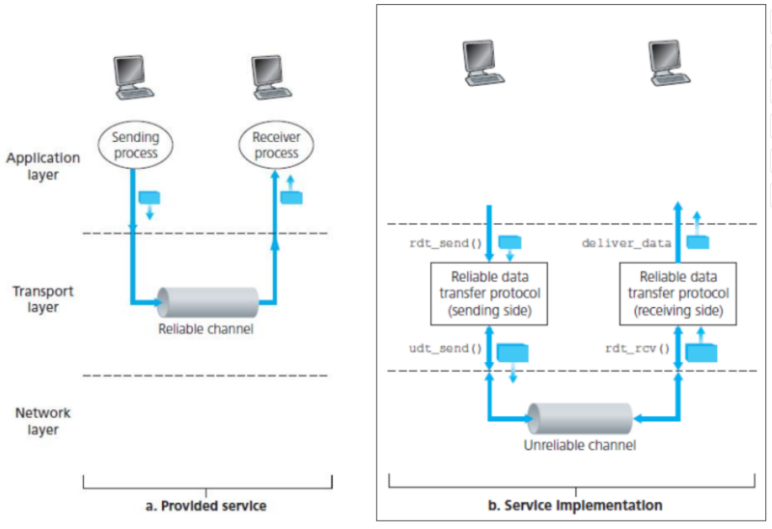
Ch3 Transport Layer

- Definition:
  - Provide logical communication between app **processes** running on different hosts
    - Network layer: logical communication between hosts
  - end-end
  - processes: 程序，网页
  - 主要作用：
    - **send side**: breaks app messages into *segments*, passes to **network layer** （向下）
    - **rcv side**: reassembles segments into messages, passes to **app layer** （向上）
  - eg. Internet: TCP and UDP
    - TCP
      - reliable, in-order delivery
      - congestion control: 拥堵控制发送速率
      - flow control: 怕对方接受不了而减少发送速率
      - connection setup
    - UDP
      - unreliable, unordered delivery
        - 若丢失不会重传
        - 无序发送
      - no-frills extension
        - 设计简单
      - process-to-process data delivery and error checking 用端口号标识，有数据校验字段
  - Transport vs network
    - 网络层：hosts间的逻辑通信
    - 传输层：processes间的逻辑通信
    - 网络层会限制传输层的协议——delay / bandwidth
    - 网络层未提供的服务可以由传输层提供——Reliable data transfer; security
- multiplexing and demultiplexing
  - host-to-host ---> process-to-process delivery service for applications running on the hosts. 复用和解复用的机制使得网络协议能够将从一个主机发送到另一个主机的数据，进一步细化到将数据交付给具体的应用程序
  - **Multiplexing** at sender:
    - handle data from multiple sockets

        - add transport header
    - **Demultiplexing** at receiver: use header info to deliver received segments to correct socket
- **UDP: User Datagram Protocol**
    - 特性
        - Multiplexing/demultiplexing; light error checking
        - no congestion control: Lost, delivered out-of-order to app
            - 减少发送数据包的数量或降低传输速率
            - 立即传输到网络层，无阻塞
        - no handshake
    - 优劣
        - 优势
            - No congestion control: Immediately pass the segment to network layer
            - No handshake: no connection-establish delay
            - No connection state: server cans support more clients 每个数据包独立
            - Smaller packet overhead
        - 劣势
            - No congestion control: congestion, overflow, fairness
            - Not reliable
    - 标识：dst IP & dst.port
    - when host receive UDP segment
        - check dst.port: 确定交给哪个应用
        - 交给对应的socket
    - 相同端口多源IP
        - 也会被定向到相同的socket——一个端口可以处理多个服务器的请求
    - 结构：32bits
        - src port + dst port: 一个16bit, [0,2^16-1]=[0, 65535]
        - length + checksum
            - length: 16bit, [0, 65535], bytes, data+header
            - **checksum**: 检测错误
                - sender: 把head+data分割成16bit(2B)为一组的segment，把segments求和，循环进位，求完以后再按位取反
                - receiver: add all the segment of 16 bit, if one of the bits is a 0, then errors occur. 也循环进位
                - 1001100110011111101010101010101010
                - 1001001110011111101010101010101010

