TCP Implementation

Jeff Chase
Department of Computer Science
Duke University
Durham, NC 27708-0129
chase@cs.duke.edu

Chapter Objectives

After completing this chapter, the readers should be able to:

—Understand the structure of typical TCP implementations, and the data structures and actions to respond to TCP-related events.

—Outline the implementation of extended standards for TCP over high-performance networks.

—Understand the sources of end-system overhead in typical TCP implementations, and techniques to minimize them.

—Quantify the effect of end-system overhead and buffering on TCP performance over high-bandwidth networks.

—Understand the role of Remote Direct Memory Access (RDMA) extensions and TCP offload technologies for high-performance IP networking.

On the fastest networks, performance of applications using TCP is often limited by the capability of the end systems to generate, transmit, receive, and process the data at network speeds. For network servers, connection management overheads may also be a limiting factor. End-system performance is determined by a combination of factors relating to the host hardware, interactions between the host and the network adapter, and host system software. In particular, high-performance networking involves fundamental structural issues in the end hosts and operating systems: integration of network buffering with operating system memory management, movement of data across the system/application boundary, and division of protocol-related processing between host CPUs and the network adapter—the Network Interface Controller or NIC.

This chapter discusses end-system software implementation issues for TCP, focusing on issues that affect performance of bulk data transfer on high-speed networks. We bypass several aspects of TCP implementation, including urgent data, the PUSH flag, connection resets, options processing, and state transitions for connection setup and shutdown.

Many of the implementation issues and techniques discussed in this chapter concern the relationship between the TCP protocol stack and the surrounding system, rather than the protocol implementation itself. Some do not affect interoperability, and thus fall outside the scope of the TCP-related RFCs. Even so, they are increasingly important as Ethernet and other IP network technologies advance. As a result of these advances TCP often serves as a standard transport for storage access and server-server coordination in datacenter environments, which were previously the domain of more specialized networking technologies such as FibreChannel. This places additional pressure on TCP/IP implementations to deliver competitive end-to-end (application-to-application) performance.

This chapter first gives a structural overview of a typical TCP implementation followed by a discussion of the protocol-related extensions for high-performance networks, the sources and impacts of software overhead for TCP/IP, end-system techniques for low-overhead TCP/IP networking, approaches to copy avoidance, and the role of protocol offload.

1. TCP IMPLEMENTATION OVERVIEW

RFC 793 [22] sketches the internal structure of a TCP end system, and outlines implementation of basic TCP functions including reliable data transfer, flow control, connections, and multiplexing. It also outlines a set of primitives for the "TCP/user interface", including *SEND* and *RECEIVE* interface signatures. This interface specifies a minimal set of user functions that must be present in any TCP implementation. The TCP architects conceived that the *TCP user* calling this interface is an application built directly above TCP, but it may also be an upper-layer protocol (ULP) such as HTTP or NFS.

Figure 1 depicts the structure of a TCP implementation. The TCP protocol code is separated into a *sender* and a *receiver*; since TCP connections are bidirectional the sender and receive are active on both sides of the connection. RFC 793 explicitly assumes that the TCP sender and receiver are implemented in software as operating system modules. However, it does not preclude implementing the TCP/IP protocol modules on the NIC rather than in the host (see Section 5).

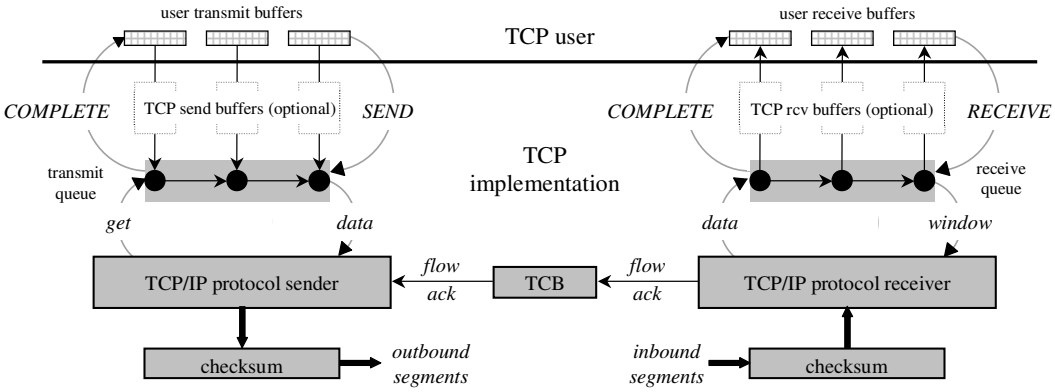

Fig. 1. Structure of a TCP implementation.

There are three primary data structures associated with each TCP connection endpoint:

—The *transmission control block* (TCB) or *protocol control block* stores the connection state and related variables used by both the sender and receiver. TCB state variables include various pointers into the send and receive sequence number spaces.

—The *transmit queue* is the sender's list of buffers containing outgoing data that the TCP user has posted for sending (e.g., using *SEND*) but for which acknowledgments have not yet been received.

—The *receive queue* is the receiver's list of buffers for inbound data that has not yet been delivered to the TCP user.

The TCP sender and receiver modules are state machines that initiate actions on the connection in response to external events, based on state information in the shared TCB. The TCP receiver runs asynchronously to process segments (packets) as they arrive. In typical operating system kernel implementations the TCP receiver runs in the context of a user process or system network input thread invoked by a wakeup signal from the NIC receive interrupt service routine, or a handler invoked by a software interrupt. The TCP receiver may invoke the TCP sender directly if the received segment requires an acknowledgment or if it updates the flow and/or congestion window to allow transmission of new data. The TCP sender may also run in the context of a timer handler or a user process issuing a *SEND* or *RECEIVE* command.

This section summarizes the key aspects of TCP protocol implementation that affect the performance of bulk data transfer. Section 1.1 gives an overview of buffering and data movement within each TCP endpoint, and Section 1.2 discusses data movement between the protocol implementation and the TCP user. Section 1.3 outlines the events that prompt actions from the TCP sender and receiver, and the interaction of the two endpoints to allow rate-controlled data exchange, and 1.4

deals with the timing of retransmissions. Section 1.5 outlines congestion avoidance and control.

1.1 Buffering and Data Movement

The role of the transmit and receive queues is to mediate buffering between the protocol implementation and the TCP user. One should view the receive queue and transmit queue as separate from the TCP protocol implementation itself. For example, in Unix implementations derived from the Berkeley (BSD) reference software releases, these buffer queues reside in the protocol-independent socket layer within the operating system kernel. With TCP protocol offload engines the buffers may reside in host memory while the protocol modules reside on the NIC (see Section 5).

The TCP sender and receiver may access the buffer queues through a simple interface that is similar to any I/O driver, as illustrated in Figure 1. The TCP sender upcalls to the transmit queue to obtain the data for a given sequence range. The TCP receiver notifies the receive queue of correct arrival of incoming data. The TCP receiver may also upcall to the transmit queue to release buffer space when a received segment acknowledges transmitted data. Extraction and merging of segments with specific sequence ranges may take place behind this interface. The BSD TCP receiver maintains the tail of the receive queue internally as a *reorder buffer* to merge segments that arrive out of order.

The mechanisms to coordinate data movement through the buffer queues are paramount for end-system performance. For example, BSD-derived kernels base their buffering and data movement on flexible network buffers called *mbufs*, which incorporate features to move data by reference and reduce the need to copy it. The system constructs packets by stringing together chains of mbufs passed between the levels of the protocol stack. Protocol modules add and remove headers by adding or removing buffers on the chain, or by manipulating the mbuf fields to append data, prepend data, or remove data from the buffer regions. Specific mbuf types may reference storage in another system buffer, such as a virtual memory page or a block in the file cache. Each mbuf is reference-counted to allow fast copying of buffer chains by reference. Current network devices are capable of scatter/gather direct memory access (DMA) to send and receive directly from these buffer chains.

A TCP implementation must bound the amount of memory committed to its buffer queues. Most implementations commit buffer space to the queues lazily, so the queues consume memory only when the bandwidth of the network path does not match the rate at which the TCP user produces or consumes the data. For example, Unix systems set buffer bounds as configurable attributes of the socket object associated with each connection. Data builds up on the queues as lists of mbuf chains whose aggregate buffer size must not exceed the configured socket buffer maximum. If the transmit queue is full then *SEND* operations block or fail; if the receive queue is full then the TCP receiver cannot accept incoming segments. The buffer queue sizes can limit performance, as discussed in Section 1.3.

1.2 Accessing User Memory

The TCP buffering scheme must provide for movement of data to and from the memory of the TCP user. The mechanisms to accept and deliver data are closely

related to the TCP/user interface.

This section deals with data movement for the *SEND* and *RECEIVE* interfaces outlined in RFC 793, which form the basis for the *socket* APIs in Unix/Posix/Linux and other operating systems. In particular, the *SEND* and *RECEIVE* primitives specify a user buffer address and length for the data to be sent or received. Two variants of this basic API are common in TCP/IP implementations.

