传输层

传输层服务和协议

不同主机应用进程间的逻辑通信

- 发送端
  - 将应用消息分为段 segments, 传递给网络层
- 接收端
  - 重组 segments 为消息, 传递给应用层应用

传输层有两个可用的协议TCP和UDP

多路复用和多路分解

多路分解：将报文端的数据交付到正确的套接字
多路复用：生成报文段并传输到网络层

分解原理

主机使用 IP 地址和 port 号将 segment 转发给相应的 socket

UDP分解

UDP socket 使用**两元组**区别

IP 数据报具有不同源 IP 地址不管是否一样的源端口号都定向到相同的 socket

校验和

目标：检测在被传输报文段中的差错（比如比特反转）

- 发送方：
  - 将报文段的内容视为16bit的整数
  - 校验和：报文段的加法和
- 接收方
  - 计算接收到的报文段的校验和
  - 和校验和字段比较

TCP分解

接收主机使用所有的**四个字段**来重定向 segment 到相应的 socket

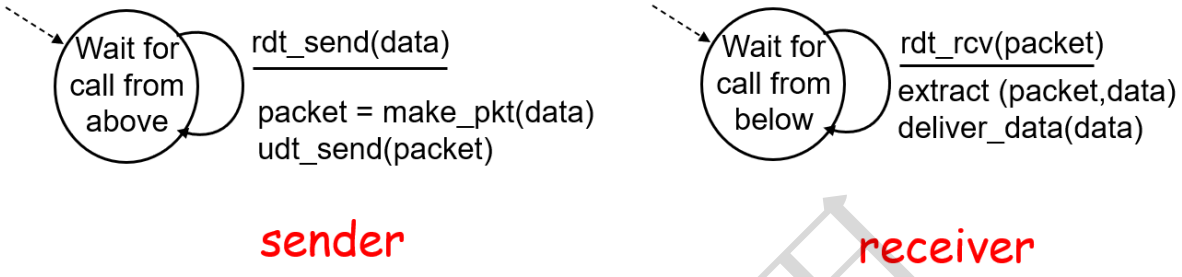
Web 服务器为不同的 client准备不同的 sockets

可信数据传输原理

rdt1.0

假设下层信道是完美信道

- sender, receiver 有自己的FSM：
