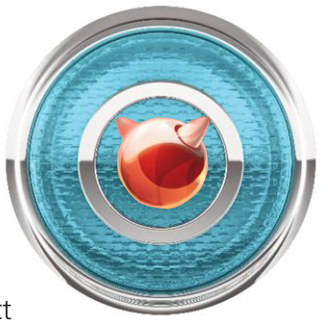
SEE TEXT ONLY

TCP IMPROVEMENTS in FreeBSD 11

BY HIREN PANCHASARA

One of the major obstacles preventing us from experimenting with TCP was fear of the unknown. Because TCP is complex and the code itself was non-modular in its past form, it was difficult to segregate a fix that wouldn't impact the rest of the code. However, it was relatively easy to introduce subtle regressions.



THE TCP ALTERNATIVE STACK FRAMEWORK attempts to fix this problem by modularizing the stack so that anyone can come up with a new TCP stack and have both (or more) stacks coexist on the same machine at the same time and serving different connections. The code is modularized so different TCP operations, like input or output processing, segment handling, timer processing, and socket options parsing, are subdivided into different function blocks and are made accessible with function pointers. Thus, a developer can design an alternate stack with a new way of handling input or output processing and still use the rest of the functionality from the default stack. One can select a stack per connection using socket options which provide the ability to do real A/B testing when experimenting with various TCP features.

The following sysctls can be used for configuration:

```
net.inet.tcp.functions_available - List all available stacks
net.inet.tcp.functions_default - Set/Get default stack
```

What follows is a short example of a server-side setting '`mytcp`' as a stack to use on the listen socket. Each new connection it accepts would get '`mytcp`' as its TCP stack.

```
socklen_t slen;
char *stack_name="mytcp";
struct tcp_function_set fsn;
int error;

memset(&fsn, 0, sizeof(fsn));
strcpy(fsn.function_set_name, stack_name);
slen = sizeof(fsn);
error = setsockopt(s, IPPROTO_TCP, TCP_FUNCTION_BLK, &fsn, slen);
if (error)
printf("Could not set TCP stack to %s, error:%d\n", func_blk, errno);
```

TCPPCAP—A Debugging Feature

At times, TCP problems surface in weird ways, and we want to have access to the last few packets or TCP state transitions so that we can figure out what went wrong. This feature provides that by saving the last configurable number of packets in the corresponding TCP control block. The number of packets can be specified on a global, i.e., per-system basis, or per-connection using a socket option.

A note of caution: Although this feature does a good job of keeping a cap on the maximum amount of memory that can be allocated to it, the number should be chosen carefully, as saving packets consumes memory.

This feature can be enabled with the TCPPCAP option in the kernel configuration. Even then, the feature is deactivated. When activated, the TCP control block has `t_inpkts`, which is the list of input packets saved, and `t_outpkts`, which is the list of output packets saved.

The following sysctl can be used for configuration:
`net.inet.tcp.tcp_pcap_packets` - Enable feature and set number of packets to capture per connection in each direction
`net.inet.tcp.tcp_pcap_clusters_referenced_max` - Maximum `mbuf` clusters allowed to be referenced

The following sysctls can be used to monitor memory usage by this feature:
`net.inet.tcp.tcp_pcap_clusters_referenced_cur` - Total `mbuf` clusters referenced
`net.inet.tcp.tcp_pcap_alloc_reuse_ext` - Total reused `mbufs` with external storage
`net.inet.tcp.tcp_pcap_alloc_reuse_mbuf` - Total reused `mbufs` with internal storage
`net.inet.tcp.tcp_pcap_alloc_new_mbuf` - Total new `mbufs` allocated

Server-side TCP Fast Open (TFO)

TCP performs a three-way handshake (`3-WHS: SYN, SYN-ACK, ACK`) between client and server before any actual data can be transferred on the connection. For short-lived connections, this can be a significant part of the total time spent on the transaction, i.e., latency experienced by the client.

TCP Fast Open (TFO), introduced by rfc 7413, addresses this problem by allowing `SYN` and `SYN-ACK` packets to carry data if the client has a valid security cookie with which the server can authenticate the client.

