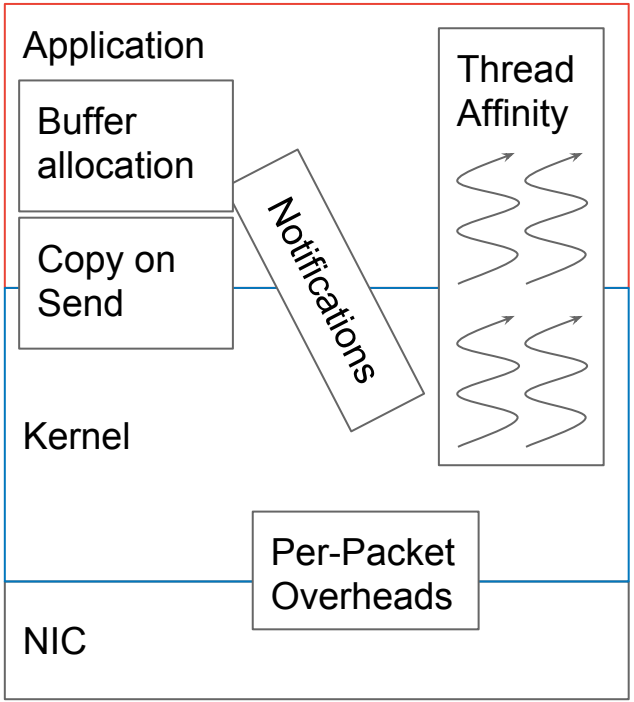
Making TCP faster and cheaper for applications

