The ISP Column
A monthly column on things Internet

May 2012
Geoff Huston

Bemused Eyeballs

Tailoring Dual Stack Applications in a CGN Environment

How do you create a really robust service on the Internet? How can we maximise speed, responsiveness, and resiliency? How can we set up an application service environment in today's network that can still deliver service quality and performance, even in the most adverse of conditions? And how can we engineer applications that will operate robustly in the face of the anticipated widespread deployment of Carrier Grade NATs (CGNs) as the Internet lumbers into a rather painful phase of a depleted IPv4 free pool and continuing growth pressures. Yes, IPv6 is the answer, but between here and there are a few challenges. And one of these is the way applications behave in a dual stack environment. I'd like to look at this in a little more detail here.

An Aside - The BitTorrent Protocol

One of the more successful, and probably one of the most capable, protocols in today's Internet has a pretty lousy reputation. BitTorrent is typically associated with the distribution of material without the appropriate or prerequisite authority. This association is so widely held that it is not uncommon for service providers to deploy deep packet inspection and interception equipment and attempt to detect torrent flows within their network and deliberately disrupt them.

> On the whole this attempt to disrupt torrent traffic has not been all that effective, and the disruption falls into the class of social behaviour that is best described as *pantomime* - to be seen to be doing something, often taking considerable time and spending considerable sums of money to be seen to be doing something, while the actions being taken are, in fact, largely ineffectual, and recognised by all to be largely ineffectual!

The bad reputation BitTorrent has acquired is thoroughly undeserved, if you are looking at BitTorrent purely as a protocol. In fact, BitTorrent appears to be an extremely effective data communications protocol. The other extremely popular protocol in use today is HTTP, the protocol behind the Web. If the same intervention and disruption measures in use today to target BitTorrent flows were targeted against the HTTP protocol, then an ISP's customers would quickly experience a dramatically degraded service. But it appears that BitTorrent has largely shrugged off such hostile measures and continues to operate in a largely unaffected manner. Why is BitTorrent so resistant to such attempts to deliberately disrupt it? How is this application so robust, and so resilient to such forms of attack?

BitTorrent is more than robust. It's also extremely fast. For data that is served by many sites BitTorrent self-optimises. It distributes the data flow across a wide number of individual TCP sessions so that a large number of feed sites provide an incoming data flow. It also optimises across the set of feed sites,

in that high capacity low delays feed sites are asked to provide more data than slower performing feed sites. BitTorrent is an example of an application that offers speed, load capacity, performance and robustness in its service offering. What can we learn from this application in terms of robust service design?

The Torrent Data Architecture

It seems to me that the most notable feature of the BitTorrent architecture is that it exploits diversity and redundancy in its data model. BitTorrent does not just open up a single TCP session to a single remote point to perform a data download. BitTorrent exploits the fact that there are many copies of the same data. It opens up many sessions to many feed points, and distributes block data download requests across the entire set of feed points. Like the underlying Transmission Control Protocol (TCP), BitTorrent is also a rate adaptive protocol. In this way if a torrent download finds that one or more feed points are far faster than the others, it will end up directing many more block requests to the faster feed points than the slower points. If a feed point goes offline, then the Torrent application redirects its block queries to other feed points. And when new feed points appear the Torrent application will also adapt and add these new feed points into its active peer set.

This behaviour exposes a more general observation about the Internet, namely that use of parallelism in communications, and diversity and redundancy in data publication, can be harnessed to generate a superior service outcomes in terms of speed, robustness, optimisation, and availability, and that this approach can offer continued service robustness and performance even in extremely adverse conditions.

Dual Stack Applications

Lets apply this observation to the Dual Stack world of the Internet in transition to IPv6. Are we able to exploit this diversity of protocol availability between a client and a server to generate a better quality of outcome?

Is a dual stack Internet capable of showing greater levels of robustness and better performance outcomes than a single protocol IPv4 Internet, or even a single protocol IPv6 network? Can we add a "BitTorrent-like" operation to a dual protocol application? In other words, can we make dual stack servers and clients that are more robust and perform better than their single-stacked counterparts?

