140:50c5::69:dead (a)
     0                                      → SYN 2a01:4f8:140:50c5::69:dead (b)
  6000                                      → SYN 2a01:4f8:140:50c5::69:dead (a)
     0                                      → SYN 2a01:4f8:140:50c5::69:dead (b)
 12000                                      → SYN 2a01:4f8:140:50c5::69:deaf (c)
     0                                      → SYN 2a01:4f8:140:50c5::69:deaf (d)
  3000                                      → SYN 2a01:4f8:140:50c5::69:deaf (c)
     0                                      → SYN 2a01:4f8:140:50c5::69:deaf (d)
  6000                                      → SYN 2a01:4f8:140:50c5::69:deaf (c)
     0                                      → SYN 2a01:4f8:140:50c5::69:deaf (d)
 12000                                      → SYN 2a01:4f8:140:50c5::69:deae (e)
     0                                      → SYN 2a01:4f8:140:50c5::69:deae (f)
  3000                                      → SYN 2a01:4f8:140:50c5::69:deaf (e)
     0                                      → SYN 2a01:4f8:140:50c5::69:deaf (f)
  6000                                      → SYN 2a01:4f8:140:50c5::69:deaf (e)
     0                                      → SYN 2a01:4f8:140:50c5::69:deaf (f)
 12000       → SYN 88.198.69.81 (g)
     0       → SYN 88.198.69.81 (h)
   296         ← SYN+ACK 88.198.69.81 (g)
     0       → ACK 88.198.69.81 (g)
 -----
 57956
```

The noted behaviour in Explorer is to attempt to connect on all available IPv6 addresses before failing over to IPv4. In this admittedly pathological bad case of 3 unresponsive IPv6 addresses the connection time is extended by 57 seconds. In a more conventional situation with a single unresponsive IPv6 address, Explorer will send 3 SYN probes to the IPv6 address with backoff timers of 3, 6 and 12 seconds, then switch over to probe IPv4.

Firefox on Windows 7

```
    OS: Windows 7
    Browser: Firefox 8.0.1

       URL: xxx.rx.td.h.labs.apnic.net

      Time     Activity

               IPv4                           IPv6

     0       → DNS A? xxx.rx.td.h.labs.apnic.net
   335         ← A 88.198.69.81
     1                                      → DNS  AAAA? xxx.rx.td.h.labs.apnic.net
   325                                        ← AAAA  2a01:4f8:140:50c5::69:dead
     5                                      → SYN 2a01:4f8:140:50c5::69:dead (a)
  3000                                      → SYN 2a01:4f8:140:50c5::69:dead (a)
     0                                      → SYN 2a01:4f8:140:50c5::69:dead (b)
  6000                                      → SYN 2a01:4f8:140:50c5::69:dead (a)
     0                                      → SYN 2a01:4f8:140:50c5::69:dead (b)
 12000       → SYN 88.198.69.81 (c)
     0       → SYN 88.198.69.81 (d)
   296         ← SYN+ACK 88.198.69.81 (c)
     0       → ACK 88.198.69.81 (c)
 -----
 19962
```

This is very similar to the behaviour of Explorer in so far as it serialises the DNS queries, then serialises the TCP connection attempts. There is one curious artefact here, and that is that the initial connection is made on one port, but after the first SYN timeout of three seconds it is joined by a second parallel port, and both connections then perform a 6 second SYN time out. This strikes me as some form of implementation bug, but whether it's Firefox or Windows at fault here is hard to tell.

When I enable Firefox's `network.http.fast-fallback-to-IPv4` the behaviour is switched from sequential to parallel for the TCP connection, but, curiously, the DNS name resolution remains a sequential operation.

```
URL: xxx.rx.td.h.labs.apnic.net

Time     Activity

         IPv4                         IPv6

   0     → DNS A? xxx.rx.td.h.labs.apnic.net
 334       ← A 88.198.69.81
   2                                  → DNS  AAAA? xxx.rx.td.h.labs.apnic.net
   1     → SYN 88.198.69.81 (a)
 325                                    ← AAAA  2a01:4f8:140:50c5::69:dead
   5                                  → SYN 2a01:4f8:140:50c5::69:dead (b)
   9       ← SYN+ACK 88.198.69.81 (a)
   0     → ACK 88.198.69.81 (a)
   0     [HTTP session]

                                      The IPv6 connection just keeps on trying
                                      even though the IPv4 connection has
                                      completed and the HTTP session is already
                                      underway

2991                                  → SYN 2a01:4f8:140:50c5::69:dead (b)
6000                                  → SYN 2a01:4f8:140:50c5::69:dead (b)

         this is the closure of the
         active Ipv4 session

7100     [FIN handshake]

         This is the strange part - its been 21 seconds since the start of the
         IPv6 connection attempt, and when it reports failure back to Firefox
         then Firefox reopens the IPv4 connection, even though the HTTP
         session has completed!

