计算机网络传输层笔记

1. 传输层的服务和协议

- 为不同主机的进程之间提供 **logical communication running on different host**
- 传输层协议运行在端系统 **send side** : 将消息分为 **segment** passes to network layer **rcv side** : reassembles **segments into messages** passes to app layer
- Internet 中存在两种传输层协议 **TCP**(传输控制协议) and **UDP**(用户数据报(datagram)协议)

1.1. 传输层VS网络层

network layer : **主机 hosts**之间的 logical communication **transport layer** : **进程 prcesses**之间的 logical comminication。(传输层最终还是要调用网络层的服务)

1.2. Inter transport-ayer protocols

1.2.1. TCP reliable , in-order delivery

- 报文段： segment
- 拥塞控制： congestion control
- 可靠的数据传输
- 流量控制

1.2.2. UDP unreliable , unordered delivery

- 报文 : datagram
- 流量不可调节
- 不可靠传输

1.2.3. IP层的服务

- 尽力而为
- 不保证顺序和完整性

2. 多路复用/多路分解

2.1. 多路复用

multiplexing at sender:

从上到下，从socket中收集数据，加报文头传递到网络层 (可以理解为把一个主机里的数个process的message一起通过传输层发送出去)

2.2. 多路分解

demultiplexing at receiver

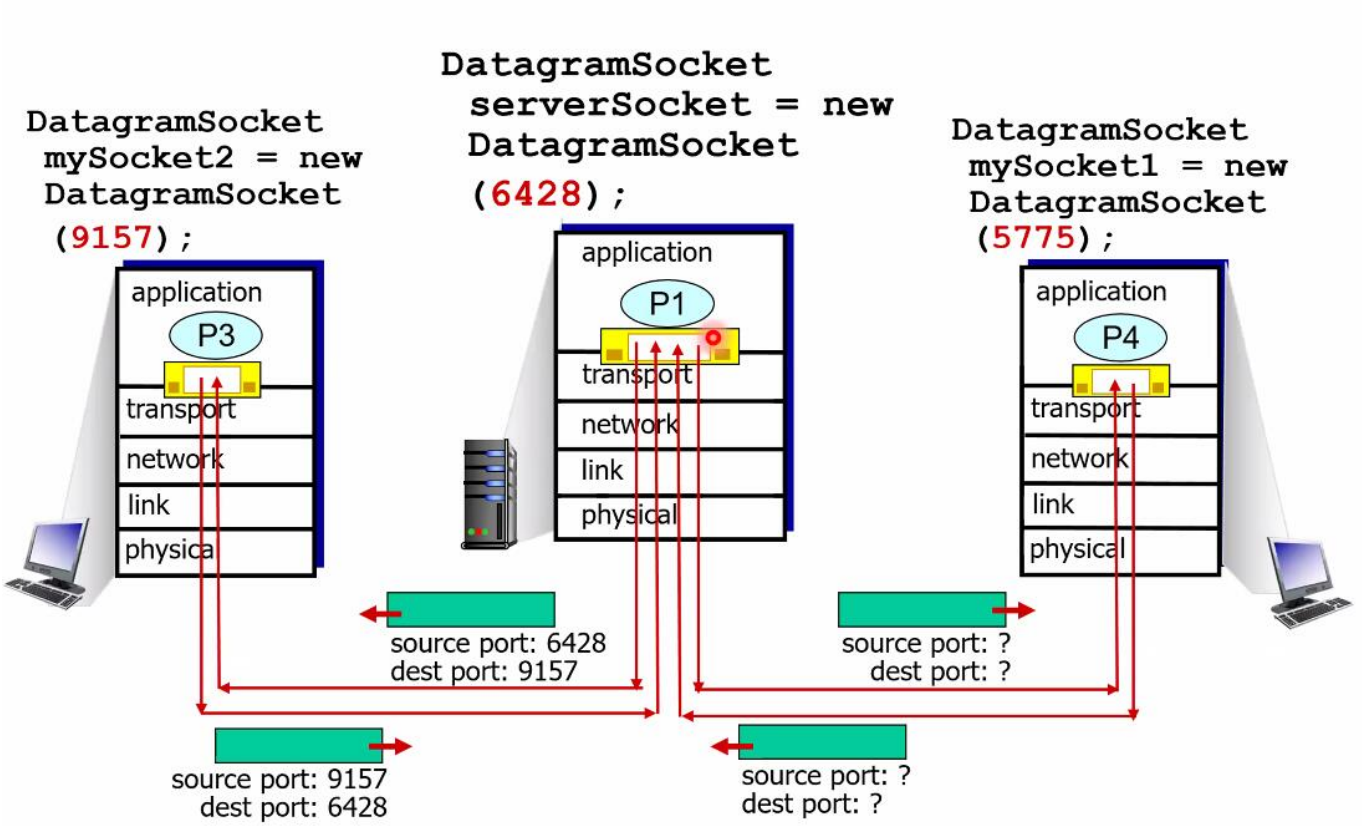
自下而上，传输层的报文交给正确的sorckt

如何实现：

1. IP报文里有源主机和目的主机的IP
2. 传输层报文里有源端口号和目的端口号
3. 通过 IP 和 port 就可以定位到主机的进程实现精确的分解

2.2.1. 面向无连接的多路分解（UDP）

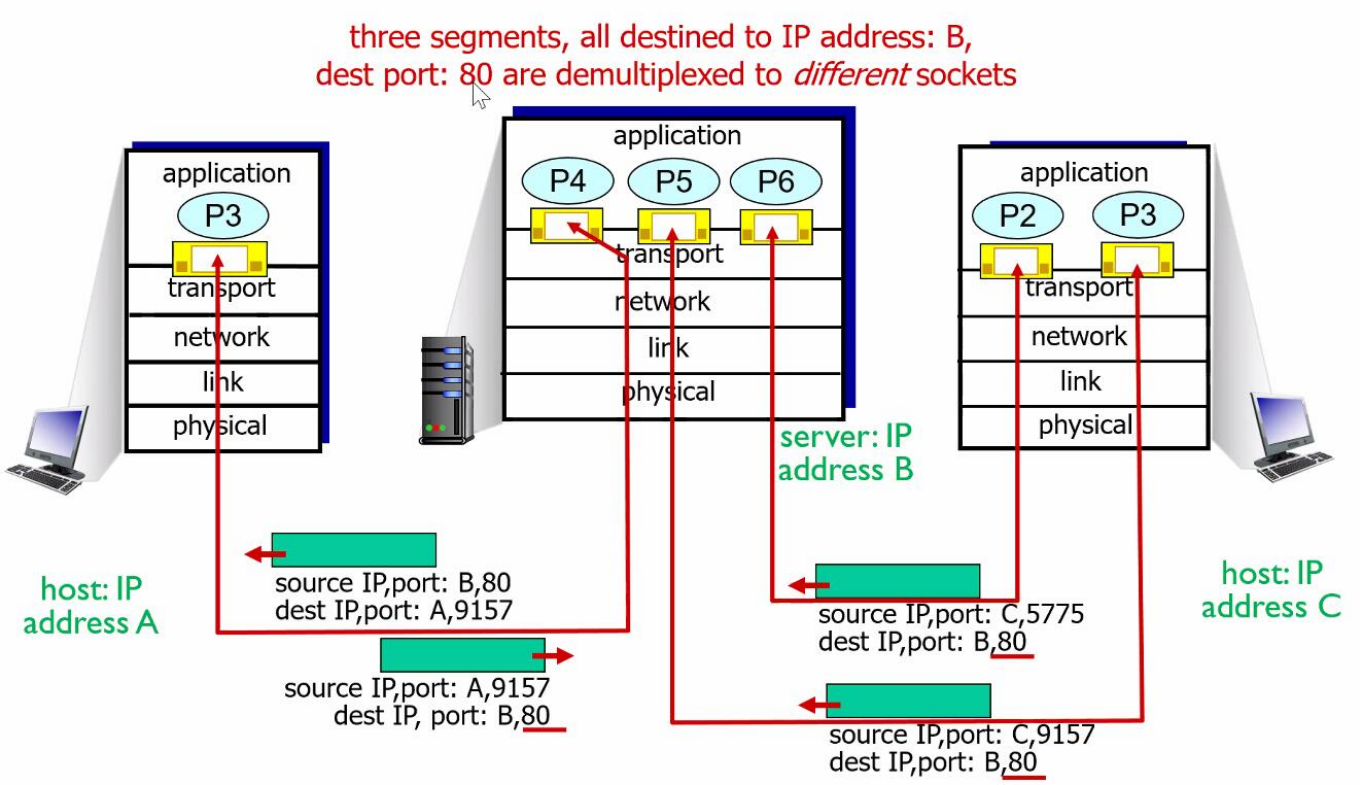
因为是面向无连接的，所以多个客户端的进程对应一个服务端的进程（客户端的进程只需要标明服务端进程的IP和端口号），所有客户端的报文都发送到客户端的**一个socket**上（不建立连接，不像TCP一样还要建立新的用来传输的sorckt**）



2.2.2. 面向连接的多路分解

TCP socket identified by 4-tuple:** (source IP , source port ,dest IP, dest prot)

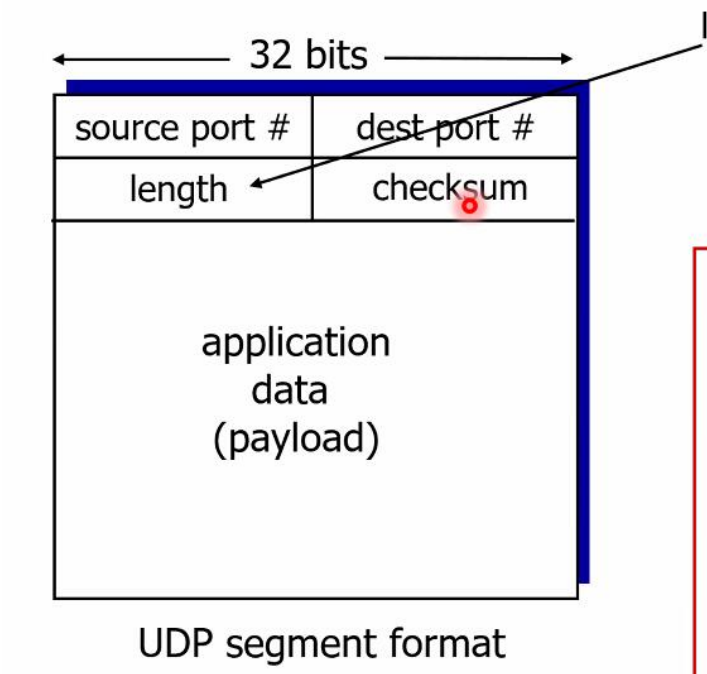
对于面向连接的多路分解，在建立连接的时候，如上所述，客户端还需要给出自己的 IP和port(四元组)，可以唯一地识别不同的客户端进程，服务端在建立连接之后还会重新创建一个**socket**用来传输，换言之，**每一个客户端sorckt都对应一个服务端socket**



3. UDP

- "no frills" , "bare bones" Internet transport protocol(基本服务)（UDP没有多余的修饰，提供的是最基本的服务）
- “best effort" service(UDP追求效率) UDP segments may be : 1.lost 2. delivered out-of-order to app
- connectionsless: 1.没有握手过程 2.每个UDP segment分别处理
- UDP典型应用： 1.流媒体 2.DNS 3.SNMP (这些应用追求传输效率)
- 提升UDP的可靠性？ 在应用层加入一些查错机制

3.1. UDP 报文



为什么需要 UDP:

- 没有连接的建立
- 简单，没有连接状态
- 较小的头部，仅有8个字节
- 无拥塞控制

头部只有 8个字节

3.1.1. UDP 校验和

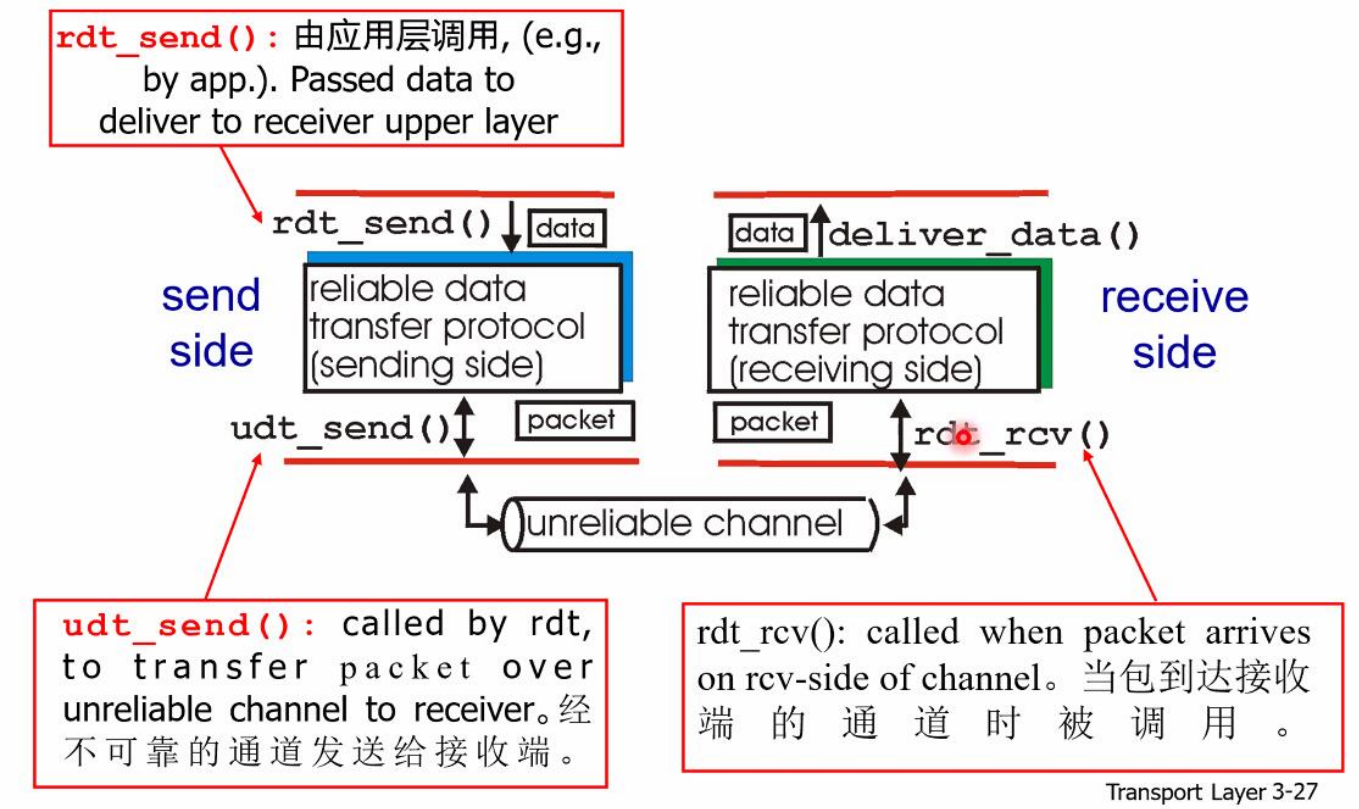
checksum 有 16 bit

(代研究) 个人理解： 把所有的位都划分成16位的字，然后一个字一个字进行累加 ， 如果在某次加法有进位 就把 1 加到最后一位，然后接着累加，最后所有字相加的结果取反

https://blog.csdn.net/DB_water/article/details/78468455

4. 可靠数据传输协议设计

过程的大致描述：

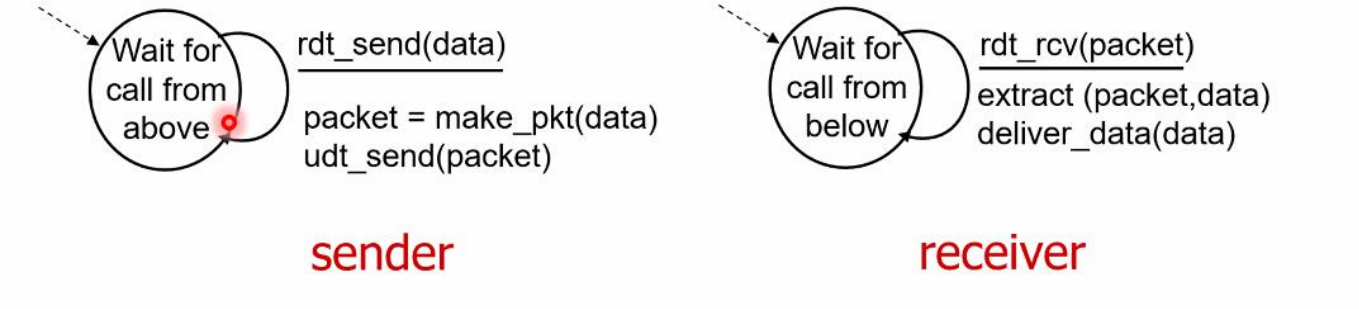


4.1. 如果底层的 channel 是可靠的

rdt1.0

- 不丢失 bit
- 不丢包

只有两个状态，接受状态，发送状态， 发送端不断重复： **接受任务 -> 打包 ->发送** 的过程 接收端不断重复：**收包 -> 解包 -> 传递给应用层**的过程



4.2. 如果 channel 可能发生 bit errors

rdt2.0

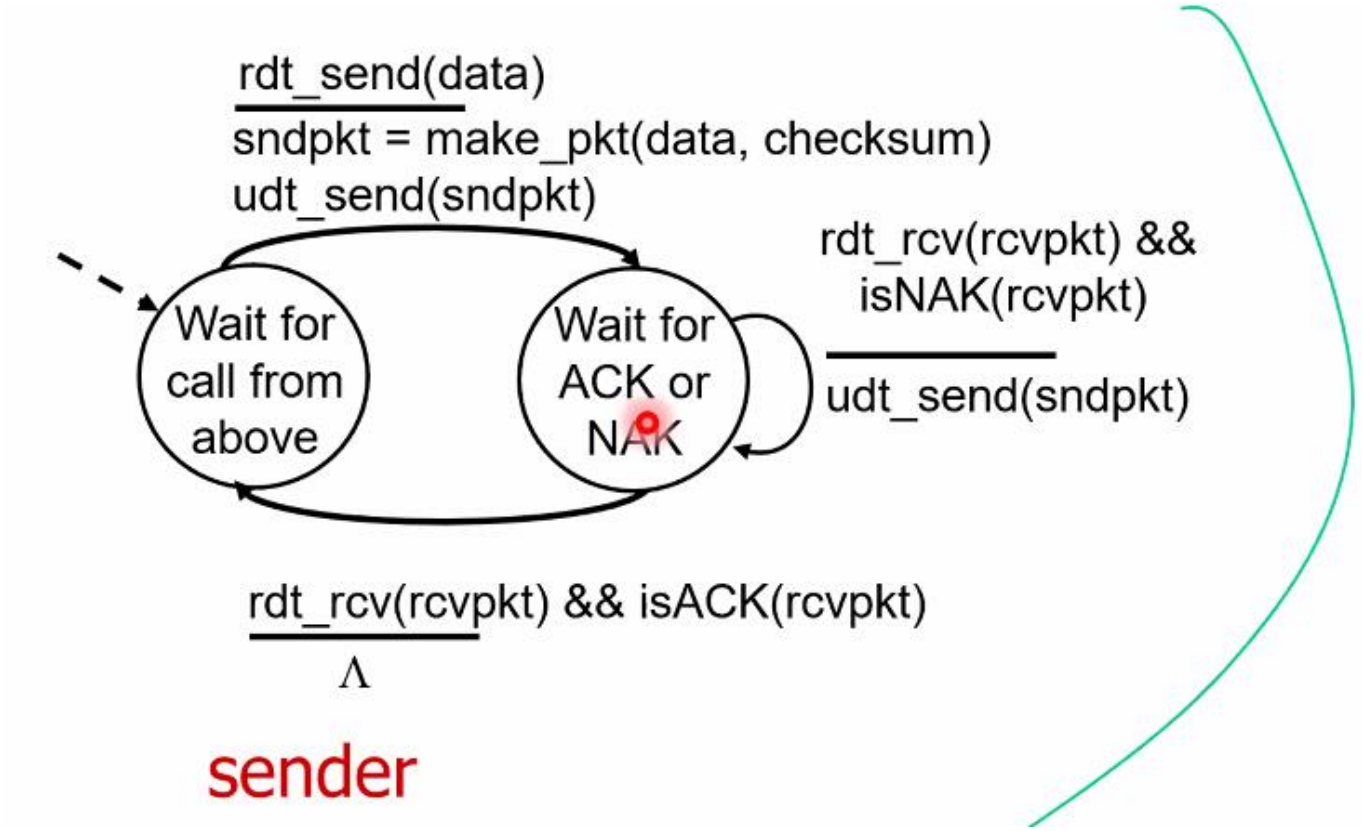
- 底层的传输可能会发生bit翻转(通过checksum 来检测 bit error)
- 如何纠错？如何从错误中恢复? 通过应答的方法

rdt2.0 引入新的机制

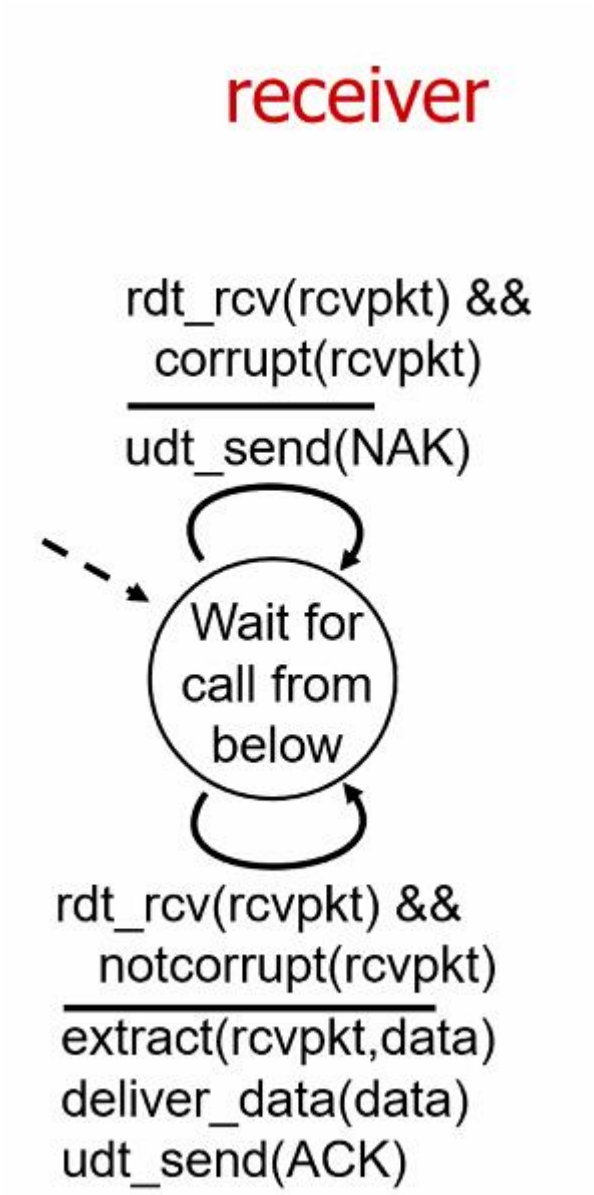
- 错误检测

- 应答（ACKs, NAKs）
- 出错重传

sender 两种状态，reciever一种状态



ps: 在每次发送的时候，等待应答，如果应答是 NAK(错误应答),就重传。如果应答是ACK,就进行下一个发送任务。



ps : 每次接受数据包的时候，先判断有没有出错，出错回复NAK 放弃这个包；没有出错，回复ACK，解包，在发给应用层。

4.2.1. rdt2.0 致命的弱点!

如果ACK/NAK也出错了怎么办?