For a number of years the answer to this question was slightly unexpected, in so far as when both server and client were dual-stack capable, then the quality of the connection process, measured in terms of the time to establish a connection, was no better, and potentially far worse then the situation when the client is mono-stack client. Even more unexpected was the answer that the performance experiences by a dual stack client was more likely to be worse, and possibly likely to be far worse, than when the client was an IPv4-only client.

What we've been experiencing with dual stack clients using the popular operating system and browser combinations when connecting to dual stack servers can be summarized summarised as "one protocol good, two protocols bad."

Given that the path to IPv6 transition relies on a period of operation of dual stack clients and services, this outcome is hardly encouraging. What's going on?

A quick look inside TCP Connections

To answer this, it might be useful to look at the implementation of the TCP protocol and the way in which it establishes a connection.

TCP uses a three way connection packet exchange. The client sends a TCP packet to the server, with the SYN flag set in the TCP flags field. The server responds with a TCP packet to the client with the SYN and ACK flags set (a "SYN/ACK"), and the client completed the connection by sending the server a final connection establishment packet, with just the ACK flag set.

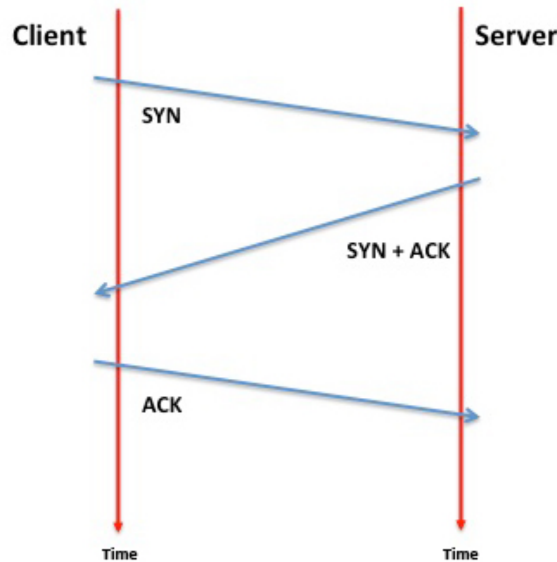

Figure 1 – TCP 3-Way Handshake

What will the client do if it sends a SYN but does not get back a SYN/ACK? The behaviour of TCP in this case varies a little between implementations, but the basic approach is that TCP is not prepared to give up so easily. TCP will resend the initial SYN packet a few more times and be a little more persistent in waiting for the matching SYN+ACK response

Windows systems (Windows XP, Windows Vista and Windows 7) use 2 retransmits, waiting 3 and 6 seconds respectively, and it will wait a further 12 seconds before handing back to the application a connection failure indication. The entire process takes a total of 19 seconds for the local TCP stack to reach the conclusion that the connection attempt has failed.

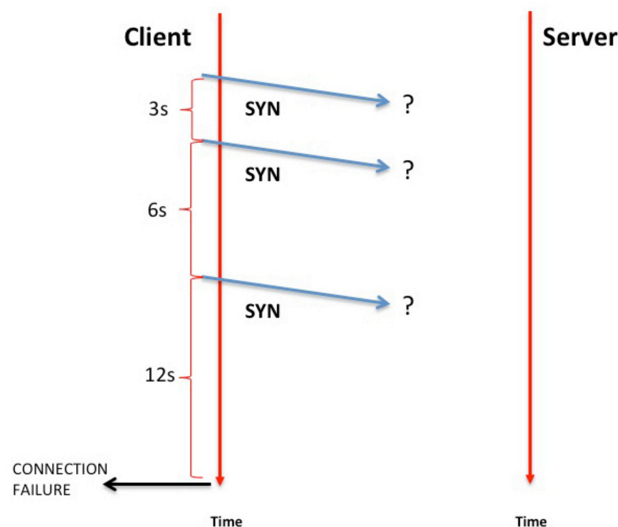

Figure 2 – TCP Connection Failure

It's important to note that in this case the failure may be anywhere along the forward path of the SYN packet or along the reverse path of the SYN+ACK response. From the client's perspective all that the local TCP session knows is that it has sent a SYN and received no matching SYN+ACK response.