- 应用：用于需要时效的场景
  - 游戏，流媒体
  - dns

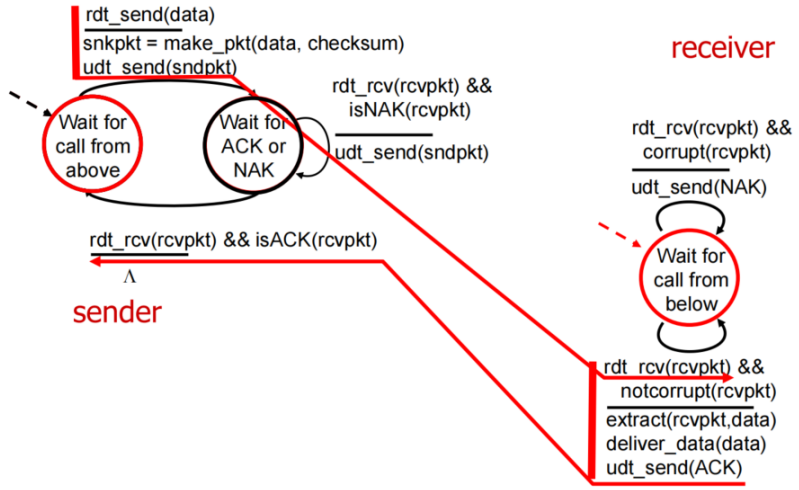
- **principles of reliable data transfer (rdt)**
  - UDP
    - unreliable but faster
    - 直播，流媒体，games
  - TCP
    - reliable, slower
    - email
    - 三次握手
    - 标识：src IP & src port && dst IP & dst port
    - receiver:
      - 用这4个标识寻找socket
      - 只要有一个不同，就会被demultiplexed到不同的socket
    - Web server 对于每个Client都有不同的Socket
      - 初始连接（三次握手）&有HTTP请求的dst port=80(Server)
    - 在使用单个进程的情况下，服务器可以通过创建线程来处理多个客户端连接
      - 主进程用于管理所有客户链接
      - 每当有新的Client请求，server就会为连接创造一个子进程，并用新的socket来处理
  - 用rdt protocol 放在sender-channel & channel-receiver 之间，传输层

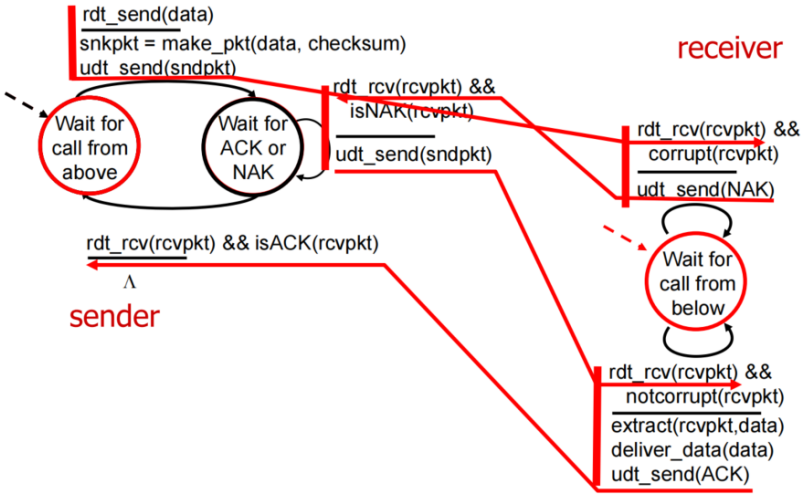


  - rdt1.0
    - sender sends data into underlying channel
      - 从上层接收data call
      - make packet and send

