The ISP Column
A monthly column on things Internet

November 2009

Geoff Huston

~~Dateless~~ Stateless and ~~Des~~DNSperate !

I've often heard it said that the world is full of bad ideas. But no matter how many bad ideas there may be, the good news is that there is always room for one more! So in the spirit of "more is better" I'd like to offer the following as yet another Bad Idea (http://bert.secret-wg.org/BIF/index.html). There is also the intriguing possibility that this flawed concept could be made to work, making this a Useless Tool (http://bert.secret-wg.org/Tools/index.html) at the same time!

If you look at a standard description of the IP protocol stack, then at the level above the Internetworking level of IP there is the end-to-end transport level. There are now a number of end-to-end transport protocols, but the most commonly used ones are the User Datagram Protocol, or UDP, and the Transmission Control Protocol, or TCP.

UDP is, as its name suggests, a stateless datagram protocol. When an application passes data to a UDP transport interface there is no assurance that the intended recipient will receive any data at all. Despite the rather tenuous nature of the communication there is a very real role for UDP in networking transactions. The common use for UDP is in very lightweight transactions on the form of a simple query and an equally direct and immediate response. There is also a role for such a low overhead protocol in real time streaming applications where dropped data does not require a stop and restart operation.

TCP is more or less the opposite, in that when an application requires a reliable data transaction then TCP comes into its own. However, TCP is not a lightweight protocol like UDP, in that it requires time to both set up and tear down the connection. TCP requires the completion of a rendezvous phase before any data transmission can take place. All sent data must be positively acknowledged by the receiver before the sender can consider that the data can be "sent." The sender will, as necessary, retransmit data if it appears that the original transmission has been dropped in transit. Also, the sender can only have a certain amount of outstanding unacknowledged data before it stalls and awaits for acknowledgement from the receiver. The TCP session also requires a "FIN handshake" to close down the connection in each direction, adding a further time overhead to the total transaction time.

These two transport protocols have their distinct areas of application, and their respective weaknesses. For bulk data transfer where integrity of the data is important, then TCP comes into its own. For data transfer operations involving large quantities of data the overhead of the opening and closing packet exchanges is negligible, and TCP is capable of controlling the transfer at a rate that is as fast as the combination of the sender, receiver and the intervening network path can sustain. For simple client / server transactions, where the client makes a small request and expects the server to deliver back an immediate short response, then TCP can be seen as a more cumbersome approach than UDP. Not only does TCP require more than a round trip time to complete the rendezvous handshake before the request can even be sent to the server, it also requires time to release the TCP connection at the end of the transaction. The other aspect of this is that both the client and the server need to maintain their TCP connection state for the lifetime of the connection, consuming resources on the server. For busy servers that are processing simple relatively short transactions, such as DNS queries, TCP is not the most obvious choice for a transport protocol. UDP is an obvious candidate for short, simple transactions. By "short" I mean anything that can comfortably site into a single UDP packet.

But what about transactions that are larger than "short", yet still not that large. What about, say, a query to a DNSSEC-signed root zone of the DNS?

```
This is the result of a query to a DNSSEC-signed zone

$ dig @mirin.rand.apnic.net 192.in-addr.arpa. in any
;; Truncated, retrying in TCP mode.

; <<>> DiG 9.6.0-APPLE-P2 <<>> @mirin.rand.apnic.net 192.in-addr.arpa. in any
; (3 servers found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 64484
;; flags: qr rd ra; QUERY: 1, ANSWER: 18, AUTHORITY: 7, ADDITIONAL: 6

;; QUESTION SECTION:
;192.in-addr.arpa.         IN      ANY

;; ANSWER SECTION:
192.in-addr.arpa.  86397    IN      SOA     chia.arin.net. dns-ops.arin.net.
2009102917 1800 900 691200 10800

[remainder of answer section removed]

;; Query time: 62 msec
;; SERVER: 2001:dc0:2001:7:2c0:9fff:fe44:b207#53(2001:dc0:2001:7:2c0:9fff:fe44:b207)
;; WHEN: Fri Oct 30 10:33:30 2009
;; MSG SIZE  rcvd: 2260
```

In this case the original UDP response was truncated to a 451 octet response, and the client then re-posed the request to the server using TCP eliciting the complete 2,260 octet response. This then poses an interesting question: as DNSSEC deployment gathers pace will we see the use of TCP for DNS queries become more prevalent? And if that’s the case are DNS servers capable of sustaining the same response throughput using TCP as they are using UDP? Is TCP posing some real limitation here?