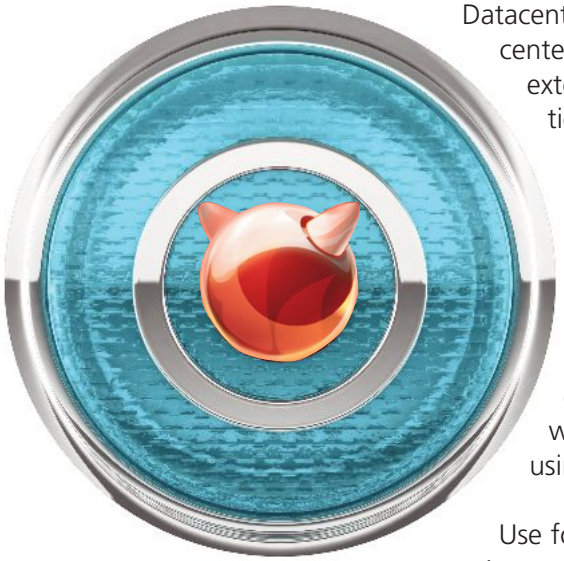
FreeBSD now has server-side support for this feature. The main part of TFO is the security cookie. The client first requests the cookie by sending `SYN` with a `FAST OPEN` option and an empty cookie field. The server generates a cookie and sends it back with the `FAST OPEN` option of a `SYN-ACK` packet. The client caches that cookie to use it in `FAST OPEN` connections to that same server. Once the client has the cookie, it sends `SYN` with the data and the cached cookie in the `FAST OPEN` option field. If the cookie is valid, the server acks both the cookie and the data and sends the data to the application. If the cookie is not valid, the server drops the data and only acks the `SYN`. If the server accepts the data provided in the `SYN`, it can send the response data before the `3-WHS` finishes. The client acks both the `SYN` and data if present. If not, it only acks the server's `SYN`. The rest of the connection proceeds as a regular TCP connection.

The client can make TFO connections until the cookie is valid and its validity is configurable on the server side. Thus, if a client is making a lot of short-lived connections to the same server, this feature can help a lot in reducing overall latency.

This feature can be enabled by adding '`options TCP_RFC7413`' to the kernel configuration. How many multiple concurrent keys to use can be specified with '`options TCP_RFC7413_MAX_KEYS=<num-keys>`' in the kernel configuration. Currently it defaults to `2` configuration sysctls:
`net.inet.tcp.fastopen.enabled` - Enable/Disable this feature
`net.inet.tcp.fastopen.autokey` - Automatic key renew timeout; defaults to 120secs

`net.inet.tcp.fastopen.acceptany` - Accept any key client supplied cookie; disabled by default
`net.inet.tcp.fastopen.keylen` - The key length in bytes
`net.inet.tcp.fastopen.maxkeys` - Max keys supported
`net.inet.tcp.fastopen.numkeys` - Total keys installed

Datacenter TCP



Datacenter TCP (DCTCP) is a congestion control mechanism to be used in datacenters. It uses explicit congestion notification (ECN) to estimate the extent/amount of congestion instead of just detecting that some congestion has occurred.

Traditional congestion control mechanisms underperform inside datacenters because the traffic is high throughput, low latency, and bursty in nature. Because ECN can estimate the severity of congestion, DCTCP uses that to reduce the congestion window accordingly, which helps in using available bandwidth efficiently.

DCTCP is now included as a congestion control module which can be the default congestion control for all connections or can be set per connection using socket options. The current implementation can also work in a one-sided fashion where only one side (sender or receiver) is using DCTCP.

Use following sysctls to configure this feature:
`net.inet.tcp.cc.dctcp.slowstart` - Determines whether to reduce the congestion window by half after the first slowstart.
`net.inet.tcp.cc.dctcp.shift_g` - Determines convergence time and bandwidth utilization. The IETF Draft suggests keeping it at '`4`' which we follow in the implementation.
`net.inet.tcp.cc.dctcp.alpha` - Determines how aggressive you want DCTCP to be at the cost of queueing delay at the beginning of the connection. The IETF Draft suggests being less aggressive with value '`1`,' but we decided to be more aggressive and kept the default at '`0`' in FreeBSD.

Packetization Layer Path MTU Discovery for Blackhole Detection

The maximum transmission unit (MTU) is the largest amount of data that can be sent in a frame. If a device is in the path with outgoing interface MTU less than the size of the frame, and if the '`don't fragment`' (DF) bit is set, the frame is discarded and an ICMP message '`Fragmentation needed and DF set`' is sent back to the sender with the outgoing interface MTU size in it. Upon receiving this message, the sender can adjust its MTU and resend the frame.

Sometimes these ICMP messages are blocked because of firewalls or some other mysterious reasons, and packets are silently discarded without the server ever knowing about them. That is called a PMTU blackhole. In such cases, being able to detect a blackhole without needing ICMP is really useful and the Packetization Layer Path MTU Discovery (PLPMTUD) blackhole detection introduced by rfc 4821 does exactly that.