4900     → SYN 88.198.69.81 (c)
 296       ← SYN+ACK 88.198.69.81 (c)
   0     → ACK 88.198.69.81 (c)

         This is the Linux behaviour of sending a gratuitous SYN+ACK 3 seconds
         After connection startup on an idle connection

3200       ← SYN+ACK 88.198.69.81 (c)
   0     → ACK 88.198.69.81 (c)

         After five seconds Firefox closes off this extraneous session

1800     [FIN Handshake]
```

The session startup is performed in parallel, but curiously even when the IPv4 connection successfully completes, the IPv6 stream is still attempting to connect. More curiously, when the IPv6 connection fails, it then falls back to IPv4, and sits idle for 5 seconds before shutting down the session. Also visible here is the Linux server's spurious idle SYN+ACK that appears to be generated after 3 seconds of initial inactivity at the server. This "hanging" connection attempt is not visible to the user, and the user sees a rapid connection and a rapid download, and the "dangling" extra session just trails off in the background. But it would be good to see this obvious implementation bug cleaned up in a future release of this browser.

Chrome on Windows 7

```
OS: Windows 7
Browser: Chrome 15.0.874.121 m

   URL: xxx.rx.td.h.labs.apnic.net
   Time     Activity

            IPv4                        IPv6

     0      → DNS A? xxx.rx.td.h.labs.apnic.net
   333        ← A 88.198.69.81
     2                                  → DNS  AAAA? xxx.rx.td.h.labs.apnic.net
   325                                    ← AAAA  2a01:4f8:140:50c5::69:dead
    11                                  → SYN 2a01:4f8:140:50c5::69:dead (a)
     0                                  → SYN 2a01:4f8:140:50c5::69:dead (b)
   250                                  → SYN 2a01:4f8:140:50c5::69:dead (c)
    50      → SYN 88.198.69.81 (d)
     0      → SYN 88.198.69.81 (e)
   250      → SYN 88.198.69.81 (f)
    48        ← SYN+ACK 88.198.69.81 (d)
     0      → ACK 88.198.69.81 (d)
     0      → [start HTTP session (d)]
 -----
  1599
```

As with Mac OS X, Chrome in Windows 7 launches 2 connection attempts in IPv6 immediately, and opens a third port after 250ms of inactivity. When 300ms expires Chrome then switches over to IPv4 and opens two ports immediately, and a third port in another 250ms.

Dual Stack Behaviour on Linux

The default behaviour for Linux 2.6 is to attempt to connect using 5 SYN probes, as compared to 11 SYN probes taking 75 seconds with FreeBSD (and Mac OS X) and 3 probes taking 21 seconds with Windows. With Linux the wait time for each probe is, like Windows, an exponentially increasing set of times, starting at 3 seconds.

The default SYN probe sequence used by Linux is:

```
OS: Linux 2.6

   Time     Activity                    Elapsed Time

     0      → SYN
    3s      → SYN - retry 1               3s
    6s      → SYN - retry 2               9s
   12s      → SYN - retry 3              21s
   24s      → SYN - retry 4              45s
   48s      → SYN - retry 5              93s
   96s      return connection failure   189s
```

This 3 minute failover period is excessive these days, and can be altered by setting the sysctl parameter `net.ipv4.tcp_syn_retries` to a lower value. It is also possible for the application itself to override this by using a local timer and terminating the connection attempt when the application level timer expires.

Why are the default timeout settings for TCP connection so long?

These days it takes some 350ms for a packet to pass around the entire globe on today's terrestrial fibre networks. Indeed searching for a network path that takes longer than 400ms using entirely terrestrial routes is a challenge and when it occurs it's often symptomatic of a routing failure rather than the outcome of deliberate network engineering. So why do we see default settings in shipped operating systems with connection timers of 21 seconds, 75 seconds and even 189 seconds? In that 189 seconds that same erstwhile packet could've whizzed around the earth 540 times! Surely its possible to conclude that an address is unresponsive in 1 or 2 seconds, so why impose a 3 minute wait upon the long suffering user?

The challenge for protocol stack engineers is to design a system that is robust under all conditions, and that means that it's necessary for the stack to work reliably under some of the more extreme and challenging situations. Oddly enough one of the most challenging situations is a dialup modem. The desire for useable speed means that dialup modems attempt to provide extreme payload compression as well as performing the highest level of signal compression it can. The combination can be used to extract a phenomenal 56Kbps out of a 300 baud analogue circuit, but the cost is in imposed delay, and it is not uncommon to see a dialup connection adding some 3 seconds to a round trip measurement. In addition, even 56Kbps of bandwidth is nowhere near enough, and many applications will swamp the connection with traffic, causing instances of sustained packet loss. So when you are engineering a system to be robust over a lossy high delay dialup connection, then an algorithm of 11 probes over 75 seconds starts to sound like a decent strategy, and even 6 probes over 189 seconds may well outlast any transient congestion event on a dialup connection.

So where to set the default? Should the default retry behaviour be set to optmize the best possible case, in the certain knowledge that if the system is ever placed behind a dialup modem then it simply will not work if there is any form of extended delay or loss? Or should the default be set to try to elicit a response in the worst case, in the knowledge that most of the network is highly reliable most of the time, and on a high speed connection these default connection failure retry timers will be used rarely? Most protocol engineers design their default for the latter, and hope that most users will simply not encounter these massive timeouts in their normal use of the network.

I have been waiting for the time when these protocol stacks employ a little more active probing of their environment and adapt their connection timers based on recent experience.

The recent changes to the MAC OS X protocol stack, where the estimated RTT times for each destination are cached and used to optimise the system's connection behaviour are truly a delight to see. Yes it probably still needs a little tuning as far as I can tell, but it is a major evolutionary step from the rather primitive blind guesswork that we've used so far in designing protocol behaviour. It appears to make the entire Dual Stack scenario about as painless as possible, even in some of the more pathological cases of partial connectivity.

I won't reproduce the complete set of packet traces for the browsers tested on Linux, but note that Firefox 8.0.1 uses a truncated connection sequence that terminates after 4 retries (a total of 5 probes) and 93 seconds, and when fast failover is enabled, performs a fully parallel connection sequence in each protocol. And Chrome again uses the 300ms staggering of the two parallel connection attempts, allowing each attempt to run through to the full 189 seconds. Opera will wait for the full 189 seconds before failing over.

Dual Stack Behaviour on iOS

```
OS: iOS 5.0.1
Browser: Safari
```

The iPhone supports a Dual Stack environment on its WiFi Interface.

```
URL: www.rx.td.h.labs.apnic.net

Time     Activity

         IPv4                        IPv6

   0     → DNS A? www.rx.td.h.labs.apnic.net
   0                                 → DNS  AAAA? www.rx.td.h.labs.apnic.net
 417                                   ← AAAA  2a01:4f8:140:50c5::69:dead
   0       ← A 88.198.69.81
 180                                 → SYN 2a01:4f8:140:50c5::69:dead
 721     → SYN 88.198.69.81
 290                                 → SYN 2a01:4f8:140:50c5::69:dead
   4       ← SYN+ACK 88.198.69.81
  35     → ACK 88.198.69.81
   0     [HTTP session]
```

On an iOS platform the Safari browser initiates the DNS queries in parallel. The system initially starts an IPv6 session, and after a 721ms timeout it starts a parallel session using IPv4. After a 991ms interval it sends a second SYN probe to the IPv6 address. When the IPv4 SYN probe responds, Safari latches onto this session and completes the HTTP request.

Now lets try this with a URL with three unresponsive IPv6 addresses.

