Red Hat Enterprise Linux Network Performance Tuning Guide

Authors: Jamie Bainbridge and Jon Maxwell
Reviewer: Noah Davids
Editors: Dayle Parker and Chris Negus
03/25/2015

Tuning a network interface card (NIC) for optimum throughput and latency is a complex process with many factors to consider.

These factors include capabilities of the network interface, driver features and options, the system hardware that Red Hat Enterprise Linux is installed on, CPU-to-memory architecture, amount of CPU cores, the version of the Red Hat Enterprise Linux kernel which implies the driver version, not to mention the workload the network interface has to handle, and which factors (speed or latency) are most important to that workload.

There is no generic configuration that can be broadly applied to every system, as the above factors are always different.

The aim of this document is not to provide specific tuning information, but to introduce the reader to the process of packet reception within the Linux kernel, then to demonstrate available tuning methods which can be applied to a given system.

PACKET RECEPTION IN THE LINUX KERNEL

The NIC ring buffer

Receive ring buffers are shared between the device driver and NIC. The card assigns a transmit (TX) and receive (RX) ring buffer. As the name implies, the ring buffer is a circular buffer where an overflow simply overwrites existing data. It should be noted that there are two ways to move data from the NIC to the kernel, hardware interrupts and software interrupts, also called SoftIRQs.

The RX ring buffer is used to store incoming packets until they can be processed by the device driver. The device driver drains the RX ring, typically via SoftIRQs, which puts the incoming packets into a kernel data structure called an **`sk_buff`** or “skb” to begin its journey through the kernel and up to the application which owns the relevant socket. The TX ring buffer is used to hold outgoing packets which are destined for the wire.

These ring buffers reside at the bottom of the stack and are a crucial point at which packet drop can occur, which in turn will adversely affect network performance.

Interrupts and Interrupt Handlers

Interrupts from the hardware are known as “top-half” interrupts. When a NIC receives incoming data, it copies the data into kernel buffers using DMA. The NIC notifies the kernel of this data by

raising a hard interrupt. These interrupts are processed by interrupt handlers which do minimal work, as they have already interrupted another task and cannot be interrupted themselves. Hard interrupts can be expensive in terms of CPU usage, especially when holding kernel locks.

The hard interrupt handler then leaves the majority of packet reception to a software interrupt, or SoftIRQ, process which can be scheduled more fairly.

Hard interrupts can be seen in **`/proc/interrupts`** where each queue has an interrupt vector in the 1st column assigned to it. These are initialized when the system boots or when the NIC device driver module is loaded. Each RX and TX queue is assigned a unique vector, which informs the interrupt handler as to which NIC/queue the interrupt is coming from. The columns represent the number of incoming interrupts as a counter value:

```
# egrep “CPU0|eth2” /proc/interrupts
           CPU0       CPU1       CPU2       CPU3       CPU4       CPU5
 105:  141606          0          0          0          0          0  IR-PCI-MSI-edge      eth2-rx-0
 106:       0     141091          0          0          0          0  IR-PCI-MSI-edge      eth2-rx-1
 107:       2          0     163785          0          0          0  IR-PCI-MSI-edge      eth2-rx-2
 108:       3          0          0     194370          0          0  IR-PCI-MSI-edge      eth2-rx-3
 109:       0          0          0          0          0          0  IR-PCI-MSI-edge      eth2-tx
```

SoftIRQs

Also known as “bottom-half” interrupts, software interrupt requests (SoftIRQs) are kernel routines which are scheduled to run at a time when other tasks will not be interrupted. The SoftIRQ's purpose is to drain the network adapter receive ring buffers. These routines run in the form of **`ksoftirqd/cpu-number`** processes and call driver-specific code functions. They can be seen in process monitoring tools such as **`ps`** and **`top`**.

The following call stack, read from the bottom up, is an example of a SoftIRQ polling a Mellanox card. The functions marked **`[mlx4_en]`** are the Mellanox polling routines in the **`mlx4_en.ko`** driver kernel module, called by the kernel's generic polling routines such as **`net_rx_action`**. After moving from the driver to the kernel, the traffic being received will then move up to the socket, ready for the application to consume:

```
  mlx4_en_complete_rx_desc [mlx4_en]
  mlx4_en_process_rx_cq [mlx4_en]
  mlx4_en_poll_rx_cq [mlx4_en]
  net_rx_action
  __do_softirq
  run_ksoftirqd
  smpboot_thread_fn
  kthread
  kernel_thread_starter
  kernel_thread_starter
1 lock held by ksoftirqd
```

SoftIRQs can be monitored as follows. Each column represents a CPU:

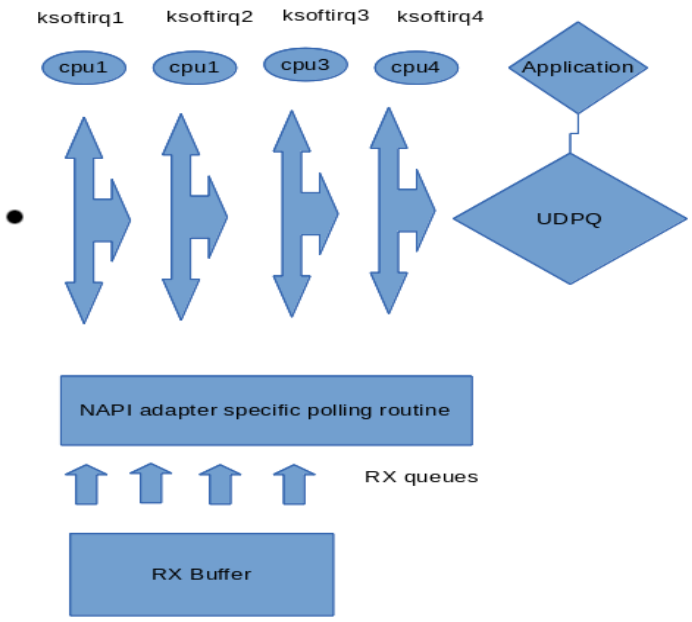
```
# watch -n1 grep RX /proc/softirqs
# watch -n1 grep TX /proc/softirqs
```

NAPI Polling

NAPI, or New API, was written to make processing packets of incoming cards more efficient. Hard interrupts are expensive because they cannot be interrupted. Even with interrupt coalescence (described later in more detail), the interrupt handler will monopolize a CPU core completely. The design of NAPI allows the driver to go into a polling mode instead of being hard-interrupted for every required packet receive.

Under normal operation, an initial hard interrupt or IRQ is raised, followed by a SoftIRQ handler which polls the card using NAPI routines. The polling routine has a budget which determines the CPU time the code is allowed. This is required to prevent SoftIRQs from monopolizing the CPU. On completion, the kernel will exit the polling routine and re-arm, then the entire procedure will repeat itself.

Figure1: SoftIRQ mechanism using NAPI poll to receive data



Network Protocol Stacks

Once traffic has been received from the NIC into the kernel, it is then processed by protocol handlers such as Ethernet, ICMP, IPv4, IPv6, TCP, UDP, and SCTP.

Finally, the data is delivered to a socket buffer where an application can run a receive function, moving the data from kernel space to userspace and ending the kernel's involvement in the receive process.

