Chapter 4 End-to-End Protocol Layer

Problem Statement

The end-to-end protocol layer, also known as the transport layer, is like the skin of the whole protocol hierarchy that can provide valuable services to application programs. In Chapter 2, the physical layer provides node-to-node single-hop communication channels between directly-linked nodes so that issues such as “how fast to send the data?” and “does the data reliably reaches the receiver attached on the same wired or wireless link?” arise and are resolved in Chapter 2. In Chapter 3 the IP provides host-to-host multi-hop communication channels across the Internet so that the same questions above in Chapter 2 arise again and shall be resolved in this chapter. Since there may be multiple application processes running on a host, the end-to-end protocol layer needs to provide process-to-process communication channels between applications processes of different Internet hosts. Similar to the link layer, the services provided by the end-to-end protocol layer include (1) addressing, (2) error control, (3) data reliability, and (4) flow control. Addressing determines to which application process a packet belongs; error control detects if the received data is valid; data reliability guarantees the whole transferred data can safely reach its destination; flow control controls how fast the sender should send the data.

Driven by different sophistication of demands from applications, how to incorporate the end-to-end services is one big issue. Two transport layer protocols have evolved to dominate: the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). While TCP and UDP exercise the same addressing scheme and similar error control, they differ much in data reliability and flow control. TCP elaborates all the mentioned services while UDP completely omits the data reliability and flow control services. Due to the sophisticated services provided by TCP, TCP has to establish a connection first (i.e. connection-oriented protocol) and become stateful, at the hosts but not the routers, to keep necessary per-connection (also known as per-flow) information to realize per-flow data reliability and flow control for a specific process-to-process channel. On the contrary, UDP is stateless without having to establish a connection (i.e. connectionless protocol) to exercise its addressing and error control services.

However, the services provided by TCP and UDP are limited without
considering timing and synchronization information between hosts running real-time transfers. So on top of the primitive end-to-end service, real-time applications often has to incorporate an extra protocol layer to enhance the services. One standard protocol of such service is Real-time Transport Protocol (RTP). It provides services such as synchronization between audio and video streams, data compression and de-compression information, and path quality statistics (packet loss rate, end-to-end delay and its variation).

Since the end-to-end protocol layer is directly coupled with its upper layer – the application layer, the programming interface, i.e. socket, for network programmers to access the underlying services is important. Nevertheless, the TCP and UDP socket interfaces are not the only services accessible by the application layer. Applications can bypass the end-to-end services and directly uses the services provided by the IP or link layers. We discuss how Linux programmers access the services down to the end-to-end, internetworking, or even link layers through the socket interfaces.

Throughout this chapter, we intend to answer (1) why the end-to-end services provided by the Internet architecture were designed into the way it is today, and (2) how Linux realizes the end-to-end protocols. The chapter is organized as follows: Section 4.1 identifies the objectives and issues in the end-to-end protocol layer, and then compares them with those of the link layer. Section 4.2 and 4.3 then describe how Internet resolves the end-to-end issues. Section 4.2 illustrates the most primitive end-to-end protocol in the Internet – UDP, which provides basic process-to-process communication channels and error control. Section 4.3 focuses on the most widely used Internet protocol – TCP, which equips applications with not only process-to-process communication channel and error control, but also data reliability, flow control, and congestion control. The services discussed so far, including those in Chapter 2, 3 and 4, can be directly accessed by application programmers through the socket interfaces. Section 4.4 gives the Linux way of realizing the socket interfaces. Because the extra software layer for real-time applications, RTP is often embedded, as library functions, in the applications themselves. Section 4.5 describes how RTP and RTCP are employed by the application layer.

4.1 General Issues

The end-to-end protocol, as the name suggests, defines the protocol between the end points of a communication channel. The most apparent
service of end-to-end protocol layer is to provide process-to-process communication channels for application processes. An application running on an operating system is defined as a process. Since there may be multiple application processes simultaneously running on a single host, with the aid of process-to-process channels, any application processes running on any Internet hosts can communicate with one another. The transfer unit in this layer is defined as a segment. The traffic flowing in a process-to-process channel is defined as a flow. The issues on the process-to-process channel are very similar to those on the node-to-node channel in Chapter 2. It addresses the connectivity requirement by process-to-process communication with error control for each segment and data reliability for each flow, and the resource sharing requirement by flow control for each flow.

When communicating over the process-to-process channel, classical issues such as data reliability and flow control arise. These issues had appeared in Chapter 2, but the solutions there might not apply here. As shown in Figure 4.1, the major difference between the single-hop node-to-node and the multi-hop process-to-process channels lies in the network delay distribution. In Chapter 2, because the delay distribution between directly-linked hosts is very condensed near a certain value (this depends on the chosen link layer technology), reliability and flow control problems are easier to solve. In contrast, the network delay in the process-to-process channel is large and may vary dramatically, so reliability and flow control algorithms should accommodate large delay and delay variation.

![Figure 4.1 Differences between single-hop and multi-hop channels.](image)

Table 4.1 provides the detailed comparison between node-to-node protocols (on single-hop channels) and end-to-end protocols (on multi-hop...
channels). They provide services on top of their underlying physical layer and IP layer, respectively. The physical layer is in charge of transmitting bit-stream signals onto a wired/wireless broadcast/point-to-point link using baseband/broadband techniques. However, there may be multiple nodes attached on the link, thus the link protocol layer defines the node address (MAC address) to provide node-to-node communication channels within a link. Similarly, the IP layer provides the capability of transmitting packets onto a host-to-host route towards the destination. There may be multiple application processes running on the host of each end, thus the end-to-end protocol layer defines the *port number* to address the process on a host.

### Table 4.1 Comparison between node-to-node and end-to-end protocols.

<table>
<thead>
<tr>
<th></th>
<th>Node-to-Node Protocol Layer</th>
<th>End-to-End Protocol Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Base on what services?</strong></td>
<td>physical layer</td>
<td>internetworking layer</td>
</tr>
<tr>
<td><strong>services</strong></td>
<td>addressing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>node-to-node channel within a link (by MAC address)</td>
<td>process-to-process channel between hosts (by port number)</td>
</tr>
<tr>
<td></td>
<td>error control</td>
<td></td>
</tr>
<tr>
<td></td>
<td>per-frame</td>
<td>per-segment</td>
</tr>
<tr>
<td></td>
<td>data reliability</td>
<td>per-flow</td>
</tr>
<tr>
<td></td>
<td>per-link</td>
<td></td>
</tr>
<tr>
<td></td>
<td>flow control</td>
<td>per-flow</td>
</tr>
<tr>
<td></td>
<td>per-link</td>
<td></td>
</tr>
<tr>
<td><strong>channel delay</strong></td>
<td>condensed distribution</td>
<td>diffused distribution</td>
</tr>
</tbody>
</table>

### Addressing

The source/destination pair of *port number* is used to further address the process-to-process communication channel between application processes of two hosts. Together with the source/destination pair of IP address and the protocol id (indicating TCP or UDP, as discussed in the next two sections), the process-to-process channel can be uniquely identified. The 5 fields, totally 104 bits, together form the *5-tuple* for flow identification. The end-to-end protocol layer splits the data from the upper layer, i.e. the application process, into segments, and hands over the segments to its lower layer, i.e. the IP layer. Each segment is often put into an IP packet unless the segment is too big and needs fragmentation.

### Error Control and Data Reliability

Error control and data reliability are important because datagram networks occasionally lose, reorder, or duplicate packets. Error control focuses on detecting or recovering bit errors within a transferred unit, be it a frame or segment, while data reliability further provides retransmission mechanisms to
recover from what appears to be missing or incorrectly received transferred unit. As for error control, Table 4.1 indicates that the link protocols are per-frame while end-to-end protocols are per-segment. For data reliability, end-to-end protocols provide per-flow reliability but most link protocols, such as Ethernet and PPP, do not incorporate retransmission mechanisms. They leave the burden of retransmission to their upper layer protocols. However, some link protocols, such as IEEE802.11 wireless LAN, operating in environments that could encounter severe frame loss rates, have built-in data reliability mechanisms to reduce the inefficiency of frequent retransmissions by upper layer protocols. Consequently, for example, a huge outgoing segment from the end-to-end protocol layer, which may be split into 5 packets in the IP layer and further split into 10 frames in the link protocol layer, will have lower probability to be retransmitted because the link layer can handle the successful transmissions of all the10 frames without having to trigger its end-to-end retransmission of an entire huge segment.

The delay distribution also matters when designing the retransmission timer. In Figure 4.1, it would be OK if we set the retransmission timer of the link channel to timeout in a fixed value, say 10 ms. However, it is difficult to set that in the end-to-end channel due to the diffused delay distribution. In Figure 4.1, if we set the timeout to 150 ms, some segments will be falsely retransmitted and the network will contain many duplicate segments. If we set it to 200 ms, any segment lost will cause the sender to sleep for 200 ms, resulting in low performance. All these tradeoffs influence the design choices between the link and end-to-end channels.

Flow Control and Congestion Control

Flow control and congestion control play an important role in end-to-end protocols because in a wide area network the situation could be very much more complex than those in local area networks. Flow control runs solely between the source and the destination, while congestion control runs between the source, the destination, and the network. That is, congestion in the network could be alleviated by congestion control but not flow control. In the literature, flow or congestion control mechanisms can be divided into window-based and rate-based ones. Window-based control determines the sending rate by controlling the number of outstanding data packets that can be simultaneously in transit. In contrast, a rate-based control sender directly adjusts its sending rate when receiving an explicit notification of how fast it should send. It is reasonable and possible for the network to signal one or
more senders and ask them to slow down temporarily until the network can recover from congestion. Among the above two schemes, TCP adopts the window-based flow control.

**Synchronization**

For real-time applications that require extra information to reconstruct the original play-out, extra information other than the above should be available. They include synchronization between audio and video streams, data compression and de-compression information, and path quality statistics (packet loss rate, end-to-end delay and its variation). We shall investigate the design issues of RTP at the end of this chapter, together with a voice over IP (VoIP) example.

**Standard Programming Interfaces**

Networking applications often access the underlying services provided by the end-to-end protocol layer, IP layer, or even link layer, through the *socket programming interfaces*. The BSD socket interface semantics has become the most widely used template for most operating systems, compared to the transport layer interface (TLI) socket and its standardized version X/Open TLI (XTI), both of which are developed for AT&T Unix Systems. We shall discuss how Linux implements the socket interface to bind applications to the different underlying protocol layers. We also show how Linux kernel integrates the socket into the general read/write function calls.

---

**Open Source Implementation 4.1: Packet Flows in Call Graphs**

The interfaces of the transport layer include that with the IP layer and that with the application layer. As in Chapter 3, we examine these two interfaces through reception path and transmission path. In the reception path, a packet is received from the IP layer and then passed to application layer protocols. In the transmission path, a packet is received from the application layer, and then passed to the IP layer.

As shown in Figure 4.2, when a packet is received from the IP layer, it will be saved in an *skb* and passed into *raw_v4_input()*, *udp_rcv()*, or *tcp_v4_rcv()*, based on its protocol. Then, each protocol has its function, e.g. *_raw_v4_lookup()*, *_udp4_lib_lookup()*, and *_inet_lookup()*, to look up the *sock* structure corresponding to the packet. By the information in the *sock* structure, the transport layer can identify which connection the incoming packet belongs to. Then, the incoming packet will be inserted into the
queue of the connection by \texttt{skb\_queue\_tail()}, and the application with this connection will be notified that data are available for receiving by \texttt{sk->sk\_data\_ready()}. Next, the application may call \texttt{read()} to get the data. The \texttt{read()} function triggers a series of function calls and finally \texttt{skb\_dequeue()} is used to take out the data from the queue corresponding to the connection into an skb space, and then \texttt{skb\_copy\_datagram\_iovec()} is called to copy the data from the kernel-space memory to the user-space one.

![Call Graph for an Incoming Packet](image)

**Figure 4.2.** The call graph for an incoming packet in the transport layer

Next, Figure 4.3 displays the call graph for an out-going packet. When an application plans to send data into the Internet, it will call \texttt{write()} and then different functions, i.e. \texttt{raw\_sendmsg()}, \texttt{udp\_sendmsg()}, and \texttt{tcp\_sendmsg()}, are called based on the protocol specified when the socket is created. For raw and udp socket, \texttt{ip\_append\_data()} is used. Then, \texttt{sock\_alloc\_send\_skb()} and \texttt{getfrag()} are called to allocate the skb buffer and copy data from the user-space to the kernel-space memory, respectively. Finally, the skb will be inserted into \texttt{sk\_write\_queue}. On the other hand, \texttt{ip\_push\_pending\_frame()} repeatedly takes data out the queue and then forwards them to the IP layer. Similarly, for TCP socket,
tcp_write_queue_tail() and skb_copy_to_page() are used to get the tail skb and copy data into the kernel-space memory, respectively. If the written data is more than the space available in the tail skb, new skb can be allocated by sk_stream_alloc_page(). Finally, ip_queue_xmit() is called to forward data from the sk_write_queue into the IP layer via ip_output().

Figure 4.3. The call graph for an outgoing packet in the transport layer

4.2 Unreliable Connectionless Transfer

For applications that only need process-to-process communication channel, i.e. addressing, and optional error control, without the need for data reliability and flow control, User Datagram Protocol (UDP) can satisfy their needs. It is due to its simplicity that UDP does not require keeping state information of each process-to-process communication channel. The sending and receiving of each segment is independent of any previous/later sent/received segments. As such UDP is often referred to as an unreliable connectionless transport protocol.
For real-time applications that require not only process-to-process channels but also timing information of each segment, the real-time transport protocol (RTP) is often built on top of UDP to provide the extra timing services. These topics are discussed herein.

