Challenges Using the Linux Network Stack for Real-Time Communication

Michael M. Madden[*]
NASA Langley Research Center, Hampton, VA, 23681

Starting in the early 2000s, human-in-the-loop (HITL) simulation groups at NASA and the Air Force Research Lab began using the Linux network stack for some real-time communication. More recently, SpaceX has adopted Ethernet as the primary bus technology for its Falcon launch vehicles and Dragon capsules. As the Linux network stack makes its way from ground facilities to flight critical systems, it is necessary to recognize that the network stack is optimized for communication over the open Internet, which cannot provide latency guarantees. The Internet protocols and their implementation in the Linux network stack contain numerous design decisions that favor throughput over determinism and latency. These decisions often require workarounds in the application or customization of the stack to maintain a high probability of low latency on closed networks, especially if the network must be fault tolerant to single event upsets.

I. Introduction

The human-in-the-loop (HITL) simulation groups at NASA and the Air Force Research Lab have used the Linux network stack for some real-time communication since the early 2000s [1, 2]. More recently, SpaceX adopted Ethernet as the primary bus technology for its Merlin engines, and the Falcon and Dragon flight computers use a Linux kernel for the operating system and network communication [3, 4]. This paper examines the use of the User Datagram Protocol (UDP) and Transport Control Protocol (TCP) executed on top of the Internet Protocol (IP) for real-time communications. UDP/IP and TCP/IP have been designed for communication over the open Internet, which provides no latency guarantees. Furthermore, the Linux network stack implements these protocols using optimizations that favor throughput over determinism and latency. These decisions often require workarounds in the application or customization of the stack to maintain a high probability of low latency, especially if the network must be fault tolerant to single event upsets. Even then, a high probability of low-latency is possible only on closed networks. Communication on a local area network that is open to a wide-area network or the Internet is subject to the unpredictable processing of external traffic and vulnerable to malicious actors. Therefore, this paper assumes the IP communication occurs on a closed network whose traffic only originates from the applications and devices within the network.

A. What is Real-Time Communication?

Not every form of communication performed in the context of a real-time application is itself real-time. Like real-time applications, real-time communication are defined by the presence of deadlines, after which content is considered obsolete and invalid. This paper evaluates the ability of the Linux stack to support the following three types of real-time communication:

- Synchronous: Under synchronous real-time communications both the send and receive operations are sequential and must complete within a defined execution window.
- Scheduled receive: The receive operation occurs at a scheduled time, usually periodically. The send operation can occur anytime between scheduled receive operations (e.g., within the frame) as long as the data will be ready when the receive operation starts.
- Limited lifespan data: The send operation may be periodic or aperiodic but the data has an expiration time. The reader can receive the data at any time before it expires.

However, real-time applications can also perform communications with no deadlines such as recording telemetry. For all communication in a real-time application, the execution of send and receive operations within the

[*] Chief Engineer for Modeling and Simulation; Senior Member of AIAA.

application's context must have deterministic bounds. However, communications without deadlines can tolerate delays within the network stack for either emitting outgoing data or processing incoming data. Such delays typically manifest as delayed kernel work performed outside the application context, such as softirqs[†]. On the other hand, to assure deadlines for real-time communication, it is often preferable that the application data be emitted in the context of the application's send operation or that received data is placed in the application's receive buffers with minimal delay starting from the time of the associated NIC interrupt signal. This paper highlights behaviors in the network stack that either add uncertainty to the execution of send or receive operations in the application context or that add delays (i.e., latency) to data processing within the network stack.

B. Methods of Configuring IPv4 Communications

Linux provides four mechanisms that developers, system administrators, or operators can use to configure IPv4 communications. These are kernel parameters, socket options, socket flags, and route metrics. The kernel parameters apply system-wide. Some parameters will apply to all IP sockets and others to all sockets using a given protocol, TCP or UDP. Kernel parameters can be set when compiling the kernel, when booting the kernel, or at runtime using the sysctl command. Socket options apply to a given socket, and the application sets them using the setsockopt() function. Socket flags configure a specific operation [e.g., a send() call], and the application passes the flags as arguments to the operation. Route metrics are cached information that initialize new connections to a destination with the metrics determined by a prior connection to that destination. These metrics apply to a specific route and affect all sockets that communicate over that route. Operators can use the ip command to define those metrics before loading the application in order to establish a desired initial state for its connections.

II. Using the Maximum Transmission Unit in Output Block Design

One influence over data transmission latency is the Maximum Transmission Unit (MTU) since it determines how many fragments the network stack must divide an output block into for transmission. Two MTU measures affect IP transmissions, the device MTU and the path MTU. The device MTU is the largest size, in bytes, of the payload in an Ethernet frame that a network device can transmit.[‡] The path MTU is the smallest device MTU among the network devices (NICs, switches, and routers)[§] along the route between two nodes. The Ethernet standard supports a maximum device MTU of 1500 bytes that is contained in an Ethernet frame of up to 1542 bytes (including the interpacket gap). In a closed network, where the system engineers control the selection, design, and configuration of all networking devices, the path MTU between any two nodes should equal the Ethernet maximum of 1500 bytes.

If an application sends an IP datagram larger than the path MTU, then the IP layer divides the datagram into fragments no larger than the path MTU. The disassembly of the datagram at the source and reassembly of the datagram at the destination increase the latency of the transmission and consumes resources on both the source and destination node s. Therefore, designing output blocks so that the resulting IP datagrams are equal to or less than the path MTU can improve both efficiency and reliability of the communications. Furthermore, real-time applications should also avoid processing and bandwidth penalties that occur if the application sends numerous messages much smaller than the path MTU. An application should attempt to consolidate multiple small output blocks into fewer IP datagrams that approach the path MTU. IP fragmentation has an additional disadvantage. If any fragment becomes lost (e.g., due to corruption), then the destination node discards the whole datagram. Linux disables IP fragmentation for UDP/IP and TCP/IP communications by default to perform Path MTU Discovery (see section III) so this loss condition is normally prevented. Nevertheless, there are circumstances where allowing IP fragmentation makes sense to accomplish real-time communications with low latency (see section II.A).

The IP header reduces payload available for application data in the Ethernet frame by 20 bytes. When using UDP, the UDP header occupies 8 bytes. Thus, a UDP datagram can contain 1472 bytes of application data without causing IP fragmentation and this would be the optimal message size for UDP communication. Moreover, it is the largest UDP datagram that a socket can send under defaults that disable IP fragmentation. When using TCP, the

[†] Linux device drivers are divided into two halves: an interrupt service routine and a softirq. The softirq implements driver work that can be interrupted. However, the kernel also uses the softirq feature to defer other critical kernel work, including data movement within the network stack.

[‡] Ethernet has no concept of a 'Maximum Receive Unit'. The Ethernet standard requires that all networking devices are capable of receiving the maximum MTU defined by the standard.

[§] A switch, as differentiated from a router, does not typically have a configurable MTU. It simply forwards the packets that it receives and its effective MTU is the Ethernet maximum. Modern switches, however, may have features that blur the line between a switch and a router to include a configurable MTU.

Linux TCP implementation does a reasonably good job of ensuring that the data stream is divided into optimally sized IP datagrams (by using Path MTU Discovery), and the socket should not require fine tuning by the application. However, the application can still optimize performance by ensuring that it tunes message sizes to be at or just below a multiple of 1448 bytes (1500 byte MTU - 20 byte IP header - 20 byte TCP header - 12 byte timestamp).

A. Can IP fragmentation be a good thing?

IP fragmentation should only be used with a UDP socket, since many aspects of TCP are designed for efficiency assuming there is no IP fragmentation of segments. Use of IP fragmentation can benefit real-time communications when the following is true of the output block:

- The block is larger than can fit in the path MTU.
- The application cannot make use of data in a partially received block.
- Transmission of a block is subject to a low-latency deadline, e.g., the length of an execution frame.
- The whole block is invalid if it is late, i.e., the application drops any late blocks.
- The system is not required to attempt data retransmission as part of its recovery response when all or part of a block is lost or corrupted. Otherwise, the application would be better off dividing the block into MTU-sized chunks so that partial retransmission is possible.

UDP is limited to a size of approximately 64 kilobytes, so data blocks any larger than this would need to be broken into multiple UDP datagrams. Furthermore, the receiving application must then be made robust to lost or late datagrams since UDP provides no reliable retransmission or guaranteed deadlines.

B. Jumbo Ethernet Frames

Some Gigabit Ethernet devices support jumbo frames that allow an increase in device MTU up to 9000 bytes. If the hardware and drivers support it, configuring the application, OS, and hardware to use a jumbo frame can improve performance for transferring larger data sets. However, for TCP/IP, Jumbo Ethernet Frames can also increase the retransmission latency when a packet is lost (see section XI.I).

C. Device MTU of Loopback Interface

Linux provides a loopback interface that enables two tasks on the same computer to communicate with each other using IP. The device MTU of the loopback interface is typically set to 65536 bytes. Therefore, the loopback interface allows UDP datagrams up to 64 kB to be sent without fragmentation. Furthermore, TCP packets will have a 64 kB size over the loopback interface.

D. Cork data transmissions

The cork feature is a method for combining multiple small send operations into fewer IP datagrams. The application uses cork to tell the operating system that additional data is forthcoming in the send buffer and that it can consolidate this data for transmission. Linux provides corking for both TCP and UDP communications. However, cork is not defined in the RFCs governing TCP/IP and is not portable. Under Linux, both TCP and UDP can enable cork with the send()/sendto() call by setting the MSG_MORE flag. The operating system can then allow data to accumulate until a send()/sendto() call is made without the flag set. Both TCP and UDP may also enable cork using setsockopt() though the flags are protocol specific (TCP_CORK and UDP_CORK). When this option is used, all send operations will cork until the flags are unset using setsockopt(). In the case of UDP, all the 'corked' data is collected into a single datagram and is subject to datagram size restrictions. Therefore, UDP will not transmit any corked data until the cork is lifted. TCP could process corked data once it represents a full-sized segment (see XI.C.1); however, it is not clear from the Linux man page for TCP whether Linux does this. Additionally, TCP places a 200 millisecond timeout on the cork.

