第五讲 网络传输

中国科学院计算技术研究所

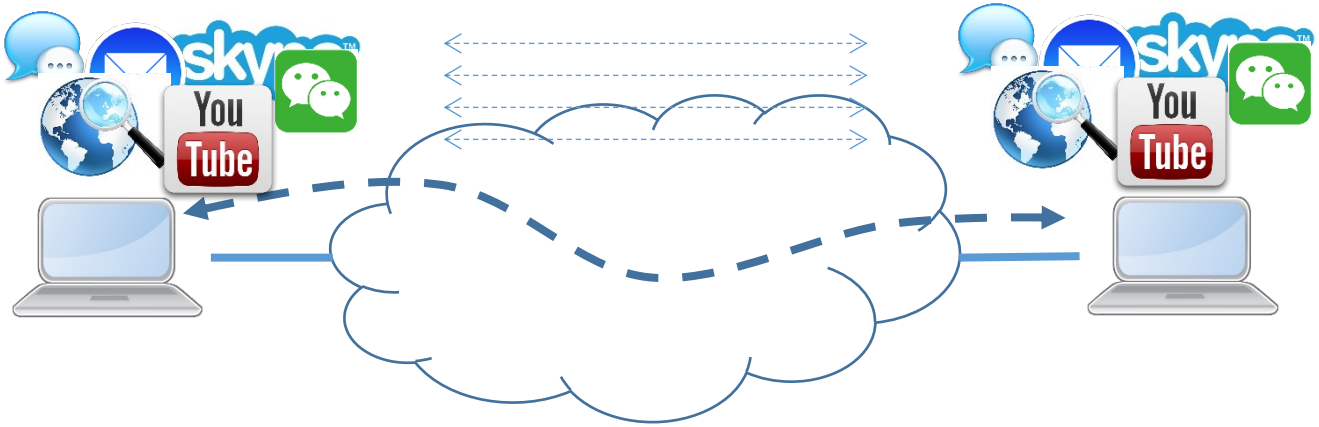
网络技术研究中心

本讲提纲

- 网络传输
  - 最简单的传输协议 UDP
  - 可靠传输协议 TCP
    - 连接管理
    - 数据传输
    - 拥塞控制
    - TCP优化
  - 多路径TCP

网络传输

- 网络层提供了端到端的连接功能
  - 无连接的、尽最大努力交付（best-effort delivery）的数据报服务
- 为了支持网络应用间的数据传输，主机端还需要实现很多功能



UDP

- 多路复用

TCP

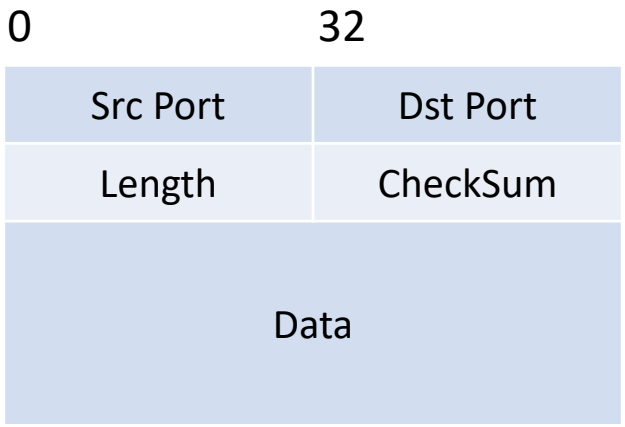
- 连接管理
- 拥塞控制
- 丢包检测与恢复
- 按序传输

传输端口和多路复用

- 端口用一个 16 位整数来标识
- 端口号的意义
  - 网络意义：标识主机提供一个特定的服务
  - 本地意义：区分本计算机应用层中的各进程
- 三类端口
  - 熟知端口， 0~1023
  - 登记端口号，1024~49151，供服务提供商使用
  - 客户端端口，49152~65535，供客户端使用
- 源、目的IP地址 + 源、目的端口
  - 构成4元组，可唯一标识互联网中的TCP或UDP传输

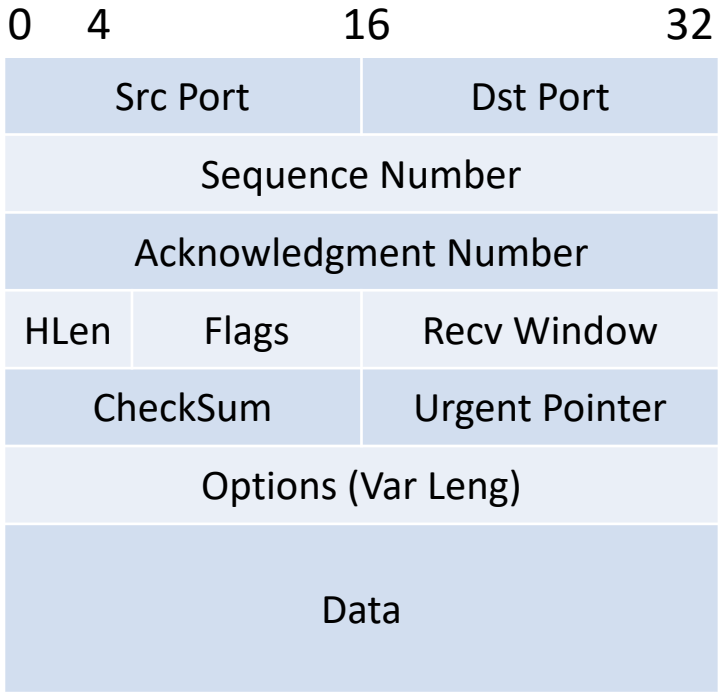
UDP (User Datagram Protocol)

- UDP：最简单的传输层协议
  - 根据端口号进行多路复用
- 为什么需要UDP协议？
  - 不需要建立连接、不需要维护状态
    - 减少启动延迟，例如DNS
  - 不需要可靠传输
    - 例如视频直播等应用
  - 作为最基本的传输层协议，上层应用可实现更多功能、按需定制
    - 例如UDT、QUIC等



TCP (Transport Control Protocol)

- 应用最广泛的传输层协议
  - 占据了目前互联网的90%以上流量
- 多路复用
  - Src Port, Dst Port
- 连接管理
  - Flags + Seq + Ack
- 可靠传输
  - Seq + Ack + Options
- 流控
  - Recv Window + Options
- 拥塞控制



TCP协议特征

- TCP完全实现在传输两端之上
  - 符合端到端原则 (end-to-end principle)
- 效率、公平性、收敛效率是TCP的三个设计目标
- TCP一直在演进
  1. 改变两端传输策略（只改端点实现）
  2. 增加TCP扩展选项
  3. 改变TCP标准头部
- TCP遵从向后兼容性 (Backward compatibility)
- TCP改进方向
  - 适应不同网络环境（无线网络、数据中心、多路径）
  - 提升应用性能（减少延迟、降低卡顿率等）

TCP演进历史

年份	名称	简介
1974	TCP	由Vent Cerf和Robert Kahn提出基本概念
1982	RFC 793	定义了现有TCP的基础：协议格式、连接管理、数据传输
1983	BSD Unix	TCP实现和大规模应用
1987	Nagle's Algorithm	合并小包，减少包发送量
1988	TCP Tahoe	拥塞控制、拥塞避免
1990	TCP Reno	Tahoe + 快速恢复
1993	TCP Vegas	基于延迟的拥塞避免
1994	ECN	显式的拥塞提醒（需交换机/路由器支持）
1996	Selective ACK	选择性确认，基于SACK的丢包恢复
1996	FACK	SACK + 更快的丢包恢复

TCP演进历史（续）

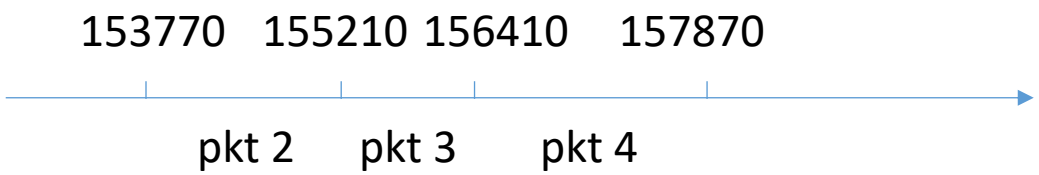
年份	名称	简介
2004	TCP NewReno	Reno + 快速恢复
2005	Fast TCP	基于丢包+延迟的拥塞控制，Akamai
2005	Compound TCP	基于丢包+延迟的拥塞控制，Windows
2007	TCP Cubic	基于三次函数的窗口管理，Linux
2010	Initial window	初始拥塞窗口由3增大到10
2010	Data Center TCP	适用于数据中心的TCP（更精确的ECN）
2011	Multi-Path TCP	多路径传输
2013	TLP	TCP流尾部丢包的快速检测
2016	RACK	TCP丢包的快速检测
2016	BBR	在不降低吞吐率的前提下减少延迟
2019	BBR v2	面向数据中心网络的BBR
2020	ORCA	性能目标导向的学习型TCP
2020	TACK	动态调节ACK频率提升在WiFi网络下的传输性能

TCP设计

- TCP连接管理
  - 建立连接、关闭连接
- TCP数据传输
  - 流控
  - 丢包恢复
- TCP拥塞控制
  - 拥塞控制算法

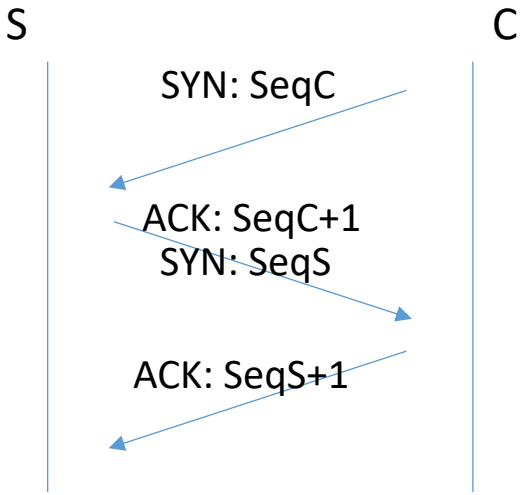
TCP序列号

- TCP是一种面向字节流的传输协议
  - 每个传输字节都对应一个32位整数序列号
    - 当数据大于32位表示时，整数进行环绕
  - 连接建立时用一随机数作为初始序列号
    - 否则容易产生安全性问题
- TCP将数据分割到不同的数据包
  - 数据包中的数据不多于 (1500 - IPHDR - TCPHDR)
  - 每个数据包的序列号是其包含的第一个字节对应的序列号



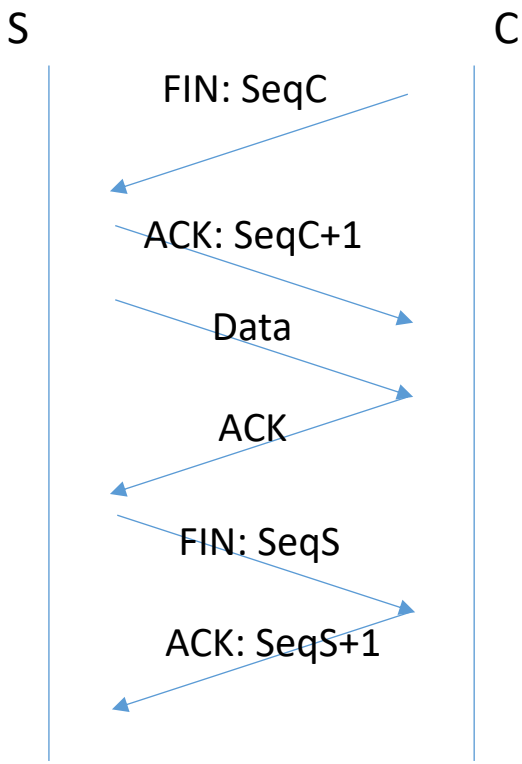
建立连接

- 双方确定建立连接用的端口号(port)
  - 被动建立连接的一方使用固定端口
    - 例如，SSH: 22, HTTP: 80, HTTPS: 443
  - 主动建立连接的一方使用随机端口
    - 由协议栈确定
- 连接双方确定自己的初始序列号并通知对方 (SYN)
  - 两端的初始序列号相互独立
- 双方在收到对方的初始序列号后，回复确认 (ACK)
  - 表示收到对方的初始序列号
- 第一个ACK通常和第二个SYN合在一个数据包中
  - 三次握手，Three-Way handshake

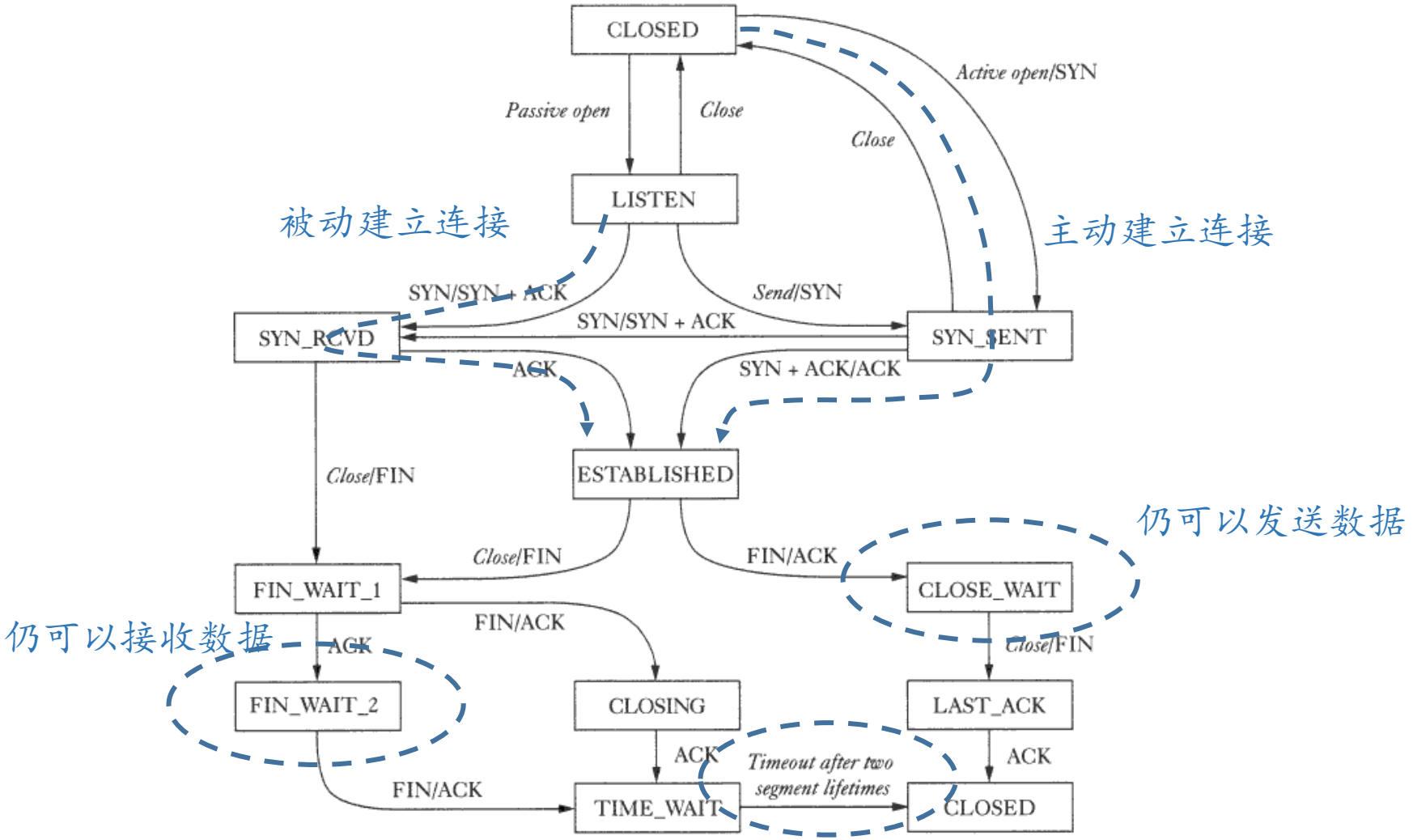


断开连接

- 连接任何一方都可以主动关闭连接
  - 发送FIN数据包
  - 表示己方不再发送数据
- 另一端可以继续发送数据
  - 对方仍需要对接收数据进行确认
  - TCP是一个全双工传输协议
- 任何一方都可以发送RST包关闭连接
  - 一方发送后不应再有数据传输
  - 正常情况下避免使用RST



TCP状态迁移图



主动关闭的一方需要等待1-4分钟才能回到CLOSED状态

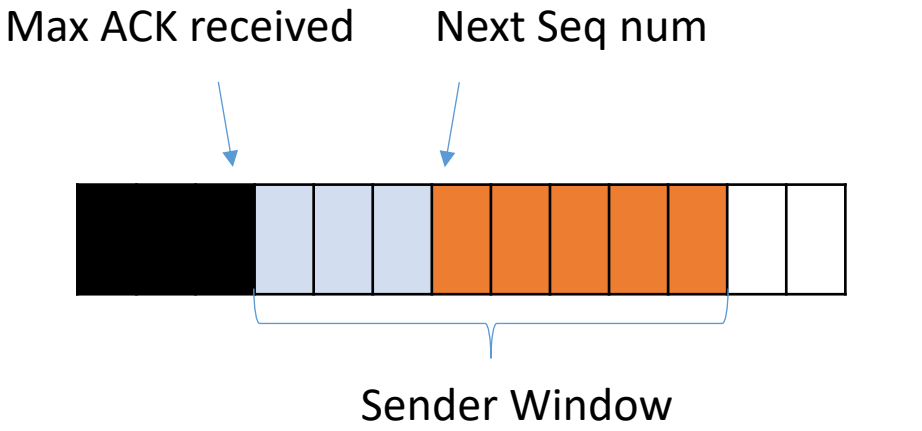
TCP数据传输

- 高效数据传输
  1. 丢包检测和重传
  2. 发送速率不能超过接收方接收能力
  3. 尽可能多利用网络带宽

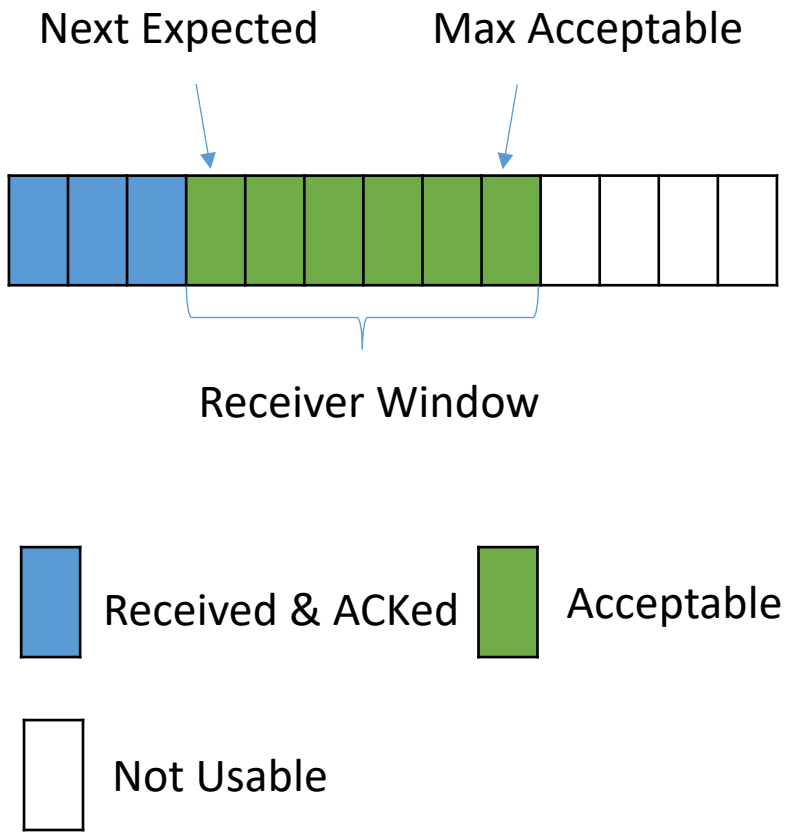
- 回忆数据链路层传输机制
  - 停等机制 (Stop-and-Wait)
  - 滑动窗口机制 (Sliding-Window)

滑动窗口（回顾）

发送方

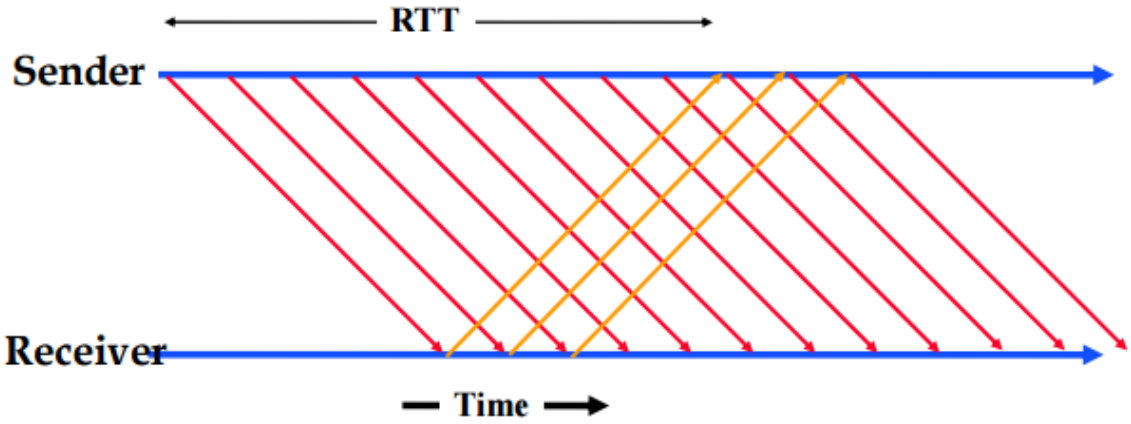


接收方



网络允许的最大传输窗口

- 带宽-延迟乘积 (Bandwidth-Delay Product, BDP)



- 同样的， Max Throughput = Max Window Size / RTT
- 具有较大BDP值的传输路径叫做长肥管道(Long-Fat Pipe)
  - 拥塞控制研究的目标之一就是提升TCP在长肥管道下的传输性能

流控 (Flow Control)

- 为了防止快发送方给慢接收方发数据造成接收崩溃
  - 注意与后面所讲的拥塞控制的区别
- 发送方和接收方各自维护一个窗口大小
  - 发送窗口<=接收窗口
- 发送方按照发送窗口大小发送数据
- 接收方根据接收窗口大小接收数据
  - 若数据落在窗口以外，直接丢弃
  - 若数据落在窗口以内
    - 收到的是连续数据
      - 将数据交给上层应用，更新窗口边界值
    - 收到不连续的数据
      - 放到buffer中，不更新窗口

