Improving DNS performance using "Stateless" TCP in FreeBSD 9

David Hayes, Mattia Rossi, Grenville Armitage
Centre for Advanced Internet Architectures, Technical Report 101022A
Swinburne University of Technology
Melbourne, Australia
dahayes@swin.edu.au, mrossi@swin.edu.au, garmitage@swin.edu.au

***Abstract*—The introduction of DNSSEC and the increasing adoption of IPv6 will tend to generate DNS responses too large for standard DNS-over-UDP transport. This will create pressure for clients to switch to TCP for DNS queries, leading to a significant increase in overhead for DNS servers. Huston has proposed a "stateless" version of TCP to reduce the server-side load on DNS servers handling DNS queries over TCP. Stateless TCP observes that typical DNS-over-TCP queries may be adequately handled by a simplified TCP connection establishment that reduces the kernel state required per connection. We have implemented our own version of *statelessTCP* under FreeBSD 9 (FreeBSD's current development branch at the time of writing). This report discusses our selected design and implementation, outlines the limitations of other possible alternatives we chose not to implement, and describes preliminary experimental results showing that DNS-over-statelessTCP uses noticeably less server-side resources than regular DNS-over-TCP.**

I. Introduction

DNS clients have traditionally used DNS-over-UDP, falling back to DNS-over-TCP when UDP based queries were unsuccessful. Servers benefit from using UDP as the transport for DNS queries because only a single UDP socket is required to handle queries from everywhere. In contrast, when clients connect via TCP the server must establish and tear-down a new TCP socket per query (with all the associated churn in kernel state), and exchange additional packets during TCP connection set up and tear down. (UDP is also easier for clients to process, but they rarely generate more than a few queries per second so the difference is insignificant.)

The drawback of DNS-over-UDP is that DNS responses must fit into a single UDP packet. The BIND 9 DNS server used in our experiments allows UDP response packets to be limited to between between 512 and 4096 Bytes. However, most most DNS servers are configured to allow only a maximum UDP packet size of 512 Bytes. If a query triggers a response exceeding the maximum UDP packet size, a UDP packet containing the "truncated response" flag will be sent to the client, triggering a repeat query using DNS-over-TCP.

Using IPv6 and DNSSEC, the DNS response will always exceed 512 Bytes. Increasing the maximum UDP packet size to 4096 Byte, would allow such a response to fit within a single UDP packet, but result in additional problems:

- Firewalls and NAT boxes are known to drop DNS UDP packets which are larger than 512 Bytes
- The UDP packet might be larger than the maximum transfer unit (MTU) along the path, and be fragmented at IP level, which poses two additional issues, one concerning IPv4 and IPv6, and one concerning IPv6 only:
  1) Firewalls and NAT boxes are known to drop fragmented UDP packets, causing the UDP based DNS response to be lost. Since UDP does not perform lost packet recovery, the client will reissue the request.
  2) IPv6 does not support middle box fragmentation, but relies on the transport protocol to identify the path MTU and create packets of a size that fits. If the transport protocol sends a packet too large for an intermediate node, it is dropped.

Using TCP would solve the issues of passing through firewalls and NATs, but at the expense of greater resources invested per query by the DNS server.

This report investigates our design and implementation of a "stateless" version of TCP (which we refer to as *statelessTCP*) within FreeBSD 9 (FreeBSD's current development branch at the time of writing). Our statelessTCP will allow DNS servers to efficiently respond to TCP based DNS queries, using significantly less server

resources than standard "stateful" TCP.

We incorporate Huston's idea outlined in [1, 2] into the FreeBSD 9 kernel TCP/UDP protocol stack. The proposed modified TCP stack operates in a "stateless" manner for basic DNS queries, and as standard "stateful" TCP for other types of traffic and DNS axfr (zone transfer) queries that require substantial responses.

We discuss Huston's original userspace implementation of the idea in Section II, explain our selected design in Section III discussing alternatives in Section IV. In Section V we describe the implementation in FreeBSD and provide a comparison between the server-side resources used by DNS-over-statelessTCP, DNS-over-TCP (regular TCP) and DNS-over-UDP in Section VI. We discuss the possible employment of statelessTCP in the real world in Section VII. We conclude in Section VIII.