III. Path MTU Discovery

By default, Linux enables Path MTU Discovery (PMTUD) for both TCP and UDP sockets. To accomplish this, Linux transmits IP datagrams with the 'do not fragment' bit (DF bit) set in the IP header. Then, any device on the path with a smaller device MTU will drop the IP datagram and respond with an Internet Control Message Protocol (ICMP) message containing the device’s MTU. If the application sends a UDP datagram larger than the detected path MTU, then the network stack will generate an error and will drop the datagram. The application is left to check for the error. In order to allow transmission of larger UDP messages, one must modify the socket configuration to disable PMTUD. On the other hand, TCP automatically adjusts its maximum segment size (MSS) to match the current path MTU, so the application does not need to take action to size data for the current path MTU or check for dropped IP datagrams. Nevertheless, for a closed LAN or direct connection, the path MTU should be known and never change. In fact, only the source node determines the path MTU on the closed LAN assuming it uses a layer-2 switch and not a router. Therefore, any change to the device MTU at the source node (e.g., due to corruption) should be handled by the source node's IP layer without a need for PMTUD, and disabling PMTUD should be safe.

The kernel parameter (ip_no_pmtu_disc) or the socket option (IP_MTU_DISCOVER) can control PMTUD. The kernel parameter affects all IP sockets (both TCP and UDP), and the socket option affects a single socket. For UDP, disabling PMTUD can reduce resources used by the kernel and may speed up socket writes that fit in a single Ethernet frame since the write operation no longer has to retrieve the value of the current path MTU. Disabling PMTUD also enables UDP datagrams larger than the path MTU to a maximum of 64 kB; the stack will perform IP fragmentation for such datagrams (see section II). Disabling PMTUD may have fewer benefits for TCP. Though it would also reduce resources used by the kernel to track path MTU, it may not affect the speed of writes since TCP automatically adjusts its MSS in response to a change in path MTU detected by the receive path in the network stack. Therefore, TCP write operations do not query the current path MTU during the send operation.

IV. Address Resolution Protocol (ARP)

For the Ethernet hardware to transmit packets between nodes, the network stack needs to map IP addresses to link-layer addresses (e.g., MAC Address). The network stack can determine this mapping dynamically or can work from a static map. Dynamic mapping uses the Address Resolution Protocol (ARP) and is the default for the network stack. ARP employs a request-reply exchange of ARP packets to map an IP address to a link-layer address. The exchange delays transmission of application data by a little more than the round-trip transmission time of ARP packets. To avoid the added latency and bandwidth overhead of the ARP exchange for every IP datagram transmission, the operating system maintains an ARP cache. Entries in this cache remain valid until a time-out occurs since the last confirmed contact with the destination node. Once that time-out occurs, another ARP exchange must occur before the next IP datagram can be sent to the destination node, which adds latency to that IP datagram.

TCP sockets will inform the ARP cache of the liveliness of the destination node when an acknowledge (ACK) packet is received from that node. However, UDP sockets do not have an automated means to judge the liveliness of the connection since UDP does not utilize replies. Without action by the application, ARP cache entries associated with UDP sockets will periodically timeout and require revalidation. The default timeout on most Linux installations is 60 seconds. The application can avoid this timeout by setting the MSG_CONFIRM flag when writing to the UDP socket. However, for closed networks, disabling ARP and configuring a static map is the recommended option.

V. Send and Receive Buffers

By default, Linux dynamically tunes the send and receive buffers for TCP/IP and UDP/IP sockets. The tuning is a heuristic response to the actual traffic and the total memory consumption of sockets using the same protocol (TCP or UDP). Dynamic sizing can cause page faults during operation, it can block send operations unless the operation or socket is set to nonblocking, and can increase the latency of receive operations. Nonblocking send operations will not cause the send buffer to grow dynamically but will instead return with a failure if there is insufficient room left in the send buffer to hold the data being written to the socket. Therefore, if the buffer is not appropriately sized to handle worst-case output load, then nonblocking send operations can produce preventable failures. In extreme cases, failing to adjust the receive buffers can also lead to dropped packets. Real-time applications may, therefore, need to adjust system-wide or per-socket sizes for the send and receive buffers to assure read and write operations do not a) cause buffer sizes to change during operation, b) cause failure of nonblocking send operations, or c) cause loss of received packets. In memory constrained environments, it may also be necessary to change system-wide or per-socket sizes to prevent the IP communications from utilizing too much of the system memory when defaults are too generous given the worst-case I/O.

Per socket, send and receive buffer sizes for UDP and TCP sockets are set using the setsockopt() system call with the SO_SNDBUF or SO_RCVBUF socket options, respectively. When the application sets buffers using setsockopt, Linux allocates double the size requested and uses the extra space for administrative purposes. Furthermore, setting the buffer using setsockopt() also disables dynamic tuning of the buffer size. The buffer will remain fixed at the requested size.

System defaults for dynamic buffer tuning are set using kernel parameters, and the UDP and TCP protocols respond to different parameters. The default send and receive buffers for UDP sockets are set by the kernel parameters net.core.wmem_default and net.core.rmem_default, respectively. For TCP, the kernel parameters are net.ipv4.tcp_wmem and net.ipv4.tcp_rmem, respectively. These kernel parameters, for both TCP and UDP, define the total buffer size including both application data and administrative data; therefore, they should be set to twice the value that the application would set with a setsockopt() call. The kernel parameters net.ipv4.tcp_mem for TCP and net.ipv4.udp_mem for UDP control the thresholds, at which the kernel more aggressively manages the memory utilization of send and receive buffers. (Moreover, the value of these threshold parameters are in units of pages and not bytes.) The kernel parameters net.ipv4.tcp_wmem_min, net.ipv4.tcp_rmem_min, net.ipv4.udp_wmem_min, and net.ipv4.udp_rmem_min set the minimum size of send and receive buffers when memory is aggressively managed.

Most real-time systems communicate the same data in each rate group. Therefore, for each socket, there is a known worst-case amount of data sent in an execution frame, max_send. There is also a known worst-case amount of data received between read operations, max_receive. For a given socket connection, max_receive for the receiver is often twice the max_send of the sender(s) to cover the scenario where a send operation occurs immediately after a read operation (late send) and the next send operation occurs immediately before the next read operation (early send). For this discussion, the buffer sizes presented are the total buffer size, including both application and administrative data. For UDP sockets, data placed in the send buffer is processed immediately; therefore, a send buffer sized to at least twice max_send should be adequate. Likewise, the receive buffer for a UDP socket should be sized to at least twice max_receive. Higher sizes may be necessary if nominal and off-nominal scenarios can cause delays either in the kernel processing of send buffers or in the application's read operation. Because TCP has complex logic for when data in the send buffer is processed and when data in the receive buffer is made available to the application, send and receive buffers for TCP sockets may need to be larger in order to be robust not only to nominal latencies but to latencies induced by packet loss. Furthermore, the size of the receive buffer does impact how much data the sender can process before receiving an acknowledgment (see section XI.F), and brute force methods that maintain low latency may require increased buffer sizes (see section XI). Thus, there is no easy rule of thumb for sizing TCP buffers; it depends on the volume of data writes, the timing of data reads, and the desired maximum latency for both nominal and off-nominal scenarios.

Socket buffers, like other memory in the Linux kernel, may be subject to deferred page allocation by the Linux memory manager. In other words, Linux tells the application that it has allocated the complete memory request but often does not map the pages to the system's virtual memory until each page is first written to. Thus, to avoid page faults in send or receive operations, it may be necessary for the application to execute send operations during startup that are intended to fill the buffers on both senders and receivers as best one can.

VI. Blocked versus Nonblocked Send and Receive

By default, sockets perform send and receive as blocking operations. In other words, a send operation will block until all the application's data can be placed in the send buffer; a receive operation will block until data is available in the receive buffer. Both blocks can be limited with timeouts but those timeouts are subject to the resolution of the system clock, which is one millisecond on the PowerPC and Intel x86 platforms. The application can configure the socket to be nonblocking for all operations by applying the O_NONBLOCK option to its file descriptor using fcntl() or the nonblocking flag (MSG_DONTWAIT) can be set for select socket operations like send and receive.

For TCP/IP and UDP/IP, there is no benefit to perform a blocking send operation in a real-time application. If the send buffer is appropriately sized for the application, then the buffer should always have sufficient room when the application performs a send under normal conditions. A lack of space, therefore, represents a true error condition. For receive operations, real-time systems using IP-based communications should be designed with the expectation that data communication is inherently asynchronous. Data from sources can arrive at any time, and the system simply uses the latest data it has received when it is ready to process that data. The system should also be robust to the occasional execution frame that has no fresh data and the occasional frame containing both fresh and stale data. The application should use a nonblocking read operation for a receive operation since, if no data is ready to receive, the time spent blocking for fresh data should be treated as indeterminate.

If the real-time system attempts to use IP to emulate synchronous communication, then a blocking receive operation may appear to make sense on the surface. The receiver could set a timeout that represents the maximum time that the receiver can wait for data to arrive. A timeout would then represent an error. However, timeouts set through the socket interface have a resolution of one millisecond on the PowerPC and Intel x86 platforms. If the application requires a timeout less than a millisecond, then the application can use system calls for I/O event notification that accept timeout values with nanosecond or microsecond resolution and utilize the high resolution timer; these include select(), pselect(), ppoll(), and epoll_pwait(). These calls would be paired with a non-blocking socket receive(). Nevertheless, to improve the reliability of synchronous communication between nodes, additional action may be required to prioritize synchronous traffic over other traffic. These options are discussed further in the next section.

VII. Temporal Ordering Among Sockets is not Guaranteed

IP provides no guarantees that the temporal order of operations among different sockets will be preserved. For example, if an application performs a send operation on Socket A and subsequently performs a send operation on Socket B, IP does not guarantee that the data on Socket A will be transmitted before data on Socket B. Moreover, if Socket A and Socket B are transmitting to the same destination, there is also no guarantee that the data in Socket A will be available to applications before the data in Socket B. Whether data appears on the network in the temporal order of the socket operations is due to a number of factors including the protocol used on the socket, the occurrence of latency inducing events described elsewhere in this document, the quality of service settings for the data including socket priority and traffic control settings, and the propensity of the network hardware to reorder packets (e.g., whether the transmitting NIC may reorder packets under some circumstances, whether the software on switches or routers reorders packets, whether there exists multiple routing paths with equal metric but differing latencies). For example, if ARP has not been effectively disabled (see section IV), then data on Socket A can be delayed by an ARP exchange while data in Socket B is transmitted to a different destination.