Copy semantics. RFC 793 defines *SEND* and *RECEIVE* with *copy semantics*. The user is free to modify a send buffer after issuing a *SEND*, but TCP must transmit the data that was in the buffer at the time the *SEND* was issued. The definition of *RECEIVE* is less clear, but it does require that TCP place incoming data at the buffer addresses specified by the user. In practice, implementations meet these constraints by copying data between the buffer queues and user memory; these copy operations are often a limiting factor for TCP communication.

To illustrate, BSD-derived systems using typical NICs handle the socket variants of the *SEND* and *RECEIVE* operations as follows. On a send, the socket layer copies data from user memory across the kernel boundary into a freshly allocated *mbuf* buffer chain, and appends it to the connection's transmit queue. The system sends the data by passing the mbufs in this chain through the TCP/IP stack to the network device driver, which initiates device DMA operations on the buffer regions described by the chain. On the receiving side, the network driver allocates buffers for the device to deposit the incoming stream, and constructs a chain referencing the buffer regions for each incoming packet header and payload. It passes the chain for each arriving TCP segment through the TCP receiver, which appends the chain to the receive queue for the correct connection. When the TCP user requests data with a *read* system call or other *RECEIVE* equivalent, the kernel copies data from the front of the receive queue into the user-specified buffers, releasing each mbuf after it has copied all of that mbuf's data.

Direct access. Another variant of *SEND* and *RECEIVE* allows TCP to access the user buffers directly, bypassing the copy through the optional TCP buffers in Figure 1. This approach is common in Unix systems when the TCP user is a kernel-based upper layer protocol (ULP) such as a Network File System (NFS). If *SEND* passes its buffers by reference, then a buffer chain describing them may be linked directly onto the transmit queue, avoiding the data copy. Note, however, that if the TCP user modifies a buffer with a pending send then it can cause the TCP sender to transmit inconsistent data (e.g., sending the old data once and then the new data in a retransmission). This may occur even if the TCP implementation itself is correct. The interface must prohibit the TCP user from modifying a buffer until all pending sends on it have completed.

Defining a direct-access *RECEIVE* interface is trickier. RFC 793 explicitly allows an implementation to link user buffers directly into a receive queue. However, most NICs select the target buffer for incoming data based on arrival order; without a mechanism to place incoming data correctly in the user buffers, the system must still copy the data to the locations requested by the TCP user.

To place data correctly, the NIC must recognize the TCP connection associated with the incoming data before moving the data into the connection's receive queue buffers in host memory. This *early demultiplexing* requires special support on the NIC. Even with this support, the system might deposit data in a user buffer be-

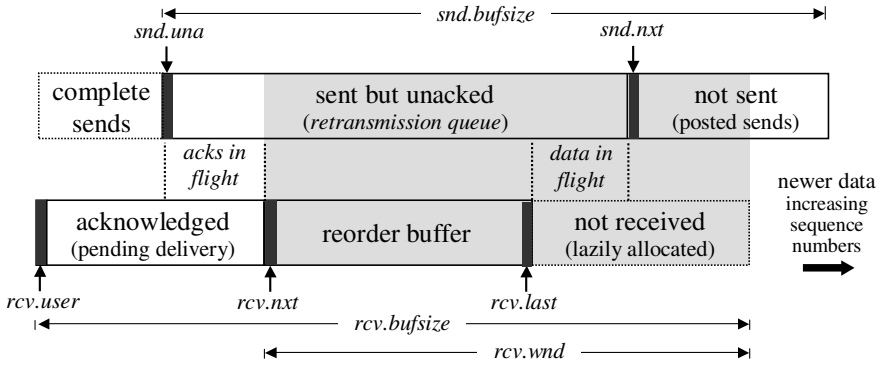

Fig. 2. Sender and receiver views of the sequence space for a unidirectional data flow.

fore TCP has validated it. While RFC 793 is silent on what it means for TCP to "deliver" data to the user, some have argued that this property violates the fundamental guarantee of reliable, in-order delivery. For correctness, the interface must prohibit the TCP user from interpreting any buffer region as valid unless and until a *RECEIVE* covering that region has successfully completed. We return to this issue in Section 4, which discusses techniques and alternative interfaces to enable efficient data movement for high-performance TCP.

1.3 TCP Data Exchange

The TCP endpoints participating in a connection act as cooperating state machines. The endpoints cooperate by exchanging segments. Each segment contains the sequence number of the first (oldest) byte in the segment (*seg.seq*), the segment data length (*seg.len*), status bits, an acknowledgment sequence number (*seg.ack*) for the last (newest) byte received in order, and an advertised receive window size (*seg.wnd*). Either side may send a segment to transfer data or to induce state changes or actions in its peer at any time.

Segment arrival may prompt a TCP endpoint to take any of several actions. If the segment's data completes a *RECEIVE*, the TCP receiver may deliver the data by notifying the TCP user. If the segment contains a new acknowledgment or window update, the TCP sender may notify the TCP user that a *SEND* has completed, or transmit one or more segments containing new data or retransmitted old data. The TCP sender may also send segments as a result of a timer expiration or a new *SEND* or *RECEIVE* command from the TCP user. Any outgoing segments may include new acknowledgments or window updates.

The TCP sender and receiver at opposite ends of a connection maintain local views of the byte sequence space for a unidirectional data flow. Each endpoint's view is captured in state variables in its TCB. The TCP implementation updates these variables as a side effect of segment exchange, or in response to other events such as *SEND* or *RECEIVE* commands. All sequence number arithmetic uses unsigned 32-bit integers modulo 2^32.

Figure 2 depicts the active portion of the sequence space—the portion for which

local buffer space may exist—and related state variables maintained in each TCB. The transmit queue bound is $snd.bufsize$; the receive queue bound is $rcv.bufsize$. The sender's transmit queue holds at least the bytes in the range $[snd.una, snd.nxt-1]$, which comprise the *retransmission queue*: $snd.una$ is the oldest byte that has been sent but not yet acknowledged, and $snd.nxt$ is the next byte to send. The transmit queue also includes posted sends for other data beyond $snd.nxt$ in the send sequence space; typical kernel implementations maintain this portion of the transmit queue at the socket layer.

Conceptually, the receive queue includes buffer space for all bytes in the range $[rcv.user, rcv.user + rcv.bufsize - 1]$, where $rcv.user$ is the next received byte to deliver to the TCP user. Within this range, $rcv.nxt$ is the next byte expected on an incoming segment, and $rcv.last$ is the newest byte received in any segment. If data arrives in order then $rcv.last = rcv.nxt - 1$. However, if segments arrive out of order then $rcv.nxt$ is the oldest missing byte of a *sequence hole*, and the range $[rcv.nxt, rcv.last]$ comprises the receiver's *reorder buffer* for sequencing the incoming data. The range $[rcv.user, rcv.nxt - 1]$ is received and acknowledged data that the receiver must retain pending delivery to the TCP user; typical kernel implementations maintain this portion of the receive queue at the socket layer.

We now summarize the mandated interactions for efficient TCP communication, focusing on aspects that are essential for bulk data transfer. The following applies to unidirectional data transfer from either sender across the connection to its peer receiver.

Basic data transfer. Each receiver maintains a sliding *window* defining the portions of the sequence space that it will accept. The receiver's flow window size is $rcv.wnd = rcv.bufsize - (rcv.nxt - rcv.user)$, and the window is the sequence range $[rcv.nxt, rcv.nxt + rcv.wnd - 1]$. The grey box in Figure 2 depicts the receiver's flow window. The receiver accepts any incoming segment whose sequence range $[seg.seq, seg.seq + seg.len - 1]$ overlaps the window, and suppresses duplicates by discarding any segment that does not. If segments arrive in order, each arrival advances $rcv.nxt$ to $seg.seq + seg.len$.

Acknowledgments. The receiver acknowledges incoming data segments with return segments containing $seg.ack = rcv.nxt - 1$. These segments constitute *cumulative acknowledgments* for all data received in sequence; each acknowledgment subsumes all acknowledgments sent before it. As each acknowledgment segment arrives, the TCP sender advances its $snd.una$ (if necessary) to $seg.ack + 1$.

Note that cumulative acknowledgments do not acknowledge any received data in the range $[rcv.nxt, rcv.last]$. If a segment is lost, the sender cannot determine which (if any) newer segments have arrived. Recent protocol extensions support *selective acknowledgments* (SACK) to convey this information; these extensions are beyond the scope of this chapter.

Flow control. The receiver advertises its flow window size by setting $seg.wnd \leq rcv.wnd$ in return segments. The sender records the window size in $snd.wnd$; The sender's flow window is the range $[snd.una, snd.una + snd.wnd - 1]$. Each time it becomes active, the TCP sender transmits any unsent data that is ready to send and is within its flow window and congestion window (see Section 1.5). Sliding window flow control allows the sender to transmit at the highest possible rate without overflowing the receiver's buffers.