II. Summary of the original DNS-proxy

In an attempt to mitigate the potential performance issues for DNS servers, if clients need to use TCP as a vehicle for requests instead of UDP, Huston proposed a stripped down "stateless" TCP [1, 2]. To illustrate the idea Huston has published some sample user-space code [3] that implements a DNS proxy. The proxy intercepts TCP based DNS queries, forwarding them to the DNS server via a UDP socket and providing appropriate responses to the TCP packets.

The basic operation of the proxy is shown in Algorithm 1:

Algorithm 1 User-space DNS-Proxy algorithm outline

Capture every TCP packet arriving at port 53 and perform the following operations, using a raw socket:

1) If it is a TCP SYN packet, send back a TCP SYN/ACK
2) If it is a TCP FIN packet, send back a TCP ACK
3) If it is a TCP DATA packet, proceed with Algorithm 2
4) If it is none of the above packet types, drop the packet

Algorithm 2 outlines how the DNS-proxy processes TCP DATA packets arriving at port 53. This algorithm is performed for each such packet.

A. FreeBSD stack implementation issues

While the DNS-proxy implementation outlined the basic idea for a "stateless" TCP and proved to work, the implementation in the FreeBSD kernel of such an

Algorithm 2 TCP-UDP-TCP translation implemented by the DNS-proxy

For each incoming TCP DATA packet on port 53 perform the following steps:

1) Create the IP response packet template:
   - Use a random number for the IP ID field
   - Set the TTL field to 255
   - Initialise the checksum field to 0
2) Create the TCP response packet template:
   - Set the acknowledgement number to the incoming sequence number + data length.
   - Set the TCP ACK flag
   - Set the window size to the maximum of 65535
   - Initialise the checksum field to 0
3) Store the acknowledgement number of the incoming packet. This is used as the sequence number for outgoing packets.
4) Use the IP packet template, and TCP packet template, add the checksum and send it without payload as a TCP ACK, in order to prevent the client to retransmit the packet.
5) Strip the first 2 bytes from the DNS query in the TCP payload, which contain a length field not used in UDP DNS query
6) Send the stripped TCP payload in over a UDP socket to the DNS server
7) Listen on the socket and read the response into a buffer
8) Add the length field to the DNS response at the start of the buffer, as it is needed for TCP DNS responses
9) Depending on the size of the response, create one or multiple TCP packets:
   - For the first TCP packet, set the sequence number to the value stored at 3)
   - For each additional packet, add the size of the previous packets payload to the previous packets sequence number and set the resulting value as sequence number
10) Calculate and set the TCP and IP checksums and send the packet(s)

idea is constrained by the stack only being able to transit packets without waiting for responses.

In the example DNS-proxy implementation, the TCP acknowledgement number (Algorithm 2, Item 2) and all TCP and IP information is stored in a template for the time the UDP DNS look-up is performed. As this is a synchronous, blocking operation (Algorithm 2, Item 7), information is preserved for the necessary time.

In the FreeBSD stack, the TCP and UDP layers receive packets from the IP layer, process them and where necessary pass them up via the socket interface to a listening application. They also process data received from the application and send the packets down to the IP layer. All these operations are asynchronous and non-blocking, and necessary information like TCP sequence numbers and basic information like incoming source address and port pairs and destination address and port pairs (if multihomed) need to be stored, in order to allow the TCP-to-UDP and UDP-to-TCP translation to work asynchronously.

III. Design of StatelessTCP

Our design attempts to implement the key elements of [1] in the FreeBSD 9 stack allowing a TCP DNS client to communicate with the DNS server, but have the server receive the queries and send the requests via a UDP socket. Unlike Huston's DNS-proxy, which translates TCP packets to UDP packets in user-space, we translate the packets completely in kernel-space, passing only UDP packets with DNS queries to the DNS sever which runs in user-space. The same applies for UDP packets generated by the DNS server – the translation to TCP occurs completely in kernel-space. Figure 1 illustrates the basic architecture.

A. TCP data as UDP

A DNS server needs to open a TCP port as well as a UDP port to listen for DNS queries (this is the default behaviour in BIND 9's named server). DNS queries received on the TCP port are handled by statelessTCP in the kernel. StatelessTCP responds to TCP control packets, but sends data packets to the application (often named) via the UDP socket, created at initialisation by the DNS server.

The protocol stack does not wait for a UDP response from the application, so it is unable to easily map the packet to TCP using a previously constructed TCP template, as was possible in the DNS proxy example. Instead it will need to be able to identify the response sent by the application and redirect it to statelessTCP.