Packet egress in the Linux kernel

Another important aspect of the Linux kernel is network packet egress. Although simpler than the ingress logic, the egress is still worth acknowledging. The process works when skbs are passed down from the protocol layers through to the core kernel network routines. Each skb contains a **dev** field which contains the address of the **net_device** which it will transmitted through:

```
int dev_queue_xmit(struct sk_buff *skb)
{
        struct net_device *dev = skb->dev; <--- here
        struct netdev_queue *txq;
        struct Qdisc *q;
```

It uses this field to route the skb to the correct device:

```
   if (!dev_hard_start_xmit(skb, dev, txq)) {
```

Based on this device, execution will switch to the driver routines which process the skb and finally copy the data to the NIC and then on the wire. The main tuning required here is the TX queueing discipline (qdisc) queue, described later on. Some NICs can have more than one TX queue.

The following is an example stack trace taken from a test system. In this case, traffic was going via the loopback device but this could be any NIC module:

```
0xffffffff813b0c20 : loopback_xmit+0x0/0xa0 [kernel]
0xffffffff814603e4 : dev_hard_start_xmit+0x224/0x480 [kernel]
0xffffffff8146087d : dev_queue_xmit+0x1bd/0x320 [kernel]
0xffffffff8149a2f8 : ip_finish_output+0x148/0x310 [kernel]
0xffffffff8149a578 : ip_output+0xb8/0xc0 [kernel]
0xffffffff81499875 : ip_local_out+0x25/0x30 [kernel]
0xffffffff81499d50 : ip_queue_xmit+0x190/0x420 [kernel]
0xffffffff814af06e : tcp_transmit_skb+0x40e/0x7b0 [kernel]
0xffffffff814b0ae9 : tcp_send_ack+0xd9/0x120 [kernel]
0xffffffff814a7cde : __tcp_ack_snd_check+0x5e/0xa0 [kernel]
0xffffffff814ad383 : tcp_rcv_established+0x273/0x7f0 [kernel]
0xffffffff814b5873 : tcp_v4_do_rcv+0x2e3/0x490 [kernel]
0xffffffff814b717a : tcp_v4_rcv+0x51a/0x900 [kernel]
0xffffffff814943dd : ip_local_deliver_finish+0xdd/0x2d0 [kernel]
0xffffffff81494668 : ip_local_deliver+0x98/0xa0 [kernel]
0xffffffff81493b2d : ip_rcv_finish+0x12d/0x440 [kernel]
0xffffffff814940b5 : ip_rcv+0x275/0x350 [kernel]
0xffffffff8145b5db : __netif_receive_skb+0x4ab/0x750 [kernel]
0xffffffff8145b91a : process_backlog+0x9a/0x100 [kernel]
0xffffffff81460bd3 : net_rx_action+0x103/0x2f0 [kernel]
```

Networking Tools

To properly diagnose a network performance problem, the following tools can be used:

netstat

A command-line utility which can print information about open network connections and protocol stack statistics. It retrieves information about the networking subsystem from the `/proc/net/` file system. These files include:

- **`/proc/net/dev`** (device information)
- **`/proc/net/tcp`** (TCP socket information)
- **`/proc/net/unix`** (Unix domain socket information)

For more information about **`netstat`** and its referenced files from **`/proc/net/`**, refer to the `netstat` man page: **`man netstat`**.

dropwatch

A monitoring utility which monitors packets freed from memory by the kernel. For more information, refer to the `dropwatch` man page: **`man dropwatch`**.

ip

A utility for managing and monitoring routes, devices, policy routing, and tunnels. For more information, refer to the `ip` man page: **`man ip`**.

ethtool

A utility for displaying and changing NIC settings. For more information, refer to the `ethtool` man page: **`man ethtool`**.

/proc/net/snmp

A file which displays ASCII data needed for the IP, ICMP, TCP, and UDP management information bases for an `snmp` agent. It also displays real-time UDP-lite statistics.

For further details see:

https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Performance_Tuning_Guide/s-network-dont-adjust-defaults.html

The **`ifconfig`** command uses older-style IOCTLs to retrieve information from the kernel. This method is outdated compared to the **`ip`** command which uses the kernel's Netlink interface. Use of the **`ifconfig`** command to investigate network traffic statistics is imprecise, as the statistics are not guaranteed to be updated consistently by network drivers. We recommend using the **`ip`** command instead of the **`ifconfig`** command.

Persisting Tuning Parameters Across Reboots

Many network tuning settings are kernel tunables controlled by the **sysctl** program.

The **sysctl** program can be used to both read and change the runtime configuration of a given parameter.

For example, to read the TCP Selective Acknowledgments tunable, the following command can be used:

```
# sysctl net.ipv4.tcp_sack
net.ipv4.tcp_sack = 1
```

To change the runtime value of the tunable, **`sysctl`** can also be used:

```
# sysctl -w net.ipv4.tcp_sack=0
net.ipv4.tcp_sack = 0
```

However, this setting has only been changed in the current runtime, and will change back to the kernel's built-in default if the system is rebooted.

Settings are persisted in the **`/etc/sysctl.conf`** file, and in separate **`.conf`** files in the **`/etc/sysctl.d/`** directory in later Red Hat Enterprise Linux releases.

These files can be edited directly with a text editor, or lines can be added to the files as follows:

```
# echo 'net.ipv4.tcp_sack = 0' >> /etc/sysctl.conf
```

The values specified in the configuration files are applied at boot, and can be re-applied any time afterwards with the **`sysctl -p`** command.

This document will show the runtime configuration changes for kernel tunables. Persisting desirable changes across reboots is an exercise for the reader, accomplished by following the above example.

Identifying the bottleneck

Packet drops and overruns typically occur when the RX buffer on the NIC card cannot be drained fast enough by the kernel. When the rate at which data is coming off the network exceeds that rate at which the kernel is draining packets, the NIC then discards incoming packets once the NIC buffer is full and increments a discard counter. The corresponding counter can be seen in **`ethtool`** statistics. The main criteria here are interrupts and SoftIRQs, which respond to hardware interrupts and receive traffic, then poll the card for traffic for the duration specified by **`net.core.netdev_budget`**.

The correct method to observe packet loss at a hardware level is **`ethtool`**.

The exact counter varies from driver to driver; please consult the driver vendor or driver documentation for the appropriate statistic. As a general rule look for counters with names like fail, miss, error, discard, buf, fifo, full or drop. Statistics may be upper or lower case.

For example, this driver increments various **rx_*_errors** statistics:

```
# ethtool -S eth3
     rx_errors: 0
     tx_errors: 0
     rx_dropped: 0
     tx_dropped: 0
     rx_length_errors: 0
     rx_over_errors: 3295
     rx_crc_errors: 0
     rx_frame_errors: 0
     rx_fifo_errors: 3295
     rx_missed_errors: 3295
```

There are various tools available to isolate a problem area. Locate the bottleneck by investigating the following points:

- The adapter firmware level
  - Observe drops in **ethtool -S ethX** statistics
- The adapter driver level
- The Linux kernel, IRQs or SoftIRQs
  - Check **/proc/interrupts** and **/proc/net/softnet_stat**
- The protocol layers IP, TCP, or UDP
  - Use **netstat -s** and look for error counters.

