Transport Layer

1. Transport services and protocols

- **provide logical communication between processes running on different hosts**
- transport protocols run in end systems
  - send side: **breaks** application **messages** into **segments**, passes down to **network layer**
  - receiving side: **reassembles** segments into messages, passes up to **application layer**
- more than one transport protocol available to apps
  Internet: `TCP` and `UDP`

1.1 Transport layer vs network layer

The transport layer lies just above the network layer in the protocol stack.

Whereas a **transport-layer protocol** provides logical communication between **processes running on different hosts**, a **network-layer protocol** provides logical communication between **hosts**.

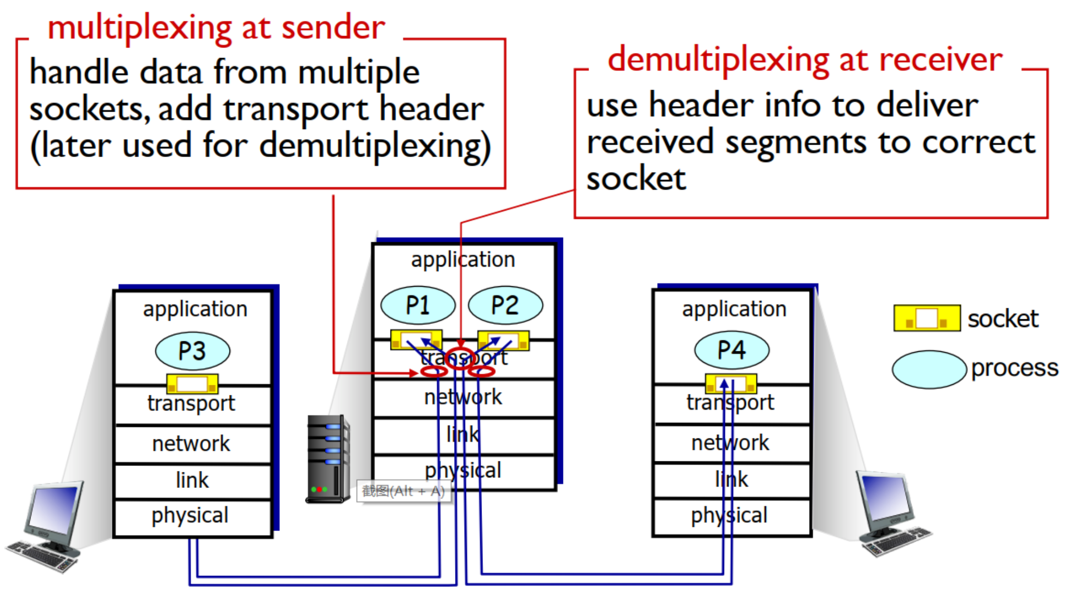
2. Internet transport-layer protocols

- reliable, in-order delivery (`TCP`)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: `UDP`
  - “best-effort” delivery service, IP

3. Multiplexing and Demultiplexing (多路复用和多路分解)

`socket API`: the interface between application and transport layer

Multiplexing/demultiplexing


multiplexing at sender
handle data from multiple sockets, add transport header (later used for demultiplexing)
demultiplexing at receiver
use header info to deliver received segments to correct socket
application
P1
P2
P3
P4
transport
network
link
physical
socket
process

3.1 Demultiplexing (多路分解)

Definition of demultiplexing:

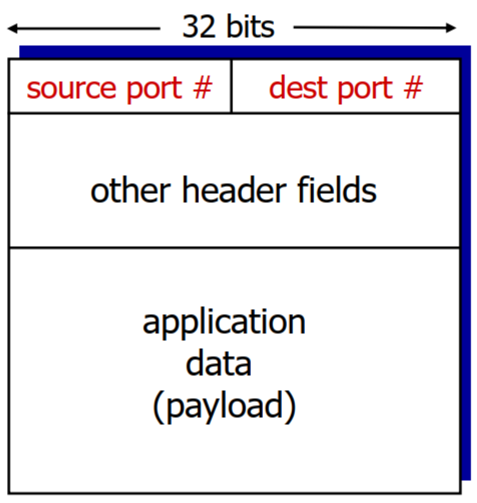
Use the hearder info to deliver received segments to the correct socket in the receiver.