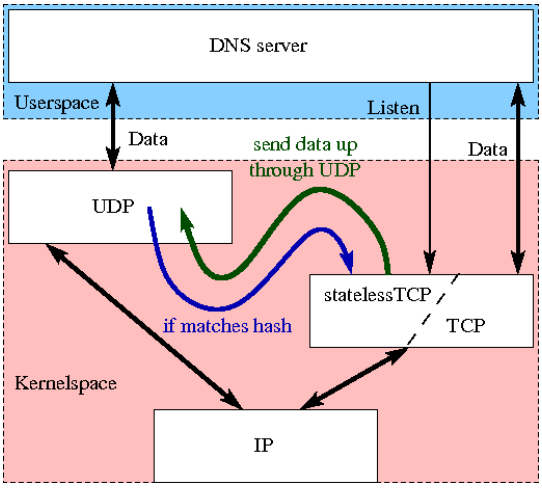

Fig. 1. TCP as UDP

The addresses and port numbers for the TCP segment can be extracted from the UDP packet. The send sequence number could be set to '1' for each packet, but the acknowledgement number will need to match the DNS query packet (See Algorithm 2, Item 2). StatelessTCP will need to store this from the incoming TCP packet in an identifiable way to be able to generate the correct TCP headers for the response. The information necessary to match the response UDP packet with the query TCP packet are:

- source and destination addresses
- source and destination ports

In our design the UDP stack has to check outgoing packets with a hash kept of statelessTCP connections. If there is a match, the packet is rerouted via statelessTCP with the necessary acknowledgement number. We use the existing SYN cache for this purpose instead of creating a new hash table. This stores more than the minimum required information, but allows for the use of random initial sequence numbers, and more "realistic" TCP packets.

The resource usage is still low, as statelessTCP connections do not require a per connection socket nor the in_pcb and tcb data structures.

B. Can zone transfers be catered for?

Many DNS servers run additional servers (slaves) for backup, fallback and load distribution purposes, which simply mirror the contents of the primary (master) DNS server. For these purposes it is necessary to transfer whole zones from one server to another. The amount of data transferred usually exceeds the maximum UDP

packet size and needs to be transferred reliably, so TCP is used as transport for the purpose[1].

StatelessTCP uses the same protocol number and listens on the same port as TCP, and will be presented with zone transfers queries. We solve the problem by "peeking" into the DNS query to determine whether it is a zone transfer or not, and revert to "stateful" TCP if it is.

IV. Alternative designs

This section outlines some of the alternatives we have looked at.

A. Pure Stateless

This idea aims to keep no state at all. The idea relies on the ability to correctly construct a TCP response packet based on the TCP packet just been received from the client as follows:

- Source address = received destination address
- Destination address = received source address
- Source port = received destination port
- Destination port = received source port
- Sequence number = 1 for SynAck and 2 for data response
- Acknowledgement number = received sequence number + data length (or + 1 if it is a Syn or Fin)
- Use no TCP options

The above idea works well for responding to the Syn packet, Fin packet, and acknowledging the data packet. When the response comes back from the application (via UDP or a special stateless socket[2]), all elements of the TCP header can be filled in, except the acknowledgement number.

For the TCP stack to store and recover the acknowledgement number, it will need to be able to link the outgoing DNS response with the correct acknowledgement number for the DNS query packet. This problem is solved in our design using the SYN cache (see Section III).

1) Aside – sending an incorrect acknowledgement number: An alternative to finding the correct acknowledgement number is to use an incorrect acknowledgement number. In FreeBSD, if the TCP stack receives a packet that acknowledges data that has not been sent, in the established and closing states, it will drop the packet and send an acknowledgement packet indicating the data it is expecting.