If additional buffer space becomes available on the receiver—for example, if a *RECEIVE* completes with copy semantics (advancing $rcv.user$) or the TCP user posts a new *RECEIVE* with direct-access semantics (advancing $rcv.bufsize$)—then the receiver opens the window and advertises the window update. Although a receiver should never shrink its window unexpectedly, the TCP sender should tolerate this behavior if it occurs.

Segment size. To use the network efficiently, TCP implementations should aggressively coalesce data and status information, and piggyback status updates on data transfers. In particular, larger segments are more efficient because they amortize the overhead of header space and protocol processing for each segment across a larger number of bytes. The *maximum segment size* (MSS) is a property of the network path between the sender and receiver. Each side should advertise its expected MSS—the maximum transmission unit (MTU) of its incoming network interface minus the size of the TCP/IP headers—as a TCP option at connection setup. The MSS for each direction is initially the minimum of the receiver's advertised MSS and the sender's MTU less the TCP/IP header size. The MSS may be constrained further by the network path, as described in Section 2.3.

Buffer size. For peak bandwidth, $snd.bufsize$ and $rcv.bufsize$ must be larger than the network path's round-trip bandwidth-delay product. If a site's receive buffer queue is smaller than the bandwidth-delay product, then its peer can exhaust the buffer by sending at full bandwidth for less time than it takes for the first returned acknowledgment to arrive. This forces the sender to idle until an acknowledgment arrives with an accompanying window update. A small transmit queue may also impact bandwidth, since it limits the rate at which the TCP user can generate data; this increases the probability that the TCP sender exhausts the transmit queue, forcing it to idle for more data. Note that for good performance each buffer queue size must also be significantly larger than the MSS to allow pipelining of segments in the network.

Delayed transmission. If the TCP user produces or consumes data in small units, then TCP must take care to coalesce these units so the connection may continue exchanging data in units of MSS. A TCP receiver should delay acknowledgments for new data to allow the TCP user to consume the data and free buffer space before the window update, and it must advertise $seg.wnd = 0$ if the new window is smaller than the MSS. This policy helps to avoid the phenomenon of *silly window syndrome* in which the window size degrades to small values, producing a stable pattern of data exchange using small segments. In addition, the *Nagle algorithm* dictates that the TCP sender should delay sending newly posted data until it can send a full segment, unless it can send all of its pending data into the current window and $snd.nxt = snd.una$. If the receiver closes its window, the TCP sender uses a *persist timer* to drive periodic transmission of zero-length segments to probe the window.

Acknowledgment pacing. TCP is *self-clocking*; once the sender's flow window or congestion window is exhausted, transmission of new data is driven by arrival of acknowledgments. Thus a TCP receiver must not delay acknowledgments so long as to force the sender to idle the connection and transmit data in bursts. A TCP implementation must impose a time bound on acknowledgment delays, and it must generate at least one acknowledgment for every two full segments worth of data

($2 * MSS$). Failure to do so is an *ack stretch* error. Thus endpoints must have an accurate notion of the MSS; a common flaw is to stretch acks after a change to the MSS as a result of path MTU discovery (Section 2.3).

1.4 Retransmissions

The TCP sender uses a *retransmission timer* to drive retransmission of unacknowledged data. Failure to receive an acknowledgment may result from a lost data segment or a lost acknowledgment. In either case, the sender must recover by setting a timer to fire after a retransmission timeout (RTO) elapses. The TCP sender sets the timer when it transmits a segment and the timer is not already set. If the timer fires before the expected acknowledgment arrives, then the TCP sender retransmits the segment.

Ideally, the retransmission timer fires immediately after a missing acknowledgment is due to arrive, i.e., just over one round-trip time (RTT) after the unacknowledged segment was sent. The difficulty is that network conditions may cause the RTT to vary during the lifetime of the connection. If the timer is too aggressive ($RTO < RTT$) then the sender retransmits too aggressively, wasting network bandwidth and generating overhead. If the timer is too conservative ($RTO > RTT$), then a lost segment causes the connection to idle until the timer fires. Timer management is critical for performance on network paths that are lossy or that exhibit high RTT variance.

RFC 2988 [21] specifies a procedure for managing an adaptive retransmission timer with back-off, using techniques pioneered by Jacobson [12]. This scheme uses integer arithmetic to compute exponentially weighted moving averages of the RTT and its deviation. The RTO is set to the mean RTT plus a safety margin that is a constant factor of the mean deviation. The exponentially weighted moving averages allow the RTO to adapt to persistent changes in network conditions while preserving stability in the presence of short-term fluctuations in the RTT. Scaling the safety margin causes the TCP sender to be more conservative when the RTT variance is high. This approach has proven to be effective in practice.

To retransmit, most implementations simply send the maximum allowable segment beginning with the oldest unacknowledged byte (*snd.una*). Although this approach may retransmit more data than necessary for interactive applications with small segments, it frees the sender from the need to maintain a history of unacknowledged segments.

1.5 Congestion

A retransmission event signals to the TCP sender that congestion may exist in the network. The policies for avoiding congestion and responding to it are critical to the correct functioning of TCP and of the Internet. Congestion management is purely a function of the end systems; the current Internet architecture requires that the network itself respond to congestion merely by dropping packets. While congestion management is beyond the scope of this chapter, we briefly summarize the policies for congestion management using *Additive Increase Multiplicative Decrease* (AIMD), as mandated by RFC 2581 [1] at the time of this writing.

In addition to the receiver-advertised flow window (*snd.wnd*), the TCP sender maintains a *congestion window* (*cwnd*) for each connection, bounding the amount

of data it may inject into the network. At any time, the data transmitted must never exceed $min(snd.wnd, cwnd)$. The sender adaptively determines the value of *cwnd* based on the pattern of acknowledgments.

The sender detects congestion by observing (from a missing acknowledgment) that a segment may have been dropped in the network. To signal congestion early, the receiver immediately responds to each out-of-sequence segment with an acknowledgment segment. Lost or reordered segments are visible to the sender as duplicate acknowledgments. The *fast retransmit* policy permits the sender to retransmit a segment after receiving three duplicate acknowledgments, even if the retransmission timer has not fired. The triple-duplicate acknowledgment suggests a high probability of a lost segment, rather than a reordering that does not require retransmission.

In the absence of a congestion signal, the TCP sender ramps up its transmission rate by steadily increasing *cwnd* as acknowledgments arrive. The purpose is to "probe" the network to converge the data transfer to the available bandwidth of the network path, while adapting to fluctuations in the available bandwidth due to changes in the competing traffic. Increases in *cwnd* are governed by a dynamic threshold value *ssthresh*. If $cwnd \geq ssthresh$, then the TCP sender increments *cwnd* by at most one segment each RTT; this slow additive growth of *cwnd* is termed *congestion avoidance*. If $cwnd < ssthresh$, then the sender ramps up more aggressively, typically by doubling *cwnd* each RTT; this is confusingly known as *slow start*.

If the sender detects congestion, then it must throttle its sending rate by a multiplicatively decreasing *cwnd* (*congestion control*). Expiration of the retransmission timer suggests severe congestion with multiple segment losses; the sender aggressively throttles back by setting *cwnd* to one segment. If the retransmission is prompted by a triple-duplicate acknowledgment, then the sender reduces *cwnd* by half and sets $ssthresh = cwnd$; the arrival of acknowledgments for data still in the network drive additive increases to *cwnd* (*fast recovery*).

2. HIGH-PERFORMANCE TCP

This section introduces protocol features and implementation features for high-performance TCP. After discussing these features, subsequent sections focus on reducing overheads by streamlining the TCP implementation and restructuring it to reduce host overheads for data transfer.

2.1 High Bandwidth-Delay Products

High-bandwidth networks and high-latency networks (such as satellite networks) have large bandwidth-delay products. Such networks are sometimes called "Long Fat Networks" (LFNs). LFNs introduce several performance considerations for TCP implementations.

Most obviously, LFNs require large window sizes, beyond the 16-bit window sizes supported by TCP as originally defined. For this reason, the *window scale option* extends TCP to allow each endpoint to specify a *scale factor* for interpreting its advertised receiver window sizes. An endpoint may use this option to tell its peer that its advertised 16-bit window sizes must be shifted left up to 14 bits. This effectively extends the maximum TCP window size to one gigabyte.

One problem for dealing with large windows is that the window scaling option must be specified at connection setup time. An endpoint may not know in advance that a given network path is an LFN and requires window scaling. Unix-based implementations may select window scaling based on the configured maximum size of the receive socket buffer size ($rcv.bufsize$). This presupposes that the queue sizes are set correctly to exceed the bandwidth-delay product, which is also a property of the network path. One alternative is to set window scaling based on the maximum buffer size supported by the system; the only cost to employing window scaling is that it limits the system to specify window sizes in units of up to 16K.

2.2 Round-Trip Estimation

Another potential pitfall with LFNs is that the accuracy of RTT estimation (Section 1.4) depends on frequent sample measurements of the RTT. Many implementations record only one transmit time per connection, effectively sampling the RTT once per window. The percentage of segments sampled decreases with larger windows, so this may be insufficient for LFNs. In addition, RTT samples from retransmitted segments must be discarded; in this case, the acknowledgment might have resulted from either the original or retransmitted segment, and these cases are indistinguishable.

