Networking

- How the Kitchen Sink and Statistics Explain and Treat Dropped Packets
- WireGuard
- Wifi
- Generate Random MAC address

How the Kitchen Sink and Statistics Explain and Treat Dropped Packets

Posted by dougb in Wired Ethernet on Nov 4, 2009 12:18:30 PM source

It has been said there are lies, darn lies and statistics. Well here in the Wired Ethernet world, we tend to frown on that saying. The statistics can be down right useful in figuring out problems, in either your software or your network. Today I'll look at using the stats for maximizing the performance of your implementation when it comes to dropped packets. And a kitchen sink will show us the way. Hope you brought your towel.

The Intel 1 Gigabit productshave two sets of stats that are useful in this regards. First is the Receive No Buffer Count or RNBC. It will increment when a frame has been successfully loaded into the FIFO, but can't get out to host memory where the buffers are because there are no free buffers to put it into.

Second is the Missed Packets Count, or MPC. This is the count of frames that were discarded because there was no room in the MAC FIFO to store the frames before they were DMA'ed out to host memory. You will typically see RNBC growing before you will see MPC grow. But, and this is a key point, you don't need to have a event increment RNBC before you can MPC increment.

First a primer on the MAC architecture that Intel Wired Networking uses. Coming from the physical layer, either Copper or SerDes (or other), the packet will be stored in the MAC FIFO. It gets processed to get there, and it gets processed some more before going to the DMA block for the actual trip out to host memory. If there is a buffer available, the DMA block sends it on its way. The descriptor associated with the buffer is updated and the world is good. Well good enough. In the case RNBC, the frame is happily in the FIFO, but without a buffer to head home to, it has no where to go. In the MPC case, the poor frame can't even get into the FIFO and is dropped off the wire. MPC is also sometimes called an overrun, because the FIFO is overrun with data. An underrun is a TX error, so that's out of bounds for this talk. Plus they are pretty rare these days.

As you can tell, RNBC is not too bad, but points to bigger problems. MPC is pretty bad, because you are dropping frames. So how can you have MPC without RNBC? Imagine if you will, an interconnect bus that is slow. Very slow. Like a PCI 33hz bus. Now attach that to a full line rate 1 Gigabit 64 byte packet data stream. At one descriptor per packet, that's about 1.4 million descriptors per second. In this case the software is very fast, faster than the bus. So the number of available descriptors is always kept a level that keeps the buffers available to the hardware to conduct a DMA. But

because the bus is so slow, data backs up into the FIFO. Now that is what the FIFO is for. By buffering the packet, it tries to give the packet the best chance at making into host memory alive. In our slow case, the buffering isn't enough and the FIFO fills up. It is draining slower than its filling, its just a like a slow draining kitchen sink. Eventually it overflows and makes a big mess. Thank goodness things like TCP/IP will tell the applications data has been dropped, but if using a lossy frame type like UDP its just too bad, your frame is lost to the ether. If you need to keep track, but need to use UDP, you'll need to monitor the MPC count and decide what you want to do when it goes up.

As already noted, RNBC doesn't always lead to MPC, but it is a warning flag that it will happen. Here is how the RNBC can climb while MPC stays low. Imagine we have a slow CPU, but a wicked fastbus. The software is very slow to process the descriptors and return them, but once the descriptors are given to the hardware, it empties the backlog (read the FIFO) faster than the incoming frames are filling the FIFO. Returning to our kitchen sink analogy, the water is coming in at a fairly constant rate. But imagine the stopper is down, making the sink fill up. Just before it over flows, the drain is opened and down it goes. Once the water doesn't go down the drain would be the same moment our RNBC would be incremented. The kitchen sink itself becomes our FIFO and if the FIFO is big enough, it can save frame for quiet some time. This is 1 Gigabit (or faster) that we're talking about, so with a good sized FIFO (24K RX for example) that's only 375 frames at 64 bytes, or 267microseconds of data. That's not very much time. But in a world full of 2 and 3 Gigahertz CPUs that's long enough. If you have 2048 descriptors for it to dump into, that is almost 8 times the amount of packet time before the FIFO starts filling up.