Unix-based TCP implementations are similar to Windows TCP, but tend to be more persistent in terms of waiting for a response to the SYN packet. MAC OSx 10.7.2 (whose TCP stack and configuration settings closely resemble that of FreeBSD) uses 10 retransmits spaced at intervals of 1, 1, 1, 1, 2, 4, 8, 16, and 32 seconds, and a generic TCP connection attempt will wait for a total of 75 seconds before returning a connection failure indication to the upper level application. The Linux operating system uses a slightly different retransmit settings. A setting I've encountered frequently is one with 5 SYN retransmits with successive wait intervals of 3, 6, 9, 12, 24 and 48 seconds. Linux will wait a further 96 seconds before reporting connection failure. That's an impressive 189 seconds, or slightly over 3 minutes between the application opening the session and triggering TCP to send the first SYN packet and receiving a connection failure indication from the TCP protocol engine.

Dual Stack Connections

Lets now take the case of a dual stack client system that is enabled with IPv4 and IPv6. In this case I want to look at the situation where the IPv6 connection to a server is not working. Obviously this could be due to a myriad of issues, from placing a spurious AAAA record in the DNS, through over-zealous filtering, to some form of network based fault.

> This situation can be constructed artificially. This URL:
>
> http://test.rx.td.h.labs.apnic.net/1x1.png
>
> has both A and AAAA DNS records for the domain name, which implies to a client that the server is a dual stack server. However in this case the IPv6 part, the AAAA record uses an unreachable IPv6 address.

Lets look at a configuration with Windows 7 and the Explorer browser application when encountering this situation.

```
Operating System: Windows 7
Browser: Explorer 9.0.8112.16421

 Time  Protocol     Payload

    0  IPv4 UDP DNS A?    12344.rx.td.h.labs.apnic.net.
  325  IPv4 UDP DNS A       88.198.69.81
    1  IPv4 UDP DNS AAAA? 12344.rx.td.h.labs.apnic.net.
  322  IPv4 UDP DNS AAAA    2a01:4f8:140:50c5::69:dead
    4  IPv6 TCP SYN     port 49261 -->  2a01:4f8:140:50c5::69:dead port 80
    0  IPv6 TCP SYN     port 49262 -->  2a01:4f8:140:50c5::69:dead port 80
 3013  IPv6 TCP SYN     port 49261 -->  2a01:4f8:140:50c5::69:dead port 80
    0  IPv6 TCP SYN     port 49262 -->  2a01:4f8:140:50c5::69:dead port 80
 6000  IPv6 TCP SYN     port 49261 -->  2a01:4f8:140:50c5::69:dead port 80
    0  IPv6 TCP SYN     port 49262 -->  2a01:4f8:140:50c5::69:dead port 80
12000  [report failure]
-----
19665
```

Why are these TCP implementations so slow to report failure?

The best answer I can provide is because TCP is far older than IPv6!

When many of the behaviours for TCP were devised, it was a world of a single protocol and a single service address and sometimes unreliable networks where packet loss was not entirely uncommon. In such an environment a certain level of persistence in attempting a connection was often an advantage, and had little downside. Given that there was no "plan B", then TCP was generally configured to be extremely persistent and patient in trying to establish a connection. It was plan A or nothing!

But then we introduced the added complication of dual protocol stacks.

The initial algorithm used by most browsers when encountering a dual stack scenario was quite simple (and quite simple-minded): First try to connect using IPv6 and when the TCP connection attempt reports back failure, then fail back to IPv4.