Soheil Hassas Yeganeh

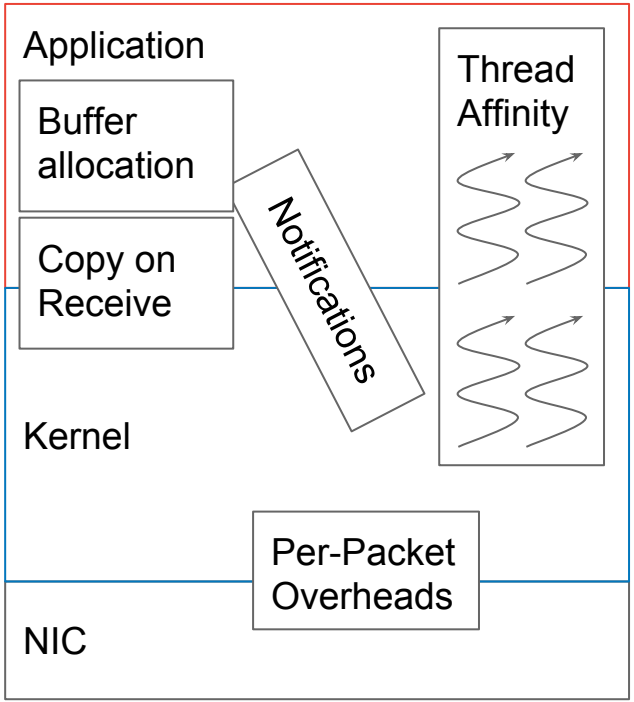
IETF 102 tcpm

Google

Sender



Receiver

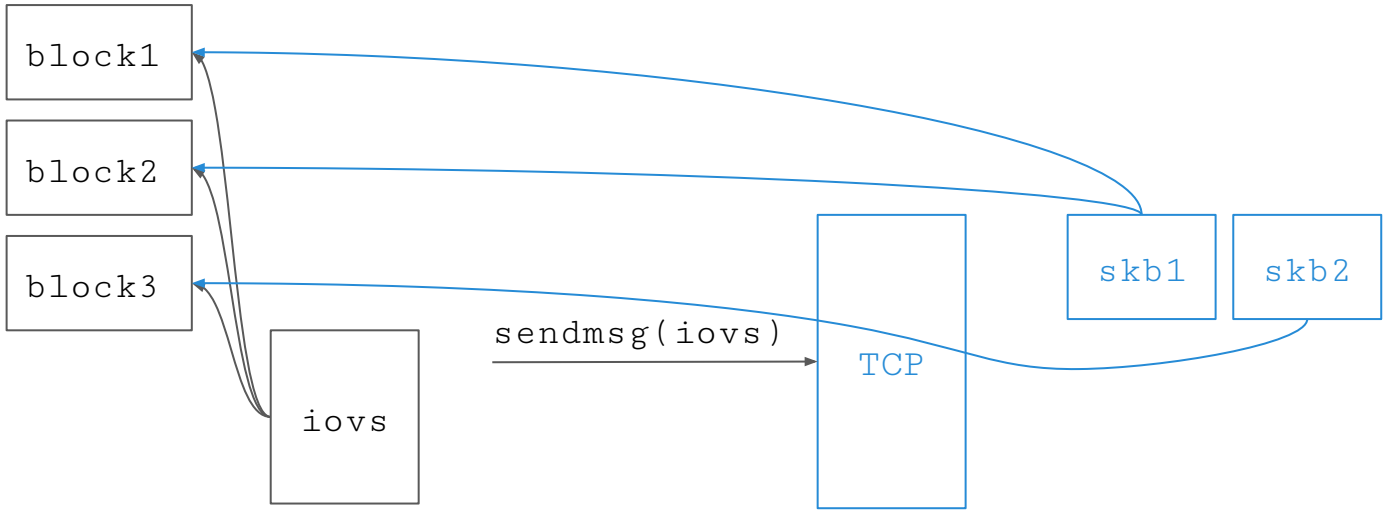


TCP bottlenecks in the wild, beyond congestion control

- TCP on emerging platforms should to be efficient, high-throughput and low-latency.
- Several bottlenecks beyond protocol and congestion control:
  - Copies upon send and receive are expensive.
  - Userspace and kernel processes handling the same socket can be scheduled on different CPUs.
  - Userspace threads can be woken up before they can actually do useful work.
  - System call overheads after recent security mitigations.

TCP Transmit ZeroCopy

- Implemented by Willem de Bruijn
- Enabled on a socket using setsockopt(SO_ZEROCOPY).
- Data sent via sendmsg(data, MSG_ZEROCOPY) system call.
- Packets (e.g., skbuffs in Linux) keep references to the data.
  - There are some exceptions. We will discuss this in caveats.



TCP Transmit ZeroCopy

- Return value of sendmsg(MSG_ZEROCOPY) is the identical to a normal send.
- A sequential ID assigned to each successful sendmsg(MSG_ZEROCOPY).
- When all of the SKBs for a sendmsg #ID are freed, userspace is notified (using the socket's error queue in Linux).

- Data must remain unmodified while an skb points to it, otherwise undefined behavior.

TCP Transmit ZeroCopy

- Saves a copy.
- Adds extra notification processing.

- Has more overheads for small sends, because copy 1B is much cheaper than processing a release notification.
- We observe 10%+ efficiency gains of large sends and 20%+ memory BW savings.

send:

```
iov[0].base = block1;
iov[0].iov_len = block1_size;
iov[1].base = block2
iov[1].iov_len = block2_size;
msghdr hdr;
hdr.msg_iov = iov;
hdr.msg_iovlen = 2;
...
do {
  r = sendmsg(fd, &msghdr,
              MSG_ZEROCOPY);
} while (r < 0 && errno == EINTR);
if (ret < 0) { // handle error
  return;
}
vector<...> blocks;
for (i = 0; i < 3 && r > 0; ++i) {
  blocks.push_back(iov[i]);
  if (r < iov[i].iov_len) break;
  r -= iov[i].iov_len;
}
zcopy_blocks[id++] = blocks;
```

release:

```
do {
  ret = recvmsg(fd, &msg,
                MSG_ERRQUEUE)
} while (ret < 0 && errno == EINTR);
if (ret < 0) { // handle error
    return;
}
for (cmsghdr cm in msg) {
  if (IsZeroCopy(cm)) {
    auto serr = (sock_extended_err*)
                        CMSG_DATA(cm);
    uint32 lo = serr->ee_info;
    uint32 hi = serr->ee_data;
     for (uint32 i=lo; i<=hi; ++i) {
       FreeBlock(zcopy_blocks[i])
     }
  }
}
```

this code is simplified and has to be synchronized for multi-threaded app.