And you're probably sitting there saying "I was told there wasn't going to be math on this blog!" Moving on(and that's enough about the bad news), lets talk about the good news. Both RNBC and MPC are either treatable in software, or can be minimized by careful design. RNBC is really a software problem at its core, but a fast bus never hurt. If you‘re getting the RNBC moving up, add more descriptors and buffers. Make sure your ISR or polling loop is running often enough to get back to the business of adding more resources to the card before the descriptors stash runs out. Using our example from above, if your expecting 64 byte frames, you'll need to poll every millisecond or so if you have 2048 descriptors. Looking at it from the other direction, if your trying to do 1 Gigabit of traffic with only 8 descriptors, RNBC is going to jump around like a cat in a rocking chair store. Consult your documentation (link goes to the 8257x Open Source SDM) on how to add more descriptors, all our major O/S products support it. There may be times when you've added all the buffers the driver will let you and your still seeing RNBC errors. When this happens, it's a sign that the stack might be the limit. In modern operating systems, the buffers are O/S buffers and while we might have 2048 of them, if the O/S has ownership of 2047 of them, RNBC will be just part of life. Most stacks have their own buffers that you can tinker with their count, so that can help. Check the stats of your stack to see if it is having troubles keeping up. There will be times when the RNBC will go up, but it will look like the stack and driver have a ton of buffers but work is not being done. If you have a task that is eating up the CPU, the ISR or polling routines won't refill the buffers fast enough and RNBC will happen.

MPC is treatable depending on what RNBC is doing. If RNBC isn't moving around much, there is room for the data, its just not getting out of the FIFO fast enough. Much like the movie where if the bus goes below a certain speed bad things will happen, the same applies to MPC. Maybe without all

the flash of a Hollywood movie, but the principle is the same. The bus is the limit. Move to a more traffic friendly slot. While slot topology and its impact on performance is a whole 'nother post; it's only common sense that a x4 card may drop frames if put into a x1 connected slot. Give that card room to DMA and most MPC errors will go away when the slot speed matches the maximum speed the card supports. If you have MPC and RNBC climbing at the same rate, most likely the bus isn't the limit, the buffer reload speed is. Treat the RNBC issue first and then see if MPC is still going out of control.

Even with a super fast bus and a ton of buffers, there will be times when RNBC will happen and there will be time when MPC happen. Sometimes a big burst of traffic comes just as the descriptor count gets low, sometimes the ISR doesn't run exactly when you hoped it would. The trick is not letting either one be a big percentage of the total number of packets. When it does get out of control, follow this post and you'll see improvement in the percentage.

Time for the big finish.

1. RNBC is a warning sign of a slow drain from the MAC and can be treated by adding more buffers.
2. MPC is a failure condition leading to dropped packets and can be treated with more buffers and faster interconnect buses.
3. Thanks for using Intel networking products.

WireGuard

WireGuard® is an extremely simple yet fast and modern VPN that utilizes state-of-the-art cryptography. It aims to be faster, simpler, leaner, and more useful than IPsec, while avoiding the massive headache. It intends to be considerably more performant than OpenVPN. WireGuard is designed as a general purpose VPN for running on embedded interfaces and super computers alike, fit for many different circumstances. Initially released for the Linux kernel, it is now cross-platform (Windows, macOS, BSD, iOS, Android) and widely deployable.

Generating Keys

The communication protocols and cryptography is different than SSH but the concepts are the same where both public and private keys are generated by both the client and server and exchanged prior to communication.

All traffic to the server is encrypted with the `server-public-key` and can only be decrypted with the `server-private-key`. Similarly all traffic to the client is encrypted with the `client-public-key` and can only be decrypted wtih the `client-private-key`.

The steps for both server and client are similar

1. set the umask so all the files we're about to create to be 700 (rwx------)
2. use `wg` to generate a private key and write it to a file and pipe the content to `wg` to generate a public key that is also written to a file

Server

```
# umask 077
# wg genkey | tee server.key | wg pubkey > server.pub
# cat server.key
<server-private-key>
# cat server.pub
<server-public-key>
```

Client

```
# umask 077
# wg genkey | tee client.key | wg pubkey > client.pub
```

```
# cat client.key
<client-private-key>
# cat client.pub
<client-public-key>
```

Server Configuration

```
[Interface]
PrivateKey = <server-private-key>
Address = 10.0.99.1/24
ListenPort = 51820

[Peer]
PublicKey = <client-public-key>
AllowedIPs = 10.0.99.2/32
```

Client Configuration

```
[Interface]
PrivateKey = <client-private-key>
Address = 10.0.99.2/32
ListenPort = 51820

[Peer]
PublicKey = <server-public-key>
AllowedIPs = 0.0.0.0/0
```

Generate PresharedKey (optional)

If an additional layer of symmetric-key crypto is required (for, say, post-quantum resistance), WireGuard also supports an optional pre-shared key that is mixed into the public key cryptography. When pre-shared key mode is not in use, the pre-shared key value used below is assumed to be an all-zero string of 32 bytes.

```
# wg genpsk
<psk>
```

Add the same PresharedKey parameter to both `[Peer]` sections in server and client configuration files.

```
[Peer]
...
PresharedKey = <psk>
```

Wifi

Signal Strength

Signal Strength Basics

Channels

Understanding Channels

Generate Random MAC address

```
printf '00:60:2f:%02x:%02x:%02x\n' $[RANDOM%256] $[RANDOM%256] $[RANDOM%256]
```