  - sender 发送数据进入下层信道
  - receiver 从下层信道读取数据



直接传输

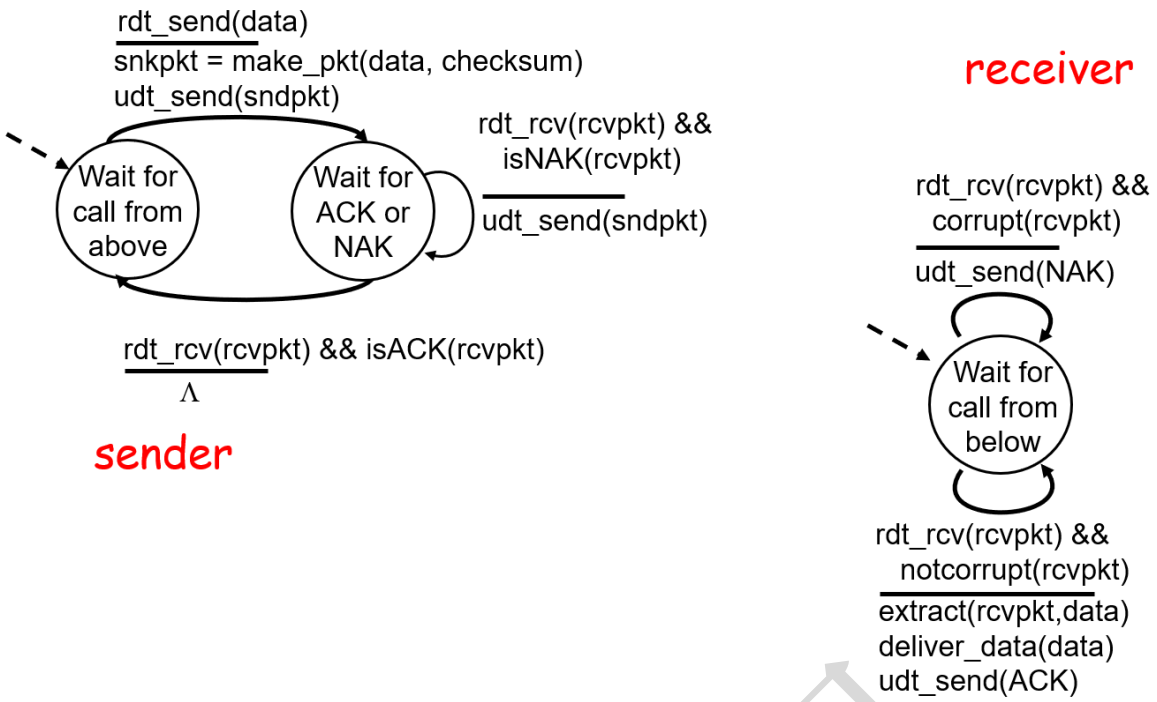
rdt2.0

假设具有比特位错误的信道

数据包在下层信道传输可能出现比特位反转错误

新机制：

- 错误检测
- 接收者反馈（ACK,NAK）
- 重传



对于发送方来说，只有接收方返回ACK后才认为发送结束

缺陷

如果ACK和NAK出错，发送方不知道接收方的情况

解决办法：发送方在每个分组中加入序号，如果ACK/NAK出错就重传当前分组，接收方丢弃重复分组,得到**rdt2.1**

rdt2.1

目的：处理混淆的ACK和NAK

- 发送方：
  - 在分组中加入序号
  - 必须检测ACK和NAK是否出错
- 接收方
  - 必须检测接收到的分组是否重复

发送方不对ACK和NAK做回应

rdt2.2

功能同**rdt2.1**但是只使用ACK

接收方对最后**正确收到**的分组发ACK

发送方当收到**重复的ACK**或**错误的ACK**时与收到NAK采取同样的操作，重传当前分组

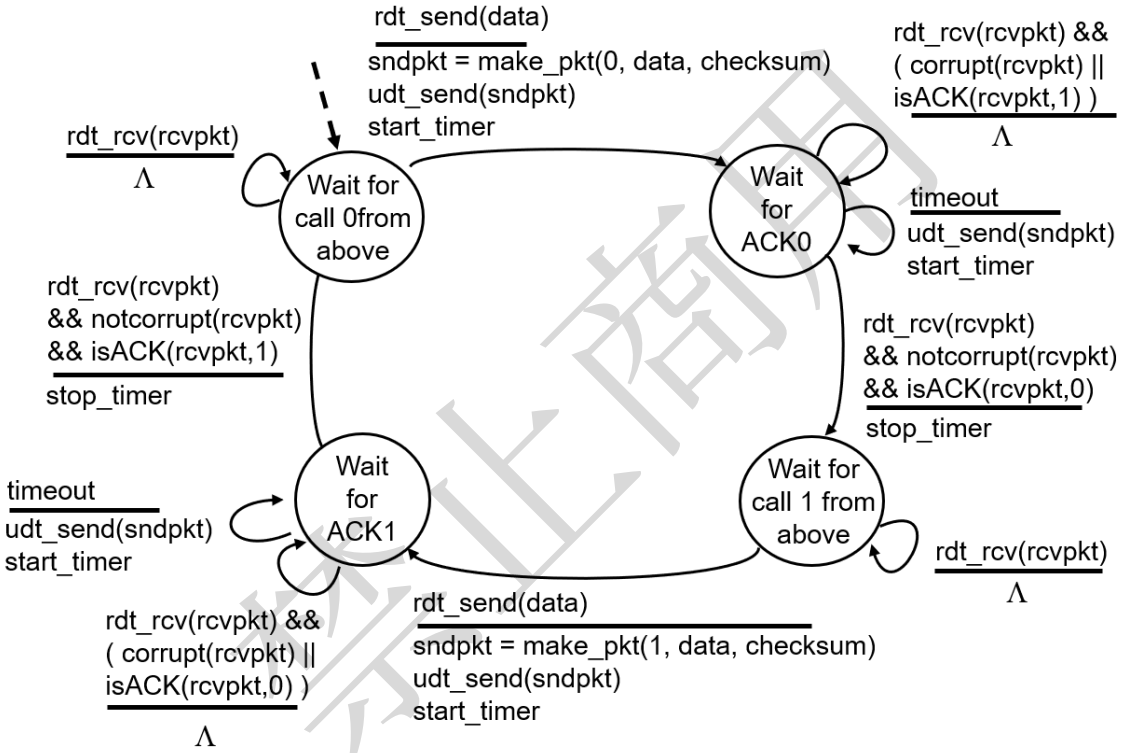
有利于一次发送多个数据单位做一个准备

rdt3.0

一个可能出错和丢包的信道

相对于rdt2.2**新机制**:

- 发送者等待ACK足够时间后就重传
- 使用计时器



缺陷

停等操作：$U_{sender} = \frac{L/R}{RTT+L/R}$

发送者忙于发送的时间和空闲的时间之比非常小，信息发送速度慢，性能很差，严重影响链路资源利用

解决方法：使用流水线

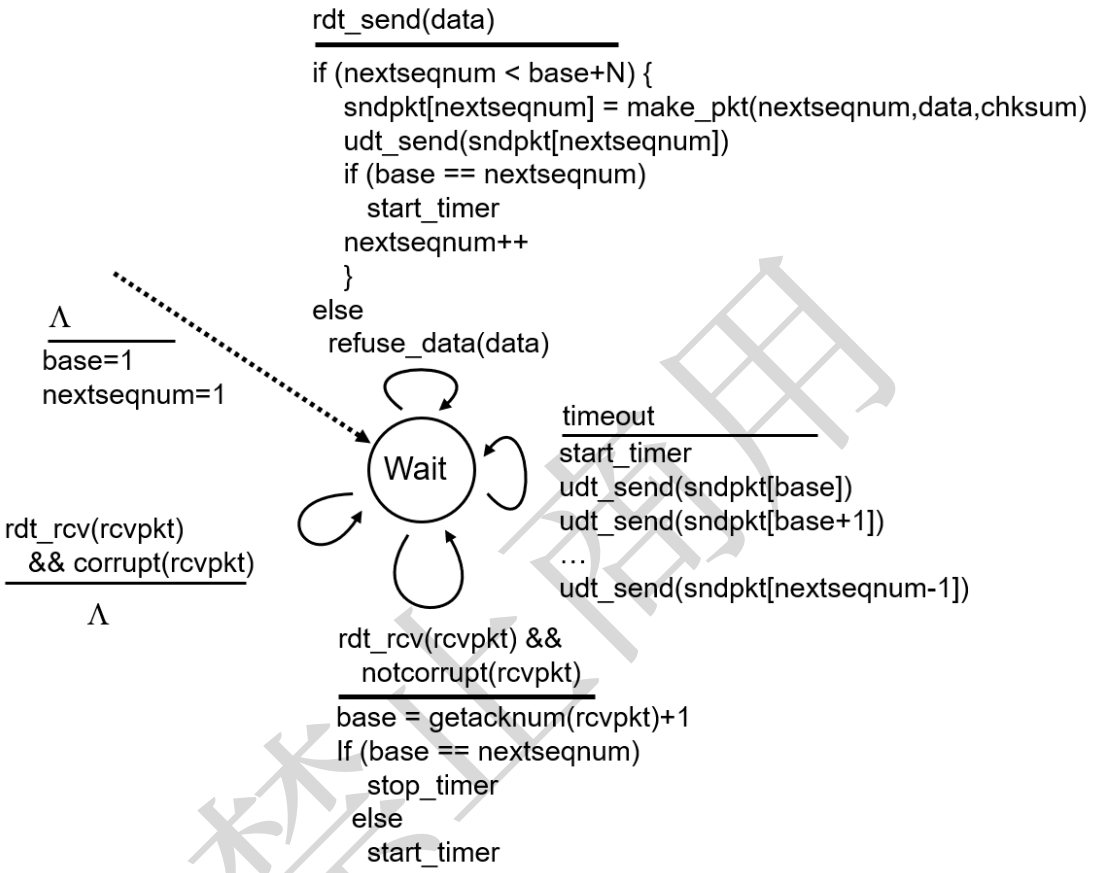
流水线协议

允许发送者发送多个等待确认的数据报

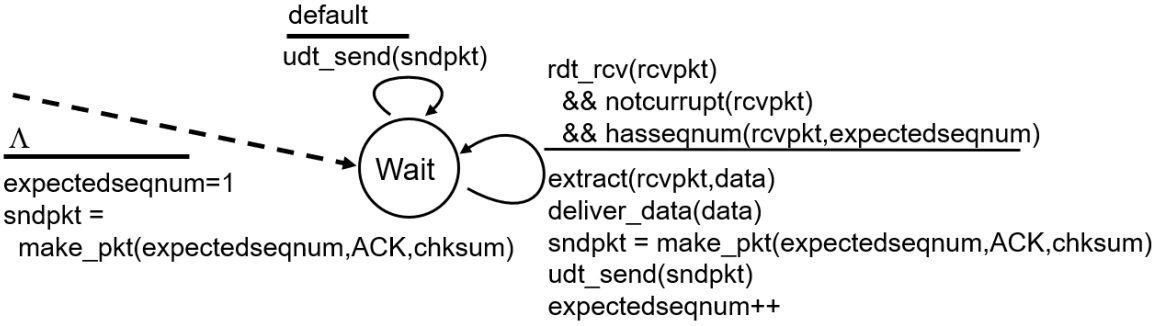
GBN（Go back N）

- 发送者在流水线中最多有N个未确认的数据报
- 接受者仅发送累计的确认
- 发送者对**最久未确认**的数据报进行计时
  - 如果计时器到点，重传所有未确认的数据报