4.2.1 Objectives of UDP

User Datagram Protocol (UDP) provides the simplest services: (1) process-to-process communication channel and (2) per-segment error control:

Addressing: Process-to-Process Communication Channel

To provide a process-to-process channel between any two application processes that may reside on different IP hosts in the Internet, each application process should be bound to a port number on its local host. The port number must be unique within the host. The operating system often searches an unused local port number when asked to allocate a port number to an application process. Since an IP address is globally unique, the source/destination port numbers, concatenated with the source/destination IP addresses and protocol ID (i.e., UDP here) in the IP header, form a socket pair. A socket pair uniquely identifies a flow, i.e. the globally unique process-to-process communication channel. Note that a socket pair is full-duplex, namely data can be transmitted through the connection in either direction simultaneously. In Figure 4.1, any outgoing packets from application process AP1 flow from its source port to the destination port bound by application process AP3. Data encapsulated by the five fields in application process AP1 on IP host 1 can be accurately transported to application process AP3 on the IP host 2 without ambiguity.

The header of the UDP is shown in Figure 4.4. A UDP port accepts user datagrams from a local application process, fills the 16-bit source and destination port numbers and other fields, breaks them up into pieces of no more than 64K bytes, which is usually 1472 bytes because Ethernet’s maximum transfer unit is set to 1500, and the IP header needs 20 bytes while the UDP header needs 8 bytes. Then, each piece is sent as a separate IP datagram and forwarded hop-by-hop to the destination as illustrated in Chapter 3. When the IP datagrams containing UDP data reach their destination, they are directed to the UDP port, which is bound by the receiving application process. The binding of ports to processes is handled independently by each host. However, it proves useful to attach frequently used processes (e.g., a WWW server process) to fixed sockets which are
made well known to the public. These services can then be accessed through the well-known ports.

![UDP header format](image)

Figure 4.4 UDP header format.

UDP provides a way for applications to send segments directly without having to establish a connection first. (As an analogy, we can send short messages to pagers or cellular phones without dialing a call first.) Providing such simple process-to-process channel and error control is the basic function of the end-to-end protocols. Many issues on this channel, such as data reliability, flow control and congestion control, are not covered in the UDP. TCP in section 4.3 addresses these issues. For UDP, the only extra service other than port multiplexing is per-segment error control.

**Error Control: Per-Segment Checksum**

UDP segment header provides a checksum to check for the integrity of each packet. Figure 4.4 indicates that the checksum is a 16-bit field. Since the checksum for UDP segment is an option, it can be disabled by setting the field to zero. However, for the case that the computed checksum is zero actually, 0xFFFF would be filled into the field. The checksum is generated and filled in by the sender and is to be verified by the receiver. To ensure that each received segment is exactly the same as the corresponding segment by the sender, the receiver re-calculates the checksum from received header and payload fields and verifies if it is equal to the UDP checksum field. UDP receivers will drop the packets whose checksum field is not the same as what the sender has calculated. This mechanism ensures per-segment data integrity. Note that checksum itself can only check for per-segment error but cannot provide per-segment data reliability because it requires extra retransmission mechanisms. However, UDP does not provide the retransmission mechanism.

The UDP checksum field is the 16-bit one's complement of the one's complement sum of all 16-bit words in the header and data octets to be check-summed, the
last octet is padded on the right with zeros to form a 16-bit word for checksum purposes. The pad is not transmitted as part of the segment. While computing the checksum, the checksum field itself is replaced with zeros. The checksum also covers a 96-bit pseudo header, consisting of four fields in the IP header, including the Source IP Address, the Destination IP Address, the Protocol, and UDP length. Checksum covering the pseudo header enables the receiver to detect the segments of incorrect delivery, protocol or length. The UDP checksum calculation described herein is the same as how the IP layer calculates IP checksum where input bits are treated as a series of 16-bit words. However, while IP checksum only covers IP header, UDP checksum is calculated from not only UDP header and data, but also several selected fields from the IP header. The purpose is to let UDP double-check that the data has arrived at the correct destination.

Although UDP checksum is optional, it is highly recommended because some link layer protocols, such as SLIP, don’t have any form of checksum. But why on earth should UDP checksum be optional? Omitting error control means that error control is less important in some applications. In the next subsection we discuss the real-time application where error control is less meaningful compared to the channel situations, such as the end-to-end delay and jitter, between the application processes.

TCP checksum calculation is exactly the same as that of UDP. It also covers some fields in the IP header to ensure that the packet has arrived at the correct destination. While UDP checksum is optional, TCP checksum is mandatory. However, although both protocols provide checksum field for data integrity, the checksum is quite a weak check from the modern standard, compared to a 32-bit cyclic redundancy check.

Open Source Implementation 4.1: UDP and TCP Checksum

The flowchart of checksum calculation, together with IP checksum, in Linux 2.6 can be learnt by tracing code from the function tcp_v4_send_check() in tcp_ipv4.c. TCP checksum flowchart is exactly the same as that of UDP. Figure 4.5 lists the partial code in tcp_v4_send_check(). The application data is first check-summed into skb->csum, and then skb->csum is check-summed by csum_partial() again with the transport layer header, indicated by pointer th. The calculated result is again check-summed with the IP source and destination address in the IP header by tcp_v4_check(), which wraps csum_tcpudp_magic(), and the result is stored in the TCP/UDP checksum field. On the other hand, the
IP checksum is independently computed from the IP header, which could be found in `net/ipv4/af_inet.c` by searching the term `iph->check`.

```c
th->check = tcp_v4_check(len, inet->saddr, inet->daddr,
    csum_partial((char *)th,
    th->doff << 2,
    skb->csum));
```

Figure 4.5 The partial code for the checksum procedure of TCP/IP.

According to the above description, the flowchart of checksum calculation is plotted in Figure 4.6. We can summarize several findings from the figure: (1) the transport layer checksum is calculated from the checksum of application data; (2) the IP checksum does not cover any field within the IP payload. In Figure 4.6, D stands for the pointer to the application data, lenD for the length of application data, T for the pointer to the transport layer header (TCP or UDP), lenT for the length of transport layer header, lenS for the length of the segment (including the segment header), iph for the pointer to the IP header, SA for source IP address, and DA for the destination IP address.

![Checksum calculations of TCP/IP headers in Linux 2.6.](image)

**Figure 4.6 Checksum calculations of TCP/IP headers in Linux 2.6.**

### 4.2.2 Carrying Unicast/Multicast Real-Time Traffic

Due to its simplicity, UDP is the most suitable carrier for unicast/multicast real-time traffic. Because real-time traffic has the following properties: (1) it does not need per-flow reliability (retransmitting a lost real-time packet is meaningless because the packet may not arrive in time), and (2) its bit-rate (bandwidth) depends mainly on the selected codec and is less likely to be flow controllable. UDP is simple enough to meet these two requirements. These two properties simplify the end-to-end protocol layer for real-time traffic to offer only the port-multiplexing service.
Though real-time traffic requires only the addressing service, i.e. the process-to-process communication channel, it could work better when some specific services are available. They include synchronization between audio and video streams, data compression and de-compression information, and path quality statistics (packet loss rate, end-to-end delay and its variation). These services may help to enhance the quality of the playback. We shall investigate the design of a standard real-time protocol, Real-time Transport Protocol (RTP), built on top of UDP in Section 4.5.

### 4.3 Reliable Connection-Oriented Transfer

The majority of networking applications today use Transmission Control Protocol (TCP) to communicate because it can provide a reliable channel. Furthermore, it can automatically adjust its sending rate to adapt to network congestion or the receiving capability of the receiver.

#### 4.3.1 Objectives of TCP

TCP can provide (1) process-to-process communication channels, (2) per-segment error control, (3) per-flow data reliability, (4) per-flow flow control and congestion control. Port-multiplexing and per-segment error control services are the same as those in UDP. Since the latter two concerns per-flow issues, we first discuss how a TCP flow is established and released in Subsection 4.3.2, then data reliability and flow/congestion control of TCP are illustrated in Subsection 4.3.3 and Subsection 4.3.4, respectively.

#### 4.3.2 Connection Management

Connection management deals with the process of connection establishment and disconnection. Each connection is uniquely specified by a *socket pair* identifying its two sides. Just like dialing a phone, we must pick up the phone. Then, we choose the phone number (IP address) and its extension (port number). Next, we dial to the party (issuing a connection request), wait for response (connection establishment), and begin to speak (transferring data). Finally, we say goodbye and close the connection (disconnection). The TCP protocol, though similar to dialing a phone, should be as formal as possible to avoid ambiguity. The details of connection establishment, disconnection and TCP state transition are described herein.

Establishing a connection sounds easy, but it is actually surprisingly tricky. This is due to the fact that the Internet occasionally lose, store, and duplicate...
packets. Storing packets in the Internet introduces delay and duplication that can confuse a sender or a receiver. It is very complicated to resolve the ambiguities if packets can live forever in the network. TCP chose to restrict the maximum life time of a packet to 120 seconds. Under this agreement, TCP can employ the Three-Way Handshake protocol proposed by Tomlinson in 1975 to resolve the ambiguities caused by the delayed duplicate packets.

**Connection Establishment/Termination: Three-Way Handshake Protocol**

When a client process wants to request a connection with a server process, as shown in Figure 4.7(a), it sends a SYN segment specifying the port number of the server that the client wants to connect to. The server responds a segment with SYN and ACK bits set to acknowledge the request. Finally, the client process must also acknowledge the SYN from the server process to initiate the connection. Note that the sequence numbers and acknowledgement numbers must follow the semantics depicted in Figure 4.7(a) to notify the *Initial Sequence Number* (ISN) of each direction. ISN is randomly chosen at connection startup by both sides to reduce the ambiguous effects introduced by the delayed duplicate packets. This protocol is known as the *three-way handshake protocol*.

Different from the connection establishment, the TCP connection termination takes four segments rather than three. As shown in Figure 4.7(b), it is a two-way handshaking for each direction. A TCP connection is *full-duplex*, namely, data flowing from client to server or server to client is independent with each other. Thus, closing one direction (sending a FIN) does not affect the other direction. The other direction should also be closed by issuing a FIN segment.

![Figure 4.7 Handshake protocols for TCP connection establishment and termination.](image)

The party that sends the first SYN is said to perform an *active open*, while
its peer is said to perform a passive open. Similarly, the party that sends the first FIN is said to perform an active close and its peer performing a passive close. Their detailed differences can be illustrated by the TCP state transition diagram described next.

TCP State Transition

A connection progresses through a series of states during its lifetime. The eleven states are: LISTEN, SYN-SENT, SYN-RECEIVED, ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT, and the fictional state CLOSED. CLOSED is fictional because it represents the state when the connection is terminated. Briefly the meanings of the states are:

- **LISTEN** - waiting for a connection request from any remote TCP client.
- **SYN-SENT** - waiting for a matching connection request after having sent a connection request.
- **SYN-RECEIVED** - waiting for a confirming connection request acknowledgment after having both received and sent a connection request.
- **ESTABLISHED** - an open connection, data can be sent in both directions. The normal state for the data transfer phase of the connection.
- **FIN-WAIT-1** - waiting for a connection termination request from the remote TCP, or an acknowledgment of the connection termination request previously sent.
- **FIN-WAIT-2** - waiting for a connection termination request from the remote TCP.
- **CLOSE-WAIT** - waiting for a connection termination request from the local user.
- **CLOSING** - waiting for a connection termination request acknowledgment from the remote TCP.
- **LAST-ACK** - waiting for an acknowledgment of the connection termination request previously sent to the remote TCP.
- **TIME_WAIT** - waiting for enough time before transitioning to a closed state to ensure the remote TCP received its last ACK.

As defined in RFC 793, a TCP sender or a receiver works by running a state machine as shown in Figure 4.8. Both TCP sender and receiver employ this state transition diagram. Bold arrows and dashed arrows correspond to
normal state transitions of client and server, respectively. Readers are encouraged to trace the detailed state transitions with RFC 793. In the following we focus on the main theme of TCP: data reliability and flow control.

4.3.3 Reliability of Data Transfers

TCP uses checksum for per-segment error control and uses acknowledgement for per-flow data reliability. Their differences in objectives and solutions are described herein.

Per-Segment Error Control: Checksum

TCP checksum is the same as that of UDP as described in Section 4.2.1. It also covers some fields in the IP header to ensure that the packet has arrived at the correct destination.

Per-Flow Data Reliability: Sequence Number and Acknowledgement

Per-segment checksum is inadequate to guarantee that the whole transferred data of a process-to-process channel can safely reach the destination. Since the packetized data sent out sequentially may get lost occasionally in the Internet, there must be a mechanism to retransmit the lost ones. Moreover, because packets sent in sequence may get received out of order due to the nature of Internet, another mechanism must exist to re-sequence the out-of-order packets. These two mechanisms rely on
acknowledgements (ACKs) and sequence number, respectively, to provide per-flow reliability.

Conceptually, each octet of data is assigned a sequence number. Then, the sequence number of a segment is just the sequence number of its first octet, which is stored in the 32-bit sequence number header field of the segment. Then, on receiving a data segment, a TCP receiver replies an acknowledgement segment whose TCP header carries an acknowledgment number indicating the next expected segment sequence number. The TCP sender numbers its sent octets and waits for their acknowledgements. The TCP receiver acknowledges the successfully received segment by replying an ACK=$x$, where $x$ indicates: “The next expected segment’s sequence number is $x$. Send it to me.”