- 重新发送当前数据包
- 给每个包增加序号
- 接收者抛弃部分数据包

由此引入 rdt2.1

4.2.2. rdt2.1: 应对受损的 ACK/NAK

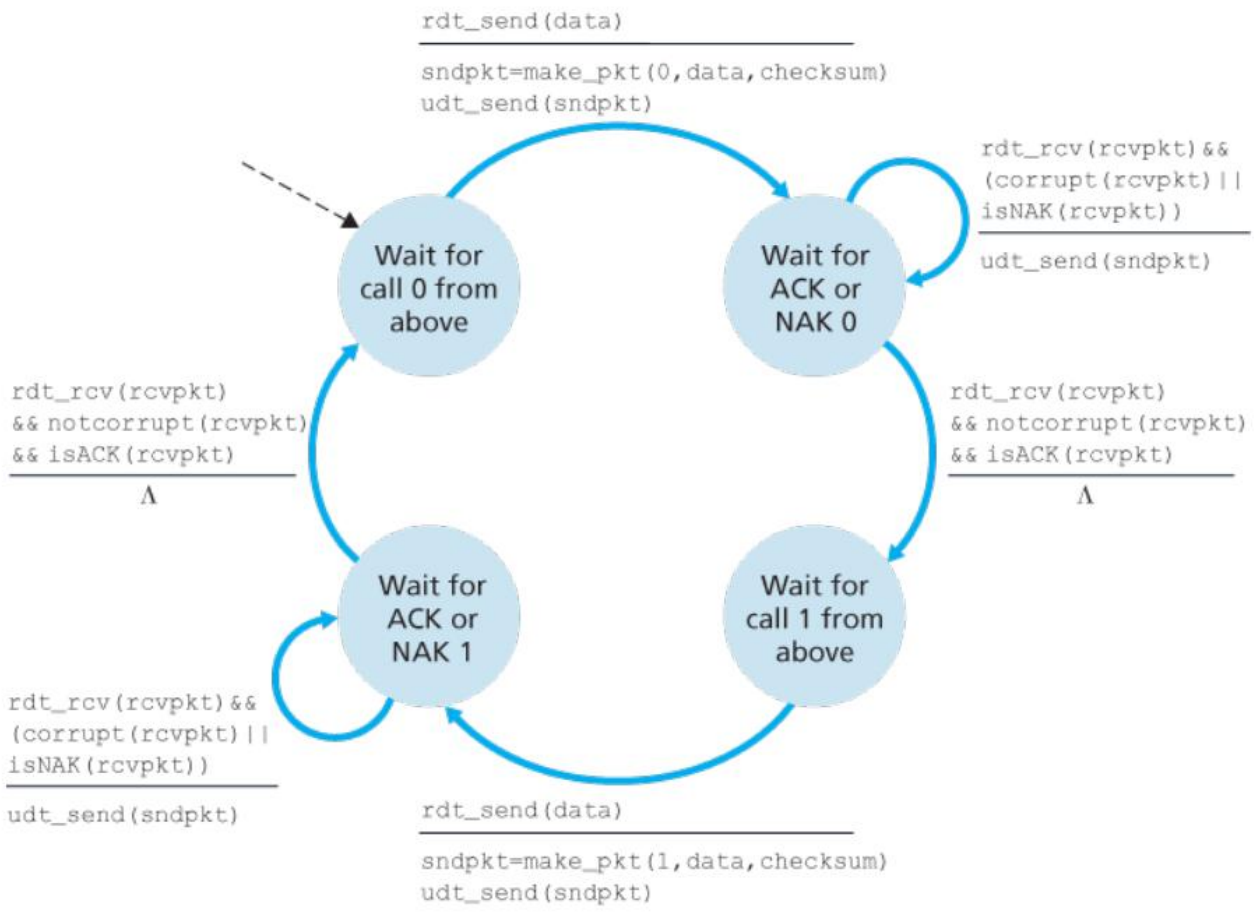
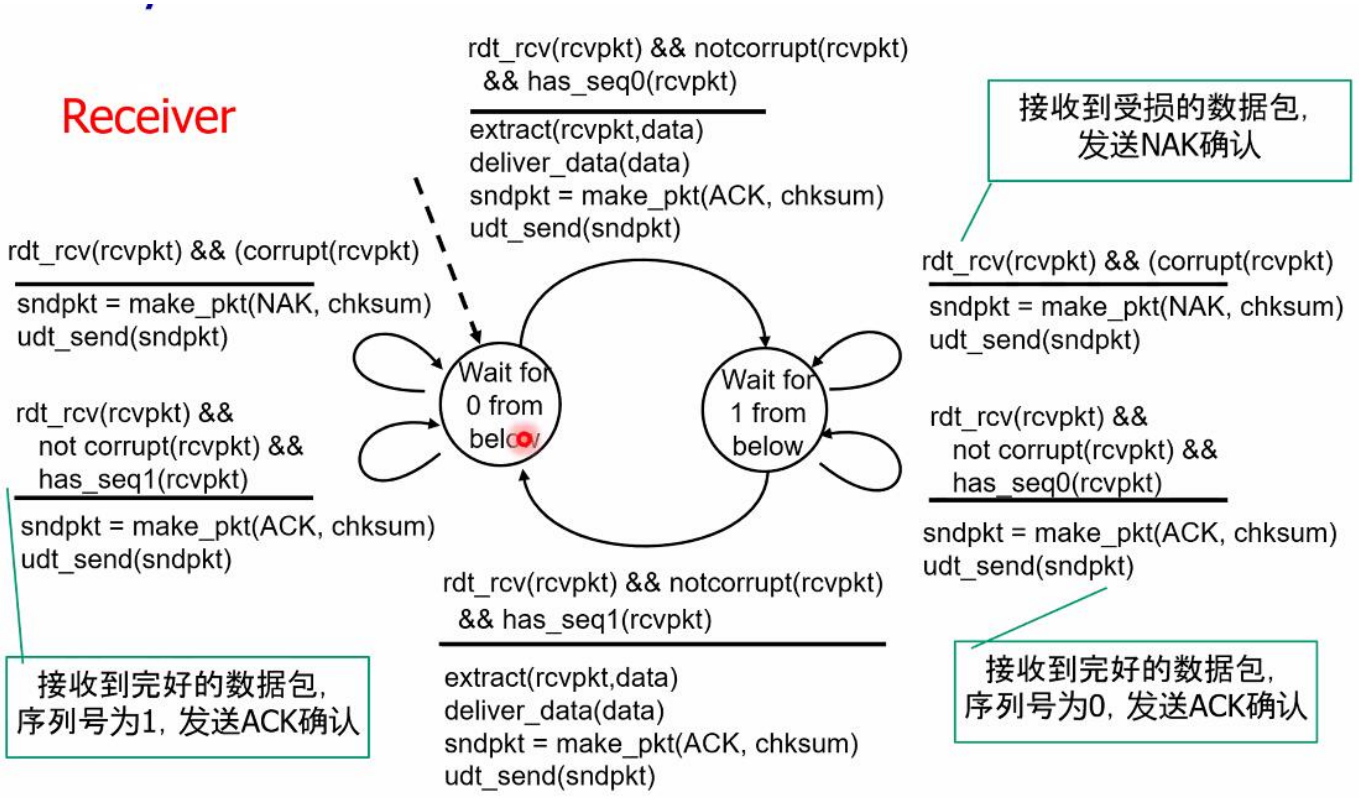


Figure 3.11 *rdt2.1* **sender**

ps:发送方同时还要检测 ACK/NAK的包是否出错，如果出错了（即使受到的是ACK）那么，发送方会重新发送当前的数据包（并且要给当前的数据包编号，编号和上一次的发送相同，这个可以让接收方判断是不是收到同一个包）

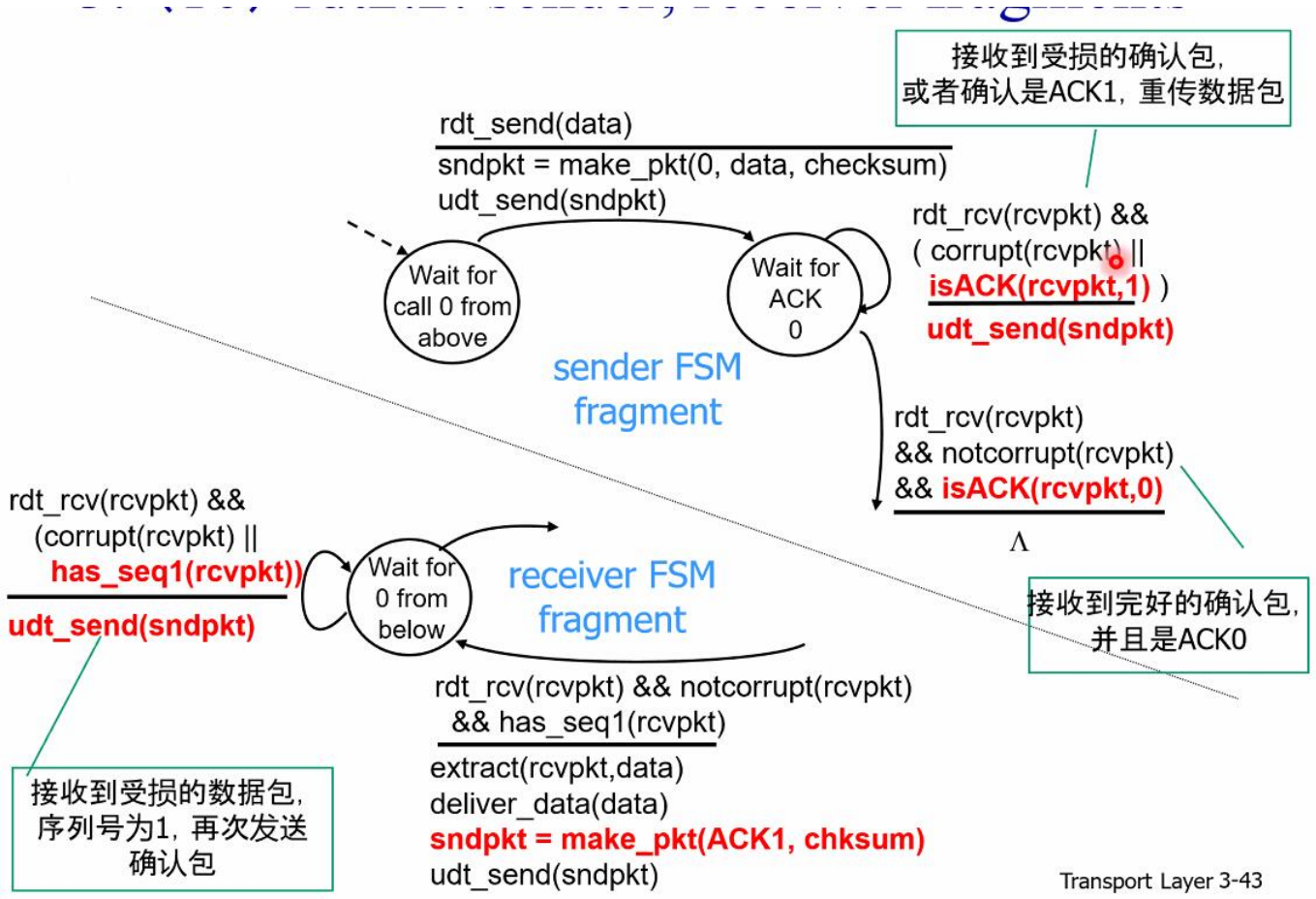


（接收方 增加一个检测包序号，如果序号不是需要的，说明发送方的 ACK回复受损了，接收方重新回复ACK给

发送方（不需要解包，因为当前的包已经有了））

ps:序号只需要 0 和 1,因为我们只需要用 0 和1 来标识收到的包是不是同一个包。

4.2.3. rdt2.2: a NAK-free protocol



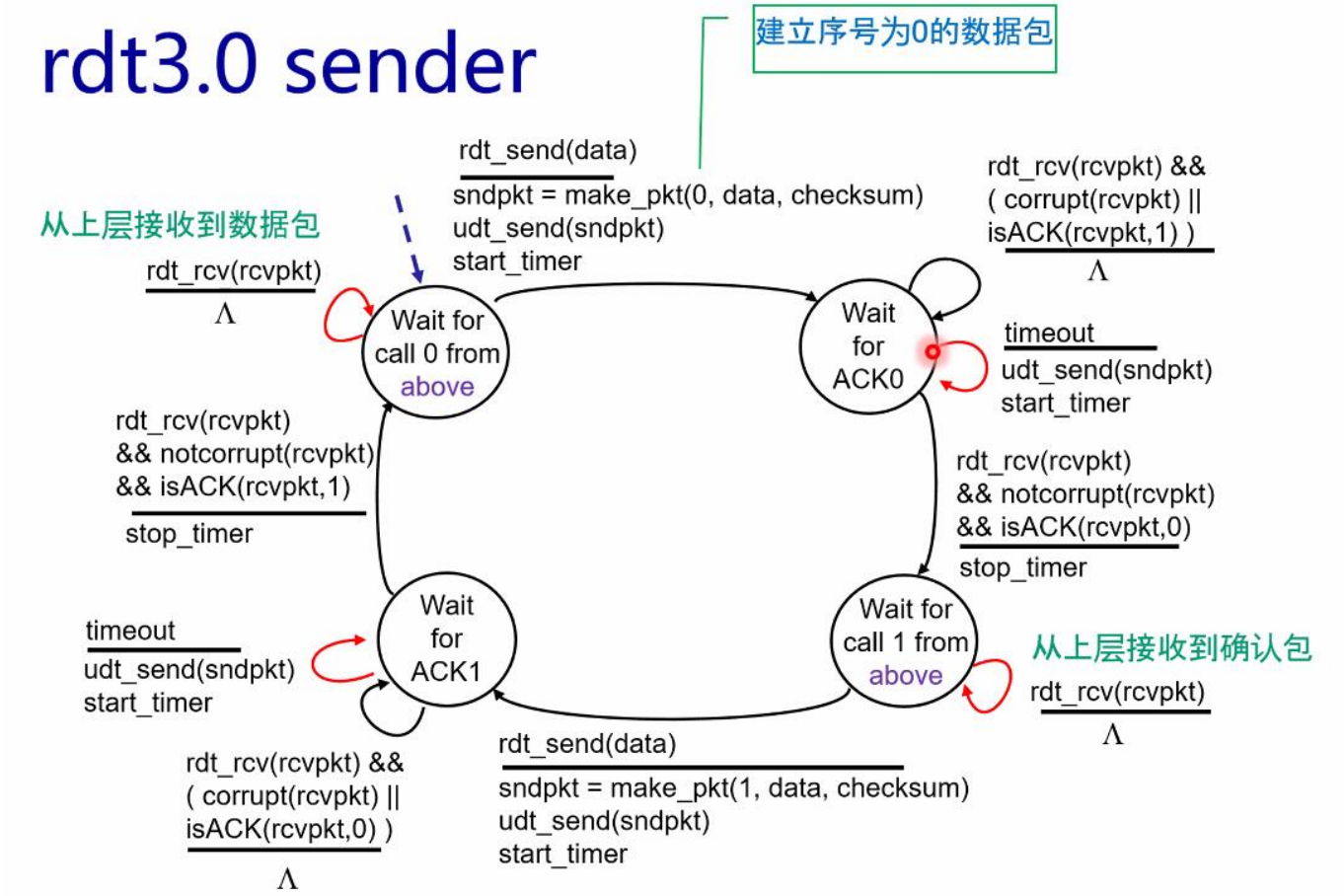
rdt2.2 去除了 NAK，因为在 0 状态下 接收方期望的是 ACK0， 因此 ACK1可以替代NAK的作用。所以只需要ACK0,和ACK1两个

4.3. 如果发生丢包怎么办?