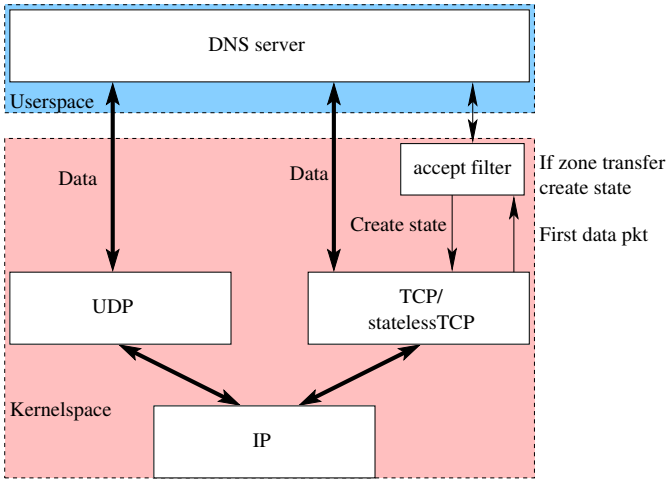

Fig. 2. Idea 2: No TCB unless a zone transfer

B. TCP socket connections without TCP state

This idea attempts to provide a version of TCP (stateless TCP) which has no TCP control block (tcb). In addition, accept filters[3] could be used to distinguish between requests requiring full TCP, and those that can operate with stateless TCP.

The main problem with this idea is that even though the stateless TCP stack does not keep TCP state in stateless mode, there is still quite a lot of state being kept. A new socket will be needed for each connection, thus having at least the overheads of a new socket with the IP protocol control block (in_pcb).

V. FreeBSD 9 Implementation of StatelessTCP

Our implementation of the design in section III uses the SYN cache for connection look-ups.

A. Basic Algorithm

Figure 3 shows a message sequence chart of the typical communication between the client and the server (over TCP), and the inter-working within the server's kernel protocol stack between TCP↔statelessTCP↔UDP. Stepping through the sequence from top to bottom:

[1] Although the DNS protocol can be used to do this, other protocols can also be used to achieve this outcome

[2] A special type of socket would require changes to the socket layer and the DNS server application

[3] Accept filters can sit between the TCP stack and the DNS application. When a TCP connection is received, the accept filter usually delays passing the information on to the listening application until after the first data packet has been received.

1) BIND 9/named[4], or another DNS server application, is listening on both, the TCP and UDP ports of the same port number.
2) The initial SYN – SYN/ACK exchange is handled by the TCP stack. A syncache entry is created, but no connection and state are established in the server.
3) The ACK from the client is parsed by statelessTCP and dropped.
4) The query from the client is redirected to statelessTCP, and the appropriate ACK is sent back (this stops the client resending the same query if the response is delayed).
5) If zone transfer detection is enabled (axfr_detect), the packet is inspected and if it's an axfr query, the packet is processed via normal TCP.
6) Otherwise, the query is repackaged as a UDP packet and passed to the UDP stack, which forwards it to the application, in this case named.
7) The answer from named is redirected from the UDP stack to statelessTCP.
8) The answer is segmented, if necessary, and sent as TCP packets using the information that has been stored in the syncache entry.
9) The last packet in the answer has the FIN flag set.
10) The client responds to the FIN closing its connection with a FIN/ACK – ACK exchange.
11) StatelessTCP removes the syncache entry

Placing the FIN on the last packet in the answer [see item 9)] saves one TCP packet in the exchange. Alternatively, we could wait and respond with a FIN to the Client's FIN there will be one additional ACK returning from the Client to ACK our FIN. The advantage of doing that is that StatelessTCP could redirect multiple UDP answer packets, however, named does not operate in this way.

B. Configuration parameters

StatelessTCP defines the following system control (`sysctl`) parameters shown with their defaults:

- net.inet.tcp.stateless.timeout: 3000
  - statelessTCP lookup table timeout (ticks).
- net.inet.tcp.stateless.loglevel: 1
  - statelessTCP logging level: 0 – no logging, 1 – log only errors, 2 – detailed log, 3 – very detailed logging for debugging.
- net.inet.tcp.stateless.port: 53
  - Port monitored for statelessTCP.
- net.inet.tcp.stateless.active: 0
  - When this is not 0 statelessTCP is activated on the port defined in net.inet.tcp.stateless.port.
- net.inet.tcp.stateless.axfr_detect: 0
  - When this is not 0 statelessTCP checks queries to see if they are zone transfers. If they are it allows them to operate over regular TCP.

