# TCP Performance problems caused by interaction between Nagle’s Algorithm and Delayed ACK

Stuart Cheshire  
20th May 2005

This page describes a TCP performance problem resulting from a little-known interaction between Nagle’s Algorithm and Delayed ACK. At least, I believe it’s not well known: I haven’t seen it documented elsewhere, yet in the course of my career at Apple I have run into the performance problem it causes over and over again (the first time being the in the [PPCToolbox over TCP](http://www.stuartcheshire.org/quicktime/ppctb.html) code I wrote back in 1999), so I think it’s about time it was documented.

The overall summary is that many of the mechanisms that make TCP so great, like fast retransmit, work best when there’s a continuous flow of data for the TCP state machine to work on. If you send a block of data and then stop and wait for an application-layer acknowledgment from the other end, the state machine can falter. It’s a bit like a water pump losing its prime — as long as the whole pipeline is full of water the pump works well and the water flows, but if you have a bit of water followed by a bunch of air, the pump flails because the impeller has nothing substantial to push against.

If you’re writing a request/response application-layer protocol over TCP, the implication of this is you will want to make your implementation at least double-buffered: Send your first request, and while you’re still waiting for the response, generate and send your second. Then when you get the response for the first, generate and send your third request. This way you always have two requests outstanding. While you’re waiting for the response for request *n*, request *n*+1 is behind it in the return pipeline, conceptually pushing the data along.

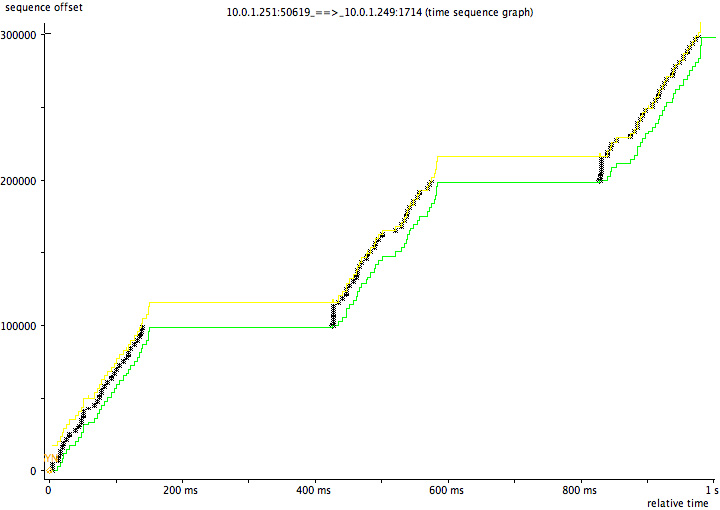
Interestingly, the performance benefit you get from going from single-buffering to double, triple, quadruple, etc., is not a linear slope. Implementing double-buffering usually yields almost all of the performance improvement that’s there to be had; going to triple, quadruple, or n-way buffered usually yields little more. This is because what matters is not the sheer quantity of data in the pipeline, but the fact that there is something in the pipeline following the packet you’re currently waiting for. As long as there’s at least a packet’s worth of data following the response you’re waiting for, that’s enough to avoid the pathological slowdown described here. As long as there’s at least four or five packet’s worth of data, that’s enough to trigger a fast retransmit should the the packet you’re waiting for be lost.

## Case Study: WiFi Performance Testing Program

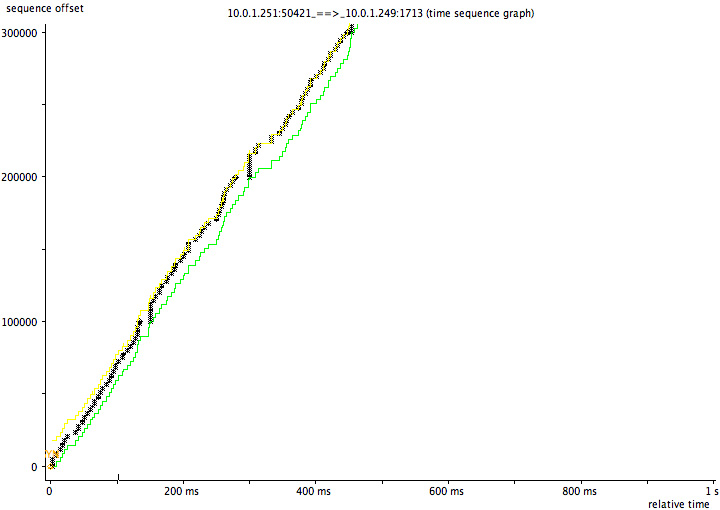
In this document I describe my latest encounter with this bad interaction between Nagle’s Algorithm and Delayed ACK: a testing program used in WiFi conformance testing. This program tests the speed of a WiFi implementation by repeatedly sending 100,000 bytes of data over TCP and then waiting for an application-layer ack from the other to confirm its reception. Windows achieved the 3.5Mb/s required to pass the test; Mac OS X got just 2.7Mb/s and failed. The naive (and wrong) conclusion would be that Windows is fast, and Mac OS X is slow, and that’s that. The truth is not so simple. The true explanation was that the test was flawed, and Mac OS X happened to expose the problem, while Windows, basically through luck, did not.