流控实现问题

- TCP标准头部的Recv Window为16位，最大表示65535字节
  - 当RTT为100ms时，接收速率最大为640KB/s
- 引入Window Scale选项，只在建立连接时声明，后续数据传输都使用该值
  - Scaled Window = Recv Window * (2 ** Window Scale)
  - 最大值为 65535 * (1 << 14)

Kind = 3	Length = 3	shift.cnt

TCP Window Scale

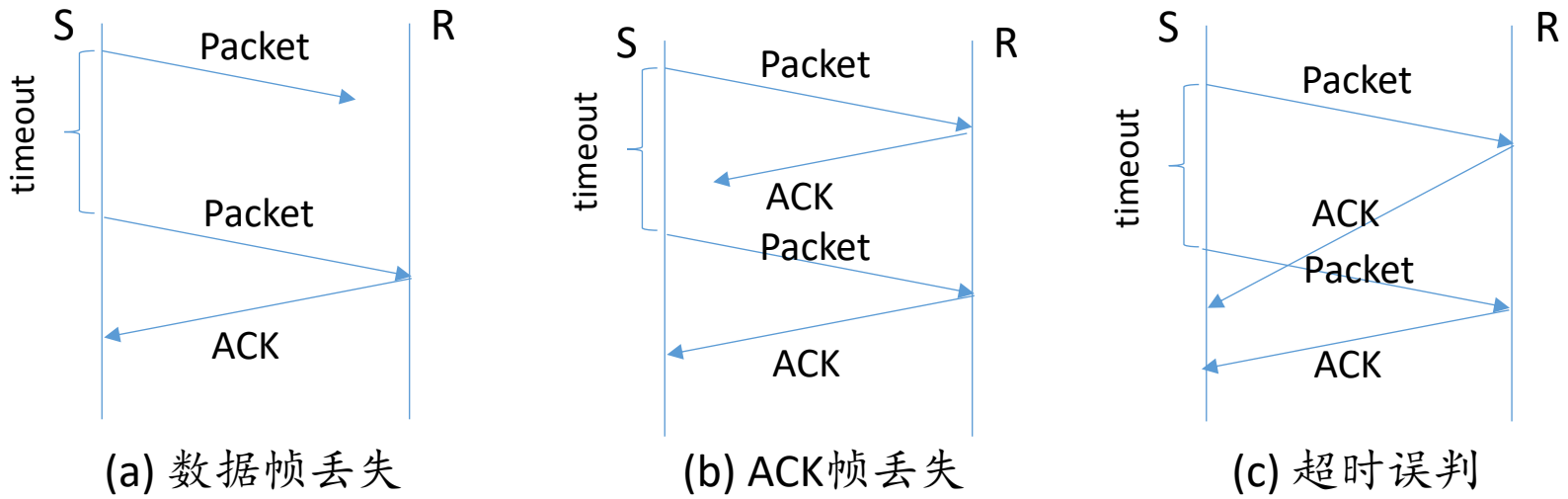
TCP SACK (Selective ACK)

- 接收方需要对每个收到的数据进行回复确认
  - 不考虑Delayed ACK情况
- 当收到连续的数据时
  - 回复ACK值为 (Packet Seq + Data Length)
    - 连续确认 (Cumulative ACK)，表示收到连续数据的上界
- 当收到不连续的 (out-of-order) 数据时
  - 除回复连续确认外，还需要附加非连续数据的边界
    - Cumulative ACK, (SACK_left, SACK_right), …

Kind = 5	Length = 8*n + 2	Left Edge of Block 1	Right Edge of Block 1	…	Left Edge of Block n	Right Edge of Block n

丢包检测和重传

- 通过超时来判断数据包丢失
  - 超时定时器设置为多少？
    - 较大值：恢复丢包效率低
    - 较小值：导致误重传 (Spurious Retransmission)



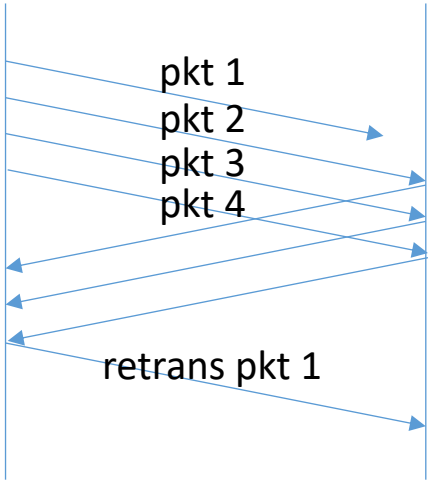
(a) 数据帧丢失　(b) ACK帧丢失　(c) 超时误判

超时重传定时器设置

- 定时器至少大于一个RTT (Round Trip Time)
- 定时器必须能够适应RTT变化
  - 当网络负载较高时，RTT变动较大
- 因此，定时器必须同时反映出RTT大小和RTT变化
  - Timer = sRTT + 4 * RTTVar
  - 对于每个RTT采样：
    - sRTT = 7/8 * sRTT + 1/8 * (RTT sample)
    - RTTVar = 3/4 * RTTVar + 1/4 * Dev
- Timer通常是百毫秒级的，在Linux实现中最小值是200ms

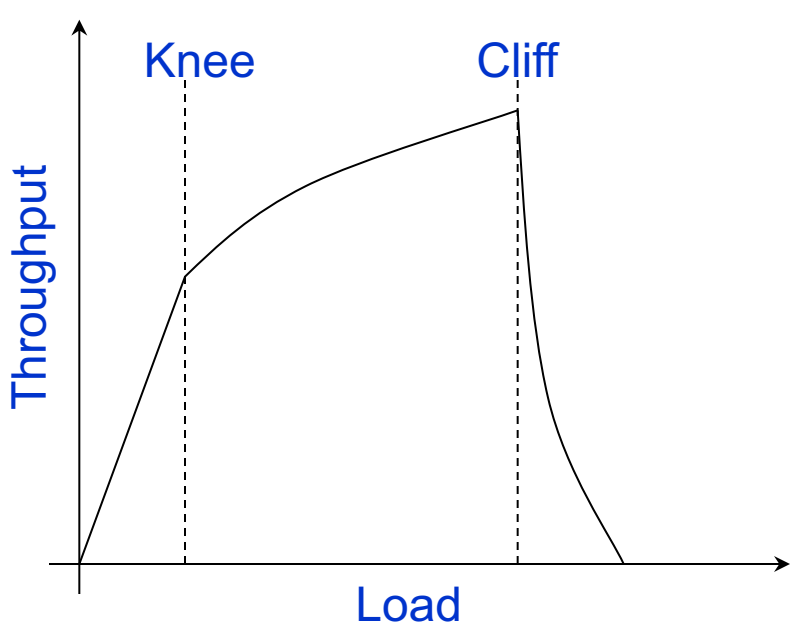
快速重传

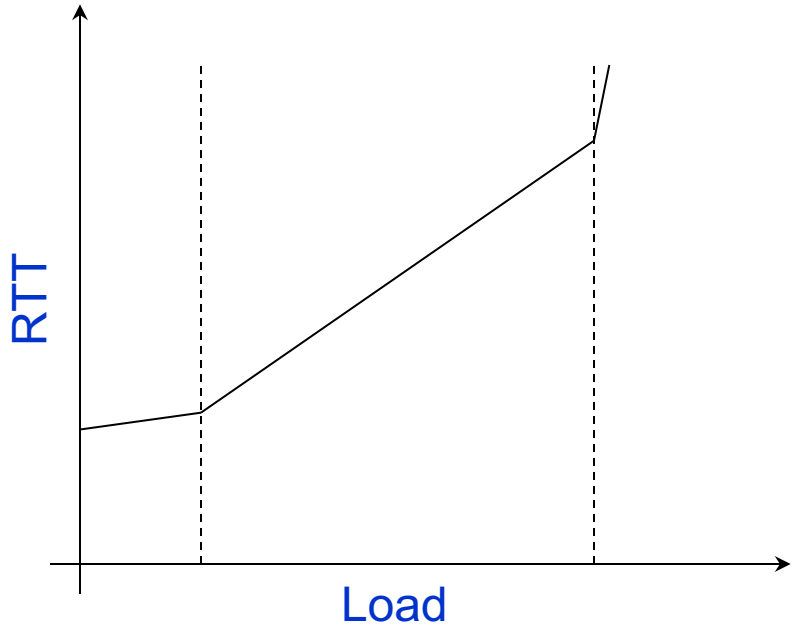
- Observation：一般情况下，先发送的数据包应该先到达
  - 如果后发送的数据包先被确认，可推测先发送的数据包丢失
- 快速重传
  - 如果一个数据包后面的三个数据包都被确认，
  - 而该数据包还未收到确认，
  - 则认定该数据包丢失，并重传该数据包
- 数据包通常都是连续发送的
  - 快速重传通常可以在1个RTT内重传数据丢包



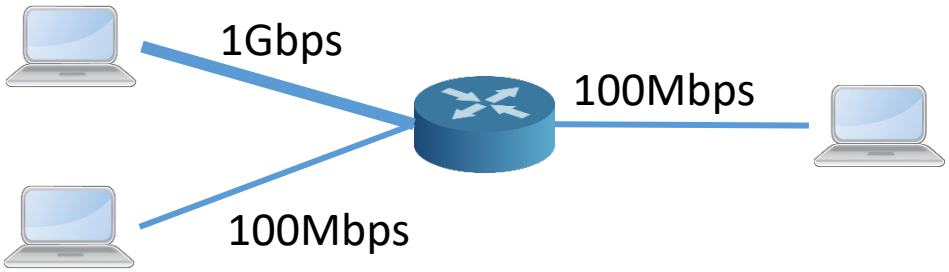
网络负载与网络性能

- 当在网络中存在过多的数据包时，网络的性能会下降
  - 当负载超过某阈值后，性能急剧下降





网络拥塞 (Congestion)



- 多个传输流需要共享(争用)网络内资源
- 当资源需求超过网络容量时，产生问题
  - 每条流不知道当前网络资源分配情况
  - 每条流也不知道其它(竞争)流的存在
- 后果
  - 丢包率升高、往返时间增大、甚至网络崩溃 (Network Collapse)
- 挑战
  - 如何协调网络内各条流，使其可以高效公平的利用网络带宽资源？

互联网级别的拥塞控制 (Congestion Control)

- 为一个很简单的网络设计拥塞控制策略是比较容易的
- 为互联网级别的网络设计拥塞控制策略，需满足：
  1. 高效利用网络资源
     - $\max \sum X_i$
  2. 节点间公平分享网络资源
     - $\max(\sum X_i)^2/(n\ \sum X_i^2)$
  3. 防止网络崩溃 (Congestion Collapse)
     - 防止拥塞崩溃是网络正常运行的前提

两种拥塞控制思路

- 端到端的拥塞控制
  - 不需要网络设备的拥塞提醒
    - 端设备通过丢包、延迟变化等推测网络拥塞状况
  - 优点：
    - 网络中间设备设计简单
  - 缺点：
    - 当拥塞推断策略较差时，网络资源利用率会很低

TCP使用端到端的拥塞控制策略

- 网络辅助的拥塞控制
  - 网络设备对端设备提供反馈
    - 通过标志位提醒拥塞 (ECN)
    - 显式的规定发送速率 (ATM)
  - 优点：
    - 资源利用率更高
  - 缺点：
    - 网络中间设备设计更复杂
    - 每条流维护一个状态，可扩展性差

TCP的拥塞控制算法基础

- 网络设备的功能非常简单
  - 共享buffer，进行先进先出 (FIFO) 调度
  - 通过丢包进行拥塞反馈 (binary feedback)

1. TCP端设备遇到丢包时，认为网络拥塞，减慢发送速率
   - 但是，丢包不一定是由网络拥塞引起的
2. TCP端设备定期通过增大发送速率来探测更多可用带宽

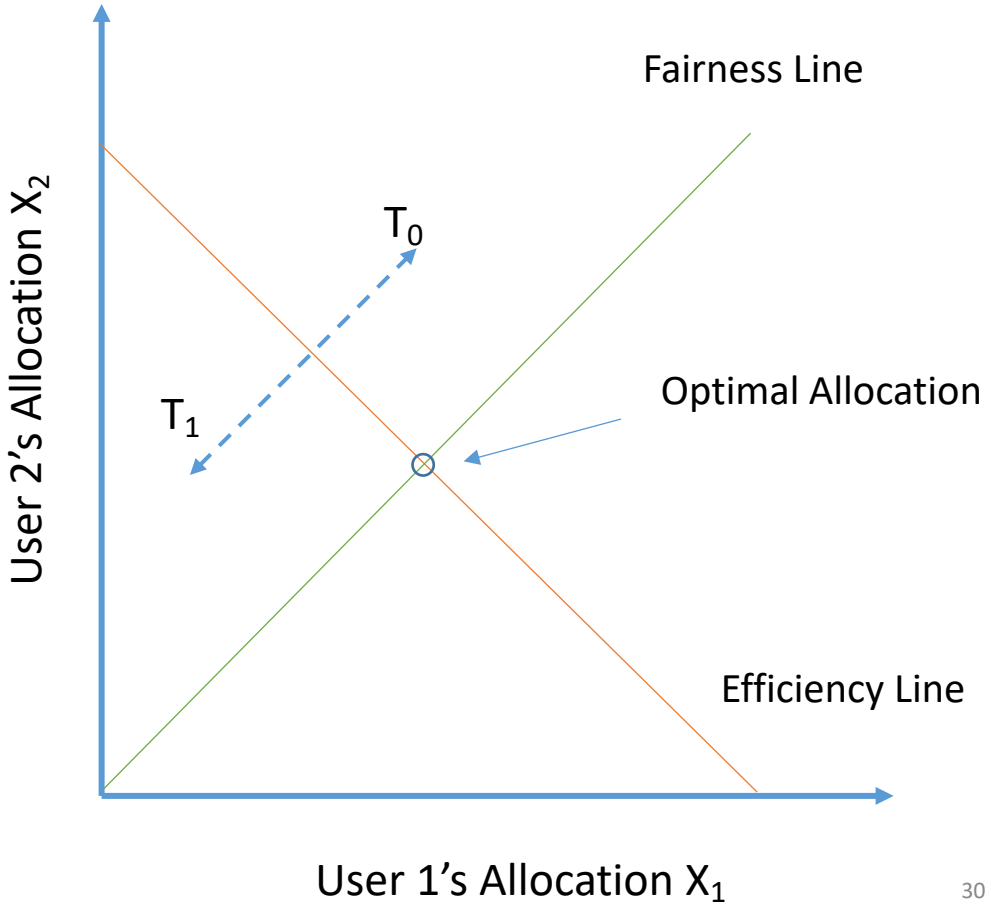
- 核心问题：
  - 增减控制策略：减慢，减慢多少？探测，增大多少？

增减控制策略

- 有很多不同的增减控制策略组合
  - 考虑最简单的线性控制策略 (Linear Control)
    - Window(t+1) = a * Window(t) + b
    - 增系数：a_i，b_i，减系数：a_d，b_{d}
- 可以支持很多不同的增减策略组合
  - 和性(Additively) 增加/减少 (AI/AD)
  - 乘性(Multiplicatively) 增加/减少 (MI/MD)
- 问题：在四种组合里，选出最优的，满足：
  - 收敛性、高资源利用率、公平性。。。

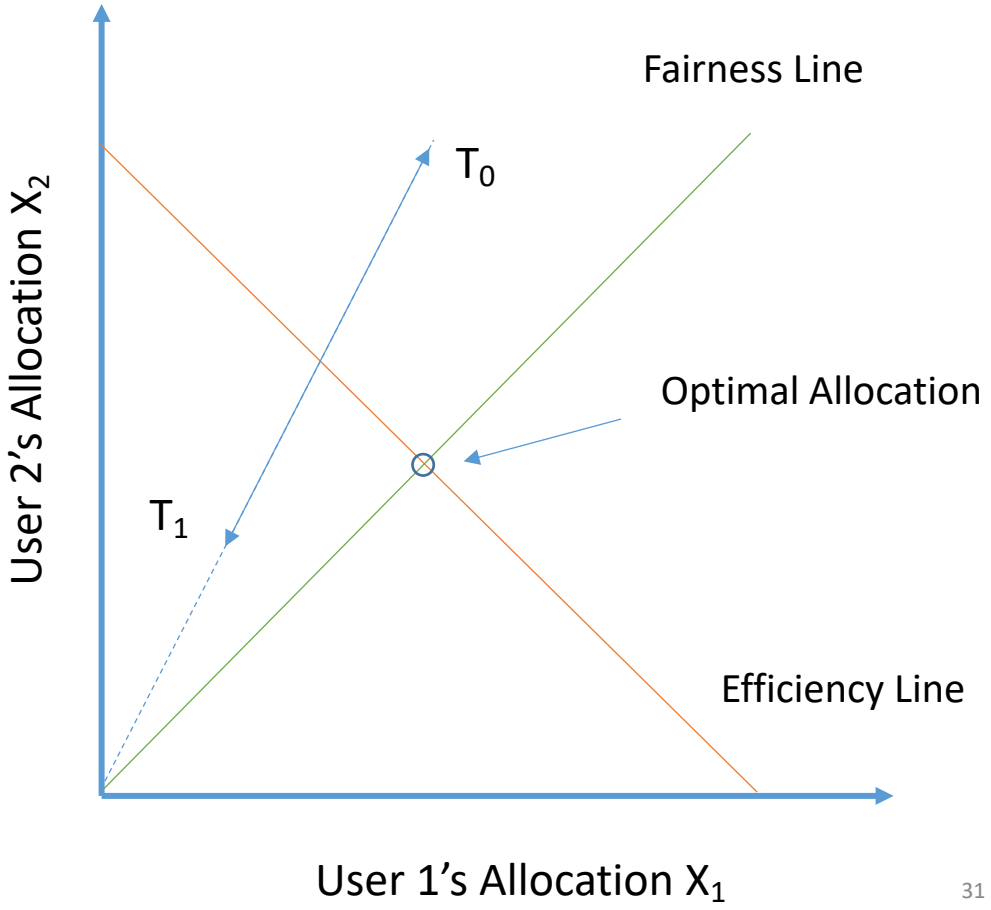
和性增，和性减 (AI + AD)

- X1和X2和性增加、减少相同的量
  - 和性增加改进公平性
  - 和性减少降低公平性



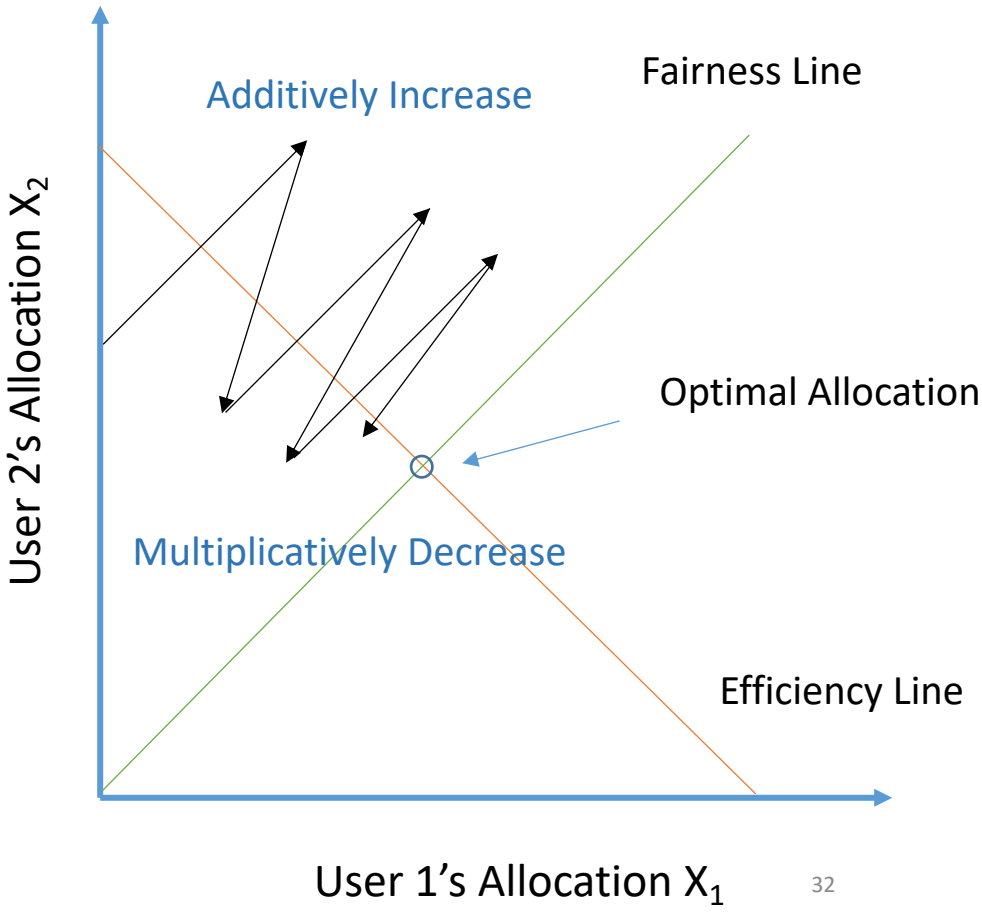
乘性增，乘性减 (MI + MD)

- X1和X2乘性增加、减少相同的因子
  - 公平性不会发生变化



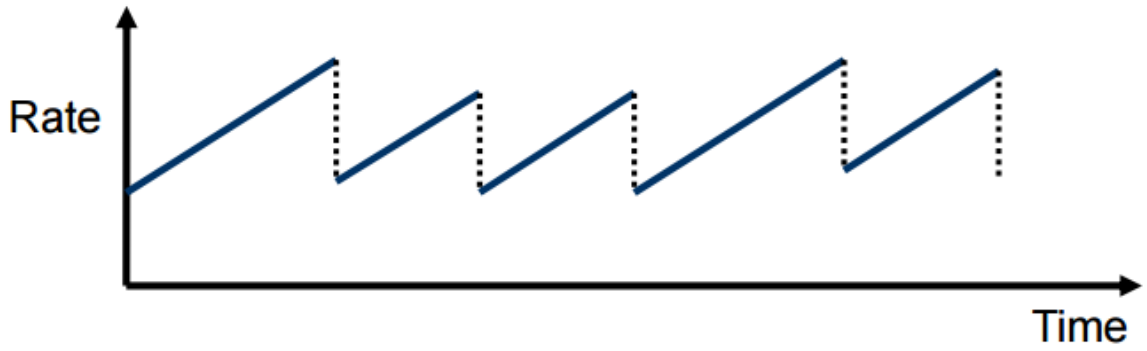
最优选择

- AIMD （和性增，乘性减）
  - AI: W(t+1) = W(t) + 1
  - MD: W(t+1) = 1/2 * W(t)
  - 向最优分配点收敛
- 也就是TCP中的拥塞控制