There is a related issue here, when looking at the use of IPv6 as the transport protocol for the UDP DNS query with ENDS0.

Because IPv6 UDP does not support fragmentation in flight, if the path MTU between the server and the DNS client is less than the IPv6 interface MTU at the server, then the server will emit a large UDP packet matching the local interface MTU. This will cause the packet to be dropped at the path MTU bottleneck, and an ICMPv6 "packet too big" response sent back to the server. The server has no remembered state of the original query, and cannot reformat its response. The client will not receive any response to its original query, and will time out. In such a case, the client turning to any other server may not help either. If all servers are delivering the same large response via UDP and there is a path MTU blackhole close to the client, then the client may well have a problem!

It’s a bit of a "Goldilocks problem", where it seems that there are scenarios where neither TCP nor UDP provide "just the right" behaviour. TCP imposes higher overheads in terms of server load and longer transactions times, and increased vulnerability of the server to TCP-based attacks. UDP has limitations in terms of fragmentation handling for large UDP packets, firewall treatment of fragments and lingering issues with IPv6 UDP handling of path MTU mismatches.

Stateless TCP?

So, now for the bad idea: is it possible to create a "stateless" TCP server, which looks to a TCP client like it is a conventional TCP service over at the server, but the server treats its client's transactions requests in a manner very similar to a UDP-based service?

So to see if this is possible, lets look at a "conventional TCP transaction from the perspective of the client:

```
client.53910 > server.http: Flags [S], seq 1237395672, length 0
server.http > client.53910: Flags [S.], seq 3565371242, ack 1237395673, length 0
client.53910 > server.http: Flags [.], ack 1, length 0
client.53910 > server.http: Flags [P.], seq 1:248, ack 1, length 247
server.http > client.53910: Flags [P.], seq 1:468, ack 248, length 467
server.http > client.53910: Flags [F.], seq 468, ack 248, length 0
client.53910 > server.http: Flags [.], ack 468, length 0
client.53910 > server.http: Flags [.], ack 469, length 0
client.53910 > server.http: Flags [F.], seq 248, ack 469, length 0
server.http > client.53910: Flags [.], ack 249, length 0
```

This is a tcpdump report, showing each packet in sequence. The first fields show the source and destination of the packet, showing the name of the party and the port address. The next field shows the TCP flags, and the acknowledgement and sequence numbers. If there is a payload attached, the length field shows the length of the payload.

This is a pretty standard HTTP request. So see the components, let break up this it’s three phases of startup, request/response and shutdown

1. The initial 3-way TCP handshake

```
client.53910 > server.http: Flags [S], seq 1237395672, length 0
server.http > client.53910: Flags [S.], seq 3565371242, ack 1237395673, length 0
client.53910 > server.http: Flags [.], ack 1, length 0
```

The client starts the session with a SYN packet directed to the server, the server responds with a SYN + ACK packet, to which the client responds with an ACK. The zsession is now set up.

2. The request and response

```
client.53910 > server.http: Flags [P.], seq 1:248, ack 1, length 247
server.http > client.53910: Flags [P.], seq 1:468, ack 248, length 467
server.http > client.53910: Flags [F.], seq 468, ack 248, length 0
client.53910 > server.http: Flags [.], ack 468, length 0
```

The client sends the request. The server sends its response, and as the server has no more to send, it terminates its sending end (but not the listener) with a FIN packet. The client acknowledges receipt of the response data with an ACK.

3. client end shutdown

```
client.53910 > server.http: Flags [.], ack 469, length 0
client.53910 > server.http: Flags [F.], seq 248, ack 469, length 0
server.http > client.53910: Flags [.], ack 249, length 0
```

The client acknowledges receipt of the server's FIN, and then indicates it is closing down by sending a FIN to the server, who, in turn responds with its acknowledgement.

Its clear that in this form of transaction, in each case the server's actions are triggered by an incoming TCP packet from the client. The "rules" that describe the server's behaviour are:

- An incoming SYN packet generates a SYN+ACK response from the server
- An incoming packet with a data payload generates a data response from the server, followed by a FIN
- An incoming FIN packet generates an ACK response from the server
- All other incoming packets generate no server response at all.

This provides the essential elements of a "stateless" TCP server.

I should note that this "stateless" approach has a number of limitations. The requests must fit into a single TCP packet, so that the stateless TCP server does not have to enter a stateful wait mode to assemble all the packets that relate to a single request. Similarly, the response has to be sufficiently small that it will fit into the receiver's window, so that the server does not have to hold on to part of the response and await incoming ACKs from the client before sending the entirety of the response.