For flight software, the impact of temporal reordering of socket operations is normally a concern only when multiple sockets are used to communicate between the same two end-points. Under the default behavior of point-to-point or switched, local-area networks with single paths between end-points, the order of packets from a given source to a given destination is normally preserved. Only when quality of service features are enabled on the networking hardware and the outgoing packets have differentiated type-of-service (TOS) will the network potentially reorder packets between two given end-points. Therefore, absent quality of service, reordering occurs as the result of protocol logic and kernel activity. The other sections discuss techniques to reduce latency and increase determinism of the UDP and TCP protocols. Applying these techniques will also reduce but alone cannot guarantee protection against reordering of socket operations. TCP sockets may be especially difficult if not impossible to configure for guaranteed ordering of socket operations since a number of features can stall transmission of some or all of the output block. The following additional strategies may be necessary to preserve the order of select socket operations across a closed network:

- Increase the time span between socket operations by such means as inserting other application work between operations, reordering operations, or inserting delays. On transmission, a socket using TCP may delay the transmission of some or all of the data for one or more round trip times and some options like TCP Segmentation Offload (see section XI.G) may introduce longer delays. During such delays, other sockets may continue to transmit data resulting in out-of-order transmission. On the receiving end, closely spaced socket operations may result in the IP stack processing the data from multiple sockets as a group. This may result in the data for multiple sockets having near-simultaneous availability to the application. In this case, the operations are not reordered but the data from earlier operations is delayed by the close arrival of data from later operations.
- Use the same socket for all transmissions to a destination. The Linux kernel does its best to maintain the order of send operations within a given socket. However, this imposes a single protocol and serial transmission for all the data, and it increases the complexity and scope of off-nominal scenarios when multiple read operations are performed on the socket at the destination. Take the example of a UDP socket that is used for five send operations of differing sized data blocks, and five receive operations are performed by different software components at the destination. If the packet containing the third data block is lost, then the third read may instead receive the data block intended for the fourth read. The component performing that third read must be able to identify the miscommunication and the fault is not isolated to that component. The fault affects the next two components performing read operations on the socket.
- Configure traffic control to prioritize packets based on the order in which send operations are performed on the sockets (see VII.A). This action alone will not guarantee that packet order resembles the order of send operations. Protocol delays may still cause reordering.

A. Traffic Control and Socket Priority

The Linux kernel provides a rich traffic control feature to manage and manipulate packet traffic. Traffic control may be useful when the real-time system has periods of heavy I/O that mix synchronous real-time communications, other forms of real-time communication, and asynchronous communications. Traffic control becomes available when 'advanced routing' is enabled in the Linux kernel (option CONFIG_IP_ADVANCED_ROUTER). Many distributions set this option by default. Under this option, the default queueing discipline (qdisc) for the network stack becomes a three-band, first-in-first-out (FIFO) packet queue called pfifo_fast. Each time the kernel processes the pfifo_fast qdisc, the kernel processes each band in order from zero to two. The type-of-service (TOS) field in the

IP header determines the band into which the network stack places a packet. The TOS field can be set directly for all packets in a socket using the socket option IP_TOS. However, the recommended method is to set TOS indirectly using the socket option SO_PRIORITY since this option is not protocol specific. The tc-prio(8) man page presents a table that maps socket priority to TOS setting and to the pfifo_fast bands. A real-time application can use socket priority to assure, for example, that packets from sockets performing synchronous I/O get priority for transmission over packets from other sockets. Traffic control, however, only makes a difference when socket send operations are performed close enough in time with large enough volume that the qdisc accumulates packets from multiple sockets. If transmissions are sparse and low volume, then each send operation is likely to place all of its data onto the NIC in the context of the send() call, and packets from different sockets will never appear together in the qdisc.

VIII. Asynchronous Error Reporting

Because Linux accomplishes IP communications with a mixture of user and kernel threads and the network stack follows a layered architecture, errors may be detected after a system call on a socket returns. A future system call on the socket will report the error instead. Thus, the error reported by a given socket operation may relate to a past operation involving an earlier packet. There is no mechanism to identify which operation and/or packet was involved in the error. In cases where the error would be expected to persist, one cannot be certain how many interim operations and packets have been affected. This uncertainty occurs, in part, because the error reported by the function call is only the most recent identified for the socket; additional errors detected between socket operations are lost. Nevertheless, sockets may also be configured to store errors in a separate receive queue that can be read using the recv() system call; this would prevent such a loss of detected errors. The error queue could also be useful for time-critical detection and recovery of some communication faults since the application can poll the error queue. Nevertheless, due to its asynchronous design, IP error reporting is not well suited for detection of faults that require a time-critical detection and recovery. Applications requiring time-critical fault detection and recovery may need to implement custom tests or use a different solution than the Linux network stack.

A. Connected sockets and the SIGPIPE signal

One exception to asynchronous error reporting is the shutdown of a connected socket, i.e., a TCP/IP socket. When this socket has been shutdown locally or by the remote end, Linux will issue a SIGPIPE signal when the application attempts a send operation. If the process does not handle this signal, it will be terminated. If the real-time application wishes to limit interruption by SIGPIPE, then the application must either set the MSG_NOSIGNAL flag when performing the send operation or queue the SIGPIPE to a file descriptor using signalfd().

IX. IPv4 Processing on Multiprocessor Systems

The receive path and transmit path of the network stack run in different contexts, potentially on different CPUs in a multiprocessor system. When portions of the network stack run on different CPUs, then some data along the receive or transmit path will need to be refreshed in the data cache of the next CPU to perform IPv4 processing. How Linux manages the receive and transmit paths of the network stack, therefore, affects processing latency of IPv4 communications in the form of CPU cache hops as data travels between application and NIC.

The receive path begins with a NIC interrupt and can end at one of three destinations: 1) the socket receive buffer, 2) the NIC transmit buffer (when it triggers a reply), or 3) within the network stack (when it only changes the state of the network stack or socket). The default configuration of the receive path is to run entirely on the CPU that services the NIC IRQ. Under this configuration, the socket receive buffer will require CPU cache refreshes when the application runs on a different CPU than the one that services the NIC IRQ. However, Linux provides options for parallel processing of receive packets that splits the receive path after the NIC ISR and allows protocol processing to occur on a different CPU. With such options, the protocol processing for a socket's packets can be moved to the applications CPU and the cache refreshes only occur for a per-CPU backlog buffer in the kernel.

The transmit path of the network stack ends at the NIC's transmit queue. However, it can begin either with the send operation of an application or with the receive path of the network stack in response to a receive packet (e.g., an ARP reply or a TCP acknowledgement). The transmit path will run on each CPU executing a socket send operation and may execute on the CPU performing the protocol processing of the receive path. However, most of the memory buffers used in the transmit path are bound to each socket until one reaches the queueing discipline prior to the NIC driver queue. This socket specific data may include the socket send buffer as well as internal kernel buffers. Unless multiple processes on different CPUs use the same socket, the socket's buffers will maintain their CPU cache affinity, but they may lose it when the transmit path is initiated by a receive path executed on a different CPU. In the latter case, portions of the socket data will be read into the receive-path CPU cache, and, at the next send operation, those socket buffers will be refreshed on the application's CPU cache. Lastly, both the queueing discipline and NIC driver queue near the end of the transmit path are shared resources protected by a lock. The associated queue data will be refreshed in the CPU cache if the last access was by a different CPU.

A. Maximizing CPU Cache Efficiency vs. Maximizing CPU Access

The CPU cache efficiency of the network stack is maximized when the same CPU executes both the receive and transmit paths of the stack. The application controls when the transmit path executes. However, the application has little control over execution of the receive path. It is the NIC interrupt that initiates the receive path. Linux can either handle the NIC interrupt on the application's CPU or handle the NIC interrupt on a different CPU but offload protocol processing of the receive path to the application's CPU using features that rely on an inter-processor interrupt (IPI). Moreover, the system may not adequately support the capability to dedicate a receive queue for the application's traffic, and the application's CPU may process receive flows for other tasks. Therefore, placing the receive path on the application's CPU increases contention for the CPU with the following disadvantages:

- The application experiences an increased number of preemptions. This occurs when the kernel treats the receive path with higher priority than the application. The performance degradation of the increased context switches may overwhelm performance gains from improved cache locality of the network stack.
- Increased indeterminacy of application preemptions from the CPU. If the application has time-critical sections, the application may not be able to protect that code from preemption by the receive path depending on kernel version and whether an IRQ interrupt or an IPI initiates the receive path on the CPU.
- Increased latency in receive processing possibly leading to lost packets. If the application has priority over the receive path, then the application is protected from preemptions by the receive path. However, a CPU intensive application may block the receive path for long periods preventing the CPU from timely processing of the receive queue, and the kernel will drop packets if the queue fills up.
- Increased processing time for send operations due to increase in pending softirq from receive operations. As explained in section X, socket send operations execute pending softirqs. Placing the receive path on the same CPU may lead to an increase in pending softirqs from those operations, and the send operation may be hijacked to perform that pending work. This can add undesirable latency to send operations for synchronous exchanges.

Systems running the preemptive kernel may therefore struggle with properly tuning scheduling parameters, the timing of socket operations, and the blocking or sleeping of tasks to assure balanced access to the CPU between the user tasks and the network stack. Nevertheless, balancing CPU access may be less of a concern when Ethernet is primarily or exclusively used for synchronous communications since both read and write actions must complete in designated time windows and little to no traffic should occur at other times. Also, the Linux kernel includes packet steering options that can reduce contention for the CPU by reducing the traffic directed at the CPU while taking a minor CPU cache hit on the receive path. (Packet steering features are discussed in the sections that follow.) The packet steering options use filters to direct different packet flows to different CPUs. In the worst cases, this steering causes a cache refresh of a per-CPU backlog queue prior to protocol processing and maximizes cache coherency for the remainder of the stack. This option can still lead to indeterminate CPU contention from the protocol processing of received packets, but the contention may be lower because the filter should send a subset of received packets to the CPU; ideally, that subset would only be addressed to the application's sockets. Filters, however, may not be fine grained enough to isolate a receive queue for an application, and the filter may still direct some unrelated traffic to the application's CPU. Furthermore, some of the steering options rely on IPIs, and IPIs run in kernel context with no option to block or delay preemption of the application.