TCP中的拥塞控制

- 隐式的拥塞反馈 + AIMD
  - 端到端的设计 、分布式，兼具效率和公平性
- 每收到网络丢包信息
  - 发送速率（窗口大小）减半
- 周期性探测可用带宽
  - 每个RTT，窗口值增加一个数据包大小



快速重传后的发包停顿现象

- 当快速重传监测到丢包时，意味着网络拥塞
  - 窗口值减半 (cwnd -> cwnd/2, new cwnd)
- 发送方未确认的 (outstanding) 数据包数目 > new cwnd
  - 不能发送新数据
  - 每收到一个ACK (SACK)，outstanding数据包数目减一
- 当outstanding数据包数目 < new cwnd时，
  - 可以发送新数据 (new cwnd – outstanding)
- 表现
  - 经过快速重传后，发送方在一段时间内不能发送任何数据 (< 1 RTT)

快速重传后又发送丢包怎么办？

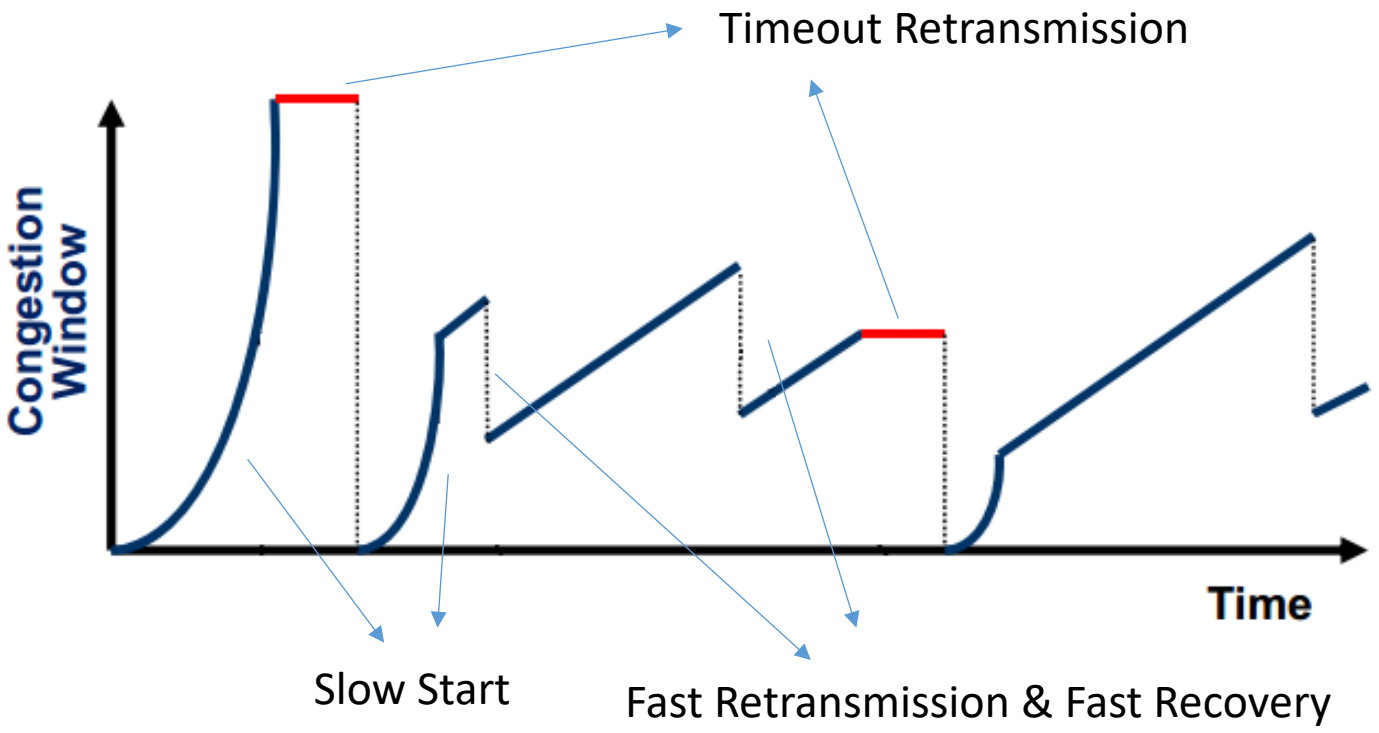
- 快速恢复(Fast Recovery)阶段
  - 起始于快速重传后，结束于RecoveryPoint数据被连续确认
    - RecoveryPoint为进入快速恢复阶段时发送的最大序列号
- 在快速恢复阶段，如果遇到其他丢包
  - 仍然使用快速重传，窗口减半？ 否
  1. new cwnd不变，且保证outstanding <= new cwnd
  2. 如果outstanding < new cwnd，发送数据包
     - 如果检测到有丢包（利用快速重传的检测技术）
       - 重传相应数据包
     - 否则，发送新的数据包
  3. 直到Cumulative ACK >= RecoveryPoint

慢启动 (Slow Start)

- 拥塞避免 (AIMD) + 快速重传 + 快速恢复
  - 可以达到稳态，且在稳态下工作良好
- 如何快速达到稳态？
  - 可工作在54kbps ADSL中，也可工作在10Gbps网络
- 慢启动机制
  - 初始窗口 initial cwnd，初始值设置为3或10
  - 慢启动门限值 ssthresh，初始值设置为1 << 31
  - 每收到ACK，窗口值加1
- 慢启动并不慢
  - 在没有丢包情况下，经过log2(target_cwnd/initial_cwnd)个RTT长到目标窗口大小

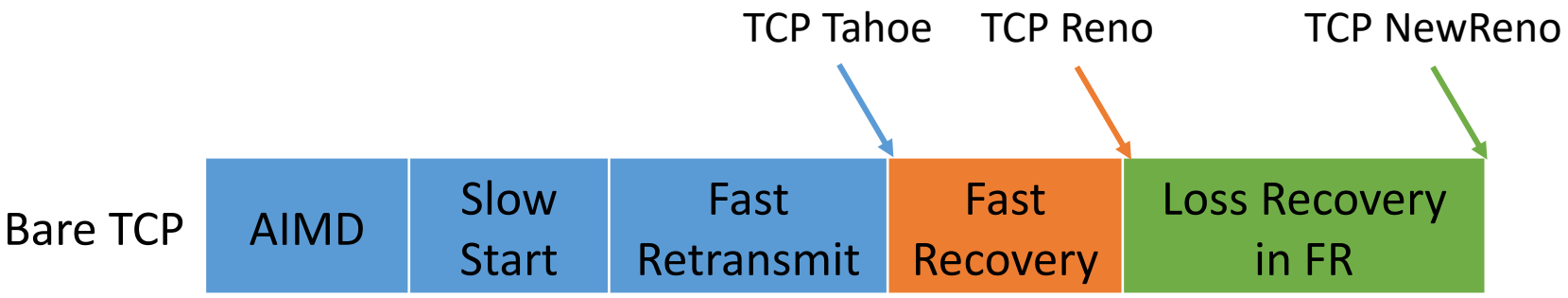
```
Initialization:
  cwnd <- initial cwnd

if cwnd < ssthresh:
  for each ack:
    cwnd += 1
else:
  for each ack:
    cwnd += 1/cwnd

when encountering loss:
  ssthresh <- cwnd
  cwnd <- cwnd/2
```

TCP锯齿状窗口行为


Timeout Retransmission
Congestion Window
Time
Slow Start
Fast Retransmission & Fast Recovery

TCP小结

- 设计目标
  - 效率、公平性、收敛速度
- 系统设计
  - 端到端原则，隐式拥塞信号，分布式，可扩展性

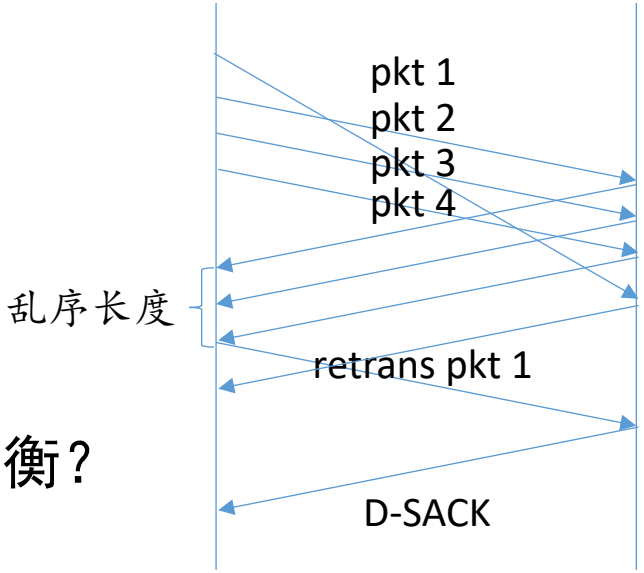


TCP优化

- TCP丢包重传优化
  - 提升重传效率，减少超时重传的比例，改进短流完成时间
  - Adaptable DupThresh, TLP, RACK
- TCP拥塞控制算法优化
  - 改进拥塞控制算法，提升特定网络环境下的带宽占用率（性能）
  - Cubic、Compound TCP、BBR

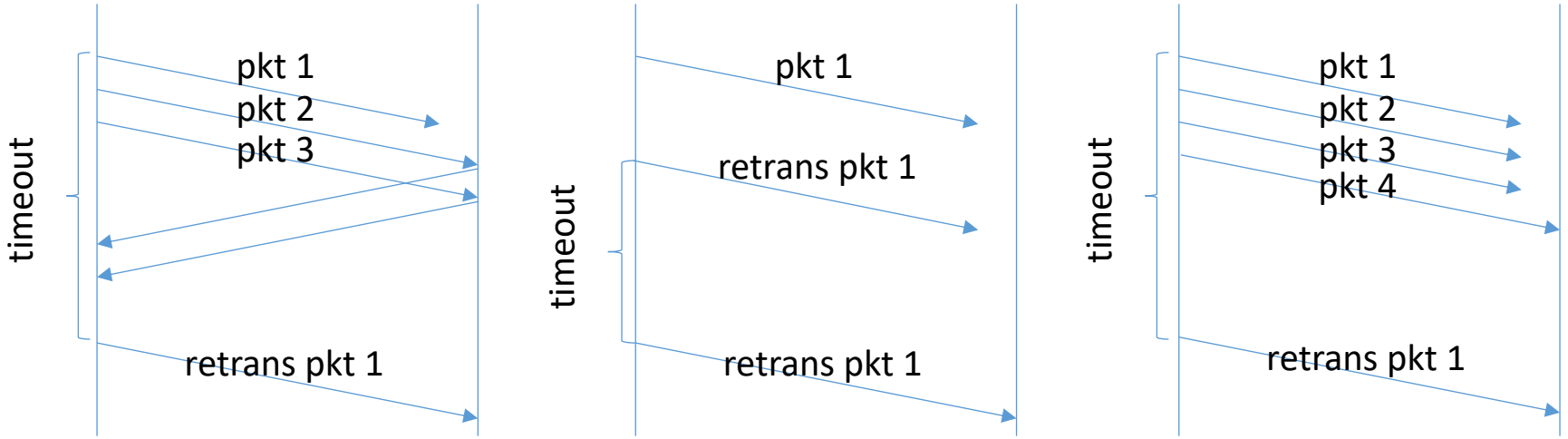
快速重传与数据乱序

- 为什么需要等3个后续数据包的确认？
  - 防止因数据包乱序引起的误重传
    - 在无线网络、多路径传输中会有部分乱序
  - 假设网络中乱序长度不大于3个数据包
- 如何在快速重传和减少误重传之间取得均衡？
  - 如何识别一个数据包没有丢失，而是乱序？
    - 通过D-SACK机制（Duplicate SACK）
  - 将快速重传门限值设置为“乱序长度+1”个数据包
    - 每次遇到D-SACK时增加重传门限值，每次超时后恢复为3



快速重传可以恢复所有丢包么？

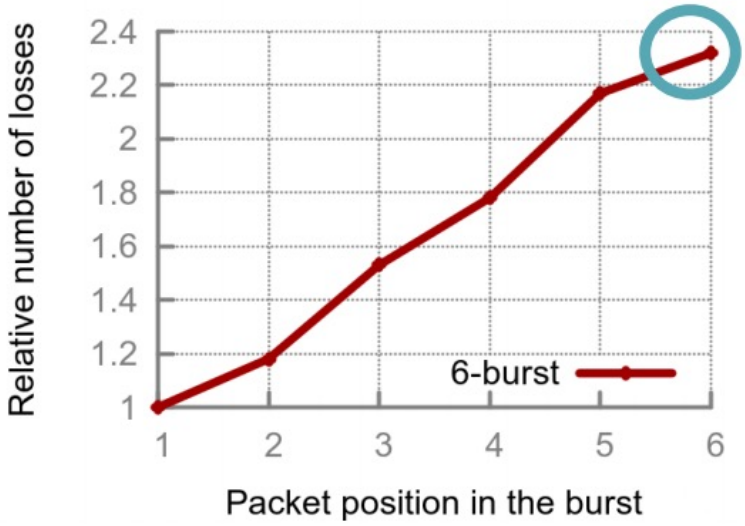
- 一般来说，快速重传可以恢复长流中的大部分丢包
  - 对于短流来说，70%以上的丢包由超时重传恢复[Google 2013]
    - 大部分Web应用都是短流



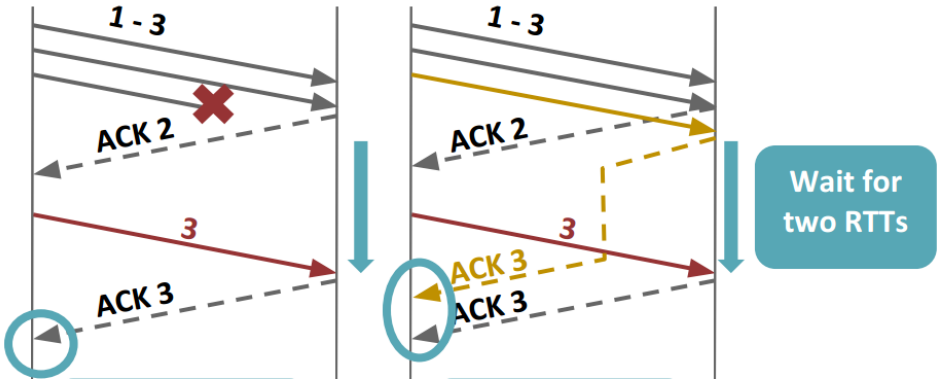
(a) 发送数据较少时　(b) 重传数据丢失时　(c) 大量丢包时

TLP (Tail Loss Probe)

- 数据包通常是以burst形式发送出去的
  - 相邻数据包的时间间隔非常短
  - 越后面的数据越有可能被丢弃



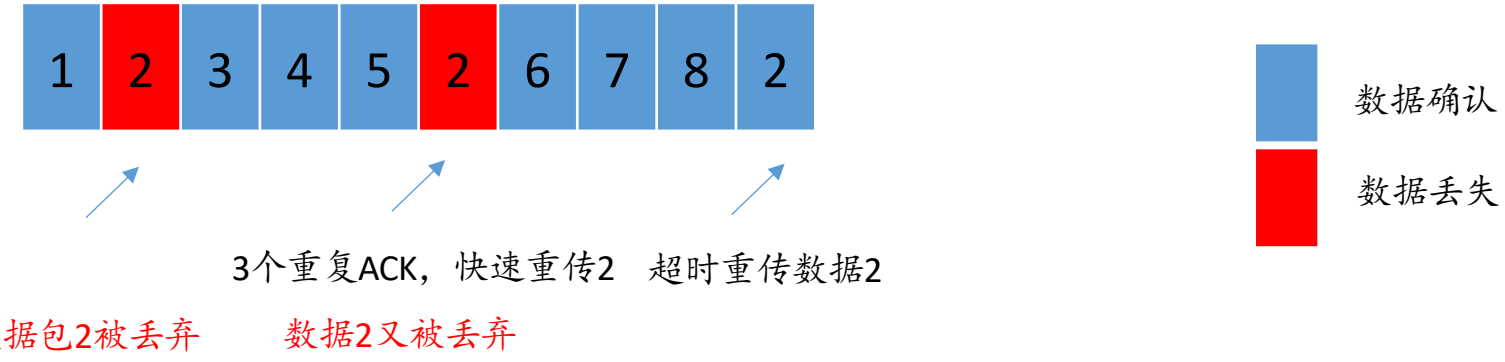
- 当发送数据较少时，可以降低触发重传的门限值
  - 当满足特定条件时，显式的去重传该数据包
  - 以稍激进的重传策略，换取快速丢包恢复和减少误重传间的平衡



重传后的丢包

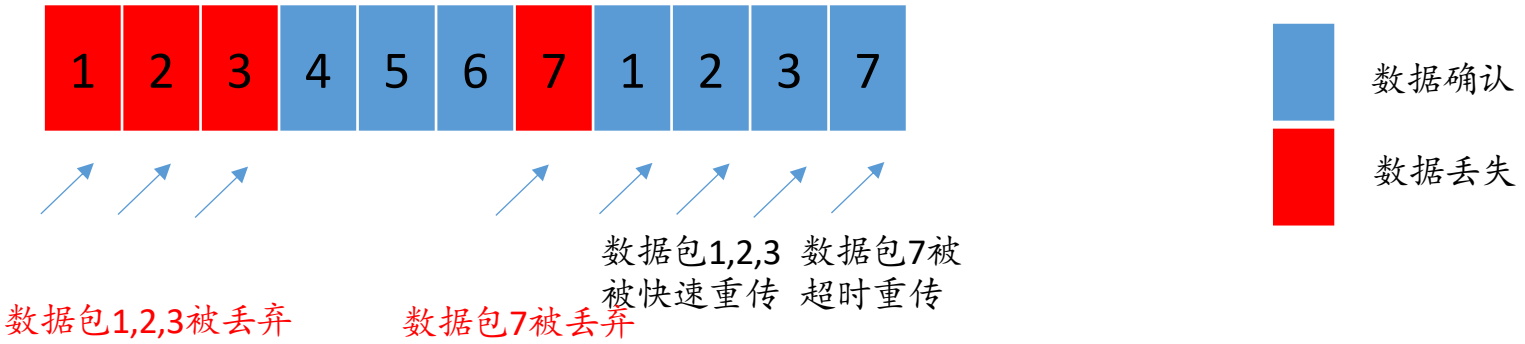
- 当网络中没有发生重传时
  - 发送顺序和序列号是一致的：序列号越大，发送顺序越靠后
    - 快速重传即是根据这一依据设计的
- 当网络中已发生重传后
  - 发送顺序和序列号变得不再一致
    - 二次重传问题
      - 如果重传数据包丢失，其不再能通过快速重传来恢复
    - 连续重传问题
      - 如果数据被重传，其不再被当做后续数据包来触发快速重传

二次重传问题



- 同一数据包被丢弃两次后，只能等待超时重传
  - 对于重传的数据，不能再依赖重复ACK来判断其是否再次被丢弃

连续重传问题

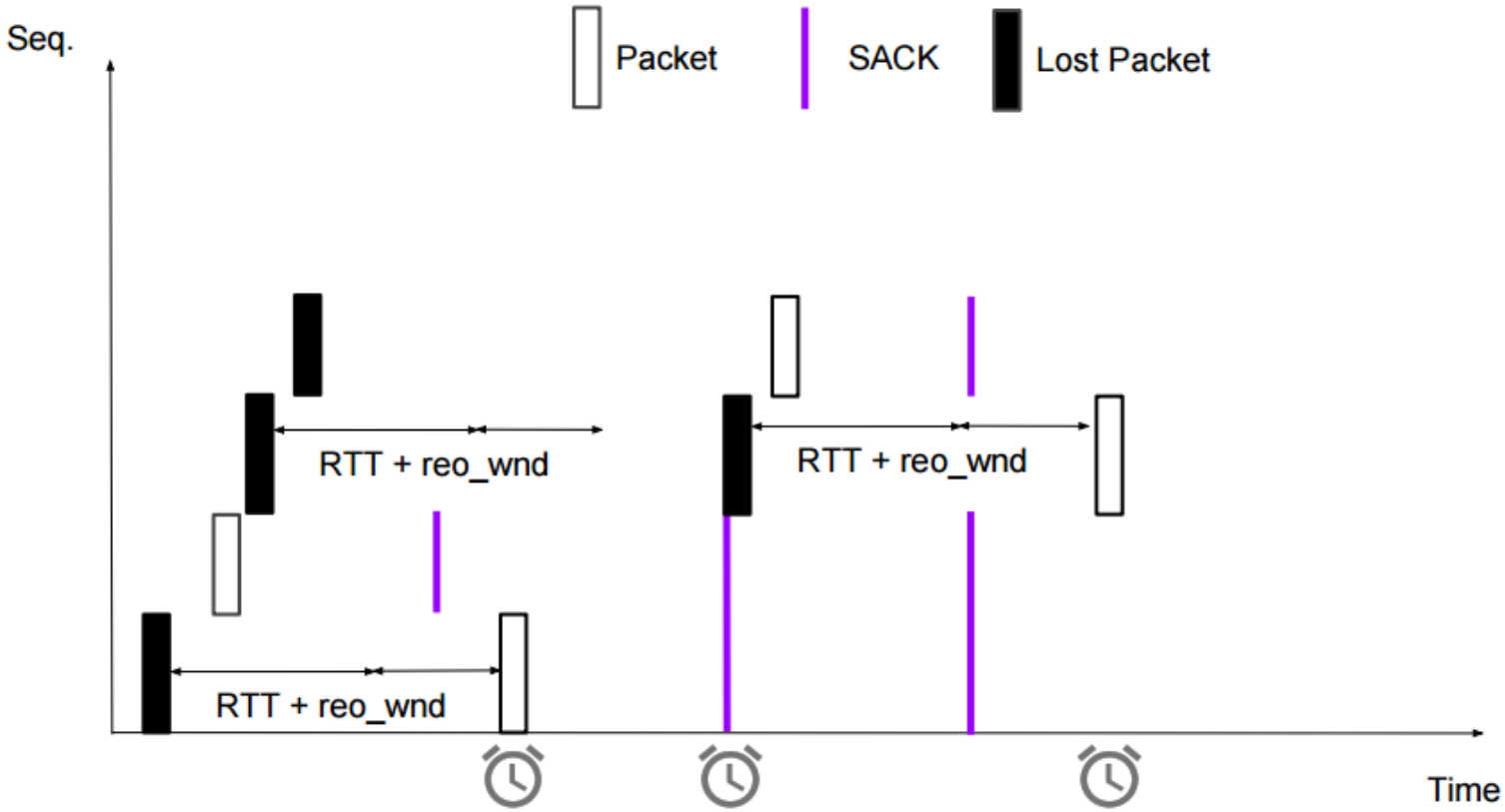


- 重传数据包的ACK不能用做后续数据包来触发快速重传
  - 其序列号可能小于其他数据包（发送顺序与序列号不再一致）

RACK

- Observation
  - 在重传后，序列号不再能表示发送先后顺序
  - 但每个数据包（包括重传数据包）的发送时间可以记录下来
- 核心思想
  - 给定先后发送的两个数据包P1和P2
    - P2先收到确认
    - P1是在RTT+delta之前发送的，且还没有收到确认
    - 则认定P1被丢弃