发送者



接受者



选择重传

- 发送者在流水线中最多有N个未确认的数据报
- 接受者对单个数据报进行确认

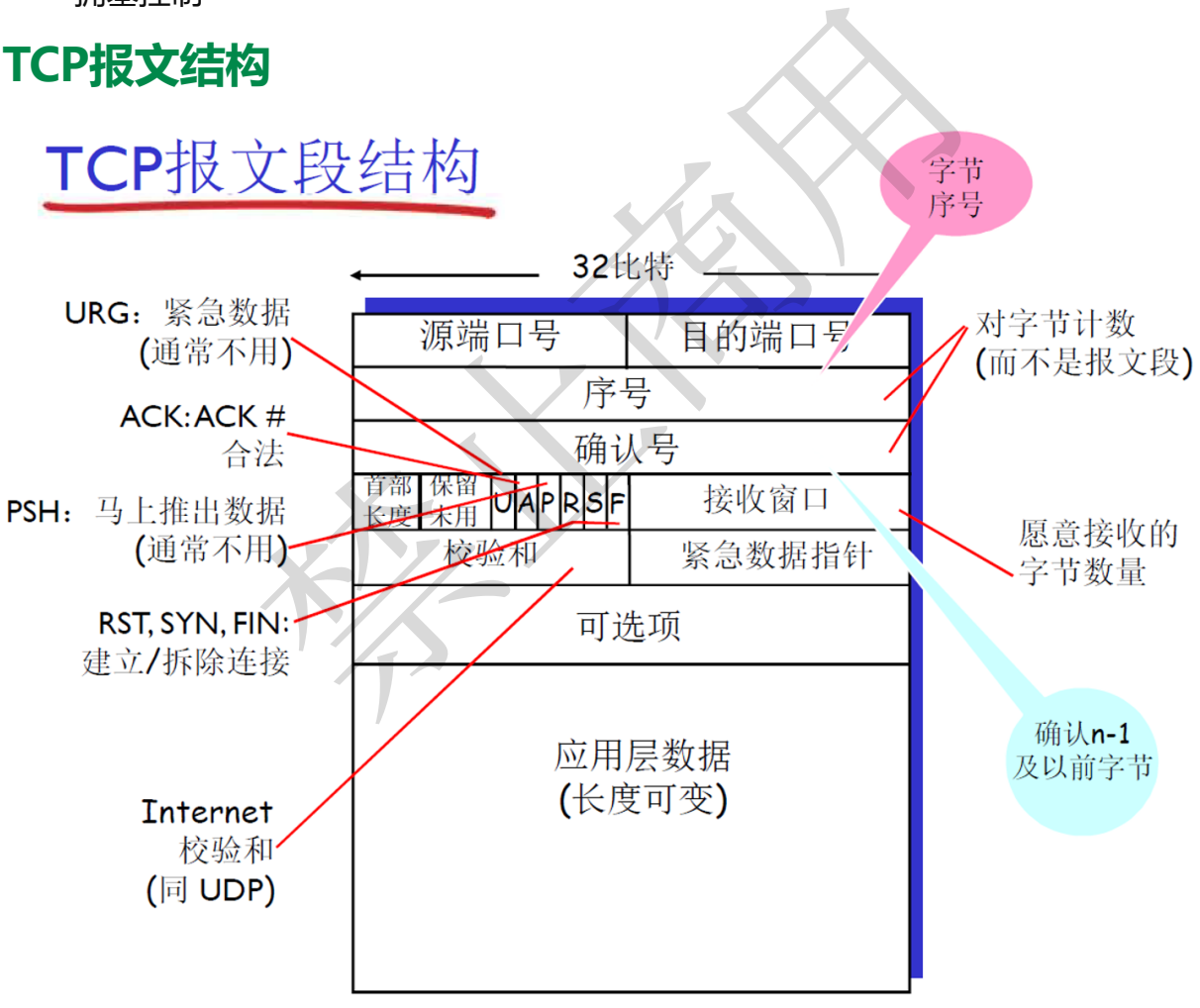
- 发送者对每个未确认的数据报计时
  - 如果计时器到点，只重传未确认的数据报

