The ISP Column
A monthly column on things Internet

May 2013
Geoff Huston

But That's Impossible!

For some time now at APNIC Labs we've been running an experiment that is intended to measure the state of IPv6 capability across the Internet. To do this we use a number of experimental data sources. A number of web site administrators have kindly put our Javascript test code on their web page, so that visitors to their web page contribute to the measurement exercise (If you operate a web service you can help out too, if you want – the details of how to add these tests to your web page are at http://labs.apnic.net/tracker.shtml). In addition to Javascript on web pages, we also use active code embedded in an online advertisement to perform the same basic IPv6 capability measurement on clients who are delivered an impression of the advertisement. Across these two experimental approaches we perform a basic IPv6 capability test on between 800,000 and 1,000,000 clients each day. Such a large scale experiment is bound to produce some anomalous behaviours, but in this case there are a couple of outcomes that, as far as I can tell, should just be impossible!

One of the tests is to get the client to retrieve a URL where the domain name part of the URL only has an IPv6 address. The idea behind passing this form of URL to the client is to see if the client can fetch the URL. If the client successfully performs the fetch, then we can infer that the client has a working IPv6 protocol stack, is prepared to perform queries for IPv6 addresses in the DNS, has a local interface configured with a working IPv6 address, and is connected on a globally routed IPv6 address prefix. And, to the extent that there is IPv6 in the Internet that's what we see most of the time. If the client has IPv6, and has a working IPv6 network connection, then the client will fetch this URL using IPv6. How can we tell? Because the URL is retrieved from a server that is part of the experimental setup, and on that server we are able to monitor all the traffic at the server.

Here is a typical example of what we see at the server in response to an IPv6 test URL:

```
10-May-2013 00:06:42
    Client: 2404:1800:210:0::xxx
    Status: 200 (OK)
    URL:    http://t10000.u1368144404413.s1385474657.i647302.v10a.r6.td.labs.apnic.net/
                    1x1.png?t10000.u1368144404413.s1385474657.i647302.v10a.r6.td
```

The log is reporting that a client using IPv6 successfully retrieved the URL. The DNS part of this domain name is `t10000.u1368144404413.s1385474657.i647302.v10a.r6.td.labs.apnic.net`, which is covered by a wildcard DNS name in the authoritative DNS zone, of the form `*.r6.td.labs.apnic.net`.

Because the authoritative DNS server is also located on the same server, and because each client is given a unique string in the DNS name used in the URL, we can also see the associated DNS query, which in this case happened 4 seconds prior to the logged URL retrieval:

```
10-May-2013 00:06:41.755
    Client: 203.114.168.2#14841
    RR:     A
    Flags:  -EDC (Recursion Disabled, ENDS0, DNSSEC OK, Checking Disabled)
    Query:  t10000.u1368144404413.s1385474657.i647302.v10a.r6.td.labs.apnic.net

10-May-2013 00:06:41.817
    Client: 203.114.168.2#36126
    RR:     AAAA
    Flags:  -EDC (Recursion Disabled, ENDS0, DNSSEC OK, Checking Disabled)
    Query:  t10000.u1368144404413.s1385474657.i647302.v10a.r6.td.labs.apnic.net
```

In this case the client generated two queries, one for an IPv4 address (A) and one for an IPv6 address (AAAA) for this domain name.

We can delve a bit deeper in the server's logs and get the complete set of interactions at the packet level. The set of transactions seen at the server to retrieve this URL is as follows:

```
Time (ms) Client                 Server

   0      DNS Query: A? ➔
   0                           ⬅ DNS A Response: 0 Records
  62      DNS Query: AAA ➔
  62                           ⬅ DNS AAA Response: 2401:2000:6660::24
 118      HTTP (v6) SYN ➔
 118                           ⬅ HTTP (v6) SYN+ACK
 261      HTTP (v6) ACK  ➔
 262      HTTP (v6) GET 1x1.png ➔
 262                           ⬅ HTTP (v6) ACK
 263                           ⬅ HTTP (v6) (1x1.png)
 406      HTTP (v6) ACK ➔
5165                           ⬅ HTTP (v6) FIN + ACK
5205      HTTP (v6) ACK ➔
5206      HTTP (V6) FIN + ACK ➔
5206                           ⬅ HTTP (v6) ACK
```

We test some 800,000 to 1,000,000 clients each day. If the client has an IPv6 protocol stack and a working IPv6 connection it will follow the DNS phase with an HTTP session, and for some 30,000 clients that we see each day this is the case. Otherwise, the client is acting as if it were an IPv4-only system, in which case it will only query for the A record, and receive a "no such resource record found" response from the DNS. The IPv4-only client cannot then proceed to the fetch phase of the test, as the URL is only accessible using IPv6. This latter behaviour corresponds to the behaviour we observe for the other 770,000 to 970,000 clients each day.