To maximize CPU access for the application, it is better for the receive path of the network stack to run on a different CPU from the application. This would normally result in CPU cache refreshes of receive path data each time the application performs a receive operation on the socket. Furthermore, it can also result in CPU cache refreshes of the transmit path data whenever the receive path performs transmit path processing; for example, this scenario is embedded in TCP's acknowledgement processing. However, the need for a CPU cache refresh is also dependent on the processor architecture. In some processor packages, the CPUs (aka cores) in a package share an L2 or L3 processor cache. In these architectures, cache coherency is maintained in the lower level cache(s) when the application and receive path run on different cores within the same package. Nevertheless, when the application needs to maximize CPU access, the application can use the following strategies to minimize loss of cache locality within the network stack:

- Offload all socket processing to a task that runs on the same CPU as the NIC ISR. In this strategy, the application sends and receives socket data via an inter-processor communication mechanism such as shared memory. The network stack maintains cache locality. Moreover, the application is not subject to preemption by the NIC ISR or by softirq processing on either the receive or transmit paths. However, the blocks of data sent and received require a CPU cache refresh with each exchange. Moreover, this solution can add scheduling latency to the socket operations when the socket task sleeps between operations.

- Favor UDP over TCP. TCP's send-acknowledge paradigm can lead to transmit path processing being performed on the receive path, and the receive path processes protocol state used by both paths. UDP should only experience this scenario when it requires an ARP exchange (see section IV), but the ARP exchange can be avoided using a static IP-to-link-layer map or by maintaining the liveliness of the ARP cache entry for the network path.

In the end, a system may require a balance between maximizing CPU cache locality of the network stack and maximizing CPU access for the application. It depends on whether the application takes a larger performance hit from CPU cache hops when some or all of the receive path is on a different CPU or from context switches that occur due to interrupt servicing in response to incoming traffic. Furthermore, an application may see an advantage to maximizing CPU cache efficiency for sockets performing synchronous communications but would offload some or all of the receive processing for other sockets onto a different CPU. The packet steering features described below are imperfect methods that can facilitate selective offloading of receive path processing.

B. Packet Steering Features

Many modern NICs support multiple receive queues that can be serviced by different CPUs. Additionally, Linux also provides features that can distribute packet processing of a receive queue over multiple CPUs; the Linux features can be used on NICs with one receive queue as well as those with multiple receive queues. These features for parallel processing of packets can decrease or improve the CPU cache locality of the network stack depending on how the features and the real-time applications are configured. Lastly, for NICs that support multiple transmit queues, Linux provides a mechanism to assign a transmit queue to the application's CPU.

1. *Receive Side Scaling (RSS)*

RSS is a hardware feature resident on NICs that provide multiple receive queues. When the platform supports programmable or static CPU affinity for interrupts, the NIC driver often establishes a receiver queue for each CPU up to the maximum receive queues supported by the hardware. The hardware then distributes received packets among the queues using a hash generated from the packet's protocol headers. Using the hash to distribute the packets preserves the receive order of packets in a flow (e.g., addressed to a port to which a socket is bound) by assuring that all those packets are placed in the same receive queue. Each receive queue also receives its own IRQ and the network stack largely treats each queue like an independent NIC. Thus, the system designer can assign a receive queue to a CPU using the platform's features for assigning IRQs to CPUs (see section IX.C). For systems that support RSS, the NIC driver often enables it by default.

Most NIC hardware implements a hash-to-queue mapping using a table of keys generated from the lower bits of the hash value (i.e., a modulo function). Using the lower 7 bits of the hash to generate 128 keys is common. The keys are then evenly distributed across the receive queues. The system designer could figure out which receive queue(s) will be associated with the application's sockets and assign the associated IRQs to the application's CPU. However, the application's CPU will also process other packet flows whose hash values are associated with keys assigned to the same receive queue. The NIC hardware may provide programmable assignment of keys to queues. This gives the system designer some additionally flexibility in assigning an application's socket flows into one or more receive queues and can reduce but may not eliminate other socket flows being sent to the same queues. Overall, RSS was primarily designed to load balance packet processing, and its limited flexibility may be insufficient to dedicate and isolate a receive queue to the application's socket flows.

2. *Receive Packet Steering (RPS)*

Introduced in Linux kernel 2.6.35, RPS provides a software implementation similar to RSS. The main differences is that Linux applies RPS to a receive queue and RPS divides packets into per-CPU backlog queues. RPS also provides programmable filters for generating hashes or, if supported by the NIC, can use hashes generated by the NIC. For a given receive queue, the user configures RPS with an affinity mask identifying the CPUs, to which RPS can distribute packets. RPS uses a simple modulo of the hash value against the number of assigned CPUs to determine which CPU will process the packet. As with RSS, the hash will be the same for all packets in a flow (e.g., destined for a given socket) and will maintain receive order of packets in that flow. However, the simple CPU selection mechanism makes it difficult to use RPS to target all flows associated with an application to that application's CPU. It is primarily designed to divide the processing load across CPUs. Though RPS decides which CPU will process a packet, it does not define which CPU will execute the NIC ISR of the receive queue. Consequently, configuring RPS to route packets to CPUs other than the one servicing the NIC IRQ, will result in a

CPU cache hop for the per-CPU backlog queue. In fact, it is possible to configure two CPU hops in the receive path when using RPS, one from NIC ISR to protocol processing and one from protocol processing to the application receive call. RPS also utilizes inter-processor interrupts (IPIs) to signal the target CPU that new packets are available for processing. An IPI will immediately interrupt the running application whenever packets are directed to the backlog queue of its CPU. RPS is part of the default kernel build for multiprocessor configurations but is disabled by default. RPS is enabled when a CPU affinity mask is set for a receive queue in the file:

```
/sys/class/net/<dev>/queues/rx-<n>/rps_cpus
```

3. *Receive Flow Steering (RFS)*

Introduced in Linux kernel 2.6.35, RFS overcomes the limitations of RPS by directing packet flows to the CPU that executes the consuming application. In fact, RFS will chase the consuming application if the load balancer migrates the application to another CPU; however, a properly configured real-time system will often lock real-time applications to a CPU. Like RPS, RFS operates above the NIC receive queue and places packets on a CPU's backlog queue. However, RFS maintains a flow table where a key generated from the RPS hash is paired with a CPU. This makes RFS mapping similar to RSS mapping except that the RFS table can be very large to reduce key sharing among hash values. In fact, kernel documentation for RFS recommends setting the table size to 32K entries for moderately loaded servers. RFS is able to identify and maintain the CPU associated with a consuming application because the kernel updates the RFS flow table each time the application performs a socket send or receive operation. Thus, RFS adds a small amount of overhead to send and receive operations including a potential CPU cache hop for the global RFS flow table. If a packet read from the receive queue does not map into the global flow table (i.e., no consumer application has been identified for it), then the packet is routed using RPS.

RFS is designed to maintain packet receive order when the application migrates to a new CPU. Therefore, RFS cannot simply start routing packets to the new CPU since packets may still be processing on the prior CPU. Thus, RFS maintains a second flow routing table for each receive queue that identifies the last CPU, to which the flow was sent, and a counter that essentially identifies the position within the backlog queue of the last packet in the flow (since multiple flows may be steered to the same CPU). RFS will continue to use the prior CPU until all packets in the flow are emptied from the backlog queue. Again, properly configured real-time systems should lock the real-time application to a CPU so this migration should not occur.

RFS can be advantageous to real-time applications because it automatically attempts to maintain cache locality of the receive path with the application. Moreover, it can significantly reduce the processing of other traffic on the application CPU especially with large flow tables. However, it still relies on an IPI to signal new data in the CPU backlog queue, and the IPI will interrupt the real-time application whenever new data is received, which may be undesirable.

RFS is compiled into the kernel by default for multiprocessor systems. However, it is disabled by default. It is enabled when the system configures both the number of entries in the global flow table and the number of entries in one or more flow tables for the receive queues. The number of entries for the global flow table is set in the file:

```
/proc/sys/net/core/rps_sock_flow_entries
```

The number of entries for the flow table of a receive queue is set in the file:

```
/sys/class/net/<dev>/queues/rx-<n>/rps_flow_cnt
```

4. *Accelerated RFS*

Some NICs support hardware acceleration of RFS by permitting the operating system to automatically direct packet flows to the receive queue associated with the CPU of the consuming application (or a neighboring CPU that shares its L2 or L3 cache). Every time the global flow table is updated, the network stack queries the NIC driver for the receive queue that is assigned to the desired CPU of the updated flow entry. The network stack then sends a request to the NIC driver to reprogram the NIC to direct the flow to that receive queue.

The advantage of Accelerated RFS is that it automatically maintains cache locality of the receive path starting with the receive queue through to the socket receive buffer. However, the NIC may not isolate the receive queue for RFS directed flows. The NIC may still direct some traffic not associated with a RFS recognized flow to the same receive queue. It also may not be able to maintain the same number of differentiated flows as software-only RFS, which has the potential to lead to additional unrelated traffic on the receive queue or migrating some of the application's flows to other CPUs. However, Accelerated RFS may be advantageous for embedded real-time systems that use a low number of flows. Another advantage to real-time systems is that accelerated RFS initiates protocol processing from a NIC IRQ. Under the preemptible Linux kernel, IRQs are scheduled threads, and scheduling priority can be used to balance access to the CPU between the application(s) and the receive queue(s). By contrast, software RFS initiates protocol processing using an IPI which leaves no option to block or delay its interruption of the application.