To provide for more accurate RTT estimation, RFC 1323 [14] introduces a *timestamp option* enabling the sender to place a transmission timestamp in each segment. With each acknowledgment, the receiver returns the timestamp value from the most recently received in-sequence segment containing the oldest byte that was not previously acknowledged and for which the sender included a timestamp. When this acknowledgment with the echoed timestamp arrives back at the sender, the sender can compute an RTT sample by subtracting the reflected timestamp from the current time. The sampled RTT accounts for delayed acknowledgment, retransmissions, and out-of-sequence data.

RFC 1323 also mandates that the 32-bit timestamp values selected by the sender are monotonically increasing with each window until the timer wraps. The receiver may reject a segment with an out-of-sequence timestamp. This policy is called PAWS: Protection Against Wrapped Sequence numbers. It can provide an important safeguard against accepting out-of-sequence data on high-bandwidth networks, which may wrap sequence numbers at the granularity of seconds. PAWS does not specify the timer tick grain used by the sender, but the timer wrap time must be longer than the maximum segment lifetime in the network.

2.3 Path MTU Discovery

TCP is most efficient when it uses the largest MSS accepted by the network path without fragmentation. Using a large MSS may have a cost: it increases the minimum latency through store-and-forward routers. In principle, this could increase the RTT and therefore increase the buffering requirements at the end systems. However, a larger MSS reduces the segment overhead per byte of data transmitted, yielding higher throughputs when adequate buffer space is available. It also reduces per-segment overhead on the end systems (see Section 5).

Path MTU discovery (RFC 1191 [18]) enables a TCP sender to automatically discover the largest acceptable MSS for a network path. An initial MSS is estab-

lished for each connection at setup time, but it may be necessary to reduce the MSS if the MTU of some hop along the network path is too small to accommodate it. Path MTU discovery is increasingly important as new networking standards support larger MTUs to improve performance For example, standard IEEE 802.3 Ethernet frame sizes impose an MTU of 1500 bytes, but many Gigabit Ethernet systems now support "jumbo" frame sizes up to 9000 bytes. While jumbo frames are not widely supported in wide-area transit paths, path MTU discovery enables end systems to use these larger packets when allowable (e.g., in LAN settings) without compromising interoperability.

Dynamic routing may cause the path MTU to change at any time. To discover the maximum MTU of the limiting hop, the TCP sender sets the Don't Fragment (DF) bit in each IP datagram. This forces the router at the limiting hop to drop the packet rather than fragmenting it. The router must generate an ICMP Unreachable error specifying the next-hop MTU. The sender uses this information to reset the MSS before retransmitting. If the MSS is constrained by a limiting hop, the TCP sender periodically probes the network with a segment larger than the MSS to determine if the limitation still exists.

One pitfall for path MTU discovery is that it depends on proper ICMP support along the network path from the TCP sender to the limiting hop. Some routers do not properly generate the ICMP Unreachable error, and some network paths discard ICMP packets because they may act as a vector for denial-of-service attacks. If ICMP Unreachable packets are suppressed for any reason, then TCP connections will hang unless the TCP sender reduces the MSS after a sufficiently long sequence of lost segments. The sender must issue a sequence of probes with progressively larger segment sizes to discover the path MTU in this case.

A TCP implementation must correctly handle dynamic changes to the MSS. For example, each endpoint must ensure that it never leaves more than $2 * MSS$ bytes of data unacknowledged to avoid an ack stretch error. Also, MSS changes may force the TCP sender to resegment data for retransmission if the original segments were dropped just prior to an MSS change.

3. REDUCING END-SYSTEM OVERHEAD

Data transmission using TCP imposes processing overheads in the host operating system (OS) facilities for memory and process management, as well as the TCP/IP protocol stack and the network device and its driver. This overhead adds directly to latency. More importantly at high speeds, overhead may consume a significant share of host CPU cycles and memory system bandwidth, siphoning off resources needed for application processing of the data. The end system may ultimately saturate under the combined load of application processing and networking overhead, limiting delivered throughput.

It is tempting to suppose that the advances in CPU power will render communication overhead increasingly irrelevant even with faster networks, but this is not the case. This section explores the effects of overhead using a simple model that assumes the CPU processing cost is linear with the network I/O rate, but in practice memory system limitations may force the CPU to stall as the network I/O rate increases; the larger number of memory cycles per instruction increases the CPU cost. Often the limiting factor is not CPU processing power itself but the ability to

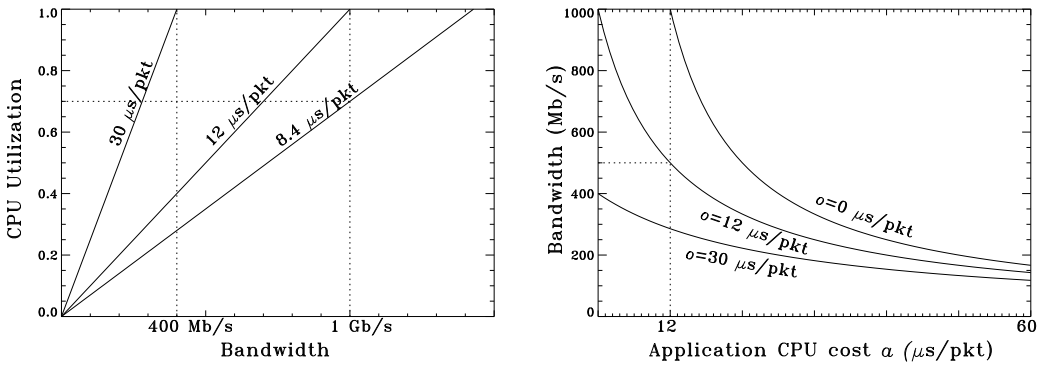

Fig. 3. The effect of communication overhead (o) and application processing cost (a) on CPU utilization and bandwidth. The left-hand graph illustrates a simple linear relationship between bandwidth and CPU utilization. High CPU costs per packet saturate the CPU as bandwidth increases, limiting throughput. The right-hand graph shows the achievable throughput for host-limited systems with varying a and o per 1500-byte packet. The impact of overhead is most significant when the application can process data at high bandwidth.

move data through the host I/O system and memory. Wider datapaths can improve raw hardware bandwidth, but more bandwidth is invariably more expensive for a given level of technology. Advances in network bandwidth follow a step function, but the fastest networks tend to stay close to the limits of the hosts. In particular, optimizations to reduce end system overhead will be critical for with 10 Gigabit Ethernet for several years after its introduction. Multiprocessors or multithreading may improve performance to the extent that the system is able to extract parallelism from the network processing; in particular, multiprocessors may yield higher *aggregate* throughputs for servers handling multiple concurrent streams.

This section outlines the factors that affect end-system performance for TCP, gives an overview of optimizations to reduce overhead, and discusses their implications for the network interface and TCP implementation. Some of these approaches to low-overhead networking are now emerging into common practice. The network interface plays a key role for some of these features, and an increasing number of commercial network adapters support them.

3.1 Overhead, CPU Utilization, and Bandwidth

We first present a simple model to illustrate the impact of overhead on raw communication bandwidth. Suppose that the CPU on a uniprocessor system incurs a fixed processing cost c per byte of data transferred. The cost c is measured in CPU time, e.g., cycles. For convenience, we represent c as the fraction of the processing power available in one second. Then network communication at bandwidth B, measured in bytes per second, yields CPU utilization Bc. The CPU saturates at bandwidth $1/c$, causing the system to become *host-limited*; this is the maximum throughput achievable on this system.

Slow networks rarely expose these limitations: if the effective network bandwidth $B < 1/c$ then the system is *network-limited* and overhead does not affect raw

throughput. For example, suppose a given TCP system incurs 20 cycles of overhead per byte to receive a network stream. A gigahertz CPU running this software incurs a cost of about 30 μs per 1500-byte Ethernet packet, and saturates at a peak bandwidth of 50 MB/s, or 400 Mb/s. The system is network-limited with a 100 Mb/s Ethernet network, but upgrading the network to Gigabit Ethernet causes it to become host-limited at 400 Mb/s rather than delivering the expected order-of-magnitude increase. Since the end system can consume data no faster than 400 Mb/s, TCP flow control prevents a sender from transmitting data at a higher rate.

Software structures that reduce end-system overhead are critical to reaching the performance potential of high-speed networks. In a host-limited system, reducing overhead directly improves throughput. Suppose that the processing cost consists of a fixed cost x per byte and a variable overhead y per byte that may be eliminated through some optimization. The optimization increases the CPU saturation bandwidth from $B = 1/(x+y)$ to $B = 1/x$, yielding a relative throughput improvement of $(x+y)/x$. In the example, reducing the overhead by 25% to 15 cycles per byte yields a 33% improvement in raw bandwidth.