Here are some common examples of bottlenecks:

- IRQs are not getting balanced correctly. In some cases the **irqbalance** service may not be working correctly or running at all. Check **/proc/interrupts** and make sure that interrupts are spread across multiple CPU cores. Refer to the **irqbalance** manual, or manually balance the IRQs. In the following example, interrupts are getting processed by only one processor:

```
# egrep "CPU0|eth2" /proc/interrupts
           CPU0  CPU1  CPU2  CPU3  CPU4    CPU5
 105: 1430000     0     0     0     0       0  IR-PCI-MSI-edge   eth2-rx-0
 106: 1200000     0     0     0     0       0  IR-PCI-MSI-edge   eth2-rx-1
 107: 1399999     0     0     0     0       0  IR-PCI-MSI-edge   eth2-rx-2
 108: 1350000     0     0     0     0       0  IR-PCI-MSI-edge   eth2-rx-3
 109:   80000     0     0     0     0       0  IR-PCI-MSI-edge   eth2-tx
```

- See if any of the columns besides the 1st column of **/proc/net/softnet_stat** are increasing. In the following example, the counter is large for CPU0 and budget needs to be increased:

```
# cat /proc/net/softnet_stat
0073d76b 00000000 000049ae 00000000 00000000 00000000 00000000 00000000 00000000 00000000
000000d2 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0000015c 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

- SoftIRQs may not be getting enough CPU time to poll the adapter as per Figure 1. Use tools like **sar**, **mpstat**, or **top** to determine what is consuming CPU runtime.
- Use **ethtool -S ethX** to check a specific adapter for errors:

```
# ethtool -S eth3
     rx_over_errors: 399
     rx_fifo_errors: 399
     rx_missed_errors: 399
```

- Data is making it up to the socket buffer queue but not getting drained fast enough. Monitor the **ss -nmp** command and look for full RX queues. Use the **netstat -s** command and look for buffer pruning errors or UDP errors. The following example shows UDP receive errors:

```
# netstat -su
Udp:
    4218 packets received
    111999 packet receive errors
    333 packets sent
```

- Increase the application's socket receive buffer or use buffer auto-tuning by not specifying a socket buffer size in the application. Check whether the application calls **setsockopt(SO_RCVBUF)** as that will override the default socket buffer settings.
- Application design is an important factor. Look at streamlining the application to make it more efficient at reading data off the socket. One possible solution is to have separate processes draining the socket queues using Inter-Process Communication (IPC) to another process that does the background work like disk I/O.
- Use multiple TCP streams. More streams are often more efficient at transferring data.

  Use **netstat -neopa** to check how many connections an application is using:

```
tcp        0 0 0.0.0.0:12345 0.0.0.0:*     LISTEN      0 305800 27840/./server off (0.00/0/0)
tcp 16342858 0 1.0.0.8:12345 1.0.0.6:57786 ESTABLISHED 0 305821 27840/./server off (0.00/0/0)
```

- Use larger TCP or UDP packet sizes. Each individual network packet has a certain amount of overhead, such as headers. Sending data in larger contiguous blocks will reduce that overhead.
  This is done by specifying a larger buffer size with the **send()** and **recv()** function calls; please see the man page of these functions for details.
- In some cases, there may be a change in driver behavior after upgrading to a new kernel version of Red Hat Enterprise Linux. If adapter drops occur after an upgrade, open a support case with Red Hat Global Support Services to determine whether tuning is required, or whether this is a driver bug.

Performance Tuning

SoftIRQ Misses

If the SoftIRQs do not run for long enough, the rate of incoming data could exceed the kernel's capability to drain the buffer fast enough. As a result, the NIC buffers will overflow and traffic will be lost. Occasionally, it is necessary to increase the time that SoftIRQs are allowed to run on the CPU. This is known as the **`netdev_budget`**. The default value of the budget is 300. This will cause the SoftIRQ process to drain 300 messages from the NIC before getting off the CPU:

```
# sysctl net.core.netdev_budget
net.core.netdev_budget = 300
```

This value can be doubled if the 3rd column in **`/proc/net/softnet_stat`** is increasing, which indicates that the SoftIRQ did not get enough CPU time. Small increments are normal and do not require tuning.

This level of tuning is seldom required on a system with only gigabit interfaces. However, a system passing upwards of 10Gbps may need this tunable increased.

```
# cat softnet_stat
0073d76b 00000000 000049ae 00000000 00000000 00000000 00000000 00000000 00000000 00000000
000000d2 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0000015c 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0000002a 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

For example, tuning the value on this NIC from 300 to 600 will allow soft interrupts to run for double the default CPU time:

```
# sysctl -w net.core.netdev_budget=600
```

Tuned

Tuned is an adaptive system tuning daemon. It can be used to apply a variety of system settings gathered together into a collection called a profile.

A tuned profile can contain instructions such as CPU governor, IO scheduler, and kernel tunables such as CPU scheduling or virtual memory management. Tuned also incorporates a monitoring daemon which can control or disable power saving ability of CPUs, disks, and network devices.

The aim of performance tuning is to apply settings which enable the most desirable performance. Tuned can automate a large part of this work.

First, install tuned, start the tuning daemon service, and enable the service on boot:

```
# yum -y install tuned
# service tuned start
# chkconfig tuned on
```

List the performance profiles:

```
# tuned-adm list
Available profiles:
- throughput-performance
- default
- desktop-powersave
- enterprise-storage
...
```

The contents of each profile can be viewed in the **/etc/tune-profiles/** directory. We are concerned about setting a performance profile such as **throughput-performance**, **latency-performance**, or **enterprise-storage**.

Set a profile:

```
# tuned-adm profile throughput-performance
Switching to profile 'throughput-performance'
...
```

The selected profile will apply every time the tuned service starts. The tuned service is described further in **man tuned**.

Numad

Similar to tuned, numad is a daemon which can assist with process and memory management on systems with Non-Uniform Memory Access (NUMA) architecture. Numad achieves this by monitoring system topology and resource usage, then attempting to locate processes for efficient NUMA locality and efficiency, where a process has a sufficiently large memory size and CPU load.

The numad service also requires cgroups (Linux kernel control groups) to be enabled.

```
# service cgconfig start
Starting cgconfig service:                                 [  OK  ]

# service numad start
Starting numad:                                            [  OK  ]
```

By default, as of Red Hat Enterprise Linux 6.5, numad will manage any process with over 300Mb of memory usage and 50% of one core CPU usage, and try to use any given NUMA node up to 85% capacity.

Numad can be more finely tuned with the directives described in **man numad**. Please refer to the **Understanding NUMA architecture** section later in this document to see if your system is a NUMA system or not.

CPU Power States

The ACPI specification defines various levels of processor power states or “C-states”, with **C0**

being the operating state, **C1** being the halt state, plus processor manufacturers implementing various additional states to provide additional power savings and related advantages such as lower temperatures.

Unfortunately, transitioning between power states is costly in terms of latency. As we are concerned with making the responsiveness of the system as high as possible, it is desirable to disable all processor “deep sleep” states, leaving only operating and halt.

This must be accomplished first in the system BIOS or EFI firmware. Any states such as **C6**, **C3**, **C1E** or similar should be disabled.