- demultiplexing is the process of handing an incoming packet to the right application
    - ...via the right socket
- demultiplexing task of the transport protocol
    - decision based on info in packet headers
- demultiplexing at receiver

  use header info to deliver received segments to correct socket

3.1.1 How demultiplexing works

- host receives IP datagrams
    - each datagram has source IP & destination IP address and carries one transport-layer segment
    - each segment has source & destination port number
- host uses IP addresses & port numbers to direct segment to appropriate socket



TCP/UDP segment format

3.2 Multiplexing (多路复用)

Definition of multiplexing:

- The transport layer handles data from **source host** from different sockets, encapsulating each data chunk with header information (that will later be used in demultiplexing) to create segments,
- **passing the segments to the network layer** to be sent to a receiving host.

multiplexing at sender

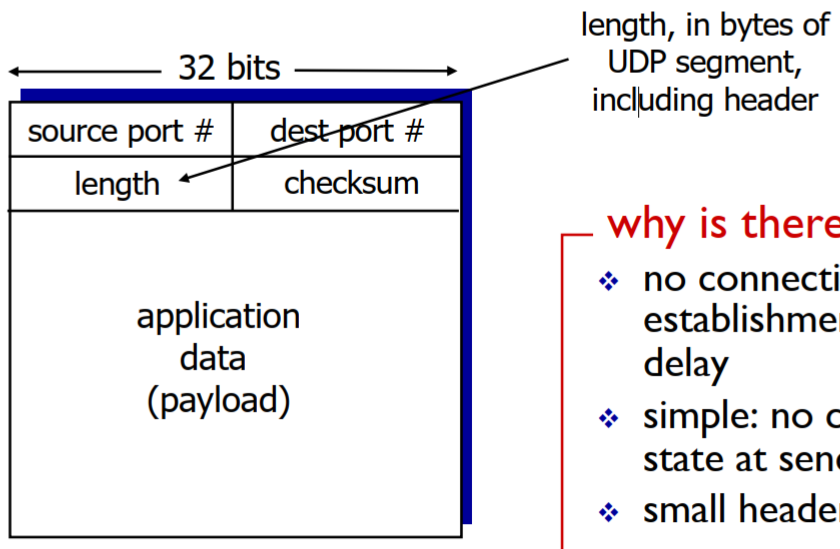
handle data from multiple sockets, add transport header (later used for demultiplexing)

3.3 `UDP`: User Datagram Protocol

3.3.1 characteristics:

- do just about **as little as** a transport protocol can do. Aside from the multiplexing/demultiplexing function and some light error checking, it adds nothing to IP (bare-bone Internet transport protocol)
- **best effort service**
- UDP segments may be
    - **lost**
    - **delivered out-of-order**
- **connectionless**
    - no handshaking between `UDP` sender, receiver (不建立连接，不需要任何准备即可进行数据传输)
    - each `UDP` segment handled independently of others
- `UDP` used for
    - streaming multimedia apps (loss tolerant, rate sensitive)
    - DNS
    - sensor data

3.3.2 UDP: segment header


length, in bytes of UDP segment, including header
32 bits
source port #
dest port #
length
checksum
application data (payload)

UDP segment format

why is there a UDP?

- no connection establishment , thus no delay
- simple: no connection state at sender, receiver
- small header size
- no congestion control: UDP can blast away as fast as desired

application data (payload, 也是 **data field, 数据字段**)以外的部分均为**首部字段 (header field)**

- The **application data** occupies the **data field** of the UDP segment.
- The `UDP` **header** has only **four fields**, each consisting of **two bytes**.
- the **port numbers** allow the **destination host** to pass the **application data** to the **correct process** running on the **destination end system** (that is, to perform the **demultiplexing** function).
- The **length field** specifies the **number of bytes** in the UDP segment (**header plus data**).
- The **checksum** is used by the **receiving host** to **check** whether **errors** have been introduced into the segment.

3.3.3 UDP: checksum (检验和)

goal: detect “errors” (e.g., flipped bits) in segments. That is, the checksum is used to determine whether bits within the UDP segment have been altered.

- for sender:
  - treat segment contents, including header fields, as sequence of 16-bit integers
  - checksum: bit addition of segment contents, then one complemented
  - sender puts checksum value into UDP checksum field
- for receiver:
  - compute checksum of received segment
  - then...

Why UDP provides a checksum in the first place?