RACK示例


Seq.
Packet
SACK
Lost Packet
RTT + reo_wnd
RTT + reo_wnd
RTT + reo_wnd
Time

拥塞控制算法优化

- TCP NewReno吞吐率经验公式

$$\text{Throughput} = \frac{C * MSS}{RTT * \sqrt{loss\ \text{rate}}}$$

- NewReno拥塞控制算法存在的问题：
  1. RTT不公平
     - 两条流，其他各条件都相同，RTT大的带宽占用率低
  2. 大带宽、高延迟网络中性能较差
     - 即使很小的丢包率也会导致极低的带宽占用率
     - 1000Mbps带宽、100ms延迟、0.01%丢包：带宽占用率只有3%

TCP Cubic

- 核心思想
  - 窗口增长与RTT无关，只取决于距离上次快速重传的时间
  - 利用三次函数特性，高效的窗口恢复和窗口探测

$$\text{cwnd} = C(t-k)^3 + W_{max}\ ,$$

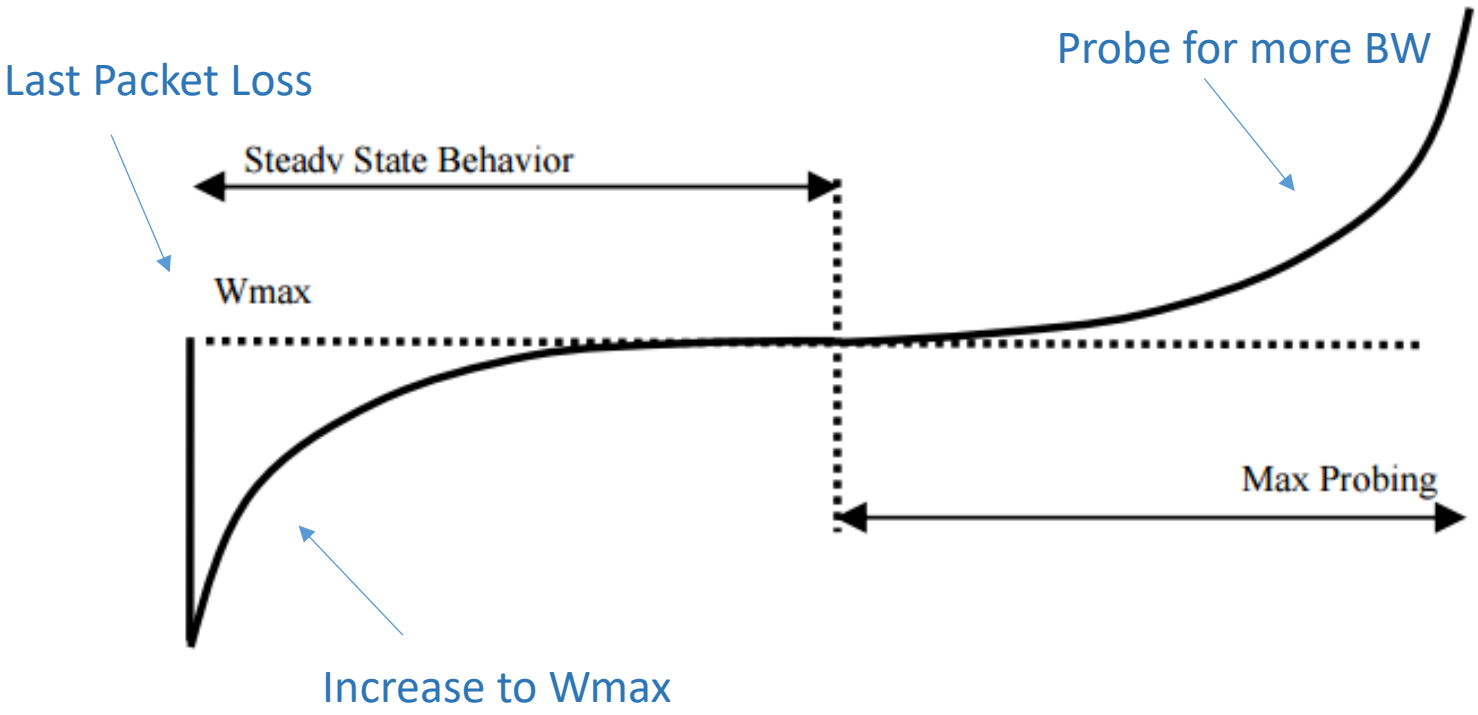
W_{max}: cwnd before last reduction

$k = \sqrt[3]{W_{max}\beta/C}$, where β is a factor

C: scaling factor

t: time elapsed since last reduction

TCP Cubic窗口行为



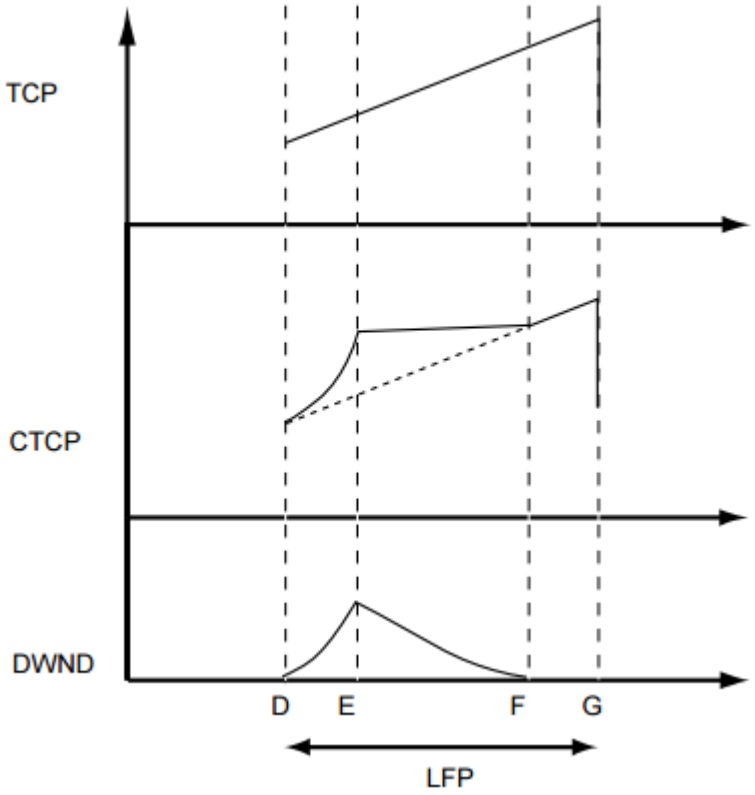
Compound TCP

- 核心思想
  - 丢包不是网络拥塞的唯一信号，还有延迟变化
  - 将基于丢包计算的窗口值与基于延迟计算的窗口值之和作为窗口值

$$cwnd(compound) = cwnd(reno) + dwnd, \text{where}$$

$$dwnd(t+1) = \begin{cases} dwnd(t) + (\alpha \cdot win(t)^k - 1)^+, & \text{if } diff < \gamma \\ \left(dwnd(t) - \zeta \cdot diff\right)^+, & \text{if } diff \geq \gamma \\ \left(win(t) \cdot (1-\beta) - cwnd/2\right)^+, & \text{if loss is detected} \end{cases}$$

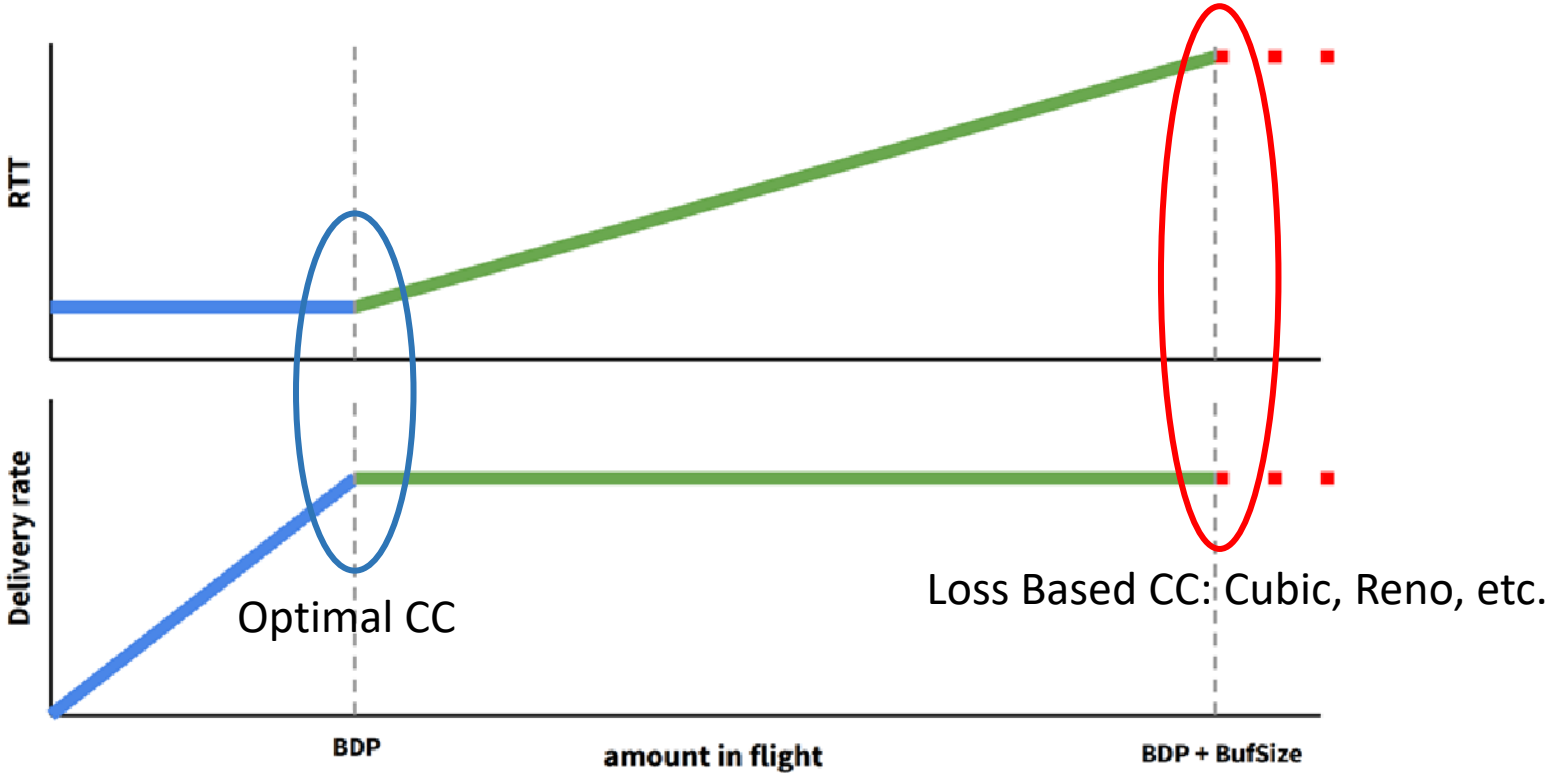
Compound TCP窗口行为



- D -> E
  - 快速进入稳定状态
- E -> F
  - 稳定状态
- F -> G
  - 窗口探测阶段

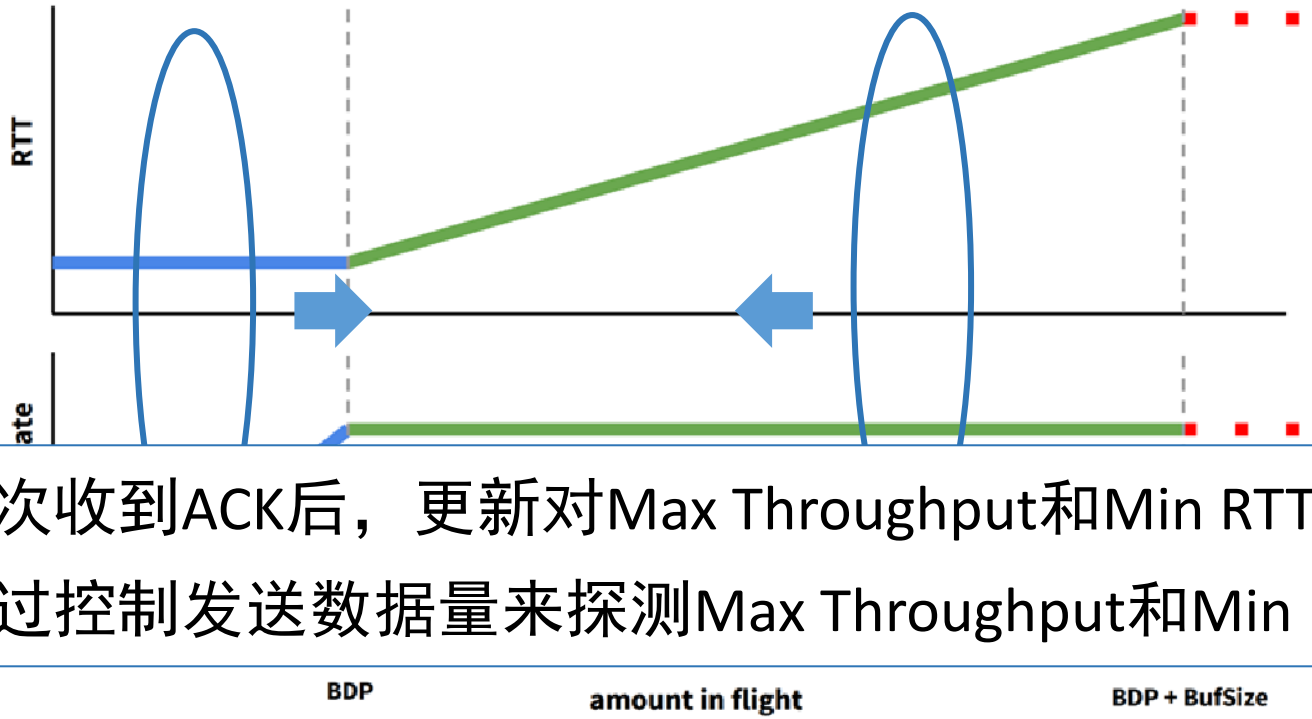
BBR (Bottleneck Bandwidth and RTT)

- Throughput = max Window size / RTT



BBR设计

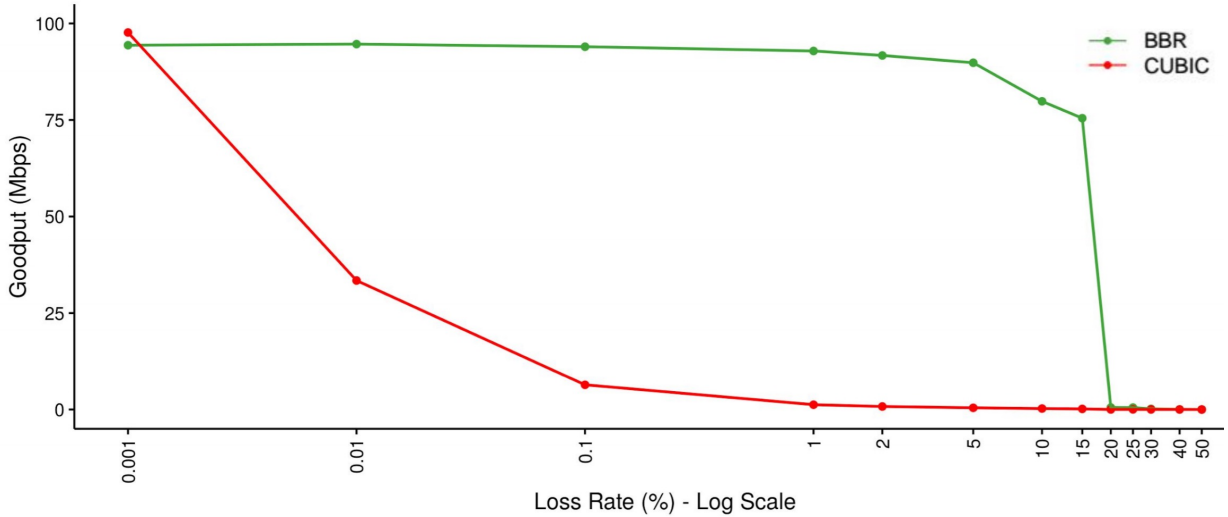
- 设计目标：在不降低吞吐率的同时，减少网络延迟
  - Maximize Throughput, Minimize RTT



1. 每次收到ACK后，更新对Max Throughput和Min RTT的估计
2. 通过控制发送数据量来探测Max Throughput和Min RTT

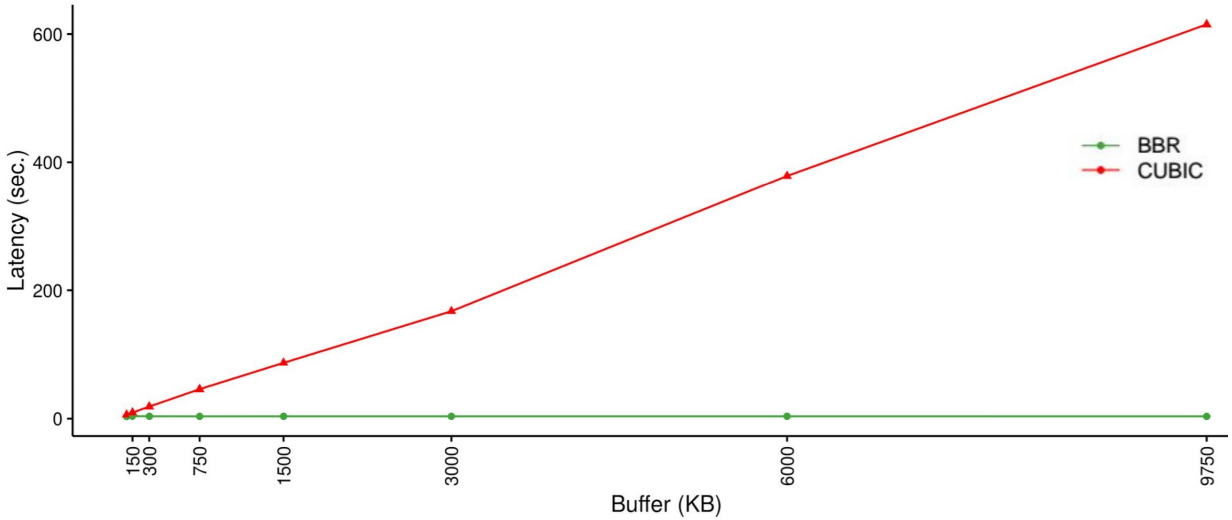
BBR传输性能

吞吐率



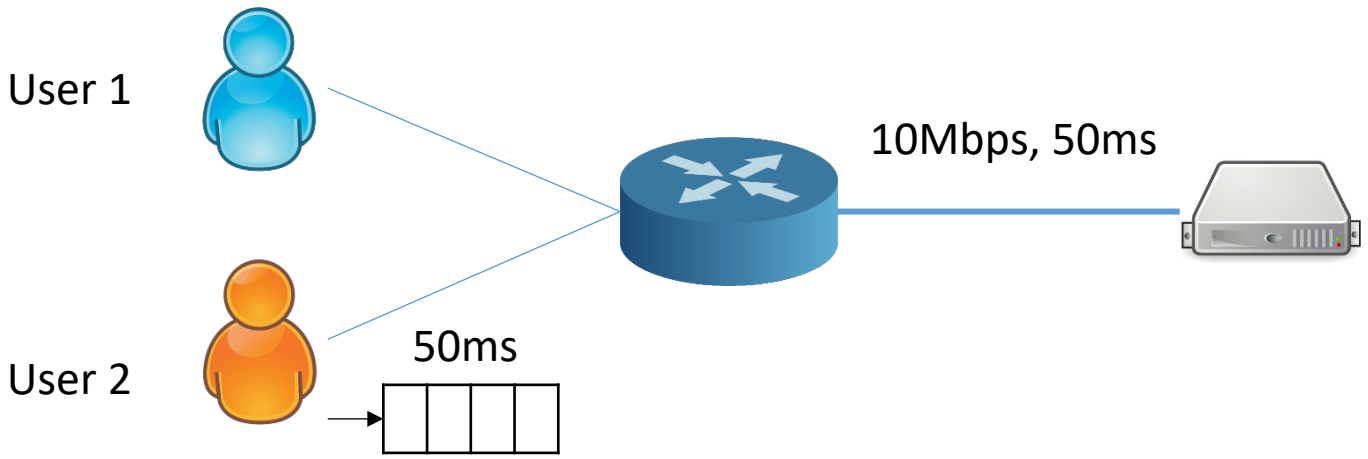
BBR vs CUBIC: synthetic bulk TCP test with 1 flow, bottleneck_bw 100Mbps, RTT 100ms

网络延迟



BBR vs CUBIC: synthetic bulk TCP test with 8 flows, bottleneck_bw=128kbps, RTT=40ms

