The Caveats of TCP_NODELAY

Nov 23, 2015

One common mistake I see in socket programming is incorrect usage of `TCP_NODELAY` (the socket option that disables Nagle's Algorithm). There's a ton of advice on the internet where people are promoting the usage of this socket option as a panacea for network latency. Often the posts I've seen suggesting that `TCP_NODELAY` will be helpful are actually technically correct but miss the larger picture of the context in which it's appropriate to use this option, and generally how to write high performance socket programs.

This came up today at work. Someone was trying to load test their service with a program written using the Tornado HTTP client. This library comes in two varieties: a high performance libcurl binding and a pure Python version that just uses the low level Tornado IOStream stuff. In both cases the benchmark was getting poor latency numbers even though the benchmark was set up with a concurrency of ten. The engineer writing the load test refactored it to use the Python multiprocessing module with the regular blocking urllib2 library and got a huge speedup (again, with concurrency ten), in line with the expectation for the service.

A bunch of theories were thrown back and forth about why this was happening. One of the engineers looking at this had read Julia Evans' recent blog post about TCP_NODELAY and suggested that this may be the culprit. The engineer even found a related Tornado Github issue where Tornado wasn't setting `TCP_NODELAY` in earlier versions. There was some confusion about which Tornado version we were using in the benchmarks, whether urllib2 was using `TCP_NODELAY`, and so on.

I was kind of skeptical that `TCP_NODELAY` was related for a few reasons. First, in my experience most applications that use this flag are not written correctly in the first place. Indeed, in the blog post Julia references discussing the issue with Ruby's `Net::HTTP` module, you can see right off the bat that the Ruby library is not written well. To wit:

> Ruby's Net::HTTP splits POST requests across two TCP packets - one for the headers, and another for the body. curl, by contrast, combines the two if they'll fit in a single packet.

Unnecessarily sending data out in multiple packets here is bad programming practice in the first place (more on this in a second). The benchmark that came up at work was making HTTP GET requests which don't have bodies and should fit in a single packet, even if the library was written sloppily. Additionally one variation of the Tornado load tester was using the libcurl bindings, and as noted in this post libcurl properly buffers its writes to send full packets.

In this post I'm going to discuss:

- How the TCP stack in your kernel works, as well as how the protocol itself works. Too often people understand how the protocol works without understanding the kernel layer which leads to sloppy code (like the Ruby `Net::HTTP` example).
- What to do before turning on `TCP_NODELAY`.
- When it is appropriate to use `TCP_NODELAY`.
- A fun and related socket option on Linux, `TCP_CORK`.

A Brief Overview of TCP and the Kernel

The programming interface exposed to you by the kernel for TCP sockets (actually, all `SOCK_STREAM` sockets) is a file descriptor that you can `write(2)` and `read(2)` on. As you might guess, the `write(2)` system call sends data, and the `read(2)` system call reads data.

In reality how this works is that the kernel has two buffers allocated for the file descriptor in kernel memory: a write buffer and a read buffer. The `write(2)` system call doesn't actually cause TCP packets to get sent; what it actually does is copy data into the kernel's write buffer for the socket. Likewise `read(2)` fetches data out of the read buffer (or blocks if the read buffer is empty). There are some further details about blocking and non-blocking sockets and other advanced socket options, but those details aren't relevant for this discussion.

When there is unsent data in the write buffer the kernel tries to send it out over the network. This takes the form of a data payload wrapped in a TCP packet, which itself is wrapped in an IP packet, which itself is wrapped in a link layer packet (e.g. an Ethernet frame). The time when this TCP packet it sent is potentially much later than when you actually issued the `write(2)` call that copied data into the outgoing write buffer. You can't use the `write(2)` call to know if the data has been sent (and even if you could, sent data can be dropped or lost and may need to be transmitted anyway).