需要正确选择窗口大小和顺序序号范围

面向连接的传输TCP

- 点对点
- 全双工
- 面向连接
- 可靠、有序的字节流
- 流水线
- 流量控制
- 拥塞控制

TCP报文结构



Seq：该报文第一个字节在字节流中的序号
ACKs：从对方期望收到的下一个报文段首字母字节序号

对于TCP超时的设置

- 应该大于RTT，但是RTT会变化
- 太短

    - 超时误报
    - 出现不必要的重传
- 太长
    - 对报文段丢失反应不灵敏

估计RTT

$$EstimatedRTT = (1-a) * EstimatedRTT + a * SampleRTT$$

典型取值$a = 0.125$

安全边界

$$DevRTT = (1-b) * DevRTT + b * |SampleRTT - EstimatedRTT|$$

典型取值$b = 0.25$

超时设置

$$TimeoutInterval = EstimatedRTT + 4 * DevRTT$$

TCP实际使用的协议

TCP的差错恢复机制可以看成是GBN和SR的混合体

TCP在减小定时器开销和重传开销方面要优于GBN 和 SR

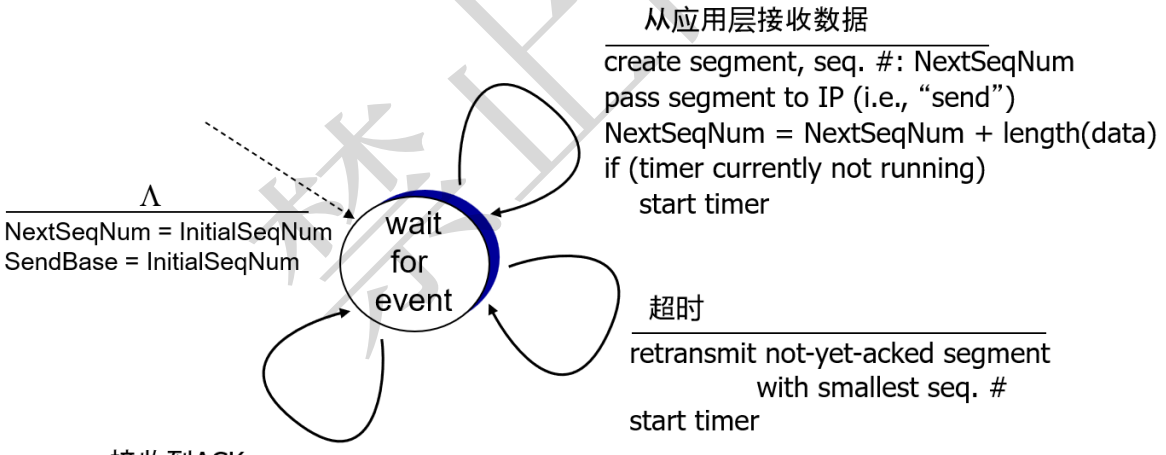
接收方

- 使用累积确认
- 缓存失序的报文段
- 对失序报文段发送重复ACK
- 增加了推迟确认

事件	TCP 接收方动作
期望的报文段到达。所有期望序号及其以前的数据都已确认。	延迟的ACK。 等待下一个报文段，最多等500ms。 如果没有下一个报文段，则发送一个ACK。
期望的报文段到达。但前一个ACK报文段还在等待。	立即发送单个累计的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，检测到数据流中的间隔。	立即发送冗余ACK，指明下一个期望报文段的序号
能部分或完全填充接收数据间隔的报文段到达	倘若该报文段起始于间隔的低端，则立即发送ACK。

发送方

- 只对最早未确认的报文段使用一个定时器
- 超时后仅重传最早未确认的报文段
- 增加了快速重传



接收到ACK

```
if (y > SendBase) {
   SendBase = y
   /* SendBase–1: last cumulatively ACKed byte */
   if (there are currently not-yet-acked segments)
       start timer
     else stop timer
   }
```

快速重传

在定时器过时之前重发报文段

通过重复的ACK来检测报文的丢失，如果发送方收到同一数据的三次冗余ACK就重传最小序号段

TCP流控

发送者通过控制发送速度，消除接收方缓存溢出的可能性

UDP不需要流控，当应用进程消费数据不够快时，接收缓存溢出，报文数据丢失，UDP不负责任。

接收缓存中可用空间称为**接收窗口**

非零窗口通告

- 发送方
  - 当接收窗口为0时，发送方必须停止发送
  - 发送方收到“零窗口通告”后，可以发送“零窗口探测”报文段从而接收方可以发送包含接收窗口的响应报文段
- 接收方
  - 当接收窗口变为非0时，接收方应通告增大的接收窗口