We can ensure the kernel never requests a C-state below C1 by adding **processor.max_cstate=1** to the kernel line in the GRUB bootloader configuration.

In some instances, the kernel is able to override the hardware setting and the additional parameter **intel_idle.max_cstate=0** must be added to systems with Intel processors.

The sleep state of the processor can be confirmed with:

```
# cat /sys/module/intel_idle/parameters/max_cstate
0
```

A higher value indicates that additional sleep states may be entered.

The **powertop** utility's **Idle Stats** page can show how much time is being spent in each C-state.

IRQ Balance

IRQ Balance is a service which can automatically balance interrupts across CPU cores, based on real time system conditions. It is vital that the correct version of **irqbalance** is running for a particular kernel. For NUMA systems, **irqbalance-1.0.4-8.el6_5** or greater is required for Red Hat Enterprise Linux 6.5 and **irqbalance-1.0.4-6.el6_4** or greater is required for Red Hat Enterprise Linux 6.4. See the **Understanding NUMA architecture** section later in this document for manually balancing **irqbalance** for NUMA systems.

```
# rpm -q irqbalance
irqbalance-0.55-29.el6.x86_64
```

Manual balancing of interrupts

The IRQ affinity can also be manually balanced if desired. Red Hat strongly recommends using **irqbalance** to balance interrupts, as it dynamically balances interrupts depending on system usage and other factors. However, manually balancing interrupts can be used to determine if **irqbalance** is not balancing IRQs in a optimum manner and therefore causing packet loss. There may be some very specific cases where manually balancing interrupts permanently can be beneficial. For this case, the interrupts will be manually associated with a CPU using SMP affinity.

There are 2 ways to do this; with a bitmask or using **smp_affinity_list** which is available from Red Hat Enterprise Linux 6 onwards.

To manually balance interrupts, the **irqbalance** service needs to be stopped and persistently disabled:

```
# chkconfig irqbalance off
# service irqbalance stop
Stopping irqbalance:                                       [  OK  ]
```

View the CPU cores where a device's interrupt is allowed to be received:

```
# grep “CPU0|eth3” /proc/interrupts
        CPU0   CPU1   CPU2   CPU3   CPU4   CPU5
 110:   1136      0      0      0      0      0  IR-PCI-MSI-edge      eth3-rx-0
 111:      2      0      0      0      0      0  IR-PCI-MSI-edge      eth3-rx-1
 112:      0      0      0      0      0      0  IR-PCI-MSI-edge      eth3-rx-2
 113:      0      0      0      0      0      0  IR-PCI-MSI-edge      eth3-rx-3
 114:      0      0      0      0      0      0  IR-PCI-MSI-edge      eth3-tx

# cat /proc/irq/110/smp_affinity_list
0-5
```

One way to manually balance the CPU cores is with a script. The following script is a simple proof-of-concept example:

```
#!/bin/bash
# nic_balance.sh
# usage nic_balance.sh <number of cpus>
cpu=0
grep $1 /proc/interrupts|awk '{print $1}'|sed 's/://'|while read a
do
echo $cpu > /proc/irq/$a/smp_affinity_list
echo "echo $cpu > /proc/irq/$a/smp_affinity_list"
        if [ $cpu = $2 ]
        then
                cpu=0
        fi
        let cpu=cpu+1
done
```

The above script reports the commands it ran as follows:

```
# sh balance.sh eth3 5
echo 0 > /proc/irq/110/smp_affinity_list
echo 1 > /proc/irq/111/smp_affinity_list
echo 2 > /proc/irq/112/smp_affinity_list
echo 3 > /proc/irq/113/smp_affinity_list
echo 4 > /proc/irq/114/smp_affinity_list
echo 5 > /proc/irq/131/smp_affinity_list
```

The above script is provided under a Creative Commons Zero license.

Ethernet Flow Control (a.k.a. Pause Frames)

Pause frames are Ethernet-level flow control between the adapter and the switch port. The adapter will send “pause frames” when the RX or TX buffers become full. The switch will stop data flowing for a time span in the order of milliseconds or less. This is usually enough time to allow the kernel to drain the interface buffers, thus preventing the buffer overflow and subsequent packet drops or overruns. Ideally, the switch will buffer the incoming data during the pause time. However it is important to realize that this level of flow control is only between the switch and the adapter. If packets are dropped, the higher layers such as TCP, or the application in the case of UDP and/or multicast, should initiate recovery.

Pause frames and Flow Control need to be enabled on both the NIC and switch port for this feature to take effect. Please refer to your network equipment manual or vendor for instruction on how to enable Flow Control on a port.

In this example, Flow Control is disabled:

```
# ethtool -a eth3
Pause parameters for eth3:
Autonegotiate: off
RX:            off
TX:            off
```

To enable Flow Control:

```
# ethtool -A eth3 rx on
# ethtool -A eth3 tx on
```

To confirm Flow Control is enabled:

```
# ethtool -a eth3
Pause parameters for eth3:
Autonegotiate: off
RX:            on
TX:            on
```

Interrupt Coalescence (IC)

Interrupt coalescence refers to the amount of traffic that a network interface will receive, or time that passes after receiving traffic, before issuing a hard interrupt. Interrupting too soon or too frequently results in poor system performance, as the kernel stops (or “interrupts”) a running task to handle the interrupt request from the hardware. Interrupting too late may result in traffic not being taken off the NIC soon enough. More traffic may arrive, overwriting the previous traffic still waiting to be received into the kernel, resulting in traffic loss.

Most modern NICs and drivers support IC, and many allow the driver to automatically moderate the number of interrupts generated by the hardware. The IC settings usually comprise of 2 main components, time and number of packets. Time being the number microseconds (u-secs) that the NIC will wait before interrupting the kernel, and the number being the maximum number of

packets allowed to be waiting in the receive buffer before interrupting the kernel.

A NIC's interrupt coalescence can be viewed using **`ethtool -c ethX`** command, and tuned using the **`ethtool -C ethX`** command. Adaptive mode enables the card to auto-moderate the IC. In adaptive mode, the driver will inspect traffic patterns and kernel receive patterns, and estimate coalescing settings on-the-fly which aim to prevent packet loss. This is useful when many small packets are being received. Higher interrupt coalescence favors bandwidth over latency. A VOIP application (latency-sensitive) may require less coalescence than a file transfer protocol (throughput-sensitive). Different brands and models of network interface cards have different capabilities and default settings, so please refer to the manufacturer's documentation for the adapter and driver.

On this system adaptive RX is enabled by default:

```
# ethtool -c eth3
Coalesce parameters for eth3:
Adaptive RX: on  TX: off
stats-block-usecs: 0
sample-interval: 0
pkt-rate-low: 400000
pkt-rate-high: 450000

rx-usecs: 16
rx-frames: 44
rx-usecs-irq: 0
rx-frames-irq: 0
```

The following command turns adaptive IC off, and tells the adapter to interrupt the kernel immediately upon reception of any traffic:

```
# ethtool -C eth3 adaptive-rx off rx-usecs 0 rx-frames 0
```

A realistic setting is to allow at least some packets to buffer in the NIC, and at least some time to pass, before interrupting the kernel. Valid ranges may be from 1 to hundreds, depending on system capabilities and traffic received.

