THE PERFORMANCE ANALYSIS OF LINUX NETWORKING – PACKET RECEIVING

W. Wu*, M. Crawford†, Fermilab MS-368, Batavia, IL 60510, USA

Abstract

The computing models for HEP experiments are becoming ever more globally distributed and grid-based, both for technical reasons (e.g., to place computational and data resources near each other and the demand) and for strategic reasons (e.g., to leverage technology investments). To support such computing models, the network and end systems (computing and storage) face unprecedented challenges. One of the biggest challenges is to transfer physics data sets – now in the multi-petabyte (10^{15} bytes) range and expected to grow to exabytes within a decade – reliably and efficiently among facilities and computation centers scattered around the world. Both the network and end systems should be able to provide the capabilities to support high bandwidth, sustained, end-to-end data transmission. Recent trends in technology are showing that although the raw transmission speeds used in networks are increasing rapidly, the rate of advancement of microprocessor technology has slowed down over the last couple of years. Therefore, network protocol-processing overheads have risen sharply in comparison with the time spent in packet transmission, resulting in the degraded throughput for networked applications. More and more, it is the network end system, instead of the network, that is responsible for degraded performance of network applications. In this paper, the Linux system's packet receive process is studied from NIC to application. We develop a mathematical model to characterize the Linux packet receive process. Key factors that affect Linux systems' network performance are analyzed.

PACKET RECEIVING PROCESS

Figure 1 demonstrates generally the trip of a packet from its ingress into a Linux end system to its final delivery to the application [1][2][3]. In general, the packet's trip can be classified into three stages:

- Packet is transferred from network interface card (NIC) to ring buffer. The NIC and device driver manage and controls this process.
- Packet is transferred from ring buffer to a socket receive buffer, driven by a software interrupt request (*softirq*) [2][4]. The kernel protocol stack handles this stage.
- Packet data is copied from the socket receive buffer to the application, which we will term the *Data Receiving Process*.

In the following sections, we detail these three stages. It will be assumed that the medium is Ethernet-like and that the driver uses the Linux "New API" (NAPI) to reduce interrupt load on the CPU.

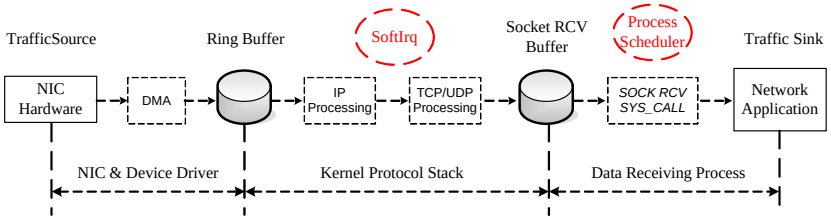

Figure 1 Linux Networking Subsystem: Packet Receiving Process

NIC and Device Driver Processing

The Linux kernel uses a structure *sk_buff* [2] to hold any single packet up to the MTU (Maximum Transfer Unit) of the network. The device driver maintains a "ring" of these packet buffers, known as a "ring buffer," for packet reception (and a separate ring for transmission). A ring buffer consists of a device- and driver-dependent number of packet descriptors. To be able to receive a packet, a packet descriptor should be in "ready" state, which means it has been initialized and pre-allocated with an empty *sk_buff* memory-mapped into I/O space for DMA. When a packet comes, one of the ready packet descriptors in the reception ring will be used, the packet will be transferred into the pre-allocated *sk_buff*, and the descriptor will be marked as used. A used packet descriptor should be reinitialized and refilled with an empty *sk_buff* as soon as possible for further incoming packets. If a packet arrives and there is no ready packet descriptor in the reception ring, it will be discarded.

Figure 2 shows a general packet receiving process at NIC and device driver level. When a packet is received, it is transferred into main memory and an interrupt is raised only after the packet is accessible to the kernel. When CPU responds to the interrupt, the driver's *interrupt handler* is called, which schedules a softirq. It puts a reference to the *device* into the *poll queue* of the interrupted CPU and disables the NIC's receive interrupt until the packets in its ring buffer are processed.

* wenji@fnal.gov
† crawdad@fnal.gov

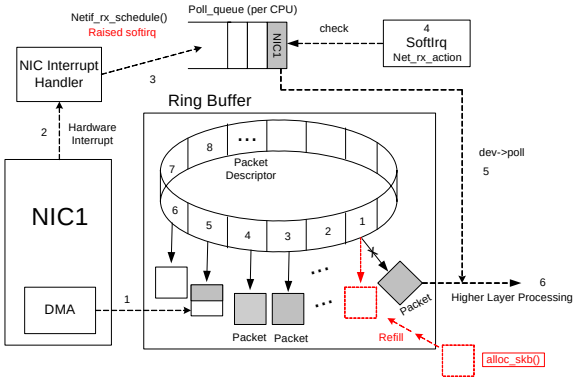

Figure 2 NIC & Device Driver Packet Receiving

The softirq is serviced shortly afterward. The CPU polls each device in its poll queue to get the received packets from the ring buffer by calling the *poll* method of the device driver. In *dev->poll()*, each received packet is passed upwards for further processing. After a packet is dequeued from its receiving ring buffer, its corresponding packet descriptor needs to be reinitialized and refilled.