But each day we also see a quite small set of clients that do something entirely different. Here's one such case:

```
14-May-2013:14:38:04
    Client: 173.178.235.xxx
    Status: 200 (OK)
    URL:    http://t10000.u1368542281649.s1235962557.i647302.v10a.r6.td.labs.apnic.net /
                    1x1.png?t10000.u1368542281649.s1235962557.i647302.v10a.r6.td
```

What has changed here is that the client used an IPv4 address to retrieve the URL. In other words the client believes that this IPv6-only URL is accessible using IPv4.

But that should be impossible!

How did the client find an IPv4 address for this IPv6-only URL?

This is a wildcard domain name – here's the DNS zone file for `r6.td.labs.apnic.net`.

```
;       zone contents

$ORIGIN r6.td.labs.apnic.net.

v6 5 IN  AAAA     2401:2000:6660::24

; wildcard

*  5 IN  AAAA     2401:2000:6660::24
```

There is no A record in this zone file. And there is certainly nothing in the DNS to indicate that there is any IPv4 address associated with this domain name.

What did this client query for?

```
14-May-2013 14:38:02.092
    Server: dnsd.labs.apnic.net
```

```
        Resolver: 70.80.36.185#44386
        RR:     A
        Flags:  - (recursion disabled)
        Query:  t10000.u1368542281649.s1235962557.i647302.v10a.R6.TD.laBS.ApNIC.net.
        Response: 0/1/0 ns: R6.TD.laBS.ApNIC.net. NS dnsrd.laBS.ApNIC.net.

    14-May-2013 14:38:02.619
        Server: dnsrd.labs.apnic.net
        Resolver: 70.80.36.185#46857
        RR:     A
        Flags:  - (recursion disabled)
        Query:  t10000.u1368542281649.s1235962557.i647302.v10a.R6.TD.laBS.ApNIC.net.
        Response: 0/1/0 ns: R6.TD.laBS.ApNIC.net. SOA R6.TD.laBS.ApNIC.net. ggm.ApNIC.net.
                  2013051101 10800 3600 604800 10800
```

So we see a part of the client walking "down" the DNS hierarchy, asking each server in turn for the DNS name its trying to resolve. The authoritative name servers for the parent domains provide the name server records for the child zone, so in this case the first query shows the DNS resolver client querying the name server for the `td.labs.apnic.net` DNS zone. The response from the name server from the parent domain is that it does not have the answer to the query, but the response provides the name server for the `r6.td.labs.apnic.net zone`, namely `dnsrd.labs.apnic.net`. The DNS resolver client then sends the same original query to this name server. Because the query is for an IPv4 address (an 'A' resource record query), the response from the authoritative name server is the SOA field for the `r6.td.labs.apnic.net` zone file, indicating that no such resource record can be found in this zone. So the DNS did not appear to provide an IPv4 address to the client.

And what did we see in the web server log?

```
    14-May-2013 14:38:04.180
        Client: 173.178.235.xxx
        Server: 203.133.238.24
        Status: 200 (OK)
        URL:    http://t10000.u1368542281649.s1235962557.i647302.v10a.r6.td.labs.apnic.net
                       /1x1.png?t10000.u1368542281649.s1235962557.i647302.v10a.r6.td
        Browser: Mozilla/5.0 (Macintosh;.Intel.Mac.OS.X.10_5_8) AppleWebKit/534.50.2
                       (KHTML,.like.Gecko).Version/5.0.6.Safari/533.22.3
```

So, somehow, the client has selected the IP address of 203.133.238.24 to use for this URL, and proceeded with a HTTP fetch.

> Actually, the server answered the query anyway. As part of this experiment we ran up a single Apache web server and set it up to listen on all bound IPv4 and IPv6 addresses. The distinction in the URLs between IPv4-only, IPv6-only and Dual Stack was a distinction made by the DNS.