实现

- 发送端收到零窗口通告时，启动一个坚持定时器
- 定时器超时后，发送端发送一个零窗口探测报文段（序号为上一个段中最后一个字节的序号）
- 接收端在响应的报文段中通告当前接收窗口的大小
- 若发送端仍收到零窗口通告，重新启动坚持定时器

糊涂窗口综合症

- 当数据的发送速度很快、而消费速度很慢时，零窗口探测的简单实现带来以下问题：
  - 接收方不断发送微小窗口通告
  - 发送方不断发送很小的数据分组
  - 大量带宽被浪费

解决方案

接收方启发策略

- 接收端避免糊涂窗口综合症的策略：
  - 通告零窗口之后，仅当窗口大小显著增加之后才发送更新的窗口通告
  - 什么是显著增加：窗口大小达到缓存空间的一半或者一个MSS，取两者的较小值

- TCP执行该策略的做法：
  - 当窗口大小不满足以上策略时，推迟发送确认（但最多推迟500ms，且至少每隔一个报文段使用正常方式进行确认），寄希望于推迟间隔内有更多数据被消费

发送方启发式策略

发送方应积聚足够多的数据再发送，以防止发送太短的报文段

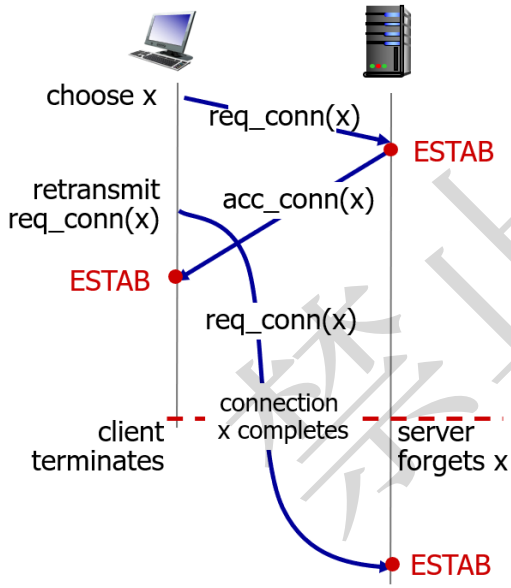
使用Nagle算法确定等待时间

Nagle算法的优点：

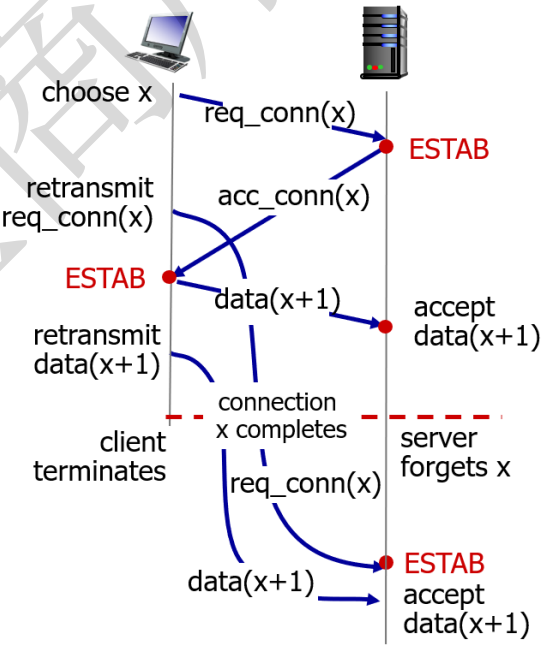
- 适应网络延时、MSS长度及应用速度的各种组合
- 常规情况下不会降低网络的吞吐量

连接管理

2次握手失败场景：



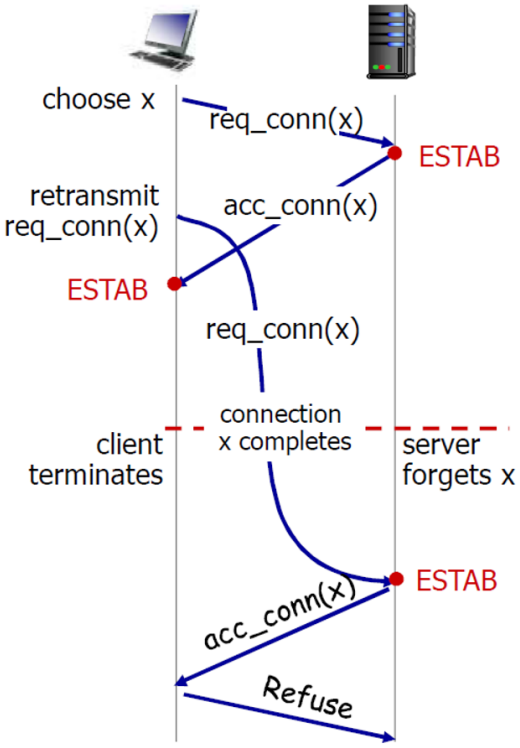
半连接
(只在服务器端维护了连接!)