```
URL: www.rxxx.td.h.labs.apnic.net

Time     Activity

         IPv4                        IPv6

   0     → DNS A? www.rxxx.td.h.labs.apnic.net
   1                                 → DNS  AAAA? www.rxxx.td.h.labs.apnic.net
 378       ← A 88.198.69.81
   1                                   ← AAAA  2a01:4f8:140:50c5::69:dead
                                               2a01:4f8:149:50c5::69:deae
                                               2a01:4f8:149:50c5::69:deaf

 296                                 → SYN 2a01:4f8:140:50c5::69:deae (a)
 718                                 → SYN 2a01:4f8:140:50c5::69:deaf (b)
 189                                 → SYN 2a01:4f8:140:50c5::69:deae (a)
 425                                 → SYN 2a01:4f8:140:50c5::69:dead (c)
 131     → SYN 88.198.69.81 (d)
 153                                 → SYN 2a01:4f8:140:50c5::69:deaf (b)
 189       ← SYN+ACK 88.198.69.81 (d)
 116                                 → SYN 2a01:4f8:140:50c5::69:deae (a)
   3     → ACK 88.198.69.81 (d)
   0     [HTTP session (d)]
```

This is somewhat unexpected. Again IPv6 is probed first, and after 718ms a second probe is initiated, but this time it uses the second IPv6 address. After a total of a 907ms timeout the first probe is resent. After a further 425ms the third IPv6 address is probed, and after another 131ms, or a total of 1574ms since the initial probe was commenced it now opens up a probe on IPv4. The browser now has 4 probe sessions active, three in IPv6 and one in IPv4. After 992ms to the second unresponsive IPv6 address the browser sends a second probe. When the IPv4 SYN generates a SYN+ACK response curiously the browser does not immediately send an ACK and close off the IPv6 probes. Instead it waits a further 116ms and, 899ms after the second probe, Safari sends a third probe to the first unresponsive IPv6 address. Only then does it respond to the IPv4 SYN+ACK and start up the fetch.

It seems that Safari uses a 990ms retransmit timer, and attempts to connect on all IPv6 addresses before performing a fallback to IPv4.

So, finally, lets try iOS on a URL where there are 2 addresses in each protocol, and all are unresponsive.