The left-hand graph of Figure 3 illustrates this simple model of the effect of communication overhead. CPU utilization grows linearly with bandwidth, with the slope determined by overhead (given per 1500-byte segment in the figure). An overhead of 30 μs per packet saturates the CPU at 400 Mb/s, as in the example. An overhead of roughly 12 μs per packet enables communication at 1 Gb/s before the CPU saturates; in this case, a 60% reduction in overhead improves raw bandwidth by a factor of 2.5. With per-packet overhead reduced further to 8.4 μs, the system becomes network-limited on a 1 Gb/s network with a CPU utilization of 70%.

3.2 The Role of Application Processing

Even if raw communication is not host-limited, reducing overhead can improve application throughput by freeing resources for application processing. The same formula applies: if a is the application's cost to process its data, and o is the communication overhead, then delivered throughput is $1/a + o$ whe the system is host-limited. The improvement in application throughput from eliminating overhead grows with the ratio of o to a, and is effectively unbounded if $o >> a$. To illustrate, the right-hand graph of Figure 3 shows saturation bandwidth as a function of application processing cost a per packet, for communication overheads o of 0, 12, and 30 μs per packet. With $o = 12\mu$s, the system achieves a raw bandwidth of 1 Gb/s without saturating, but throughput falls off as a increases; eliminating the overhead increases delivered throughput (the $o = 0$ line). In the high-overhead case ($o = 30\mu$s) even raw communication is host-limited; eliminating the overhead more than triples bandwidth at $a = 12\mu$s.

Of course, Amdahl's Law dictates that the benefit of reducing overhead o also declines as a grows and the application itself becomes more CPU-intensive. This effect is shown at the far right of Figure 3; o has a minimal effect on throughput when $o << a$. Even so, consider the reasonable case in which the network is well-matched to the system, allowing raw communication at network speed B ($Bo \leq 1$), and the system is powerful enough to run the application processing at network speed ($Ba \leq 1$). It is easy to see that reducing communication overhead improves application throughput by up to a factor of two in this case. For example, the

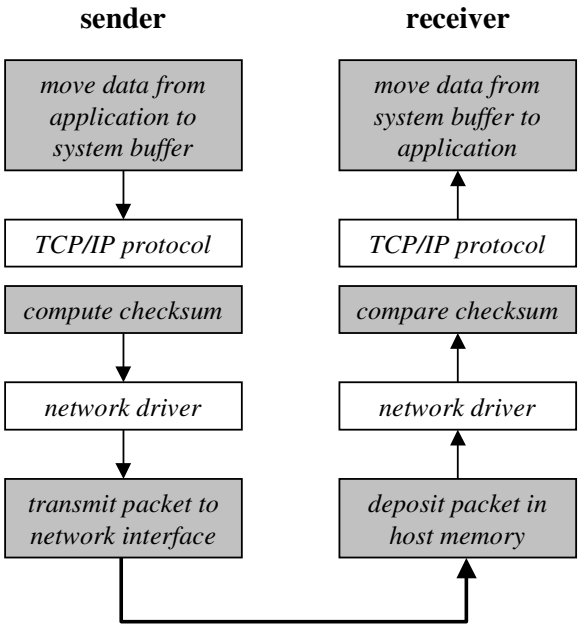

Fig. 4. Sources of end-system overhead for TCP/IP.

$o = 12$ line shows throughput of 500 Mb/s when $a = o = 12\mu$s; eliminating the communication overhead doubles achievable throughput to 1 Gb/s.

3.3 Sources of Overhead for TCP/IP

Figure 4 depicts the key sources of overhead in transmitting data over a TCP connection in a typical system. We can divide overhead into several classes:

—*Per-transfer* overhead includes the cost for each *SEND* or *RECEIVE* operation from the TCP user. These include the costs to initiate each operation—such as kernel system call costs—and the cost to notify the TCP user that it is complete. It also includes the costs to allocate, post, and release buffers for each transfer.

—*Per-packet* or *per-segment* overhead is the cost to process each network packet, segment, or frame. These include the costs to execute the TCP/IP protocol code, allocate and release packet buffers (e.g., mbufs), and field NIC interrupts for packet arrival and transmit completion.

—*Per-byte* overhead includes the cost to copy data within the end system and to compute checksums to detect data corruption in the network. The system incurs these data-touching costs for each byte sent or received at the stages shown in grey in Figure 4.

Let o_t, o_s, and o_b denote the per-transfer, per-segment, and per-byte overheads respectively. Then with transfer size T and segment size S (the MSS), the total overhead o per byte transferred is:

$$o = o_b + o_s/S + o_t/T \tag{1}$$

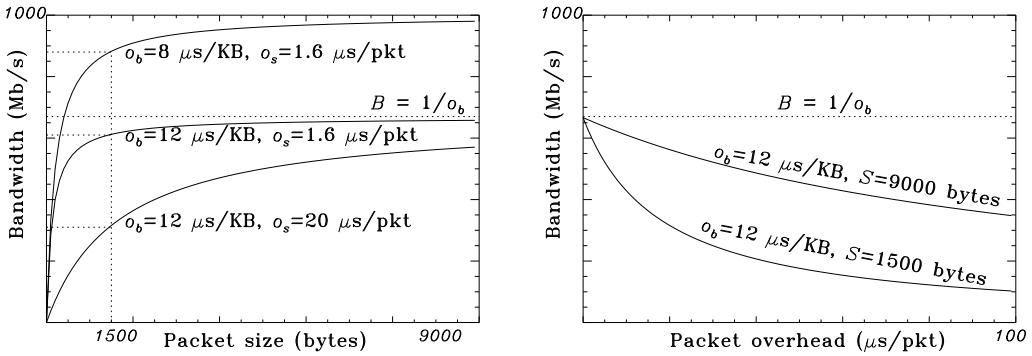

Fig. 5. The relationships packet size, per-packet overhead, and achievable bandwidth for a host-limited system on a 1 Gb/s network.

The formula shows that one simple way to reduce overhead is for the TCP user to use the largest possible transfer size T. This can effectively eliminate the impact of transfer overheads for bulk data transfer. Similarly, the sender and receiver can reduce per-segment costs by using path MTU discovery to identify the largest possible segment size S allowable by the underlying network path, as previously discussed. Note, however, that per-packet and per-transfer costs for buffer management are proportional to the volume of data transferred, given a fixed buffer size. For example, early BSD-based implementations used small mbufs to reduce memory fragmentation, and incurred high per-packet overheads to spread packet data across chains of multiple mbufs [15]. Current implementations use larger mbufs as a result of declining memory costs and faster networks. It is common to place packet data in a single mbuf or two mbufs at most, reducing buffer management overhead.

Sections 3.4 and 3.5 discuss per-packet overheads in more detail. Per-byte data-touching costs—copying and checksumming—tend to dominate per-packet costs and per-buffer costs for bulk data transfer because they are fundamentally limited by the host memory system rather than the CPU. In principle, a well-structured system can entirely eliminate most per-byte overheads. The only unavoidable per-byte costs occur in the application and in moving data between host memory and the NIC. Since copy avoidance is a substantial topic in itself, we leave this issue to the next section. Section 3.6 discusses checksum offloading to reduce host data-touching overheads for checksumming.

3.4 Per-Packet Overhead

Increasing packet size can mitigate the impact of per-packet and per-segment overheads. Figure 5 illustrates the impact of end system per-packet or per-segment overheads on delivered bandwidth for host-limited systems on a 1 Gb/s network. The left-hand graph shows how increasing segment size S increases achievable bandwidth for varying per-segment overhead o_s and per-byte overhead o_b. We assume that other costs (e.g., per-transfer costs) are held constant and are included in o_b. A

per-segment overhead of 1.6 μs has a significant impact on bandwidth for standard 1500-byte Ethernet MTUs. The impact is most pronounced when byte overheads are low enough to allow the system to achieve the full 1 Gb/s of raw bandwidth; in this case, $o_s = 1.6\mu$s limits the system to about 800 Mb/s of raw bandwidth. As packet size grows, the effect of per-packet overhead becomes less significant, and the system converges to bandwidth $1/o_b$. The right-hand graph in Figure 5 illustrates the effect of varying the per-packet overhead for packet sizes of 1500 bytes and 9000 bytes (jumbo frames). These basic relationships also apply for per-transfer costs and per-buffer costs, and on higher-bandwidth networks.

Since packet sizes are constrained by the properties of the network path, often the most effective way to limit the impact of per-packet overhead is to reduce the overhead itself. TCP/IP protocol processing (header processing) itself is one source of per-segment overhead. While TCP/IP protocol processing has often been cited as a potential bottleneck, several studies have indicated that these costs are typically small relative to data-touching overheads and the cost to interface the TCP/IP stack to the NIC device and the surrounding operating system [8; 15; 13; 7]. Jacobson's scheme for *header prediction* was a significant step in reducing TCP protocol overhead [13; 14]. Header prediction identifies and optimizes common-case code paths in the TCP receiver, yielding a highly efficient code path for the common bottleneck path of data packets arriving in-sequence without special needs such as options processing. Mosberger et. al. [19] explored related path optimizations to improve cache locality and reduce memory cycles per instruction (mCPI) for protocol processing.