There is no guarantee that all the links between source and destination provide error checking. UDP must provide error detection at the transport layer, on an end-end basis, if the end-end data transfer service is to provide error detection (**end-end principle**).

3.4 Connection-Oriented Transport: `TCP`

3.4.1 The TCP Connection

`TCP` protocol runs only in the **end systems** and not in the intermediate network elements (routers and link-layer switches).

- **connection-oriented**

  Before one application process can begin to send data to another, the two processes must **first "handshake"** with each other—that is, they must send some **preliminary segments** to each other to establish the parameters of the ensuing data transfer.

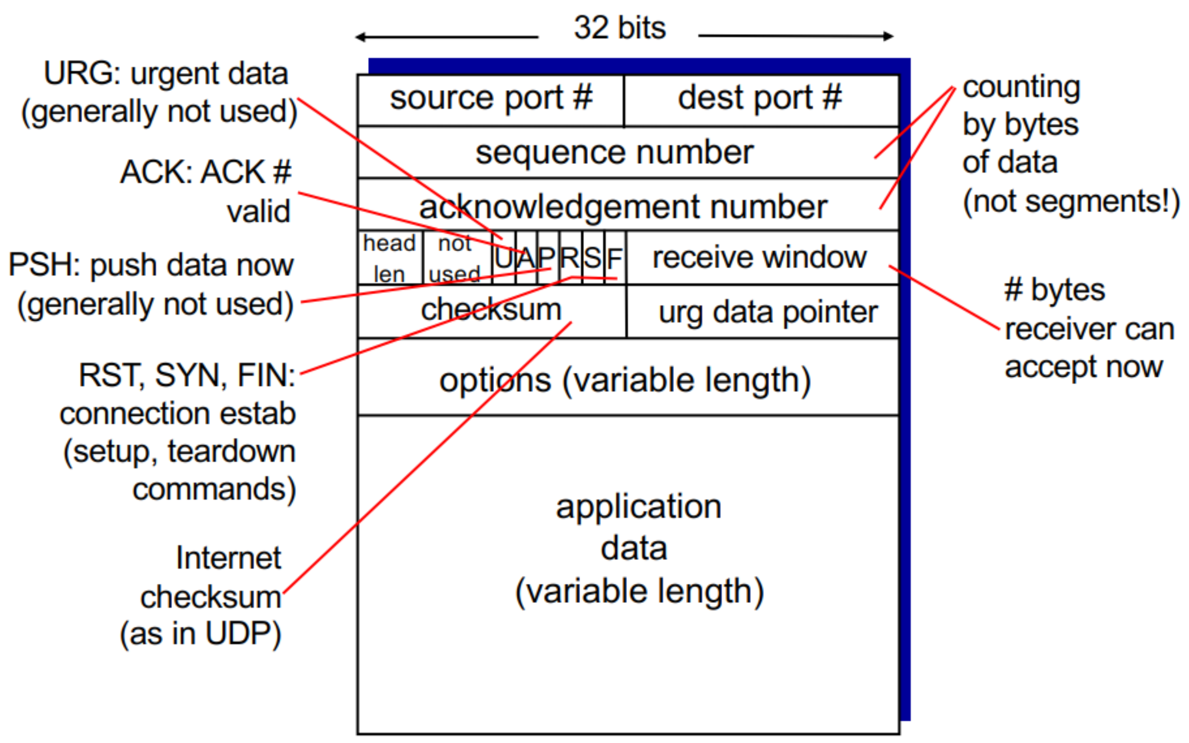
- **full-duplex service** (双全工服务)

  If there is a TCP connection between Process A on one host and Process B on another host, then application layer data can flow from Process A to Process B at the same time as application layer data flows from Process B to Process A.