Note that the ACK is cumulated acknowledgement, indicating that all previous data octets prior to the specified ACK number have been successfully received. This leads to an interesting situation when packets are received out-of-sequence at their destination: the receiver replies duplicate ACKs upon receiving new data segments if there are missing segments at the receiver. In the following sections we shall see that the sender treat consecutive triple duplicate ACKs (TDA) as an evidence of packet loss.

### 4.3.4 TCP Flow Control

TCP employs window-based flow/congestion control mechanisms to determine how fast it should send in various conditions. By flow/congestion control the TCP sender can know how much resource it can consume without overflowing its receiver’s buffer (called flow control), and without overburdening the globally shared network resources (called congestion control). As shown in Section 4.1 and Table 4.1, this issue had appeared and been resolved in single-hop channels, i.e. links in Chapter 2, but the difference is that the propagation delay in the end-to-end environments varies dramatically. The delay distribution becomes so diffused that the TCP source needs to be intelligent and dynamic enough to maximize the performance while being polite to other senders and its receiver’s buffer space.

### Sliding-Window Flow Control

The window-based flow control employs the sliding window mechanism. In Figure 4.9, in order to send the segmented byte-stream data in sequence, the window only slides from left to right; in order to control the amount of outstanding segments in transit, the window augments and shrinks
dynamically as shown in Figure 4.10. In Figure 4.9, as the data segments flow towards the destination, the ACK segments flow backwards to the sender to trigger the sliding of the window. Whenever the window covers the segments that have not been transmitted, the segments are kicked out to the network pipe.

![Figure 4.9 Visualization of a TCP Sliding Window](image)

**Augmenting and Shrinking of Window Size**

Another important issue of the sliding window flow control is the window size. The window size is determined by the minimum of two window values: *receiver window* (RWND) and *congestion window* (CWND), as illustrated in Figure 4.10. A TCP sender always tries to simultaneously consider its receiver's capability (RWND) and network capacity (CWND) using \( \min(RWND, CWND) \) to constrain its sending rate. The RWND is advertised by the receiver while CWND is computed by the sending host as will be explored in the next subsection. Note that the window is counted in bytes rather than number of packets. A TCP receiver advertises the amount of bytes available in its buffer while a TCP sender infers the amount of bytes in units of Maximum Segment Size (MSS) allowed to be in the network.
Open Source Implementation 4.3: TCP Sliding Window Flow Control

To write packets onto the network, Linux 2.6 kernel implements the `tcp_write_xmit()` routine in `tcp_output.c` to advance the `send_head`. Therein it does check if it can kick out anything by consulting the `tcp_snd_test()` routine in which the kernel does several tests. First, the kernel judges whether the number of outstanding segments, including normal and retransmitted segments, is more than the current network's capacity (`cwnd`) by `tcp_packets_in_flight() < tp->snd_cwnd`. Secondly, the kernel determines whether the latest sent segment has been beyond the limit of the receiver's buffer by `after(TCP_SKCB(skb))->end_seq, tp->snd_una + tp->snd_wnd)`. The “after(x,y)” routine is a Boolean function corresponding to the “x>y”. If the latest sent segment (`end_seq`) has already been beyond the unacknowledged octet (`snd_una`) plus the window size (`snd_wnd`), the sender should stop sending. Thirdly, the kernel performs the Nagle’s test by `tcp_nagle_check()` which will be addressed in Subsection 4.3.7. Only if the segment passes these checks can the kernel call the `tcp_transmit_skb()` routine to kick out one more segment.

Another interesting behavior we can learn from this implementation is that Linux 2.6 kernel has the finest granularity in sending out the segments within the window size. This is because it emits only one segment upon passing all the above tests and repeats all the tests for the next segment to be sent. If any window augmenting or shrinking happen during the period of sending out segments, the kernel can immediately control the number of segments on the network. However, it introduces larger overhead because it sends a segment at a time.

4.3.5 TCP Congestion Control

A TCP sender is designed to infer network congestion by detecting segment loss events. Hereby the sender politely slows down its transmission rate to release its occupied resources, i.e. bandwidth, to others. This process is called congestion control, which alleviates network congestion while achieving efficient resource utilization. Broadly speaking, the idea of TCP congestion control is for each TCP sender to determine how much capacity is available in the path of the network, so it knows how many segments can be in transit safely. Internet congestion control can be done by TCP senders or by the network. Network-based congestion control often employs various queuing
disciplines, scheduling algorithms, or even artificial TCP window sizing, at intermediate routers, to avoid network congestion. Sender-based congestion control relies on the self-control of each TCP sender not to send too much data to the network. Network-based congestion control is beyond the scope of this chapter and shall be addressed in Chapter 6.

**From basic TCP, Tahoe, Reno to NewReno, SACK/FACK, Vegas**

The TCP protocol has evolved for over two decades and many versions of TCP have been proposed to elevate transmission performance. The first version of TCP, standardized in RFC 793 (1981), defines the basic structure of TCP, i.e., the window-based flow control scheme and a coarse-grain timeout timer. Note that the RFC 793 does not define congestion control mechanisms because the Internet traffic is not so much as that nowadays. TCP congestion control was introduced into the Internet in the late 1980’s by Van Jacobson, roughly eight years after the TCP/IP protocol suite had become operational. At that time, the Internet had begun suffering from congestion collapse – hosts would send their packets into the Internet as fast as the receiver’s advertised window would allow, congestion would occur at some routers causing packets to be dropped, and the hosts would timeout and retransmit the lost packets, resulting in even more serious congestion. The second version, TCP Tahoe (release in BSD 4.2 in 1988), added the congestion avoidance scheme and the fast retransmission proposed by Van Jacobson. The third version, TCP Reno, extended the congestion control scheme by including fast recovery scheme. Reno was standardized in RFC 2001 and generalized in RFC 2581. TCP Reno had become the most popular version after year 2000. Recently, TCP NewReno has become the majority in a recent report.

Several shortcomings exist in TCP Reno. First, the multiple-packet-loss problem is that Reno often causes a timeout and results in low utilization when multiple segments are lost in a short interval. NewReno, SACK (Selective ACKnowledgement, defined in RFC 1072) and Vegas seek to resolve this problem with three different approaches. The TCP FACK (Forward ACKnowledgement) version then further improves the TCP SACK version. We first examine the basic TCP congestion control version – TCP Reno. Further improvements through NewReno, SACK, FACK, Vegas are discussed in Subsection 4.3.8.
Modern Computer Networks: An open source approach

TCP Reno Congestion Control

Reno uses a congestion window (cwnd) to control the amount of transmitted data in one round-trip time (RTT) and a maximum window (mwnd) to limit the maximum value of cwnd. Reno estimates the amount of outstanding data, awnd, as snd.nxt – snd.una, where snd.nxt and snd.una are the sequence numbers of the next un-sent data and un-acknowledged data, respectively. Whenever awnd is less than cwnd, the sender continues sending the new packets. Otherwise, the sender stops. The control scheme of Reno can be divided into five parts, which are schematically depicted in Figure 4.11 and interpreted as follows:

Sidebar – Historical Evolution: Statistics of TCP Versions

TCP NewReno does gradually become the major version of TCP in the Internet. According to an investigation report from Internet Computer Science Institute (ICSI), among all the 35,242 Web servers successfully identified in the report, the percentage of servers using NewReno TCP is increased from 35% at 2001 to 76% at 2004. Besides, the percentage of servers supporting TCP SACK also increases from 40% at 2001 to 68% at 2004. Furthermore, TCP NewReno and SACK are enabled in several famous OS, such as Linux, Windows XP, and Solarios. Contrasted to the increasing usage of NewReno and SACK, the percentage of TCP Reno and Tahoe are decreased to 5% and 2%, respectively. One of the reasons that TCP NewReno and SACK can be deployed fast is that it is useful to improve the throughput of a connection when it passes the Internet and encounters packet losses. Therefore, either the OS developers or the end users are willing to take it as their transport protocol.
Figure 4.11 TCP Reno congestion control algorithm.

(1) **Slow-start**: The slow-start stage aims at fast probing available resource (bandwidth) within a few RTTs. As a connection starts or after a retransmission timeout occurs, the slow-start state begins. The initial value of $cwnd$ is set to one packet, i.e. MSS, in the beginning of this state. The sender increases $cwnd$ exponentially by adding one packet each time it receives an ACK. So the $cwnd$ is doubled (1, 2, 4, 8, etc.) with each RTT as shown in Figure 4.12. Thus, slow-start is not slow at all. Slow-start controls the window size until $cwnd$ achieves the slow-start threshold ($ssthresh$), and then the congestion avoidance state begins. Note that the $ssthresh$ is initially set to the maximum value of the $ssthresh$ (which depends on the data type to store $ssthresh$) when a connection starts so as not to limit the slow-start’s bandwidth-probing.
Figure 4.12 Visualization of packets in transit during slow start.

(2) **Congestion avoidance:** Congestion-avoidance aims at *slowly* probing available resource (bandwidth) but *rapidly* responding to congestion events. It follows the *Additive Increase Multiplicative Decrease (AIMD)* principle. Since the window size in the slow-start state expands exponentially, the packets sent at this increasing speed would quickly lead to network congestion. To avoid this, the congestion avoidance state begins when $cwnd$ exceeds $ssthresh$. In this state, $cwnd$ is added by $1/cwnd$ packet every receiving an ACK to make the window size grow *linearly*. As such the $cwnd$ is normally incremented by one with each RTT (by $1/cwnd$ with each receiving ACK) but *halves* itself within only one RTT when congestion occurs. Figure 4.13 depicts the behavior of additive increase.
Open Source Implementation 4.4: TCP Slow Start and Congestion Avoidance

The slow start and congestion avoidance in tcp_cong.c of Linux 2.6 kernel are summarized in Figure 4.14. Note that in the congestion avoidance the adding of cwnd on every receiving ACK is simplified by adding a full-size segment (MSS bytes) upon receiving all ACKs of cwnd segments.

```c
if (tp->snd_cwnd <= tp->snd_ssthresh) { /* Slow start*/
    if (tp->snd_cwnd < tp->snd_cwnd_clamp)
        tp->snd_cwnd++;
} else {
    if (tp->snd_cwnd_cnt >= tp->snd_cwnd) { /* Congestion Avoidance*/
        if (tp->snd_cwnd < tp->snd_cwnd_clamp)
            tp->snd_cwnd++;
        tp->snd_cwnd_cnt=0;
    } else
        tp->snd_cwnd_cnt++;
}
```

Figure 4.14 TCP slow start and congestion avoidance in Linux 2.6.

*tp* is the pointer to the tcp_opt structure, which contains *snd_cwnd*, *snd_ssthresh* for storing congestion window and slow-start threshold, *snd_cwnd_cnt* for simplifying the congestion avoidance implementation without having to add 1/cwnd packet on receiving each ACK, and *snd_cwnd_clamp* for limiting the congestion window (non-standard).
(3) **Fast retransmission:** Fast-retransmit targets at immediately transmitting the lost packet without waiting for a timer to expire. As shown in Subsection 4.3.3, the duplicate ACK is caused by an out-of-order packet received at the receiver and is treated by the sender as a signal of a packet delay or a packet loss. If three or more duplicate ACKs are received in a row, packet loss is likely. The sender performs retransmission of what appears to be the missing packet, without waiting for a coarse-grain timer to expire.

(4) **Fast recovery:** Fast-recovery concentrates on preserving enough outstanding packets in the pipe to retain TCP’s *self-clocked* behavior. When fast retransmission is performed, $ssthresh$ is set to half of $cwnd$ and then $cwnd$ is set to $ssthresh$ plus three (the three duplicate ACK has exited the pipe). The $cwnd$ is added by one packet on every received duplicate ACK, representing that another packet has exited the pipe. A more correct thought is $awnd$ minus three for three duplicate ACKs, which trigger this fast retransmission. Also the $awnd$ is reduced by one on every received duplicate ACK, which represents that the receiver successfully receives a new packet. However, in Reno, the calculation of $awnd$ is $snd.nxt - snd.una$, which is fixed in this state. Hence Reno increases $cwnd$, rather than reducing $awnd$, to achieve the same purpose. When the ACK of the retransmitted packet is received, $cwnd$ is set to $ssthresh$ and the sender re-enters the congestion avoidance. In other words, $cwnd$ is reset to half of the old value of $cwnd$ after fast recovery.

(5) **Retransmission timeout:** Retransmission timeout provides the last resort to retransmit the lost packet. The sender maintains a retransmit timer, which is used to check for timeout of awaiting an acknowledgement that can advance the left edge of the sending window. If a timeout occurs, the sender resets the $cwnd$ to one and restarts from slow-start. The timeout value highly depends on the RTT and the variance of the RTT. The more fluctuating the RTT is measured, the larger the timeout value should be kept so as not to retransmit an already arrived segment; the more stable the RTT is measured, the closer the timeout value can be set against the RTT in order to rapidly retransmit the lost segment. As such TCP adopts a highly dynamic algorithm proposed by Van Jacobson in 1988 that constantly adjusts the timeout interval based on continuous measurements of RTT to be discussed in Subsection 4.3.7. However, one problem encountered by the dynamic estimation of RTT is what to do when a segment times out and is sent again. When an acknowledgement comes in, it is unclear whether the acknowledgement refers to the first transmission or a later one. Wrong
guess could seriously contaminate the estimation of RTT. Phil Karn discovered this problem in 1987 and proposed not to update RTT on any segments that have been retransmitted. Instead, the timeout is doubled on each failure until a segment gets through on the first time. This fix is known as the Karn’s algorithm.

