TCP Tuning for the Web

Jason Cook - @macros - jason@fastly.com

Me

- Co-founder and Operations at Fastly
- Former Operations Engineer at Wikia
- Lots of Sysadmin and Linux consulting

Etsy AddThis github theguardian SHAZAM wikia

DISQUS Firebase swiftype voxer skimlinks imgIX WANELO Yammer

WISTIA thinglink. PicMonkey PubNub WANTFUL instructables Parse

The Goal

- Make the best use of our limited resources to deliver the best user experience

Focus

Linux

- I like it
- I use it
- It won't hurt my feelings if you don't
- Examples will be aimed primarily at linux

Small Requests

- Optimizing towards small objects like html and js/css

Just Enough TCP

- Not a deep dive into TCP

The accept() loop

Entry point from the kernel to your application

- Client sends SYN
- Kernel hands SYN to Server
- Server calls accept()
- Kernel sends SYN/ACK
- Client sends ACK

Backlog

- Number of connections allowed to be in a SYN state
- Kernel will drop new SYNs when this limit is hit
- Clients wait 3s before trying again, then 9s on second failure

Backlog Tuning (kernel side)

- net.ipv4.tcp_max_syn_backlog and net.core.somaxconn
- Default value of 1024 is for "systems with more than 128MB of memory"
- 64 bytes per entry
- Undocumented max of 65535

Backlog Tuning (app side)

- Set when you call listen()
- nginx, redis, apache default to 511
- mysql default of 50

DDoS Handling

The SYN Flood

- Resource exhaustion attack
- Cheaper for attacker than target
- Client sends SYN with bogus return address
- Until the ACK is completed the connection occupies a slot in the backlog queue

The SYN Cookie

- When enabled, kicks in when the backlog queue is full
- Sends a slightly more expensive but carefully crafted SYN/ACK then drops the SYN from the queue
- If the client responds to the SYN/ACK it can rebuild the original SYN and proceed
- Does disable large windows when active, but better than dropping entirely

Dealing with a SYN Flood

- Default to syncookies enabled and alert when triggered
- tcpdump/wireshark
  - Frequently attacks have a detectable signature such as all having the same initial window size
- iptables is very flexible for matching these signatures, but can be expensive
- If hardware filters are available, use iptables to identify and hardware to block

The Joys of Hardware

Queues

- Modern network hardware is multi-queue
- By default assigns a queue per cpu core
- Should get even balancing of incoming requests, irqbalance can mess that up
- Intel ships a script with their drivers to aid in static assignment to avoid irqbalance

Packet Filters

- Intel and others have packet filters in their nics
- Small, only 128 wide in the intel 82599
- Much more limited matchers than iptables
  - src,dst,type,port,vlan

Hardware Flow Director

- Same mechanism as filters, just includes a mapping destination
- Set affinity to put both queue and app on same core
- Good for things like SSH and BGPD
  - Maintain access in the face of an attack on other services

TCP Offload Engines

Full Offload

- Not so good on the public net
- Limited buffer resources on card
- Black box from a security perspective
- Can break features like filtering and QoS

Partial Offload

- Better, but with their own caveats

Large Receive Offload

- Collapses packets into a larger buffer before handing to OS
- Great for large volume ingress, which is not http
- Doesn't work with ipv6

Generic Receive Offload

- Similar to LRO, but more careful
- Will only merge “safe” packets into a buffer
- Will flush at timestamp tick
- Usually a win and you should test

TCP Segementation

- OS fills a 64KB buffer and produces a single header template
- NIC splits the buffer into segements and checksums before sending
- Can save lots of overhead, but not much of a win in small request/ response cycles

TX/RX Checksumming

- For small packets there is almost no win here

Bonding

- Linux bonding driver uses a single queue, large bottleneck for high packet rates
- teaming driver should be better, userspace tools only worked in modern fedora core so gave up
- Myricom hardware can do bonding natively

TCP Slow Start

Why Slow Start

- Early TCP implementations allowed the sender to immediately send as much data as the client window allowed

- In 1986 the internet suffered first congestion collapse
  - 1000x reduction in effective throughput
- 1988 Van Jacobsen proposes Slow Start

Slow Start

- Goal is to avoid putting more data in flight than there is bandwidth available
- Client sends a receive window size of how much data they can buffer
- Server sends an inital burst based on server inital congestion window
- Double the window size for each received ACK
- Increases until a packet drop or slow start threshold is reached

Tuning Slow Start

- Increase your congestion window
  - 2.6.39+ defaults to 10
  - 2.6.19+ can be set as a path attribute
    - `ip route change default via 172.16.0.1 dev eth0  proto static initcwnd 10`

Proportional Rate Reduction

- Added in linux 3.2
- Prior to PRR a loss would halve to congestion window potentially below the slow start threshold
- PRR paces retransmits to smooth out
- Makes disabling net.ipv4.tcp_no_metrics_save safer

TCP Buffering

Throughput = Buffer Size / Latency

Buffer Tuning

- 16MB buffer at 50ms RTT = 320MB/s max rate
- Set as 3 values; min, default, and max
- net.ipv4.tcp_rmem = 4096 65536 16777216
- net.ipv4.tcp_wmem = 4096 65536 16777216

TIME_WAIT

- State entered after a server has closed the connection
- Kept around in case of delayed duplicate ACKs to our FIN

Busy servers collect a lot

- Default timeout is 2 * FIN timeout, so120s in linux
- Worth dropping FIN timeout
  - net.ipv4.tcp_fin_timeout = 10

Tuning TIME_WAIT

- net.ipv4.tcp_tw_reuse=1
  - Reuse sockets in TIME_WAIT if safe to do so
- net.ipv4.tcp_max_tw_buckets
  - Default of 131072, way too small for most sites
  - Connections/s * timeout value

net.ipv4.tcp_tw_recyle

- EVIL!

net.ipv4.tcp_tw_reuse

- Allows the reuse of a TIME_WAIT socket if the client's timestamp increases

- Silently drops SYNs from the client if they don't, like can happen behind NAT

- Still useful for high churn servers when you can make assumptions of local network

SSL and Keepalives

- Enable them if at all possible
- The initial handshake is the most computationally intensive part
  - 1-10ms on modern hardware
- 6 * RTT before user gets data really sucks over long distances
  - SV to LON = 160ms = 1s before user starts getting content

SSL and Geo

- If you can move SSL termination closer to you users, do so
- Most CDNs offer this, even for non-cacheable things
- EC2 and Route53 is another option

In closing

- Upgrade your kernel
- Increase your initial congestion window
- Check your backlog and time wait limits
- Size your buffers to something reasonable
- Get closer to your users if you can

Thanks!

Jason Cook
@macros
jason@fastly.com