The Adapter Queue

The **`netdev_max_backlog`** is a queue within the Linux kernel where traffic is stored after reception from the NIC, but before processing by the protocol stacks (IP, TCP, etc). There is one backlog queue per CPU core. A given core's queue can grow automatically, containing a number of packets up to the maximum specified by the **`netdev_max_backlog`** setting. The **`netif_receive_skb()`** kernel function will find the corresponding CPU for a packet, and enqueue packets in that CPU's queue. If the queue for that processor is full and already at maximum size, packets will be dropped.

To tune this setting, first determine whether the backlog needs increasing.

The **`/proc/net/softnet_stat`** file contains a counter in the 2rd column that is incremented when the netdev backlog queue overflows. If this value is incrementing over time, then **`netdev_max_backlog`** needs to be increased.

Each line of the **softnet_stat** file represents a CPU core starting from CPU0:

```
Line 1 = CPU0
Line 2 = CPU1
Line 3 = CPU2
```

and so on. The following system has 12 CPU cores:

```
# wc -l /proc/net/softnet_stat
12
```

When a packet is unable to be placed into a backlog queue, the following code is executed where **get_cpu_var** identifies the appropriate processor queue:

```
      __get_cpu_var(netdev_rx_stat).dropped++;
```

The above code then increments the **dropped** statistic for that queue.

Each line in the **softnet_stat** file represents the **netif_rx_stats** structure for that CPU. That data structure contains:

```
struct netif_rx_stats
{
        unsigned total;
        unsigned dropped;
        unsigned time_squeeze;
        unsigned cpu_collision;
        unsigned received_rps;
};
```

The 1st column is the number of frames received by the interrupt handler.

The 2nd column is the number of frames dropped due to **netdev_max_backlog** being exceeded.

The 3rd column is the number of times ksoftirqd ran out of **netdev_budget** or CPU time when there was still work to be done.

The other columns may vary depending on the version Red Hat Enterprise Linux. Using the following example, the following counters for CPU0 and CPU1 are the first two lines:

```
# cat softnet_stat
0073d76b 00000000 000049ae 00000000 00000000 00000000 00000000 00000000 00000000 00000000
000000d2 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0000015c 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0000002a 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
...
```

For the above example, **netdev_max_backlog** does not need to be changed as the number of drops has remained at 0:

```
For CPU0
Total   dropped no_budget       lock_contention
0073d76b        00000000        000049ae        00000000

For CPU1
Total   dropped no_budget       lock_contention
000000d2        00000000        00000000        00000000
```

The statistics in each column are provided in hexadecimal.

The default **netdev_max_backlog** value is 1000. However, this may not be enough for multiple interfaces operating at 1Gbps, or even a single interface at 10Gbps. Try doubling this value and observing the **/proc/net/softnet_stat** file. If doubling the value reduces the rate at which drops increment, double again and test again. Repeat this process until the optimum size is established and drops do not increment.

The backlog can be changed with with following command, where X is the desired value to be set:

```
# sysctl -w net.core.netdev_max_backlog=X
```

Adapter RX and TX Buffer Tuning

Adapter buffer defaults are commonly set to a smaller size than the maximum. Often, increasing the receive buffer size is alone enough to prevent packet drops, as it can allow the kernel slightly more time to drain the buffer. As a result, this can prevent possible packet loss.

The following interface has the space for 8 kilobytes of buffer but is only using 1 kilobyte:

```
# ethtool -g eth3
Ring parameters for eth3:
Pre-set maximums:
RX:             8192
RX Mini:        0
RX Jumbo:       0
TX:             8192
Current hardware settings:
RX:             1024
RX Mini:        0
RX Jumbo:       0
TX:             512
```

Increase both the RX and TX buffers to the maximum:

```
# ethtool -G eth3 rx 8192 tx 8192
```

This change can be made whilst the interface is online, though a pause in traffic will be seen.

These settings can be persisted by writing a script at **/sbin/ifup-local**. This is documented on the knowledgebase at:

> **How do I run a script or program immediately after my network interface goes up?**
>
> https://access.redhat.com/knowledge/solutions/8694

Adapter Transmit Queue Length

The transmit queue length value determines the number of packets that can be queued before being transmitted. The default value of 1000 is usually adequate for today's high speed 10Gbps or even 40Gbps networks. However, if the number transmit errors are increasing on the adapter, consider doubling it. Use **`ip -s link`** to see if there are any drops on the TX queue for an adapter.

```
# ip link
2: em1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master br0 state UP
mode DEFAULT group default qlen 1000
    link/ether f4:ab:cd:1e:4c:c7 brd ff:ff:ff:ff:ff:ff
    RX: bytes  packets  errors  dropped overrun mcast
    71017768832 60619524 0       0       0       1098117
    TX: bytes  packets  errors  dropped carrier collsns
    10373833340 36960190 0       0       0       0
```

The queue length can be modified with the **`ip link`** command:

```
# ip link set dev em1 txqueuelen 2000
# ip link
2: em1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master br0 state UP
mode DEFAULT group default qlen 2000
    link/ether f4:ab:cd:1e:4c:c7 brd ff:ff:ff:ff:ff:ff
```

To persist this value across reboots, a udev rule can be written to apply the queue length to the interface as it is created, or the network scripts can be extended with a script at **`/sbin/ifup-local`** as described on the knowledgebase at:

> **How do I run a script or program immediately after my network interface goes up?**
>
> https://access.redhat.com/knowledge/solutions/8694

Module parameters

Each network interface driver usually comes as loadable kernel module. Modules can be loaded and unloaded using the **`modprobe`** command. These modules usually contain parameters that can be used to further tune the device driver and NIC. The **`modinfo <drivername>`** command can be used to view these parameters. Documenting specific driver parameters is beyond the scope of this document. Please refer to the hardware manual, driver documentation, or hardware vendor for an explanation of these parameters.

The Linux kernel exports the current settings for module parameters via the sysfs path **`/sys/module/<drivername>/parameters`**

For example, given the driver parameters:

```
# modinfo mlx4_en
filename:       /lib/modules/2.6.32-246.el6.x86_64/kernel/drivers/net/mlx4/mlx4_en.ko
version:        2.0 (Dec 2011)
license:        Dual BSD/GPL
```

```
description:    Mellanox ConnectX HCA Ethernet driver
author:         Liran Liss, Yevgeny Petrilin
depends:        mlx4_core
vermagic:       2.6.32-246.el6.x86_64 SMP mod_unload modversions
parm:  inline_thold:treshold for using inline data (int)
parm:  tcp_rss:Enable RSS for incomming TCP traffic or disabled (0) (uint)
parm:  udp_rss:Enable RSS for incomming UDP traffic or disabled (0) (uint)
parm:  pfctx:Priority based Flow Control policy on TX[7:0]. Per priority bit mask (uint)
parm:  pfcrx:Priority based Flow Control policy on RX[7:0]. Per priority bit mask (uint)
```

The current values of each driver parameter can be checked in sysfs.
For example, to check the current setting for the **udp_rss** parameter:

```
# ls /sys/module/mlx4_en/parameters
inline_thold  num_lro  pfcrx  pfctx  rss_mask  rss_xor  tcp_rss  udp_rss

# cat /sys/module/mlx4_en/parameters/udp_rss
1
```

Some drivers allow these values to be modified whilst loaded, but many values require the driver module to be unloaded and reloaded to apply a module option.