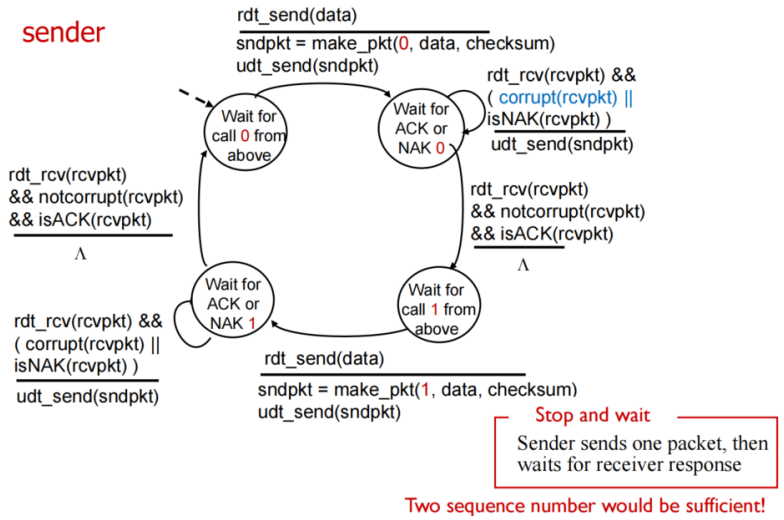
- receiver read data
- no need for feedback （no control message）

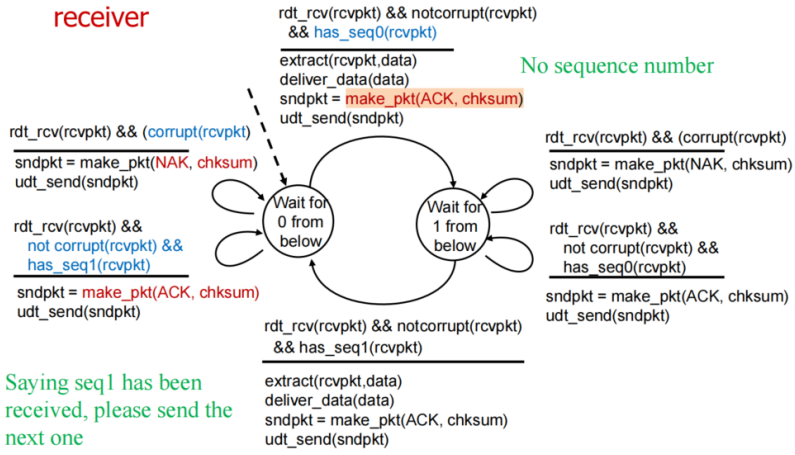
- rdt2.0
  - underlying channel may flip bits in pkt
  - error detection: checksum
  - handle error?
    - control msgs: ACK & NAK
    - if NAK: retransmits pkt
  - FSM