This was not an individual aberration from a single client. In a 24 hour period for the 14th May 2013 there are 150 such instances of this kind of fetch. In the 14 months since the 1st March 2012 we've seen 77,450 such cases where IPv4 has been used to fetch a URL that only has an IPv6 address associated with its domain name. To put this into perspective, these 77,450 cases represent 0.016% out of the total of 487,842,549 individual IPv6 tests that were performed over this same period from the 1st March 2012 until mid-May 2013.

These 77,450 cases involve 31,678 unique source IPv4 addresses. There is no obvious clustering of these clients. Their IP addresses geo-locate to 136 countries, and the prevalence in each country appears to be associated with the number of IPv6 tests performed in each country. These same client addresses are associated with 2,259 unique origin networks (by Autonomous System Number), and in each case we see a significantly greater number of clients from each of these networks that do not behave in this rather odd manner. It does not seem to be something related to the behaviour of equipment used by a set of Internet Service Providers, otherwise some AS's would have a far higher rate of this anomalous behaviour. From this data its reasonable to suspect that the issue is closer to the client than the service provider.

Is it the browser or the operating system platform?

We collect the user agent strings from all clients, and in the set of clients who behave in this odd manner we see almost all browsers. On the Microsoft side we see clients using all published versions of Internet Explorer from 5.0 through to 10.0. Similarly, we see Safari, Chrome, Firefox, Opera, and browsers from Samsung, SonyEricsson, Vodaphone and ZTE. The distribution of clients is much the same as the distribution of clients in the larger set of all clients who performed this IPv6 test. So it's not obvious that this behaviour is the outcome of a browser-related problem for any particular browser or browser/operating system pairing.

Is it the DNS?

At the authoritative name server, the query for an A RR from the authoritative name server for the `r6.td` domain receives a "no such RR" response. But is the DNS resolver client substituting a V4 address back to the end client? To some extent this is a harder question to answer, as the authoritative name server does not get to see the actions of all the DNS resolvers in a forwarder chain. If a DNS resolver uses a forwarder, then the authoritative name server will only see the forwarded query, and not the original query. But let's see if there is a discernable pattern in the DNS resolver behaviour. Are the set of clients who have managed to substitute a IPv4 address for a IPv6 address all clustered behind a small number of DNS resolvers? Taking one particular day, the 14th May, we saw 150 tests that resulted in a fetch of an IPv6-only URL using IPv4. These clients used 98 different DNS resolvers. All of these resolvers were used by other clients on that day, and these other clients did not fetch the IPv6-only URL using IPv4. It's reasonable to conclude that this behaviour is not the outcome of a systematic problem with those DNS resolvers that are directly visible to authoritative nameservers. So while we can't completely rule out an errant DNS resolver, we appear to be able to say that its unlikely. Its also unlikely that its an errant cached entry, as for each of the cases where the client used IPv4 to fetch the URL we saw a query at the authoritative DNS name server.

So if it's not the browser or the operating system and its not the DNS than what's left?

Perhaps we can gain some insight from the particular IPv4 addresses used by these clients when they are fetching the IPv6-only object. What else refers to this particular IPv4 address within the scope of this experiment? To assist here we used distinct IPv4 addresses for each part of the experiment, using a different IP address for the authoritative DNS name servers, and a different IPv4 for each part of the experimental setup (the dual stack test and the IPv4-only test). What we have observed is that consistently we see a sequence of fetches from the client in the order:

```
http://t10000.u1368652592154.s1800083713.i245974.v10i.r4.td.labs.apnic.net/1x1.png
http://t10000.u1368652592154.s1800083713.i245974.v10i.rd.td.labs.apnic.net/1x1.png
http://t10000.u1368652592154.s1800083713.i245974.v10i.r6.td.labs.apnic.net/1x1.png
```

The IPv4 address used in the last fetch when the client erroneously loads the IPv6-only URL over IPv4 was consistently that of the preceding URL, namely the IPv4 address that is bound to the wildcard `*.rd.td.labs.apnic.net`.

