STORAGE DEVELOPER CONFERENCE

SDC 22

Fremont, CA
September 12-15, 2022

BY Developers FOR Developers

A SNIA. Event

TCP Networking for Storage Software Developers

A survey of tricks to make your network stack fly

Ben Walker

Notices and Disclaimers

Outline

Level Setting

What is unique about “storage” network software

The Basics

Blocking vs. Non-blocking Operations

Grouping Connections

The Good Stuff

Better Connection Grouping

System Calls vs. Data Copies

Zero Copy Transmit

The Future

io_uring

What Makes Networking “Storage Networking”?

NVMe-oF and iSCSI Similarities

- Modeling fixed-size rings over a TCP stream
- Interleaving of small (< 100 bytes) commands and large (multiple of 4KiB) data on one connection
- Long-lived connections

The Basics

Berkeley Sockets

Every OS we’ll discuss today represents a connection as a socket

- Berkeley socket API is from 1983!
- There are OS-specific extensions and behavioral differences

Data is sent using send()

- The send() call copies data into an OS buffer and returns immediately. It does not wait for the data to actually arrive at the destination.
- If the OS buffer is full, send() can either block or perform a partial send, depending on the flags passed.

Data is received using recv()

- The recv() call attempts to receive up to the number of bytes requested
- Whether it waits for all of the data, waits for some amount of data, or never waits can be controlled with flags

A Simple Server

- Spawn 1 thread per socket
- Perform blocking send() and recv() operations on that socket
- The “Apache HTTPD model”
- Does not scale well
- Ok for simple clients

Processing Many Sockets From One Thread

Eliminate thread swapping overhead

- The “NGINX” model

To do this we’ll need:

- Some way to make our send() and recv() operations not block, but instead just tell us to try again later
- Some way to efficiently group together the connections so we don’t have to iterate the entire list repeatedly

Blocking vs. Non-blocking

Many ways to control behavior

The socket can be globally switched into non-blocking mode

- Linux/FreeBSD: fcntl to set O_NONBLOCK

The socket can be created in non-blocking mode

- Linux/FreeBSD: SOCK_NONBLOCK parameter to socket()

Individual recv() and send() calls have flags

- Linux/FreeBSD: MSG_WAITALL, MSG_DONTWAIT

A Simple Non-Blocking Server

Spawn 1 thread per core

On each thread, loop over sockets, performing non-blocking send() and recv() calls

No context switches! But now we’re going to get hammered by system call overhead.

Observe: Most sockets on each loop are idle

Grouping Connections

Operating on Sets of Connections

Berkeley sockets defines poll() to check or wait for one or more sockets in a set to become ready to read, ready to write, or to have an error.

Pass in an array of pollfd objects, which have the socket and a flag that's updated when poll() returns to indicate there was an event.

Loop over the pollfd array, find the ones flagged as ready, and process them

Poll returns if/when one of those sockets is ready to be processed.

Less system call overhead, but still iterating every socket in the set

Better Grouping

- epoll (Linux) and kqueue (FreeBSD)
  - The set of sockets is created in the kernel and persists.
  - The set of sockets can be modified at any time via system calls
  - When a network event happens, the kernel checks if the socket is in any groupings and notes that it is ready for processing. This is O(1).
  - When the user checks the grouping, it can quickly return the list of sockets that are ready without any iteration.

A Simple Non-Blocking Server, v2

Spawn 1 thread per core

On each thread, create an epoll/kqueue object.

Loop, checking the epoll/queue object on each iteration.

Much better! No full iterations over the set.

This is considered “state of the art” by most

Better Connection Grouping

We made it through the level-setting portion!

How Should Connections Be Assigned To Threads?

Simple: Round-robin as they arrive

- This is fine

Better: Distribute based on activity

- This is sometimes an improvement, but activity levels of sockets can quickly change so you end up rebalancing constantly

But…should we consider the hardware?

Packets Arrive on Hardware Rx Queues From the NIC

NICs have different ways to select which rx queue to put a packet into

- Round-robin per packet
- Round-robin based on the 4-tuple
- Consistent hashing based on the 4-tuple

Packet arrives on Core 0, recv called on Core 1, get locking/message passing!