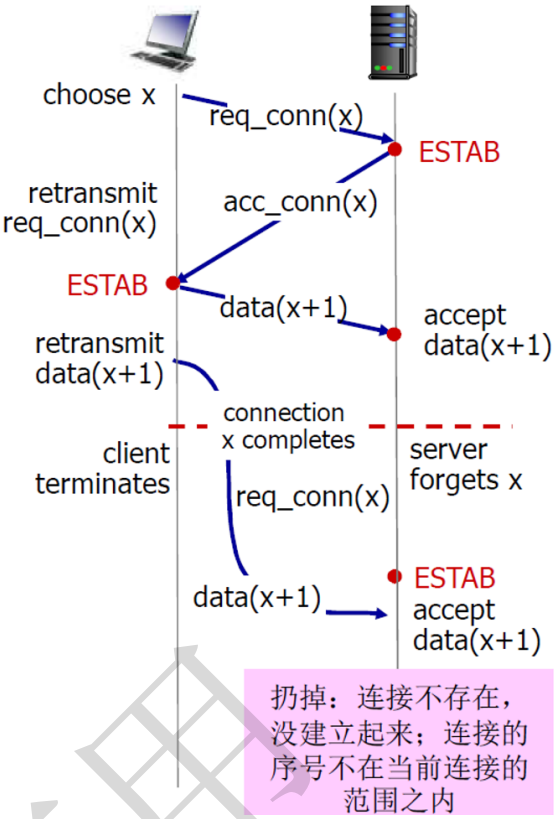


老的数据被当成新的数据接受了

3次握手

可以解决半连接和接收老数据的问题



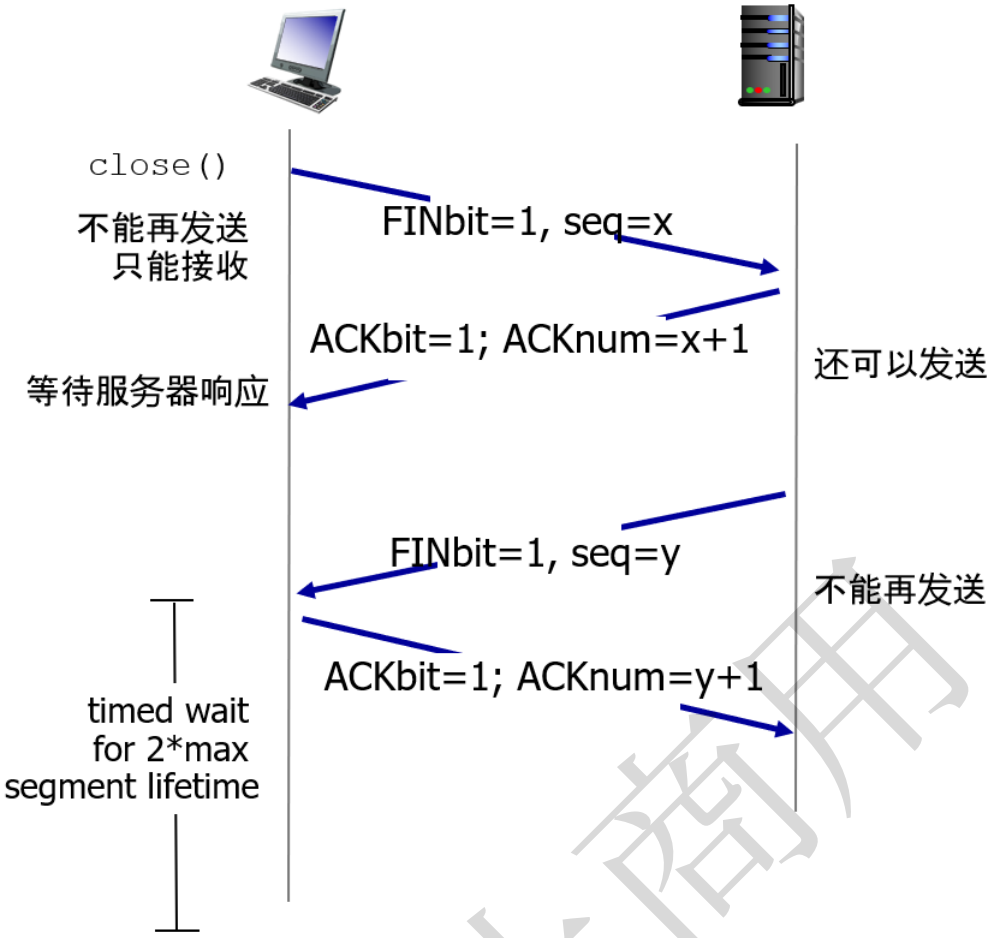


初始序号选取

必须避免新、旧连接上的序号产生重叠

- 基于时钟的起始序号选取算法：
  - 每个主机使用一个时钟，以二进制计数器的形式工作，每隔ΔT时间计数器加1
  - 新建一个连接时，以本地计数器值的最低32位作为起始序号
  - 该方法确保连接的起始序号随时间单调增长
- ΔT取较小的值（4微秒）：
  - 确保发送序号的增长速度，不会超过起始序号的增长速度
- 使用较长的字节序号（32位）：
  - 确保序号回绕的时间远大于分组在网络中的最长寿命

关闭连接



拥塞控制原理

原因

- 路由器缓冲有限
- 丢包导致的不必要的重传
- 当一个分组沿一条路径被丢弃时,每个上游路由器用于转发该分组到丢弃该分组而使用的传输容量最终被浪费了

方法

端到端拥塞控制

- 没有来自网络的显式反馈
- 端系统根据延迟和丢失事件推断是否有拥塞

TCP拥塞控制

拥塞感知

超时事件：拥塞

有关某个段的3个重复ACK：轻微拥塞

速率控制方法

维持一个拥塞窗口的值CongWin

CongWin是动态的。

- 当超时时
  - CongWin降为1MSS，进入SS阶段开始倍增
- 3个重复ACK时
  - CongWin降为CongWin/2，进入CA阶段
- SS阶段
  - 成倍增加
- CA阶段线性增加

联合控制方法：$SendWin = \min\{CongWin, RecWin\}$

阻塞控制策略

- 慢启动
  - 连接开始时指数增加（每个RTT）直到发生丢失事件
  - 每收到一个ACK，CongWin+1
- 线性增，乘性减少
  - 丢失事件后CongWin降为1，将CongWin/2作为阈值，进入慢启动
  - 当到达阈值之后如果一个RTT没有发生丢失，CongWin+1
- 超时事件后的保守政策
  - 当超时时
