Linux Multiqueue Networking

David S. Miller

Red Hat Inc.

New York City, 2009

TRENDS

- More CPUs, either less powerful (high arity) or same (low arity) as existing CPUs
- Flow counts increasing
- Networking hardware adjusting to horizontal scaling
- Single queue model no longer works
- Routers and firewalls have different needs than servers

CPU Design

- Traditionally single CPUs or very low count SMP
- The move to high-arity CPU counts
- One model: Sun's Niagara
- Lower powered CPUs, but many of them
- Other model: x86 based systems
- High powered CPUs, but not as high increase in arity as Niagara approach, starting with hyperthreading
- Future: Best of both worlds, high arity and power

End Nodes vs. Intermediate Nodes

- End Nodes: Servers
- Intermediate Nodes: Routers and Firewalls
- Intermediate nodes have good flow distribution implicit in their traffic
- Also, processing a packet occurs purely within the networking stack itself, no application level work
- End nodes also usually have good flow distribution
- However, there is the added aspect of application cpu usage
- Completely stateless flow steering
- Or, application oriented flow steering

Networking Hardware Design

- Traditionally a single-queue model
- Limitations of bus technology, f.e. PCI
- Advent of MSI and MSI-X interrupts
- RSS based flow hashing
- Multiple TX and RX queues
- Stateless flow distribution
- Extra sophistication: Sun's Neptune 10G Ethernet
- TCAMs and more fine-grained flow steering
- Intel's IXGBE "Flow Director"

NAPI: "New API"

- Interrupt mitigation scheme designed by Jamal Hadi Salim and Robert Olsson
- On interrupt, further interrupts are disabled and software interrupt is scheduled
- Software interrupt "polls" the driver, which processes RX packets until no more pending packets or quota is hit
- Quota provides DRR (Distributed Round Robin) sharing between links
- When polling is complete, chip interrupts are re-enabled

LIMITATIONS OF NAPI

- All state embedded literally inside of “struct netdevice”
- Ideally we want some kind of “NAPI instance” for each chip interrupt source
- But we had no direct way to instantiate such instances structurally
- Fixes were in order

Stephen Hemminger to the Rescue

- Extracted NAPI state into seperate structure
- Device driver could create as many instances as necessary
- Multiple RX queues could be represented using multiple NAPI instances
- And this is exactly what multiqueue drivers do
- Oh BTW: Nasty hacks...

Packet Scheduler

- Sits between network stack and device transmit method
- Supports arbitrary packet classification and an assortment of queueing disciplines
- Has to lock QDISC and then device TX queue to get a packet to the device
- SMP unfriendly, and just like NAPI had state embedded in netdevice struct
- Root qdiscs cannot be shared
- Complicated qdisc and classifier state has “device scope”
- Luckily the default configuration is a stateless and simple qdisc

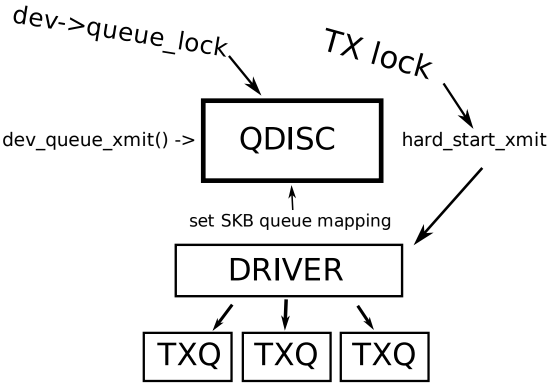
Driver TX Method

- Manages TX queue flow control assuming one queue
- Need to add queue specifier to flow control APIs
- But do so without breaking multiqueue-unaware drivers
- With NAPI we could totally break the API and just fix all the drivers at once
- Only a relative handful of drivers use NAPI
- Breaking the flow control API would require changes to roughly 450 drivers
- So, backward compatible solutions only.