Although Reno is the most popular TCP version to date, it has the multiple-packet-loss problem that degrades its performance. We will further investigate the problem and its solutions in Subsection 4.3.6.

**Open Source Implementation 4.5: TCP Congestion Control Behaviors**

Linux 2.6 is a joint implementation of various TCP versions, including NewReno, SACK, and FACK that will be studied in Subsection 4.3.8. However, the basic behavior (under one packet loss scenario) is much the same with that of Reno. Figure 4.15 displays an example snapshot of congestion control of Linux 2.6. It is generated by processing the kernel logging of window size and the sniffed packet headers.

In Figure 4.15 (a) the cwnd rapidly grows beyond the figure boundary using slow-start before congestion occurs at 1.45 second. However, note that the rwnd almost remains at 21 packets all the time such that the sending rate is bounded by 21 packets/RTT between 0.75 and 1.45 second as shown in Figure 4.15 (b). This is because the actual window size is determined by the minimum of the cwnd and rwnd. As such the cwnd from 0.75 to 1.45 second grows with somewhat a less aggressive behavior than that of 0-to-0.75 second. The result is caused by the fact that the rate of incoming ACKs is fixed during the 0.75 to 1.45 second. From 0.75 to 1.45 second, the full-duplex network pipe is constantly filled up with 21 packets where about half of them are ACKs if the network’s forward path and reverse path are symmetric.

When the congestion occurs at 1.5 second, the triple duplicate ACKs trigger the fast retransmission to retransmit the lost segment. The TCP source hereby enters the fast recovery stage, resetting the ssthresh to \( \min(cwnd, rwnd) = 10 \) and cwnd to ssthresh+3. During the fast recovery the TCP sender increments the cwnd by one MSS whenever receiving more duplicate ACKs to remain enough segments in transit. The fast recovery stage ends at 1.7 second when the lost segment is recovered. At this moment, cwnd is reset to what ssthresh contains (previously set to 10) and increment itself using the congestion avoidance algorithm. After that, the cwnd is incremented by one MSS when receiving all ACKs of the sliding window.
4.3.6 TCP Header Format

In this subsection we examine other fields of the TCP header in Figure 4.16 that we have not mentioned so far. As indicated in Subsection 4.2.1, a TCP socket contains a 16-bit source port number, a 16-bit destination port number, a 32-bit sequence number, and a 32-bit acknowledgement number. These fields are carried in the TCP segment header to transmit over the network. The sequence number corresponds to the first data octet in this segment (except when SYN is present). If SYN is present, the sequence number is the Initial Sequence Number (ISN) and the first data octet is ISN+1. If the ACK control bit is set, the acknowledgement number field contains the value of the next sequence number that the sender of the ACK segment is expecting to receive. Following the acknowledgement number is a 4-bit header length field. It indicates the number of 32-bit words in the TCP header, including the TCP options. Technically, it also implies where the application data begin. The 16-bit window in Figure 4.16 is used only when the segment is an acknowledgement (has the ACK control bit set). It specifies the number of data octets beginning with the one indicated in the acknowledgement field which
the sender of this segment, i.e. the TCP receiver, is willing to accept. This value depends on the socket buffer size and the speed of the data receiving end. The socket buffer size can be programmed using `setsockopt()` socket API.

The header length field is followed by the 6-bit control bits. The first bit is the URG bit. It is set to 1 to indicate that the 16-bit Urgent pointer field is in use. The pointer is used to indicate a byte offset from the sequence number field where the payload data begins, i.e. the byte offset right after the urgent data. This mechanism facilitates the in-band signaling of a TCP connection. For example, users can use Ctrl+C to trigger an urgent signal to cancel an operation performing on the peer end. The next bit comes the ACK bit which specifies that the acknowledgement number field is valid. If the ACK bit is not set, the acknowledgement field is ignored. The following control bit is the PSH bit whose job is to notify the receiver of the PSH-set packet to push out its data in its buffer immediately without waiting for sufficient application data to fill the buffer. The next bit is RST which is used to reset a connection. Any host with a RST-set packet received should immediately close the socket pair associated with the packet. The next bit, SYN bit, is employed to initialize a connection as shown in Subsection 4.3.2. The last bit, FIN, as illustrated in Subsection 4.3.2, is to indicate that no more data will be from the sender and both sides can close the connection.

The TCP header, along with options which will be discussed, must be an integral number of 32 bits long. Variable padding bits is appended to ensure that the TCP header ends and data begin on a 32-bit boundary. The padding is composed of zeros.

![Figure 4.16 TCP header format.](image-url)
TCP Options

Options may occupy space at the end of the TCP header and are multiple of 8 bits in length. All options are included in the checksum. An option may begin on any octet boundary. The option-length counts the two octets of option-kind and option-length as well as the option-data octets. Note that the list of options may be shorter than the data offset field might imply. The content of the header beyond the End-of-Option option must be header padding (i.e., zeros). Currently defined options include end-of-option-list, no-operation, maximum-segment-size, window-scale-factor, and timestamp options. Figure 4.17 depicts their formats.

![Figure 4.17 TCP options.](image)

The end-of-option option code indicates the end of the option list. This might not coincide with the end of the TCP header according to the Data Offset field. This is used at the end of all options, not the end of each option, and need only be used if the end of the options would not otherwise coincide with the end of the TCP header. The no-operation option code may be used between options, for example, to align the beginning of a subsequent option on a word boundary. There is no guarantee that senders will use this option, so receivers must be prepared to process options even if they do not begin on a word boundary.

If the Maximum Segment Size (MSS) option is present, then it communicates the maximum receive segment size at the TCP which sends this segment. This field must only be sent in the initial connection request (i.e.,
in segments with the SYN control bit set). If this option is not used, any
segment size is allowed.

The 32-bit sequence number will be run out if the transferring size is
larger than $2^{32}$ bytes. Normally this may not be a problem because the
sequence number can wrap around. However, in high speed networks the
sequence number may wrap around very fast such that the wrapped-around
sequence numbers may be confusing. Thus, the Protection Against Wrapped
Sequence number (PAWS) is required to avoid the side effect. Would it be
possible to send so fast? Yes. If the TCP window scaling option is used, the
TCP receiver can advertise a very large window size by negotiating a shift
count with the sender to interpret the scale of window size. In such
environments the sender can send very fast. So in order to enforce PAWS, the
TCP timestamp option is used to attach a timestamp to each segment sent.
The receiver will copy the timestamp value to its corresponding ACK so that
the segments with wrapped around sequence number can be recognized
without confusing the RTT estimator.

TCP SACK option is used to improve the performance in the fast recovery
stage of TCP congestion control. The option contains two fields indicating the
start and end of the sequence numbers of consecutively received segments.
TCP SACK will be studied in details in Subsection 4.3.8.

4.3.7 TCP Timer Management

Each TCP connection keeps a set of timers to drive its state machine in
Figure 4.9 and Figure 4.10 to work even when there is no incoming packet to
trigger the transitions of states. In this subsection, we are to study two
mandatory timers, i.e. the retransmit and persistence timers, and one optional
timer, i.e. the keepalive timer, in details. These timers are implemented in
different ways among operating systems due to the concern of performance.

(1) TCP Retransmit Timer

The role of the TCP retransmit timer has been introduced in Subsection
4.3.5 and this section studies the internal design of the RTT estimator. The
estimator adopts the Exponential Weighted Moving Average (EWMA), which
takes $1/8$ of the new RTT measure but $7/8$ of the old smoothed RTT value to
form the new estimate of the RTT. The “8” is the exponential value of 2 so that
this operation can be done with simply a three-bit shift instruction. The “moving
average” indicates that this calculation is based on a recursive form of average.
Similarly, the new mean deviation is calculated from 1/4 of the new measure and 3/4 of the previous mean deviation. The "4" can just be implemented with a two-bit shift instruction.

**Open Source Implementation 4.6: TCP Retransmit Timer**

In the literature, the default value of clock used for the round-trip ticks is 500ms, i.e., the sender checks for a timeout every 500ms. Retransmission timeout can severely degrade the TCP performance if the timer granularity is as coarse as 500ms. Linux 2.6 keeps a fine-grained timer.

When there is an incoming ACK from the IP layer, it is passed to the tcp_ack() function in tcp_input.c, in which it updates the sending window by the tcp_ack_update_window() function, sees if anything can be taken off the retransmission queue by the tcp_clean_rtx_queue() function, and sees whether or not to adjust the cwnd accordingly by the tcp_cong_avoid() function. The tcp_clean_rtx_queue() function updates several variables and invokes tcp_ack_update_rtt() to update the RTT measurements. If the timestamp option is used, the function always calls tcp_rtt_estimator() to calculate the smoothed RTT, as will then be described in Figure 4.18, and use the smoothed RTT to update the Retransmission TimeOut (RTO) value using tcp_set_rto() function. If no timestamp option is used, the above updates will not be executed when the ACK is acknowledging a retransmitted segment (the Karn’s algorithm mentioned in Subsection 4.3.5).

The contents of the tcp_rtt_estimator(), as shown in Figure 4.18, follows Van Jacobson’s suggestion in 1988 (and his further refinement in 1990) to compute a smoothed RTT estimate. Note that srtt and mdev are scaled versions of RTT and mean deviation so as to calculate the result as fast as possible. RTO is initialized to 3 seconds defined in RFC 1122 and will vary within 20 ms to 120 seconds during the connection. These values are defined in net/tcp.h.

In Figure 4.18, m stands for the current measured RTT measurement, tp is the pointer to the tcp_opt data structure, as will be seen in Open Source Implementation 4.6, mdev refers to mean deviation, and srtt represents the smoothed RTT estimate. >>3 is equivalent to division by 8 while >>2 division by 4.
(2) TCP Persistence Timer

The TCP persistence timer is designed to prevent the following deadlock. The receiver sends an acknowledgement with a window size of 0, telling the sender to wait. Later, the receiver updates the window, but the packet with the update is lost. Now both the sender and the receiver are waiting for each other to do something, which is a deadlock. Thus, when the persistence timer goes off, the sender transmits a probe to the receiver. The response to the probe gives the window size. If it is still zero, the persistence timer is set again and the cycle repeats. If it is nonzero, data can now be sent.

(3) TCP Keepalive Timer (non-standard)

Detecting crashed systems over TCP/IP is difficult. TCP does not require any transmission over a connection if the application is not sending anything, and many of the media over which TCP/IP is used (e.g. Ethernet) do not provide a reliable way to determine whether a particular host is up. If a server does not hear from a client, it could be because it has nothing to say, some network between the server and client may be down, the server or client’s network interface may be disconnected, or the client may have crashed. Network failures are often temporary (for example, it often takes a few minutes for new routes to stabilize when a router goes down) and TCP connections should not be dropped as a result.

Keepalive is a feature of the socket APIs that an empty packet be sent periodically over an idle connection; this should evoke an acknowledgement from the remote system if it is still up, a reset if it has rebooted, and a timeout if it is down. These are not normally sent until the connection has been idle for a
few hours. The purpose is not to detect a crash immediately, but to keep unnecessary resources from being allocated forever.

If more rapid detection of remote failures is required, this should be implemented in the application protocol. Currently most FTP and TELNET daemon applications detect if the user has been idle for a period. If yes, the daemon closes the connection.

**Open Source Implementation 4.7: TCP Persistence Timer and Keepalive Timer**

In Linux 2.6 kernel, the persistent timer is called probe timer. It is maintained by the tcp_probe_timer() routine in tcp_timer.c. The routine calls tcp_send_probe0() to send out a probe packet. The “zero” means the zero window updated by the receiver. If the receiver has a retransmission timeout, the sender will send a zero window probe segment which contains an old sequence number to trigger the receiver by replying a new window update.

The keepalive timer is maintained by the tcp_keepalive() in tcp_timer.c. The default calling period of the keep-alive timer is 75 seconds. When it fires, it checks every established connection for idle ones and emits new probes for them. The number of probes for each connection is limited to 5 in default. So if the other end crashes but not reboot, the probe-sender clears the TCP state in the tcp_keepopen_proc() routine; if the other end crashes and reboot within the 5 probes, it will reply a RST when receiving a probing packet. The probe-sender then can clear the TCP state.

**4.3.8 TCP Performance Problems and Enhancements**

Transmission styles of TCP Applications can be categorized into (1) interactive connections and (2) bulk-data transfers. Interactive applications, such as telnet and WWW, perform transactions, which consist of successive request/response pairs. In contrast, applications that use bulk-data transfers, such as downloading/uploading files using FTP or HTTP, to transfer large files. These two styles of data transmission have their own performance drawbacks, as shown in Table 4.2, if the previous mentioned TCP is used. This subsection introduces the problems and presents their solutions, if any.

<table>
<thead>
<tr>
<th>Transmission style</th>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 33 -
(1) Performance Problem of Interactive TCP: Silly Window Syndrome

The performance of window-based flow control such as TCP for interactive transactions suffers under a well-known condition called silly window syndrome (SWS). When it occurs, small packets are exchanged across the connection, instead of full-sized segments, which implies more packets are necessary for the same amount of data. Since each packet has a fixed size of header, transmitting in small packets means the bandwidth waste, which is particularly severe in WAN although may be insignificant in LAN. Take “telnet” for example, where each keystroke generates a packet and an ACK. Telneting across large-RTT WAN wastes the globally shared WAN bandwidth. However, telneting across small-RTT LAN do not have such problem because the LAN bandwidth is large

Solution to Silly Window Syndrome

The SWS condition could be caused by either end: the sender can transmit a small packet without waiting for more data from the sending application to send a full-sized packet; the receiver can advertise a small window (smaller than a full-sized packet) without waiting for more data being remove from the buffer to the receiving application.