Turning a Bad Idea into a Useless Tool

The next challenge is to code up this bad idea. I'm using a FreeBSD platform to try this out. The immediate problem here is that FreeBSD, like most Unix platforms, is remarkably helpful or unhelpful depending on your perspective, and has decided that all IP, TCP and UDP functions should occur in the kernel of the operating system. So if you want to write some user code to perform such packet level operations then you have to be a little inventive.

1. Capturing "RAW" IP

The first task is to get the kernel to "release" an incoming TCP packet to my application in its entirety without sending it through the kernel's own IP and TCP processing path. As I've no great desire to build a new kernel to test out this idea all thoughts of modifying the TCP/IP source code in the kernel are out of bounds! So as long as I'm prepared to modify my original requirement such that my application gets to receive all incoming packets and allow the kernel to also attempt to process them at the same time, then I think I can make some progress here.

The *tcpdump* package, originally written by Van Jacobson, Craig Leres and Steven McCanne, while they were all oatthe Lawrence Berkeley National Laboratory, University of California, Berkeley, looks to do precisely what I'm after. In particular, the *pcap capture library* allows a user level program to capture a copy of incoming packets that have been received by the local host. *Pcap* allows the allocation to set up a filter expression to specify exactly which packets are of interest to the application. So if my interest is to make a lightweight stateless http server, then I'm interested in capturing all TCP packets with a destination port address of port 80 and a destination IP address of this host.

The *pcap* interface allows the application to define a packet handler which is invoked each time a packet is assembled that matches the filter expression.

So I'm almost there with capturing the raw IP packets.

2. Disabling Kernel Processing

One would think that as long as there are no daemons that have invoked a *listen* call to as associated with completed TCP handshakes for a given port number, and as long as *inetd* or your local equivalent hasn’t decided that it wants to managed this port address then the kernel would simply ignore the TCP incoming packet.

FreeBSD is certainly more helpful than that, and by default it likes to send a TCP RESET packet (a TCP packet with the RST flag set) in response to all incoming TCP packets that are addressed to unassociated ports. I need to stop this behaviour, and there is a kernel parameter that can be toggeled that will do precisely this:

```
# sysctl net.inet.tcp.blackhole=2
```

This raises an interesting question about good security practice with online servers. Generally its better not to advertise un-associated ports. The information being leaked by this advertisement is the implication is that those ports that do not generate such a response are then "live" ports that are conceivably targets for an attack.

The kernel's default behaviour is to send a TCP RST in response to received TCP packets addressed to un-associated ports, and an ICMP Port unreachable packet in response to incoming UDP packets addressed to un-associated UDP ports.

To change the kernel's default behaviour it is necessary to add the following to /etc/sysctl.conf:

```
net.inet.udp.blackhole=1
net.inet.tcp.blackhole=2
```

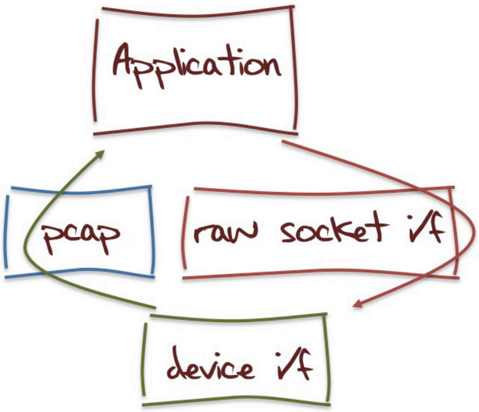
(http://www.freebsdblog.org/51/reduce-server-visibility/)

That’s the input side done. How about generating TCP packets where the TCP fields are constructed by the application, and not by the kernel's own TCP module?

3. Generating "RAW" IP packets

This is quite straightforward. The *socket* call allows the application to open a SOCK_RAW socket type, which corresponds to a "raw" IP socket. The kernel will still want to have a swipe at the IP header on the way out, but we can stop that too by telling the kernel that the application will also be looking after the IP header, with a set socket option call that sets the IP_HDRINCL flag for this socket.