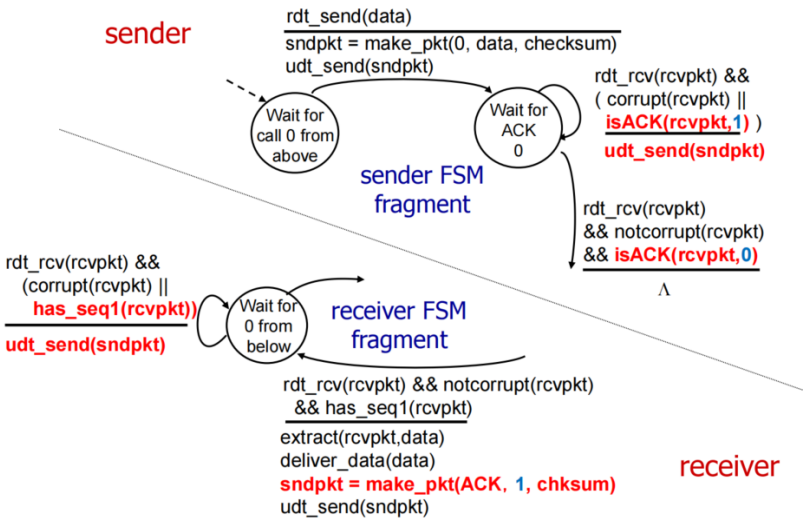


  - flaw：如果msg丢失怎么办/重复
    - 重发pkt
    - add sequence number
  - FSM



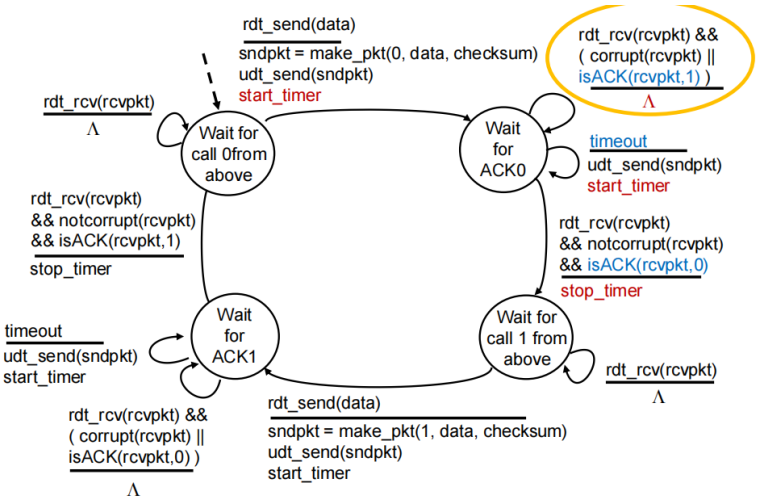


- rdt2.2: a NAK-free protocol
  - 两个同一包的ACK表明新的没收到
  - FSM



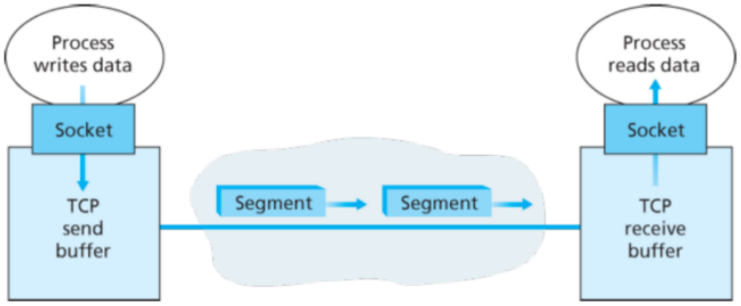
- rdt3.0
  - channels with errors and loss (ack, pkt)
  - 增加等待时间范围
    - if no ack received in this time: retransmit 当作ACK丢失
    - if pkt/ack delayed