In addition the following `sysctl` parameters may need to be adjusted for the expected traffic load:

- net.inet.tcp.syncache.hashsize
- net.inet.tcp.syncache.cachelimit
- net.inet.tcp.syncache.bucketlimit

C. Changes to FreeBSD 9

The following is a summary of changes to files in the FreeBSD 9 source tree:

- `sys/conf/files`
  - `netinet/statelesstcp.c optional inet`
- `sys/netinet/tcp_input.c`
  - addition of calls to (statlesstcp_input) conditional on V_tcp_stateless and V_tcp_stateless_port
- `sys/netinet/tcp_syncache.c`
  - changed declaration of `syncache_drop` from a static so that statelesstcp can use it.
- `sys/netinet/tcp_syncache.h`
  - added the flag definition `SCF_STATELESS_SYNINC 0x8000`
- `sys/netinet/udp_usrreq.c`
  - added call to `statelesstcp_output` in `udp_output` conditional on V_tcp_stateless and V_tcp_stateless_port.
- `sys/netinet/udp6_usrreq.c`
  - added call to `statelesstcp_output` in `udp_output` conditional on V_tcp_stateless and V_tcp_stateless_port.
  - added conditional skipping of the UDP checksum for redirections from statlesstcp.
- addition of `netinet/statelesstcp.c`
- addition of `netinet/statelesstcp.h`

VI. Resource consumption

We compare the performance of our statelessTCP implementation with the performance for normal stateful TCP and UDP. Using *dig*, queries are generated

[4]Although we use BIND 9/named as an example, statelessTCP changes the protocol stack not the DNS server application so any DNS server application should work. Similarly for the client dig.