```
 URL: www.rxxzz.td.h.labs.apnic.net

 Time     Activity

          IPv4                         IPv6

    0     → DNS A? www.rxxzz.td.h.labs.apnic.net
    1                                  → DNS  AAAA? www.rxxzz.td.h.labs.apnic.net
  379       ← A 203.133.248.96
                203.133.248.95
    2                                    ← AAAA  2a01:4f8:140:50c5::69:deaf
                                                 2a01:4f8:149:50c5::69:dead
  127                                  → SYN 2a01:4f8:140:50c5::69:deaf
  129                                  → SYN 2a01:4f8:140:50c5::69:dead
  137     → SYN 203.133.248.95
  714     → SYN 203.133.248.96
   45                                  → SYN 2a01:4f8:140:50c5::69:deaf
  202                                  → SYN 2a01:4f8:140:50c5::69:dead
  101     → SYN 203.133.248.95
  721     → SYN 203.133.248.96
    1                                  → SYN 2a01:4f8:140:50c5::69:deaf
  202                                  → SYN 2a01:4f8:140:50c5::69:dead
  105     → SYN 203.133.248.95
  733     → SYN 203.133.248.96
    0                                  → SYN 2a01:4f8:140:50c5::69:deaf
  202                                  → SYN 2a01:4f8:140:50c5::69:dead
  102     → SYN 203.133.248.95
  727     → SYN 203.133.248.96
    0                                  → SYN 2a01:4f8:140:50c5::69:deaf
  201                                  → SYN 2a01:4f8:140:50c5::69:dead
  103     → SYN 203.133.248.95
  728     → SYN 203.133.248.96
    0                                  → SYN 2a01:4f8:140:50c5::69:deaf
  202                                  → SYN 2a01:4f8:140:50c5::69:dead
  102     → SYN 203.133.248.95
  726     → SYN 203.133.248.96
 1026                                  → SYN 2a01:4f8:140:50c5::69:deaf
  202                                  → SYN 2a01:4f8:140:50c5::69:dead
  102     → SYN 203.133.248.95
  720     → SYN 203.133.248.96
 3491     → SYN 203.133.248.95
    1                                  → SYN 2a01:4f8:140:50c5::69:deaf
    0                                  → SYN 2a01:4f8:140:50c5::69:dead
  624     → SYN 203.133.248.96
    0     → DNS A? www.rxxzz.td.h.labs.apnic.net
    0                                  → DNS  AAAA? www.rxxzz.td.h.labs.apnic.net
  724                                    ← AAAA  2a01:4f8:140:50c5::69:dead
                                                 2a01:4f8:149:50c5::69:deaf
    3       ← A 203.133.248.95
                203.133.248.96
 6669     → SYN 203.133.248.95
    0                                  → SYN 2a01:4f8:140:50c5::69:dead
    0                                  → SYN 2a01:4f8:140:50c5::69:deaf
  626     → SYN 203.133.248.96
 5222     → DNS A? www.rxxzz.td.h.labs.apnic.net
    0                                  → DNS  AAAA? www.rxxzz.td.h.labs.apnic.net
  393       ← A 203.133.248.96
                203.133.248.95
    0                                    ← AAAA  2a01:4f8:140:50c5::69:dead
                                                 2a01:4f8:149:50c5::69:deaf
10283     → SYN 203.133.248.95
    1                                  → SYN 2a01:4f8:140:50c5::69:dead
    0                                  → SYN 2a01:4f8:140:50c5::69:deaf
  204     → SYN 203.133.248.96
 1983     → DNS A? www.rxxzz.td.h.labs.apnic.net
    0                                  → DNS  AAAA? www.rxxzz.td.h.labs.apnic.net
  380       ← A 203.133.248.95
                203.133.248.96
   22                                       ← AAAA  2a01:4f8:140:50c5::69:dead
                                                    2a01:4f8:149:50c5::69:deaf
12515     → DNS A? www.rxxzz.td.h.labs.apnic.net
    1                                     → DNS  AAAA? www.rxxzz.td.h.labs.apnic.net
  393                                       ← AAAA  2a01:4f8:140:50c5::69:dead
                                             2a01:4f8:149:50c5::69:deaf
    3       ← A 203.133.248.95
                203.133.248.96
12000     [failure notice to the application]
-----
64300
```

The total time before the connection failure was passed back to the application was 64.3 seconds. Each address was probed 10 times, with a backoff timer set of 1, 1, 1, 1, 1, 2, 4, 8, 16, and 32 seconds between successive SYN probes. What is quite different here was that at 12.8 seconds, 26.1 seconds, 38.9 seconds and 51.9 seconds (approximately at 12 second intervals) the DNS query was repeated. The timing of the backoff algorithm appears to be based on some internal estimate of the RTT of the default route. In this case it appears that the system has a cached IPv6 default RTT of 130ms and a cached IPv4 RTT of some 700ms. The fallback algorithm appears to be an ordering of addresses by RTT estimate, and each address is added to the probe set upon the expiration of the RTT timer of the pervious address.

Summary

This table shows the outcomes of dual stack failover tests for various operating systems and browser combinations. Each cell in this table contains the version of the browser that was tested on this system, the time taken to fail over when the preferred protocol was unresponsive, and the default protocol preference (where 'x' means that the faster connection is preferred).

	Firefox	**Firefox fast-fail**	**Chrome**	**Opera**	**Safari**	**Explorer**
MAC OS X 10.7.2	8.0.1 75s IPv6	8.0.1 0ms x	16.9.912.41 b 300ms DNS	11.52 75s IPv6	5.1.1 270ms x	
Windows 7	8.0.1 21s IPv6	8.0.1 0ms x	15.0.874.121 m 300ms DNS	11.52 21s IPv6	5.1.1 21s IPv6	9.0.8112.16421 21s IPv6
Windows XP	8.0.1 21s IPv6	8.0.1 0ms x	15.0.874.121 m 300ms DNS	11.52 21s IPv6	5.1.1 21s IPv6	9.0.8112.16421 21s IPv6
Linux 2.6.40.3-0.tc15	8.0.1 96s IPv6	8.0.1 0ms x	16.9.912.41 b 300ms DNS	11.60 beta 189s IPv6		
iOS 5.0.1	-	-			? 720ms x	

Conclusions

There is much to be improved in the way in which operating systems and browsers generally handle dual stack situations. The operation of Firefox 8 with fast failover is perhaps the most efficient of all the browsers tested in terms of its ability to optimise its behaviour across all of the operating systems. It's not clear that the 300ms staggering used by Chrome is of any particular value. Noting the predilection for a number of these browsers to open 2 or even 3 parallel ports from the outset, considerations about imposing too high a TCP connection load on the server now appear to have become a mere historical curiosity in the unending search for blindingly fast browser performance. So Chrome's restraint with a 300ms timer is certainly polite, but on the other hand Chrome is the more profligate in terms of parallel connections as it starts off with 3 parallel connections when the RTT is over 250ms. I'd offer the view that Chrome would be faster in the corner cases if it dropped the third port and at the same time dropped the fallback timer from 300ms to 0ms.