To prevent the sender from SWS, John Nagle in 1984 proposed a simple but elegant algorithm known as Nagle’s Algorithm, which reduces number of packets when the bandwidth is saturated: don’t send a small new segment unless there is no outstanding data. Instead, small segments are gathered together by TCP and sent in a single segment when the ACK arrives. Nagle’s algorithm is elegant due to its self-clocking behavior: if the ACK comes back fast, the bandwidth is likely to be wide so that the data packets are sent fast; if the ACKs come back with a long RTT, which may mean a narrowband path, Nagle’s algorithm will reduce the number of tiny segments by sending full-size segments. On the other hand, to prevent the receiver from SWS, the solution proposed by David D Clark is used. The advertised packet would be delayed until the receiver buffer is half empty or available to a full-size segment, which thus guarantees to advertise a large window for the sender.

(2) Performance Problem of Bulk-Data Transfers

The performance of window-based flow control for bulk-data transfers is
best understood by the Bandwidth Delay Product (BDP) or the pipe size. In Figure 4.19, we can see the visualization of a full-duplex end-to-end TCP network pipe consisting of a forward data channel and a reverse ACK channel. You can imagine a network pipe as a water tube whose width and length correspond to the bandwidth and the RTT, respectively. Using this analogy, the pipe size then corresponds to the amount of water that can be filled in the water tube. If the bottleneck full-duplex channel (the slow links) is always full, we can easily derive the performance of such connections as

\[
\text{Throughput} = \frac{\text{Pipe Size}}{\text{RTT}}. \tag{4.1}
\]

Intuitively speaking, equation 4.1 means that the amount of the data in the pipe can be delivered in an RTT. The throughput, of course, is equal to the bandwidth of the bottleneck slow link. However, the pipe could not always be full. When a TCP connection starts, encounters packet loss, TCP senders will adjust their windows to adapt to the network congestion. Before a TCP can fill up the pipe, its performance should be derived as

\[
\text{Throughput} = \frac{\text{outstanding bytes}}{\text{RTT}} = \frac{\min(\text{CWND, RWND})}{\text{RTT}}. \tag{4.2}
\]

In a sense, if the RTT of a TCP connection is fixed, the throughput is then limited by maximum of the network capacity (pipe size), the receiver’s buffer (RWND), and the network condition (CWND).

Figure 4.19 Visualization of end-to-end, full-duplex network pipes.

Because better performance implies better effective utilization of the network pipe, the process of filling the pipe significantly influence the performance. Figure 4.20 illustrates the steps of filling a network pipe using TCP.
Figure 4.20 Steps of filling the pipe using TCP.

In Figure 4.20 (1) to (6) demonstrate the first packet sent from the left party to the right party, and the receiver replied an ACK to sender. After receiving the ACK the sender raises its congestion window to 2 in Figure 4.20 (7). This process continues as we can see in the following subfigures in Figure 4.20. After the congestion window reach 6 in Figure 4.20 (35), the network pipe becomes full.

Note that the throughput of bulk data transfer using TCP can be modeled as a function of several parameters such as RTT and packet loss rate. Evolution of this field targets on accurate prediction of a TCP source’s throughput. The major challenge lies in how we interpret previous sampled packet loss events to predict future performance. The intervals between packet losses can be independent or correlative. An easy-to-understand modeling refers to Padhye’s work which considers not only the packet loss recovered by fast retransmit algorithm but also by RTO.

In the following we will study two major performance problems encountered by bulk-data transfers: the ACK-Compression problem and the TCP Reno’s Multiple-Packet-Loss problem. Suggestions or solutions are discussed therein.

**ACK-Compression Problem**

In Figure 4.21, the full-duplex pipe only contains data stream from the sender in the left side, so the spacing between the ACKs can be a fixed clock.
rate to trigger out new data packets from the sender. However, when there are also traffic generated from the right side, as indicated in Figure 4.21, consecutive ACKs could have improper spacing because the ACKs in the reverse channel could be mixed with data traffic in the same queue. Since the transmission time of a large data packet is far beyond than that of a 64-byte ACK, the ACKs could be periodically compressed into clusters and causes the sender to emit bursty data traffic, resulting in rapid queue fluctuations in intermediate gateways. Since the end-to-end channel is essentially a hop-by-hop system, cross traffic in the intermediate Internet gateways can also cause this phenomenon.

![Figure 4.21 ACK-compression phenomenon.](image)

Currently there is no obvious way to cope with the ACK-Compression problem. Zhang suggested using pacing of data packets by the TCP sender rather than by solely relying on the ACK-clocking to alleviate the phenomenon. The clocking of ACKs has shown to be unreliable in Figure 4.21.

**TCP Reno’s Multiple-Packet-Loss Problem**

In Reno, when packet losses occur within one window, since the receiver always responds with the same duplicate ACK, the sender understands at most one new loss per RTT. Thus, in such a case, the sender must spend numerous RTTs to handle these losses. As well, a retransmission timeout commonly occurs because only few packets, which are limited owing to reduced cwnd, can be sent. For example as depicted in Figure 4.22, the ACK of packet 30 was received and the sender transmitted packets 31 to 38. For clarity, the acknowledgement number in the ACK packet is the sequence number of the received packet, rather than the sequence number of the next packet the receiver wants to receive.
Assume that \( cwnd \) is equal to 8 packets and packets 31, 33 and 34 were lost during transmission. Since packets 32, 35, 36, 37, and 38 were received, the receiver sent five duplicate ACKs. The sender discerns that packet 31 was lost when it receives the third duplicate ACK of packet 30, and then immediately sets \( cwnd \) to \( \lfloor 8/2 \rfloor + 3 \) packets and retransmits the lost packet. After receiving two more duplicate ACKs, the sender continues to increase \( cwnd \) by 2 and can forward a new packet 39. After receiving the ACK of packet 32, the sender exits fast recovery, enters congestion avoidance, and sets \( cwnd \) to 4 packets. Then, the sender receives one duplicate ACK of packet 32. When \( cwnd \) equals 4 and \( awnd \) equals 7(40-33), then the sender stops sending any packet, which results in a timeout. Note that not Reno does not always timeouts when losing more than one segments within a window of data. When the multiple-loss event happens on the situation that \( cwnd \) is very large, any partial ACKs may not only bring Reno out of fast recovery, but may also trigger another fast retransmit because of another triple duplicate ACKs. If too many packets lost within the RTT, causing the \( cwnd \) halving too many times in the
following RTTs such that very few segments are outstanding in the pipe to trigger another fast retransmit, Reno will time out.

To alleviate the multiple-packet-loss problem, the NewReno and the SACK (Selective ACKnowledgement, defined in RFC 1072) versions seek to resolve this problem with two different approaches. In the former, the sender continues to operate within fast recovery, rather than return to congestion avoidance, on receiving partial acknowledgements. In contrast, SACK, which was first proposed in RFC 1072, modifies the receiver behavior to report the non-contiguous sets of data, which have been received and queued, with additional SACK options attached in duplicated acknowledgements. Via the information within SACK options, the sender can retransmit the lost packets correctly and quickly. Mathis and Mahdavi then proposed Forward ACKnowledgment (FACK) to improve the fast recovery scheme in SACK. Compared to NewReno/SACK/FACK keeping on polishing the Fast Retransmission and Fast Recovery mechanisms, TCP Vegas proposed at 1995 uses the fine-grain RTT to assist in the detection of packet losses and congestions, which thus decreases the probability of the occurrence of timeout in Reno.


Herein we further detail how the Reno’s MPL problem is alleviated in NewReno, SACK, FACK, and Vegas, by using the same example as Figure 4.22.

Solution (I) to TCP Reno’s Problem: TCP NewReno

NewReno, standardized in RFC 2582, modifies fast recovery phase of Reno to alleviate the multiple-packet-loss problem. It departs from fast recovery when the sender receives the ACK, which acknowledges the latest transmitted packet before detecting the first lost packet. Within NewReno, this exited time is defined as “end point of fast recovery” and any non-duplicate ACK prior to that time is deemed a partial ACK.

Reno considers a partial ACK as a successful retransmission of the lost packet so that the sender reenters congestion avoidance to transmit new packets. In contrast, NewReno considers it as a signal of a further packet loss, thus the sender retransmits the lost packet immediately. When a partial ACK is received, the sender adjusts cwnd by deflating the amount of new data acknowledged and adding one packet for the retransmitted data. The sender remains in fast recovery until the end point of fast recovery. Thus, when
multiple packets are lost within one window of data, NewReno may recover them without a retransmission timeout.

For the same example illustrated in Figure 4.21, the partial ACK of packet 32 is transmitted when the retransmitted packet 31 in step 4 is received. Figure 4.23 illustrates the NewReno modification. When the sender receives the partial ACK of packet 32, it immediately retransmits the lost packet 33 and sets cwnd to (9-2+1) where 2 is the amount of new data acknowledged (packet 31 and 32) and 1 represents the retransmitted packet that have exited the pipe. Similarly, when the sender receives the partial ACK of packet 33, it immediately retransmits the lost packet 34. The sender exits fast recovery successfully until the ACK of packet 40 is received, and without any timeout occurring.

Solution (II) to TCP Reno’s Problem: TCP SACK

Although NewReno alleviates the multiple-packet-loss problem, the sender only learns of one new loss within one RTT. However, SACK option, proposed in RFC 1072, resolves this drawback. The receiver responds to the out-of-order packets by delivering the duplicate ACKs coupled with SACK options. RFC 2018 refines the SACK option and describes the behaviors of both the sender and receiver exactly.

One SACK option is applied to report one non-contiguous block of data, which the receiver successfully receives, by via the two sequence numbers of the first and last packets in each block. Owing to the length limitation of the TCP option, there is a maximum number of SACK options within one duplicate ACK. The first SACK option must report the latest block received, which contains the packet that triggers this ACK.

SACK adjusts awnd directly, rather than cwnd. Thus, upon entering fast recovery, cwnd is halved and fixed during this period. When the sender either sends a new packet or retransmits an old one, awnd is incremented by one. However, when the sender receives a duplicate ACK with a SACK option indicating that new data has been received, it decreased by one. Also, the SACK sender treats partial ACKs in a particular manner. That is, the sender decreases awnd by two rather than one, because a partial ACK represents two packets that have left the network pipe: the original packet (assumed to have been lost) and the retransmitted packet.
Figure 4.23 Solution (I) to TCP Reno’s problem: NewReno.
Figure 4.24 illustrates an example of SACK algorithm. Each duplicate ACK contains the information of the data blocks that were successfully received. When the sender received three duplicate ACKs, it knew that packets 31, 33 and 34 were lost. Therefore, if allowable, the sender could retransmit the lost packets immediately.

**Solution (II) to TCP Reno’s problem: TCP SACK option.**

Receiver replied partial ACKs for received retransmitted segments.

Sender exited fast recovery after receiving ACK of segment 38.

Sender received duplicate ACKs and began retransmitting the lost segments reported in the SACK options. Awnd was set to 8-3+1 (three duplicate ACKs and one retransmitted segment). Receiver replied partial ACKs for received retransmitted segments. Sender received partial ACKs, reduced awnd by two, and thus retransmitted two lost segments. Receiver replied ACKs for received retransmitted segments. Sender received ACK of segment 30 and sent segment 31-38. Receiver sent five duplicate ACKs with SACK options of segment 30. Sender received duplicate ACKs and began retransmitting the lost segments reported in the SACK options. Awnd was set to 8-3+1 (three duplicate ACKs and one retransmitted segment).

Solution (III) to TCP Reno’s Problem: TCP FACK

FACK was proposed to be an auxiliary for SACK. In FACK, the sender
uses the SACK options to determine the forward-most packet that was received. FACK estimates \( \text{awnd} \) for improved accuracy to \((\text{snd.nxt} - \text{snd.fack} + \text{retran_data})\), where \( \text{snd.fack} \) is the forward-most packet reported in the SACK options plus one and \( \text{retran_data} \) is the number of retransmitted packets after the previous partial ACK. Since the sender may have a long wait for three duplicate ACKs, FACK enters fast retransmission earlier. That is, when \((\text{snd.fack} - \text{snd.una})\) is larger than three, the sender enters fast retransmission without waiting for three duplicate ACKs.

Figure 4.25 depicts the FACK modification. The sender initiates retransmission after receiving the second duplicate ACK because \((\text{snd.fack} - \text{snd.una})\), \((36 - 31)\), is larger than 3. The lost packets can be retransmitted in FACK sooner than they can be in SACK since the former calculates \( \text{awnd} \) correctly. Thus, in figure 4.25, it is evident that the number of outstanding packets is constantly stable at four.
Open Source Implementation 4.8: TCP FACK Implementation

Linux 2.6 is a joint implementation of NewReno, SACK, and FACK. There are many FACK-related code segments but the most important part is in tcp_output.c and shown as follows:

```c
if (tcp_packets_in_flight() > snd_cwnd)
    return;

put more data on the transmission queue
```

The `tcp_packets_in_flight()` computes the `(packets_out - packets_out + retrans_out)` which indicates "packets sent once on transmission queue" minus "packets acknowledged by FACK information" plus "packets fast retransmitted".