- **point-to-point**

  between a single sender and a single receiver

3.4.2 TCP segment structure


32 bits
URG: urgent data (generally not used)
ACK: ACK # valid
PSH: push data now (generally not used)
RST, SYN, FIN: connection estab (setup, teardown commands)
Internet checksum (as in UDP)
source port #
dest port #
sequence number
acknowledgement number
head len
not used
U A P R S F
receive window
checksum
urg data pointer
options (variable length)
application data (variable length)
counting by bytes of data (not segments!)
# bytes receiver can accept now

The **header** (首部) includes:

- **source and destination port numbers** 源端口号和目的端口号

  which are used for multiplexing/demultiplexing data from/to upper-layer applications

- **checksum field** 检查和字段
- **sequence number field** (32 bit) 序号字段

  used by the TCP sender and receiver in implementing a reliable data transfer service
- **acknowledgment number field** (32 bit) 确认号字段

  used by the TCP sender and receiver in implementing a reliable data transfer service
- **receive window field** (16-bit) 接收窗口字段

  which is used for flow control
- **header length field** (4-bit) 首部长度字段

  specifies the length of the TCP header in 32-bit words
- **options field** (he optional and variable-length) 选项字段

  used when a sender and receiver negotiate the **maximum segment size (MSS)** or as a window scaling factor for use in high-speed networks
- **flag field** (6 bit) 标志字段
  - **ACK** bit is used to indicate that the value carried in the acknowledgment field is valid
  - The **RST**, **SYN**, and **FIN** bits are used for connection setup and teardown
  - Setting the **PSH** bit indicates that the receiver should pass the data to the upper layer immediately
  - the **URG** bit is used to indicate that there is data in this segment that the sending-side upper-layer entity has marked as "urgent."

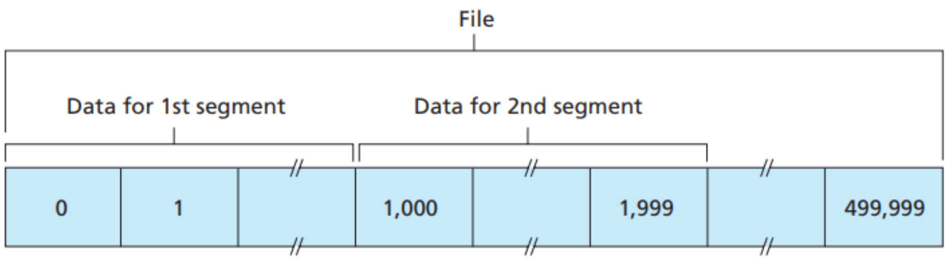
Sequence Numbers and Acknowledgment Numbers

Two of the most important fields in the TCP segment header are the **sequence number field** and the **acknowledgment number field**. These fields are a critical part of TCP's reliable data transfer service.

TCP views data as an unstructured, but ordered, stream of bytes.

- **Sequence Numbers** (在发送方那边)

  The sequence number for a segment is the byte-stream number of the first byte in the segment.

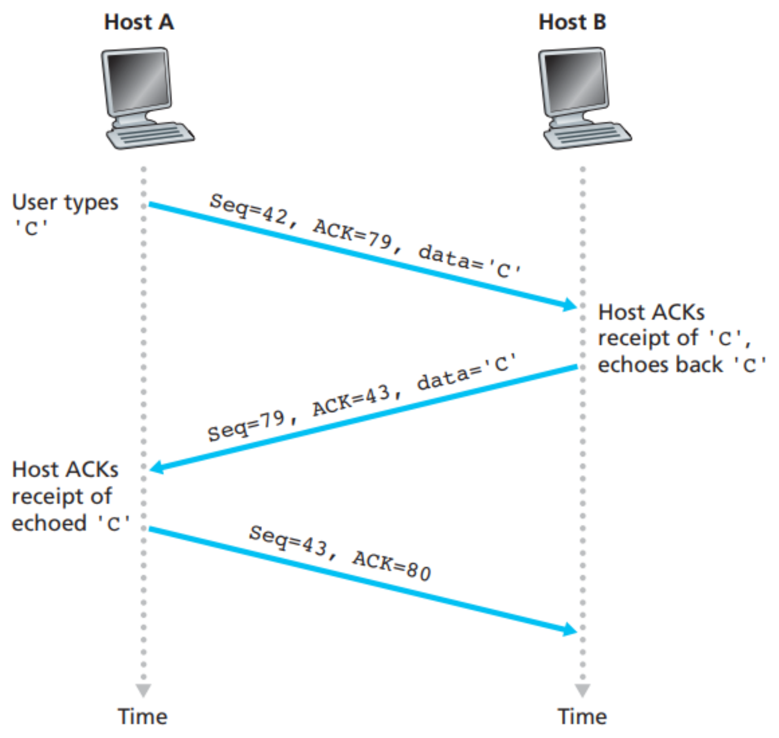


- **Acknowledgment numbers** (从接收方发回发送方)
  - seq # of next byte expected from other side
  - cumulative acknowledgments
  - TCP does not say how to handle out-of-order packets

报文段丢失，ACK就会重复发送确认上一个未收到的报文段第一个序号

ACK = SEQ + bytes of data

Example:



The first segment is sent from the client to the server, containing the 1-byte ASCII representation of the letter ‘C’ in its data field. This first segment also has 42 in its sequence number field, because the client has not yet received any data from the server, this first segment will have 79 in its acknowledgment number field.