Kernel Protocol Stack

- *IP Processing*

The IP protocol receive function *ip_rcv()* is called when an IP packet is dequeued. It verifies packet integrity and invokes any *netfilter* hooks, then calls *ip_rcv_finish()*, which routes the packet over the network or delivers it locally by *ip_local_deliver()*. There, the higher layer protocol is determined and the corresponding handler function is invoked: *tcp_v4_rcv()* and *udp_rcv()* are two examples.

- *TCP Processing*

After header sanity checks, the TCP code looks up the *socket* associated with the packet. Each socket has a lock to protect its data from un-synchronized modification. If the socket is locked, the packet is placed on the *backlog queue* and all further processing is deferred. If the socket is not locked, and its *Data Receiving Process* is waiting for data, the packet is added to the socket's *prequeue* and will be processed in batch in the *process context*, instead of the *interrupt context* [4] If the prequeue mechanism does not accept the packet, then the socket is not locked and no process is waiting for input on it. The packet must be processed immediately by a call to *tcp_v4_do_rcv()*, which is also called to drain the backlog queue and prequeue. It is through this function that packets are acknowledged, and window and round-trip time estimates are updated. Here, we focus on the data packet processing.

The packet is handled on the so-called *fast path* if it is exactly the next packet expected in a received sequence with no gaps. If, in addition, the socket belongs to the currently active process and the data fits into the application-supplied buffer, the data will be copied directly from the *sk_buff* to user memory. This direct copy can also occur on the *slow path* if the packet fills the beginning of the first hole in the received stream. On either path, if the data cannot be copied directly to user

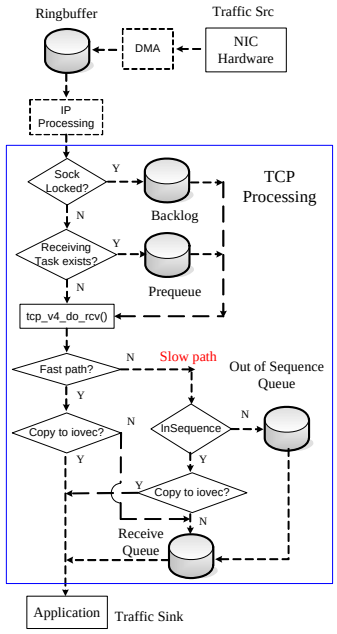

Figure 3 TCP Processing - Interrupt context

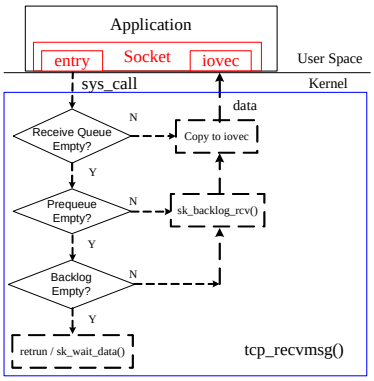

Figure 4 TCP Processing – Process Context

space, it goes onto the *receive queue*. Figure shows the various queues the packet may occupy, including the *out-of-sequence queue*, which holds packets received out of order until the gaps preceding them are filled. Unlike the other three queues, packets in the *receive queue* are guaranteed in order, acked, and without gaps. Packets in out-of-sequence queue are moved to the *receive queue* when additional packets fill the preceding holes in the data stream.

As previously mentioned, the *backlog* and *prequeue* are drained in the *process context* (except in the case of prequeue overflow). The socket's *data receiving process* obtains data from the socket through socket-related receive system calls. For TCP, all such system calls result in the final calling of *tcp_recvmsg()*, which is the top end of the TCP transport receive mechanism. As shown in Figure 4, *tcp_recvmsg()* first checks the *receive queue*. Since packets in the receive queue are guaranteed in order, acknowledged, and without gaps, their data is copied to user space directly. Then *tcp_recvmsg()* will process the *prequeue* and *backlog queue*, respectively, if

they are not empty and space in the user buffer remains. *tcp_v4_do_rcv()* is called for both of these queues. Afterward, processing is performed similar to that in the interrupt context.