Although TCP/IP protocol processing is efficient, Section 5 discusses the alternative of offloading it to the NIC. This allows most per-segment processing to occur on the NIC in parallel with application processing on the host CPU, and—more importantly—it introduces new opportunities to reduce per-byte copying costs.

3.5 Interrupts

Interrupts are a significant source of per-packet overhead. With 1500-byte packets, a host receiving data at 1 Gb/s can incur over 80,000 interrupts per second to signal packet arrival events, and another 40,000 interrupts per second to signal transmit completion events on acknowledgment segments. In addition, some systems post a second software interrupt to schedule TCP/IP protocol processing for each incoming segment.

Most high-speed network interfaces support *interrupt coalescing* or *interrupt suppression* to amortize interrupt overheads across multiple packets. This technique selectively delays interrupts if more packets are pending transmission or delivery to the host [10]. This reduces the total number of interrupts delivered; the host interrupt handler may process multiple event notifications from each interrupt. While some interrupt coalescing schemes increase end-to-end latency slightly, they reduce interrupt costs during periods of high bandwidth demand, and may deliver better cache performance for processing the notifications.

Another way to reduce TCP overhead is to execute the TCP receiver common-case path (i.e., the header prediction fast path) directly from the NIC receive interrupt handler, rather than scheduling a separate thread or software interrupt handler to process the headers. In addition, some systems disable some or all

transmit-complete interrupts, which serve to notify the host that it may release transmit buffers mapped onto the NIC. A TCP sender cannot release buffer space for outgoing data segments until the receiver acknowledges the data.

3.6 Checksums

To detect data corruption in the network, TCP/IP generates an end-to-end checksum for each packet covering all protocol headers and data. In most implementations, software running on the host CPU computes these checksums on both sending and receiving sides. This requires the CPU to load all of the data through the memory hierarchy for a sequence of add operations.

One approach to reducing per-byte costs is to coalesce multiple operations that touch the data, completing multiple steps in a single traversal of the packet [9]. For example, some TCP/IP implementations integrate checksumming with the data copy to or from application memory. A similar approach could apply to other processing steps such as compression or encryption. These optimizations are highly processor-dependent and may not yield the expected benefits [7]. Borman [3] demonstrated large reductions in checksum overhead using vector processing in his pioneering work with high-speed TCP on Cray supercomputers.

Another way to eliminate this cost is to compute the checksum in hardware on the network adapter's DMA interface as the packet passes through on its way to or from host memory. The TCP/IP stack and the network device and driver must cooperate in *checksum offloading*. Many high-performance NICs support checksum offloading in the host DMA engine, which computes the raw 16-bit one's complement checksum of each DMA transfer as it moves data to and from host memory. This is an effective optimization, although it sacrifices some protection against data corruption in the host I/O system or low-level software.

Changes to the the TCP/IP stack to use the hardware checksum are typically minor; simply bypass the software checksum. However, three factors complicate hardware checksumming for IP:

—A packet's data may span multiple host buffers; the device accesses each buffer with a separate DMA. A common solution is to combine these partial checksums on the adapter using one's complement addition.

—TCP and UDP actually use two checksums: one for the IP header (including fields overlapping with the TCP or UDP header) and a second end-to-end checksum covering the TCP or UDP header and packet data. In a conventional system, TCP or UDP computes its end-to-end checksum before IP fills in its overlapping IP header fields (e.g., options) on the sender, and after the IP layer restores these fields on the receiver. Checksum offloading involves computing these checksums below the IP stack; thus the driver or NIC must partially dismantle the IP header in order to compute a correct checksum.

—Since the checksums are stored in the headers at the front of each IP packet, a sender must complete the checksum before it can transmit the packet headers on the link. If the checksums are computed by the host-NIC DMA engine, then the last byte of the packet must arrive on the NIC before the firmware can determine the complete checksum.

The last issue may require a compromise between host overhead and packet

latency, since the adapter must handle large packets in a store-and-forward fashion.

3.7 Connection Management

While this section has focused primarily on data transfer, Internet servers are among the most visible performance-critical TCP applications. Many servers manage large numbers of connections for a large user population, although the amount of data transmitted on each connection may be small. This places performance pressure on the mechanisms for connection setup, connection teardown, identifying the connections for incoming segments, buffer allocation across connections, and notifying server applications of connection state. For example, TCB lookup on packet arrival can impose a significant mature TCP implementations use variants of hashing [16] for fast TCB lookups with large numbers of connections. Packet streams often show sufficient locality to benefit further from caching the previously accessed TCB. [17].

Also of concern is the mechanism to notify a user process which subset of a large set of open connections is ready to send or receive data. For example, arrival of an acknowledgment may release buffer space in a connection's transmit queue, allowing the TCP user to successfully initiate a new *SEND* operation on that connection. Similarly, data arriving on a connection may allow the user to complete a *RECEIVE* operation on that connection, and begin processing the data.

Experiments with Internet servers under load has shown that interfaces that poll the state of a set of connections—such as the Unix *select* operation—are not scalable. Instead, TCP implementations should provide a notification queue mechanism to pass connection-related events to the server application [2; 6].

4. COPY AVOIDANCE

In principle, it is possible to eliminate all data copying within the host. Modern network devices access host packet buffers directly in host memory using scatter/gather DMA, eliminating the need for a host CPU to copy data to and from the network interface. The OS can avoid other internal data copying with flexible mechanisms for buffer management, such as the buffer chaining mechanisms discussed in Section 1.1. True "zero-copy" implementations are achievable with the right system structure.

Most problematic is the copying of data between the transmit and receive queues and user buffers. Section 1.2 outlines the *SEND* and *RECEIVE* interface mandated by RFC 793, which is similar to the socket interface to TCP in many systems. This interface is simple to specify, understand, and use, which has contributed to the success of TCP/IP. However, its copy semantics inhibit high-performance data movement because supporting the interface generally requires actually copying the data. On the sender, copying the data protects it from modification if the process reuses the memory before the transmit of the old data completes. On the receiver, copying allows the system to place received data at arbitrary virtual addresses specified by the application.

Avoiding this copying involves complex relationships among the network hardware and software, operating system buffering schemes, and the TCP/user interface. Section 1.2 outlines a direct-access variant of the *SEND* and *RECEIVE* interface that allows TCP to directly access the user buffers to avoid copying the data. This example illustrates two key points about copy avoidance: (1) avoiding data copying

for a *RECEIVE* interface generally requires specific support on the NIC to process the incoming packet stream and deposit the data in suitable host memory buffers, and (2) the effects on the semantics of TCP user operations are subtle and must be defined carefully. The problem is particularly difficult when moving data into a user process with a protected virtual address space, and other receive cases where buffer alignment constraints exist. Copy avoidance is still an active area of research and development, and there are no fully general solutions available.

This section surveys several approaches to copy avoidance for high-performance TCP. We first discuss *page remapping*, a common technique that uses virtual memory to reduce copying across the TCP/user interface. While page remapping preserves copy semantics, it has many limitations. We then present two more general solutions that involve a comprehensive restructuring of the end systems, extending the RFC 793 interface with new variants of the *SEND* and *RECEIVE* primitives to enable fast-path data movement for high-performance TCP.

Some TCP experts favor a strict reading of RFC 793 to preclude additional variants of these primitives that do not precisely match the mandated interfaces. However, RFC 793 notes that "considerable freedom is permitted to TCP implementors to design interfaces which are appropriate to a particular operating system environment", as long as the "minimum functionality" of the mandated interfaces is present. While the alternative interfaces are more complex to define and use, they are appropriate for Upper Layer Protocols (ULPs) using TCP as a transport, and may be exposed directly to performance-critical applications.

4.1 Page Remapping

Page remapping—also called *page flipping*—leverages virtual memory management in the operating system to reduce copying in kernel-based TCP implementations. The idea is to use page-grained virtual memory mappings to change buffer access permissions or address bindings without copying the data to a new buffer. On the sender, virtual memory allows the system to protect against modifications to user buffers with pending transmits, even if they are accessible in a user process address space. On the receiver, virtual memory allows the system to deliver data to user buffers under certain conditions by changing the physical page frame mappings for the virtual page addresses covering the user buffers. Research systems have used page remapping for several years [4; 11; 7], and it is now emerging into more common use.

The primary support for page remapping typically resides at the socket layer in the OS kernel. The optimizations trigger only if the data transfer sizes requested by the application are larger than the page size. On the receiver, page remapping requires an MSS matched to the page size, page-aligned application buffers, and a network interface that deposits packet data on a page boundary. In general, this last requirement means that the network interface must split the packet header from its data for common protocols such as TCP.

Page remapping activates when a process requests the kernel to transfer a page or more of data between a network socket and a page-aligned user buffer. Instead of copying data to or from a buffer chain, as described in Section 1.2, the system passes the data by reference. For example, in a BSD-derived Unix system using mbufs (Section 1.1), *SEND* places the new data on the transmit queue by creating and

appending *external mbufs* that reference the page frames backing the application's virtual buffer. The buffer chain and its pages are then passed through the TCP/IP stack to the network driver, which initiates DMA directly from the pages. To preserve copy semantics, the kernel marks any page with a pending transmit as *copy-on-write*, disabling write access. The kernel releases the copy-on-write mapping when the receiver acknowledges the data.