Accelerated RFS requires a kernel compiled with CONFIG_RFS_ACCEL set, a NIC that supports accelerated RFS, and enabling ntuple filtering using ethtool.

5. *Transmit Packet Steering (XPS)*

First introduced in kernel 2.6.38, XPS allows the system designer to assign one or more CPUs to each NIC transmit queue. The network stack's transmit path will then give preference to a transmit queue assigned to the running CPU, if any. If more than one queue is assigned to the CPU, then XPS will use a hash algorithm similar to RPS to select one of the transmit queues for each packet flow. Like RFS, XPS does try to maintain the order of the packet transmission. If an application migrates to a new CPU, transmitted packets may continue to be routed to the last selected transmission queue until no outstanding packets are left in the flow. However, this should not normally be an issue for RT applications regardless of protocol used because RT applications should be locked to a CPU. Cache locality of the transmit queue is maximized when each CPU is assigned a unique transmit queue.

XPS is compiled into the kernel by default for SMP systems; however, it is disabled by default. To enable XPS, set a CPU affinity mask in the following files:

```
/sys/class/net/<dev>/queues/tx-<n>/xps_cpus
```

Unlike RFS, there is no downside to assigning a transmit queue to the CPU of the RT application since it is the application on the CPU that determines when the transmit path is traversed on that CPU.

C. IRQ Balancing and IP Communications

On multiprocessor x86 platforms, the Linux kernel will, by default, balance IRQ interrupts across processors. This includes the IRQs for each receive queue of a NIC. For real-time IP communications, it can improve performance to set a processor affinity for the IRQs associated with Ethernet interfaces. As stated in section IX.A setting the IRQ of a receive queue to run on the CPU of the application can improve the cache locality of the network stack. However, setting the IRQs for a CPU other than the application's CPU improves the application's uninterrupted access to its CPU and may produce better overall performance when the receive queue receives frequent bursts of traffic. On systems where CPU cores share an L2 or L3 cache, setting the IRQ of receive queues to run on a neighboring CPU in the same processor package can achieve both objectives to some extent.

How NIC IRQs are distributed to cores on non-x86 platforms depends on the board support package or firmware for that platform. IRQs may be fixed to CPU0. The firmware may allow static assignment of IRQs to specific processors, or the board support package may enable IRQ balancing.

X. SoftIRQ and Socket Send()

A send operation will attempt to traverse the network stack down to the driver queue within the context of the send() call. However, if there is not enough room in the driver queue for the full output block, then the send() call will separately queue what does not fit and raise a softirq to transmit that queued data later. The send() call will also raise a softirq if the device queue is locked because another process is also sending data to the NIC. When the kernel later processes the softirq is not simple to determine. First, it depends on the kernel version, especially when using the CONFIG_PREEMPT_RT patch, and can therefore change as the application is migrated to future kernels. It also depends on other kernel activity that process softirqs. One of those activities is the send() call, so a subsequent send() call, even on a different socket, can process the remaining packets. If the CONFIG_PREEMPT_RT patch models the softirq as a thread, then execution of the softirq may also depend on the scheduling priority of the application, and it may be possible for the application to block the softirq for extended periods. Otherwise, the softirq may also interrupt the application at an inopportune time. Generally, conditions for softirq processing make it unusual for a softirq to remain pending for very long. However, long delays can occur under heavy loads.

Thus, a softirq can add uncertainty to the completion of a data block transmission in one of two ways. The first is when a softirq is necessary to complete the transmission at a later point as explained in the prior paragraph. Synchronous communication, in particular, may need to avoid raising a softirq during a send operation. This requires assurance that the data size and rate of send operations do not lead the stack to exceed the capacity of the ring buffer. How much data can be sent in a single call or in the time prior to a synchronous data call without raising a softirq? It depends on the length of the driver queue, the path MTUs, and the features and speed of the NIC. Based on the kernel code, Linux uses a driver queue with 64 entries. Some NIC features like segmentation offload (see section XI.G) allow the entry to represent more than one packet. However, a simple calculation assumes that each entry represents a packet whose length is equal to the path MTU and that the NIC empties the driver queue at the rate that it can transmit the data on the wire (accounting for the full length of the resulting Ethernet frame). For example, with a 100 Mbit Ethernet connection and a 1500 byte MTU (a 1542 byte Ethernet frame), a send call should avoid raising a softirq if it does not transmit a block larger than 92,762 bytes through a TCP socket (64 packets with 1448 user bytes per packet).[**] In another example, if the synchronous send block is 34752 bytes (24 packets), then the application should not send more than 57920 bytes (40 packets) in the prior 5 milliseconds. Note that these same guidelines for data sizes and spacing in send operations should also cause the network stack to maintain temporal ordering of the data on transmission.

The second way softirq can induce delay is due to the fact that the send operation processes softirqs. The send operation will process softirqs when the call succeeds in reaching the IP layer where it creates packets ready to queue. Here, the send operation will process all pending softirqs not just the network transmission softirq because the Linux kernel does not provide a mechanism to execute individual softirq entries. As Thomas Gliexner observes, the softirq is a "*conglomerate of mostly unrelated jobs, which run in the context of a randomly chosen victim w/o the ability to put any control on them*" [5]. In other words, the send() call can cause the real-time application to complete the work of other tasks. Although there are no limits on the content of this deferred work, softirqs have restrictions and characteristics that lower the probability of hijacking the application. First, the pending softirqs must have been raised on the same CPU. Second, softirq tasks tend to be compact, and the tasks cannot sleep. Third, most kernel activities that can raise a softirq also process softirqs including interrupt service routines (ISRs); this makes a large build-up of pending work less likely. Nevertheless, it remains possible, if improbable, that enough pending work will accumulate in the softirq mechanism to disrupt the real-time application and delay transmission of the output block when the application calls send(). To further reduce or eliminate the possibility of large softirq workloads, the real-time application should run on a CPU that services few or no interrupts, and the application should not share the CPU with other tasks performing writes to high volume devices or setting timers, both of which can raise softirqs. The one debatable exception to this is placing the application on the same CPU servicing the NIC IRQs used for real-time communication as covered in section IX.

XI. Adapting TCP/IP for Real-Time Communications

The overriding design criterion for TCP/IP is reliable, in-sequence reception. TCP/IP will sacrifice end-to-end latency and throughput for this criterion. In its default configuration, the upper bound on TCP latency is measured in tens of minutes though such an extreme only occurs when large numbers of sequential packets are lost forcing timeouts and packet retransmissions. The typical latency will often be orders of magnitude lower. On closed networks, the potential for lost packets is significantly reduced and is further reduced when the system design guarantees that nominal and off-nominal traffic cannot induce network congestion. However, TCP/IP is designed to induce packet loss as a method of detecting the congestion limit along a path; therefore, real-time programs may need to design TCP/IP transactions to suppress this behavior if low latency is desired. Furthermore, packet losses may still occur as a result of a single event upset or failing hardware. This paper looks at the simplest case of losing a single packet to illustrate the difficulty (and potential impossibility) of configuring TCP for low-latency detection and recovery of packet loss.

A. Caveats

TCP/IP is defined by numerous Requests for Comments (RFCs) issued by the Internet Engineering Task Force. Some RFCs are normative, some are optional extensions to TCP/IP, and some are experimental. Linux kernels include some optional and experimental extensions and enables some of them by default. Many of these are documented in the man pages for TCP. However, some can only be discovered by reviewing kernel documentation, forum discussions on the kernel, or reviewing the kernel code. Furthermore, Linux has some of its own tweaks to TCP that may deviate from the RFC description and these are often discovered, again, through review of kernel documentation, forum discussions on the kernel, or the kernel code. Also, recognize that the RFCs describe behavior in the context of a large continuous data stream. That behavior often manifests differently for periodic discrete bursts or so-called thin TCP/IP transmissions where the number of bytes outstanding on the network never exceeds four packets (aka segments). The discussion here is based on the 3.2.76 kernel and was informed by selected searches and skimming of the kernel code to answer some simple questions in an attempt to fill knowledge gaps. An

[**] Note that it may be impossible for the network stack to send this much data in the context of the send call when using TCP because TCP limits the amount of data outstanding on the network before it receives an acknowledge packet from the receiver. The remaining data will be transmitted in the context of the network receive softirq that processes the incoming acknowledge packet. UDP is restricted from sending this much data in one call. Therefore, for a send call to raise a softirq may require other close-proximity writes to the driver queue.

exhaustive review of the kernel code or runtime experimentation will be necessary to verify the TCP logic described here for other kernel versions.

B. When is TCP/IP recommended for real-time communications?

Use of TCP/IP for real-time communications is recommended when the data stream is required to be preserved in its entirety even if that causes the data to be delayed for more than 120 milliseconds (see section XI.I). An example of this could be the writing of telemetry data to an onboard storage system. The latency that TCP/IP can add to communications, under either lossless conditions or single-packet loss, is dependent on the options set for the socket, the size of the receive buffer (which relates to the advertised receive window), the amount and timing of data sent in an execution frame, the Linux kernel version, and the features of the other TCP nodes (which may not run Linux). Fine-tuning TCP for low latency communication often requires addressing each of these dependencies and can be brittle to changes in any of those dependencies. Such tuning may also be impossible if tolerance against multiple packet loss is required. Additionally, the complexity of TCP carries with it computational overhead that can outweigh what benefits a specific use may gain from TCP.

It is often tempting to use TCP/IP to send data blocks larger than a MTU or when automated recovery of packet loss is desirable to curtail the occurrence of stale data. However, for the real-time communication types presented in section II.A, where late data is invalid and dropped, a real-time application is often better off using UDP/IP. Even when recovery from packet loss is desired, the real-time application is often better off implementing a custom loss detection and retransmit method on top of UDP/IP. For these types of transfers, UDP/IP provides better guarantees of end-to-end latency for lossless and packet loss scenarios. Moreover, the OS and application are not forced to wait for invalid stale data before fresh data can be read. In part to substantiate these claims about TCP/IP, this section details how TCP/IP inserts latency into its communications.