While this approach sounds good, in attempting to use IPv6 in preference to IPv4, and holding IPv4 in reserve as the "plan B" for the connection, it's the details that make this ugly. The algorithm will first call up TCP with the IPv6 address. Either the connection call will return with success, or it will perform a set of SYN retransmits, and after 19, 75 or even 189 seconds, depending on the operating system and the browser, it will report failure, and then and only then will be browser shift to IPv4 retry the connection. Lets go back to that same Windows 7 + Explorer combination and look at the complete connection trace:

```
Operating System: Windows 7
Browser: Explorer 9.0.8112.16421

 Time  Protocol     Payload

    0  IPv4 UDP DNS A?    12344.rx.td.h.labs.apnic.net.
  325  IPv4 UDP DNS A       88.198.69.81
    1  IPv4 UDP DNS AAAA? 12344.rx.td.h.labs.apnic.net.
  322  IPv4 UDP DNS AAAA    2a01:4f8:140:50c5::69:dead
    4  IPv6 TCP SYN     port 49261 -->  2a01:4f8:140:50c5::69:dead port 80
    0  IPv6 TCP SYN     port 49262 -->  2a01:4f8:140:50c5::69:dead port 80
 3013  IPv6 TCP SYN     port 49261 -->  2a01:4f8:140:50c5::69:dead port 80
    0  IPv6 TCP SYN     port 49262 -->  2a01:4f8:140:50c5::69:dead port 80
 6000  IPv6 TCP SYN     port 49261 -->  2a01:4f8:140:50c5::69:dead port 80
    0  IPv6 TCP SYN     port 49262 -->  2a01:4f8:140:50c5::69:dead port 80
12000  IPv4 TCP SYN     port 49263 -->  88.198.69.81 port 80
    1  IPv4 TCP SYN     port 49264 -->  88.198.69.81 port 80
  319  IPv4 TCP SYN+ACK port 49263           <-- 88.198.69.81 port 80
    0  IPv4 TCP SYN+ACK port 49264           <-- 88.198.69.81 port 80
    0  IPv4 TCP ACK     port 49263 -->  88.198.69.81 port 80
    1  IPv4 TCP ACK     port 49264 -->  88.198.69.81 port 80
-----
21986
```

That's 22 seconds from the time of the request to the time the data flow can commence!

Admittedly I used a test with a very high client/server round trip time of 319 ms in this example, but even if one were to use a client/server pair with a 10ms round trip time that would still only shave off 1 second from the total wait time.

What is happening in this configuration is that application is first attempting to make the connection using IPv6, firing off two connection attempts in parallel, and then waiting for the operating system's TCP implementation to report success or failure. Given that the IPv6 address is deliberately set up to be unresponsive, then failure is inevitable, and after two retries and a total of 19 seconds we see the application switch over and try to connect using IPv4, which is immediately responsive. However, the amount of time between entering the URL in the browser and the response is now some 20 seconds. Instead of an instantaneous response that you may be used to, what is experienced is a service that redefines "slow" into "glacial". And the problem can be traced back to TCP.

IPv6 Failure

Is failure that rare? How often do we see this form of delay and protocol fallback?

Surprisingly often. The following is based on measuring up to 600,000 end users per day for IPv6 capability and failure characteristics. What is evident from this measurement exercise is that the connection failure rate for IPv6 is some 50% of all connection attempts!

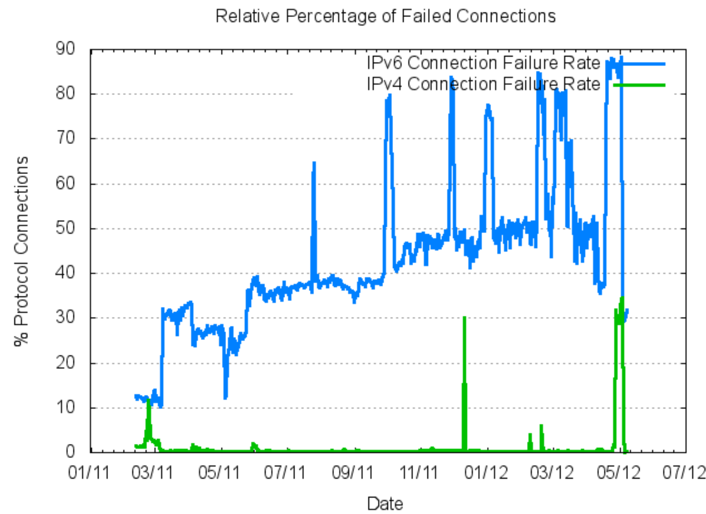

Figure 3 – IPv4 and IPv6 Connection Failure Rates

This is an amazingly high number, but it has a simple explanation. In an effort to increase the penetration of IPv6 the Microsoft Windows platform has for many years been shipping systems that include a combination of dual stack functionality and auto-tunnelling of IPv6 over IPv4. This IPv6 auto-tunnelling functionality had to be manually be turned on in Windows XP, but was on by default for the more recent Windows 7 and Windows Vista platforms. If the Windows unit was located behind a NAT it used Teredo as the auto tunnelling mechanism and if not then it used 6to4. And, like all forms of auto-tunnelling, the experience has been less than impressive. Teredo has a failure rate where 40% - 45% off all connection attempts simply fail to complete. 6to4 is only a little better, where we are able to measure a failure rate of between 10% to 25%. This behaviour has been in shipped products for many years, but it was only identified as a significant cause of user frustration quite recently. Why? Because so few services were configured as dual stack. So this was a problem just waiting to be exposed, and of course the increased levels of urgency over IPv6 transition due to the exhaustion of IPv4 addresses was the trigger.