The TCP protocol itself has a rule for when the kernel is allowed to send TCP packets. There's a window size for the TCP connection that tracks how many unacked packets can be in transit at any time. When this window size is hit, data written by the application will be copied into the kernel write buffer but not sent immediately. There are two cases when an application is likely to run into this: (1) new TCP connections start with a tiny window size and therefore write delays are common, and (2) applications that are actually saturating the network bandwidth will eventually run up against the window size. The situation relevant here is (1), the behavior of a new TCP connection. In the original RFC for TCP the window size started at one packet. This means that the first `write(2)` call on a new TCP stream will cause a packet to be immediately sent, but a quick second `write(2)` will not cause data to be sent if the first packet hasn't been acked. This has been somewhat fixed; RFC 3390 increases the initial window size to up to four packets, and there have been efforts to increase that even further.

Besides the TCP rules for when it is permissible to send packets, there are some recommendations for how TCP stacks should work. In some cases your operating system may choose to delay sending TCP packets even when the current window size would allow a packet to be sent. There are two relevant options here: Nagle's algorithm (which can be disabled by `TCP_NODELAY`) and TCP delayed acknowledgement (which on Linux sort of can be controlled by `TCP_QUICKACK`, with some caveats).

Nagle's algorithm notes that a userspace application might be making lots of small calls to `write(2)` it may be beneficial to have the kernel try to coalesce the writes. If the kernel naively translated all of these writes into TCP packets then the kernel would end up sending a lot of small packets. For instance, consider the statsd protocol. In this protocol a stat is sent by the stat name, a value, and optionally a "format". A typical stat sent in this protocol is very small; say, 50 bytes. An application that is trying to send 10 stats might therefore issue 10 calls to `write(2)` each sending 50 bytes. It's easy to imagine how this would be the case: you structure your library so there's a `send_stat()` function which writes the data, and then call that function 10 times. What Nagle's algorithm does is says: if there is unacked data sent, and if the write buffer in the kernel is smaller than the MTU, then wait a little to see if the application writes more data. If the write buffer reaches the MTU size then the data will be transmitted. If the in flight data is acked then the data will also be transmitted, even if it is smaller than the MTU. In the example I just gave, this would mean that the kernel will automatically coalesce the stats sent into a single 500 byte packet. As you can see, this is a really good thing, and generally not something you want to disable.

The TCP delayed ack feature is again an attempt to minimize the number of small packets sent. The way it works is a TCP packet can ack multiple data packets at once. Therefore a TCP stack implementing the delayed ack feature may wait up to some amount of time before acking packets in the hope that it will be able to ack more packets at once. On Linux this can cause up to a 40 ms delay when acking packets. Again, this is usually a good thing since it decreases the number of packets that have to be sent (which is usually the limiting factor in network performance).

The Problem

The problem happens if Nagle's algorithm causes a sent packet to be delayed *and* when the sending program needs the packet to be sent before it can read data on the connection. This situation is actually problematic even if Nagle's algorithm alone is on, but delayed acks amplify the problem.

If a program writes a single packet and then reads, then the packet will be transmitted immediately (since there are is no unacked data the rule about unfull buffers doesn't apply). The opposite end will process the packet and then send its response (which will include an ack flag). Thus, there is no unnecessary delay.

If a program issues multiple writes and doesn't have to read anything then there is no problem. The worse case here is that delayed ack is turned on which means that at the end of the write sequence there is at most a 40 ms artificial delay before the sender knows the data is acked. However if the sender isn't blocked while it waits for a response (e.g. because there are no responses, or because the communication is fully bidirectional) then this won't be noticed.

But consider what happens when a program issues multiple writes followed by a read that is dependent on the write data. The final write may queue due to Nagle's algorithm. The other end won't process the request (and therefore will not start sending a response) because it's waiting for the final packet of data. The other end also won't ack the data it got until the delayed ack timer fires. When this finally does happen the sending program gets its acks and is finally able to actually transmit its last packet. This more or less guarantees the 40 ms delay (unless the write payload just happened to exactly be a multiple of the MTU, which is unlikely), a delay which will be noticed in many applications.