C. A quick overview of TCP/IP exchanges and latency

TCP/IP is a send-acknowledge protocol whereby the amount of data the sender can have outstanding (i.e., not yet acknowledged) on the network is capped. Once the cap is reached, new data cannot be sent until prior data has been acknowledged. This cap is called the congestion window because it is dynamically adjusted to maximize transmission rate while minimizing packet loss (which is presumed to be due to network congestion). Furthermore, the sender is also restricted to sending no more data that the receiver tells the sender that it can receive; this is called the receiver window, and it is the number of bytes that remain free in the receiver's receive buffer. The congestion and receiver windows can be managed in units of bytes or segments (a.k.a. packets). When measured in segments, the RFCs often assume those segments are full-sized, i.e., fully populated for the given path MTU. Linux tracks and manages the windows using units of segments. Therefore, this paper describes windows in units of segments.

Under lossless data exchanges, latency in TCP/IP is introduced:

- When the TCP stack delays placing data into a TCP/IP packet
- When the outstanding packets on the network reach the congestion window limit
- When the receiver delays sending acknowledgements
- When the receiver buffer nears capacity

Thus, latency is minimized when

- Features that delay placing data into a TCP/IP packet are disabled (or worked around)
- The rate of transmission and the rate of acknowledgement maintains an equilibrium of outstanding packets that is below the congestion window.
- The rate of transmission and the rate of consumption maintains an equilibrium of free space in the receive buffer that is above the equilibrium of outstanding packets.

1. Maximum Segment Size (MSS) and Segment Counting

When the RFCs discuss TCP behavior in units of segments, the RFCs assume those segments are full-sized. The size of the data payload that constitutes a full-size segment is called the maximum segment size (MSS). A further complication to determining size of a 'full-segment' is that TCP/IP is designed to dynamically determine MSS for both send and receive paths since it differs across the internet. The MSS of the send path is normally determined through Path MTU Discovery (see section III) or, if Path MTU discovery is disabled, the send path MSS is set to the device MTU of the local NIC minus the bytes for the TCP/IP headers and timestamp. The receive path MSS is set based on the larger of 524 bytes[††] or the largest packet received. Thus, it is possible for the receive path to estimate a

[††] This is the minimum MTU of 576 defined by the Ethernet standards minus protocol headers and timestamp.

lower MSS than the path MTU if the destination node never replies with a full-sized segment. For closed networks, the MSS should be 1448 bytes when using standard Ethernet frames and 8948 bytes when using jumbo frames.

By default, Linux implements TCP/IP logic by counting segments (whether or not they are full-sized). However, Linux provides the 'appropriate byte count' (ABC) option to base some TCP logic on byte counts. For reasons that will be highlighted in later sections, the default behavior is preferred for real-time applications. Therefore, the remaining sections will discuss behavior in units of segments rather than bytes, except where Linux uses byte counting regardless of ABC setting.

2. Round Trip Time (RTT), Buffer Sizing, and Latency

Another important metric in TCP behavior is round trip time (RTT). RTT is the time between transmitting a segment and receiving its acknowledgement. When discussing TCP behaviors based on RTT, the RFCs assume RTT is defined in the context of a continuous stream of data. Bursty or thin data streams may inflate the measured RTT since the receiver may, under default TCP behaviors, delay acknowledgement of one or more packets. Even for continuous data streams, RTT does exhibit some variation. Where latency of an operation is a function of RTT, the real-time system should assume the worst case RTT for the operation.

For real-time applications, latency is minimized when all the data in an execution frame can be transmitted within one RTT. This occurs when the congestion window and receive window allow all the data in an execution frame to be transmitted before an acknowledgement is received. This requires sizing the receive window and priming the congestion window so that both are at least as large the data transmitted in the frame. The receive window is equal to the available space left in the receive buffer. Thus, the receive buffer must large enough to hold worst-case amount of data that can be buffered between read operations plus the worst-case amount of data transmitted each frame. The congestion window must grow to become the amount of data to be transmitted each frame, at a minimum (see section XI.H). However, to maintain real-time communication under loss of a single packet, the congestion window should grow to twice the amount of data transmitted each frame (see section XI.I.2).

D. Kernel Parameter tcp_low_latency

To improve bandwidth, Linux implements a prequeue for incoming data. This prequeue allows the network softirq to quickly offload the incoming data and be ready to accept new data; it also offloads some TCP processing onto the application. The tcp_low_latency parameter disables the prequeue and forces the softirq to process data into the TCP receive queue. (Unless the socket is locked, then the softirq places the incoming data onto the TCP backlog queue and the application completes the protocol processing.) The tcp_low_latency parameter is a system wide setting that affects all TCP sockets. Whether and by how much tcp_low_latency reduces latency depends on the system design (both hardware and software) and the network traffic. Enabling tcp_low_latency can make sense in multiprocessor systems where the real-time application and the softirq operate on different processors. In this situation, enabling tcp_low_latency shifts computation for TCP/IP receive operations onto the softirq freeing more of the CPU for the application. However, if TCP/IP is being used to emulate synchronous communication, then enabling tcp_low_latency may make only a miniscule difference in the time from packet arrival at the network interface to receive operation at the application. The tcp_low_latency setting basically eliminates extra data copies to/from the prequeue. The savings may only be on the order of nanoseconds to microseconds depending on the computing architecture.

E. TCP_NODELAY

By default on Linux, TCP uses the Nagle algorithm that delays transmission of partial frames when the host is waiting for an acknowledge (ACK) from the destination. This allows any additional data to accumulate in the interim and reduces the number of Ethernet frames transmitted when an application performs many small send operations. Instead the application should use data design (see section II), TCP cork (see section II.D), or a writev() call to perform its own data consolidation that reduces the Ethernet frames transmitted. Even when an application sends a large data buffer, the Nagle algorithm will delay the transmission of that last frame if it is a partial frame. For real-time applications, the Nagle algorithm is undesirable because the output block may be subject to the latency of receiving the next acknowledge. The receiver normally delays sending the ACK packet unless delayed ACK is disabled (see section XI.F); then delays are based on round trip time (see section XI.C.2). The Nagle algorithm is disabled by setting the TCP_NODELAY socket option. The TCP_NODELAY option may also be necessary when using brute force methods for time-critical packet loss detection and recovery (see section XI.I).

F. Delayed Acknowledgement

RFC 1122 recommends that TCP implementations include delayed acknowledgement, and Linux performs this function. Under delayed acknowledgement, the receiver does not immediately acknowledge each packet received.

Instead, the receiver waits either for a specified delay or for every two segments in a stream of full-sized segments. The RFC allows a delay of up to 500 milliseconds. However, Linux implements a delay of 40 milliseconds that is not modifiable (except on the Red Hat Enterprise distribution where it is exposed as a kernel parameter). There are two reasons for the delay. First, a delay allows the receiver to consolidate acknowledgement of multiple small packets into one ACK packet. Second, it provides time for the receiver to piggyback data that the receiving application sends in response. Overall, it reduces the number of packets exchanged, improving network efficiency at the cost of latency. However, the Linux default and the default that may exist on non-Linux receivers can interfere with attempts to exchange data at the execution rates typical of real-time aerospace software.

Linux provides a nonportable socket option TCP_QUICKACK to disable delayed acknowledgement. However, this option is not permanent, and the socket can revert to delayed acknowledgement on "subsequent operation of the TCP protocol… depending on internal protocol processing and factors such as delayed ACK timeouts occurring and data transfer" [6]. Thus, to maintain TCP_QUICKACK, the receiving application must reset it after every TCP operation on the socket. Even then, the language of the man page is ambiguous enough to allow reversion to delayed acknowledgement during kernel processing of TCP packets that occurs between the application's operations on the socket. To identify all the circumstances, under which Linux will revert to delayed acknowledge, requires thorough review of the TCP source code. Furthermore, a feature similar to TCP_QUICKACK may not be available on non-Linux receivers. For these reasons, this paper does not carry the TCP_QUICACK solution into later sections.

One can also use brute force work arounds to prevent ACK delays. The application can pad the output block so that the data fits in an even number of full-sized segments. (Linux uses byte counting rather than segment counting for delayed acknowledgement, even when 'appropriate byte count' is disabled.) If the system also needs to avoid the extra latency of transmitting padded information, then the sender application can set the maximum segment size (MSS) using the socket option TCP_MAXSEG to a size that is an even integer divisor of the output block. This option must be set before the connection is established to take effect. However, this option also has the side effect of capping the size of response traffic from the receiver because the kernel will also advertise this number as the maximum segment size it can receive. Modifying MSS is most useful when the same amount of data is periodically transmitted and the receiver sends less data per frame. Unlike TCP_QUICKACK, these workarounds are portable and do not require continued action by the application to maintain.

G. TCP Segmentation Offload

Many network interface cards (NICs) manufactured in the last few years are able to offload the TCP segmentation of the send buffer from the CPU. This feature is called TCP Segmentation Offload (TSO). Linux tries to take advantage of TSO when it can and will occasionally defer data transmission in an attempt to place a larger send buffer onto the NIC. The worst case for the delay is two jiffies (clock ticks). On PowerPC and x86 platforms, a jiffy is equal to one millisecond. Therefore, TSO adds up to two milliseconds of latency when a TSO deferral occurs. However, this deferral occurs only when the amount of data the TCP stack is allowed to transmit falls below a threshold. (Note this is not the amount of data in the send buffer but the remaining data size that the TCP stack can send before receiving the next acknowledge!) The threshold is a fraction of the lesser of the congestion window or receive window; the fraction, by default, is one-third but is configurable. If the fraction is set to zero, then the threshold is equal to the kernel parameter tcp_reordering multiplied by the current MSS. By default, tcp_reordering is three resulting in a threshold of three full-sized segments. For synchronous and periodic asynchronous transfers, it is therefore possible to tune the receive window, the congestion window, or the divisor so that the deferral should never occur. However, in those cases where the platform supports TSO, it is simpler to disable the feature.