Loading and unloading of a driver module is done with the **modprobe** command:

```
# modprobe -r <drivername>
# modprobe <drivername>
```

For non-persistent use, a module parameter can also be enabled as the driver is loaded:

```
# modprobe -r <drivername>
# modprobe <drivername> <parm>=<value>
```

In the event a module cannot be unloaded, a reboot will be required.
For example, to use RPS instead of RSS, disable RSS as follows:

```
# echo 'options mlx4_en udp_rss=0' >> /etc/modprobe.d/mlx4_en.conf
```

Unload and reload the driver:

```
# modprobe -r mlx4_en
# modprobe mlx4_en
```

This parameter could also be loaded just this time:

```
# modprobe -r mlx4_en
# modprobe mlx4_en udp_rss=0
```

Confirm whether that parameter change took effect:

```
# cat /sys/module/mlx4_en/parameters/udp_rss
0
```

In some cases, driver parameters can also be controlled via the **ethtool** command.

For example, the Intel Sourceforge **igb** driver has the interrupt moderation parameter **InterruptThrottleRate**. The upstream Linux kernel driver and the Red Hat Enterprise Linux driver do not expose this parameter via a module option. Instead, the same functionality can instead be tuned via **ethtool**:

```
# ethtool -C ethX rx-usecs 1000
```

Adapter Offloading

In order to reduce CPU load from the system, modern network adapters have offloading features which move some network processing load onto the network interface card. For example, the kernel can submit large (up to 64k) TCP segments to the NIC, which the NIC will then break down into MTU-sized segments. This particular feature is called TCP Segmentation Offload (TSO).

Offloading features are often enabled by default. It is beyond the scope of this document to cover every offloading feature in-depth, however, turning these features off is a good troubleshooting step when a system is suffering from poor network performance and re-test. If there is an performance improvement, ideally narrow the change to a specific offloading parameter, then report this to Red Hat Global Support Services. It is desirable to have offloading enabled wherever possible.

Offloading settings are managed by **ethtool -K eth*X***. Common settings include:

- GRO: Generic Receive Offload
- LRO: Large Receive Offload
- TSO: TCP Segmentation Offload
- RX check-summing = Processing of receive data integrity
- TX check-summing = Processing of transmit data integrity (required for TSO)

```
# ethtool -k eth0
Features for eth0:
rx-checksumming: on
tx-checksumming: on
scatter-gather: on
tcp-segmentation-offload: on
udp-fragmentation-offload: off
generic-segmentation-offload: on
generic-receive-offload: on
large-receive-offload: on
rx-vlan-offload: on
tx-vlan-offload: on
ntuple-filters: off
receive-hashing: on
```

Jumbo Frames

The default 802.3 Ethernet frame size is 1518 bytes, or 1522 bytes with a VLAN tag. The Ethernet header consumes 18 bytes of this (or 22 bytes with VLAN tag), leaving an effective maximum payload of 1500 bytes. Jumbo Frames are an unofficial extension to Ethernet which network equipment vendors have made a de-facto standard, increasing the payload from 1500 to 9000 bytes.

With regular Ethernet frames there is an overhead of 18 bytes for every 1500 bytes of data placed on the wire, or 1.2% overhead.

With Jumbo Frames there is an overhead of 18 bytes for every 9000 bytes of data placed on the wire, or 0.2% overhead.

The above calculations assume no VLAN tag, however such a tag will add 4 bytes to the overhead, making efficiency gains even more desirable.

When transferring large amounts of contiguous data, such as sending large files between two systems, the above efficiency can be gained by using Jumbo Frames. When transferring small amounts of data, such as web requests which are typically below 1500 bytes, there is likely no gain to be seen from using a larger frame size, as data passing over the network will be contained within small frames anyway.

For Jumbo Frames to be configured, all interfaces and network equipment in a network segment (i.e. broadcast domain) must support Jumbo Frames and have the increased frame size enabled. Refer to your network switch vendor for instructions on increasing the frame size.

On Red Hat Enterprise Linux, increase the frame size with **`MTU=9000`** in the **`/etc/sysconfig/network-scripts/ifcfg-`** file for the interface.

The MTU can be checked with the **`ip link`** command:

```
# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9000 qdisc pfifo_fast state UP qlen 1000
    link/ether 52:54:00:36:b2:d1 brd ff:ff:ff:ff:ff:ff
```

TCP Timestamps

TCP Timestamps are an extension to the TCP protocol, defined in **RFC 1323 - TCP Extensions for High Performance** - http://tools.ietf.org/html/rfc1323

TCP Timestamps provide a monotonically increasing counter (on Linux, the counter is milliseconds since system boot) which can be used to better estimate the round-trip-time of a TCP conversation, resulting in more accurate TCP Window and buffer calculations.

Most importantly, TCP Timestamps also provide **Protection Against Wrapped Sequence Numbers** as the TCP header defines a Sequence Number as a 32-bit field. Given a sufficiently fast link, this TCP Sequence Number number can wrap. This results in the receiver believing that the segment with the wrapped number actually arrived *earlier* than its preceding segment, and incorrectly discarding the segment.

On a 1 gigabit per second link, TCP Sequence Numbers can wrap in 17 seconds. On a 10 gigabit per second link, this is reduced to as little as 1.7 seconds. On fast links, enabling TCP Timestamps should be considered mandatory.

TCP Timestamps provide an alternative, non-wrapping, method to determine the age and order of a segment, preventing wrapped TCP Sequence Numbers from being a problem.

Ensure TCP Timestamps are enabled:

```
# sysctl net.ipv4.tcp_timestamps
net.ipv4.tcp_timestamps = 1
```

If the above command indicates that **`tcp_timestamps = 0`**, enable TCP Timestamps:

```
# sysctl -w net.ipv4.tcp_timestamps=1
```

TCP SACK

TCP Selective Acknowledgments (SACK) is TCP extension defined in **RFC 2018 - TCP Selective Acknowledgment Options** - http://tools.ietf.org/html/rfc2018

A basic TCP Acknowledgment (ACK) only allows the receiver to advise the sender which bytes have been received. When packet loss occurs, this requires the sender to retransmit all bytes from the point of loss, which can be inefficient. SACK allows a sender to specify which bytes have been lost and which bytes have been received, so the sender can retransmit only the lost bytes.

There is some research available in the networking community which shows enabling SACK on high-bandwidth links can cause unnecessary CPU cycles to be spent calculating SACK values, reducing overall efficiency of TCP connections. This research implies these links are so fast, the overhead of retransmitting small amounts of data is less than the overhead of calculating the data to provide as part of a Selective Acknowledgment.

Unless there is high latency or high packet loss, it is most likely better to keep SACK turned off over a high performance network.