BBR存在的问题



初始场景：

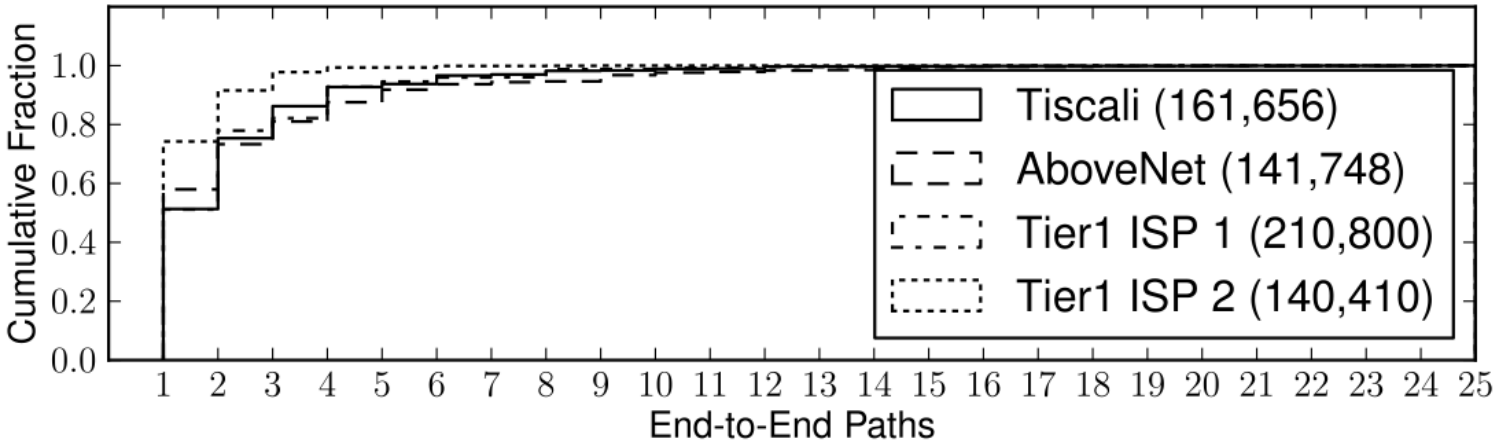
- User 1 Throughput: 5Mbps
- User 2 Throughput: 5Mbps

用户2修改配置后：

- User 1 Throughput: 1Mbps
- User 2 Throughput: 9Mbps

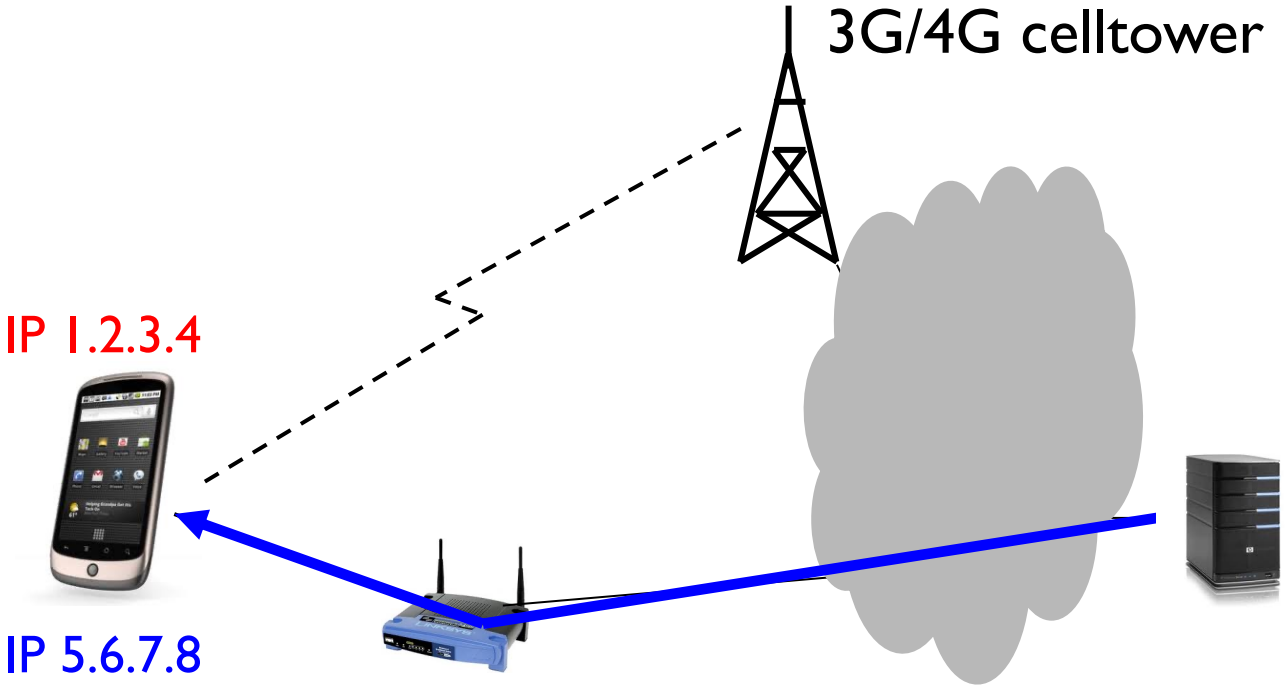
网络的多路径特征

运营商网络的多路径



- 数据中心的可用多路径更多！
- 传统TCP连接只能利用单条路径
  - TCP本身不能感知多条路径，多路径分发导致的乱序会降低TCP性能

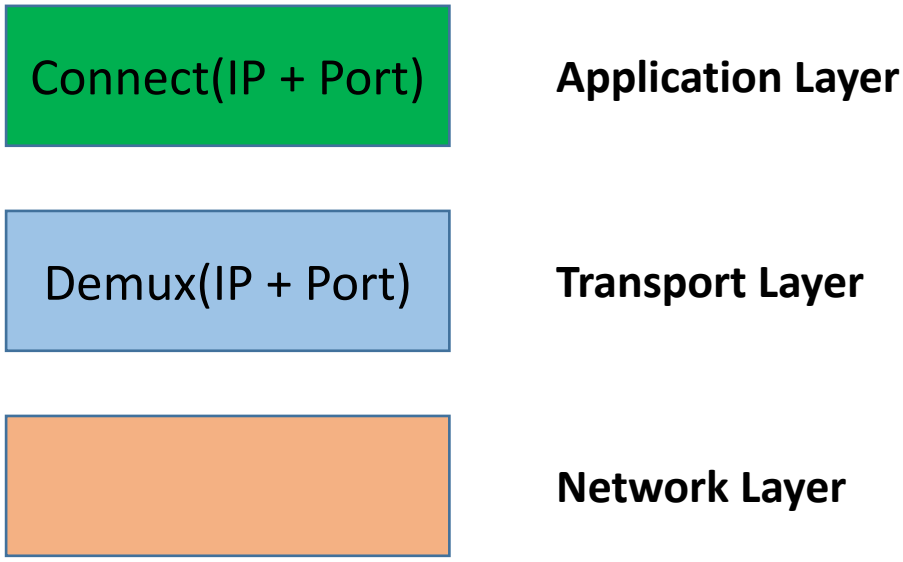
端节点的多接入和移动性



- TCP连接与IP地址绑定，一旦因为移动发生地址切换，原有的TCP连接会断开

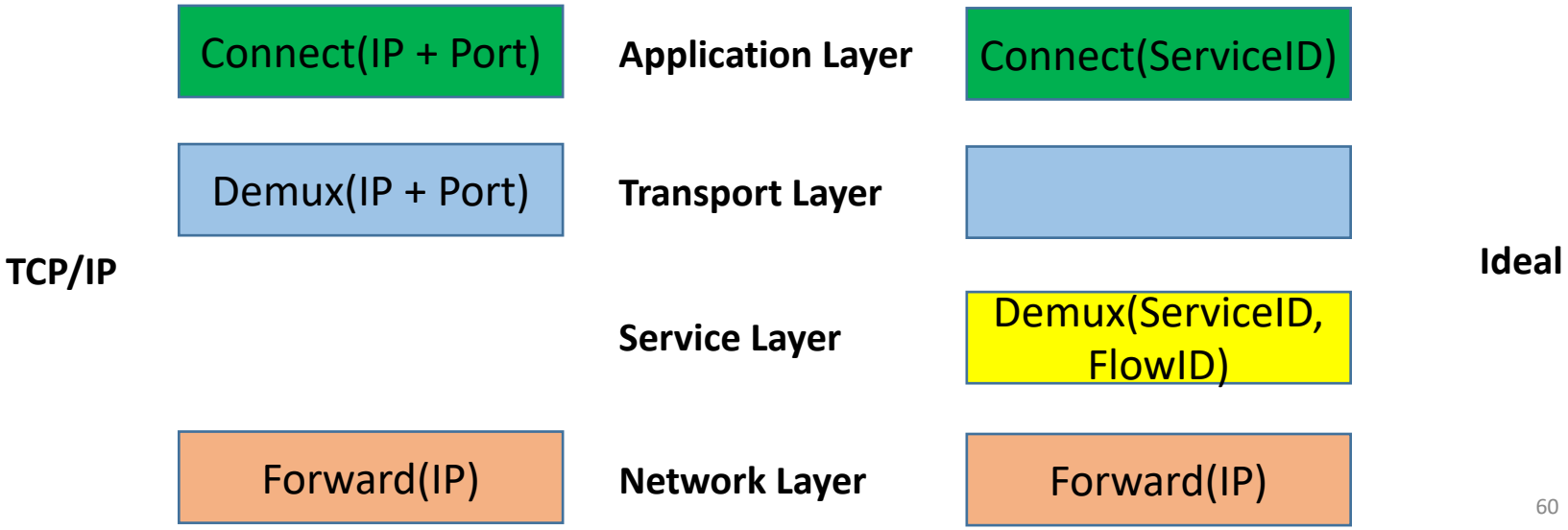
网络、连接与应用的抽象

- 对于应用程序（Socket）来说，
  - 连接（或者流）绑定到了具体的一个IP地址和具体的一个网络接口
  - 因为绑定到了IP地址，一个Socket不能使用多个路径/接口
  - 因为绑定到了IP地址，当地址变化时，连接只能断开



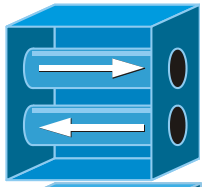
解耦合、增加中间层

- ServiceID：表示用户最终想获取的服务或进程
- FlowID：数据传输的载体，不随地址改变而改变
- Location：主机所在的位置，可变

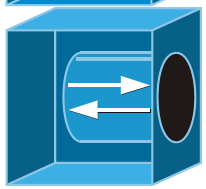


Middleboxes

Web Security Appliance

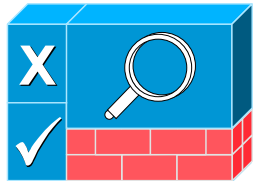


VPN Concentrator

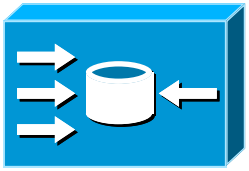


SSL Terminator

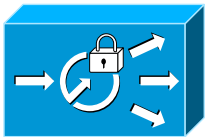
NAT



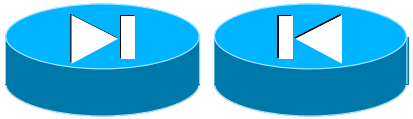
NAC Appliance



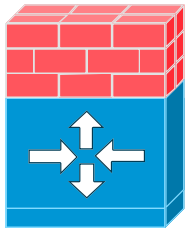
Content Engine



ACE XML Gateway

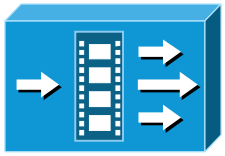


PIX Firewall Right and Left



Cisco IOS Firewall

IP Telephony Router



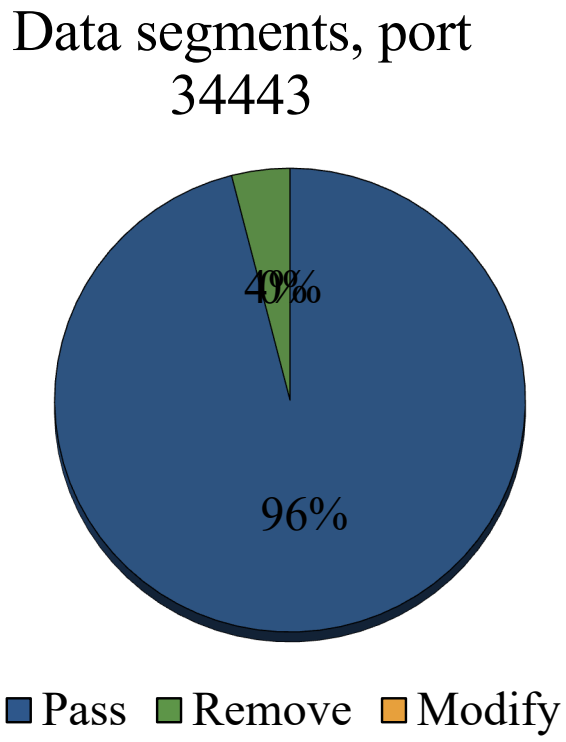
Streamer

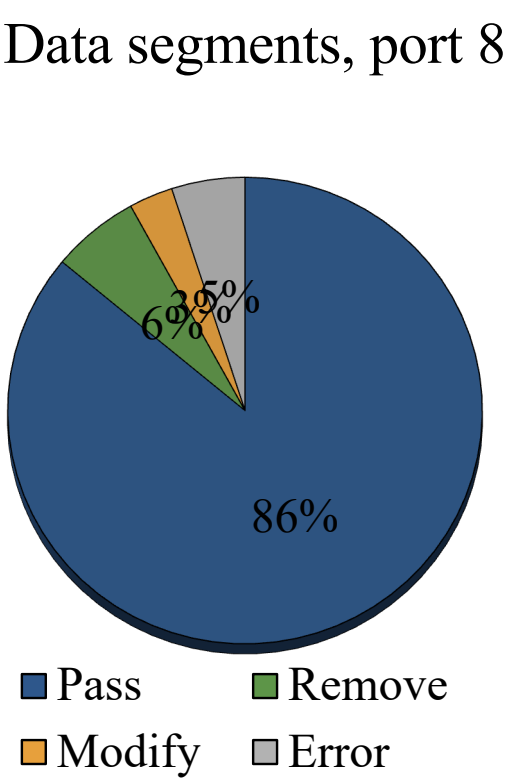
Voice Gateway

- 据统计，企业网中的Middlebox设备的数目与交换机/路由器一样多

Middlebox与TCP选项

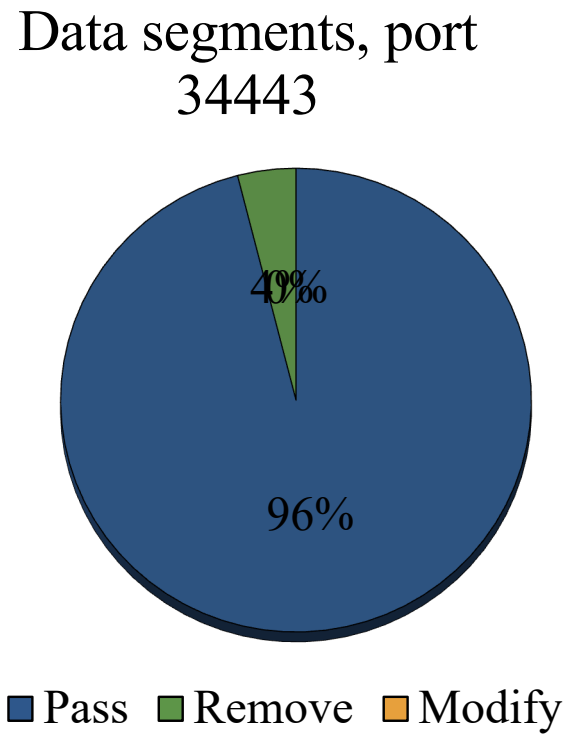
- TCP时间戳选项(Time Stamp)