    - CongWin降为1MSS，进入SS阶段开始倍增
  - 3个重复ACK时
    - CongWin降为CongWin/2，进入CA阶段

事件	状态	TCP 发送端行为	解释
以前没有收到ACK的data被ACKed	慢启动(SS)	CongWin = CongWin + MSS If (CongWin > Threshold) 状态变成 “CA”	每一个RTT CongWin 加倍
以前没有收到ACK的data被ACKed	拥塞避免 (CA)	CongWin = CongWin+MSS * (MSS/CongWin)	加性增加, 每一个RTT对CongWin 加一个 1 MSS
通过收到3个重复的ACK，发现丢失的事件	SS or CA	Threshold = CongWin/2, CongWin = Threshold+3, 状态变成 “CA”	快速重传, 实现乘性的减. CongWin 没有变成1 MSS.
超时	SS or CA	Threshold = CongWin/2, CongWin = 1 MSS, 状态变成 “SS”	进入slow start
重复的 ACK	SS or CA	对被ACKed 的segment， 增加重复ACK的计数	CongWin and Threshold 不变

TCP是公平的

网络辅助的拥塞控制

- 路由器提供给端系统反馈信息
  - 显式提供发送段可以采取的速率

ATM/ABR拥塞控制

ABR: available bit rate:

- “弹性服务”
- 如果发送端的路径 “轻载”
  - 发送方使用可用带宽
- 如果发送方的路径拥塞了
  - 发送方限制其发送的速度到一个最小保障速率上

RM (资源管理) 信元:

- 由发送端发送,在数据信元中间隔插入
- RM信元中的比特被交换机设置 (“网络辅助”)
  - NI bit: no increase in rate (轻微拥塞)速率不要增加了
  - CI bit: congestion indication 拥塞指示
- 发送端发送的RM 信元被接收端返回, 接收端不做任何改变