Perhaps a possible explanation here is some form of consumer modem or similar which includes a DNS proxy function. It is possible that when the DNS resolver in the device cannot obtain the requested value, it provides the most recently provided value instead. It may also be the case that a proxy DNS resolver in maintains a local cache of DNS answers, but the lookup key in the case is limited to around 40 characters. As the URLs in this test are quite long, and are the same for the first 49 characters, such a matching algorithm using limited length lookup keys could be confused into thinking that this is the same DNS name, and the locally cached value substituted in place of the null answer received from the external DNS resolver. Its not a totally satisfactory explanation, but there are few other consistent clues as to why this is happening across such a broad set of operating systems, browsers, locales and DNS resolving domains.

But that's not all we've seen that is supposedly impossible from these tests. As part of this test we were wanting to see how many clients had an active IPv6 protocol stack, and in this measurement we were wanting to include Windows platforms that used the NAT traversal auto-tunnelling mechanism of Teredo. However if a system only has Teredo and no other IPv6 connection then Windows will not query the DNS for an IPv6 address, thereby leaving the Teredo protocol stack essentially dormant. But there is one way tickle this IPv6 protocol stack into life, and that's by giving the client a URL where the hostname is not a DNS name, but a literal IPv6 address.

The URL we use in this case places a IPv6 address inside [] braces:

```
http://[2401:2000:6660::f003]/1x1.png?t10000.u1368537153656.s118810765.ianon.v10c.v6lit
```

The intended result is that upon presentation of this URL the client will initiate a TCP connection to port 80 of `2401:2000:6660::f003`. If the client does not support IPv6, or has no working IPv6 connection, then the URL will not be fetched. Or so we thought.

What we see in the web logs is intriguing:

```
15-May-2013:20:52:02
    Client: 118.148.0.xxx
    Server: 203.133.248.45
    Status: 200 (OK)
    URL:    http://[2401:2000:6660::f003]/
                    1x1.png?t10000. u1368652083526.s2013711009.i647302.v10a.v6lit
```

This client has used an IPv4 connection to retrieve this URL. How did the client pick this particular IPv4 address? There is no DNS query associated with this retrieval. The IP address used is 203.133.248.45, which is that used by the `xxx.r4.td.labs.apnic.net`. (IPv4-only URL).

What happened here at the packet level?

```
# TCP Handshake
20:52:02.645511   118.148.0.xxx:53247 → 203.133.248.45:80  Flags [SYN]
20:52:02.645529                         ←                  Flags [SYN+ACK]
20:52:02.687948                         →                  Flags [ACK]

# initial HTTP GET of IPv4 object
20:52:02.688254                         →                  Flags [PUSH+ACK]
                 Payload: GET http://t10000.u1368652083526.s2013711009.i647302.
                          v10a.r4.td.labs.apnic.net/1x1.png?t10000.u1368652083526.
                          s2013711009.i647302.v10a.r4.td
20:52:02.688270                         ←                  Flags [ACK]

# Server Response: image file
20:52:02.688597                         ←                  Flags [PUSH+ACK]
                 Payload: HTTP/1.1 200.OK
                          Content-Length: 157
                          Content-Type: image/png
20:52:02.731497                         →                  Flags [ACK]


# second HTTP GET of IPv6 literal object
20:52:02.939605                         →                  Flags [PUSH+ACK]
                 Payload: GET http://[2401:2000:6660::f003]/1x1.png?t10000.
                          u1368652083526.s2013711009.i647302.v10a.v6lit
20:52:02.939620                         ←                  Flags [ACK]

# Client RESETS the TCP connection
20:52:02.939621                         →                  Flags [RESET]

# Server ignores the Client RESET and responds with image file
20:52:02.939908                         ←                  Flags [PUSH+ACK]
                 Payload: HTTP/1.1 200.OK
                          Content-Length: 157
                          Content-Type: image/png

# Server retransmits the last unacknowledged TCP segment 5 times (130 ms interval)
20:52:03.268463                         ←                  Flags [PUSH+ACK]
20:52:03.726462                         ←                  Flags [PUSH+ACK]
20:52:04.442487                         ←                  Flags [PUSH+ACK]
20:52:05.674465                         ←                  Flags [PUSH+ACK]
20:52:07.938470

# Server sends a FIN
20:52:07.940497                         ←                  Flags [FIN+ACK]

# Server retransmits the last unacknowledged TCP segment 7 times (exponential backoff)
20:52:11.210466                         ←                  Flags [PUSH+ACK]
```

```
20:52:17.554466                 🡐                  Flags [PUSH+ACK]
20:52:30.042467                 🡐                  Flags [PUSH+ACK]
20:52:54.818466                 🡐                  Flags [PUSH+ACK]
20:53:44.170467                 🡐                  Flags [PUSH+ACK]
20:54:33.522466                 🡐                  Flags [PUSH+ACK]
20:55:22.874465                 🡐                  Flags [PUSH+ACK]

# Server times out and RESETS the connection
20:56:12.226465                 🡐                  Flags [RESET]
```