The result is an application that is configured for I/O as follows:



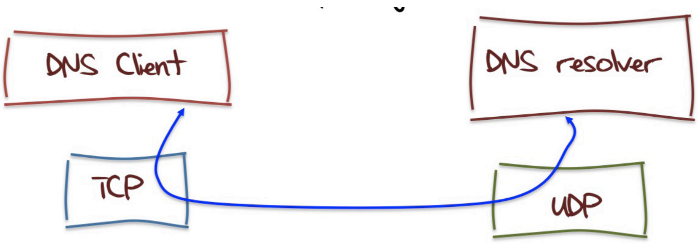
To provide a proof of concept of this approach I've been playing with such a "stateless" web server. This server responds to incoming http requests using this "stateless TCP" concept, using a "backend" conventional TCP connection to a real TCP server to generate the response (in this case the web page corresponding to the original request)

Here's a trace of the "stateless" WEB server delivering a simple small web page. Larger pages generate a back-to-back sequence of data packets.

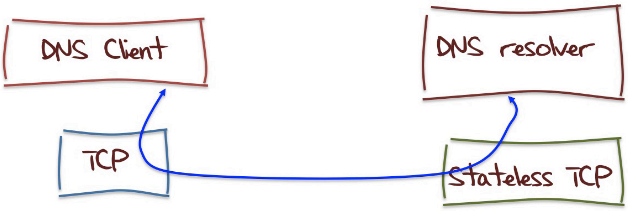
```
[the client sets up the connection with the stateless web server using the
TCP 3 way handshake]

IP client.55371 > server.http:  S 926841556:926841556(0) win 65535 <mss 1460>
IP server.http > client.55371:  S 1256778255:1256778255(0) ack 926841557 win 65535
                                  <mss 1220>
IP client.55371 > server.http:  . ack 1 win 65535

[The client sends its http request]

IP client.55371 > server.http:  P 1:246(245) ack 1 win 65535

[the stateless server immediately ACKs this request, in order to prevent
the client timing out and retransmitting the request segment]

IP server.http > client.55371:  . ack 246 win 65535

[the client now turns to the back send server and resends the client's
query, using conventional TCP. This trace shows the three way handshake,
the request and response]]

IP server.56447 > backend.http: S 3055447836:3055447836(0) win 65535 <mss 1460>
IP backend.http > server.56447: S 2086147938:2086147938(0) ack 3055447837 win 65535
                                  <mss 1460>
IP server.56447 > backend.http: . ack 1 win 8326
IP server.56447 > backend.http: P 1:248(247) ack 1 win 8326>
IP backend.http > server.56447: P 1:468(467) ack 248 win 8326

[the server sends the backend's response to the client]

IP server.http > client.55371:  . 1:468(467) ack 246 win 65535

[the backend and the server close down their TCP session]

IP backend.http > server.56447: F 468:468(0) ack 248 win 8326
IP server.56447 > backend.http: . ack 469 win 8326
IP server.56447 > backend.http: F 248:248(0) ack 469 win 8326
IP backend.http > server.56447: . ack 249 win 8325

[the server and the client now perform a similar close of the stateless
TCP session]

IP server.http > client.55371:  F 468:468(0) ack 246 win 65535
IP client.55371 > server.http:  . ack 468 win 65535
IP client.55371 > server.http:  . ack 469 win 65535
IP client.55371 > server.http:  F 246:246(0) ack 469 win 65535
IP server.http > client.55371:  . ack 247 win 65535
```

Applying Bad Ideas to the DNS

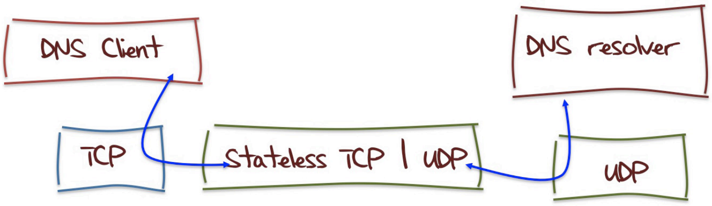
So far so good, but that's just HTTP. What about the DNS? The idealised model here is to see if its possible to create a hybrid combination of server and client where the server is using UDP and the client is using TCP. The client uses TCP to perform segment reassembly rather than rely on IP packet fragment reassembly, while the server uses the stateless UDP service model to allow for the higher service capacity.



Maybe that’s just such a terribly bad idea that its not possible to construct such a hybrid approach, but what about if we introduce stateless TCP into the mix?