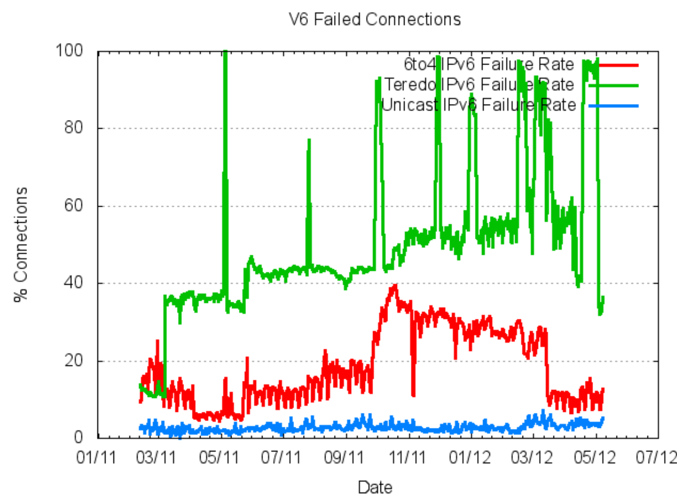

Figure 4 –IPv6 Connection Failure Rates by Access Type

There was a simple correction, which was applied to the operating system in the form of a preference table. In general, browsers will prefer to use IPv6 over IPv4, unless their local IPv6 interface is a 6to4 or Teredo auto-tunnel, in which case they will prefer to use IPv4.

This was an improvement, in so far as Teredo was used quite sparingly when this preference change was applied . Even so, the Teredo connection failure rate is disgustingly bad! In addition, Teredo uses a small set of anycast relays to setup the connection, and the more Teredo is used the greater the load on these relays and the higher the potential for load-related failures to add to Teredo's woes.

There was a further simple correction, which is in place in the most recent patches to Windows Vista and Windows 7, namely that if the only local IPv6 interface is a Teredo interface then the local system will not even poll the DNS for an IPv6 address for a URL.

> This has both solved and not solved the problem. It has solved the problem for Windows browser use, where a Windows system with IPv6 turned on and a local Teredo interface will not use the Teredo interface and conventional tests of browser behaviour and failure modes now shows that Teredo's absymal failure rate does not drag down their performance numbers. It has not solved the problem because of the ever escalating arms race between torrent applications and ISPs. These days many ISPs deploy deep packet inspectors (DPIs) to try and identify torrent traffic and either shape the traffic to reduce its traffic flow rates or in the most extreme cases to try and disrupt the torrent flows to the point of making the application unusable for the client. Needless to say the rather agile torrent authors are on top of this and one reaction was to use encapsulation to defeat the DPIs. It should come as no surprise to learn that from all the data collected so far a sizeable proportion, if not the overwhelming majority of connections on IPv6 today are torrent flows to and from IPv4 hosts using the Teredo interface. And because IPv6 is on by default and Teredo is an auto-tunnel interface most of the folk who have these end systems are entirely unaware that this is happening! Of course the Teredo connection failure rate is still a whopping 40% - 45% of all connection attempts, but BitTorrent is one of the few applications where this is not a significant problem. Because the nature of Torrent data sets, where massive redundancy in data sources is exploited, this kind of others crippling individual session failure rate for Teredo has little overall impact on the efficiency of torrent to deliver the data from the swarm to the end user.

But even if you ignore the auto-tunnel problems and just look at IPv6 and IPv4 in so-called "native" mode, there is still a residual performance issue that makes the dual stack experience worse for end users. For a variety of reasons there is still a residual 3% - 4% failure rate for IPv6 connections that has no counterpart in IPv4. Some 1 in 30 users they will use IPv6 and find that web pages load at a pace that is measured not in milliseconds but in tens or even hundreds of seconds - for these 1 in 30 users visiting a dual stack service turns the Internet's time clock from being so fast its beyond human perception to being so slow it is also beyond human perception! With a web page with a few hundred elements loaded into the domain object manager, which is not unusual in today's web, adding a 3 minute delay to every element in the page turns the pace of delivering content from dual stack servers back down to a pace that is unacceptable.