H. Slow Start and Hystart

In TCP, an algorithm called slow start first manages the size of the congestion window. Under slow start, the sender initializes the congestion window (CWND) with a defined value. Then, the sender quickly raises that window as it receives acknowledgements until packet loss occurs (which is treated as congestion). Once the congestion window exceeds the receiver advertised window, then transmission rates are governed by the receiver window but the congestion window may continue to grow. Under Linux, the initial value of the congestion window is hard coded. Before Linux Kernel 2.6.39, the initial value followed the RFC 3390 recommendation of two to four segments targeting an initial total of 4464 unacknowledged bytes (3 MSS with standard Ethernet frames), though under Jumbo Frames this could be as high as 17896 bytes (two MSS). Starting with Linux Kernel 2.6.39, the initial value was raised to 10 segments based on empirical evaluations performed by Google [7]. Each successful ACK received in sequence will raise the congestion window by a segment. The slow start algorithm produces exponential growth of the congestion window in each RTT when transmitting a large continuous data stream since the sender would receive CWND of ACK packets each RTT if every sent segment were acknowledged. Some Linux behaviors cause the congestion window to grow more slowly than the ideal exponential rate. Delayed acknowledgement (see section XI.F) causes the congestion window to grow one segment for every two segments acknowledged. Linux also does not increase the congestion window when the difference between the congestion window and inflight segments does not exceed the reordering threshold (three segments by default) or, if TSO is enabled (see section XI.G), when that difference is not large enough to use TSO. For example, if one sends x amount of data each frame with TSO enabled, then the congestion window will stop growing when it reaches 1.5 x (with the default TSO divisor of 3).

Slow start can become an issue if the application is designed to send more than ten segments in a real-time frame and does not start sending data until the application is in a real-time mode. The initial data exchanges will be subject to latency since two or more RTTs will be needed to transmit the complete output block. Therefore, to avoid increased latency in send operations for the first few frames, the application must prime the condition window. This can be done by sending enough data during startup that the congestion window (CWND) grows to the desired size. (Note that section XI.I.2 discusses setting the target CWND to larger than the data sent per execution frame.) However, the Linux slow start logic makes this difficult. For example, one empirically derived strategy is to make 2*CWND send() calls of MSS bytes every two RTTs until CWND reaches the desired size; this strategy also works best when the receive buffer is equal to or greater than the target CWND. A more efficient strategy to set the congestion window is to create an entry in the route table for the connection that sets the congestion window to the desired size before launching the application. Before entering real-time operation, the application can check the size of the congestion using the socket option TCP_INFO; however, this mechanism is not portable. Nevertheless, verifying the congestion window before entering operation is advisable since a lost packet occurrence during initialization will modify the congestion window. In fact, per RFC 5681, the initial CWND must start at one segment if a packet is lost during the initial three-way handshake that establishes the TCP socket connection.

Once the congestion window exceeds the slow-start threshold (ssthresh), then Linux uses a congestion control algorithm to manage the congestion window. Congestion control algorithms are discussed in section XI.I.2. The congestion control algorithm also controls the value of ssthresh. Linux initializes ssthresh to an "arbitrarily high value" (per RFC 5681) of 2147483647 segments unless an entry exists for the route in the route table; then the cached ssthresh value is used. RFC 5681 describes a congestion control algorithm nicknamed Reno that does not change ssthresh until packet loss occurs. Then ssthresh is the set to half the number of segments in-flight; the Linux implementation of Reno actually sets ssthresh to half the current congestion window. (These are equivalent when sending a large continuous data stream but would differ for burst transmissions or thin data streams.) In either case, this forces Linux to stop slow start and start using the congestion algorithm.

However, starting in Linux 3.2, the default congestion algorithm changed to CUBIC. CUBIC implements a modification to the slow start algorithm called Hystart. Hystart differs from slow start in that, while in slow start, it uses the arrival time of acknowledgements to estimate the throughput of the route and changes ssthresh based on that estimate. Thus, Hystart will operate in slow start while it collects data and, after it makes its estimate, until the congestion window meets or exceeds the computed ssthresh. Linux limits time measurements for CUBIC to a resolution of one millisecond. Therefore, Hystart estimates of throughput are most accurate when the application sends enough data to keep the network busy for at least one millisecond. Even then, the first estimate is likely to be a fraction of actual throughput. Empirical testing using large TCP data streams on a gigabit network resulted in initial ssthresh estimates in the neighborhood of 23 segments.

When Linux changes to a congestion control algorithm, the growth of the congestion window can slow dramatically. Many congestion control algorithms use a linear increase, i.e., increase CWND by one segment for each round-trip time (RTT) without packet loss, at least for some initial period. Thus, if the target congestion window to minimize latency for lossless and single-packet loss scenarios, is substantially higher than the initial ssthresh computed by Hystart, then an application may want to configure the TCP socket to use 'reno' or other congestion algorithm that maintains slow start for a longer period. In fact, one can write a custom congestion control algorithm that implements a custom slow-start algorithm that immediately or more quickly climbs to the desired congestion window. Then, the TCP socket could be set to use that algorithm. However, this requires modifying the kernel. Nevertheless, Hystart is not a concern if one sets the initial congestion window to the desired size using the route cache.

Linux will return to slow start mode if the socket is left idle for the retransmission time-out (RTO) period. As explained in section XI.F, the RTO is at least one second. If the real-time application uses TCP/IP to transmit aperiodic data that has low latency requirements and the socket can be left idle for more than the RTO, then the operating system should be configured to disable slow start after idle using the kernel parameter, tcp_slow_start_after_idle.

I. Single Packet Loss and Latency

Loss of a single packet is a simple off-nominal case that highlights how TCP retransmission and recovery injects latency following the packet loss. It also provides enough of a foundation to anticipate the increasing difficulty in capping latency should a typical loss scenario involve multiple packets. Recall that the loss of a packet inherently induces latency because TCP preserves the stream in order and the TCP stack will not provide any data that follows the lost packet to the application until the lost packet has been successfully retransmitted.

1. Latency of Detecting Single Packet Loss

TCP detects a lost packet either as a result of a retransmission timeout (RTO) or when triggered by fast retransmit. The RTO is a function of round-trip transit time (RTT) but cannot be less than 1 second per RFC 6298. The minimum RTO is too large to support most types of real-time communications. Thus, real-time communication with TCP remains possible only with fast retransmit. Under fast retransmit, the receiver will send an acknowledgement (ACK) in response to out-of-order packets, subject to delayed acknowledgement (see section XI.F). When packets remain out-of-sequence, then the sequence number in the ACK remains unchanged. When the sender receives three duplicate ACK packets, it will assume packet loss and retransmit the packets in the retransmission queue. This is somewhat inefficient since the duplicate ACK responses indicate that the receiver has received some of the packets in the retransmission queue. Another TCP feature, selective acknowledgement, allows the recipient to inform the sender, as part of the duplicate ACK, of those packets that were successfully received that are not the next packet in the sequence. This allows the sender to identify packets that do not need retransmission when retransmission occurs. Linux implements both fast retransmission and selective acknowledgement.

Fast retransmit establishes the expected latency of a retransmit given the scheduling of send operations, their data lengths, and delayed acknowledgement behavior. For example, if a TCP socket is used to send a single segment each execution frame, then a lost packet will not be detected for the lesser of 7 execution frames or 120 milliseconds under the Linux implementation of delayed acknowledgement. At least seven full-sized segments must be transmitted each frame to reduce the latency of packet loss detection to one frame. (This covers the worst case of losing the last segment sent in a frame). To enable detection within the same frame as the transmission, one must employ the brute force technique of sending additional packets of padding data. In the worst case, where the last packet of real data is lost, enough padding packets must be sent to trigger three duplicate acknowledgements. With delayed acknowledgement, six full-sized packets of padding data would be required. If other tuning was performed to assure both real and padding data would normally be transmitted in one RTT, then the brute force strategy will bound the retransmission latency of the real data to approximately one RTT plus the transmission time of the pad packets over the hardware. Therefore, a typical latency between modern computers over a gigabit network should be O(100 μs) and is largely driven by the system response of both nodes to incoming packets. This brute-force technique, of course, sacrifices effective bandwidth for latency. Also, it is important to note that if the lost packet is one of the padding packets, then the loss of the padding packet will not be detected until the following frame and one RTT of latency will occur in that frame before the real data will be made available to the application.

Linux does provide a couple of options that can be employed to reduce the number of padding packets. First, the system can be configured to detect packet loss using a single duplicate acknowledgement by setting the kernel parameter tcp_reordering to one. Note that this is a system-wide rather than per socket configuration. Linux discourages changing this parameter to avoid unnecessary retransmissions and their consequences, but that warning assumes traffic may be routed across multiple local area networks. On a closed network with only switches or a point-to-point connection, reordering should not occur within a TCP stream and this setting should be a safe, quick, and accurate detection of lost packets. With this option, only two full-sized padding packets would be necessary with delayed acknowledgement.

Another option is to set TCP_THIN_DUPACK on the socket. This option will detect packet loss with a single duplicate acknowledgement when the number of outstanding segments on the network is less than four and the socket is not in slow-start. Thus, this option will detect the loss of any segment containing real data within the same frame when only four or fewer total segments of real and padding data are sent in a frame. (In this scenario, losing the first segment is also a boundary case.) Under normal delayed acknowledgement, this restricts the sending of real data to just one segment per frame because a loss of a second segment of real data will not produce a duplicate ACK in the same frame. Instead, when a third padding segment is received, the ACK will increment the sequence counter to cover the received first segment, and it will not be a duplicate ACK. Upon receiving a fourth segment of padding data, the receiver will still require one more segment before it will issue an ACK, but that segment will not come until the next frame executes the send operation. In addition, to activate this option, the socket must already be under congestion control and not in slow-start when the send operation begins. For some congestion control algorithms, this only occurs after a packet loss first occurs. Therefore, often it is better to use this option with a congestion control algorithm like CUBIC that enters congestion control based on heuristics. (One could also use cached route entries to initialize the connection in a congestion control state.)

2. Fast Recovery and Congestion Control after Packet Loss

TCP assumes network congestion is the cause of packet loss and will throttle throughput following detection. This could add latency to future transmissions. Immediately following retransmission of the packet, TCP enters fast recovery, which governs how data in the send buffer is processed until the retransmitted packet is acknowledged. Assuming no further loss, the retransmitted packet should be acknowledged in one RTT.