This is a certainly curious. The client appears to be pipelining HTTP requests, and after a successful GET for the IPv4-only object, the client then passes a GET request for the IPv6 literal down the already open TCP session. However, in the case shown above, as soon as the server acknowledges this pipelined GET request, the client sends a TCP RESET and appears to close the connection. The server appears to ignore this reset and sends the response in any case. The server then attempts retransmission of this TCP segment a further 12 times over the ensuing 248 seconds, before sending a RESET of its TCP side of the connection (which appears to be a TCP retransmit function that is related to an ACK reception timeout). So in this case its not exactly the case that the client has loaded the V6 literal URL over V4, but something is strange with the pipeline control of the web system at the client's end of the connection.

> Something is also somewhat broken about the behaviour of our Apache 2.2 server. The received RESET should've caused the connection to break immediately, and the server should not have attempted to send the response, and should not have logged this as a successful delivery in its logs. But that's a different problem!

Is this form of behaviour always the case in these strange anomalies?

In one day, the 15th May, we saw 9 clients perform this feat. Of these, 5 are located in AS 38793 (New Zealand), NZCOMMS-AS-AP Two Degrees Mobile Limited, and the consistency with which this particular AS is an origin AS in this anomalies tends to suggest that there is some proxy web service in this AS that is being challenged by the use of IPv6 literals. All the fetches from this particular origin AS look like the above example, namely a pipelined request with a TCP reset, which tends to reinforce the view that this is some form of web proxy behaviour we are observing here. The other four are located in AS 4134 (China), AS 4837 (China), AS 45671 (Australia) and AS 4739 (Australia). Three of these are similar to what is shown above, namely that a TCP session is opened, the client fetches either the r4.td or the rd.td object, then in the same TCP session attempts a fetch of the IPv6 literal URL, which seems to be some kind of errant behaviour of a local proxy web cache in a combined local firewall / web cache unit in the way in which it handles literal IPv6 addresses. In the other case the client opens up a new TCP session to fetch the IPv6 literal, but uses the most recently used IPv4 address, namely the address of the `rd.td` URL in this case. Again, I'd guess this is a similar base problem, that of a local web proxy not understanding the syntax of an IPv6 literal and just reusing the most recent IPv4 address it used in its place.

This is not a widespread problem by any means. We see between 10 and 25 instances per day of this form of cross-protocol fetch in the course of the larger IPv6 test program. In the period since the 1st March 2012 we saw 7,463 such instances, involving 3,644 unique source addresses, or 0.0015% of the client set. These 3,644 client IPv4 addresses were from 304 different origin ASs. The most likely explanation here is that there is some item of consumer equipment that includes a web proxy function, and this equipment had, or still has, a software bug that does not correctly handle URLs that use the IPv6 Literal address format. When presented with a URL of this form, then in its confusion, or perhaps in desperation to do something, it simply reuses the most recently used IPv4 address to perform the fetch anyway!

This is illustrative of the evolving nature of the Internet. These days it's not only a case of understanding the behaviour of the various applications that sit within the end devices at either end of a connection, but it's also critical to understand the behaviour of the middleware that sits on the wire. It's often the case that the lowest cost and most poorly maintained device in the entire chain is the consumer’s interface to the network, a device that may cost as little as $40, but contains a firewall filter, DNS resolver, and a web proxy engine in addition to the modem and router functionality. Its also a device that rarely gets upgraded after sale, so even if the vendor has located and rectified bugs in the code, its not uncommon to see the overall majority of the deployed devices still running the original code. So, evidently, no, these strange cross-protocol mix ups are no longer impossible. Thanks to some pretty ordinary middleware out there they are indeed quite possible. I suppose that the good news so far is that the incidence of this aberrant form of behaviour is really quite low. So far.

Disclaimer

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