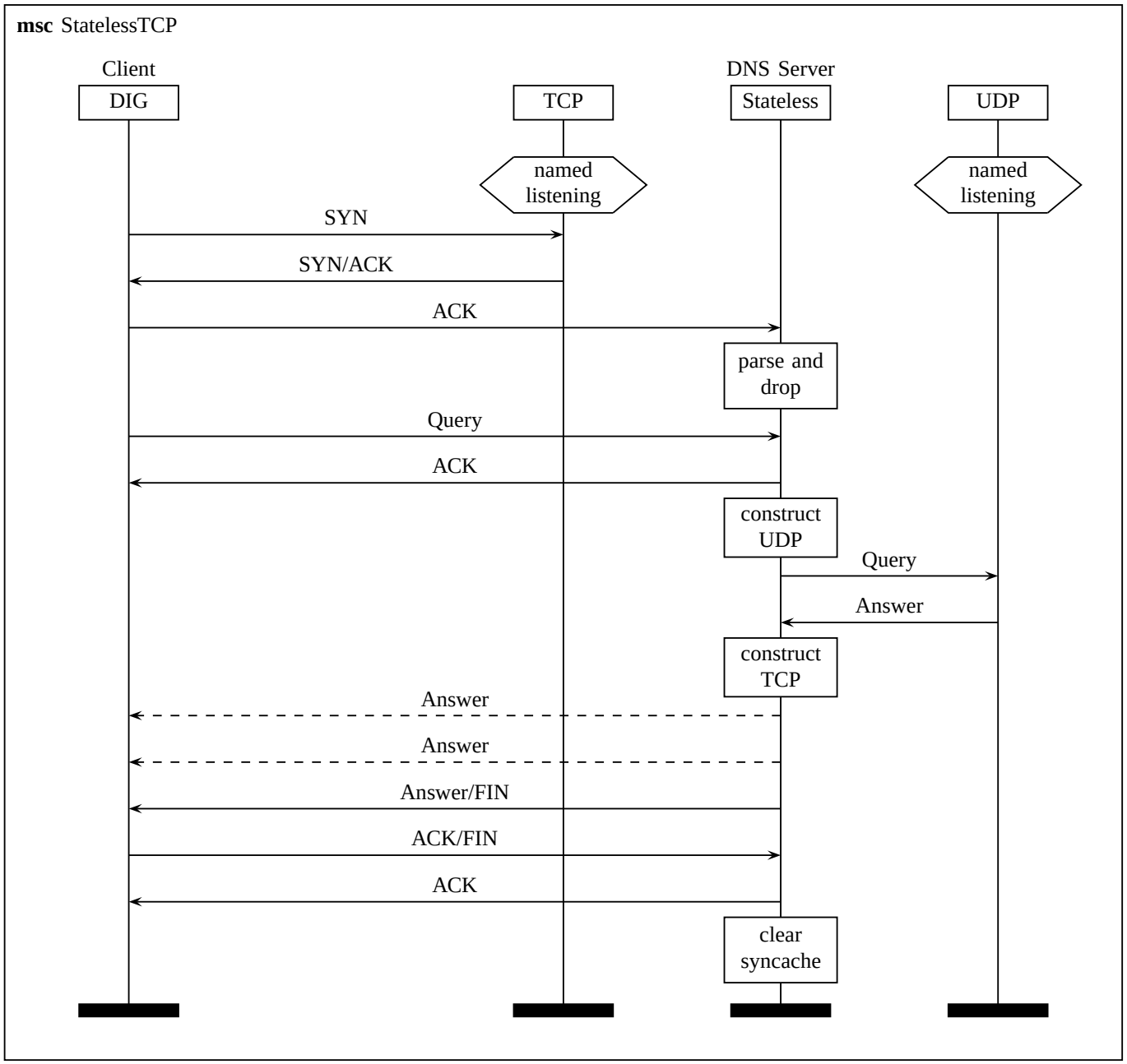

Fig. 3. Typical Client ↔ Server interaction with StatelessTCP

randomly from 5 hosts at total average rates of 50 queries/second up to 500 queries/second in steps of 50 queries/second. The CPU time consumed by each of the kernel and named processes is measured over 20 ten-second intervals. The additional memory used by TCP and statelessTCP relative to UDP is then estimated based on the stack control block. These tests use IPv4.

For performance tests we have adopted the following configuration parameters:

- net.inet.tcp.stateless.timeout = 3000 (default)
- net.inet.tcp.stateless.loglevel = 0 (turn off logging)
- net.inet.tcp.stateless.port = 53 (default)
- net.inet.tcp.stateless.active: set to 1 for the stateless tests, and 0 otherwise.
- net.inet.tcp.stateless.axfr_detect: set to 0 for the stateless tests, and 1 for the axfr detection test.
- net.inet.tcp.syncache.hashsize = 512 (default, though a prime number is best)
- net.inet.tcp.syncache.bucketlimit = 30 (default)

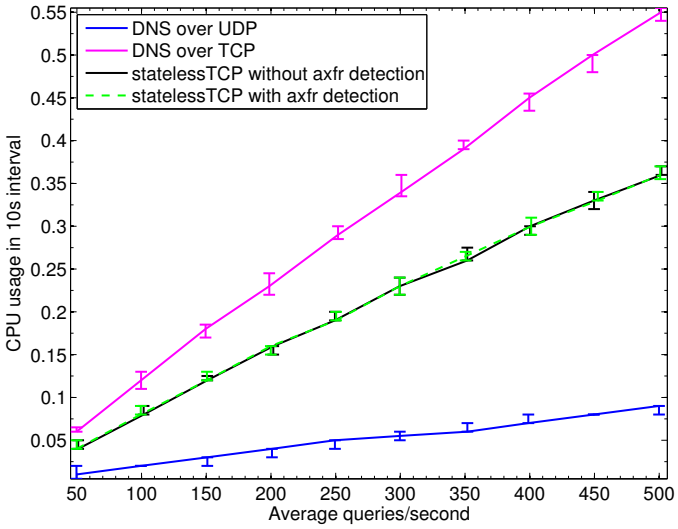

Fig. 4. Graph of the kernel CPU time for UDP, TCP, statelessTCP, and statelessTCP with afxr detection tests against DNS query arrival rates

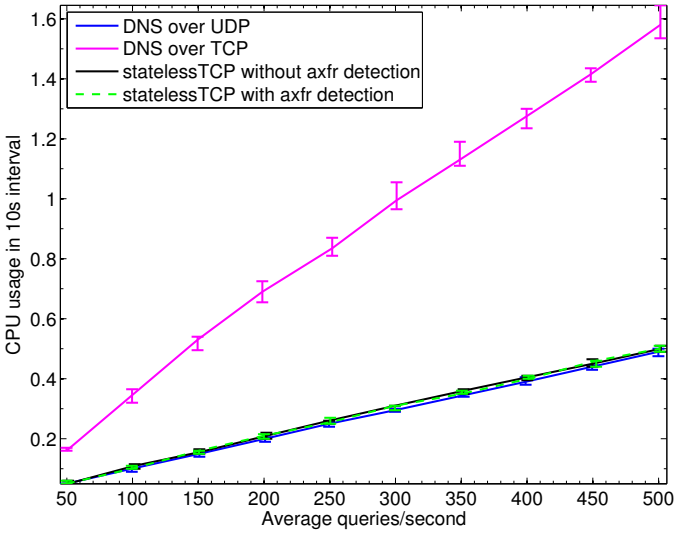

Fig. 5. Graph of the named CPU time for UDP, TCP, statelessTCP, and statelessTCP with afxr detection tests against DNS query arrival rates