Solution (IV) to TCP Reno’s Problem: TCP Vegas

Vegas first revise Reno in its opportunity to trigger Fast Retransmission. Once a duplicate ACK is received, Vegas determines whether to trigger Fast Retransmission by examining whether the time between the ACK and its replying Data packet is larger than a timeout timer. If yes, Vegas triggers Fast Retransmission without waiting more duplicate ACKs. Besides, if the first ACK packet after a retransmission still returns later than a time-out period, TCP Vegas will resend the packets to catch any previous lost packets.

Actually, TCP Vegas also uses the fine-grain RTT in improving the congestion control mechanisms. Compared to Reno reacting to packet losses and then decreasing rate to alleviate the congestion, Vegas intends to anticipate the congestion and then decrease rate early to avoid congestion and packet losses. To anticipate the congestion, during the connection Vegas keeps on the minimum RTT in a variable named BaseRTT. Then, by dividing cwnd with BaseRTT, Vegas learns the Expected sending rate, which the connection can use without causing any packets queued in the path. Next, Vegas compares the Expected sending rate with the current Actual sending rate, and adjusts cwnd accordingly. Let Diff = Expected – Actual and give two thresholds, a<b, defined in terms of KB/s. Then, cwnd in Vegas is increased 1 per RTT when Diff <a, decreased 1 if Diff > b, and fixed if Diff is between a and
b. Adjusting rate to keep Diff between a and b represents that the network buffer occupied by a Vegas connection on average would be a bytes at least to well utilize the bandwidth and no more than b bytes per second to avoid increasing the loading of the network. By the suggestion from Vegas’s authors, a and b are assigned to 1 and 3 times of MSS/BaseRTT, respectively.

4.4 Socket Programming Interfaces

Networking applications use services provided by underlying protocols to perform special-purpose networking jobs. For example, applications telnet and ftp use services provided by the end-to-end protocol; ping, traceroute, and arp directly use services provided by the IP layer; packet capturing applications running directly on link protocols may be configured to capture the entire packet, including the link protocol header. In this section, we shall see how Linux implements the general-purpose socket interfaces for programming the above applications.

4.4.1 What is a socket?

A socket is an abstraction of the endpoint of a communication channel. As the name suggests, “end-to-end” protocol layer controls the data communications between the two endpoints of a channel. The endpoints are created by networking applications using socket APIs with appropriate type. Networking applications can then perform a series of operations on that socket. The operations that can be performed on a socket include control operations (such as associating a port number with the socket, initiating or accepting a connection on the socket, or releasing the socket), data transfer operations (such as writing data through the socket to some peer application, or reading data from some peer application through the socket), and status operations (such as finding the IP address associated with the socket). The complete set of operations that can be performed on a socket constitutes the socket APIs (Application Programming Interfaces).

To open a socket, a programmer first calls the socket() function to initialize his/her preferred end-to-end channel. When you open a socket with the standard call sk=socket(domain, type, protocol), you have to specify which domain (or address family) you are going to use with that socket. Commonly used families are AF_UNIX, for communications bounded on the local machine, and AF_INET, for communications based on IPv4 protocols.
Furthermore, you have to specify a type for your socket and possible values depend on the family you specified. Common values for type, when dealing with the AF_INET family, include SOCK_STREAM (typically associated with TCP) and SOCK_DGRAM (associated with UDP). Socket types influence how packets are handled by the kernel before being passed up to the application. Finally, you specify the protocol that will handle the packets flowing through the socket.

The values of the parameters depend on what services your program will employ. In the next three subsections we investigate three types of socket APIs. They correspond to accessing the end-to-end protocol layer, the IP layer, and the link protocol layer, respectively, as we can see in the following open source implementation.

**Open Source Implementation 4.9: Socket Implementation in Linux 2.6**

Figure 4.26 displays the relative locations of each mentioned part of the Linux 2.6 kernel source code. General kernel socket APIs and their subsequent function calls reside in the `net` directory. IPv4 specific source codes are put separately in `ipv4` directory, as is also the case for IPv6. The BSD socket is just an interface to its underlying protocols such as IPX and INET. The currently widely used IPv4 protocol corresponds to the INET socket if the socket address family is specified as AF_INET. The dominant link-level technology – Ethernet, has its header built within the `net/ethernet/eth.c`. After that, the Ethernet frame is moved from the main memory to the network interface card by the Ethernet driver that resides in the `drivers/net/` directory. Drivers in this directory are hardware dependent because many vendors have Ethernet card products with different internal designs.
4.4.2 Binding Applications & End-to-End Protocols

The most widely used services by networking applications are those provided by end-to-end protocols such as UDP and TCP. A socket descriptor initiated by `socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)` is initialized as a UDP socket, where `AF_INET` indicates Internet address family, `SOCK_DGRAM` stands for datagram service, and `IPPROTO_UDP` indicates UDP protocol. A series of operations can be performed on the descriptor, such as those functions in Figure 4.27.

In Figure 4.27, before the connection is established, the UDP server as well as client creates a socket and use `bind()` system call to assign an IP address and a port number to the socket. Then, after a UDP server binds to a port, it is ready to receive requests from the UDP client. The UDP client may loop through the `sendto()` and `recvfrom()` to do something until it finishes its job. The UDP server continues accepts requests, process the requests, and feedback the results using `sendto()` and `recvfrom()`.

![UDP Server](image)

Figure 4.27 Socket functions for simple UDP client-server programs.

Similarly, a socket descriptor initiated by `socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)` is then initialized as a TCP socket, where `AF_INET` indicates Internet address family and `SOCK_STREAM` stands for the reliable byte-stream service. The functions to be performed on the descriptor are depicted in Figure 4.28.
Figure 4.28 Socket functions for simple TCP client-server programs.

The flowchart of the simple TCP client-server programs is a little complex due to the stateful property of TCP. It contains connection establishment, data transfer, and connection termination stages. Besides `bind()`, it calls `listen()` to allocate the connection queue to the socket and waits for connection request from the clients. The `listen()` system call expresses the willingness of the server to start accepting incoming connection requests. Each listening socket contains two queues: (1) partially-established request queue and (2) fully-established request queue. A request will first stay in the partially-established queue during the three-way handshake. When the connection is established (finished the three-way handshake), the request will be moved to the fully-established request queue. The partially-established request queue in most operation system has a maximum queue length, e.g. 5, even if the user specifies a value larger than 5. Thus, the partially-established request queue is the source of the Denial of Service (DoS) attack. If a hacker continuously sends packets with SYN bit set (initialize a three-way handshake) without finishing the three-way handshake (only sends a SYN), the request queue will be full and cannot accept new connection requests from well-behavior clients. This attack is known as SYN-flooding attack. SYN-flooding cannot be easily solved. If we shorten the three-way handshake timeout from 30 seconds to 5 seconds, we can drain the partially-established request queue more quickly. However, if the hacker increases the sending rate...
of SYN packets, the service will still be unavailable.

The listen() system call is commonly followed by the accept() system call, whose job is to dequeue the first request from the fully-established request queue to initialize a new socket pair and returns the new socket. That is, the accept() system call provided by the BSD socket results in the automatic creation of a new socket, largely different from that in the TLI sockets (see Page 6), where application must explicitly create a new socket for the new connection. Note that the original listening socket is still listening for accepting new connection request. Of course the new socket pair contains the IP address and port number of the client. The server program can then decide whether or not to accept the client.

The TCP client uses connect() API to invoke the three-way handshaking process to establish the connection. After that, the client and the server can perform byte-stream transfers in between.

Open Source Implementation 4.10: Socket Read/Write Inside out

The internals of the socket API used by simple TCP client-server programs in Linux is illustrated in Figure 4.29. Programming APIs invoked from the user-space programs are translated into sys_socketcall() kernel call and are then dispatched to their corresponding sys_*( ) calls. The sys_socket() (in net/socket.c) calls sock_create() to allocate the socket and then calls inet_create() to initialize the sock structure according to the given parameters. The other sys_*( ) call their corresponding inet_*( ) functions because the sock structure was initialized to Internet address family (AF_INET). Since read() and write() are not socket specific APIs but are commonly used by file I/O operations, their call flows follow their inode operation fields and find that the given descriptor relates to a sock structure. Subsequently they are translated into corresponding sock_read() and sock_write() functions.
The read()/write() function calls employed at client and server programs, as shown in Figure 4.29, are not socket-specific functions but are commonly used when using file I/O operations. In most UNIX systems the read()/write() functions are integrated into the Virtual File System (VFS). VFS is the software layer in the kernel that provides the file system interface to user space programs. It also provides an abstraction within the kernel which allows different file system implementations to co-exist.

In Linux 2.6, the kernel data structures used by the functions of a TCP connection as displayed in Figure 4.29 and illustrated in Figure 4.30. After the sender initializes the socket and gets the socket descriptor (assumed to be in fd[1] in the open file table), when the user-space program operates on that descriptor, it follows the arrow link to point to the file structure where it contains a directory entry f_dentry pointed to an inode structure. The inode structure can be initialized to one of various file system type information support by Linux, including the socket structure type. The socket structure contains a sock structure, which keeps network-related information and data structures from the end-to-end layer down to the link layer. When the socket is initialized as a byte-stream, reliable, connection-oriented TCP socket, the transport layer protocol information tp_pinfo is then initialized as tcp_opt structure, in which many TCP-related variables and data structures, such as congestion window snd_cwnd, are stored. The proto pointer of the sock
structure links to the `proto` structure that contains the operation primitives of the protocol. Each member of the `proto` structure is a function pointer. For TCP, the function pointers are initialized to point to the function list contained in the `tcp_func` structure. Anyone who wants to write his/her own end-to-end protocol in Linux should write follow the interface defined by the `proto` structure.

![Kernel data structures used by the socket APIs.](image)

### 4.4.3 Bypassing the Services Provided by End-to-End Protocols

Sometimes applications do not want to use the service provided by the end-to-end protocol layer, such as `ping` and `traceroute`, which directly send packets without opening a UDP or TCP socket but just use the services provided by the internetworking layer. Some applications even bypass the internetworking services and directly communicate over the node-to-node channel. For example, packet-sniffing applications, such as `tcpdump` and `ethereal`, capture raw packets directly on the wire. Such applications need to open a completely different socket compared with those of UDP or TCP. This subsection aims at exploring the programming method in Linux that can achieve the goal.
Open Source Implementation 4.11: Bypassing the End-to-End Layer

After Linux 2.0, a new protocol family called Linux packet socket (AF_PACKET) has been introduced to allow an application to send and receive packets dealing directly with the network card driver rather than the usual IP/TCP or IP/UDP protocol stack-handling. As such any packet sent through the socket can be directly passed to the Ethernet interface, and any packet received through the interface will be directly passed to the application.

The AF_PACKET family supports two slightly different socket types, SOCK_DGRAM and SOCK_RAW. The former leaves the burden of adding and removing Ethernet level headers to the kernel while the latter gives the application complete control over the Ethernet header. The protocol field given in the socket() call must match one of the Ethernet IDs defined in /usr/include/linux/if_ether.h, which represents the registered protocols that can be shipped in an Ethernet frame. Unless dealing with very specific protocols, you typically use ETH_P_IP, which encompasses all of the IP-suite protocols (e.g., TCP, UDP, ICMP, raw IP and so on). However, if you want to capture all packets, ETH_P_ALL will be used, instead of ETH_P_IP, as shown in the below example.

```c
#include "stdio.h"
#include "unistd.h"
#include "sys/socket.h"
#include "sys/types.h"
#include "sys/ioctl.h"
#include "net/if.h"
#include "arpa/inet.h"
#include "netdb.h"
#include "netinet/in.h"
#include "linux/if_ether.h"

int main()
{
    int n;
    int fd;
    char buf[2048];
    if((fd = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL))) == -1)
    {
        printf("fail to open socket\n");
        return(1);
    }
    while(1)
    {
        n = recvfrom(fd, buf, sizeof(buf),0,0,0);
        if(n>0)
            printf("recv %d bytes\n", n);
    }
    return(0);
}
```


Since the sockets of \texttt{AF\_PACKET} family have serious security implications, for example, you can forge an Ethernet frame with a spoofed MAC address, they can only be used by root.

Packet Capturing: Promiscuous Mode vs. Non-promiscuous Mode

The \texttt{AF\_PACKET} family allows an application to retrieve data packets as they are received at the network card level, but still does not allow it to read packets that are not addressed to its host. As we have seen before, this is due to the network card discarding all the packets that do not contain its own MAC address -- an operation mode called non-promiscuous, which basically means that each network interface card is minding its own business and reading only the frames directed to it. There are three exceptions to this rule:

1. A frame whose destination MAC address is the special broadcast address (FF:FF:FF:FF:FF:FF) will be picked up by any card.
2. A frame whose destination MAC address is a multicast address will be picked up by cards that have multicast reception enabled.
3. A card that has been set in promiscuous mode will pick up all the packets it senses.

Open Source Implementation 4.12: Making Myself Promiscuous

The last case of the above three exceptions is, of course, the most interesting one for our purposes. To set a network card to promiscuous mode, all we have to do is issue a particular \texttt{ioctl()} call to an open socket on that card. Since this is a potentially security-threatening operation, the call is only allowed for the root user. Supposing that ``sock'' contains an already open socket, the following instructions will do the trick:

```c
strncpy(ethreq.ifr_name,"eth0",IFNAMSIZ);
ioctl(sock, SIOCGIFFLAGS, &ethreq);
ethreq.ifr_flags |= IFF_PROMISC;
ioctl(sock, SIOCSIFFLAGS, &ethreq);
```

The \texttt{ethreq} is an \texttt{ifreq} structure defined in `/usr/include/net/if.h'. The first \texttt{ioctl} reads the current value of the
Ethernet card flags; the flags are then ORed with IFF_PROMISC, which enables promiscuous mode and are written back to the card with the second ioctl. You can easily check it out by giving the ifconfig command and observing the third line in the output.