Safari and Mac OS X combine to attempt to guess the fastest connection, rather than simply apply a brute force approach of using a parallel connection. Given that it only opens up a single port on startup its politeness to the server could be considered bordering on being parsimonious, and the additional load imposed by using a fully parallel connection setup in both protocols with a single connection in each protocol would only put it on a level of the default Firefox 8 behaviour. The efforts to cache the RTT estimates and order he connection attempts by increasing RTT are an interesting optimisation, and where different protocols give different RTTs, as in the case here where there was a 20ms difference in the paths in IPv6 and IPv4 Safari managed to learn of this and apply the correct preference, with a little tinkering with IPv6-only and IPv4-only URLs in order to prime the cache of stored RTTs. However, Safari also places an RTT time estimate on the "default" route in both IPv4 and IPv6, and uses this internal value to bias its initial connection choice in dual stack scenarios in the absence of specific information. It appears that what is not cached is unresponsive addresses, so while Safari is able to make efficient choices when there is reliable information already gathered, it makes poor choices when one of the addresses returned by the DNS is persistently unresponsive and when its estimate of the RTT to "default" (what ever that may be!) starts to drift.

There is an interesting design compromise as to whether to tightly couple the DNS and TCP connection or whether to embark on connection attempts as soon as the DNS results are returned in each protocol family. The browser wants to optimise two performance parameters, namely the initial connection time from the user's click to the time that the page starts to draw, and of course the total connection time for the entire page download. If the decision was to optimise the initial connection time then there is much to be said for setting off the two DNS queries in parallel and embarking on the associated TCP connection as soon as the DNS response is received. But if the decision is to optimise the session performance, then the lower RTT will be selected if the browser performs a rendezvous after the DNS query phase and then fires off the opening SYN packets in parallel. If there is an appreciable difference in RTT between the two protocols then the faster path will complete the TCP connection first.

My choice? Right now Firefox 8 with fast failover enabled appears to offer the best performance on the platforms I've tested when you are after the fastest connection time if you are on a Windows or Linux platform. But you will need to set the fast failover option. Without this set to "true" its dual stack connection behaviour is pretty ordinary. If you are using a Mac then despite some of the erratic outcomes from the RTT tracking function when presented with pathologically broken cases, Safari appears to offer the fastest behaviour in dual stack scenarios.

But what if you happen to love the user interface provided by your favourite browser and don't want to switch? In that case if you are still after improving connection speeds for unresponsive web sites you may certainly want to look at the SYN retransmit settings if you are on a MAC or a Linux platform and knock the number of retransmits down to a more reasonable 10 seconds in the case of the Mac or to 2 SYN retries in the case of Linux.

Test Resources

I performed these tests using `tcpdump` on a number of hosts. In setting up these tests I use a number of wildcard DNS domains. (They are wildcarded to override some browser's predilection for caching DNS answers. So any valid DNS string can be used as a prefix here.) Each of these domains has a different behaviour, as shown in the table below.

URL	Behaviour
*.r4.td.h.labs.apnic.net	IPv4 only
*.r6.td.h.labs.apnic.net	IPv6 only
*.rd.td.h.labs.apnic.net	Dual Stack
*.rx.td.h.labs.apnic.net	1 working IPv4 , 1 unresponsive IPv6
*.rxx.td.h.labs.apnic.net	1 working IPv4 , 2 unresponsive IPv6
*.rxxx.td.h.labs.apnic.net	1 working IPv4 , 3 unresponsive IPv6
*.rz.td.h.labs.apnic.net	1 unresponsive IPv4, 1 working IPv6
*.rzz.td.h.labs.apnic.net	2 unresponsive IPv4, 1 working IPv6
*.rzz.td.h.labs.apnic.net	3 unresponsive IPv4, 1 working IPv6
*.rxz6.td.h.labs.apnic.net	1 unresponsive IPv4, 1 working IPv6, 1 unresponsive IPv6
*.rxz4.td.h.labs.apnic.net	1 unresponsive IPv4, 1 working IPv4, 1 unresponsive IPv6
*.rxxzz.td.h.labs.apnic.net	2 unresponsive IPv4, 2 unresponsive IPv6

An Afterword – May 2012

As an update to this article, I received the following message:

> Subject: Dual Stack Esotropia - android, symbian, windows phone
> From: Matej Gregr
> To: Geoff Huston
>
> I recently read your emails in an IPv6 mailing list about your article "Dual Stack Esotropia". I did some testing with android, symbian and windows phone mobile devices so if you are still interested, you can find the traces in the attachment.
>
> It seems, that Windows phone with last Mango 7.5, according to my testing, was not able to request AAAA record. Symbian device (old Nokia e52 in this case) uses IPv6 to connect to the IPv6 only content. If a web server is dual-stacked, the mobile will always use IPv4 address - there is no failback mechanism to IPv6 if IPv4 is not working. I tested only the mobile web client not opera mini or opera mobile. What I find interesting, that the old symbian provides router advertisement client as well as dhcpv6 client and tries to obtain an IPv6 address from a DHCPv6 server, if managed flag is set in RA.
>
> The Android phone was Samsung Galaxy s2 with Android 2.3.3. I can do some more testing with galaxy tab with android 3. Unfortunately I don't have any device with the recent android 4, but I think, that the behaviour will be the same as with android 2.3.3.
>
> Cheers,
> Matej

Android

So lets look at these traces from Matej and see how Android behaves:

```
Device: Samsung Galaxy S2 with Android 2.3.3
Connection: test.rd.td.h.labs.apnic.net

     Time        Activity

       0        → DNS AAAA? test.rd.td.h.labs.apnic.net
     816          ← AAAA 2a01:4f8:140:50c5::69:72
      39        → DNS A? test.rd.td.h.labs.apnic.net
      37          ← A 88.198.69.81
       7        → SYN 2a01:4f8:140:50c5::69:72
      47          ← SYN+ACK 2a01:4f8:140:50c5::69:72
       2        → ACK 2a01:4f8:140:50c5::69:72
--------
     948
```

Why does AAAA DNS query take so long to respond? My assumption here is that this initial query caused the DNS resolver to resolve the name servers for the entire name chain, while the second query could take advantage of local caching of these responses.

The second observation is that this appears to be a serialised system, with both the DNS and the TCP connection attempts happening in parallel, not serial.

What about the case where the IPv6 is unreachable?

```
Device: Samsung Galaxy S2 with Android 2.3.3
Connection: test.rx.td.h.labs.apnic.net

     Time        Activity

       0        → DNS AAAA? test.rx.td.h.labs.apnic.net
```

```
        37          ← AAAA 2a01:4f8:140:50c5::69:dead
         8        → DNS A? test.rx.td.h.labs.apnic.net
        37          ← A 88.198.69.81
         3        → SYN 2a01:4f8:140:50c5::69:dead
      3016        → SYN 2a01:4f8:140:50c5::69:dead
      6008        → SYN 2a01:4f8:140:50c5::69:dead
     12040        → SYN 88.198.69.81
        35        ← SYN+ACK 88.198.69.81
         3        → ACK 88.198.69.81
  --------
     21187
```

It seems that Android is using a IPv6 preference with a 19 second total failover timer, similar to that already encountered on Windows system. Let's push this now and look at a connection that has three unresponsive IPv6 addresses and a working IPv4 address

```
Device: Samsung Galaxy S2 with Android 2.3.3
Connection: test.rxxx.td.h.labs.apnic.net

     Time        Activity

        0         → DNS AAAA? test.rxxx.td.h.labs.apnic.net
       35           ← AAAA 2a01:4f8:140:50c5::69:deaf, 2a01:4f8:140:50c5::69:dead,
                           2a01:4f8:140:50c5::69:deae
        3         → DNS A? test.rxxx.td.h.labs.apnic.net
       35           ← A 88.198.69.81
        4         → SYN 2a01:4f8:140:50c5::69:deaf
     3022         → SYN 2a01:4f8:140:50c5::69:deaf
     6008         → SYN 2a01:4f8:140:50c5::69:deaf
    12039         → SYN 2a01:4f8:140:50c5::69:dead
     3010         → SYN 2a01:4f8:140:50c5::69:dead
     6010         → SYN 2a01:4f8:140:50c5::69:dead
    12020         → SYN 2a01:4f8:140:50c5::69:deae
     3010         → SYN 2a01:4f8:140:50c5::69:deae
     6010         → SYN 2a01:4f8:140:50c5::69:deae
    12026         → SYN 88.198.69.81
       35           ← SYN+ACK 88.198.69.81
        4         → ACK 88.198.69.81
 --------
    63271
```

It seems that the Android algorithm is to try each IPv6 address, using a 3 packet 19 second test cycle per address, and only when all IPv6 addresses have been tested will the system failover to IPv4.

Again, we can push this a bit further, and the following test shows Android in operation when there are two unresponsive addresses for both IPv4 and IPv6.