解决方案： sender 等待ACK一定的时间

- 如果在规定的时间内没有受到ACK，就重传
- 但如果只是延迟很大（而不是丢包）？重传可能会造成冗余，但是可以用包序号（确定是不是相同的包）解决这个问题，**接收者为包指定序号**
- 发送方设定一个计时器。

4.3.1. rdt3.0

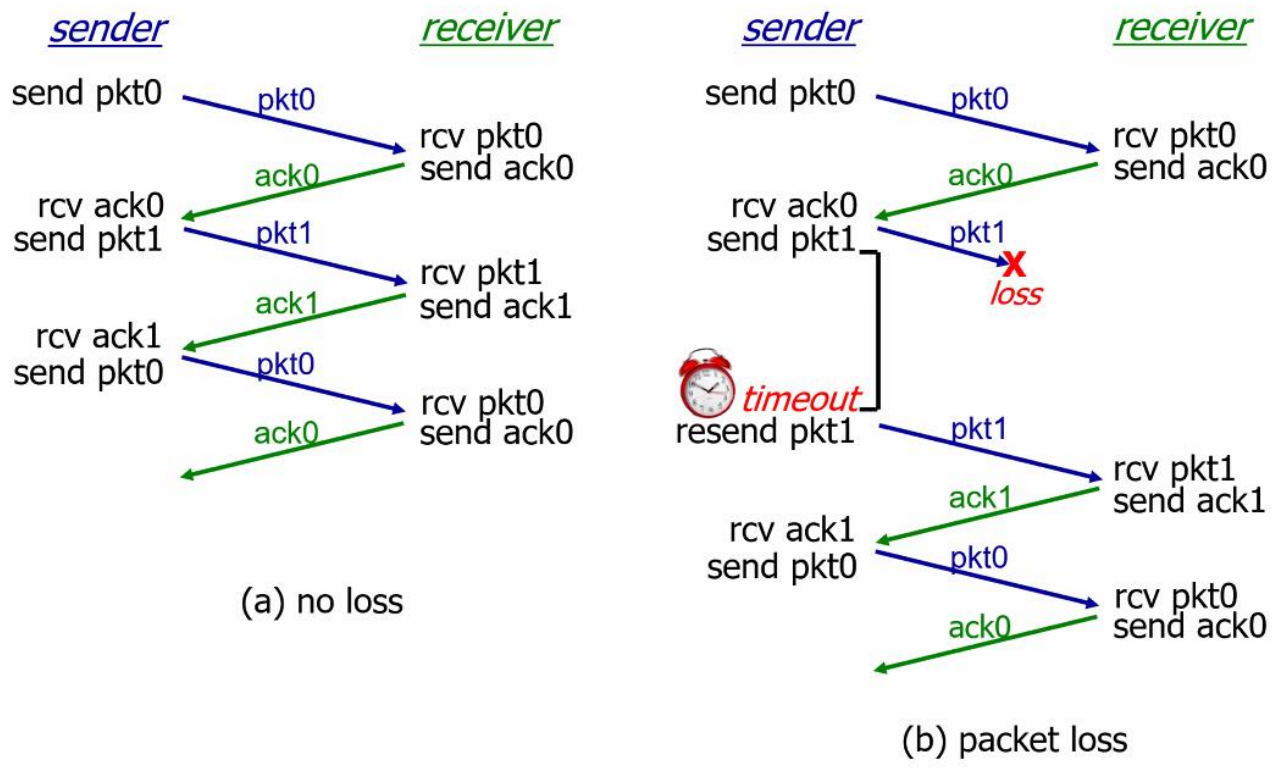


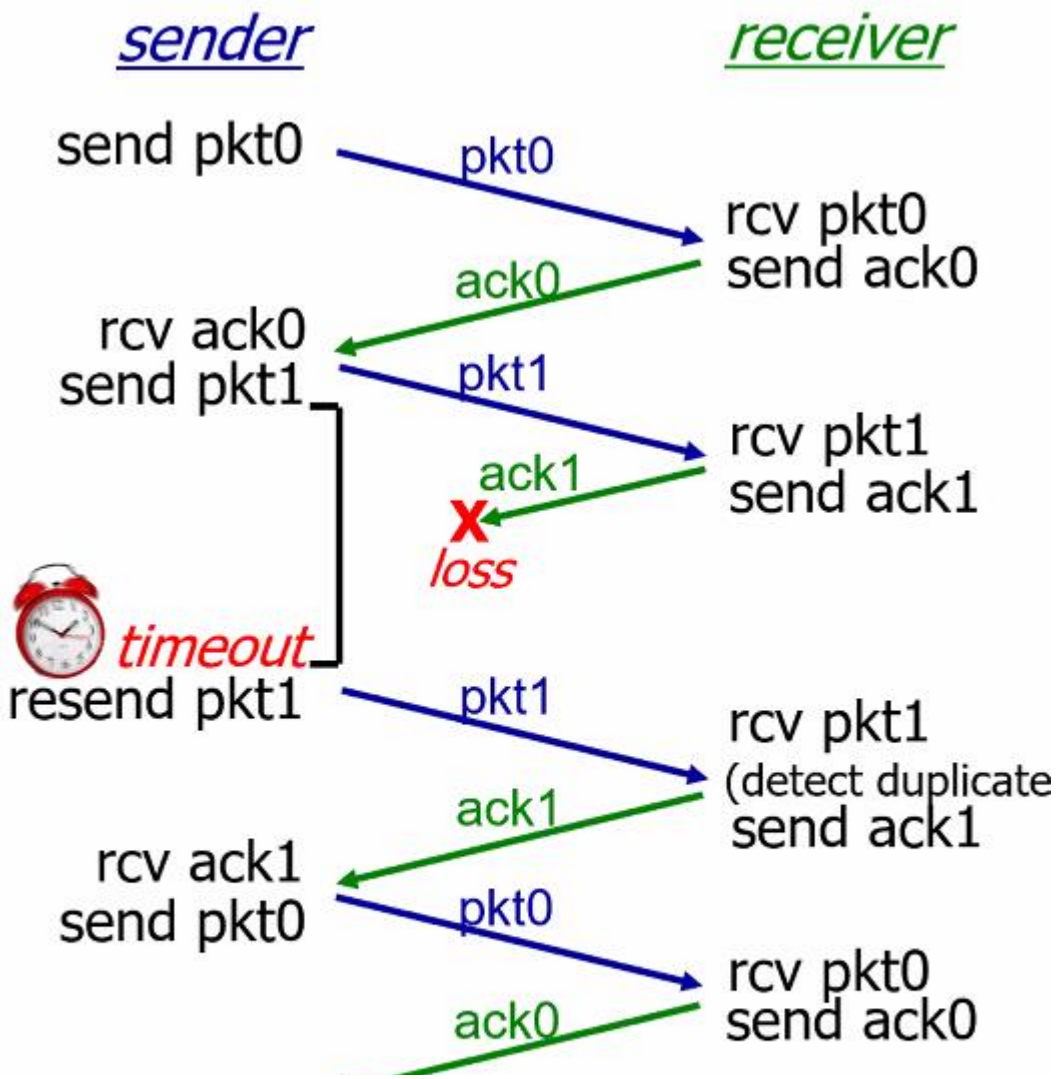
ps: 这里sender，如果timeout了，要重发数据包。注意在**wait for call 1 form above**状态里，如果收到了回复包(ACK),就直接抛弃了。因为这个包可能是延时受到的。(但是在前一步已经受到了 ACK 说明接收者已经收到了)

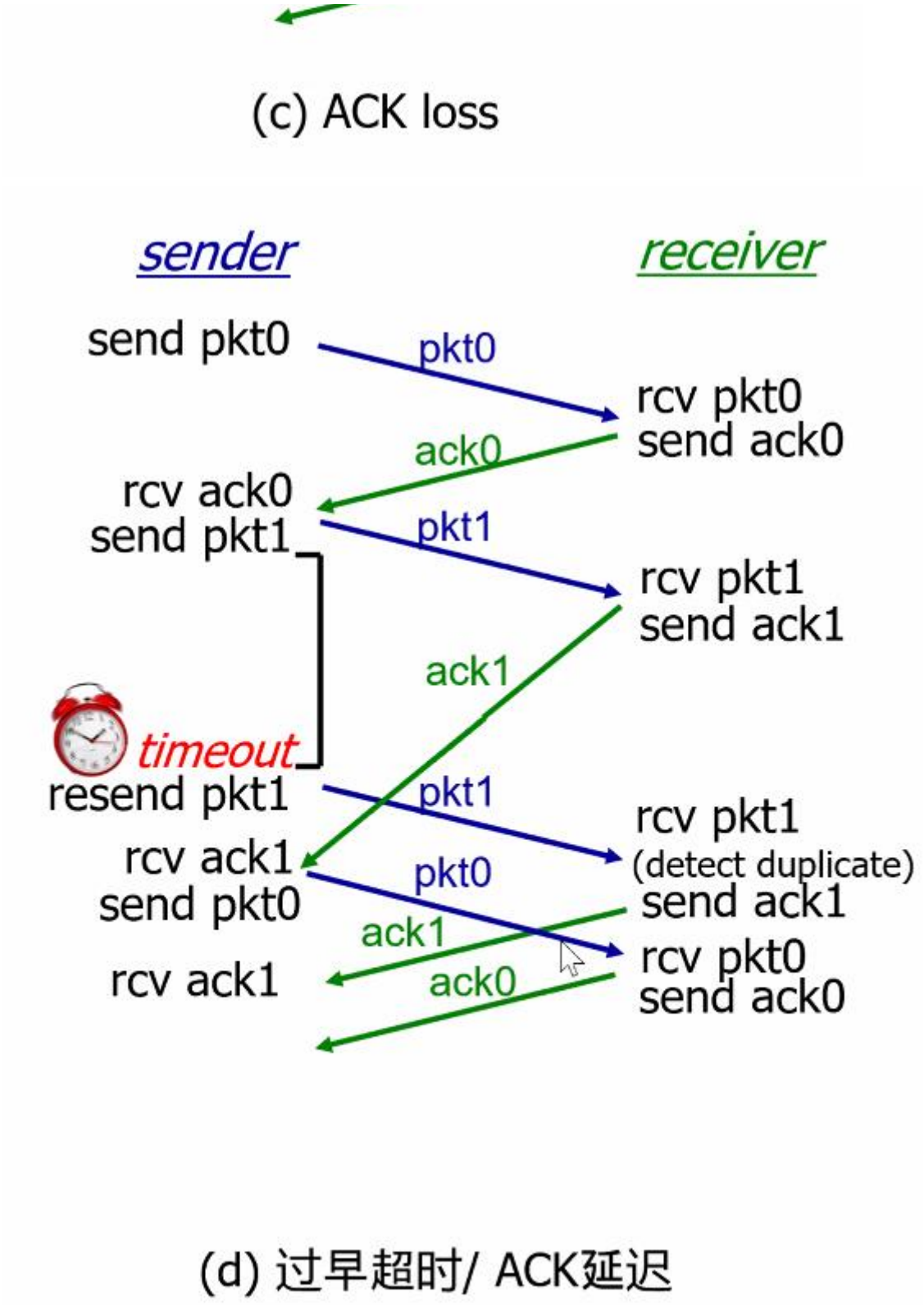
接收者同 rtd2.2

4.3.1.1. rdt3.0 动作图

rdt3.0 in action

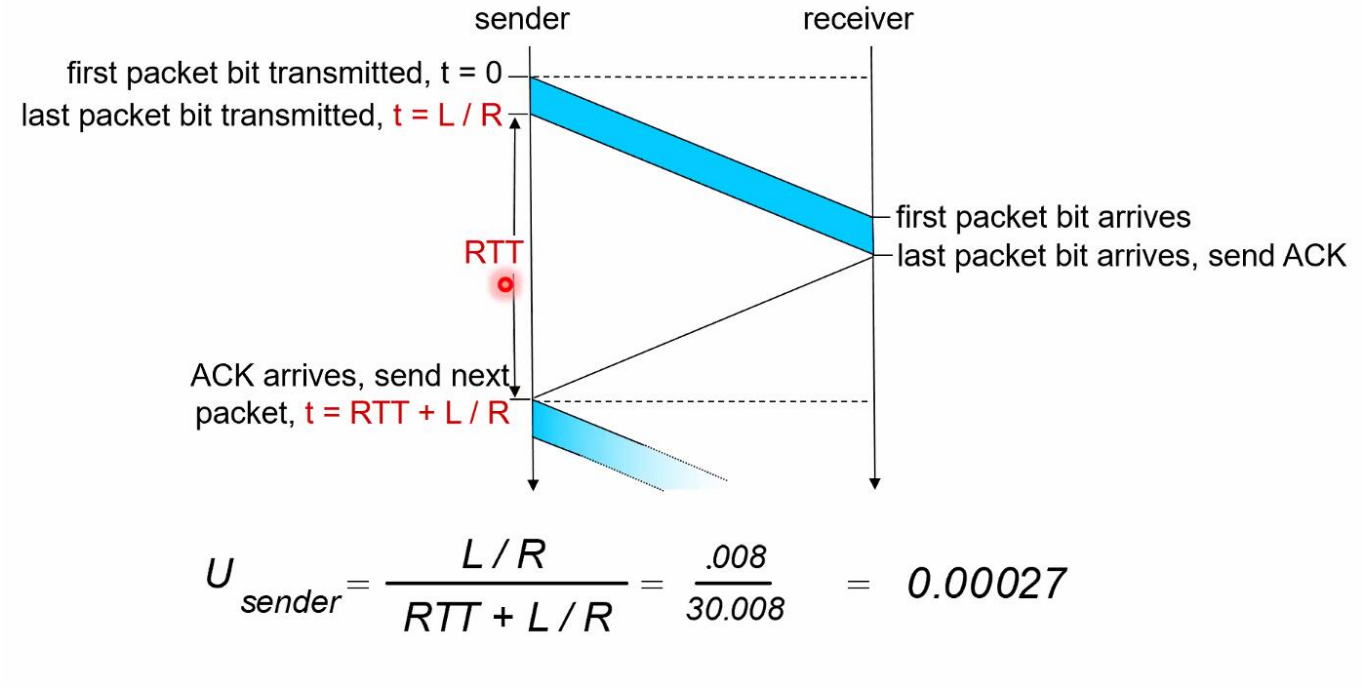






ps:在这种情况下会导致 网络上ACK包过多，会给网络造成负担（但是不会出错，因为不管受到的是哪一次的ACK，只要ACK是正确的说明接收方已经接收到了），这里也说明了需要设置一个合理的 timer 同时，在sender处要设置相应的缓存。