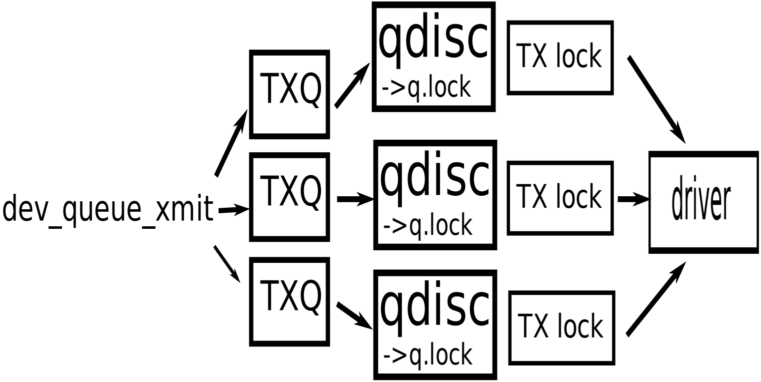
TX QUEUE SELECTION

- Selected queue stored in SKB
- Queue selection function is different depending upon packet origin
- Forwarded packet: Function of RX queue selected by input device
- Locally generated packet: Use hash value of attached socket
- Thorny cases: Devices with unequal RX and TX queues

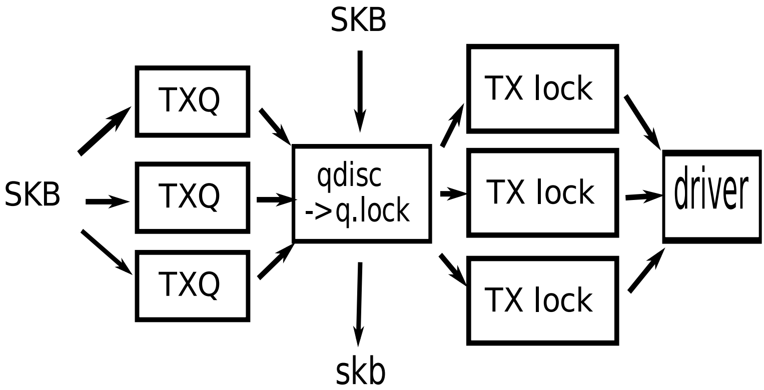
Picture of TX Engine



Picture of Default Configuration



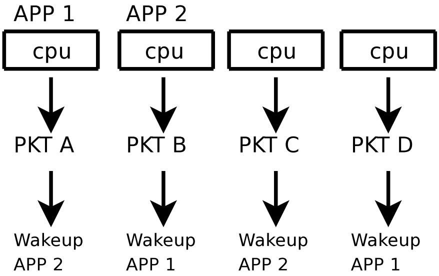
Picture with Non-trivial QDISC



Motivation

- Performance, duh...
- Many networking devices out there are not multiqueue capable
- Whilst stateless RX queue hashing is great for forwarding applications...
- It is decidedly suboptimal for end-nodes.
- Problem: Figuring out the packet's “destination” before it's “too late”

EXAMPLE SCENERIO



APP 1 handles flows B and D
APP 2 handles flows A and C

EARLY EFFORTS

- Influenced by Jens Axboe's remote block I/O completion experiments
- Up to 10 percent improvement in benchmarks where usually a 3 percent improvement is something to brag heavily about
- Generalization of remote software interrupt invocation
- Counterpart usage implemented for networking
- Basically SW multiqueue on receive
- Detrimental for loopback traffic

More Recent Work

- Patch posted by Tom Herbert at Google
- Per-device “packet steering” table, set via sysctl by user
- When packet steering is enabled, receive packets are hashed and this indexes into the table
- Entry found in table is cpu to steer packets to
- Packet steered to foreign cpus using remote SMP calls and special software interrupt
- Whole mechanism is enabled also via sysctrl
- If disabled or no valid entry found in the table, behavior is existing behavior

Another Idea: SW "Flow Director"

- CPU on which transmits for a flow occur is "remembered"
- On receive for that flow, remembered cpu is looked up and packet steered to that CPU
- Problems of space
- Problems of time
- Problems of locality

Credits

- Linus Torvalds, for sharing his kernel instead of keeping it to himself
- Ron Guerin and the rest of the NYLUG folks for letting me present today
- Stephen Hemminger and Rusty Russell for early RX multiqueue work
- Jarek Poplawski, Patrick McHardy, Jamal Hadi Salim, and Eric Dumazet for help with TX multiqueue implementation
- Robert Olsson and Herbert Xu for continuing help throughout all of this
- Tom Herbert and others at Google for ongoing efforts