- *UDP Processing*

When a UDP packet arrives from the IP layer through *ip_local_deliver()*, it is passed on to *udp_rcv()*. *udp_rcv()'s* mission is to verify the integrity of the UDP packet and to queue (*udp_queue_rcv_skb()*) one of more copies for delivery to multicast and broadcast sockets and exactly one copy to unicast sockets. When queuing the received packet in *the receive queue* of the matching socket, if there is insufficient space in the receive buffer quota of the socket, the packet may be discarded. Data within the Socket's receive buffer are ready for delivery to the user space.

Data Receiving Process

Packet data is finally copied from the socket's receive buffer to user space by *data receiving process* through socket-related receive system calls. The receiving process supplies a memory address and number of bytes to be transferred, either in a *struct iovec*, or as two parameters gathered into such a struct by the kernel. All the TCP socket-related receive system calls result in the final calling of *tcp_recvmsg()*, which will copy packet data from socket's buffers (*receive queue*, *prequeue*, *backlog queue*) through *iovec*. For UDP, all the socket-related receiving system calls result in the final calling of *udp_recvmsg()*, which is the front end to the UDP transport receive mechanism. When *udp_recvmsg()* is called, data inside *receive queue* is copied through *iovec* to user space directly.

PERFORMANCE ANALYSIS

Based on the packet receiving process described in previous sections, the packet receiving process can be described by the model in Figure 5. In the mathematical model, the NIC and device driver receiving process can be represented by the token bucket algorithm [5], accepting a packet if a ready packet descriptor is available in the ring buffer and discarding it if not. The rest of the packet receiving processes are modelled as queuing processes [6].

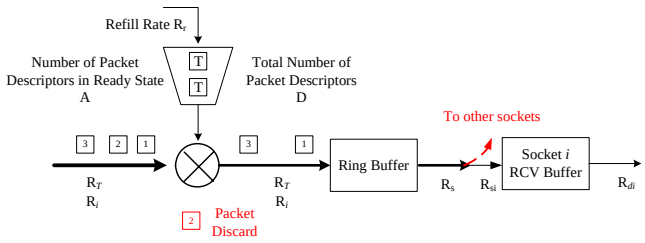

Figure 3 Packet Receiving Process - Model

We assume several incoming data streams are arriving and define the following symbols (all rates are in packets per time unit):

- $R_T(t)$, $R_{T'}(t)$: Offered, accepted total packet rate
- $R_i(t)$, $R_{i'}(t)$: Offered, accepted packet rate for data stream *i*
- $R_r(t)$: Refill rate for used packet descriptors
- D: The total number of packet descriptors in the receive ring buffer;
- $A(t)$: Number of packet descriptors in ready state
- $\tau_{\min}$: The minimum time interval between a packet's ingress into the system and its first being serviced by a softirq;
- $R_s(t)$: Kernel protocol packet service rate
- $R_{si}(t)$: Softirq packet service rate for stream i
- $R_{di}(t)$: Data receiving process packet service rate for stream i
- $B_i(t)$: Socket i's receive buffer size (Bytes);
- QB_i: Socket *i*'s receive buffer quota (Bytes);

The Token Bucket algorithm is a surprisingly good fit to the NIC and device driver receiving process. In our model, the receive ring buffer is represented as a token bucket with a depth of D tokens. Each packet descriptor in the ready state is a token, granting the ability to accept one incoming packet. The tokens are regenerated only when used packet descriptors are reinitialized and refilled. If there is no token in the bucket, incoming packets will be discarded.

The offered and accepted packet rates are related by

$$R_{T'}(t) = \begin{cases} R_T(t), & A(t) > 0 \\ 0, & A(t) = 0 \end{cases} \quad (1)$$

So to admit packets without discarding, the system should satisfy

$$\forall t > 0\ ,\ A(t) > 0 \quad (2)$$

It can be derived that

$$A(t) = D - \int_0^t R_T(\tau)d\tau + \int_0^t R_r(\tau)d\tau\ ,\ \forall t > 0 \quad (3)$$

Since $R_{T'}$ is a function of both R_T *and* $A(t)$, it cannot be controlled directly. In order to avoid or minimize packet drops, the system needs to raise its $R_r(t)$ and/or D. $R_r(t)$ depends on the following two factors: (1) Protocol packet service rate $R_s(t)$. To raise the packet service rate, approaches to reducing processing in the kernel have been proposed. For example, TCP/IP Offloading technology [7] aims to free the CPU of some packet processing by shifting tasks to the NIC or storage device. Also, when the system is in memory pressure, allocation of new packet buffers is prone to failure and a used packet descriptor cannot be refilled. Absent memory shortage, it can be assumed that $R_s(t) = R_r(t)$.

D is a design parameter of the NIC and driver. A larger D implies increased cost. D should meet the following condition to avoid packet drops:

$$D \geq \tau_{\min} * R_{\max} \quad (4)$$

The rest of the packet receiving processes are modeled as queuing processes. In the model, socket *i*'s receive buffer is a queue of size QB_i. The packets are put into the queue by softirq with a rate of $R_{si}(t)$, and are moved out of the queue by the *data receiving process* with a rate of $R_{di}(t)$.