When Not To Use TCP_NODELAY

Off the bat, you should try to send as much data to the kernel as possible in `write(2)` system calls. If that's difficult to do because of how requests are crafted then you should investigate the `writev(2)` system call which implements "vectorized I/O". This makes it easier to write applications that generate data to be sent into multiple small buffers and want to efficiently copy the data into the kernel.

In the Ruby example, if the `Net::HTTP` module issued a single `write(2)` or `writev(2)` call then the POST request would have likely fit into a single packet which would have avoided the need to use `TCP_NODELAY`. It's still possible that even if this happened the POST body could be sufficiently large to cause the request to be split. However this shouldn't be too common for most web applications. A good rule of thumb is you can typically fit at least 1400 bytes of payload before the packet will be split.

When And How to Use TCP_NODELAY

An application that is very latency sensitive, particularly if it doesn't transmit a lot of data, can safely use `TCP_NODELAY`. A good example here would be an SSH session. In an SSH session it's common that the user will be typing input to the remote server, and the user will certainly be typing at a slow pace compared to the bandwidth capabilites of the network. If Nagle's algorithm is on then that means that keystrokes will end up being delayed frequently. This will make a lot of applications difficult to use—imagine using vi or emacs with Nagle's algorithm on.

If you understand this and you want to enable `TCP_NODELAY` anyway, then you definitely can as along as you follow one rule: you must make your best effort to send large writes to the kernel. That is, it is incumbent on you to ensure that small amounts of data in the send buffer are actually because there's no more data to be sent, not because your application is doing something hacky and is issuing lots of small `write(2)` system calls. Again I'd like to encourage the use of vectorized I/O here. For instance if you have a line oriented protocol you may find it easiest to structure your program by creating multiple buffers each storing a single line, and then you can transmit all of the data at once in a single syscall (without unnecessary memory copies!) using `writev(2)`.

Multi-threaded programs that share sockets between threads without locks are particularly susceptible to problems related to Nagle's algorithm. This is because even if the application is structured so that a single thread doesn't do the write-write-read pattern, if there is one writer thread and another reader thread then the pattern can still manifest. This is one good argument for not sharing sockets between threads without mutexes; usually a better option is to use an event loop.

Note that if you are using a proxy like HAProxy then all bets are off. Even if you disable Nagle's algorithm, the proxy could have it on.

TCP_CORK, or, Reverse TCP_NODELAY

Linux implements an interesting option called `TCP_CORK`. You can think of it as an even stronger variation of Nagle's algorithm. It says: if there is data in the write buffer, and that data is less than the MTU size, and if there is no unacked data, continue to delay sending the write buffer data up to 200ms in the hope that more data will enter the write buffer. In other words: it tells the kernel to try extra hard to delay sending small packets, allowing the kernel to delay sending such packets by up to 200ms.

This is something that I added to statsrelay which *significantly* increased the throughput of that program. Prior to using this option many programs would issue tons of small writes into statsrelay. This would cause lots of small packets to be sent over the network (even though `TCP_NODELAY` was not on). In some cases we had network interfaces that were reaching their max packets-per-second due to all of these small packets even though the total bandwidth was well below what the interface could handle. After turning on `TCP_CORK` the number of packets per second the statsd boxes received dropped dramatically (up to 10x if I recall correctly). This comes at the cost of some latency, but the underlying data store (Carbon) only supported second resolution for data anyway so in practice this was not a problem.

A Final Note

If you made it this far, you may wonder what the original problem was with the Tornado load testing application. It ended up actually being an issue with how the futures were being created and collected. All of the possible requests the application could make were created as futures at once with timestamp of when the future was created. However, only ten futures at a time would actually be active. This meant that the last futures being collected were showing long response times even though the actual requests were much faster.