TCP Transmit ZeroCopy

- We do not release packets on SAck due to reneging.
- We do not release packets on Ack, because a copy of a packet can sit in the end host:
  - A retransmitted copy maybe sitting in the Qdiscs.
- Data can be released ONLY WHEN the packet is not needed anywhere in the kernel.
- For the same reason, release signals can be out of order.
  - We have packets 1 .. N on TCP's transmit queue.
  - Packet #1 is spuriously transmitted, and sitting in the qdisc.
  - We receive ack for packet #N -> Release packets #2..#N.
  - Packet #1 is transmitted -> Release packet #1.

TCP INQ

- TCP knows exactly how much data is available to read at time T.
- But, applications do not have that information to allocate buffers optimally.
- Applications can use:
  - ioctl(SIOCINQ) aka FIONREAD.
- But, it's one extra syscall per read, and remember syscalls are now more expensive.
- Linux now communicates the remaining bytes available to read from TCP:

```
// read 1024 bytes, and kernel tells the app there is 64KiB more data to read.
recvmsg(fd, {..., len=1024, cmsg={TCP_INQ, 65536}}, 0) = 1024
```

- 3% to 5% more efficient for small and large RPCs.

Thread Wakeup

- TCP/Kernel notifies one or more pollers when there is an event on an FD the poller is subscribed to.
- By default, kernel sends a notification every time we add something to the receive queue of a socket.
- Applications, on the other hand, usually process data in frames:
  - Until a **P bytes** of payload is not read, no work can be done.
- 2 alternatives to lower wake ups:
  - Kernel parses frames and infers **P**. This wouldn't work for encrypted streams.
  - Userspace parses frame header and uses **P** as the low watermark. It's tricky to get right due to buffer and memory autotuning in TCP.
- Using the second approach we see significantly less wakeups for large frames.

Thread Affinity

- TCP tries to process packets on the same CPU where the user thread is expected to read/write the data.
  - See receive flow steering as an example.
- The heuristics are very simple:
  - Set the core ID of the socket when recvmsg and sendmsg are called.
  - TCP used to set the core ID on poll, which we removed.
- These heuristics won't work without orchestration:
  - Scheduler can move user-space threads around, the previous user space thread may get blocked, pinning doesn't work for all processes, ...
- Orchestrating userspace and kernel affinities we observe 10%+ gains in efficiency and latency:
  - Try to process TCP events on the same core both in userspace and the kernel.

Lessons Learnt

- Optimizing TCP's core is *necessary*, but it's *not sufficient* to provide the ultimate performance on emerging platforms.
- TCP's performance depends vastly on *how* TCP is used.
  - There is no way to guarantee affinity without user-space orchestration.
  - We cannot cook large TSOs if application doesn't provide enough backlog.
- Moving away from historical artifacts can provide large gains.
  - We can always revisit syscall copies, notification methods, ...
- Heuristics in TCP should be guided by how applications use TCP.
  - Setting RFS core on epoll would hurt performance, when we have a single poller.
- To guide optimizations, we should evangelize old and new TCP metrics (SNMPs, Tx and Rx timestamps, TCP chronos, ...):
  - nstat, ss, and kernel timestamps are your friends.
  - Linux perf tools and flame graphs are immensely helpful.
  - ftrace and ebpf are very useful for debugging thread affinity and latency.