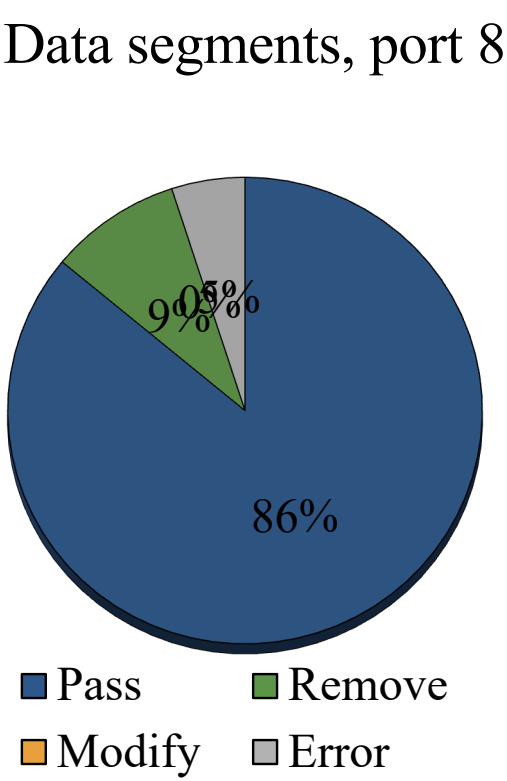




Middlebox与TCP选项

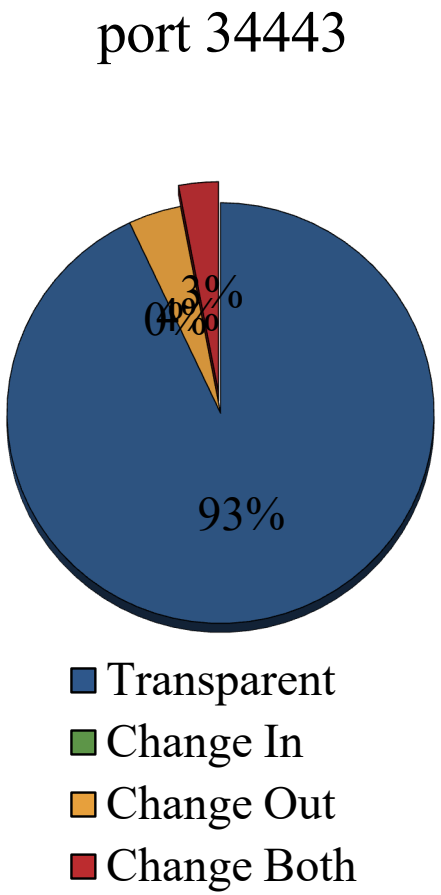
- TCP未定义选项

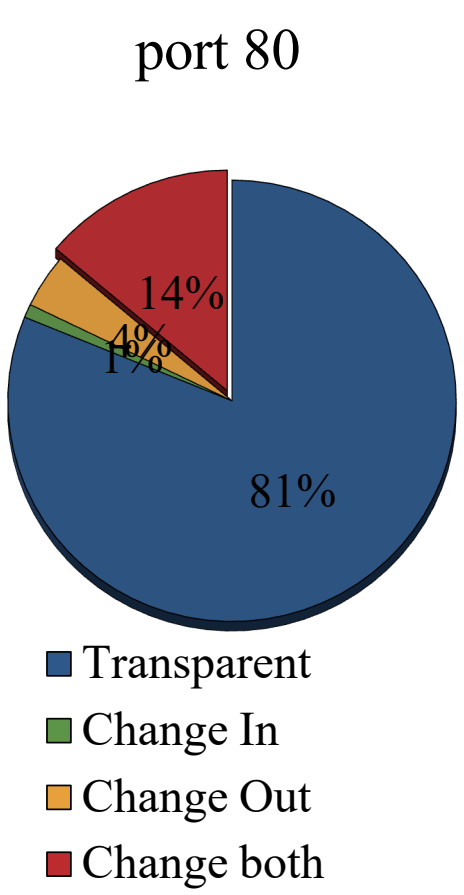




Middlebox与TCP序列号

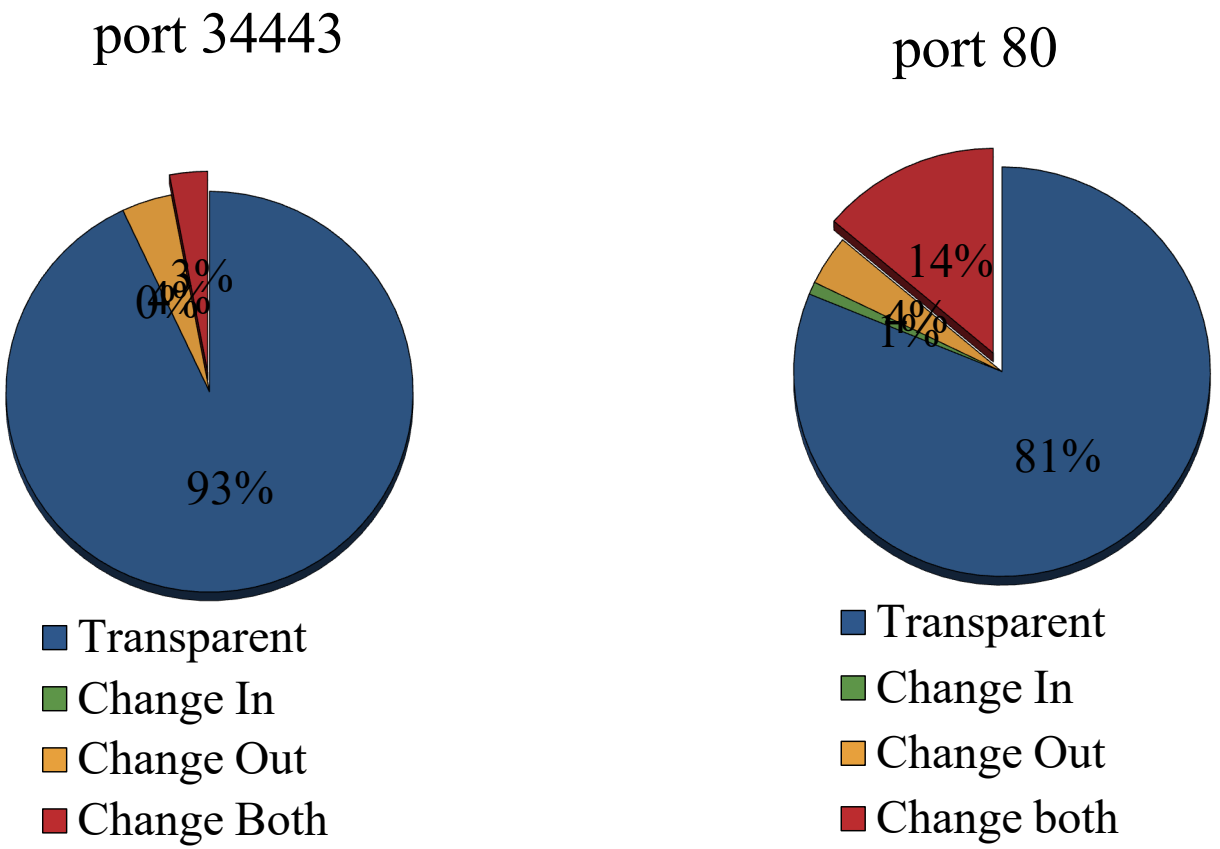
- TCP序列号



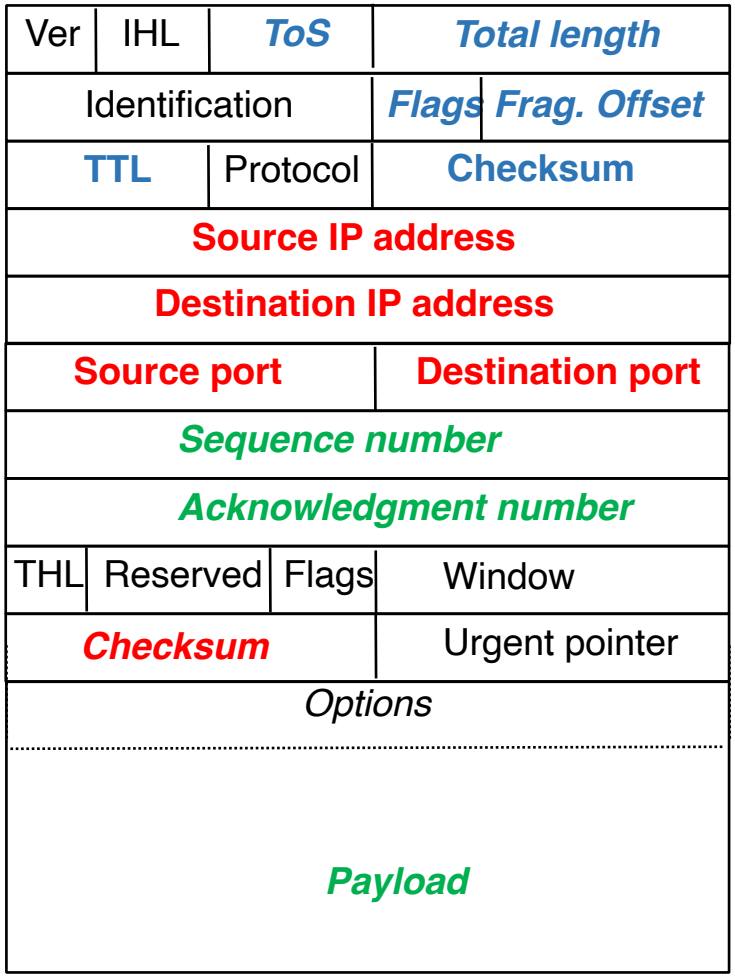


Middlebox与数据段合并

- TCP数据段合并



为什么Middlebox会做这些事情？



路由器会修改

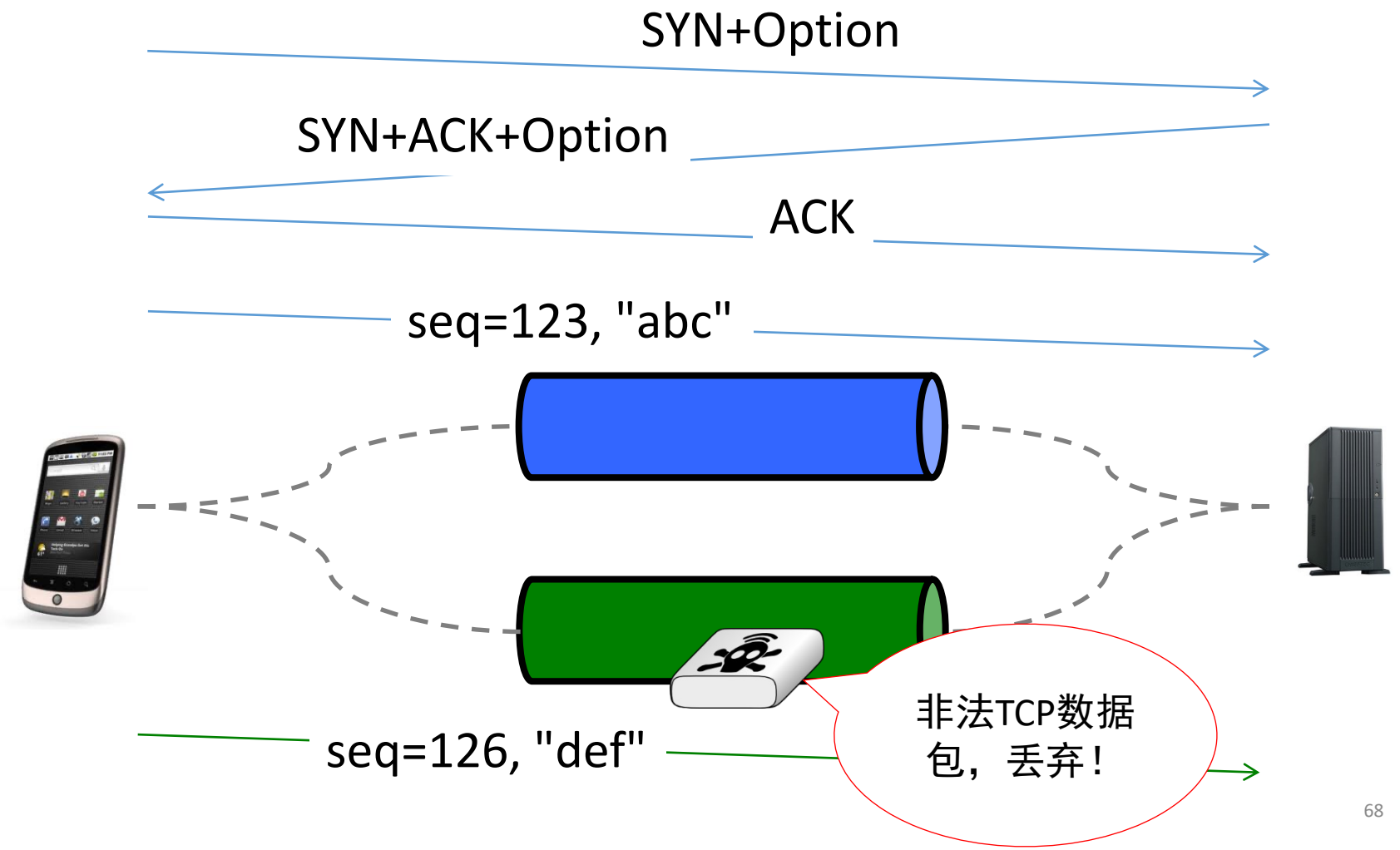
NAT设备会修改

应用层网关会修改

MPTCP机制

- 控制平面 (Control Plane)
  - 如何在多条可用路径之间进行连接管理？
  - 传统TCP: 三次握手+四次挥手
- 数据平面 (Data Plane)
  - 如何进行数据传输？
  - 传统TCP: Seq + Ack
- 拥塞控制 (Congestion Control)
  - 多条传输路径之间如何进行拥塞控制？
  - 传统TCP: newReno、Cubic

一个简单的多路径TCP机制



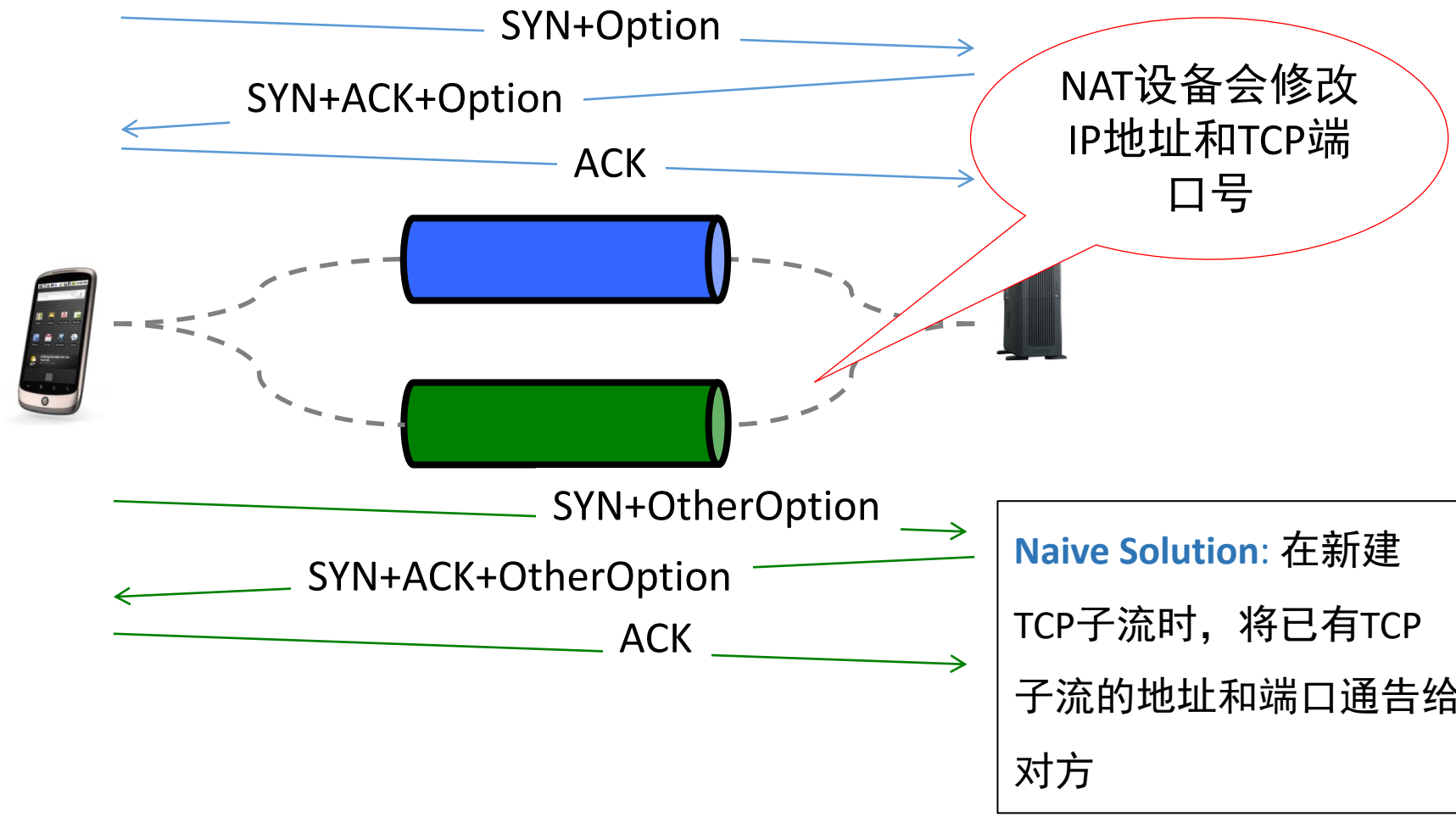
MPTCP连接

- 为了能够让MPTCP数据在每条路径上正常通过，每条路径上传输的数据包应该看起来像一个正常TCP流的数据包

- 一个MPTCP连接由一个或多个正常TCP子流构成
  - 端节点维护TCP子流与MPTCP连接之间的映射关系
  - 每个TCP子流在单条路径上传输，看起来就像一个正常TCP流

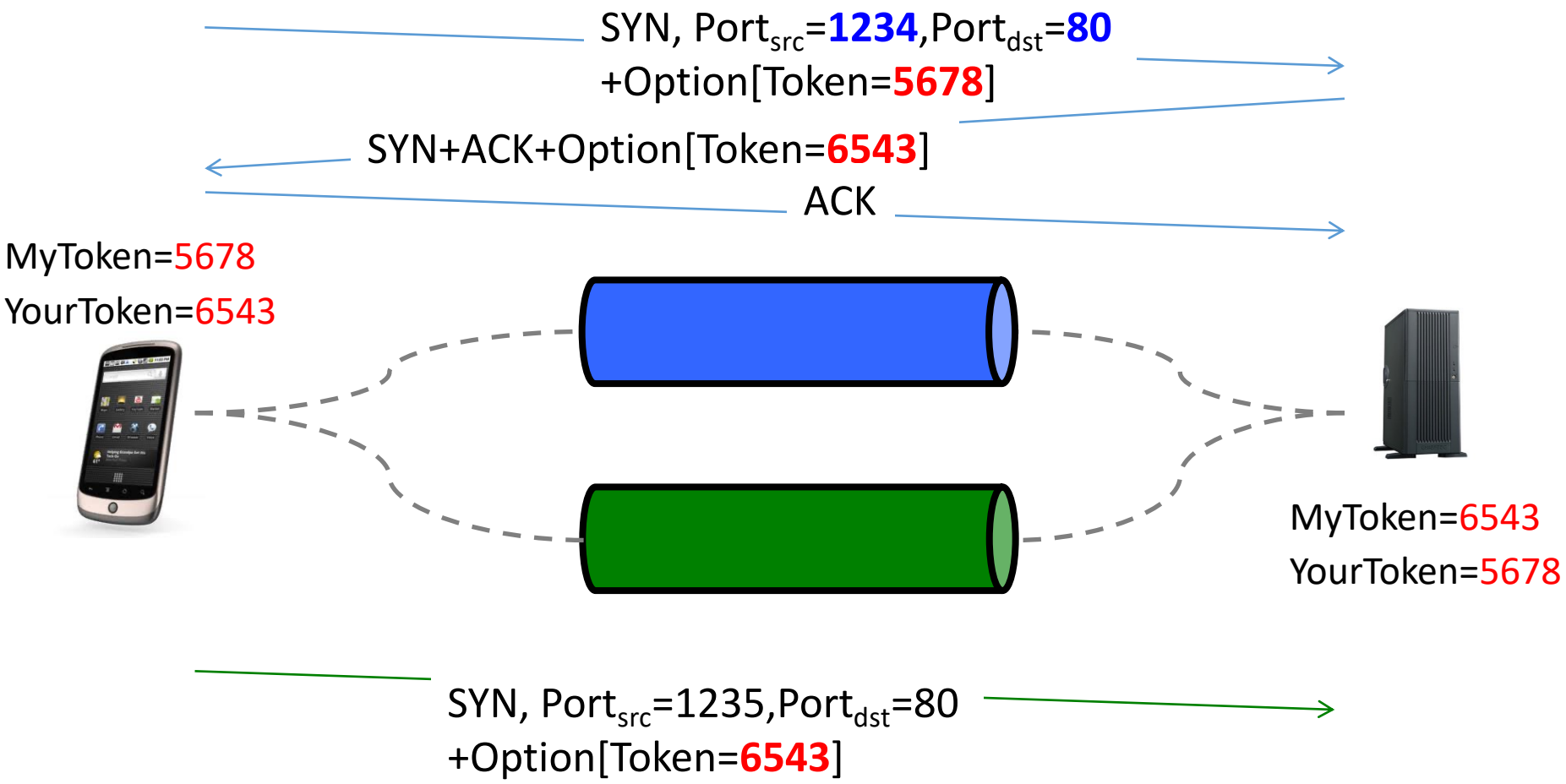
- 两个问题：
  1. 如何将不同TCP子流关联到同一MPTCP连接？
  2. 如何将数据分发到不同TCP子流进行传输？

如何关联不同TCP子流？

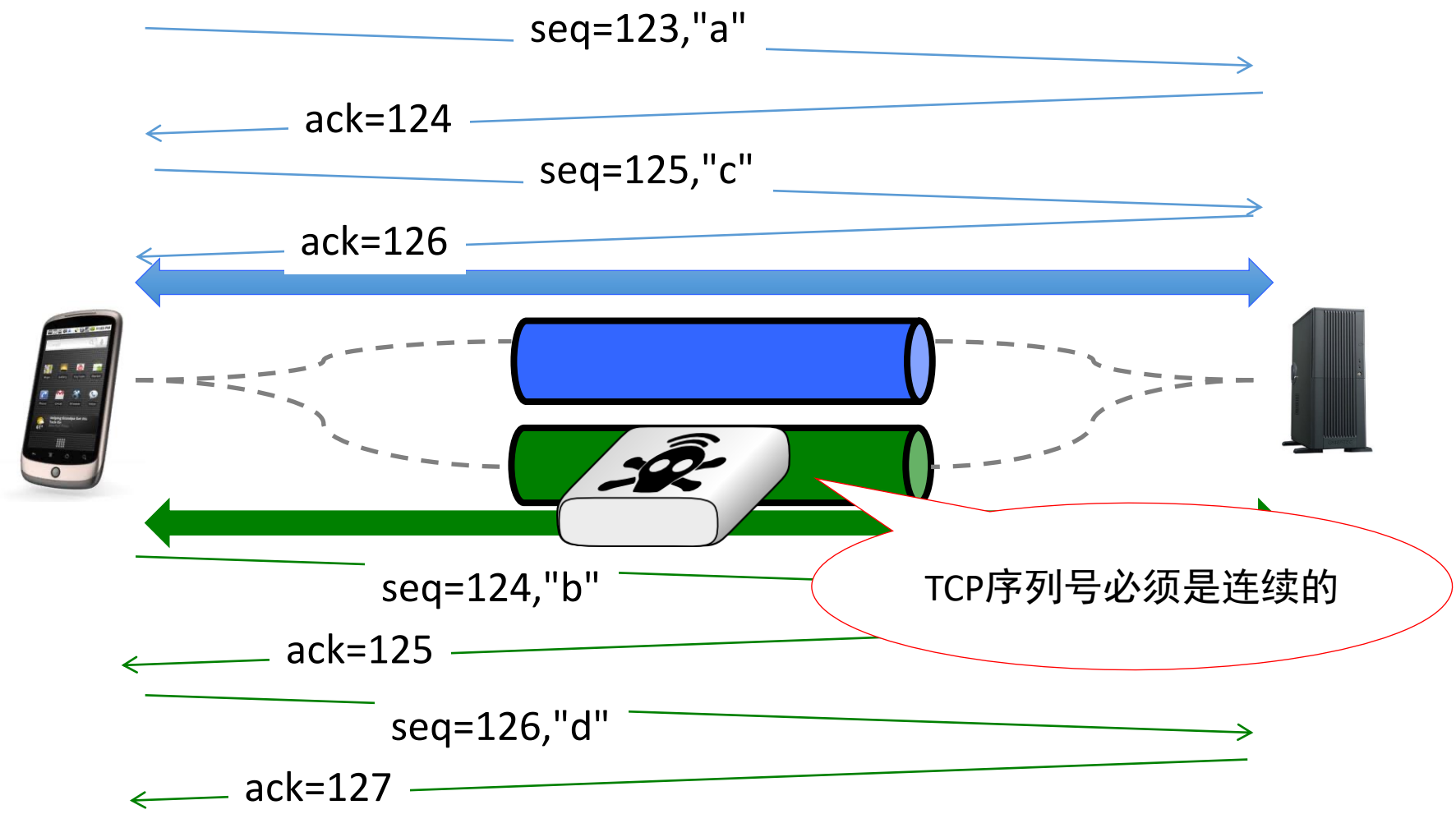


Naive Solution: 在新建TCP子流时，将已有TCP子流的地址和端口通告给对方

关联不同TCP子流

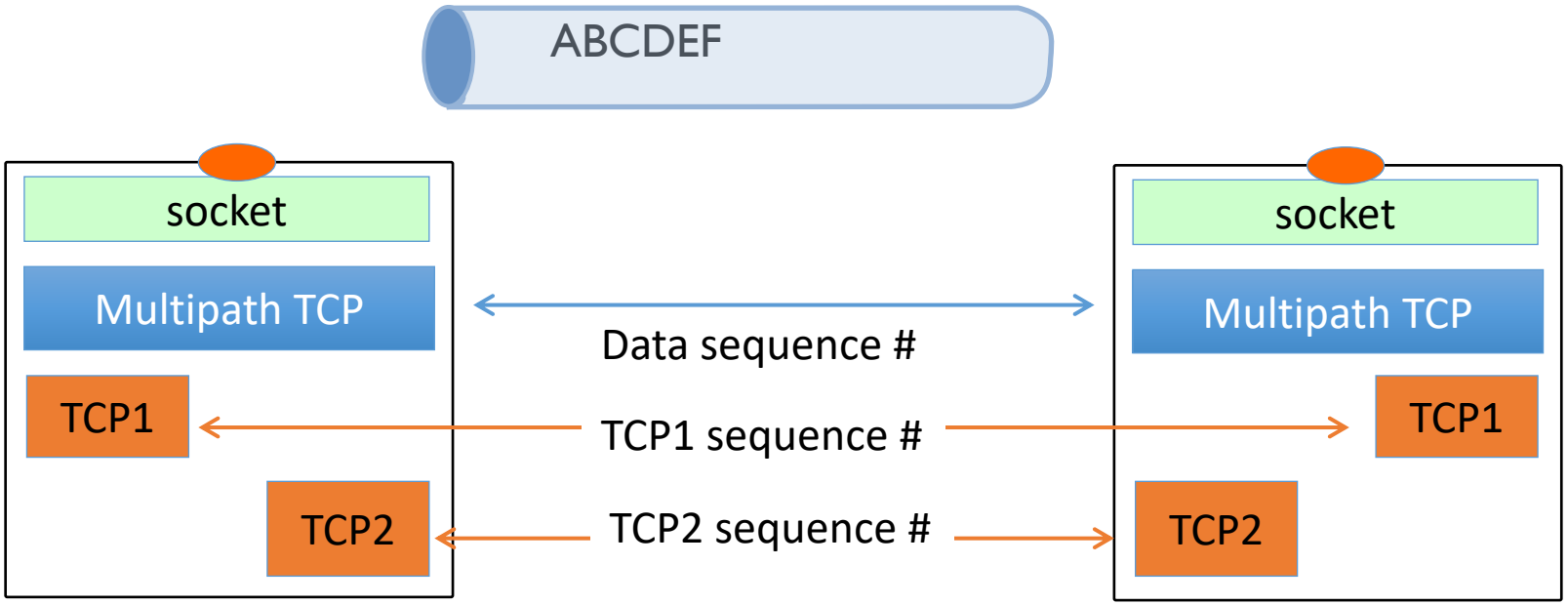

SYN, Port$_{src}$=1234,Port$_{dst}$=80
+Option[Token=5678]
SYN+ACK+Option[Token=6543]
ACK
MyToken=5678
YourToken=6543
MyToken=6543
YourToken=5678
SYN, Port$_{src}$=1235,Port$_{dst}$=80
+Option[Token=6543]