        - re-transmit & handle by seq
        - 等多久?
- FSM

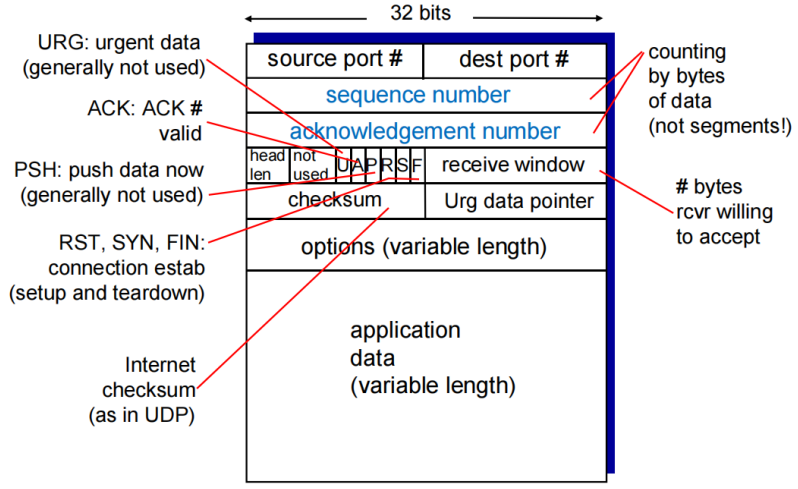


- performance--bad--->pipeline
    - go-Back-N
        - 最早的包开始计时 only one timer
        - 积累ACK
        - retransmit all pkts in window
    - selective repeat
        - 每个pkt计时
        - 单独ack
        - 只重传可能丢失的
        - Receiver needs to keep track of the out-of-packets
        - window 里面有许多seq num
            - 有些已经被ACK，有些已发送还未ACK，有些未被发送
            - 可以不按顺序
            - `send_base`
        - sender
            - 收到开始的ACK，move the window
        - receiver
            - if pkt n in [rcvbase, rcvbase+N-1]
            - 如果先收到后面的，就把它们放到buffer里，直到收到丢失的包裹，
        - 有Bug!
            - sender 一个ACK都没有收到，所以全部重新发送，seq从0开始
            - receiver所有ACK都发送了，收到sender重发的seq=0的pkt，以为是新的
            - seq num >= 2* window size

- **TCP: Transmission Control Protocol**
  - Overview
    - 点对点：one sender-receiver
      - no buffer or variables 在网路的中间设备，如router or switcher
    - reliable:
      - in-order byte stream连续的, no msg boundaries, 从buffer直接取出
      - seq & ack 基于byte而不是pkt
    - piplined: congestion and flow control set window size 动态调整窗口大小
    - full duplex data（允许数据在两个方向上同时流动）: **MSS maximum segment size**, 1460 bytes通常，except 20 bytes IP head and 20 bytes TCP head
      - MTU maximum transmission unit (link-layer): 1500bytes=app date + TCP/IP header(40B)
    - connection-oriented: handshake
    - flow control: sender will not overwhelm receiver
    -



  - segment structure
    - segment format



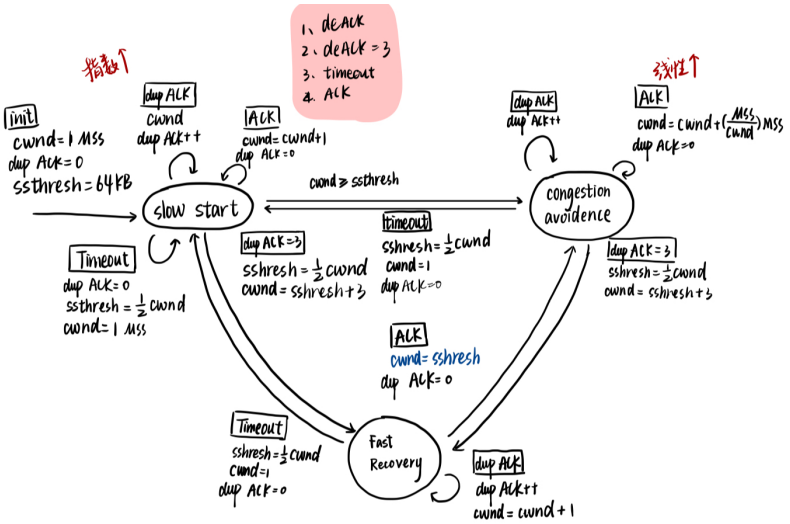
    - seq. num and ACK
      - 标识的不是seg而是byte
      - ack：下一个想要哪个byte，下一个seq
      - 若中间丢了一段，ack =丢的第一个byte编号