The fast recovery algorithm that Linux performs is called rate halving. Linux will send one segment of previous unsent data each time it receives a duplicate acknowledge, if there is data to send and the receive window allows it. (The congestion window is artificially inflated to always allow the transmission if allowed by the receive window.) Because delayed acknowledgement sends an ACK packet for every two full-sized segments received, the transmission rate is effectively halved using this exchange. Because the RTT should be much smaller than a frame and because real-time sockets should be tuned to send a frame of data within one RTT, the fast recovery algorithm should complete before the next frame of data is sent. Nevertheless, the algorithm could impact data transmitted within the frame if multiple send operations are spread over the course of the frame.

Once the packet is acknowledged, the congestion control algorithm then reduces the congestion window and this could add latency to future transmissions. The Reno congestion control algorithm decreases the congestion window by half and then linearly increases the window by one segment 'per RTT'. However, 'per RTT' assumes a continuous data stream that fills the congestion window. The Linux Kernel actually increases the congestion window by one segment for each set of acknowledgements equal to the congestion window. Therefore, if the congestion window were just large enough to send a frame's worth of data when a packet was lost, the new congestion window would be reduced by half and would be incremented by, at most, one segment per execution frame when delayed acknowledgement is active. Moreover, the Linux Kernel would only send part of the data in the first RTT and the remainder in the second RTT. Thus, the full data set would be delayed by one RTT for the number of frames needed to grow the congestion window back to a full frame of data. The CUBIC congestion control algorithm only reduces the congestion window by 30% and then uses a cubic function to grow the congestion window.

To avoid adding an RTT of latency to communications in the frames that follow packet loss, there are two options. The first is to grow the congestion window during application initialization to be a multiple of the data sent per frame. For example, the congestion window should be grown to twice the data sent per frame when using Reno. This keeps the congestion at or above the data sent per frame as long as a second packet is not lost before the congestion window recovers to twice the data sent per frame. A second option is to write a custom congestion control algorithm that uses the data sent per frame as a lower bound on the congestion window size. However, this requires modifying the kernel.

3. Boundary Conditions for Single Packet Loss

The congestion window continues to grow when no packet loss occurs. Thus, if the mean time between packet losses is long, the congestion window can often grow, sometimes exponentially, to be many times the segments sent per frame. Thus, it is easy to dismiss the need to actively manage the congestion window during initialization. However, flight critical software must also be robust to the possibility that a packet loss will occur at any time, including the first few frames of operation before the congestion window has a chance to become huge. In fact, the three boundary cases for single-packet loss include a) losing a packet during the three-way handshake that establishes the connection, b) losing the first segment of data sent in the first frame of operation, and c) losing the last segment of data sent in the first frame of operation. All three will reduce the congestion window and slow its growth before the window has had a change to grow well beyond the segments sent per frame.

4. Capping Latency with Multiple Packet Losses is a Losing Battle

If the typical loss scenario in spaceflight would result in multiple packet loss, then one can quickly see how capping latency of retransmission within one frame becomes a losing battle. If the lost packets are continuous, then the number of padding packets to brute force loss detection within the frame increases one-to-one by the maximum segment loss expected. If the typical loss scenario is multisegment but not continuous, then brute force of loss detection must add enough padding frames to cover the largest window between the first and last lost segment. In either case, this brute force method will continue to add latency and computational overhead for the transmission and processing of an increasing number of padding packets. With regard to recovery, traditional recovery responds to each lost segment individually so each lost segment adds an RTT of latency to the availability of fresh data. Each lost segment would also reduce the congestion window so it decreases exponentially with each packet lost. This decrease could add multiple RTTs of latency to the next few frames. When the multisegment loss is continuous, the kernel could use the information in selective acknowledgement to treat the group as a single retransmission event. This would reduce the RTTs needed to retransmit all lost packets and would make the post-recovery effects of continuous packet loss similar to single packet loss. However, Linux response to multi-packet loss was not researched for this paper.

5. Why Not Treat Packet Loss as Loss of Communication?

One could argue that, if packet loss is rare enough and if capping latency is so difficult, then why not simply treat packet loss as a loss of communications until the socket fully recovers? Without the brute force techniques, socket recovery can take hundreds of milliseconds or more when transmissions are periodic bursts. However, more importantly, this question posits that the driving characteristic of the TCP protocol is unimportant, i.e., reliable, in-sequence reception of all transmitted data. If the application is simply going to treat read operations unable to return a complete set of data as a loss of communication error, then (as stated in section XI.B) UDP/IP provides much better latency guarantees with lower processing overhead and will not retransmit lost packets that the application is going to ignore as late, invalid data.

J. TCP/IP and Maintaining CPU Cache Locality of the Network Stack

Under TCP, the receive path and transmit path of the network stack are intertwined. If TCP cannot transmit the entire send buffer in the context of the application's send() call, then the remaining data will be transmitted within the context of the receive path when an ACK is received. Even when there is no data pending in the send buffer, the reception of an ACK updates the TCP state utilized by both the receive and transmit paths. Therefore, executing the receive path on a different CPU than the application can degrade performance of the network stack due to an increase in CPU cache misses and the application may experience this performance degradation when it performs a send operation. Network stack performance will improve when the receive path runs on the application's CPU. However, this may introduce other undesirable impacts as detailed in section IX.A including reduced application performance and indeterminate preemption of time-critical code. TCP may, in fact, exacerbate some of these impacts because TCP generates ACK packets proportional to the number of packets sent. Each ACK may then cause a NIC interrupt to occur on the sender.

K. The Illusion of Deterministic Data Communication with TCP/IP

Under nominal conditions, periodic data exchanges over TCP/IP may appear deterministic over a select length of time. The application may show larger data exchange times on the order of one or a few RTTs for the first few exchanges as the congestion window grows to match the amount of data exchanged. However, once the congestion window allows the complete data block to be transmitted in one RTT, the data exchange times will appear to be near constant with some jitter. Therefore, it is easy to assume that determinism is a property of TCP/IP by simply examining exchange times over limited sample lengths. Nevertheless, as should be obvious from the topics above, TCP/IP is not designed for determinism and the apparently deterministic data exchange time can change. For example, in the case where TSO is active, later changes in the congestion window could trigger TSO delays. Determinism is also lost when a packet is lost and future frames will exhibit increases in their data exchange times until the TCP stack reaches a new equilibrium between data transmitted, the congestion window, and the receiver window. In addition, when tuning is taken to assure determinism, any later changes to the exchanged data, the Linux kernel version, or the computing and networking hardware can result in different behavior. Therefore, claims of achieving determinism with TCP/IP require scrutiny.

XII. Conclusion

The Linux network stack is designed for throughput over the non-deterministic Internet. The stack contains numerous compromises, some specified by protocol standards and others implemented by Linux contributors, that increase latency and forgo determinism. Even so, on closed local area networks, options for stack configuration and application design are available to perform real-time communications over IP with a high level of confidence. The following are recommended practices for real-time communication over IP under such dedicated networks:

- Design output data to minimize the Ethernet frames transmitted. Consolidate small output blocks to match the path MTU. Larger output blocks should nearly fill the last packet in the group needed to transmit the block.
- Disable Path MTU Discovery on sockets used for real-time communication
- Establish static IP-to-link layer tables for the network to avoid ARP delays
- Set send and receive buffer sizes for each real-time socket using setsockopt
- Attempt to fully populate send and receive buffers during initialization to force Linux to map all pages of the buffers into virtual memory.

- Configure the sockets to be non-blocking. For synchronous communications, the receiving application can wait for incoming data use a system call for event notification that provides microsecond or nanosecond resolution for its timeout. These include select(), pselect(), ppoll(), and epoll_pwait().
- Space calls to send() in time so that the NIC's transmit queue is clear of the previously sent data when each send() call is made. This will minimize the use of softirq to complete transmissions.
- Use UDP for real-time communications. Using TCP is not recommended. Even when reliable transmission is required, low-latency loss detection and retransmission can be better achieved with custom programming on top of UDP. TCP is too conservative in avoiding spurious retransmission over the open internet to provide low-latency loss detection and retransmission.

The following recommendations apply to multiprocessor systems:

- Limit exposure of send() calls to pending softirq work by isolating the application CPU from other tasks that raise softirqs like device drivers, tasks that set timers, or tasks that use high volume devices.
- If the NIC supports multiple transmit queues, enable transmit packing steering (XPS) and dedicate one NIC transmission queue to the application's CPU
- Enable receive flow steering (RFS) or, if supported by the NIC, enable Accelerated RFS

Additionally, the Linux network stack is not designed to provide synchronous reporting of faults; faults may not be reported until the next socket operation that the application performs. Therefore, an application that requires time-critical recovery of communication faults may need to poll the socket, implement custom tests, or use a product other than the Linux network stack.

XIII. References

[1] Cunningham, K.. Shah, G. H., Hill, M. A., Pickering, B. P., Litt, J.S., and Norin. S.. "A Generic T-tail Transport Airplane Simulation for High-Angle-of-Attack Dynamics Modeling Investigations", AIAA-2018-1168, https://doi.org/10.2514/6.2018-1168.

[2] Nalepka, J., Danube, T., Williams, G., Bryant, R., and Dube, T., "Real-time simulation using Linux", AIAA-2001-4185, https://doi.org/10.2514/6.2001-4185

[3] Edge, J. "ELC: SpaceX Lessons Learned", Eklektix, Inc., URL: https://lwn.net/Articles/540368/, March, 6, 2013.

[4] Musk, E. "SpaceX News: June 2005 – December 2005," SpaceX, Hawthorne, CA, URL: http://www.spacex.com/news/2005/12/19/june-2005-december-2005, Decemeber 19, 2005. [retrieved May 25th, 2018]

[5] Corbet, J., "Software interrupts and realtime," Eklektix, Inc., URL: https://lwn.net/Articles/520076/, October 17, 2012. [retrieved May 24, 2018]

[6] "tcp(7) – Linux man page," URL: https://linux.die.net/man/7/tcp. [retrieved May 25, 2018]

[7] Corbet, J., "Increasing the TCP initial congestion window," Eklektix, Inc., URL: https://lwn.net/Articles/427104/, February 9, 2011. [retrieved May 25, 2018]