A. *CPU load*

We measure CPU load in terms of the amount of CPU time consumed over a ten second interval. Figure 4 shows the relative performance for the kernel process. UDP has the least kernel load. Much of this is due to that fact that UDP only processes two packets (one in and one out), while TCP and statelessTCP usually process about eight packets. At 500 queries/second TCP takes 0.55 seconds of CPU time, statelessTCP 0.36, and UDP 0.09 seconds. StatelessTCP consumes about four times the kernel processing of UDP (roughly equivalent to the increase in packets), while TCP consumes more than six times the CPU time of UDP due to its increased overheads. StatelessTCP with afxr detection only slightly increases the kernel load. Detecting an afxr query type requires a search to the end of the domain name in the packet. The search is limited to 64 bytes, with these tests having a 20 byte name.

Figure 5 shows the relative performance plot for BIND 9's 'named' DNS server. Firstly, note that UDP and statelessTCP (with and without axfr detection) consume the same named resources. Since statelessTCP redirects packets via UDP, this is to be expected. TCP consumes much more CPU resources than UDP and statelessTCP. When named starts it creates a TCP listening socket. Since TCP is connection oriented, when it accepts a TCP connection, an new socket is created for the new connection. Comparing the CPU time used by the kernel and that of named, it is evident that named is where most of the CPU resources are consumed.

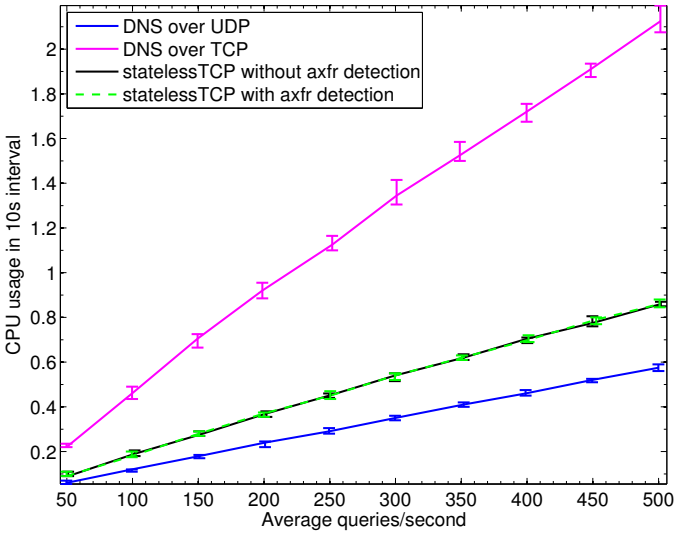

Fig. 6. Graph of the total CPU time for UDP, TCP, statelessTCP, and statelessTCP with afxr detection tests against DNS query arrival rates

Of key interest is the total CPU resources consumed. Figure 6 shows the combined CPU time for the kernel and named processes. At the 500 queries/second point UDP consumes 0.58 seconds of CPU time every ten seconds, statelessTCP consumes 0.86 seconds (about 48 % more than UDP), and TCP consumes 2.13 seconds (about 367 % more than UDP and 248 % more than statelessTCP). Further the relationship between CPU usage and queries/second is linear over the range tested. This indicates that a DNS server processing TCP queries with statelessTCP (with or without axfr detection) will

need to be about 50 % more powerful than its equivalent processing UDP queries. A DNS server using regular TCP to process DNS-over-TCP queries will need to be up to four times more powerful than its equivalent processing UDP queries.

B. Kernel memory use

Both TCP and statelessTCP use more memory resources than UDP. For TCP the extra memory can be estimated by the average number of concurrent TCP[5] sessions, and the memory resources[6] each session requires. A similar estimate can be made for statelessTCP:

$$\begin{aligned} M &\approx \big(\text{sizeof(in\_pcb)} + \text{sizeof(tcb)} \\ &\qquad + \text{sizeof(reassembly queue)}\big) \\ &\approx \big(464 + 184 + \text{sizeof(reassembly queue)}\big) \end{aligned}$$

since the reassembly queue will be very small, say

$$\approx 648\,\text{Bytes}$$

The additional memory for statelessTCP[7]:

$$\begin{aligned} M &\approx \text{sizeof(syncache)} \\ &\approx 88\,\text{Bytes} \end{aligned}$$

Therefore we estimate that TCP requires more than seven times the kernel memory of statelessTCP.