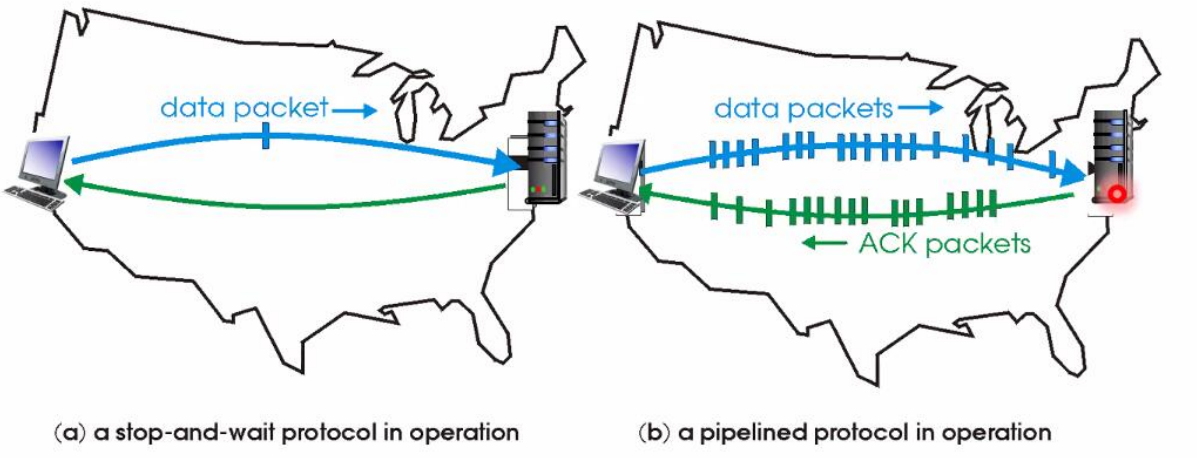
4.3.1.2. rdt3.0 性能



rdt3.0 的效率低是协议导致的，因为 **stop and wait** 机制导致，发送一个包要等待应答，在等待的时间没有新的包发送，浪费了时间。

4.4. Pipelined protocls 流水线协议

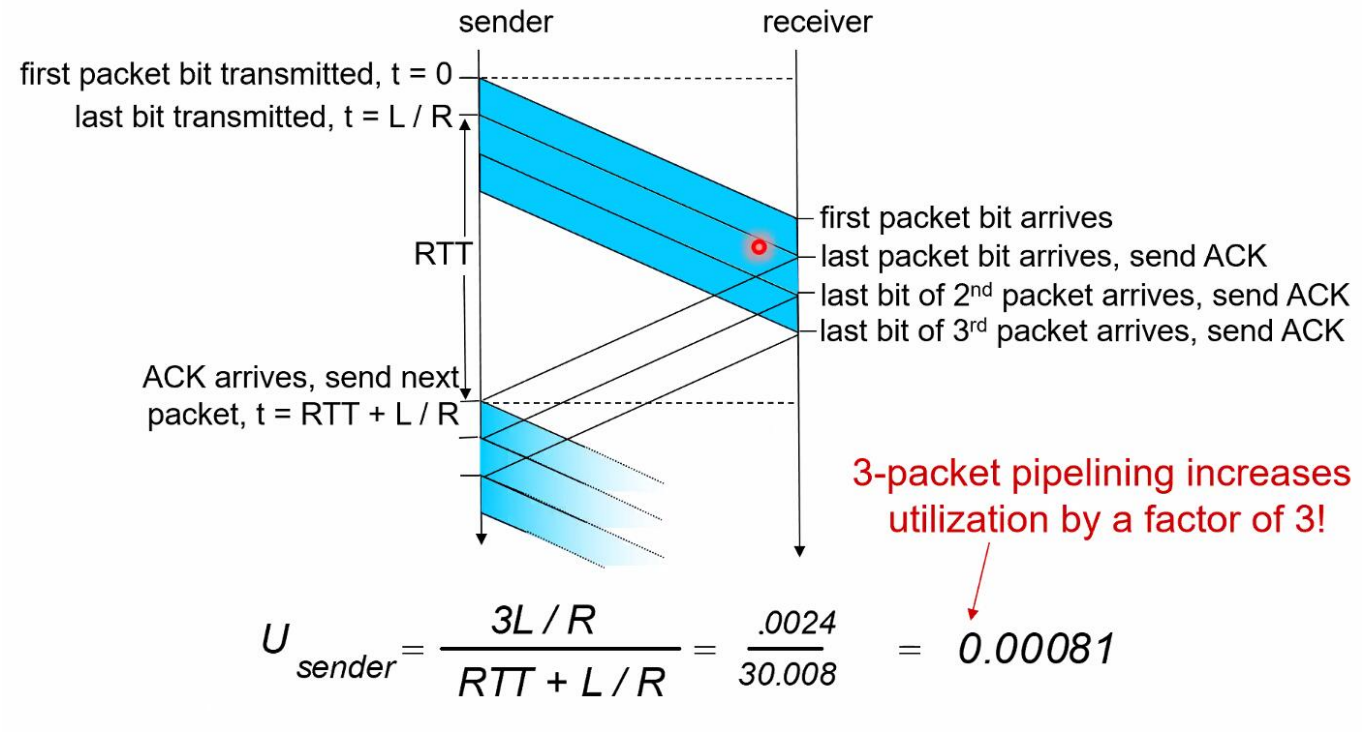
- pipeling sender allows multiple,range of sequence numbers must be increaded (一次发很多个包，因此包的序号要增加)
    - sender 和 receiver 需要缓存



- two generic forms of pipelined protocols

4.4.1. 流水线利用率增加

流水线协议可以提升利用效率。相比于 wait and stop ,通过增加在 rtt中发送其它的包来提升利用效率



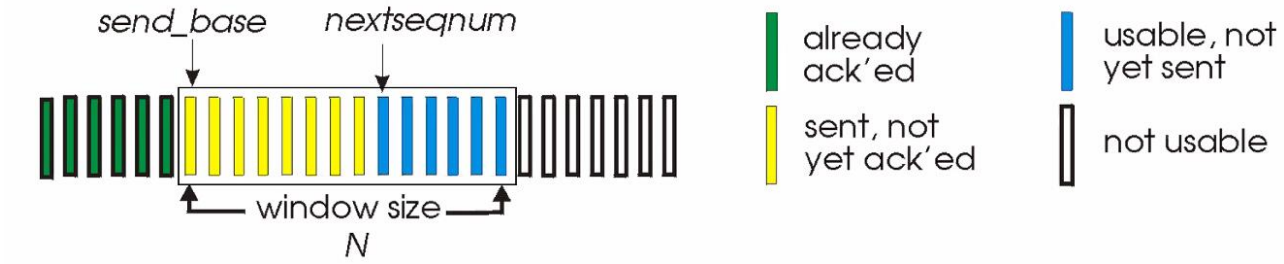
4.4.2. Go-back-N (回退N步)

- 发送端可以有N个未被确认的分组在 pipelne中
- 接收者只发送累计的ack
  - 如果序号不连续，不发送确认报文
  - 回复收到序号最大包的ack
- 发送者对最**oldest unacked packet**进行计时(防止丢包)

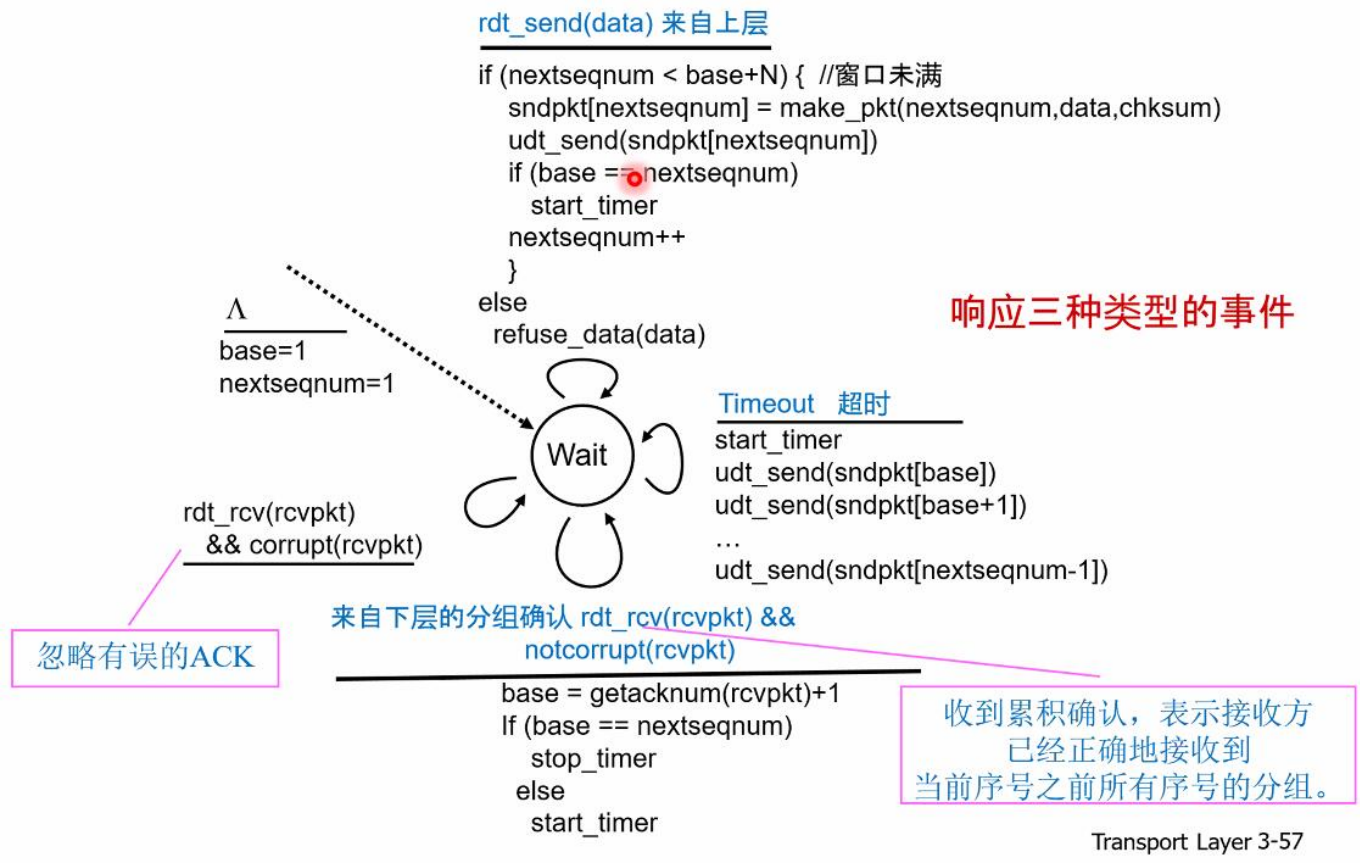
s

4.4.2.1. GBN:sender

- 分组(数据包)首部有K为的序号(seq#)
- 窗口大小为N， base:窗口开始的序号



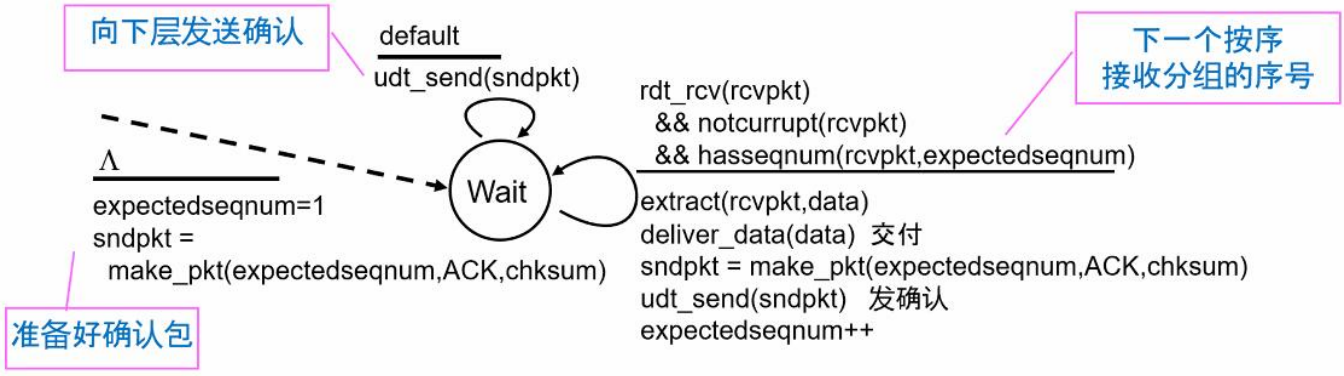
- ACK(n) : ACKs all pkts up to, including seq #n -"cumulative ACK"；累积ACK，ACK(n)表示序号小于等于n的包都已经收到了
- timer for oldest in flight pkt (为序号最小的包进行计时)
- timeout(n)，如果超时了 重传 packet n 和窗口中索引顺序号高于n,已发送但是未确认的 pkt