SACK can be turned off with kernel tunables:

```
# sysctl -w net.ipv4.tcp_sack=0
```

TCP Window Scaling

TCP Window Scaling is an extension to the TCP protocol, defined in **RFC 1323 - TCP Extensions for High Performance** - http://tools.ietf.org/html/rfc1323

In the original TCP definition, the TCP segment header only contains an 8-bit value for the TCP Window Size, which is insufficient for the link speeds and memory capabilities of modern computing. The TCP Window Scaling extension was introduced to allow a larger TCP Receive Window. This is achieved by adding a scaling value to the TCP options which are added after the TCP header. The real TCP Receive Window is bit-shifted left by the value of the Scaling Factor value, up to a maximum size of 1,073,725,440 bytes, or close to one gigabyte.

TCP Window Scaling is negotiated during the three-way TCP handshake (SYN, SYN+ACK, ACK) which opens every TCP conversation. Both sender and receiver must support TCP Window Scaling for the Window Scaling option to work. If either or both participants do not advertise Window Scaling ability in their handshake, the conversation falls back to using the original 8-bit TCP Window Size.

TCP Window Scaling is enabled by default on Red Hat Enterprise Linux. The status of Window Scaling can be confirmed with the command:

```
# sysctl net.ipv4.tcp_window_scaling
net.ipv4.tcp_window_scaling = 1
```

TCP Window Scaling negotiation can be viewed by taking a packet capture of the TCP handshake which opens a conversation. In the packet capture, check the TCP Options field of the three handshake packets. If either system's handshake packets do not contain the TCP Window Scaling option, it may be necessary to enable TCP Window Scaling on that system.

TCP Buffer Tuning

Once network traffic is processed from the network adapter, reception directly into the application is attempted. If that is not possible, data is queued on the application's socket buffer. There are 3 queue structures in the socket:

```
        sk_rmem_alloc = {
          counter = 121948
        },
        sk_wmem_alloc = {
          counter = 553
        },
        sk_omem_alloc = {
          counter = 0
```

sk_rmem_alloc is the receive queue

k_wmem_alloc is the transmit queue

sk_omem_alloc is the out-of-order queue, skbs which are not within the current TCP Window are placed in this queue

There is also the **sk_rcvbuf** variable which is the limit, measured in bytes, that the socket can receive. In this case:

```
  sk_rcvbuf = 125336
```

From the above output it can be calculated that the receive queue is almost full. When **sk_rmem_alloc > sk_rcvbuf** the TCP stack will call a routine which “collapses” the receive queue. This is a kind of house-keeping where the kernel will try the free space in the receive queue by reducing overhead. However, this operation comes at a CPU cost. If collapsing fails to free sufficient space for additional traffic, then data is “pruned”, meaning the data is dropped from memory and the packet is lost. Therefore, it best to tune around this condition and avoid the buffer collapsing and pruning altogether. The first step is to identify whether buffer collapsing and pruning is occurring.

Run the following command to determine whether this is occurring or not:

```
# netstat -sn | egrep “prune|collap”; sleep 30; netstat -sn | egrep “prune|collap”
    17671 packets pruned from receive queue because of socket buffer overrun
    18671 packets pruned from receive queue because of socket buffer overrun
```

If “pruning” has increased during this interval, then tuning is required. The first step is to increase the network and TCP receive buffer settings. This is a good time to check whether the application calls **setsockopt(SO_RCVBUF)**. If the application does call this function, this will override the default settings and turn off the socket's ability to auto-tune its size. The size of the receive buffer will be the size specified by the application and no greater. Consider removing the **setsockopt(SO_RCVBUF)** function call from the application and allowing the buffer size to auto-tune instead.

Tuning tcp_rmem

The socket memory tunable has three values, describing the minimum, default, and maximum values in bytes.

The default maximum on most Red Hat Enterprise Linux releases is 4MiB. To view these settings, then increase them by a factor of 4:

```
# sysctl net.ipv4.tcp_rmem
4096   87380   4194304
# sysctl -w net.ipv4.tcp_rmem=“16384 349520 16777216”
# sysctl net.core.rmem_max
4194304
# sysctl -w net.core.rmem_max=16777216
```

If the application cannot be changed to *remove* **setsockopt(SO_RCVBUF)**, then *increase* the maximum socket receive buffer size which may be set by using the **SO_RCVBUF** socket option.

A restart of an application is only required when the middle value of **tcp_rmem** is changed, as the **sk_rcvbuf** value in the socket is initialized to this when the socket is created. Changing the 3rd and maximum value of **tcp_rmem** does not require an application restart as these values are dynamically assigned by auto-tuning.

TCP Listen Backlog

When a TCP socket is opened by a server in **LISTEN** state, that socket has a maximum number of unaccepted client connections it can handle.

If an application is slow at processing client connections, or the server gets many new connections rapidly (commonly known as a SYN flood), the new connections may be lost or specially crafted reply packets known as “SYN cookies” may be sent.

If the system's normal workload is such that SYN cookies are being entered into the system log regularly, the system and application should be tuned to avoid them.

The maximum backlog an application can request is dictated by the **net.core.somaxconn** kernel tunable. An application can always request a larger backlog, but it will only get a backlog as large as this maximum.

This parameter can be checked and changed as follows:

```
# sysctl net.core.somaxconn
net.core.somaxconn = 128

# sysctl -w net.core.somaxconn=2048
net.core.somaxconn = 2048

# sysctl net.core.somaxconn
net.core.somaxconn = 2048
```

After changing the maximum allowed backlog, an application must be restarted for the change to take effect.

Additionally, after changing the maximum allowed backlog, the application must be modified to actually set a larger backlog on its listening socket.

The following is an example in the C language of the change required to increase the socket backlog:

```
-   rc = listen(sockfd, 128);
+   rc = listen(sockfd, 2048);
    if (rc < 0)
    {
        perror("listen() failed");
        close(sockfd);
        exit(-1);
    }
```

The above change would require the application to be recompiled from source. If the application is designed so the backlog is a configurable parameter, this could be changed in the application's settings and recompilation would not be required.

Advanced Window Scaling

You may see the “pruning” errors continue to increase regardless of the above settings.

In Red Hat Enterprise Linux 6.3 and 6.4 there was a commit added to charge the cost of the skb shared structures to the socket as well, described in the kernel changelog as **[net] more accurate skb truesize**. This change had the effect of filling the TCP receive queue even faster, therefore hitting pruning conditions quicker. This change was reverted in Red Hat Enterprise Linux 6.5.

If the receive buffers are increased and pruning still occurs, the parameter **net.ipv4.tcp_adv_win_scale** decides the ratio of receive buffer which is allocated to data vs the buffer which is advertised as the available TCP Window. The default on Red Hat Enterprise Linux 5 and 6 is **2** which equates to quarter of the buffer allocated to the application data. On Red

Hat Enterprise Linux 7 releases this defaults to **1**, resulting in half the space being advertised as the TCP Window. Setting this value to **1** on Red Hat Enterprise Linux 5 and 6 will have the effect of reducing the advertised TCP Window, possibly preventing the receive buffer from over-flowing and thereforepreventing buffer pruning.

```
# sysctl net.ipv4.tcp_adv_win_scale
2
# sysctl -w net.ipv4.tcp_adv_win_scale=1
```

UDP Buffer Tuning

UDP is a far less complex protocol than TCP. As UDP contains no session reliability, it is the application's responsibility to identify and re-transmit dropped packets. There is no concept of a window size and lost data is not recovered by the protocol. The only available tuning comprises of increasing the receive buffer size. However, if **netstat -us** is reporting errors, another underlying issue may be preventing the application from draining its receive queue. If **netstat -us** is showing “packet receive errors”, try increasing the receive buffers and re-testing. This statistic can also be incremented for other reasons, such as short packets where the payload data is less than the UDP header advises, or corrupted packets which fail their checksum calculation, so a more in-depth investigation may be required if buffer tuning does not resolve UDP receive errors.