如何传输数据？

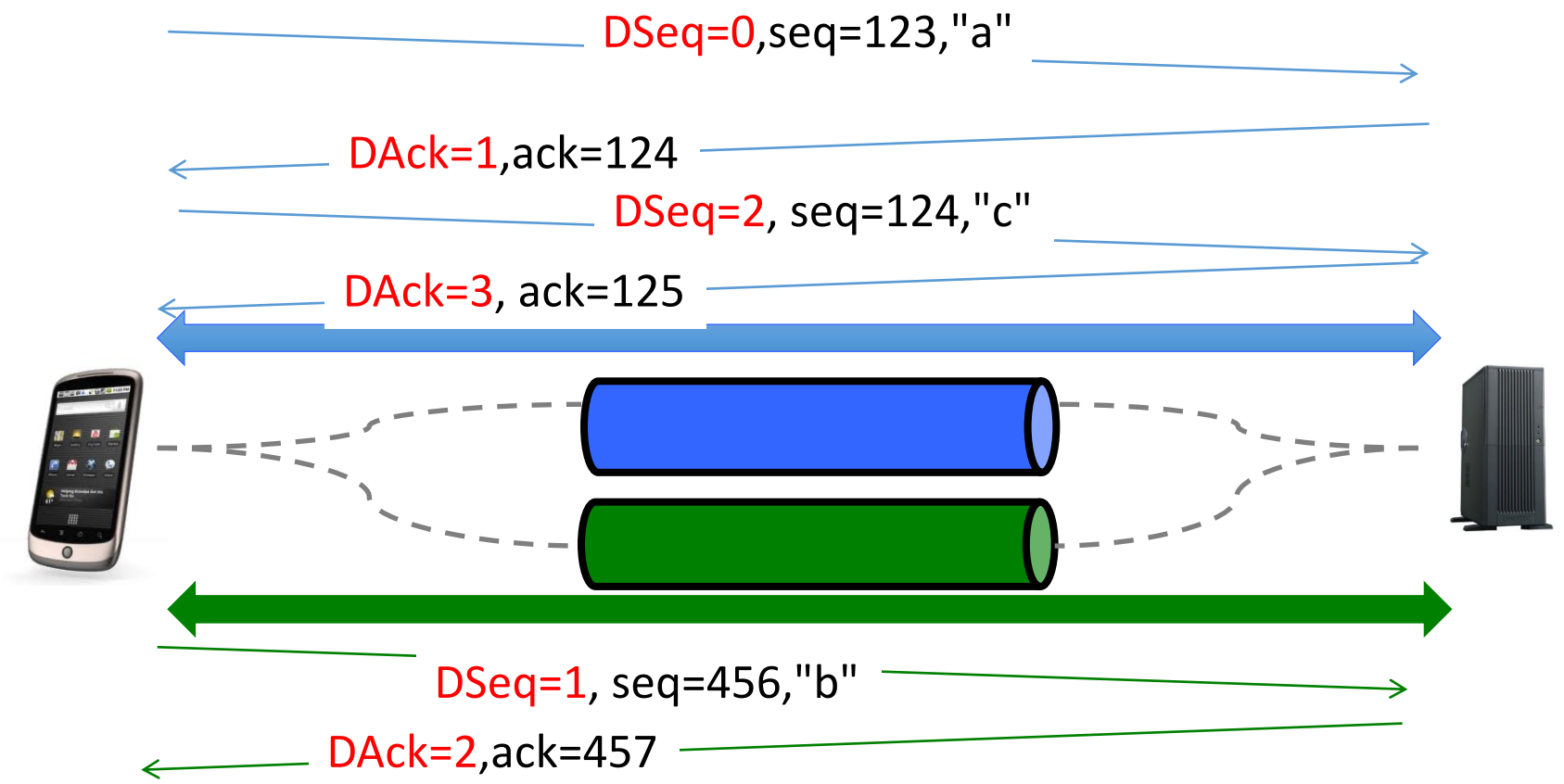

seq=123,"a"
ack=124
seq=125,"c"
ack=126
seq=124,"b"
TCP序列号必须是连续的
ack=125
seq=126,"d"
ack=127

MPTCP两层序列号结构

- 引入新的序列号空间，标准TCP头部使用子流序列号，扩展选项中使用全局序列号

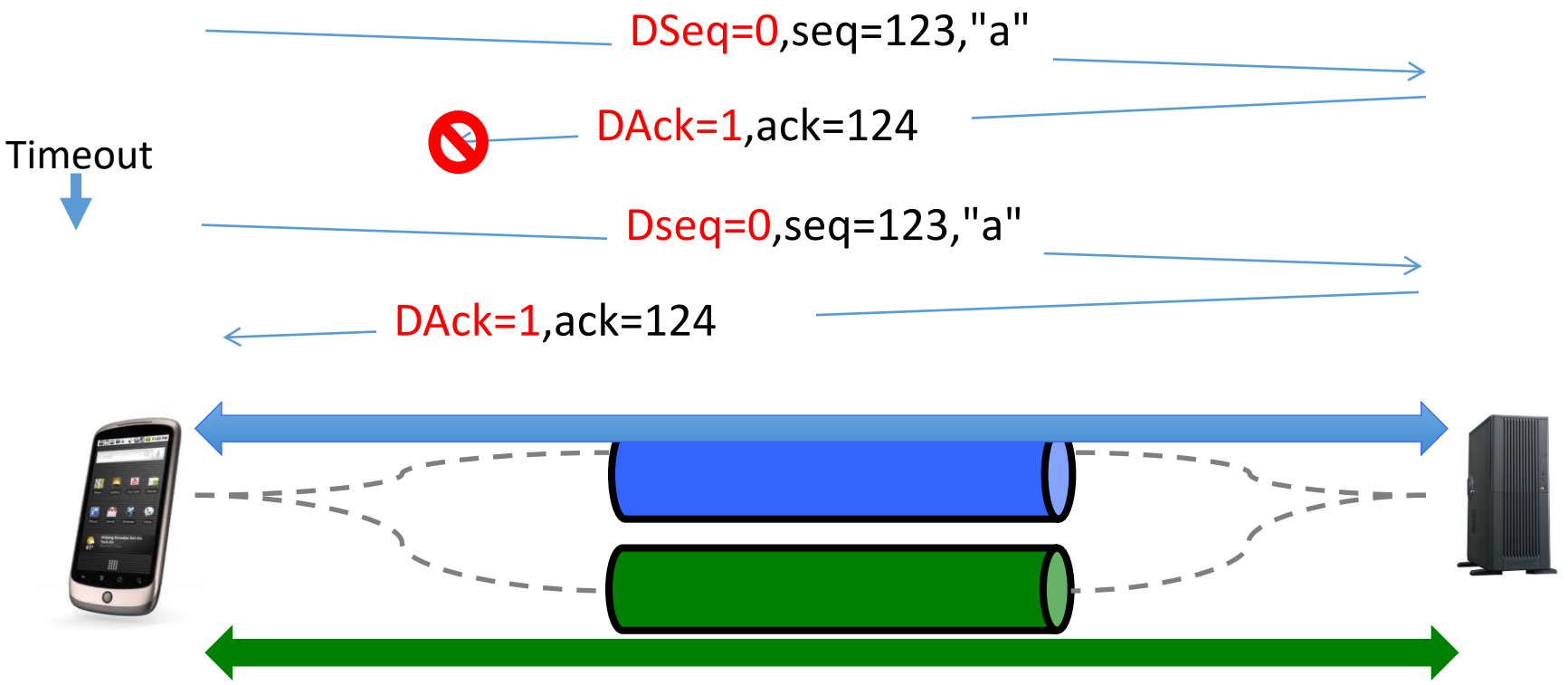


MTCP数据传输


DSeq=0,seq=123,"a"
DAck=1,ack=124
DSeq=2, seq=124,"c"
DAck=3, ack=125
DSeq=1, seq=456,"b"
DAck=2,ack=457

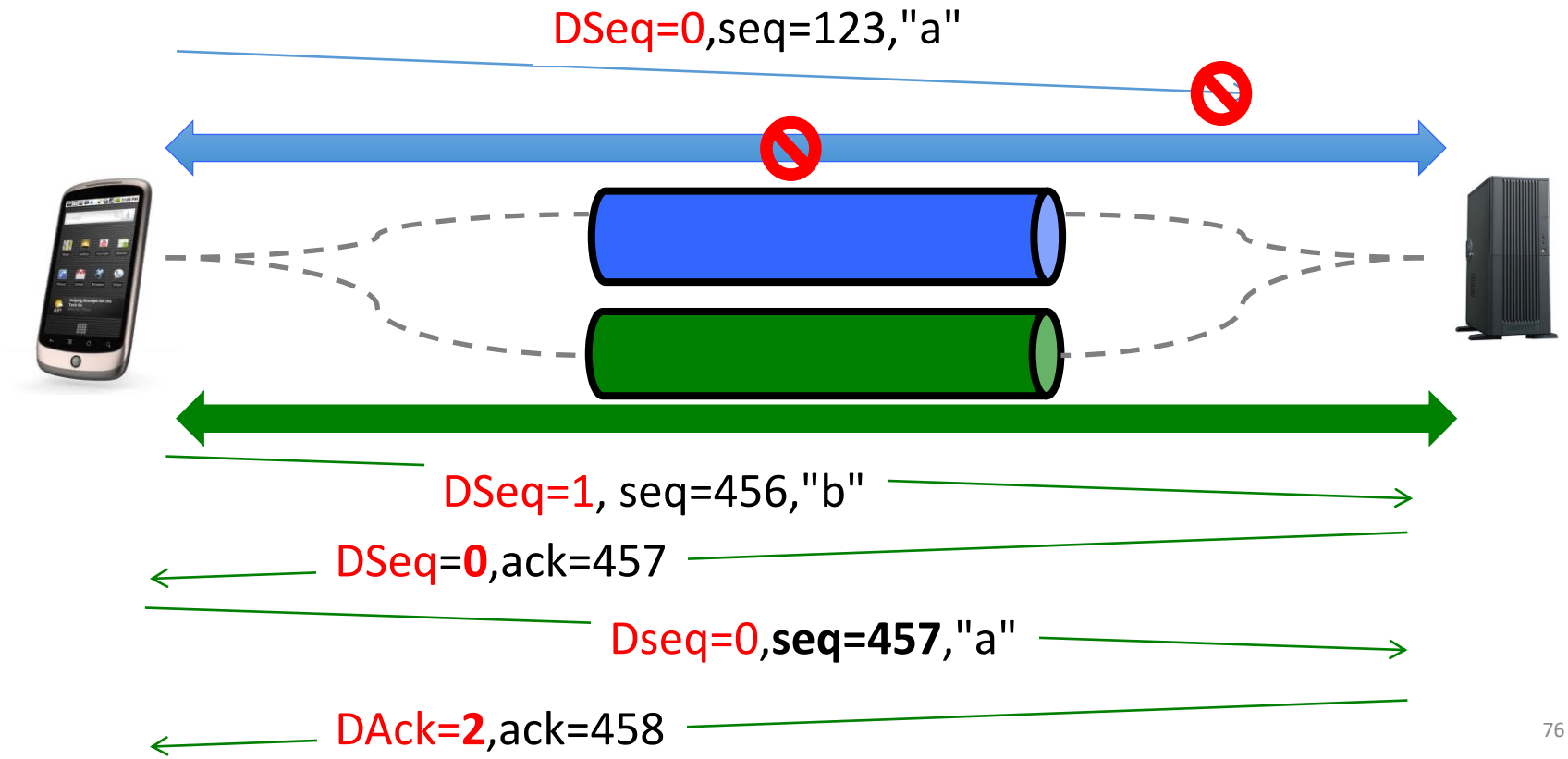
MPTCP数据丢包恢复

- 可以在同一TCP子流中进行恢复
  - 快速重传、超时重传机制与传统TCP一致



MPTCP数据丢包恢复

- MPTCP也可以跨子流进行丢包恢复
  - 当一条子流异常断开时；加速丢包恢复效率



MPTCP重传条件

- 快速重传：
  - 在同一子流中进行，如同传统TCP的快速重传
- 超时重传：
  - 当触发超时重传定时器时，评估该数据段是否由其他子流进行重传
- 当一条TCP子流异常退出后，该子流所有未被确认的数据都有其它子流重传

MPTCP消息传递

- 相比于TCP，MPTCP引入了额外控制消息
  - 数据序列号(DSeq)、数据确认号(DAck)、…

使用传统TCP扩展选项

优点：与传统TCP兼容；即使Middlebox不支持，也可以回退到传统TCP

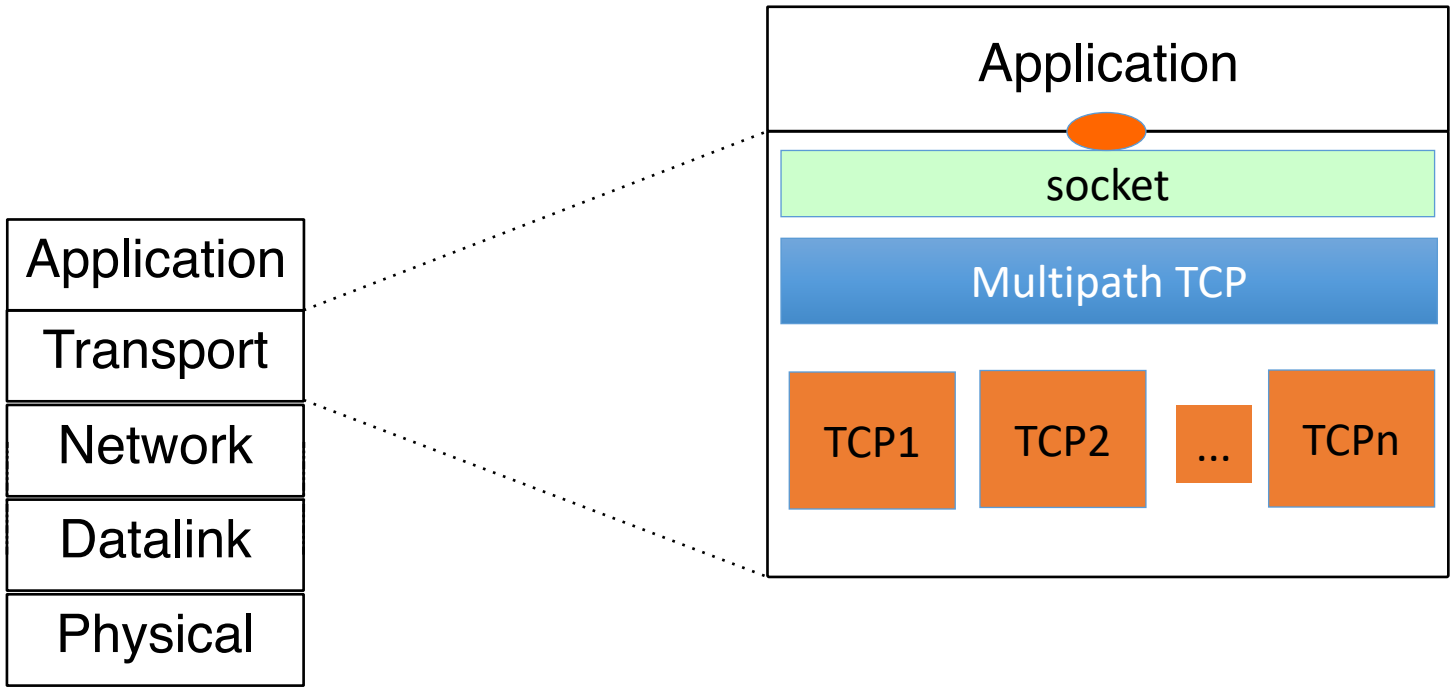
缺点：TCP扩展选项空间有限，能放置的选项数目有限

使用TLV机制 (Type-Length-Value)

优点：将TLV信息放到数据负载区中，长度不受限制，可扩展性好

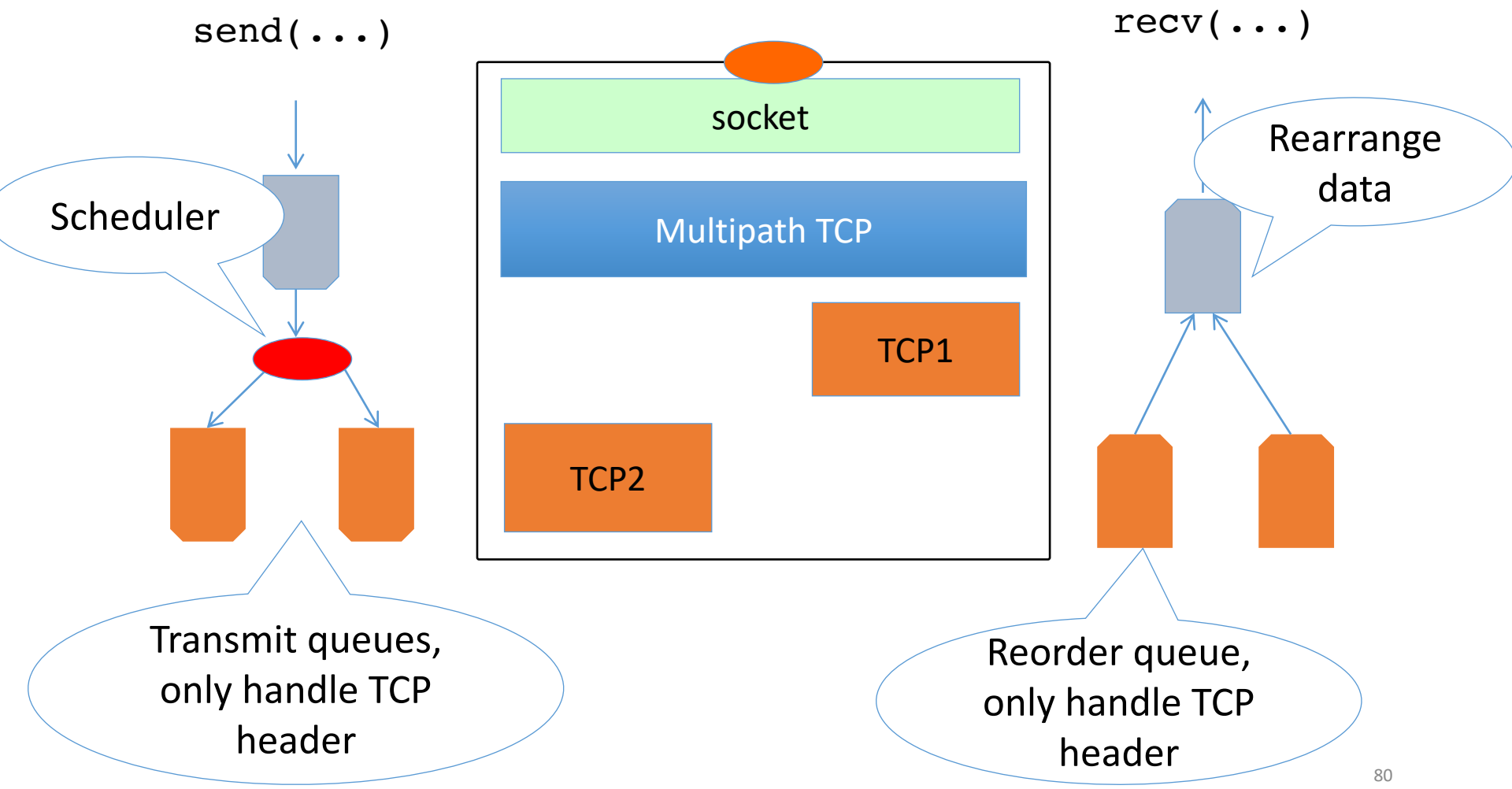
缺点：反而对Middlebox造成影响，DPI实现更复杂

MPTCP实现



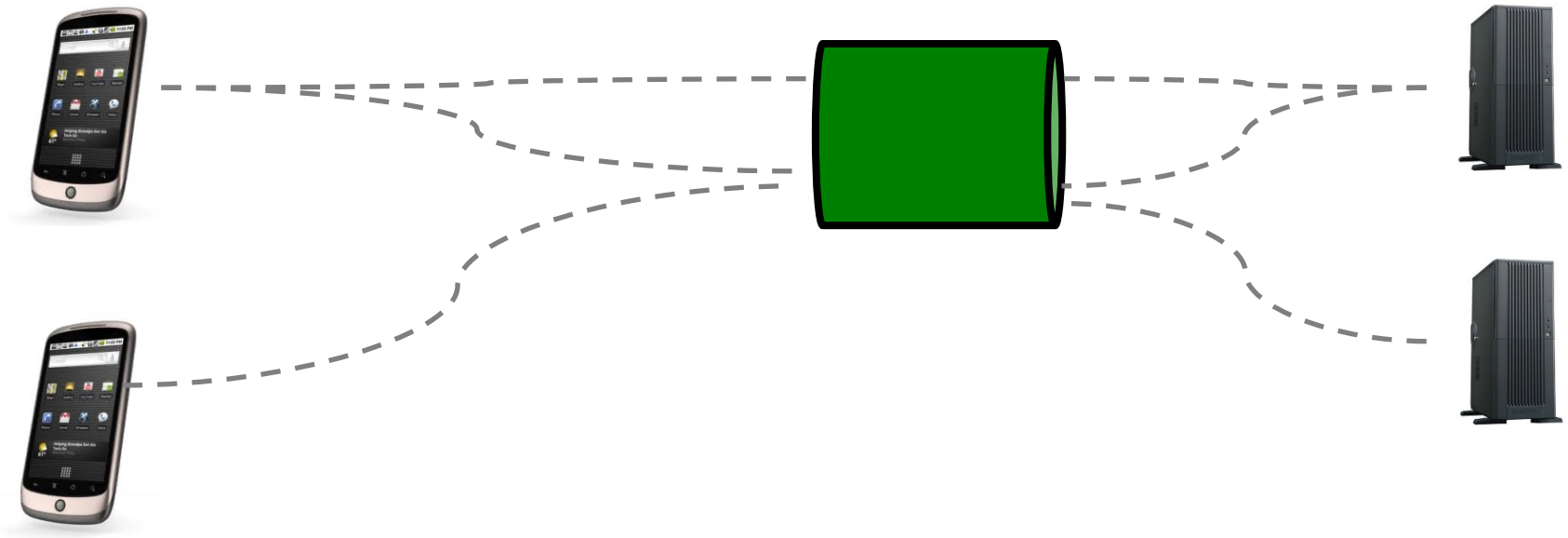
A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar, “Architectural guidelines for multipath TCP development", RFC6182 2011.

