connect() - why you so slow?
Frederick Lawler
Systems Engineer @ Cloudflare

- security_create_user_ns()
- CVE-2022-47929: traffic control noqueue no problem?
- pci_(alert|crit|dbg|emerg|err|info|notice|warn) printk macros
50k egress unicast connections to a single destination... Who does that?
Who does that?

CDN request flow for uncached assets
$ sysctl net.ipv4.ip_local_port_range
net.ipv4.ip_local_port_range = 9024   65535
bind() before connect()

```python
sk = socket(AF_INET, SOCK_STREAM)
sk.setsockopt(IPPROTO_IP, IP_BIND_ADDRESS_NO_PORT, 1)
sk.bind((src_ip, 0))
sk.connect((dest_ip, dest_port))
```

How to stop running out of ephemeral ports and start to love long-lived connections
Who does that?
2 IPv4 addresses for this service
Who does that?

tcp_v4_connect() func latency 2 IPv4 address

count vs. ns

- $2^{12} \text{ to } 2^{13}$
- $2^{13} \text{ to } 2^{14}$
- $2^{14} \text{ to } 2^{15}$
- $2^{15} \text{ to } 2^{16}$
- $2^{16} \text{ to } 2^{17}$
- $2^{17} \text{ to } 2^{18}$
- $2^{18} \text{ to } 2^{19}$
- $2^{19} \text{ to } 2^{20}$
- $2^{20} \text{ to } 2^{21}$
- $2^{21} \text{ to } 2^{22}$
- $2^{22} \text{ to } 2^{23}$
- $2^{23} \text{ to } 2^{24}$

Count vs. ns
Who does that?

1 IPv4 addresses for this service
Who does that?

IPv4 sales data. Source: [Hilco Streambank](https://hilco-streambank.com).
Who does that?
tcp_v4_connect() func latency 1 IPv4 address
Who does that?
tcp_v4_connect() func latency 3 IPv4 address (for fun)
This is fine for most workloads, but for Cloudflare...

- Customers largely still leverage IPv4
- Similar performance with 1 IPv4’s as we’d see with 3
- Leverage our infrastructure to lazily hand off excess connections ie. fail fast
Time to investigate: TCP connect() why you so slow?
Time to investigate: TCP connect() why you so slow?
inet_hash_connect() overview

- Called in both TCP IPv4 & IPv6 contexts; but we’ll be focusing on IPv4
- We assume the kernel has to pick a port

Time to investigate: TCP connect() why you so slow?
offset &= ~1U;

other_parity_scan:
    port = low + offset;
    for (i = 0; i < remaining; i += 2, port += 2) {
        if (unlikely(port >= high))
            port -= remaining;

        inet_bind_bucket_for_each(tb, &head->chain) {
            if (inet_bind_bucket_match(tb, net, port, l3mdev)) {
                if (!check_established(death_row, sk, port, &tw))
                    goto ok;
            }
            goto next_port;
        }
    }

    offset++;
    if ((offset & 1) && remaining > 1)
        goto other_parity_scan;
offset &= ~1U;

other_parity_scan:
    port = low + offset;
    for (i = 0; i < remaining; i += 2, port += 2) {
        if (unlikely(port >= high))
            port -= remaining;

        inet_bind_bucket_for_each(tb, &head->chain) {
            if (inet_bind_bucket_match(tb, net, port, l3mdev)) {
                if (!check_established(death_row, sk, port, &tw))
                    goto ok;
            }
            goto next_port;
        }
    }

offset++;
if ((offset & 1) && remaining > 1)
    goto other_parity_scan;

Time to investigate: TCP connect() why you so slow?

- Offset is randomly generated
- Offset is set to an even number
- Picked port is either “even” or “odd” based on net.ipv4.ip_local_port_range’s low port eg. 9024
offset &= ~1U;

other_parity_scan:
  port = low + offset;
  for (i = 0; i < remaining; i += 2, port += 2) {
    if (unlikely(port >= high))
      port -= remaining;
  }
  inet_bind_bucket_for_each(tb, &head->chain) {
    if (inet_bind_bucket_match(tb, net, port, 13mdev)) {
      if (!check_established(death_row, sk, port, &tw))
        goto ok;
      goto next_port;
    }
  }
  offset++;
  if ((offset & 1) && remaining > 1)
    goto other_parity_scan;

- Check if the socket is unique
- `check_established()` == `__inet_check_established()`
Is `__inet_check_established()` a problem?

- Tested benchmarks on a quiet virtual machine
- No other connections were established for the same src/dest ip + dest port
- Therefore, negligible impact
- Bind buckets will fill up eventually!

The quantum state of a TCP port
offset &= ~1U;

other_parity_scan:
    port = low + offset;
    for (i = 0; i < remaining; i += 2, port += 2) {
        if (unlikely(port >= high))
            port -= remaining;

        inet_bind_bucket_for_each(tb, &head->chain) {
            if (inet_bind_bucket_match(tb, net, port, l3mdev)) {
                if (!check_established(death_row, sk, port, &tw))
                    goto ok;
            }
        }
    }  
    goto next_port;

offset++;
if ((offset & 1) && remaining > 1)
    goto other_parity_scan;

● Loop through first half of the ephemeral range then second

● Every other port is tested in sequence
Time to investigate: TCP connect() why you so slow?

__inet_hash_connect() overview: the loop

#sk ≤ N/2

∅

0 1 2 3 4 5 6 7

sk₀

0 1 2 3 4 5 6 7

sk₁

0 1 2 3 4 5 6 7

sk₂

0 1 2 3 4 5 6 7

sk₃

0 1 2 3 4 5 6 7

#sk > N/2

∀

0 1 2 3 4 5 6 7

sk₄

0 1 2 3 4 5 6 7

sk₅

0 1 2 3 4 5 6 7

sk₆

0 1 2 3 4 5 6 7

sk₇

0 1 2 3 4 5 6 7
Is the loop the problem?