The second segment is sent from the server to the client. It serves a dual purpose. First it provides an acknowledgment of the data the server has received. By putting 43 in the acknowledgment field, the server is telling the client that it has successfully received everything up through byte 42 and is now waiting for bytes 43 onward. The second purpose of this segment is to echo back the letter ‘C.’ Thus, the second segment has the ASCII representation of ‘C’ in its data field. This second segment has the sequence number 79, the initial sequence number of the server-toclient data flow of this TCP connection, as this is the very first byte of data that the server is sending. Note that the acknowledgment for client-to-server data is carried in a segment carrying server-to-client data; this acknowledgment is said to be **piggybacked** (捎带) on the server-to-client data segment.

The third segment is sent from the client to the server. Its sole purpose is to acknowledge the data it has received from the server. (Recall that the second segment contained data—the letter ‘C’—from the server to the client.) This segment has an empty data field (that is, the acknowledgment is not being piggybacked with any client-to-server data). The segment has 80 in the acknowledgment number field because the client has received the stream of bytes up through byte sequence number 79 and it is now waiting for bytes 80 onward. You might think it odd that this segment also has a sequence number since the segment contains no data. But because TCP has a sequence number field, the segment needs to have some sequence number.

3.4.3 Round-Trip Time Estimation and Timeout (往返时间的估计与超时)

TCP uses a timeout/retransmit mechanism to recover from lost segments (处理报文丢失).

The timeout should be larger than the connection's round-trip time (RTT), that is, the time from when a segment is sent until it is acknowledged.

How to set TCP timeout value?

- longer than RTT
  - but RTT varies
- too short: premature timeout, unnecessary retransmissions
- too long: slow reaction to segment loss

3.4.3.1 Estimating the Round-Trip Time

How to estimate RTT?

- SampleRTT: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- SampleRTT will vary, want estimates that are "smoother"
  - average several recent measurements, not just current SampleRTT

The sample RTT, denoted SampleRTT, for a segment is the amount of time between when the segment is sent (that is, passed to IP) and when an acknowledgment for the segment is received.

EstimatedRTT = (1- *α*) * *EstimatedRTT + α * SampleRTT*

- exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: *α* = 0.125

In addition to having an estimate of the RTT, it is also valuable to have a measure of the variability of the RTT, that is DevRTT.

DevRTT = (1 – b) • DevRTT + b • |SampleRTT – EstimatedRTT|

Typically, b = 0.25

If the SampleRTT values have little fluctuation, then DevRTT will be small; on the other hand, if there is a lot of fluctuation, DevRTT will be large.

3.4.3.2 Timeout Interval

Timeout interval: EstimatedRTT plus "safety margin"

- **large variation in EstimatedRTT -> larger safety margin**

TimeoutInterval = EstimatedRTT + 4 * DevRTT (safety margin)

3.4.3 TCP connection management

Establish a TCP connection (three-way handshake)

1. The client-side TCP first sends a special TCP segment to the server-side TCP. This special segment contains no application-layer data. But **the SYN bit is set to 1**. For this reason, this special segment is referred to as a **SYN segment**.

   In addition, the client randomly chooses an initial sequence number (client_isn) and puts this number in the sequence number field of the initial TCP SYN segment.

2. Once the IP datagram containing the TCP SYN segment arrives at the server host, the server extracts the TCP SYN segment
   from the datagram, allocates the TCP buffers and variables to the connection, and sends a connection-granted segment to the client TCP.

   This connection-granted segment also contains no application layer data. However, it does contain three important pieces of information in the segment header.

   First, the SYN bit is set to 1.

   Second, the acknowledgment field of the TCP segment header is set to **client_isn+1**.

   Finally, the server chooses its own initial sequence number (server_isn) and puts this value in the sequence number field of the TCP segment header.