MPTCP协议栈


send(...)
recv(...)
socket
Multipath TCP
TCP1
TCP2
Scheduler
Rearrange data
Transmit queues, only handle TCP header
Reorder queue, only handle TCP header

MPTCP如何进行拥塞控制？

- 多个TCP子流共享拥塞窗口？
  - 不同路径的带宽资源不同
- 每个TCP子流维护独立的拥塞窗口？
  - TCP公平性

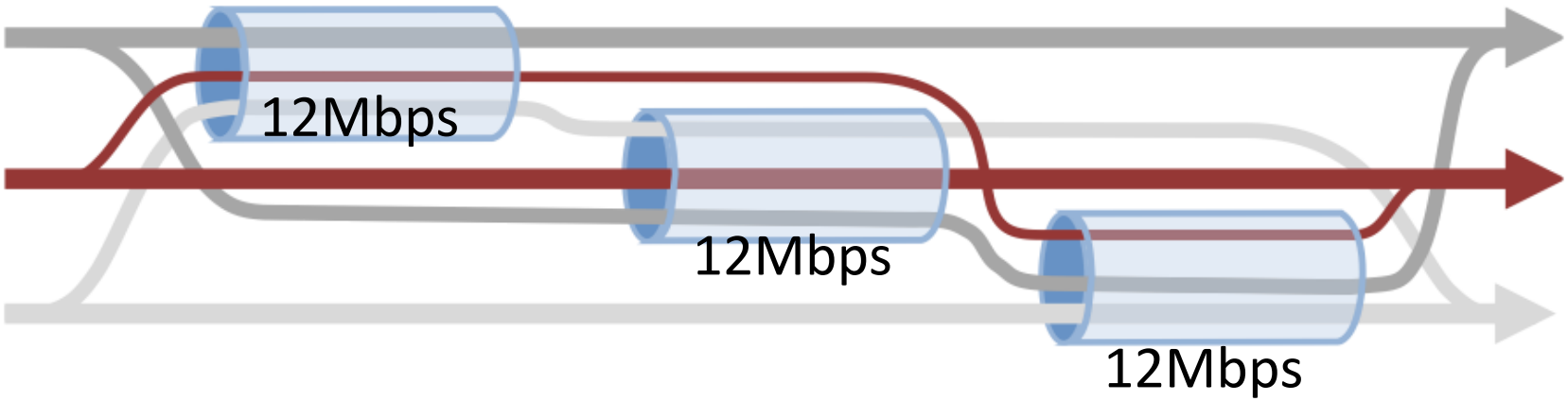


MPTCP拥塞控制

- MPTCP应该具有较好的公平性和友好性

- EWTCP
  - 对于TCP子流f，每收到一个ACK， $cwnd_f=cwnd_f+a/cwnd_f$
  - 每遇到一个丢包，$cwnd_f=cwnd_f/2$
  - 如果a = 1/sqrt(n)，则吞吐率与传统TCP相同
    - 窗口大小与a^2成正比

EWTCP主要问题



- 在上图场景中
  - 理想情况下，每个MPTCP连接的吞吐率应该为12Mbps
  - 如果使用EWTCP，则每个MPTCP连接只能获得8.5Mbps的吞吐率
    - 在两跳的路径中：3.5Mbps，在一跳的路径中：5Mbps
    - 主要问题在于，不同子流之间关联性较差

MPTCP拥塞控制改进: LIA

- LIA (Linked Increases Algorithm)
  - 对于TCP子流f，每遇到丢包，$cwnd_f = cwnd_f/2$
  - 每收到一个ACK

$$cwnd_f = cwnd_f + \min\left(\frac{\max\left(\frac{cwnd_i}{(rtt_i)^2}\right)}{\left(\sum_i \frac{cwnd_i}{rtt_i}\right)^2}, \frac{1}{cwnd_f}\right)$$

  - 不同子流之间通过alpha进行关联

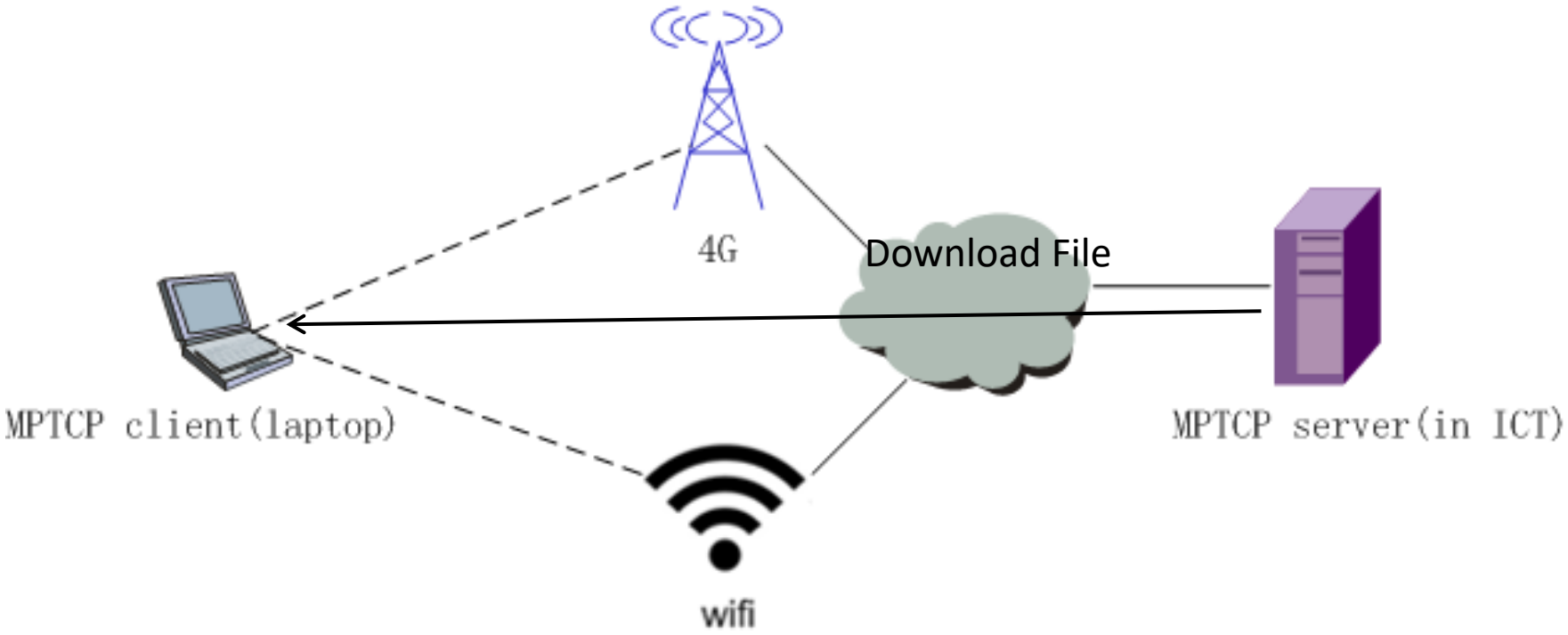
$$\text{alpha} = \max\left(\frac{cwnd_i}{(rtt_i)^2}\right) / \left(\sum_i \frac{cwnd_i}{rtt_i}\right)^2$$

其它MPTCP拥塞控制算法

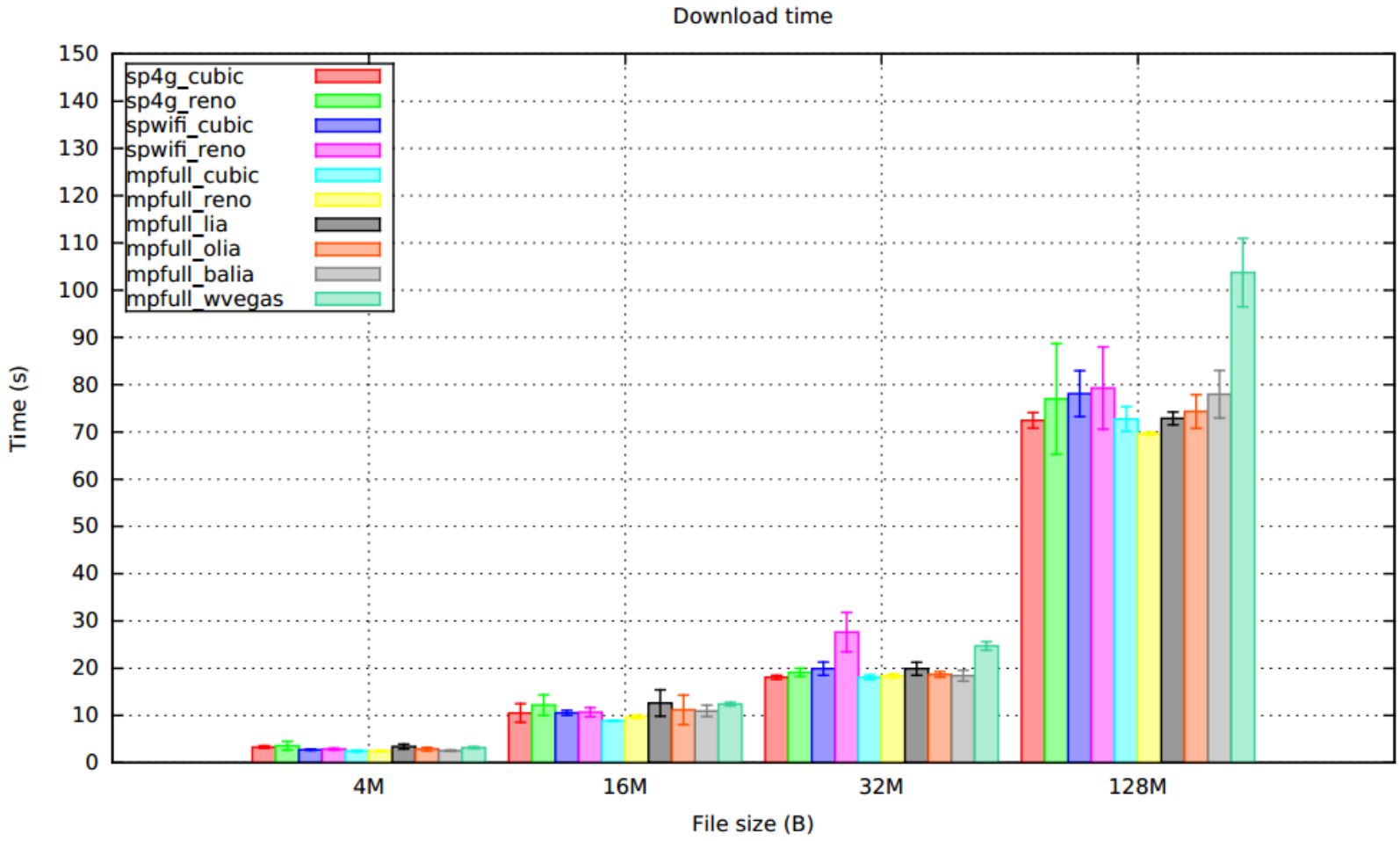
- 类似于TCP拥塞控制算法，MPTCP拥塞控制也有很多变种
  - OLIA (Opportunistic Linked Increases Algorithm)
  - BALIA (Balanced linked adaptation)
  - MPTCP Cubic
  - WVegas [Weighted Vegas)

MPTCP拥塞控制性能对比场景

- 实验环境



MPTCP不同拥塞控制算法的性能

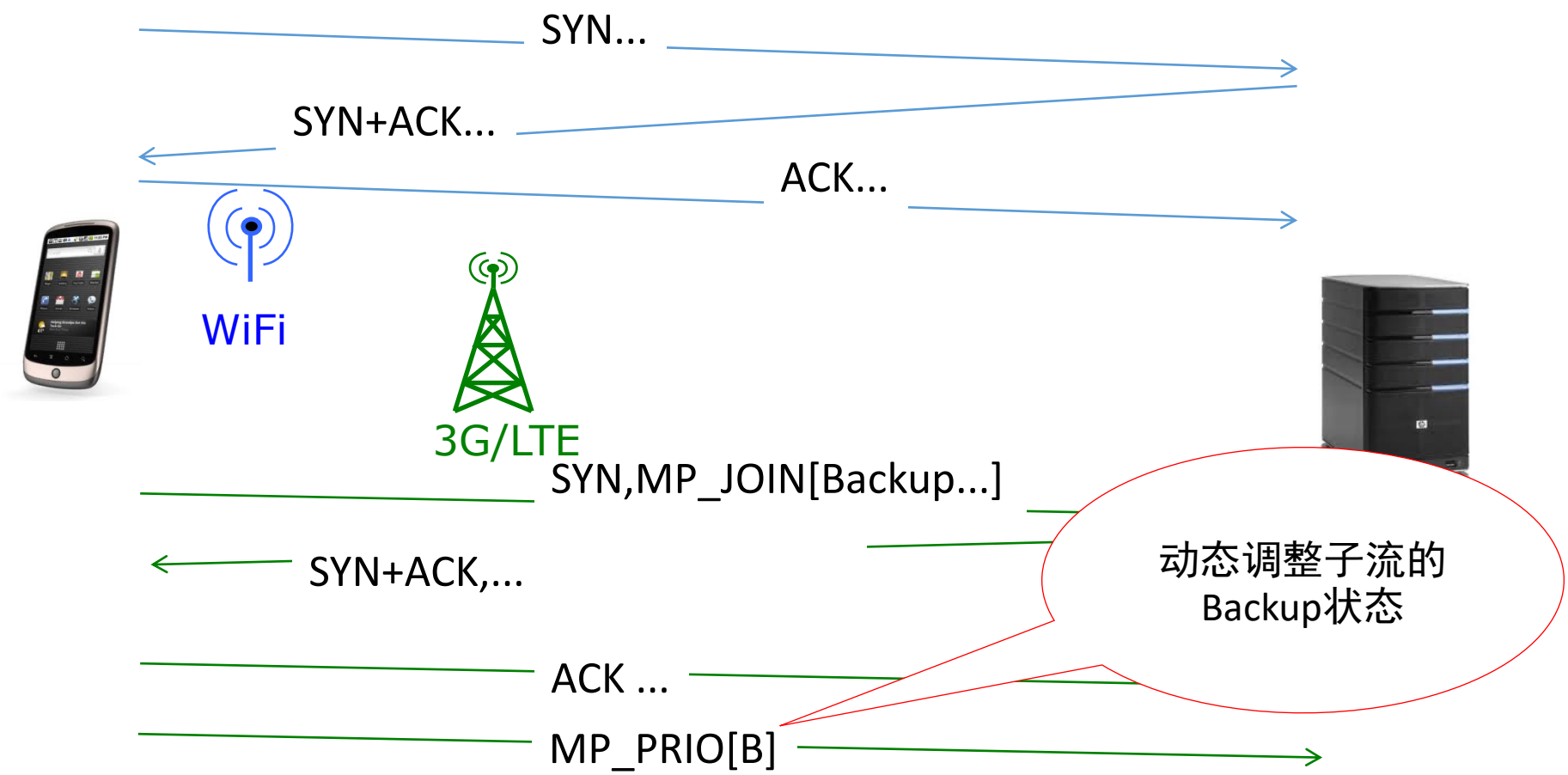


MPTCP的移动性支持

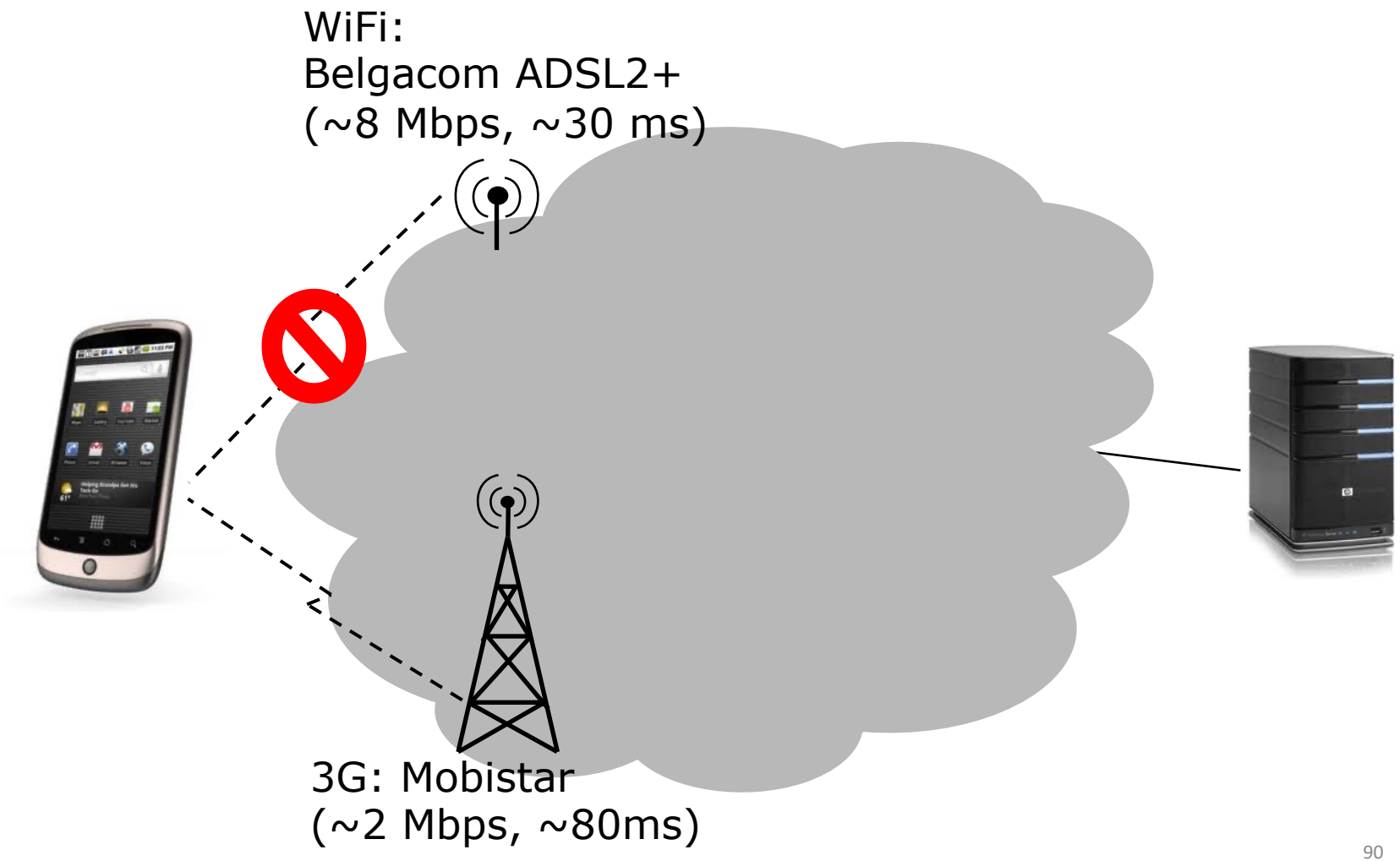
- MPTCP如何利用多路径 3G/4G和WiFi？

- Full mode：两种接入方式同时使用
- Backup mode：优先使用WiFi，并且将3G/4G 子流作为备份
- Single path mode: 同一时刻只使用一种接入方式

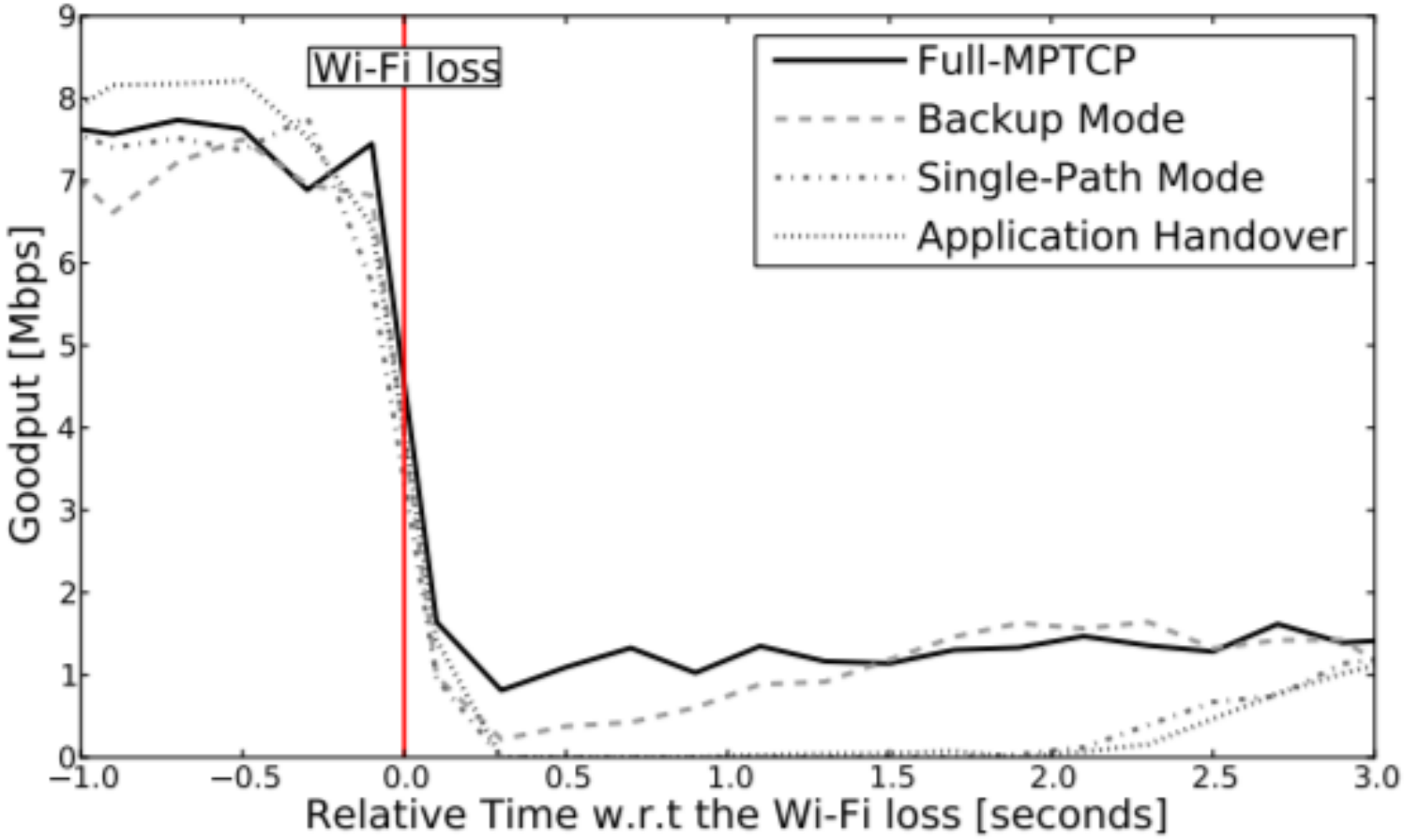
MPTCP的Backup模式



MPTCP移动切换实验场景



MPTCP移动切换性能



TCP演进方向

- TCP是网络环境与网络应用之间的性能适配器
  - TCP需要适配不同的网络环境
    - DCTCP、MPTCP等
  - TCP需要满足不同应用的性能目标
    - Throughput、Delay、Throughput/Delay
- TCP演进主要在端设备进行
  - 不修改TCP头部、选项
  - 模块化是演进的重要保证
- TCP演进遵循：
  - 端到端原则、可扩展性、公平性等
- 基于UDP的新型传输协议是一种趋势

课后阅读

- 《计算机网络 – 系统方法》
  - 第5.1、5.2、6.3节

- 新网络环境下的TCP
  - Mohammad Alizadeh et al. Data center tcp (dctcp). ACM SIGCOMM 2010
  - Costin Raiciu et al. How hard can it be? designing and implementing a deployable multipath TCP. USENIX NSDI 2012

谢谢！