- This is a major performance problem!
- Can we align our groupings of sockets to match the Rx queues the NIC will choose for those connections? YES

Some NICs Give Hints About The Rx Topology

Ask the NIC, via getsockopt

- SO_INCOMING_CPU reports the most recently associated CPU core for the Rx queue the last packet on this socket arrived on. If the NIC always routes packets for a connection to the same Rx queue, the RX queues can be deduced
- SO_INCOMING_NAPI_ID more directly reports a unique identifier per Rx queue.

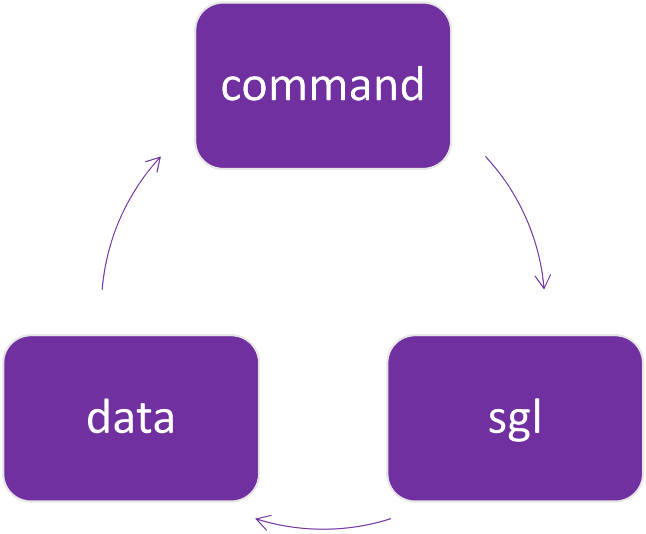
Tell the NIC, via setsockopt

- SO_MARK on some NICs allows an application to mark the desired groupings. All sockets with the same number will go to the same Rx queue.
- Must be done before connect(), so only useful on the initiator

System Calls vs. Data Copies

Parsing Storage Protocols

- recv() first command (64 bytes), look at what it says
  - If data, recv() data size into DMA buffer
  - If SGL, recv() sgl size
- Repeat



command

data

sgl

System Calls Are Expensive

Calling recv() for each segment of data will destroy performance

System calls have mostly “fixed” overhead

- Clearing registers and state

We can avoid making system calls by attempting to recv() larger chunks

- Grab multiple commands/data in one go, then parse out of our own buffer
- But if we find data, we now need to copy it out of our temporary buffer and into our DMA buffers

There’s some cut-over point where extra data copies are cheaper than system calls

- That cut-over point is about 8KiB

Zero Copy Transmit

TCP Segmentation Offload

- Send() copies the application data to the OS buffer, returns
- Kernel splits into MTU-sized packets, generates headers and checksums, and sends it on the wire
    - The NIC can hardware offload this “packetization” step using TSO. Hardware inserts packet headers and checksums.
    - Data must be held until TCP ack is received to support re-transmits

- Note: With TSO, the kernel just posts the OS buffer directly to the NIC!

Zero Copy Transmit

- If the OS could tell the app to hold onto the data buffer until it got the TCP ack, it could avoid the copy into the OS buffer.
  - Linux: Added MSG_ZERCOPY flag to send() and infrastructure to report when the transmission has really finished.
  - The OS must pin the data buffers to make them DMA safe. This has overhead
  - Zero copy is an improvement when sending at least 4KiB at a time – perfect for storage use cases!

Why No Zero Copy Recv?

- The OS drivers must keep data buffers posted to the NIC or the NIC will be forced to drop incoming packets.
- The packets are often small and scattered as they arrive
- The packets can arrive out of order
- The packets have protected headers that must be parsed and stripped before the application can see the stream

- So far, this has been a mostly intractable problem.
- Disclaimer: Linux has some limited support if you can control the MTU size on your network.

io_uring

Key Features

Asynchronous send and recv

Batching of system calls

Reduced overhead of fd operations (FIXED files)

More natural zero copy support

Pre-posting of recv buffers

System Call Batching

- Drop multiple descriptors into ring, then do one system call
- No need for epoll. Just post send() and recv() as needed.
- Pre-post buffers to pool to be used in recv().

Challenges

- Corner case behaviors are still maturing
  - When does MSG_WAITALL still result in a partial send()?
- Software needs to be adapted to asynchronous socket operations
  - This can be a big change
- The data copy between kernel and user is still as much of a problem as ever

Please take a moment to rate this session.

Your feedback is important to us.