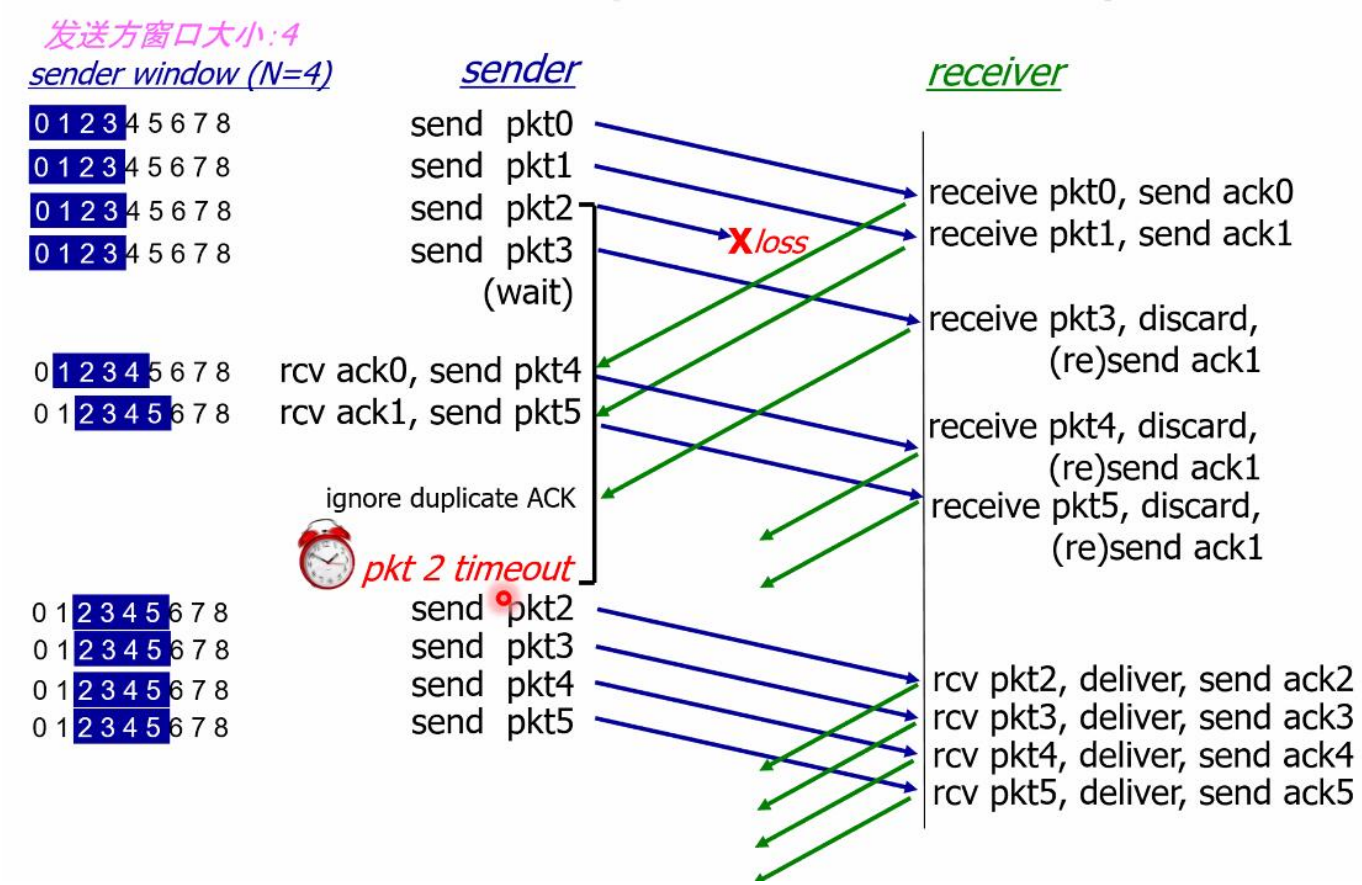
4.4.2.2. GBN:receiver

- ACK-only: always send ACK for correctly-received pkt with highest in-order seq#(只为按顺序正确接收到的序号最高的包发ACK)
  - may generate duplicate ACKs(可能产生重复的ACK，但是发送包可以处理错误的包)
  - 只需要保存一个变量:下一个按需接收的序号 expectedseqnum
- out-of-order pkt(失序包处理)
  - 丢弃，不缓存失序的包
  - 重新发送已经按序收到的序号最高包的ACK

接受方不需要缓存，收到一个就交付了



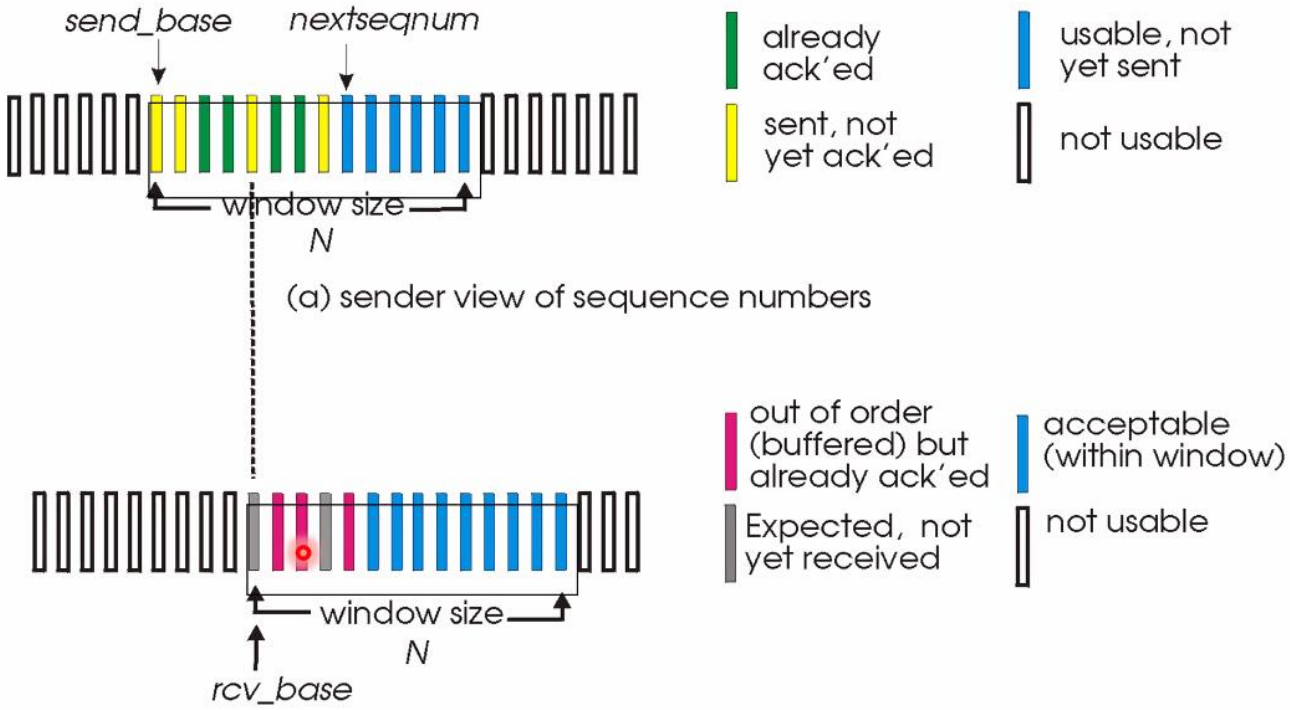
4.4.2.3. Action of GBN



这里的 receiv plt4, discard (re) send ack1 体现了 累积确认 ack (现在按序收到的序号最高的包的序号是 1)

4.4.3. Selective Repeat(选择重传)

- 发送者在流水线中最多有N个未确认的数据报
- 接收端 send individual ack for each packet(对单个数据报进行确认)
- 发送者对每一个未确认的数据报进行计时



发送方和接收方分别有一个窗口

Transport Layer 3-61

sender

data from above:

- 如果窗口中有可用 seq #， 发送 pkt

timeout(n):

- 重新发送 pkt n, restart timer

收到的ACK(n) in [sendbase,sendbase+N]:

- mark pkt n as received
- if n is smallest unACKed pkt, advance window base to next unACKed seq #（把base值提升）

receiver

pkt n in [rcvbase, rcvbase+N-1]

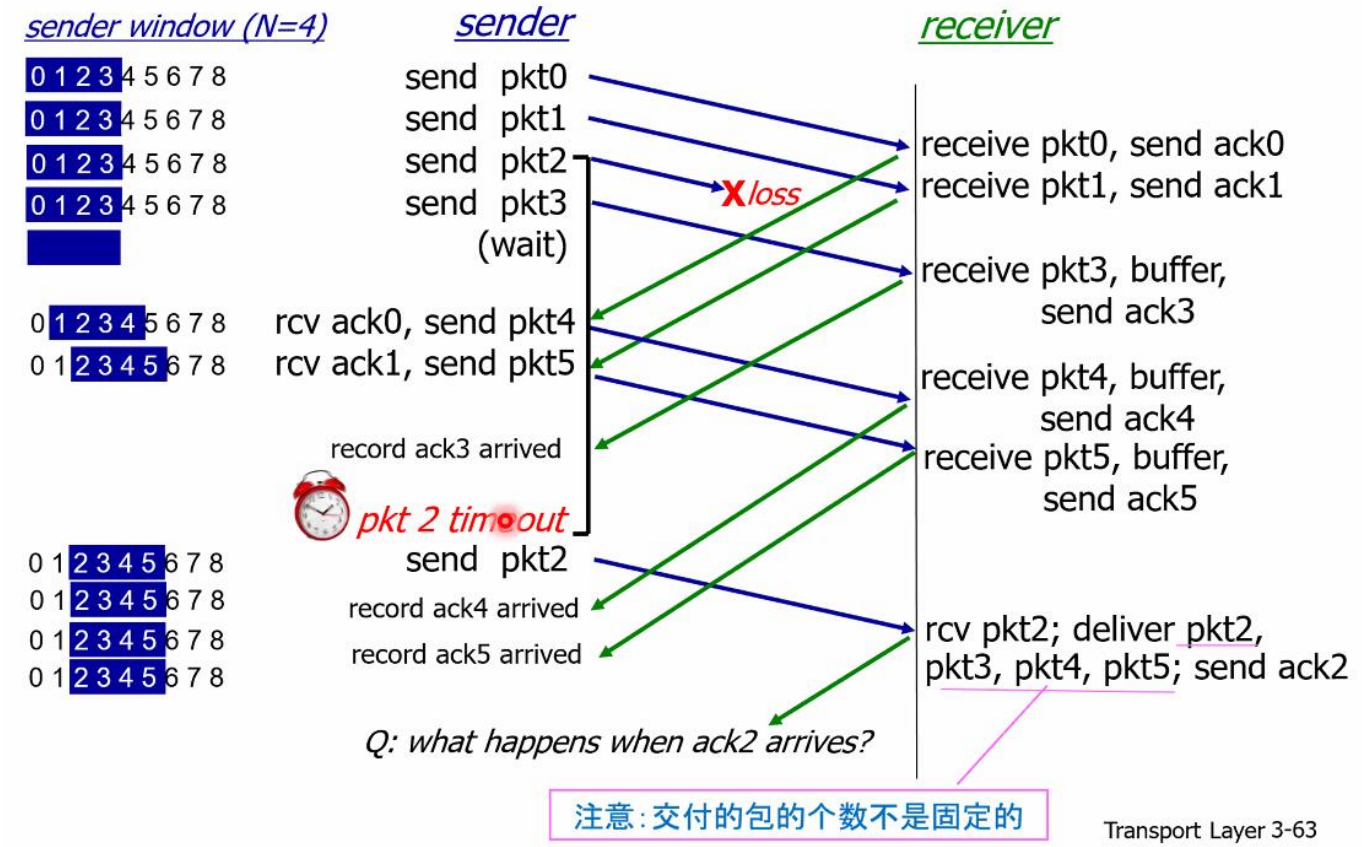
- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order pkts), 移动窗口至 next not-yet-received pkt

pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

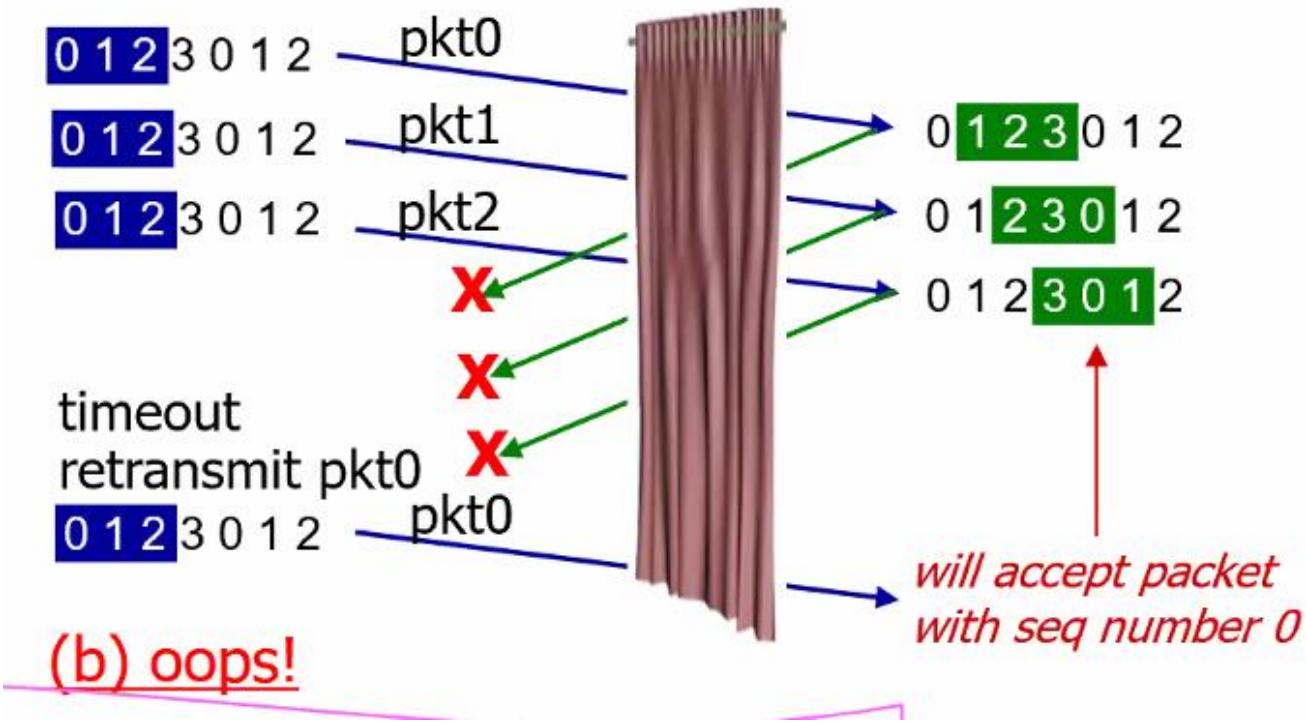
otherwise:

- ignore



最后收到 ack6之后窗口会滑到 6

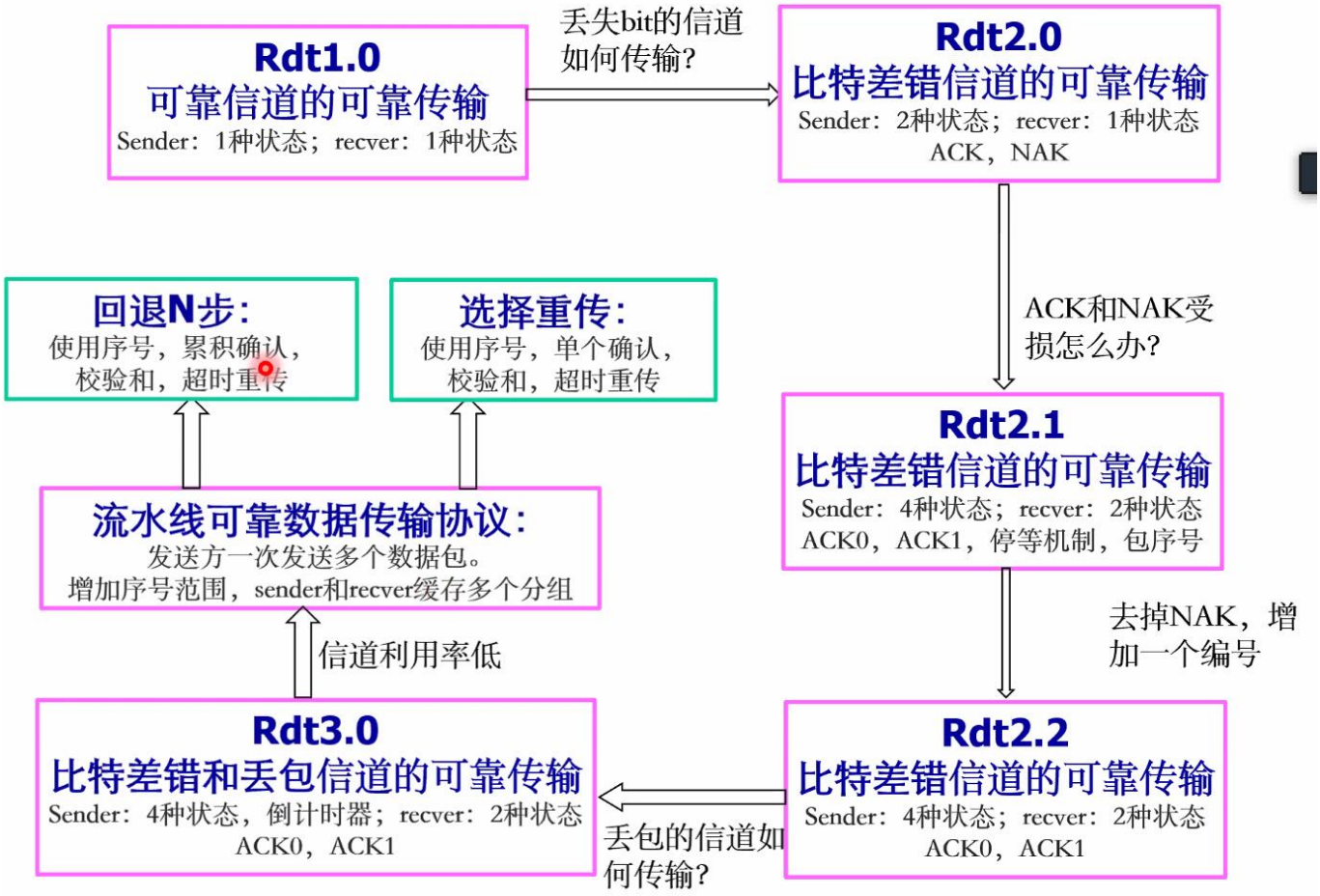
4.4.3.1. 选择重传的问题



当序号范围太小，窗口太大会出现问题，可能会出现

有一个原则: 窗口大小不大于序号范围的一半。

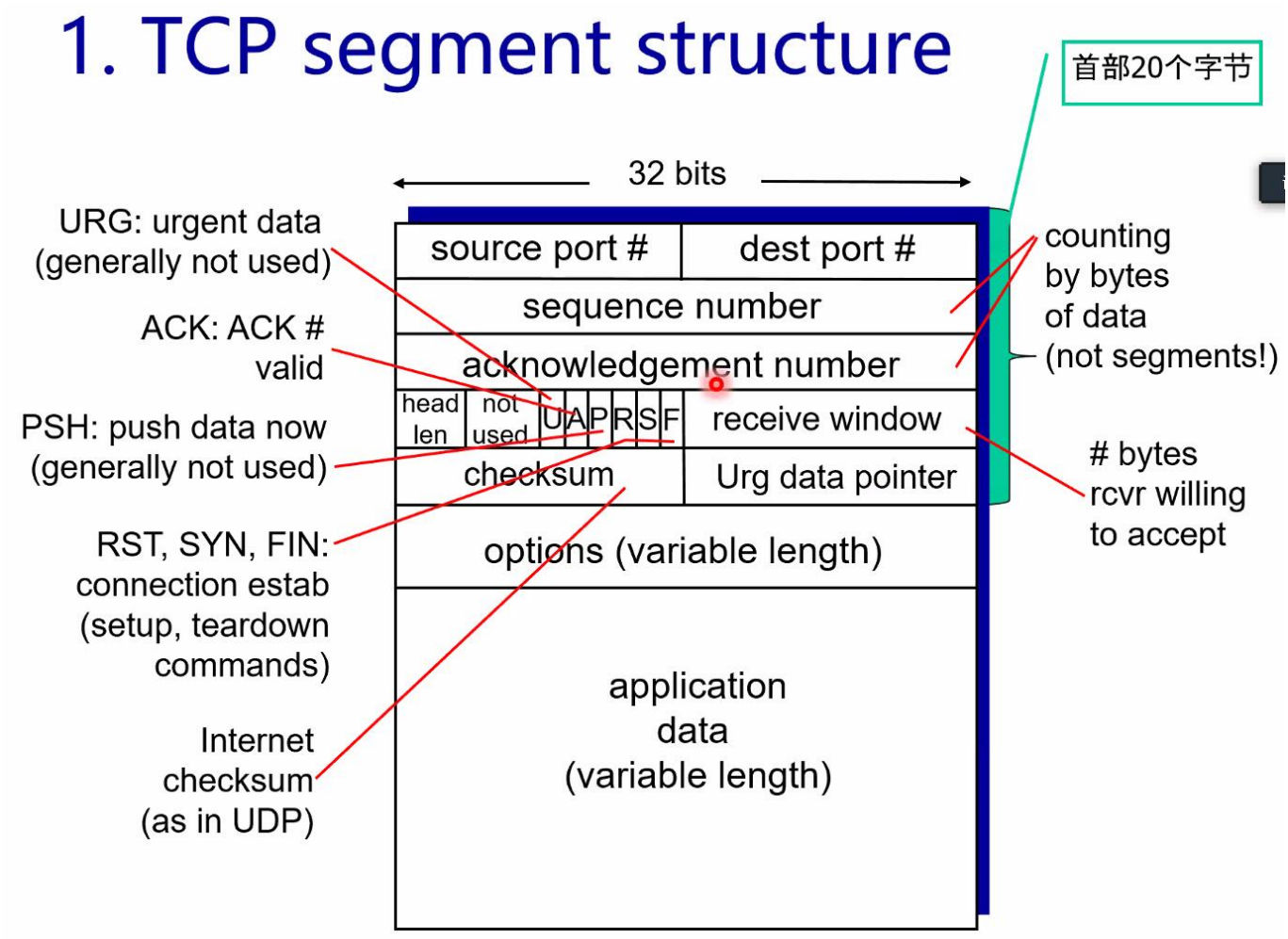
4.5. 可靠传输协议设计总结



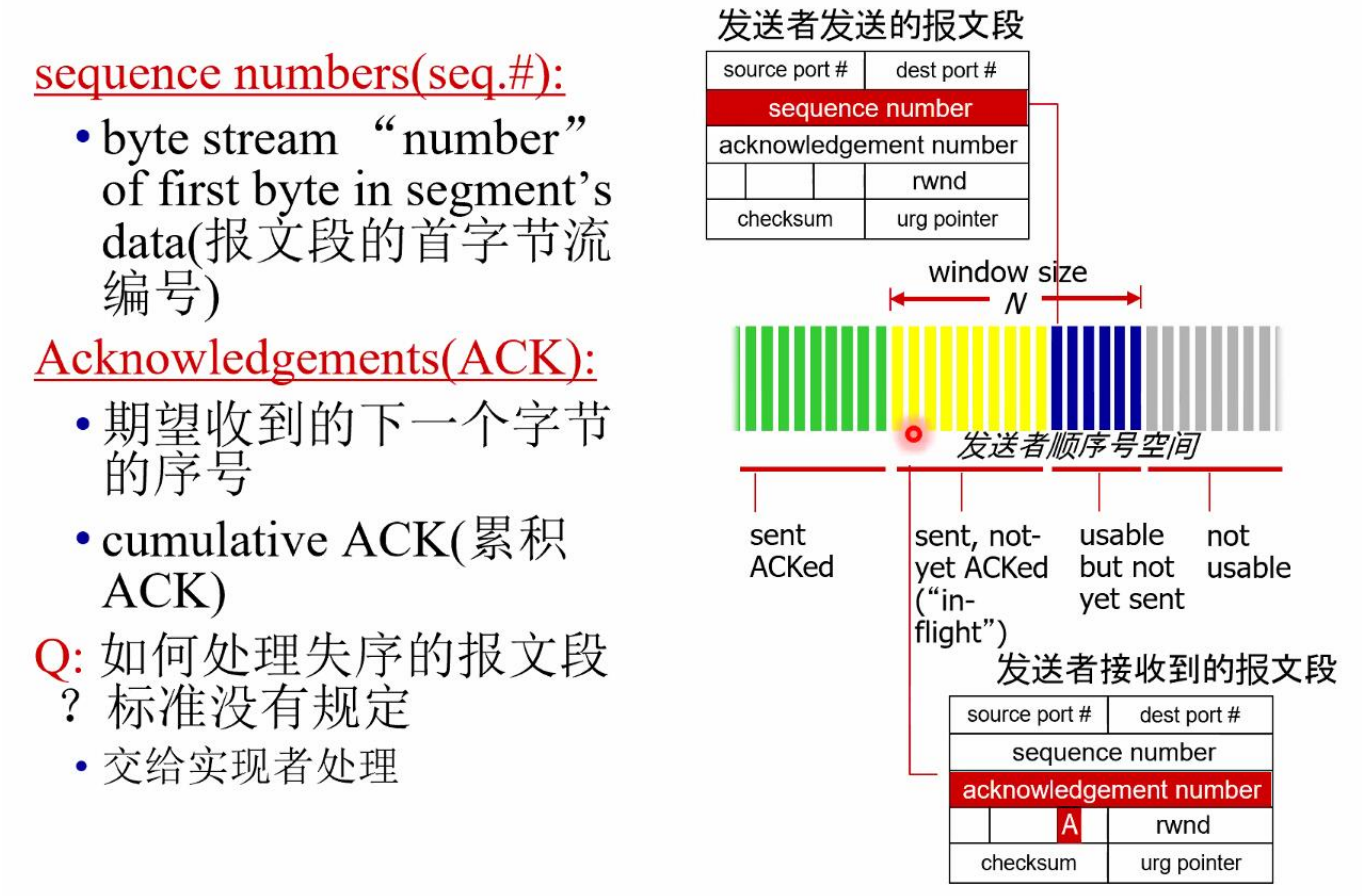
5. 面向连接的传输层协议TCP

- point-to-point(点对点)
  - one sender, one receiver
- reliable， 有序的字节流
  - 没有消息边界
- Pipelined(流水线)
  - TCP congestion and flow cotrol set window size
- 全双工
  - 在一个连接中进行双向的数据流动
  - MSS: maximum segment size(最大报文段长度)
- 面向连接
  - handshaking(exchange of control msgs) inits sender receriver state before data exchange
- flow contorlled 流量工智
  - sender will not overwhelm receiver
- 拥塞控制
  - 发送方根据网络情况调整发送速率