```
Device: Samsung Galaxy S2 with Android 2.3.3
Connection: test.rxxzz.td.h.labs.apnic.net

     Time        Activity

        0         → DNS AAAA? test.rxxzz.td.h.labs.apnic.net
       37           ← AAAA 2a01:4f8:140:50c5::69:deaf, 2a01:4f8:140:50c5::69:dead
        3         → DNS A? test.rxxzz.td.h.labs.apnic.net
       36           ← A 203.133.248.96, 203.133.248.95
        4         → [3 SYNs: 2a01:4f8:140:50c5::69:deaf]
    19047         → [3 SYNs: 2a01:4f8:140:50c5::69:dead]
    19040         → [3 SYNs: 203.133.248.96]
    19042         → [3 SYNs: 203.133.248.95]
    19147         → [3 SYNs: 2a01:4f8:140:50c5::69:deaf]
    19041         → [3 SYNs: 2a01:4f8:140:50c5::69:dead]
    19040         → [3 SYNs: 203.133.248.96]
    19042         → [3 SYNs: 203.133.248.95]
    19040           [report failure]
 --------
   152519
```

This is curiously persistent on the part of Android: it cycles through the IPv6 addresses using the same 19 second 3 packet test, and then cycles through the IPv4 addresses. At this point it then returns to the start and repeats the cycle once more.

Symbian

So now lets see how Symbian behaves.

If the server is dual-stacked, then the Symbian client will always just use IPv4, and not attempt a IPv6 connection at all. In this case it will not even query the DNS for the existence of an IPv6 address/

```
Device: Symbian, Nokia E52
Connection: test.rd.td.h.labs.apnic.net

     Time       Activity

       0        → DNS A? test.rd.td.h.labs.apnic.net
      39          ← A 88.198.69.81
       4        → SYN 88.198.69.81
      34          ← SYN+ACK 88.198.69.81
       3        → ACK 88.198.69.81
     ---
      80
```

This raises two questions: What happens when there is no IPv4 address (i.e. the target is IPv6 only)? What happens if the target has IPv4 and IPv6 addresses, but the IPv4 address is unresponsive?

Lets look at an Ipv6-only fetch from this Symbian host:

```
Device: Symbian, Nokia E52
Connection: test.r6.td.h.labs.apnic.net

     Time       Activity

       0        → DNS A? test.rd.td.h.labs.apnic.net
      36          ← no such RR
       2        → DNS AAAA? test.rd.td.h.labs.apnic.net
      36          ← AAAA 2a01:4f8:140:50c5::69:72
       4        → SYN 2a01:4f8:140:50c5::69:72
     157          ← SYN+ACK 2a01:4f8:140:50c5::69:72
       2        → ACK 2a01:4f8:140:50c5::69:72
     ---
     237
```

And secondly let's see if Symbian performs failover from an unresponsive IPv4 address to an IPv6 address.

```
Device: Symbian, Nokia E52
Connection: test.rz.td.h.labs.apnic.net

     Time       Activity

       0        → DNS A? test.rz.td.h.labs.apnic.net
      35          ← DNS A 103.10.232.30
       3        → SYN 103.10.232.30
    3010        → SYN 103.10.232.30
    6008        → SYN 103.10.232.30
   11999        → SYN 103.10.232.30
   24000        → SYN 103.10.232.30
   48002        → SYN 103.10.232.30
     ---
   93049
```

Obviously, there is no failover here. Symbian attempts five further connection attempts, spaced over a total of 93 seconds for the entire connection process. At this point Symbian gives up on the connection, and does not attempt the connection using IPv6.

Disclaimer

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

Author

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

www.potaroo.net