When sending a packet fails twice with retransmission timeout (RTO), it tries to detect whether the connection is experiencing a blackhole along the path. It does this by reducing maximum segment size (MSS) to a preconfigured value. And we continue reducing the MSS in the case of more successive failed sending attempts until we reach the minimum configured MSS value. If the packet can be sent using a lower MSS, that means we, indeed, were experiencing a blackhole along the path. But if we cannot send the packet, even with the smallest configured MSS value, it is not the blackhole, but real congestion that is causing packet drops, and in that case, we restore the MSS to its original value.

Another important part of this feature is the upward probing of the network with incrementally

higher values of MSS on successful transfers. Yet, that is not currently implemented in FreeBSD, and because of that, this feature is disabled by default.

In many congestion control mechanisms, congestion window growth depends on the size of MSS, and therefore a connection in a blackhole may experience slower growth of a congestion window and thus be slower in utilizing path capacity.

Use the following sysctls to configure this feature:

`net.inet.tcp.pmtud_blackhole_detection` - Enable/Disable
`net.inet.tcp.pmtud_blackhole_mss` - Lowered MSS for IPv4
`net.inet.tcp.v6pmtud_blackhole_mss` - Lowered MSS for IPv6

Use the following sysctls to monitor this feature:

`net.inet.tcp.pmtud_blackhole_activated` - How many times we detected a blackhole and activated MSS down-probing.
`net.inet.tcp.pmtud_blackhole_activated_min_mss` - How many times we detected a blackhole and went down to minimum MSS configured during MSS down-probing.
`net.inet.tcp.pmtud_blackhole_failed` - How many times we detected a blackhole incorrectly

Short-lived TCP Connections Scalability Effort

The layered design of the networking stack and its implementation indicate complex locking and serialization challenges. Each layer has to talk to the neighboring layers to stay in sync to accept, serve, and tear down connections. Even within each layer, depending on its functionality, there has to be proper synchronization to maintain consistency when many, many connections are sharing the same data structures and memory resources.

Struct inpcb is the IP layer protocol control block structure which captures the network layer states like host addresses and routing information for TCP, UDP, and the like.

Before this effort/change for short-lived connections, close to 80% of received packets were processed holding exclusive write lock. This means that only one cpu core can work on receiving packets and the rest of the cores have to wait for the lock to be released. Now, a new `INP_LIST` lock has been added that protects modifications to the global inpcb list. This allows `INP_INFO_RLOCK` (a read lock) in critical paths like input processing, and only occasionally needs `INP_INFO_WLOCK` (a write lock) when it is really needed, i.e., walking global inpcb list while being stable.

The maximum number of TCP connections (setup and teardown) per connection is increased from 60k/sec to 150k/sec.

Improved Protection Against Blind In-window SYN/RST Spoofed Attack

If an attacker could somehow guess the valid window of sequence numbers and forge a `SYN` or a `RST` packet, FreeBSD used to drop the connection without looking at whether it's the exact packet it expects at that point in time or not.

Now with the enhancements proposed by rfc 5961, if the incoming `RST` is the exact sequence number we are expecting, drop the connections, and if it's not that, but within the window, send a challenge `ACK`. For incoming `SYN`, if the connection is in a synchronized state, send a challenge `ACK`.

Sending a challenge `ACK` makes a genuine sender generate a `RST` that matches the exact sequence number that we expect. Basically, with this mechanism in place, it's very hard for the attacker to spoof a valid `SYN/RST` that can cause us to drop the connection. Some middleboxes or broken TCP stacks may still respond to the challenge `ACK` with a `RST` packet that has a SEQ number that we do not expect. In that case, because of this feature, we may see a lot of spurious challenge `ACK` and `RST`s going back and forth. Challenge `ACK`s should be throttled to alleviate this problem so that we only send a limited amount of challenge `ACK`s per time period. For example, send only 10 challenge `ACK`s in a 5-second period. The time and amount of acks allowed should both be tunables. This is not yet implemented in FreeBSD.

This feature is enabled by default on FreeBSD. It can be disabled by setting `net.inet.tcp.insecure_rst` and `net.inet.tcp.insecure_syn` to '1'.

TCPDEBUG Like Information with Dtrace Trace Points

TCP has a debugging functionality that can print traces of connection state changes and other useful events like sending and receiving packets. But to enable this feature, the kernel needs to be built with the TCPDEBUG option in the kernel configuration file and the associated socket needs to be marked with the `SO_DEBUG` option—which means recompiling the application.

Now, with Dtrace (a dynamic tracing framework) probe points enabling the same functionality, it can be used without needing to recompile the kernel or userland application. `$src/share/dtrace/tcpdebug` is a sample script and a nice example of how to use this feature.

HIREN PANCHASARA has been struggling his way through kernel code for a few years now, and lately more so in networking and TCP. He runs FreeBSD on all things possible and wishes to bring more and more newbies to FreeBSD by mentoring and other means.