At 500 queries/second we measured a sampled maximum of 23 (average about 10) concurrent TCP sessions and a sampled maximum of 22 (average about 10) concurrent statelessTCP sessions. They should be about the same for the same arrival rate.

C. Performance conclusions

These tests indicate that CPU usage is likely to be the dominant issue for DNS servers in the move from UDP queries to TCP queries. In our tests the user-space DNS server (in our case, BIND 9's named) consumes the majority of the CPU resources, particularly when TCP is the underlying transport. Table I summarises the performance differences between UDP, statelessTCP and regular TCP.

Most of statelessTCP's gains over full TCP seem to be due to its not requiring a new socket for every new TCP connection. If sockets can be created and closed in a more efficient manner, full TCP may see significant performance gains. This is a potential area for further study.

[5]This does not include local function variables, but stored state

[6]Assuming 32 bit pointers, 32 bit integers, and 32 bit longs

[7]Same assumptions as for TCP

Protocol	increase of CPU load	Bytes of extra memory
UDP	0 %	0
statelessTCP	48 %	88
TCP	367 %	648

TABLE I
SUMMARY OF RESOURCE USAGE OF STATELESSTCP AND TCP COMPARED TO UDP

VII. EMPLOYING "STATELESS" TCP IN THE REAL WORLD

After having shown the performance boost obtained using statelessTCP instead of regular TCP, we look at issues concerning its potential deployment on DNS servers in the Internet.

The rationale for StatelessTCP is that circumstances arise where it might be impossible for UDP based DNS responses to reach the client. However, it is currently not common for DNS clients to fall back to TCP if their UDP query fails. The DNS clients we tested fall back to TCP only when they receive a DNS response over UDP with the "truncated response" flag set[8].

Because of this behaviour, if operators of DNS servers would like to force clients to fall back to TCP, an additional option needs to be added to statelessTCP. Currently statelessTCP checks all UDP packets sent from the DNS server, and redirects for statelessTCP operation only those packets whose connection has been identified in the TCP SYN cache. To force clients to fall back to TCP for when responses are larger than 512 bytes, statelessTCP could check the size of UDP based responses and rewrite them as "truncated response" packets when they are larger then 512 bytes.

It is not clear how clients will behave in future. It might be that, before IPv6 and DNSSEC will be widely used, all firewalls will be updated and let large UDP packets pass. This would make statelessTCP unnecessary in the long term, though potentially useful during the transition. However, if market pressures instead caused clients to switch to DNS-over-TCP anyway[9], statelessTCP could become an important tool in handling the dramatic increase in server load this would produce.

[8]Observed by performing DNS lookups on MacOS X 10.6, Windows XP SP3, Windows 7, FreeBSD 8.1 and Linux running a glibc2 version 2.12.1. The client applications were Firefox 3.6 on all platforms, and Safari and Internet Explorer 8 where available

[9]Perhaps as part of a regular patch or upgrade cycle

VIII. Conclusions

Driven by problems that exist in the current DNS system for large DNS responses over UDP (as with IPv6 and DNSSEC), Huston proposed "stateless" TCP. We implement our own version of statelessTCP in the FreeBSD 9 kernel.

Using BIND 9's named under FreeBSD 9, we show that TCP-based DNS queries required 365 % more CPU than UDP-based queries. In contrast, our statelessTCP implementation handles TCP-based DNS queries with only 44 % more CPU than required to handle UDP-based queries.

Further investigation is required to better understand issues regarding when and how clients might fallback to using TCP, and how statelessTCP may be augmented to assist this transition.

Acknowledgements

This work has been made possible in part by grants from APNIC Pty Ltd and Nominet UK, and collaboration with Geoff Huston and Roy Arends.

References

[1] G. Huston, "Stateless and dnsperate!" The ISP Column, Nov. 2009. [Online]. Available: http://www.potaroo.net/ispcol/2009-11/stateless.pdf

[2] G. Huston, "Stateless and dnsperate!" Jan. 2010. [Online]. Available: http://www.potaroo.net/presentations/2010-01-29-stateless-short.pdf

[3] G. Huston, "Dns proxy." [Online]. Available: http://www.potaroo.net/tools/useless/

[4] D. Karrenberg, "Measuring dns transfer sizes - first results," Feb. 2010. [Online]. Available: http://labs.ripe.net/Members/dfk/content-measuring-dns-transfer-sizes-first-results