- Via experimentation
- Counted the even ports green, odd ports red
- Our port range dictates we always loop through even ports first

Time to investigate: TCP connect() why you so slow?

3.3 min @ 56k connections

connection attempts: 56511 errors: 0 total connections: 56511 total time: 202084.456 avg time/connection: 3.576 connections/s: 279.641 even port(s): 28256 min: 0.093 max: 4.631 avg: 0.157 odd port(s): 28255 min: 4.588 max: 18.695 avg: 6.995
Our conclusion

- Exhausting half the `net.ipv4.ip_local_port_range` is fast
- The port looping appears to be our primary bottleneck
- Evidenced by a previous attempt
  [PATCH] tcp: avoid unnecessary loop if even ports are used up and was not merged
Tracking port parity switches

#!/usr/bin/env bpftrace

kretfunc:vmlinux:inet_hash_connect /retval == 0/ {
    $port = args->sk->__sk_common.skc_num;
    @procs[comm,cgroup] += $port & 1;
}

rate(connect_port_parity_switches_total)[1m]

Prometheus exporter for eBPF metrics
What do?
Some feasible, but not viable solutions for our case

1. Split egress unicast connections over 2..N IP addresses
2. Introduce a sysctl to manipulate connect
3. Pick a random port in userspace, and bind() with that
4. Leverage the new IP_LOCAL_PORT_RANGE socket option (v6.3.y)*
Split egress unicast connections over 2..N IP addresses

- Leaks networking configuration to user space
- No ability to tell the interface to balance between assigned IP’s or IP blocks
- Requires IP_BIND_ADDRESS_NO_PORT socket option + bind() before connect() pattern
- We do this strategy now, but want to reduce to 1 IP
What do?

Introduce a sysctl to manipulate connect

- Kernel modification
- [PATCH] tcp: avoid unnecessary loop if even ports are used up
Pick a random port in userspace, and bind() before connect()

- Requires bind() before connect()
- Syscall overhead and ~8-12 attempts per connect closer to exhaustion
- Good up to ~70-80% port range utilization

```python
sys = get_ip_local_port_range()
estab = 0
i = sys.hi
while i >= 0:
    if estab >= sys.hi:
        break
    random_port = random.randint(sys.lo, sys.hi)
    connection = attempt_connect(random_port)
    if connection is None:
        i += 1
        continue
    i -= 1
    estab += 1
```
Leverage the new IP_LOCAL_PORT_RANGE socket option (v6.3.y)

- Max # of connect() as range
- Pre-allocation of partitions to balance between
- Loop problem still persists

5k window @ 1.5 sec
Leverage the new IP_LOCAL_PORT_RANGE socket option (v6.3.y)

- Lower range works better
- Overlapping ranges is possible
- Overlap is determined by implementation

1k window @ 2.2 ms
Leverage the new IP_LOCAL_PORT_RANGE socket option (v6.3.y) + random offset

connection attempts: 56511 errors: 0 total connections: 56511
total time: 202084.456 avg time/connection: 3.576 connections/s: 279.641
even port(s): 28256 min: 0.093 max: 4.631 avg: 0.157
odd port(s): 28255 min: 4.588 max: 18.695 avg: 6.995

connection attempts: 56511 errors: 1054 total connections: 55457
total time: 9393.775 avg time/connection: 0.166 connections/s: 6015.792
even port(s): 27691 min: 0.091 max: 5.683 avg: 0.162
odd port(s): 27766 min: 0.089 max: 3.405 avg: 0.163

3.3 min → 9.3 sec!
Implementation details

sys.lo = 9024; sys.hi = 65535
window.lo = 0; window.hi = 1000
range = window.hi - window.lo
offset = random(sys.lo, sys.hi - range)
window.lo = offset; window.hi = offset + range
setsockopt(SOL_IP, IP_LOCAL_PORT_RANGE, window.lo | (window.hi << 16))
Implementation details

- Overlap is OK
- Reattempts may be necessary depending on use case
- Larger net.ipv4.ip_local_port_range is better with smaller selection window
In summary

- Leverages a random port offset + random low port in range to be even or odd
- Allows kernel to perform loop over a small + configurable local port range
- Overlaps windows on top of another

3.3 min → 9.3 sec @ 56k connections 1k window
Performance 56k unicast egress connections

3.3 min → 9.6 sec
500 window

- connection attempts: 56511 errors: 1024 total connections: 55487
- total time: 9685.281 avg time/connection: 0.171 connections/s: 5834.730
- even port(s): 27732 min: 0.093 max: 3.576 avg: 0.170
- odd port(s): 27755 min: 0.094 max: 3.542 avg: 0.170

3.3 min → 9.3 sec
1000 window

- connection attempts: 56511 errors: 1054 total connections: 55457
- total time: 9393.775 avg time/connection: 0.166 connections/s: 6015.792
- even port(s): 27691 min: 0.091 max: 5.683 avg: 0.162
- odd port(s): 27766 min: 0.089 max: 3.405 avg: 0.163
Performance 56k unicast egress connections

3.3 min → 13.9 sec
5k window

3.3 min → 25.8 sec
10k window
Takeaways

- Current implementation guarantees a port is selected
- Current implementation is not great at extreme egress workloads
- We can reduce the port-range to small-N loops per socket
- Random offset + 500-1k window coupled with kernel random port picking ensures we start looping at both odd and even ports with small-N
- Purely user space implementation
Discussions/Questions?

fred@cloudflare.com