On the receiver, incoming data arrives on the receive queue as buffer chains passed up through the protocol stack. Page remapping can optimize delivery of some subset of the received data under specific conditions. Remapped data must arrive as complete pages; these pages must contain portions of the sequence space that align with virtual page boundaries in the user buffers specified in the *RECEIVE* calls. If these conditions are met, the kernel delivers each page of data by remapping the virtual page of the user's buffer to the physical page frame containing the data, e.g., as referenced by an external *mbuf* passed up from the network driver. Copy-on-write is unnecessary because there is no need to retain the kernel buffer after the read; the kernel simply releases the receive queue buffer descriptors, leaving the buffer pages mapped into the application address space. It also releases any user pages whose virtual address mappings were broken by the remapping operations, preserving equilibrium between the process and the kernel buffer pool.

Page remapping is fragile. For example, consider the case where the TCP user attempts to overwrite a send buffer before the TCP acknowledgment is received. Since the mapping is copy-on-write, the system incurs a page fault, copies the page, and maps the user's virtual buffer to the new copy before allowing the write to proceed. This is necessary to preserve emulated virtual copy semantics, but it destroys any benefit from page remapping. In general, the sending application must allocate enough virtual address space to map the entire transmit queue in order to benefit fully from page remapping. This effectively means that the virtual memory allocated to the application's send buffers must scale with the window size. The window size depends on the network path and may grow very large for LFNs (Section 2.1), and in any case is unknown to the application.

Virtual copies with page remapping are also fragile on the receiver without special support on the NIC. At minimum, the NIC must separate incoming packet headers from their payloads (*header splitting*) and deposit the payloads into buffers aligned to system virtual memory page boundaries. If the MSS is smaller than a page, as is commonly the case, then the NIC must recognize the TCP connection for each incoming packet and, and process the transport headers (see Section 5) to "pack" payloads into contiguous page-size buffers for each connection. These packets may arrive at the receiver out of order and/or interspersed with packets from other flows. Even when the received data is properly packed into page frames, the system may deliver the pages by remapping only when the application's *RECEIVE* buffers are virtually contiguous and suitably page-aligned with the data stream.

Moreover, consider the difficulty with using page remapping for a ULP such as a TCP-based storage protocol (e.g., NFS). In this case, the TCP payloads include ULP headers that affect the data alignment. The data itself (e.g., file blocks) must be page-aligned to deliver to a user process using page remapping; most systems also require aligned data to link into the system I/O cache. This means that the NIC must separate not just the TCP headers from the data, but also the ULP headers;

thus the NIC must incorporate specific support for each ULP that benefits from page remapping, as well as support for TCP itself. The NIC processing may be complex for ULPs such as NFS that use variable-length headers or that require ULP-level state to decode the incoming headers.

A final disadvantage is that page remapping requires a TLB invalidation after any change to a virtual-physical mapping and after any page access restriction restriction (e.g., copy-on write). This is expensive on most architectures, and it inhibits scalability on shared memory multiprocessors.

4.2 Scatter/Gather I/O

A more general alternative is to introduce a variant of the *SEND* and *RECEIVE* interfaces that do not require copy semantics. One approach is to use *scatter/gather I/O* to allow *SEND* and *RECEIVE* from arbitrary buffer locations, with the system rather than the TCP user selecting the addresses for incoming data. For example, the receiving NIC may deposit incoming data from each packet at any convenient location. The data is described internally by a buffer chain. Scatter/gather I/O defines a new API to allow the system to exchange these buffer chains directly with the TCP user or application.

IO-Lite [20] proposes a general form of scatter/gather I/O between an operating system kernel and user processes. IO-Lite allows more flexible data placement and reduces copying for inter-process communication and storage access as well as network communication. To allow application access to scatter/gather buffers, the application address space is given read-only access to blocks of memory containing portions of the system buffer pool. The system delivers data by passing references to locations in this pool. Buffers are immutable as long as they are referenced, allowing free exchange of buffer chains among processes and system components. IO-Lite introduces a redesigned system I/O cache to use scatter/gather buffering in other OS subsystems. To write or send data, applications may create new buffer chains containing references to received data as well as new data created by the applications. Modifications are handled by manipulating the buffer chains to insert references to new locations for the updated data, to avoid modifying the original copy.

Scatter-gather I/O as proposed in IO-Lite entails a comprehensive restructuring of the operating system and I/O interfaces. One pitfall with this approach is that data is not necessarily contiguous in memory or aligned in a useful way. For example, the application cannot in general overlay programming language data structures on the received data without copying some or all of it. Also, since received data is not page-aligned, received data cannot in general be delivered securely to user-level processes without copying it. Mapping the pages containing the received data into a user process address space exposes the containing pages in their entirety, not just the portions occupied by the received data. This may expose data that is private to other applications, unless the NIC supports early demultiplexing to isolate data arriving on different connections. While this places less stringent demands on the NIC than general support for page remapping, it undermines a potential benefit of scatter/gather I/O.

4.3 Remote Direct Memory Access (RDMA)

Another approach to copy avoidance is based on the direct-access TCP/user interface outlined in Section 1.2, coupled with NIC support to "steer" incoming data directly into user-specified buffers. This approach is directly applicable to TCP offload NICs that implement the full TCP/IP protocol on the network adapter (Section 5).

One proposal, termed *Remote Direct Memory Access (RDMA)*, is based on a messaging protocol layered above the transport, in conjunction with NIC support to recognize this protocol. The software registers buffer regions with the NIC driver, and obtains protected buffer reference tokens called *region IDs*. It may then exchange these region IDs with the connection peer by inserting them into RDMA messages sent over the connection. Special RDMA message directives enable one end system to read or write remote memory regions named by the region IDs. The receiving NIC recognizes and interprets these directives, validates the region IDs, and performs protected data transfers to the named buffer regions, e.g., to place received data payloads directly into the receive buffers registered for the TCP user. The name RDMA suggests that each end system may be viewed as a remote device that initiates direct memory transfers to and from the memory of its connection peer.

RDMA requires new facilities and interfaces for buffer management, including buffer registration and buffer protection. Limits on the number of registered buffers may complicate application buffer management. Buffer registration and deregistration are protected operations and therefore may be unsuitable if many small buffer transfers are required.

The idea of RDMA has been around by various names for many years. It is an important component of the VI and Infiniband network architectures, and of small-area network (SAN) systems based on these and related host interface standards. RDMA is also related to *sender-based memory management* [5]. RDMA has been used successfully for performance-critical applications in many settings. It handles arbitrary MTUs and buffer alignments, and supports ULP protocols structured to use it, e.g., for network storage access. The Direct Access File System protocol (DAFS) is one important ULP designed for use with RDMA. The popular Network File System (NFS) can also run over a Remote Procedure Call (RPC) protocol extended for RDMA. It is also straightforward to isolate the RDMA complexity within the kernel behind a sockets-like interface based on RFC 793, for use by unmodified applications. Microsoft's Winsock Direct is one example of such a system.

At the time of this writing, RDMA implementations exist primarily in NICs for network link technologies (e.g., SANs) that are proprietary or supplied by a single vendor. While the concepts of RDMA are compatible with standard technologies for IP networking, RDMA requires a new protocol and cooperation between the NICs at either end of the connection. This protocol and the programming interfaces for RDMA buffer management must be standardized so that system and application software can interoperate with multiple RDMA implementations. The VI architecture is one example of a complete interface specification for RDMA.

RDMA for TCP has been controversial. To reduce the impact on the TCP

transport itself, RDMA may be defined as a new request/response ULP layered as a "shim" above TCP, and thus it cannot be viewed as a general solution for all TCP/IP communication. A key point of contention is whether it is "legal" for an RDMA NIC to act on RDMA directives in out-of-sequence segments in order to perform reassembly of the incoming data stream directly in the user buffers. This would significantly reduce the cost and complexity of RDMA NICs. However, some TCP experts this as a violation of TCP ordering properties. One argument is that the direct-access interface is itself a violation of established ordering guarantees if resequencing of out-of-order data takes place directly in the user buffers (see Section 1.2). This view precludes any possibility of a general zero-copy receive interface for TCP, and it is not supported by RFC 793 or any other RFC. A more serious argument holds that out-of-sequence RDMA is a technical layer violation if it exposes the contents of out-of-order segments as a "hint" to the ULP protocol implementation (RDMA) itself. This is a conceptual problem rather than a practical one, since in-order delivery to the end application is preserved. Even so, the RDMA model does not fit naturally with the rigid sequenced byte stream model of TCP; it is better suited to new advanced IP transports such as SCTP.