High Performance Packet Capturing and Filtering

Packets in wired/wireless media can be captured by anyone who can directly access the transmission media. Applications that do such things are called packet sniffers, which are usually used for debugging network applications, i.e. to check whether a packet is sent out with correct header and payload. As being an application and running as a process in the user space, a packet sniffer process may not be scheduled immediately by the kernel when a packet comes, thus the kernel should buffer it in the kernel socket buffer until the packet sniffer process is scheduled. Besides, users may specify packet filters to the sniffer for capturing only the packets of interest. Performance of packet capturing may degrade when packets are filtered at user space because a huge amount of uninterested packets have to be transferred across the kernel-user space boundary. If sniffing at a busy network, such sniffers may not capture the packets in time before the packets overflow the socket buffer. Shifting the packet filters to the kernel can efficiently improve the performance.

Open Source Implementation 4.13: Linux Socket Filter

Figure 4.30 presents an example layered model for packet capturing and filtering. The tcpdump program accepts its user’s filter request through the command line parameters to capture an interested set of packets. Then tcpdump call the libpcap (portable packet capturing library) to access the appropriate kernel packet filters. In BSD systems, the Berkeley Packet Filter (BPF) performs the packet filtering in kernel. Linux did not equipped with kernel packet filtering until the Linux Socket Filter (LSF) appeared in Linux 2.0.36. BPF and LSF are very much the same except some minor differences such as user privilege to access the service. This layered structure is illustrated in Figure 4.31. The incoming packets are cloned from the normal protocol stack processing to the BPF. It then filters packets within the kernel level according the BPF instructions installed by the corresponding applications. Since only the packets passing through BPF will be directed to the user-space programs, the overhead of the data exchange between user and kernel spaces can be significantly reduced.
To employ a Linux socket filter onto a socket, the BPF instruction can be passed to the kernel by using the `setsockopt()` function, implemented in `socket.c`, and setting the parameter `optname` to `SO_ATTACH_FILTER`. The function will assign the BPF instruction to the `sock->sk_filter` illustrated in Figure 4.30. The BPF packet-filtering engine is written in a specific pseudo-machine code language inspired by Steve McCanne and Van Jacobson. BPF actually looks like a real assembly language with a couple of registers and a few instructions to load and store values, perform arithmetic operations and conditionally branch.

The filter code examines each packet on the attached socket. The result of the filter processing is an integer number that indicates how many bytes of the packet (if any) the socket should pass to the application level. This is a further advantage, since often you are interested in just the first few bytes of a packet, and you can spare processing time by avoiding copying the excess ones.

**4.5 Transport Protocols for Streaming**

End-to-end protocols mentioned so far are not well-designed to accommodate the requirements of real-time traffic. More fine mechanisms and tight conditions are necessary to carry streaming than non-real-time data over the Internet. For example, real-time traffic expects a stable available rate for
transmission without lag and needs timing information to synchronize between
the sender and the receiver. Also, the real-time traffic is more sensitive to the
interrupt possibly resulting from the mobility across different networks.
Compared to the transmission of non-real time data which almost all
requirements, e.g. reliability and congestion control, can be satisfied by a
single protocol, i.e. TCP, there is no single protocol satisfying all these
requirements of real-time traffic. Many protocols, aiming at different issues, are
proposed, evolving, and co-work with each other.

4.5.1 Issue: Multi-homing and Multi-streaming

Stream Control Transmission Protocol was introduced in RFC 3286 and
defined in RFC 4960. SCTP, like TCP, provides a reliable channel for data
transmission and uses the same congestion control algorithms. However, as
the term “stream” appeared in SCTP, SCTP additionally provides two
properties favorable to streaming applications, which are the supporting on
multi-homing and multi-streaming.

The support on multi-homing represents that even when a mobile user
moves from one network to another network, the user will not feel any interrupt
on its received streaming. To support the multi-homing property, a session of
the SCTP can be concurrently constructed by multiple connections through
different network adapters, e.g. one from Ethernet and one from wireless LAN.
Also, there is a heartbeat for each connection to ensure its connectivity.
Therefore, when one of the connections fails down, SCTP can transmit the
traffic through other connections immediately.

The support on multi-streaming represents that multiple streaming e.g.
audio and video streaming, could be concurrently transmitted through a
session. That is, SCTP can individually support ordered receiver for each
streaming and avoid the HOL blocking of TCP. In TCP, control or some critical
messages are often blocked because of a cloud of data packets queued in the
sender or receiver buffer.

Besides, SCTP also revise the established and close procedure of a TCP
connection. For example, SCTP proposed a 4-way handshake mechanism for
connection establish to overcome the DOS problem of TCP.

4.5.2 Issue: Smooth Rate Control and TCP-friendliness

While TCP traffic is still dominated the Internet, a bunch of research
results point out that the congestion control mechanism – AIMD – used in most
versions of TCP may cause the transmission rate too oscillatory to carry
streaming data with low jitter. Since AIMD is not suitable for streaming applications, developers tend to design their congestion control or even not use congestion control in their streaming transmission. Such activities are worrying by the Internet community because the bandwidth in the Internet is public sharing and there is no control mechanism in the Internet to decide how much bandwidth a flow should use in the Internet, which in the past is self controlled by TCP.

In 1989, a concept named TCP-friendly was proposed and promoted in RFC3412. The concept said that a flow should respond to the congestion at the transit state and use no more bandwidth than a TCP flow at the steady state when both received the same network conditions, such as packet loss ratio and RTT. Such a concept asks any Internet flow should use congestion control and use no more bandwidth than other TCP connections. Unfortunately, there is no answer for the best congestion control. Thus, a new transport protocol named Datagram Congestion Control Protocol (DCCP) is proposed by E. Kohler et al. DCCP allows free selection of a congestion control scheme. The protocol currently only includes two schemes, namely TCP-like and TFRC.
4.5.3 Issues: Playback Reconstruction and Path Quality Report

As Internet is a shared datagram network, packets sent on the Internet have unpredictable delay and jitter. However, real-time networking applications, such as Voice over IP (VoIP) and video conferencing, require appropriate timing information to reconstruct the playback at the receiver. The

Sidebar – Principle in Action: Streaming: TCP or UDP?

Why not TCP suitable for streaming? First, loss retransmission mechanism is tightly embedded in TCP, which may not be necessary for streaming and even increase the latency and jitter for the received data. Besides, continuous bandwidth detection may not be necessary for streaming. That is, although the estimation for available bandwidth may be necessary for streaming to select a coding rate, the streaming may not favor an oscillatory transmission rate, particularly the drastic respond to the packet losses, which originally is designed to avoid the potential successive losses. For streaming applications, they may accept and let the losses go. Therefore, since some mechanisms in TCP are not suitable for streaming, people turn to carry streaming over UDP. Unfortunately, UDP is so simple that providing no any mechanism to estimate the current available rate. Besides, for the security UDP packets are dropped mostly by the current intermediate network devices.

However, although TCP and UDP are not suitable for streaming, they are still the only two mature transport protocols in today’s Internet. Thus, most streaming data are indeed carried by the two protocols. UDP is used to carry pure audio streaming, like audio and VoIP. These streaming can be simply sent at a constant bit rate without much congestion, because their required bandwidth is usually lower than the available one in the current Internet. On the other hand, TCP is used for streaming which requires the bandwidth not always satisfied by the Internet, e.g. the mix of video and audio. Then, to alleviate the oscillatory rate of TCP, the side-effect of its bandwidth detection mechanism, large buffer are used in the receiver, which however prolong the delay. Although the delay is tolerated by the one-way application, like watching clips from YouTube, it is not by the interactive application, like video conference, which is why the researchers need to develop the smooth rate control, as introduced in Section 4.5.2.
reconstruction at the receiver requires the codec type to choose the right decoder to decompress the payload, timestamp to reconstruct the original timing in order to play out the data in correct rate, sequence numbers to place the incoming data packets in the correct order and to be used for packet loss detection. On the other hand, the senders of real-time applications also require path quality feedbacks from the receivers to react to network congestion. Additionally, in a multicast environment, the membership information requires to be managed. These control-plane mechanisms should be built in the standard protocol.

In summary, in data plane real-time applications need to concern the codec, sequence number, and timestamp; in control plane the focus is on the feedback report of end-to-end delay/jitter/loss and membership management. To satisfy these necessaries, RTP and RTCP are proposed, as introduced in the next two subsections. Note that RTP and RTCP are often implemented by applications themselves instead of by the operating system. Thus the applications can have full control over each RTP packet such as defining RTP header options themselves.

4.5.4 Standard Data-Plane Protocol: RTP

The RFC 1889 outlines a standard data-plane protocol: Real-time Transport Protocol (RTP). It is the protocol used to carry the voice/video traffic back and forth across a network. RTP does not have a well-known port, because it operates with different applications that are themselves identified with ports. Therefore it operates on a UDP port, with 5004 designated as the default port. RTP is designed to work in conjunction with the auxiliary control protocol RTCP to get feedback on quality of data transmission and information about participants in the on-going session.

How RTP Works?

RTP messages consist of a header portion and data portion. The real-time traffic is carried in the data field of the RTP packet. Note that RTP itself does not address resource management/reservation and does not guarantee quality-of-service for real-time services. RTP assumes that these properties are provided by the underlying network. Since the Internet occasionally loses and reorders packets and delays them by variable amounts of time, to cope with these impairments, the RTP header contains timestamp information and a sequence number that allow the receivers to reconstruct the timing produced by the source. With these two fields the RTP can ensure that
the packets are in sequence, determines if any packets are lost, and synchronize the traffic flows. The sequence number increments by one for each RTP data packet sent. The timestamp reflects the sampling instant of the first octet in the RTP data packet. The sampling instant must be derived from clock that increments monotonically and linearly in time to allow synchronization and jitter calculations. Notably, when a video frame is split into multiple RTP packets, all of them have the same timestamp, which is why the timestamp is inadequate to re-sequence the packets.

One of the fields included in the RTP header is the 32-bit **Synchronization Source Identifier (SSRC)**, which is able to distinguish synchronization sources within the same RTP session. Since multiple voice/video flows can use the same RTP session, the SSRC field identifies the transmitter of the message for synchronization purposes at the receiving application. It is a randomly chosen number used to ensure that no two synchronization sources use the same number within an RTP session. For example, branch offices may use VoIP gateway to establish a RTP session in between, as displayed in Figure 4.32. However, many phones are installed at each side so that the RTP session may simultaneously contain many call connections. These call connections can be multiplexed by the SSRC field. A synchronization source may change its data format, e.g., audio encoding, over time.

![Figure 4.32. RTP/RTCP example: Voice over IP gateways.](image)

**Codec Encapsulation**

To reconstruct the real-time traffic at the receiver, the receiver must know how to interpret the received packets. The payload type identifier specifies the payload format as well as the encoding/compression schemes. Payload types include PCM, MPEG1/MPEG2 audio and video, JPEG video, H.261 video streams, et al. At any given time of transmission, an RTP sender can only send one type of payload, although the payload type may change during transmission, for example, to adjust to network congestion.
The term codec stands for coder/decoder. A codec is that part of an integrated circuit that converts analog signals to digital signals and vice versa. Codecs are also known as coders. For voice, these circuits can exist in a central telephone office switch, a PC, or a VoIP gateway. When converting from analog signals to digital data, codecs exist at the end of any analog portion of a network or circuit. The flowchart inside a VoIP gateway in Figure 4.33 shows that the purpose of a codec is to derive from the analog waveform, a digital one that is an accurate facsimile of human speech. VoIP codecs consist of two parts: (1) Analog to digital (AD) converters and (2) Companders. The AD converters perform the digitization process while the companders compress the output of the analog to digital converters in order to conserve bandwidth in the telephony network. Companders compress data by (1) omitting silence and redundant information, and (2) mapping linear values to exponential values based on the loudness and frequency of the sound.

![Inside a VoIP Gateway Codec](image)

**Inside a VoIP Gateway Codec**

- **Analog to Digital Converter**
  - 16 bits
  - 8khz
- **VolP Gateway**
  - 128 kbps
- **Componder**
  - A-Law
  - u-Law
- **Digital output signal**
  - 64 kbps

Figure 4.33 Inside a VoIP gateway codec.

### 4.5.5 Standard Control-Plane Protocol: RTCP

RTCP is the control protocol designed to work in conjunction with RTP. It is standardized in RFC 1889 and 1890. In an RTP session, participants periodically emit RTCP packets to convey feedback on quality of data delivery and information of membership. RFC 1889 defines five RTCP packet types:

1. **RR**: receiver report. Receiver reports are sent by participants that are not active senders within the RTP session. They contain reception quality feedback about data delivery, including the highest packets number received, the number of packets lost, inter-arrival jitter, and timestamps to calculate the round-trip delay between the sender and the receiver. The information can be directly useful for adaptive encodings. For example, if the quality of the RTP session is found to be worse and worse, the sender may decide to switch to a low-bitrate encoding so that users may get a
smoother feel of real time transport. On the other hand, network administrators can evaluate the network performance by monitoring RTCP packets.

(2) **SR**: sender report. Sender reports are generated by active senders. Besides the reception quality feedback as in RR, SR contain a sender information section, providing information on inter-media synchronization, cumulative packet counters, and number of bytes sent.