This has much the same properties of the hybrid model, and just could be coerced into working. But then again I'm not keen to start hacking around inside an implementation of a DNS server, so the fastest way to prototype this approach is to put in a TCP / UDP translator between the client and the server:



The stateless TCP / UDP DNS proxy server pick up clients' DNS queries in TCP and directs them via UDP to a DNS resolver, and provides the response again using TCP.

```
[the TCP-based client DNS query]
$ dig +tcp @server rand.apnic.net in any


[The client TCP sync handshake, DNS query and ACK of the query]
```

```
client.55998 > server.domain: S, cksum 0x9159 (correct), 2201103970:2201103970(0)
                                 win 65535 <mss 1460>
server.domain > client.55998: S, cksum 0x82b9 (correct), 1256795928:1256795928(0)
                                 ack 2201103971 win 65535 <mss 1220>
client.55998 > server.domain: ., cksum 0x9986 (correct), 1:1(0) ack 1 win 65535
client.55998 > server.domain: P, cksum 0x41b2 (correct), 1:35(34) ack 1 win 65535
                                 30304+ ANY? rand.apnic.net. (32)
server.domain > client.55998: ., cksum 0x9964 (correct), 1:1(0) ack 35 win 65535

[The stateless server rephrases the query as a UDP query against a backend
DNS  server, and collects the response]

server.54054 > backend.domain: 30304+ ANY? rand.apnic.net. (32)
backend.domain > server.54054: 30304* q: ANY? rand.apnic.net. 6/0/2 rand.apnic.net.
                               SOA mirin.apnic.net. research.apnic.net.
                               2009051502 3600 900 3600000 3600, rand.apnic.net. NS
                               mirin.apnic.net., rand.apnic.net. NS sec3.apnic.net.,
                               rand.apnic.net. MX kombu.apnic.net. 100,
                               rand.apnic.net. MX karashi.apnic.net. 200,
                               rand.apnic.net. MX fennel.apnic.net. 300 ar:
                               sec3.apnic.net. A sec3.apnic.net, sec3.apnic.net.
                               AAAA sec3.apnic.net (229)

[The response is then passed back to the client via Stateless TCP]

server.domain > client.55998: ., cksum 0x421a (correct), 1:232(231) ack 35 win 65535
                               30304* q: ANY? rand.apnic.net. 6/0/2
                               rand.apnic.net. SOA mirin.apnic.net.
                               research.apnic.net.
                               2009051502 3600 900 3600000 3600, rand.apnic.net.
                               NS mirin.apnic.net., rand.apnic.net. NS
                               sec3.apnic.net., rand.apnic.net. MX kombu.apnic.net.
                               100, rand.apnic.net. MX karashi.apnic.net. 200,
                               rand.apnic.net. MX fennel.apnic.net. 300 ar:
                               sec3.apnic.net. A sec3.apnic.net, sec3.apnic.net.
                               AAAA sec3.apnic.net (229)

[the TCP connection is closed]

server.domain > client.55998: F, cksum 0x987c (correct), 232:232(0) ack 35 win 65535
client.55998 > server.domain: ., cksum 0x987d (correct), 35:35(0) ack 232 win 65535
client.55998 > server.domain: ., cksum 0x987c (correct), 35:35(0) ack 233 win 65535
client.55998 > server.domain: F, cksum 0x987b (correct), 35:35(0) ack 233 win 65535
server.domain > client.55998: ., cksum 0x987c (correct), 232:232(0) ack 36 win 65535
```

It Worked!

Admittedly, this approach has managed to bring the worst of UDP into TCP. There is no reliability, no recovery from dropped segments, no flow control, and pretty much no manners or any form of good protocol behaviour whatsoever!

This is a pretty rough proof of concept and I'm sure that anyone with a good working knowledge of the intricacies of DNS could document a pile of shortcomings, but like all bad ideas that morph into useless tools, the challenge was not to make it work well, but to make it work at all!

Code

The source code for the http and dns stateless tcp servers that I described here can be found at: http://www.potaroo.net/tools/useless/

Acknowledgements

It's often said that success has many parents, while failure, and in this case a bad idea, is an orphan! However, George Michaelson of APNIC has generously owned up to part-parentage of this particularly bad idea.

I also used the following assistance to put the code together.

> "TCP/IP Illustrated, Volume 1," by W. Richard Stevens. My copy dates from 1994, but its still a remarkably useful reference for the TCP/IP protocol suite.
>
> "Unix Network Programming," 3rd Edition, by W. Richard Stevens, Bill Fenner and Andrew M. Rudoff. This book is a really helpful reference for writing client and server code.
>
> "tcpdump and the libpcap library," Van Jacobsen et al, http://www.tcpdump.org

Disclaimer

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

About the Author

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

http://www.potaroo.net