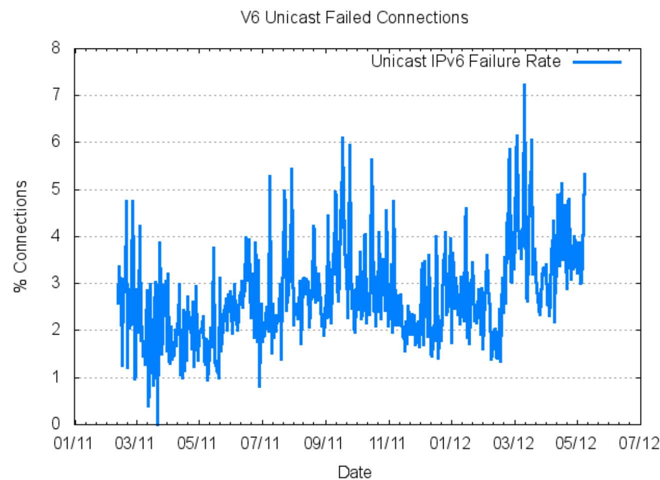

Figure 5 – IPv6 Unicast Connection Failure Rates

This was not the promise of IPv6. Something is still wrong.

Happy Eyeballs

The next step is to shift from sequential to parallel processing. This approach, "Happy Eyeballs," is described in an RFC[1]. The proposed approach is simple - if the client system is dual stack capable, then fire off connection attempts in both IPv4 and IPv6 in parallel, and use (and remember) whichever protocol completes the connection sequence first. The user benefits because there is no wait time and the decision favours speed - whichever protocol performs the connection fastest for that particular end site is the protocol that is used to carry the payload. The approach is also resilient, in so far as either protocol can be unresponsive, but this unresponsiveness is completely invisible to the end user.

There are three major browsers that have implemented a form of "Happy Eyeballs."

Safari and Mac OSx Eyeballs

Mac OSx 10.7 in combination with the Safari 5 browser have implemented something that I would call "Moderately Happy Eyeballs."

Rather than go with what is fastest, MAC OSx is using a hybrid approach of parallel and serial connection. It appears that MAC OSx keeps a running record of the delay to reach each destination address, and a running average delay for each protocol. The Safari browser direct performs a DNS resolution phase, and collects both the IPv4 and IPv6 responses. If there are both IPv4 and IPv6 addresses, then Safari will start a connection with whichever protocol has to lowest delay estimate. Safari also has a radically altered failover algorithm, and if there is no response after waiting for the estimated RTT time, it will then activate the other protocol and have both connection attempts operating in parallel. At this stage whichever protocol completes first is used. This looks good, but there is one aspect of this which is curios. If there are multiple addresses for the service point, then after the RTT timeout, instead of failing over to the other protocol, it will try with the second address, and so on. Only when all addresses have been tried will the other protocol be tested. Overall this is very fast, but multi-addressing may introduce some additional delay elements.

Chrome Eyeballs

The second browser with an implementation of Happy Eyeballs is Chrome. Again its not exactly happy eyeballs, but I'd call it "Happyish Eyeballs."

[1] Happy Eyeballs: Success with Dual-Stack Hosts, D. Wing, A. Yortchenko, RFC6555 http://tools.ietf.org/html/rfc6555

Chrome does not have an inbuilt preference between IPv4 and IPv6. Chrome will launch DNS queries for both IPv6 and IPv4 addresses in parallel (typically the IPv6 query is launched before the IPv4 query, but there is no significant gap between the two queries), and whichever DNS response is received first is the protocol that Chrome will use to try and make the connection. Chrome 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. If no response has been received after a total of 300ms then Chrome will shift over to the other protocol and repeat the same three port connection probe set.

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 far longer. The choice of which protocol is preferred for the initial connection attempt is based on the DNS response time, and I have to wonder about that. Its not clear that the DNS resolution speed is related in any clear way to the TCP delay time, so this is a somewhat surprising method of determining protocol preference.