Engineers found that reducing the buffer size from 100,000 bytes to 99,912 bytes made the measured speed jump to 5.2Mb/s, easily passing the test. At 99,913 bytes the test got 2.7Mb/s and failed. Clearly what’s going on here is more interesting than just a slow wireless card and/or driver.

The diagram below shows a TCP packet trace of the failing transfer, which achieves only 2.7Mb/s, generated using [tcptrace](http://www.tcptrace.org/) and jPlot:



This diagram shows a TCP packet trace of the transfer using 99912-byte blocks, which achieves 5.2Mb/s and passes:



In the failing case the code is clearly sending data successfully from 0-200ms, then doing nothing from 200-400ms, then sending again from 400-600ms, then nothing again from 600-800ms. Why does it keep pausing? To understand that we need to understand Delayed ACK and Nagle’s Algorithm:

### Delayed ACK

Delayed ACK means TCP doesn’t immediately ack every single received TCP segment. (When reading the following, think about an interactive ssh session, not bulk transfer.) If you receive a single lone TCP segment, you wait 100-200ms, on the assumption that the receiving application will probably generate a response of some kind. (E.g., every time sshd receives a keystroke, it typically generates a character echo in response.) You don’t want the TCP stack to send an empty ACK followed by a TCP data packet 1ms later, every time, so you delay a little, so you can combine the ACK and data packet into one. So far so good. But what if the application doesn’t generate any response data? Well, in that case, what difference can a little delay make? If there’s no response data, then the client can’t be waiting for anything, can it? Well, the application-layer client can’t be waiting for anything, but the TCP stack at the end can be waiting: This is where Nagle’s Algorithm enters the story:

### Nagle’s Algorithm

For efficiency you want to send full-sized TCP data packets. Nagle’s Algorithm says that if you have a few bytes to send, but not a full packet’s worth, and you already have some unacknowledged data in flight, then you wait, until either the application gives you more data, enough to make another full-sized TCP data packet, or the other end acknowledges all your outstanding data, so you no longer have any data in flight.

Usually this is a good idea. Nagle’s Algorithm is to protect the network from stupid apps that do things like this, where a naive TCP stack might end up sending 100,000 one-byte packets.

for (i=0; i<100000; i++) write(socket, &buffer[i], 1);

The bad interaction is that now there is something at the sending end waiting for that response from the server. That something is Nagle’s Algorithm, waiting for its in-flight data to be acknowledged before sending more.

The next thing to know is that Delayed ACK applies to a single packet. If a second packet arrives, an ACK is generated immediately. So TCP will ACK every second packet immediately. Send two packets, and you get an immediate ACK. Send three packets, and you’ll get an immediate ACK covering the first two, then a 200ms pause before the ACK for the third.

Armed with this information, we can now understand what’s going on. Let’s look at the numbers:

99,900 bytes = 68 full-sized 1448-byte packets, plus 1436 bytes extra

100,000 bytes = 69 full-sized 1448-byte packets, plus 88 bytes extra

With 99,900 bytes, you send 68 full-sized packets. Nagle holds onto the last 1436 bytes. Then:

* Your 68th packet (an even number) is immediately acknowledged by the receiver without delay
* Nagle gets the ACK and releases the remaining 1436 bytes
* The receiver gets the remaining 1436 bytes, and delays its ACK
* the server application generates its one-byte application-layer ack for the transfer,
* Delayed ACK combines that one byte with its pending ACK packet, sends the combined TCP ACK+data packet promptly,

...and everything flows (relatively) smoothly.