5.1. TCP segment structure



5.1.1. TCP seq.numbers , ACKs



- 每次发送是一个数据包，seq.number是数据包首字节的编号
- 为了重复利用，即使不单纯的ACK，也会填上ACK号，ACK表示期望的下一次收到的序列号(seq.number)

5.1.2. TCP如何设置超时时间

- 超时时间应该>RTT
- 但是RTT是不固定的。需要有方法能够估计RTT 采用指数加权移动平均的方案 估计 RTT
- 由于超时时间要大于RTT，但是要大多少？(以后再补)

5.2. TCP 可靠数据传输

- TCP 不可靠IP服务基础上建立可靠传输
  - pipelined segments(流水线数据段)
  - cumulative acks（累积确认）
  - single retransmission timer（单个重传定时器）
- retransmissions triggered by:
  - timeout events (超时事件)
  - duplicate acks (重复确认)

- 首先考虑简单的 TCP sender:
  - 忽略重复的acks
  - ignore flow control, congestion control

5.2.1. TCP sender events

data rcvd from app:

- create segment with seq #
- seq #：报文段第一个数据字节的字节流编号。
- 启动定时器
  - 针对最久未确认的报文段
  - 设计超时时间:

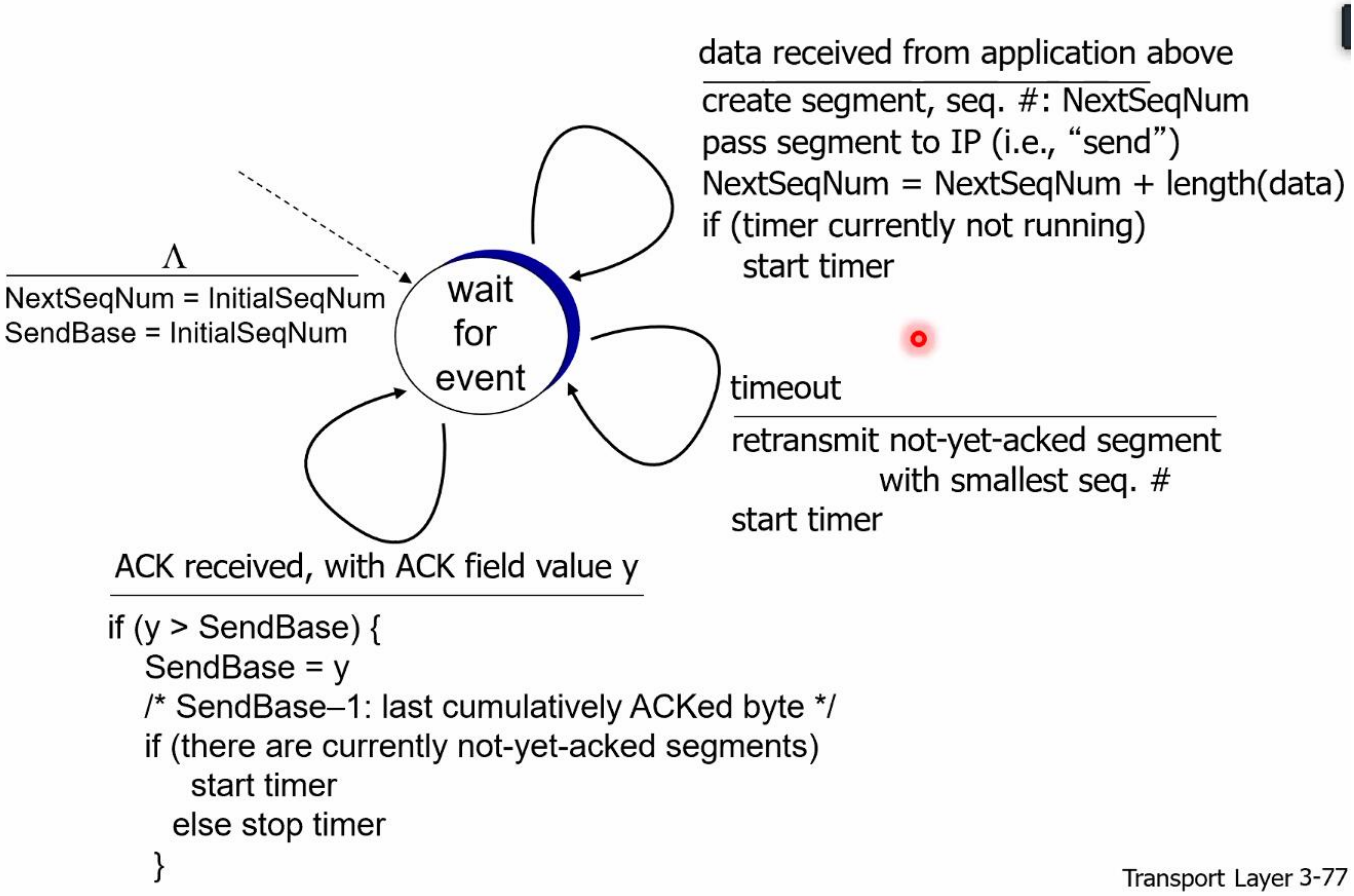
TimeOutInterval

timeout:

- 重传
- 重启定时器（restart timer）

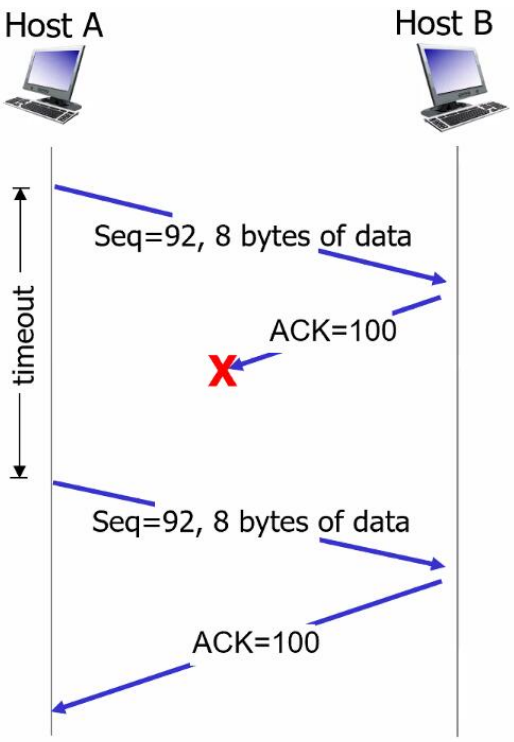
ack rcvd(接收到ACK):

- 如果接收到新的确认
  - 更新确认过报文
  - 如果有未确认段，重启定时器



Transport Layer 3-77

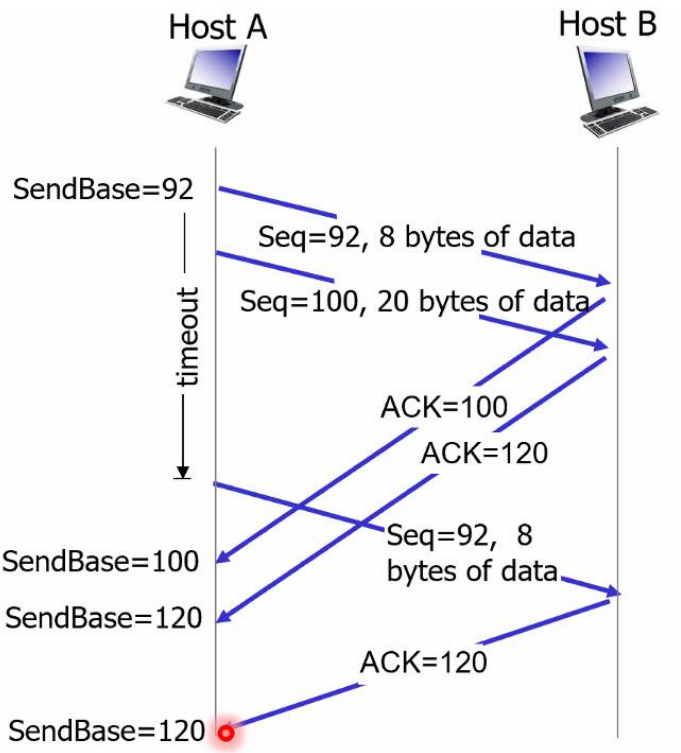
TCP : GBN(累积应答) + 选择重传(只重传序号最早的那个包,GBN是重传 序号>最早未确认的那个包的所有包)



lost ACK scenario

由于确认包丢失而重传

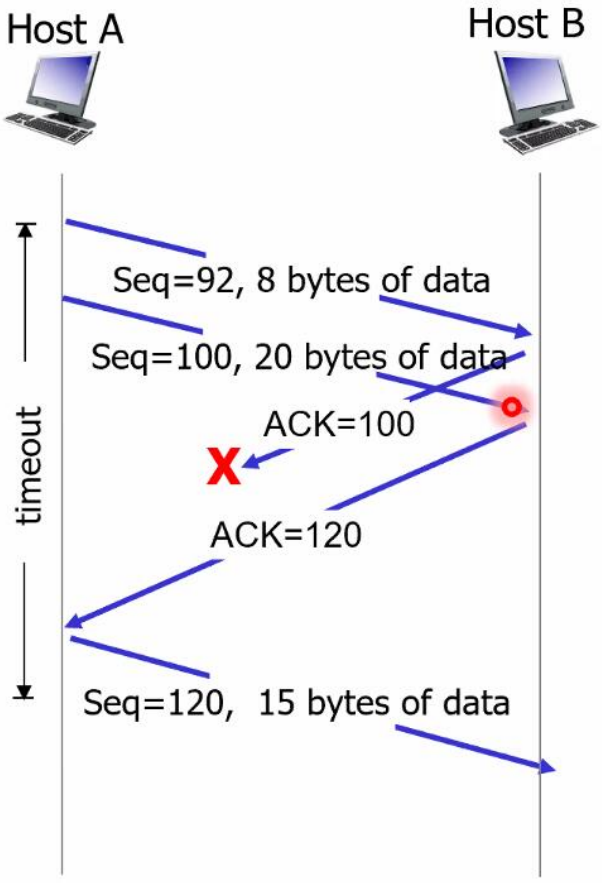
（定时器的作用）



premature timeout

只要ACK120在新的超时之前到达，

报文100没有被重传（累积应答的作用）



cumulative ACK

避免了第一个报文的重传

（累积应答的作用）

累积应答，在TCP中可以避免报文的重传

5.2.2. TCP ACK generation

event at receiver	TCP receiver action
具有所期望序号的按序报文段到达，所有期望序号之前的数据都已确认	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK（延迟应答）
具有所期望序号的按序报文段到达，另一个按序报文段等待ACK传输。	immediately send single cumulative ACK, ACKing both in-order segments 立即发送单个累积ACK，以确认两个按序报文段
比期望序号大的失序报文段到达，检测出时间间隔。	立即发送冗余ACK，指明下一个期望报文段的序号（相当于NAK）
能部分或完全填充接收数据间隔的报文到达	倘若该报文段起始于间隔的低端，则立即发送ACK（窗口）

补充解释:

1. 情况1 就是 如果收到了一个期望的包(按序到达) 那么延迟一会(500ms)再应答。延迟的原因是希望能够再收到下一个数据包，这样就可以采用累积应答的方式，只发送最新的ACK。
2. 情况2 如果在发送一个按序到达的包的ACK的时候，正好另外一个期望(按序，下一个)的数据包到达。就直接发送新的数据包的ACK(可以节省一个ACK)
3. 如果收到的数据报的序号比期望的序号大，也就是失序了，那么发送冗余的ACK(期望的序号)。相当于(NAK)
4. 如果受到的数据报能够填充数据间隔(意思就是收到数据报之后能够使得一段数据报是连续的)这时候发送ACK(按序接收的，序号最高的数据报的ACK)

ps: tcp采用累积应答，意味着序号高的ACK到达就自动表明序号低的数据报都收到了。假如出现了接收方ACK包丢失的情况，只要发送方收到了序号更高的ACK包，那么表明序号较低的数据报都收到了，就不需要再重传(ACK丢失的数据报）。