Firefox Eyeballs

Firefox appears to implement fully parallel connection form of Happy Eyeballs, if you set the parameter network.http.fast-fallback-to-IPv4 to "true".

This setting causes Firefox to start of a DNS and TCP connection process in IPv4 and IPv6 in parallel As soon as the respective DNS query generates a response then the TCP connection process is commenced immediately.

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 initially tested address is 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. At the end of the connection attempts for an unresponsive address there is a second implementation bug. In this case 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.

Bemused Eyeballs

All these implementations are fast. It seems that Firefox has gone the furthest in implementing a truly parallel connection framework, but the different between all three in the case where one protocol is unresponsive is sufficiently small in most cases that this would not reduce the responsiveness of the service in any noticeable way.

So we've fixed the problem. Yes?

Well not quite.

The entire reason why we are running dual stack servers and clients is to progress the transition to IPv6. Dual Stack is not an end point, it is a transitional phase. When much of this transition was mapped out, many years ago, there was the assumption that this industry would proceed with the transition well in advance of the exhaustion of IPv4. This assumption has not proved to be the case, and we are now trying to perform a dual stack transition with one protocol now crippled by address exhaustion. The additional factor clients and servers now need to consider is the factor of the finite

capacity of the inevitable carrier grade NAT. During this dual stack transition we can only expect the network client base to continue to expand, while the available pool of IPv4 addresses is fixed. We can reasonably expect that the pressure on CGNs to monotonically increase. All of a sudden the establishment of a TCP connection through a CGN becomes a blocking action that prevents others from establishing a connection at all. The CGN binding space becomes a critically scarce resource.

This means that opening up a connection through a CGN should not be done lightly. And that's precisely what these implementations of Happy Eyeballs do, to one extent or another.

There is a reasonable response to this , which is described in the Happy Eyeballs document:

```
Thus, to avoid harming IPv4-only hosts which can only utilize IPv4,
implementations MUST prefer the first IP address family returned by
the host's address preference policy, unless implementing a stateful
algorithm described in Section 4.2.  This usually means giving
preference to IPv6 over IPv4, although that preference can be over-
ridden by user configuration or by network configuration.  If the
host's policy is unknown or not attainable, implementations MUST
prefer IPv6 over IPv4.
```

[RFC6555]

Rather than opening up a connection in the protocol that has the faster DNS resolution, or a faster average RTT or even the first to return an ACK packet, it would be better to try IPv6 first, and then open up an IPv4 connection if the IPv6 connection has not completed within a relatively short interval.

This looks a lot like the first version of the dual stack algorithm described at the start of this article.

What's different?

The difference is the introduction of parallelism into the mix. When the browser reverts to IPv4 and kicks off the IPv4 connection attempt, that does not mean that it should immediately stop listening for a SYN/ACK response from the earlier IPv6 connection attempt. Once the IPv4 SYN has been sent, the browser should then be prepared to work with whichever protocol first responds to the outstanding connection attempts. In other words it still a transformation from a serial effort of "IPv6 first then IPv4 second" to a parallel effort of trying both IPv4 and IPv6 in parallel, and being ready to work with which responds first. But the key aspect here is that while its a race to the first SYN+ACK, its now a handicapped race, where IPv6 enjoys a slight lead at the outset of the race. The lead need not be large - Chrome's 300ms delay interval would be reasonable. Perhaps Safari's single RTT time is a little too aggressive, but if this timer were to be backed off to a minimum of 300ms or twice the estimated RTT then again this would be a reasonable balance between performance and easing the pressure on the IPv4 CGNs.

So it appears that heading from serial to full parallelism is perhaps swinging too far the other way in terms of the tradeoff between conservatism and the desire for the fastest possible customer experience. What we need here, to balance the considerations of dual stack speed and CGN binding state consumption, is not exactly a pair of happy eyeballs, but a slightly biased pair of pleasantly amused eyeballs!

Disclaimer

The views expressed are the author's and not those of APNIC, unless APNIC is specifically identified as the author of the communication. APNIC will not be legally responsible in contract, tort or otherwise for any statement made in this publication.

About the Author

Geoff Huston B.Sc., M.Sc., 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 has been active in the Internet Engineering Task Force for many years.

www.potaroo.net