Now consider the 100,000-byte case. You send a stream of 69 full-sized packets. Nagle holds onto the last 88 bytes. Then:

* The receiver ACKs every second packet, up to packet 68.
* One more data packet arrives, packet 69.
* Delayed ACK means that the receiver won’t ACK this packet until it gets   
  (a) some response data from the local process, or   
  (b) another packet from the sender.
* The local process won’t generate any response data (a) because it hasn’t got the full 100,000 bytes yet.
* The sender won’t send the last packet (b) because Nagle won’t let it until it gets an ACK from the receiver.

Now we have a brief deadlock, with performance-killing results:

* Nagle won’t send the last bit of data until it gets an ACK
* Delayed ACK won’t send that ACK until it gets some response data
* Server process won’t generate any response until it gets all the data
* but, Nagle won’t send the last bit of data... and so on.

So, at the end of each 100,000-byte transfer we get this little awkward pause. Finally the delayed ack timer goes off and the deadlock un-wedges, until next time. On a gigabit network, all these huge 200ms pauses can be devastating to an application protocol that runs into this problem. These pauses can limit a request/response application-layer protocol to at most five transactions per second, on a network link where it should be capable of a thousand transactions per second or more. In the case of this specific attempt at a performance test, it should have been able to, in principle, transfer each 100,000-bytes chunk over a local gigabit Ethernet link in as little as 1ms. Instead, because it stops and waits after each chunk, instead of double-buffering as recommended above, sending each 100,000-byte chunk takes 1ms + 200ms pause = 201ms, making the test run roughly two hundred times slower than it should.

### Why didn’t Windows suffer this problem?

On Windows the TCP segment size is 1460 bytes. On Mac OS X and other operating systems that add a TCP time-stamp option, the TCP segment size is twelve bytes smaller: 1448 bytes.

What this means is that on Windows, 100,000 bytes is 68 full-sized 1460-byte packets plus 720 extra bytes. Because 68 is an even number, by pure luck the application avoids the Nagle/Delayed ACK interaction.

On Mac OS X, 100,000 bytes is 69 full-sized 1448-byte packets, plus 88 bytes extra. Because 69 is an odd number, Mac OS X exposes the application problem.

A crude way to solve the problem, though still not as efficient as the double-buffering approach, is to make sure the application sends each semantic message using a single large write (either by copying the message data into a contiguous buffer, or by using a scatter/gather-type write operation like sendmsg) and set the TCP\_NODELAY socket option, thereby disabling Nagle’s algorithm. This avoids this particular problem, though it can still suffer other problems inherent in not using double buffering — like, if the last packet of a response gets lost, there are no packets following it to trigger a fast retransmit.

### Update, November 2012. What’s the current state of this problem?

While the double-buffering approach advocated here (keep making forward progress instead of blocking and waiting until each individual operation completes one at a time) remains the best solution, advances in networking code have now mitigated this specific Nagle’s algorithm deadlock.

Mac OS X v10.5 “Leopard”, released October 2007, and later updates, including iOS, implement Greg Minshall’s [“Proposed Modification to Nagle’s Algorithm”](http://tools.ietf.org/html/draft-minshall-nagle), which he documented back in December 1998. I do not know the implementation status in other operating systems.

Nagle’s algorithm says that if you have any unacknowledged data outstanding, then you can’t send a runt packet. Minshall’s Modification softens this rule a little to say that if you already have an unacknowledged *runt packet* outstanding, then you can’t send another. In other words, you’re allowed one free pass. You’re allowed to have one runt packet in flight at a time, but only one. Multiple runt packets still not allowed because of the harm that would do to the network. A badly-written application that’s doing lots of small writes back-to-back will still have those writes coalesced into more efficient large packets, but an application that does a single large write will not have the tail end of that write delayed. The single runt packet at the end of a large write doesn’t get delayed, because the preceding outstanding packets are all full-sized packets.

The importance of not disabling Nagle’s algorithm was spelled out in [Greg Minshall’s email](http://lists.w3.org/Archives/Public/ietf-discuss/1998Dec/0025.html):

**Please note that while, in certain cases, the current Nagle algorithm can have a negative performance impact for certain applications, turning OFF the Nagle algorithm can have a very serious negative impact on the internet. Thus, nothing in this e-mail message, or in the enclosed draft, should be taken as advice to any application developer to disable the Nagle algorithm. The current Nagle algorithm is very important in protecting the health of the internet; the proposed modification [hopefully] provides the same level of protection.**

**Copy from: www.stuartcheshire.org/papers/NagleDelayedAck/**