Work is under way within the IETF to define RDMA protocol standards for TCP and other IP transports. An RDMA Consortium has proposed draft standards for RDMA over TCP, and initial implementations of the standard are in progress. This standardization is necessary to ensure safe interoperability among NICs from different vendors and their associated software. Some NICs supporting TCP RDMA are already on the market, driven by commercially important applications, primarily IP network storage protocols such as DAFS and NFS/RDMA. One example of an early RDMA/TCP implementation is the Emulex GN9000 network adapter for Gigabit Ethernet, which supports a VI host interface.

5. TCP OFFLOAD

With increasing network speeds—including 10 Gigabit Ethernet—there is a rebirth of interest in supporting TCP/IP protocol functions directly on the adapter. NICs with direct support for general TCP/IP communication are often referred to as a *TCP Offload Engines* or TOE NICs. TOE NICs support TCP/IP protocol processing, generalizing the relatively simple and common support for TCP checksum offloading in advanced NICs (see Section 3.6). In addition, several NIC models now act as Host Bus Adapters (HBAs) for TCP-based ULPs that encapsulate device protocols—primarily network storage protocols such as iSCSI—over TCP/IP. These HBA NICs also incorporate full support for TCP offload.

TCP offload significantly reduces per-packet overheads for TCP/IP protocol processing on the host CPUs. While TCP/IP protocol processing itself is not a primary bottleneck on well-implemented systems (see Section 3.4, it can consume a significant share of the system resources. In one recent experiment with a BSD-derived implementation, TCP/IP protocol overhead alone could consume 30% of CPU time on a 500 MHz Alpha 21264 receiver using standard Ethernet frames (1500-byte MTU) at 1 Gb/s [7]. In addition, TOE NICs and HBAs independently coalesce data arriving in multiple small segments, naturally obtaining the benefits of interrupt coalescing discussed in Section 3.5.

More importantly, we have seen in the previous section how protocol offload

can help to avoid expensive copy operations within the host. A TCP-capable NIC can "pack" data in separate host buffers for each connection, with the TCP headers stripped off. These buffers can be page-sized and aligned, even if the MTU and MSS are small, enabling more general page remapping optimizations. Since a TCP NIC can determine the sequence range of each segment, it can also deposit data directly in user buffers, enabling a direct-access *RECEIVE* interface. On the sending side, a TCP-capable NIC can schedule the movement of data directly from user buffers down to the NIC, as appropriate for pacing each connection. TCP offload alone does not provide general support for ULPs, which intermix ULP headers into the data as described previously. However, TCP offload NICs provide the foundation for RDMA support to handle copy avoidance for ULPs in a general way. Copy avoidance may well prove to be a compelling benefit of TCP offload.

The host/NIC interface for TCP protocol offload presents difficult architectural choices. Since a TOE NIC performs TCP's demultiplexing functions directly on the adapter, the host/NIC interface must explicitly name connections, with separate buffer queues for each connection. In effect, the TCP implementation accesses its per-connection transmit and receive queues in host memory across the device DMA interface. To be fully general, this approach requires new device interface standards to bypass software TCP/IP support in standard operating systems. One alternative is to base this interface on the Virtual Interface Architecture (VI), as in the Emulex GN9000 NICs. One advantage of this approach is that it incorporates support for RDMA using an accepted interface, and so supports important RDMA-based ULPs such as the Direct Access File System (DAFS) in a general way.

The host interface problem is simplified for ULP-specific TCP offload NICs (HBAs). For example, an iSCSI NIC might appear to the host as any other SCSI storage controller, completely insulating the host from the TCP/IP functions. In addition to terminating TCP endpoints, the NIC can also handle ULP headers to steer ULP payloads to the correct application buffers. This enables the system to avoid copying payload data from the ULP, without the need for general RDMA support. This is similar to FibreChannel NICs (or HBAs), which implement an I/O block transport on the NIC. While this approach is simple and effective, it restricts the NIC to act as a single-function device.

While host CPU speeds may keep pace with increasing network bandwidths, offloading TCP/IP functions to the NIC device can free up a significant share of CPU resources for application-level processing. However, the TOE approach will be effective only if the TOE NIC can handle TCP communication and its host interface at wire speed for common usage patterns. Connection setup and teardown are particularly difficult to handle efficiently, and today's Internet server loads often involve many short connections. Even so, given the maturity of the TCP/IP protocol, some elements of protocol processing can be accelerated with specialized hardware. There are several hardware design and implementation alternatives that affect the performance and cost of TCP/IP offload adapters. The topic of hardware implementation of TCP is covered in the following chapter.

6. SUMMARY

For high-speed networks, the end system implementation of TCP can become the performance bottleneck. In fact, most traditional implementations, such as BSD

implementation, cannot cope with the emerging high-speed networks (e.g., 10 Gbps Ethernet) at line speed. This chapter summarizes critical performance issues for TCP implementation in end systems, and surveys solutions for improving bulk transfer performance.

7. REVIEW QUESTIONS

(1) Why is TCP implementation an important factor in high-performance TCP/IP networking?

(2) Discuss the protocol and implementation features in a typical TCP implementation that affect TCP performance.

(3) What are the sources of end-system overheads in a typical TCP implementation?

(4) Discuss three techniques to reduce end-system overheads.

(5) What is a long fat network (LFN)? What specific performance issues are introduced by LFNs for TCP implementations.

(6) Explain *data copying* in a typical TCP implementation. What is the effect of copying on TCP and application performance?

(7) What is Page Remapping? Under what conditions can copying be avoided using Page Remapping? What are the limitations of Page Remapping? What support does it require from the NIC?

(8) Explain how Scatter/Gather can avoid data copying across the TCP/user interface. What are the limitations and pitfalls of this approach?

(9) What is RDMA? Discuss how RDMA helps avoid data copying. What are the limitations of RDMA?

(10) What is TCP offload? How can TCP offload improve application performance?

REFERENCES

[1] Mark Allman, Vern Paxson, and W. Stevens. Internet Engineering Task Force, RFC 2581: TCP congestion control, April 1999.

[2] Gaurav Banga and Jeffrey C. Mogul. Scalable kernel performace for Internet servers under realistic loads. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–12, June 1998.

[3] David A. Borman. Implementing TCP/IP on a Cray computer. *ACM SIGCOMM, Computer Communication Review*, 19(2):11–15, April 1989.

[4] Jose Brustoloni and Peter Steenkiste. The effects of buffering semantics on I/O performance. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 277–291. USENIX Association, October 1996.

[5] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John Wilkes. An implementation of the Hamlyn sender-managed interface architecture. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, October 1996.

[6] Abhishek Chandra and David Mosberger. Scalability of Linux event-dispatch mechanisms. In *Proceedings of the USENIX Annual Technical Conference*, pages 231–244, June 2001.

[7] Jeffrey S. Chase, Andrew J. Gallatin, and Kenneth G. Yocum. End system optimizations for high-speed TCP. *IEEE Communications, Special Issue on High-Speed TCP*, 39(4):68–74, April 2001.

[8] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[9] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 200–208. ACM, September 1990.

[10] Peter Druschel, Larry L. Peterson, and Bruce S. Davie. Experience with a high-speed network adaptor: A software perspective. In *Proceedings of the SIGCOMM Symposium on Communications Architectures and Protocols*. ACM, August 1994.

[11] Hsiao-Keng and Jerry Chu. Zero-copy TCP in Solaris. In *Proceedings of the USENIX Annual Technical Conference*, January 1996.

[12] Van Jacobson. Congestion avoidance and control. *ACM Computer Communication Review; Proceedings of the SIGCOMM '88 Symposium, August, 1988*, 18, 4:314–329, 1988.

[13] Van Jacobson. 4BSD header prediction. *ACM Computer Communication Review*, 20(2):13–15, April 1990.

[14] Van Jacobson, Robert Braden, and David Borman. Internet Engineering Task Force, RFC 1323: TCP extensions for high performance, May 1992.

[15] Jonathan Kay and Joseph Pasquale. The importance of non-data touching processing overheads in TCP/IP. In *Proceedings of the SIGCOMM Symposium on Communications Architectures and Protocols*, pages 259–268. ACM, September 1993.

[16] Paul E. McKenney and Ken F. Dove. Efficient demultiplexing of incoming TCP packets. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 269–279. ACM, August 1992.

[17] Jeffrey C. Mogul. Network locality at the scale of processes. *ACM Transactions on Computer Systems (TOCS)*, 10(2):81–109, May 1992.

[18] Jeffrey C. Mogul and Steve E. Deering. Internet Engineering Task Force, RFC 1191: Path MTU discovery, November 1990.

[19] David Mosberger, Larry L. Peterson, Patrick G. Bridges, and Sean O'Malley. Analysis of techniques to improve protocol processing latency. In *Proceedings of the SIGCOMM Symposium on Applications, Technologies, Architectures and Protocols for Computer Communication*. ACM, August 1996.

[20] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. *ACM Transactions on Computer Systems (TOCS)*, 18(1):37–66, February 2000.

[21] Vern Paxson and Mark Allman. Internet Engineering Task Force, RFC 2988: Computing TCP's retransmission timer, November 2000.

[22] Jon Postel. Internet Engineering Task Force, RFC 793: Transmission Control Protocol, September 1981.