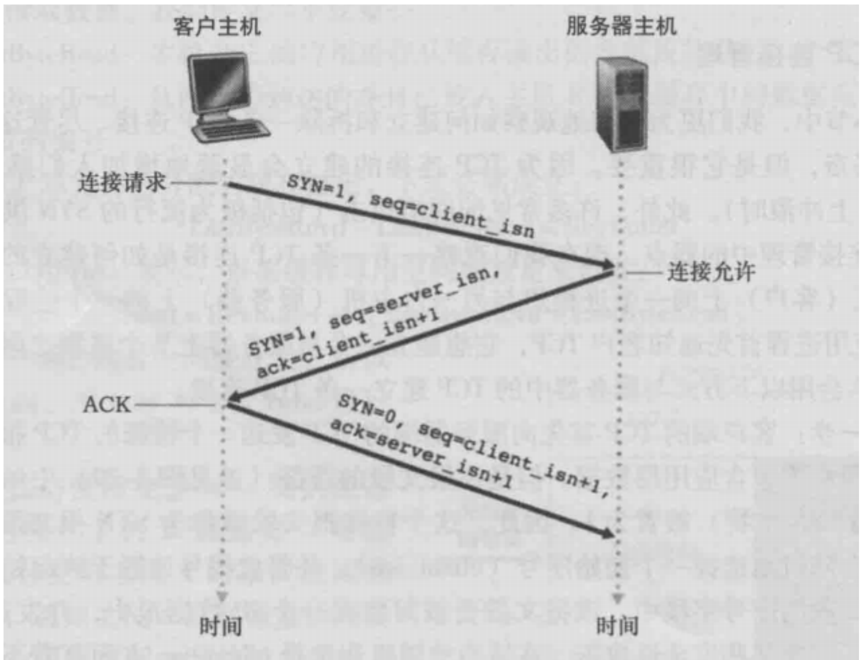
   The connection granted segment is referred to as a **SYNACK segment**.

3. Upon receiving the SYNACK segment, the client also allocates buffers and variables to the connection.

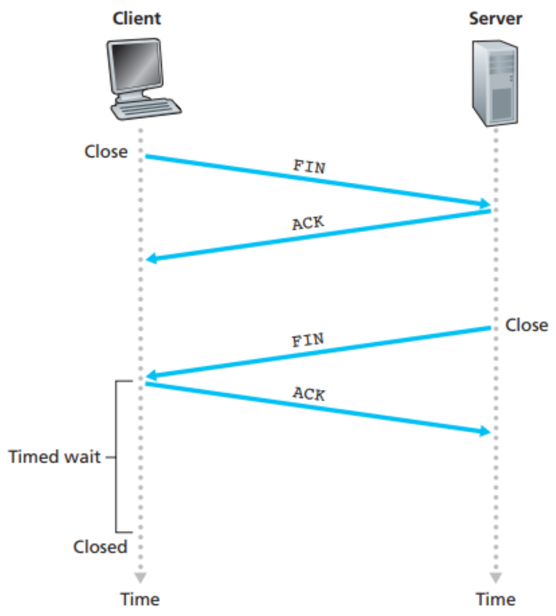
   The client host then sends the server yet another segment; this last segment acknowledges the server's connection-granted segment (the client does so by putting the value **server_isn+1 in the acknowledgment** field of the TCP segment header).

   The SYN bit is set to zero, since the connection is established.

   This third stage of the three-way handshake may carry client-to-server data in the segment payload.



Terminate the TCP connection



1. The client application process issues a close command. This causes the client TCP to send a special TCP segment to the server process. This special segment has a flag bit in the segment's header, the FIN bit is set to 1.
2. When the server receives this segment, it sends the client an acknowledgment segment in return.
3. The server then sends its own shutdown segment, which has the FIN bit set to 1.
4. Finally, the client acknowledges the server's shutdown segment.

4. TCP congestion control(TCP拥塞控制)

TCP congestion-control algorithm

- slow start (慢启动)

  the TCP send rate starts slow but grows exponentially during the slow start phase.

- congestion avoidance (拥塞避免)

  linear increase

- fast recovery (快速恢复)

Slow start and congestion avoidance are mandatory components of TCP, differing in how they increase the size of cwnd in response to received ACKs.