        - cumulative ACK 保证所有数据被接受
        - seq随机数初始化
- round-trip time estimation
    - timeout > RTT, but RTT varies
    - SampleRTT: 发送到收到ACK的时间，多次求平均——由α 确定
    - $\mathrm{EstimatedRTT} = (1-\alpha) \times \mathrm{EstimatedRTT} + \alpha \times \mathrm{SampleRTT}$
    - set the timer: $\mathrm{DevRTT} = (1-\beta) \times \mathrm{DevRTT} + \beta \times |\mathrm{SampleRTT\text{-}EstimatedRTT}|, \beta = 0.25$
    - $\mathrm{TimeoutInterval} = \mathrm{EstimatedRTT} + 4 \times \mathrm{DevRTT}$
- reliable data transfer
    - 建立在IP不可靠服务之上
    - 重传触发
        - 为每个未被确认的数据段设置一个重传定时器
        - duplicate acks：多个上一个ack说明这一个没接收到
    - fast

event at receiver	*TCP receiver action*
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send *duplicate ACK*, indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap

    - fast retransmission
        - 3 duplicate ACKs for same data
- flow control
    - 根据接受缓存动态调整接收窗口
    - rcver
        - 接收方发送的TCPheader中包含 `rwnd` 字段，表达可用的缓冲区大小
        - `RcvBuffer` : rcv buffer大小，一般通过os中socket选项配置，默认为4090B
        - `rwnd=RcvBuffer-(LastByteRcvd-LastByteRead)`
    - snder: `LastByteSent-LastByteAcked≤rwnd`
- control management
    - 3 handshake

        - A->B: SYN=1 Seq=x
        - B->A: SYN=1 Seq=y ACK=1 ACKnum=x+1
        - A->B: SYN=0 ACK=1 ACKnum=y+1
        - SYN=0
    - 4 挥手
        - A->B: FIN=1 Seq=x
            - A 不能发数据但是能接收
        - B->A: ACK=1 ACKnum=x+1
            - 还可以发数据
        - B->A: FIN=1 Seq=y
            - B不能发数据
        - A->B: ACk=1 ACKnum=y+1
    - RST
        - 当host接收到与任何socket 都不匹配的ip/port
        - host->server: RST=1
- congestion control
    - 原理
        - 由于IP层只提供传输而不管网络的拥堵情况
        - 1. 如何限制sending rate?-------**congestion window (`cwnd`)**
            - `LastByteSent-LastByteAcked≤min{rwnd,cwnd}`
        - 2. 如何感知拥塞?
            - Timeout; three duplicate ACKs
        - 3. which function f: sending rate=f(receive)?
            - $rate = \frac{cwnd}{RTT}\ \text{bytes/sec}$
            - loss -> congestion -> dec rate
            - ACK -> fine -> inc rate
    - Alogrithm: **additive increase multiplicative decrease AIMD**



- slow start: 指数级增长
  - con begin/timeout出现时
    - `ssthresh=cwnd/2` `cwnd=1`
  - init `cwnd=1 MSS` (maximun segment size)
  - double `cwnd` every RTT 只要收到ACK
  - 每个 `cwnd=cwnd+1`
  - 总共 `tot_cwnd=2*tot_cwnd`
  - `ssthresh` 慢启动阈值
- congestion avoidance: 线性增长
  - 当 `cwnd>=ssthresh`
  - 每个RTT cwnd增加一个MSS
  - 每个 `cwnd=cwnd+(MSS/cwnd)MSS Bytes`
  - 总共 `tot_cwnd=tot_cwnd+1`
- fast recovery
  - `ssthreash=cwnd/2`
  - `cwnd=ssthresh+3MSS`

- congestion：发送方没有**按时**收到接收方的确认
- approaches
  - end-to-end congestion control
    - 感知loss或者RTT增加
    - 增加win size
  - network-assisted congestion control
    - routers provide feedback directly to the sender and/or receiver
    - 2 congestion bits (ToS)
    - 路由器发送可支持的最大速率

    - Explicit Congestion Notification (ECN)
- TCP throughput
    - W : win size where loss occurs
    - $\text{avg } W = \frac{3}{4}W$
    - $\text{avg throughput} = \frac{3W}{4RTT} \text{ bytes/s}$
- TCP Fairness
    - K TCP sessions share same bottleneck link of bandwidth *R*, each should have average rate of *R/K*