UDP buffers can be tuned in a similar fashion to the maximum TCP buffer:

```
# sysctl net.core.rmem_max
124928
# sysctl -w net.core.rmem_max=16777216
```

After altering the maximum size, a restart of the application is required for the new setting to take effect.

Understanding NUMA Architecture

NUMA architecture splits a subset of CPU, memory, and devices into different “nodes”, in effect creating multiple small computers with a fast interconnect and common operating system. NUMA systems need to be tuned differently to non-NUMA systems. For NUMA, the aim is to group all interrupts from devices in a single node onto the CPU cores belonging to that node. The Red Hat Enterprise Linux 6.5 version of **irqbalance** is NUMA-aware, allowing interrupts to balance only to CPUs within a given NUMA node.

Determine NUMA Nodes

First, determine how many NUMA nodes a system has. This system has two NUMA nodes:

```
# ls -ld /sys/devices/system/node/node*
drwxr-xr-x. 3 root root 0 Aug 15 19:44 /sys/devices/system/node/node0
drwxr-xr-x. 3 root root 0 Aug 15 19:44 /sys/devices/system/node/node1
```

Determine NUMA Locality

Determine which CPU cores belong to which NUMA node. On this system, node0 has 6 CPU cores:

```
# cat /sys/devices/system/node/node0/cpulist
0-5
```

Node 1 has no CPUs.:

```
# cat /sys/devices/system/node/node1/cpulist
```

Given the above output, all devices must be on node0. For this system, it makes sense to tune IRQ affinity for all 6 of these CPUs, 0-5. On Red Hat Enterprise Linux 6, this can be achieved by stopping the **irqbalance** service and manually setting the CPU affinity:

```
# service irqbalance stop
# chkconfig irqbalance off
```

Vendors typically provide scripts to manually balance IRQs. Please refer to the NIC vendor's website to download these scripts. For example, Intel and Mellanox provide useful scripts as part of their driver downloads.

Determine Device Locality

Checking the whether a PCIe network interface belongs to a specific NUMA node:

```
# cat /sys/class/net/<interface>/device/numa_node
```

For example:

```
# cat /sys/class/net/eth3/device/numa_node
1
```

This command will display the NUMA node number, interrupts for the device should be directed to the NUMA node that the PCIe device belongs to.

This command may display **-1** which indicates the hardware platform is not actually non-uniform and the kernel is just emulating or “faking” NUMA, or a device is on a bus which does not have any NUMA locality, such as a PCI bridge.

Identifying Interrupts to Balance

Check the number RX and TX queues on the adapter:

```
# egrep “CPU0|eth3” /proc/interrupts
         CPU0     CPU1     CPU2    CPU3   CPU4    CPU5
 110:       0        0        0       0      0       0  IR-PCI-MSI-edge      eth3-rx-0
 111:       0        0        0       0      0       0  IR-PCI-MSI-edge      eth3-rx-1
 112:       0        0        0       0      0       0  IR-PCI-MSI-edge      eth3-rx-2
 113:       2        0        0       0      0       0  IR-PCI-MSI-edge      eth3-rx-3
 114:       0        0        0       0      0       0  IR-PCI-MSI-edge      eth3-tx
```

Queues are allocated when the NIC driver module is loaded. In some cases, the number of queues can be dynamically allocated online using the **ethtool -L** command. The above device has 4 RX queues and one TX queue.

Statistics are different for every network driver, but if a network driver provides separate queue statistics, these can be seen with the command **ethtool -S eth*X*** where eth*X* is the interface in question:

```
# ethtool -S eth3
     rx0_packets: 0
     rx0_bytes: 0
     rx1_packets: 0
     rx1_bytes: 0
     rx2_packets: 0
     rx2_bytes: 0
     rx3_packets: 2
     rx3_bytes: 120
```

GLOSSARY

RSS: Receive Side Scaling

RSS is supported by many common network interface cards. On reception of data, a NIC can send data to multiple queues. Each queue can be serviced by a different CPU, allowing for efficient data retrieval. The RSS acts as an API between the driver and the card firmware to determine how packets are distributed across CPU cores, the idea being that multiple queues directing traffic to different CPUs allows for faster throughput and lower latency. RSS controls which receive queue gets any given packet, whether or not the card listens to specific unicast Ethernet addresses, which multicast addresses it listens to, which queue pairs or Ethernet queues get copies of multicast packets, etc.

RSS Considerations

- Does the driver allow the number of queues to be configured?
  Some drivers will automatically generate the number of queues during boot depending on hardware resources. For others it's configurable via **ethtool -L**.
- How many cores does the system have?
  RSS should be configured so each queue goes to a different CPU core.

RPS: Receive Packet Steering

Receive Packet Steering is a kernel-level software implementation of RSS. It resides the higher layers of the network stack above the driver. RSS or RPS should be mutually exclusive. RPS is disabled by default. RPS uses a 2-tuple or 4-tuple hash saved in the **rxhash** field of the packet definition, which is used to determine the CPU queue which should process a given packet.

RFS: Receive Flow Steering

Receive Flow Steering takes application locality into consideration when steering packets. This avoids cache misses when traffic arrives on a different CPU core to where the application is running.

Receive Steering Reference

For more details on the above steering mechanisms, please refer to:
https://www.kernel.org/doc/Documentation/networking/scaling.txt

NAPI: New API

The software method where a device is polled for new network traffic, instead of the device constantly raising hardware interrupts.

skb, sk_buff: Socket buffer

There are data buffers which are used to transport network headers and payload data through the Linux kernel.

MTU: Maximum Transmission Unit

MTU defines the largest contiguous block of data which can be sent across a transmission medium. A block of data is transmitted as a single unit commonly referred to as a frame or packet. Each data unit which will have a header size which does not change, making it more efficient to sent as much data as possible in a given data unit. For example, an Ethernet header without a VLAN tag is 18 bytes. It is more efficient to send 1500 bytes of data plus an 18-byte header and less efficient to send 1 byte of data plus an 18-byte header.

NUMA: Non Uniform Memory Access

A hardware layout where processors, memory, and devices do not all share a common bus. Instead, some components such as CPUs and memory are more local or more distant in comparison to each other.

NIC Tuning Summary

The following is a summary of points which have been covered by this document in detail:

- SoftIRQ misses (netdev budget)
- "tuned" tuning daemon
- "numad" NUMA daemon
- CPU power states
- Interrupt balancing

- Pause frames
- Interrupt Coalescence
- Adapter queue (netdev backlog)
- Adapter RX and TX buffers
- Adapter TX queue
- Module parameters
- Adapter offloading
- Jumbo Frames
- TCP and UDP protocol tuning
- NUMA locality