(3) **SDES**: source description items to describe the sources. In RTP data packets, sources are identified by randomly generated 32-bit identifiers. These identifiers are not convenient for human users. RTCP SDES contain globally unique identifiers of the session participants. It may include user's name, email address or other information.

(4) **BYE**: indicates end of participation.

(5) **APP**: application specific functions. APP is intended for experimental use when new applications or features are developed.

### Open Source Implementation 4.14: RTP Implementation Resources

RTP is an open protocol that does not provide pre-implemented system calls. Implementation is tightly coupled to the application itself. Application developers have to add the complete functionality in the application layer by themselves. However, it is always more efficient to share and reuse code rather than starting from scratch. The RFC 1889 specification itself contains numerous code segments that can be borrowed directly to the applications. Here we provide some implementations with source code available. Many modules in the source code can be usable with minor modifications. The following is a list of useful resources:

- **self-contained sample code in RFC1889**.
- **vat** (http://www-nrg.ee.lbl.gov/vat/)
- **NeVoT** (http://www.cs.columbia.edu/~hgs/rtp/nevot.html)
- **RTP Library** (http://www.iasi.rm.cnr.it/iasi/netlab/gettingSoftware.html) by E.A.Mastromartino offers convenient ways to incorporate RTP functionality into C++ Internet applications.

### 4.6 Pitfalls and Fallacies
**Throughput vs. Goodput (Effective Throughput)**

Readers should carefully distinguish the difference between throughput and goodput. Throughput stands for utilization of the resource, including the resource consumed by the retransmitted packets. Goodput only includes effective throughput, which excludes the wasted bandwidth. For example, the Ethernet LAN may be very busy that its resource utilization reaches 100%. However, most transmitted Ethernet frames collide with each other and will be retransmitted again and again until successfully transmitted. Therefore, the effective throughput may be much lower than the throughput.

**Window Size: Packet Count Mode vs. Byte Mode**

Different implementation could have different interpretations of the TCP standard. Readers may get confused about window size in packet count mode and byte mode. Although \( rwnd \) reported by the receiver in bytes, previous illustrations about \( cwnd \) is in number of packets and then are translated into bytes by multiplying the MSS in order to select the window size from \( \min(cwnd, rwnd) \). Some operating system may direct use byte-mode \( cwnd \), so the algorithm should be adjusted as follows:

```plaintext
if (cwnd < ssthresh) {
    cwnd = cwnd + MSS;
} else {
    cwnd = cwnd + (MSS*MSS)/cwnd
}
```

That is, in slow start phase, rather than increment \( cwnd \) by 1 in packet count mode, we increment it by \( MSS \) in byte mode every time an ACK is received. In congestion avoidance phase, rather than increment \( cwnd \) by \( 1/cwnd \) in packet count mode, we increment it by a fraction of \( MSS \), i.e. \( MSS/cwnd \), every time an ACK is received.

**RSVP, RTP, RTCP, and RTSP**

This chapter discusses related protocols for real-time multimedia data in the Internet such as RTP and RTCP. However, the differences between other related protocols, such as RSVP and RTSP, require to be clarified:

- RSVP is the signaling protocol that notifies the network element along the path to reserve adequate resources, such as bandwidth, computing power,
or queuing space, for real-time applications. It does not deliver the data. RSVP will be studied in Chapter 6.

- RTP is the transport protocol for real-time data. It provides timestamp, sequence number and other means to handle the timing issues in real-time data transport. It relies on RVSP for resource reservation to provide quality of service.
- RTCP is the control part of RTP that helps with quality of service and membership management.
- RTSP is a control protocol that initiates and directs delivery of streaming multimedia data from media servers. It is the "Internet VCR remote control protocol". Its role is to provide the remote control. The actual data delivery is done separately, most likely by RTP.

**Further Readings**

**TCP Standard**
The headers and state diagram of TCP were first defined in [1], but its congestion control technique was later proposed in [2] and revised in [4] until 1988 because congestion was not an issue in the beginning of the Internet. A deep suggestion in implementation and some corrections for TCP were given in [3]. [5] and [6] standardize the four key behaviors of the congestion control in TCP. SACK and FACK were defined in [7] and [8], respectively.


**On TCP Versions**
The below two papers compare different versions of TCP.


Modeling TCP Throughput
Two widely referred TCP throughput formulas are proposed in [11] and [12]. By giving packet loss ratio, RTT and RTO, these formulas will answer you the mean throughput of a TCP connection.


Berkeley Packet Filter

NS2 Simulator
A network simulator widely used by the Internet research community.


Exercises

Hands-on Exercises

1. NS-2 is the most popular simulator for TCP research. It includes a package called NAM that can visually replay the whole simulation at all timescales. Many websites that introduce ns-2 can be found at [13]. Use NAM to observe a TCP running from a source to its destination, with and without buffer overflow at one intermediate router.
- Step 1: Search the ns-2 website and download suitable version for your target platform.
- Step 2: Follow the installation instructions to install all the packages.
- Step 3: Build a scenario consisting of three cascaded nodes, one for the Reno TCP source, one for an intermediate gateway, and one for the destination. The links to connect them are full-duplex 1Mbps.
- Set the gateway to have large buffer. Run a TCP source towards the destination.
- Set the gateway to have small buffer. Run a TCP source towards the destination.

For all the Reno TCP state that the Reno TCP source in the above two tests enter, screen dump them and indicate which state the TCP source is in. The figures should be correlated. For example, to represent the slow-start behavior you may display it by two figures: (1) an ACK is coming back; (2) the ACK triggers out two new data segments. Carefully organize the figures so that the result of this exercise is no more than one A4 page. Only display necessary information in the screen dumps. Pre-process the figures so that no window decorations (window border, NAM buttons) are displayed.

2. Exponential Weighted Moving Average (EWMA) is commonly used when the control needs to smooth out rapidly fluctuating values. Typical applications are smoothing the measured round-trip time, or computing the average queue length in Random Early Detection (RED) queues. In this exercise, you are expected to run and observe the result of an EWMA program at our website. Tune the network delay parameter to observe how the EWMA value evolves.

3. Reproduce Figure 4.15.
   - Step 1: Patching Kernel: Logging Time-Stamped CWND/SeqNum
   - Step 2: Recompiling (Appendix K)
   - Step 3: Installing New Kernel & Reboot (Appendix K)

4. Linux Packet Socket is a useful tool when you want to generate arbitrary types of packet. Modify the example source code available at our website to generate a packet and sniff the packet by the same program.

5. Dig out the retransmit timer management in FreeBSD 4.X Release. How does it manage the timer? Use a compact table to compare it with that of Linux 2.6.

6. How Linux integrates NewReno, SACK, and FACK in one box? Identify the
key difference in variables mentioned in Subsection 4.3.8 and find out how Linux resolves the conflict.

7. What transport protocols are used in Skype, MSN, or other communication software? Please use ethereal to observe their traffic and dig out the answer.

8. What transport protocols are used in MS media player or realmedia? Please use ethereal to observe and dig out the answer.

9. Write a server and a client by the socket interface. The client program may send out the words to the server once the user presses the key enter, and the server will respond to these words with any meaningless terms. However, the server will close the connection once receiving the word bye. Also, once a guy keys in “GiveMeYourVideo”, the server will immediately send out a 50MB data with message size of 500 bytes.

10. Write a server and client or modify the client program in problem 9 to calculate and record the data transmission rate per 0.1 second for a 50MB data transmission with message size of 500 bytes. Use xgraph or gnuplot to display the results.

11. Continue the work done in problem 9. Modify the client program to use a socket which embedded a socket filter to filter out all packets which include the term “the_packet_is_infected”. Then, compare the average transmission rate provided by the sockets for the data transmission of 50MB with that done by a client program which simply discards these messages at the user-layer.

12. Modify the programs written in problem 9 to create socket based on SCTP to demonstrate that the voice talk can continue without any blocking due the transmission of the large file, i.e. to demonstrate the benefit of multi-streaming from SCTP.

Written Exercises

1. Compare the roles of error control among data link layer, IP layer, and end-to-end layer. Of the link-layer technologies, choose Ethernet to discuss. Use a compact table with keywords to compare the objective, covered fields, algorithm, field length, and any other same/different properties. Why should there be so many error controls throughout a packet's life? Itemize your reasons.

2. Compare the roles of addressing among data link layer, IP layer, end-to-end layer, and real-time transport layer. Of the link-layer
technologies, choose Ethernet to discuss. Among the real-time transport protocols, choose RTP to discuss. Compare the objective, uniqueness, distribution/hierarchy, and other properties using a compact table filled with keywords.

3. Compare the roles of flow control between data link layer and end-to-end layer. Of the link-layer technologies, choose Fast Ethernet to discuss. Compare the objective, flow control algorithms, congestion control algorithms, retransmission timer/algorithms, and other important properties using a compact table filled with keywords. Further explanations to non-trivial table entries should be.

4. Consider that a mobile TCP receiver is receiving data from its TCP sender, what will the RTT and the RTO evolve when the receiver gets farer and then nearer? Assume the moving speed is very fast such that the propagation delay ranges from 100 ms to 300 ms within 1 second.

5. A connection running TCP transmits packets across a path with 500-ms propagation delay without bottlenecked by any intermediate gateways. What is the max throughput when window scale option is not used? What is the max throughput when window scaling option is used?

6. Given that the throughput of a TCP connection is inversely proportional to its RTT, connections with heterogeneous RTTs sharing the same queue will get different bandwidth shares. What will be the eventual proportion of the bandwidth sharing among three connections if their propagation delays are 10 ms, 100 ms, 150 ms, and the service rate of the shared queue is 200 kbps? Assume that the queue size is infinite without buffer overflow (no packet loss), and the max window of the TCP sender is 20 packets, with each packet having 1500 bytes.

7. What is the answer in Question 6 if the service rate of the shared queue is 300kbps?

8. If the smoothed RTT kept by the TCP sender is currently 30 msec and the following measured RTT are 26, 32, and 24 msec, respectively. What is the new RTT estimate?

9. TCP provide a reliable byte stream, but it is up to the application developer to “frame” the data sent between client and server. The maximum payload of a TCP segment is 65,515 bytes. Why would such a strange number be chosen? Also, why do most TCP senders only emit packets with packet size smaller than 1460 bytes, e.g. even though a client might send 3000 bytes via write( ), the server might only read 1460 bytes?

10. In most UNIX systems it is essential to have root privilege to execute
programs that have direct access to the internetworking layer or link layer. However, some common tools, such as ping and traceroute, can access the internetworking layer using normal user account. What is the implication behind this paradox? How do you make your own programs that can access the internetworking layer be similar to such tools? Briefly propose two solutions.

11. Use a table to compare and explain all socket domains, types, and protocols that are supported by Linux 2.4.

12. The RTP incorporates a sequence number field in addition to the timestamp field. Can RTP be designed to eliminate the sequence number field and use the timestamp field to re-sequence the out-of-order received packets? (Yes/No, why?)

13. Suppose you are going to design a real-time streaming application over the Internet that employs RTP on top of TCP instead of UDP, what situations will the sender and the receiver encounter in each TCP congestion control state shown in Figure 4.11? Compare your expected situations with those designed on top of UDP in a table format.

14. Recalling Figure 4.3 that it is the delay distribution that makes the different solutions to the same issues of single-hop and multi-hop environments. How will the delay distribution evolves if the transmission channel is of one-hop, two-hop, ......., and 10-hop? Draw three co-related delay distribution figures as in Figure 4.3 to best illustrate the outstanding steps of increasing the hop count (e.g. 1-,2-,and 10-hop).

15. When adding a per-segment checksum to a segment, TCP and UDP all include some fields in the IP layer. However, the segment has not been passed to its underlying layer, i.e. the IP layer, before. How could TCP and UDP know the values in the IP header?

16. The text spends a great amount of pages introducing the different versions of TCP. Identify three more TCP versions by searching http://www.lib.nctu.edu.tw/n_service/abi.html. Itemize them and highlight their contributions within three lines of words with each TCP version.

17. As shown in Figure 4.30, many parts in Linux 2.6 are not specific for TCP/IP, such as read/write functions and socket structure. As being at least a C programmer, analyze how Linux 2.6 organizes its functions and data structures that can be easily initialized into different protocols. Briefly indicate the basic C programming mechanisms to achieve the goals.

18. As described in Section 4.5, many protocols and algorithms are proposed to handle the problems on carrying streaming through the Internet. Please
find open solutions that support the transmission of the media streaming over the Internet. Then, observe these solutions to see whether and how they handle the issue addressed in Section 4.5. Do these solutions implement the protocols and algorithms introduced herein?

19. Compared with loss-driven congestion controls like that used in NewReno and SACK, TCP Vegas is an RTT-driven congestion control, which actually is a novel ideal. However, is TCP Vegas popular used in the Internet? Are there any problems when the flows of TCP Vegas compete with that of the loss-driven controls against for a network bottleneck?

20. Could you find out other RTT-driven congestion controls, except TCP Vegas? Or, do you find out any congestion controls that concurrently consider packet losses and RTT to avoid from the congestion and control the rate? Are they robust and safe to deploy in the Internet?

21. As introduced in Section 4.4.1, when you intend to open a socket for a connection between processes or hosts, you need to assign the domain argument as AF_UNIX and AF_INET, respectively. Below the socket layer, how different data flows and function calls are implemented for sockets with different domain arguments? Are there other widely used options for the domain argument? In what kind of condition will you need to add a new option to the argument?

22. Except AF_UNIX and AF_INET, are there other widely used options for the domain argument? What are their functions?