For stream *i*, based on the packet receiving process,

$$R_i(t) \le R_i(t) \quad \text{and} \quad R_{si}(t) \le R_s(t) \tag{5}$$

Similarity, it can be derived that:

$$B_i(t) = \int_0^t R_{si}(\tau)d\tau - \int_0^t R_{di}(\tau)d\tau \tag{6}$$

In transport layer protocol operations $B_i(t)$ plays a key role. For UDP, when $B_i(t) \ge QB_i$, all the incoming packets for socket *i* is discarded and all the protocol processing effort spent the dropped packet is wasted. From both the network end system and network application's perspective, this is a condition to avoid.

TCP does not drop packets at the socket level as UDP does when the receive buffer is full. Instead, it advertises $QB_i - B_i(t)$ to the sender to perform flow control. Instead, when a TCP socket's receive buffer is near full, the small window $QB_i - B_i(t)$ advertised to the sender side will throttle the data sending rate, resulting in lower TCP throughput [8].

From both UDP and TCP's perspectives, it is desirable to raise the value of $QB_i - B_i(t)$, which is:

$$QB_i - \int_0^t R_{si}(\tau)d\tau + \int_0^t R_{di}(\tau)d\tau \tag{7}$$

Clearly it is not desirable to reduce $R_{si}(t)$ to achieve the goal. But the goal can be achieved by raising QB_i and/or $R_{di}(t)$. On most operating systems, QB_i is configurable, subject to system memory limits.

$R_{di}(t)$ depends on the system load and the data receiving process' *nice* value. Linux is a *preemptive multi-processing* operating system. Processes are scheduled to run in *prioritized round robin* fashion. Each process' *time slice* is calculated based on its nice value. When a process' time slice runs out, the process is considered expired. A process with no time slice is not eligible to run until all other processes have exhausted their time slice. At that point, the time slices for all process are recalculated [4]. Figure 6 shows the data receiving process' running model.

Let's assume the data receiving process' packet service rate is constant $\hat{\lambda}$ when the process is running. For the cycle *n* in Figure 6, we have:

$$R_{di}(t) = \begin{cases} \hat{\lambda}, & 0 < t < t_1 \\ 0, & t_1 < t < t_2 \end{cases} \tag{8}$$

Therefore, to raise the rate of $R_{di}(t)$, the only way is to increase data receiving process' CPU share: either lower data receiving process' *nice* value (negative values are favored over positive), or reduce the system load to increase the process' running frequency.

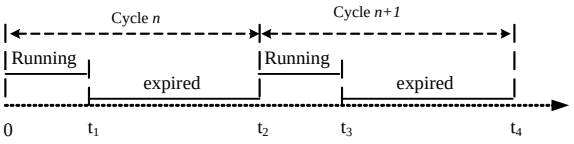

Figure 4 Data receiving process running model

OBSERVATIONS

We have instrumented the Linux network stack to snapshot the packet queues at regular intervals and record their lengths in a memory area for later examination. We were able to observe exhaustion of the ring buffer and the consequent decrease in throughput, and the fluctuations in packet processing under load. A more complete presentation of the observational results will be published elsewhere.

CONCLUSION

In this paper, the Linux system's packet receiving process is studied from NIC to application. Our mathematical model of the process implies, and instrumented observation support, that key factors affecting Linux systems' network performance include: (1) The reception ring buffer in NIC and device driver as a cause of packet drops. (2) The data receiving process' CPU share gates TCP processing and queue servicing.

ACKNOWLEDGEMENTS

This work was supported by the U. S. Department of Energy under contract DE-AC02-76CH03000. The authors are grateful for the assistance of the Fermilab Data Communications group in setting up a network testbed.

REFERENCES

[1] M. Rio, et.al., "A Map of the Networking Code in Linux Kernel 2.4.20", March 2004.

[2] K. Wehrle, et.al., The Linux Networking Architecture – Design and Implementation of Network Protocols in the Linux Kernel, Prentice-Hall, ISBN 0-13-177720-3, 2005.

[3] www.kernel.org

[4] R. Love, Linux Kernel Development, Second Edition, Novell Press, ISBN: 0672327201, 2005.

[5] A. Tanenbaum, Computer Networks, 3rd Edition, Prentice-Hall, ISBN: 0133499456, 1996.

[6] A. Allen, Probability, Statistics, and Queueing Theory with Computer Science Applications, 2nd Edition, Academic Press, ISBN: 0-12-051051-0, 1990.

[7] G. Regnier, et.al., TCP onloading for data center servers, Computer, Volume 37, Issue 11, Nov. 2004 Page(s):48 - 58

[8] Transmission Control Protocol, RFC 793, 1981