

# CAN201 - Week 6

## Transport Layer (3) & Network Layer (1)

InubashiriLix (Github)

## 0. Roadmap

本讲分两大部分：

1. 传输层 (3): TCP 拥塞控制 & 传输层功能演进
  - 拥塞的原理与代价
  - 端到端 vs 网络辅助拥塞控制, ECN
  - TCP 的 AIMD 拥塞控制(慢启动、拥塞避免、快速重传/恢复)
  - TCP 吞吐量估算
  - 面向新场景的传输层功能演进 (QUIC)
2. 网络层 (1): 网络层概览与路由器体系结构
  - 网络层服务模型、数据平面 vs 控制平面
  - 路由器体系结构与转发: 输入端口、交换结构、输出端口
  - 队列与调度: HOL 阻塞、队列管理、调度算法
  - 互联网网络层实例: IP、ICMP、路由协议
  - IP 报文格式与分片

## 1. 拥塞控制 (Congestion Control) 原理

### 1.1 拥塞的基本概念与现象

- 非正式定义:
  - “\* 太多源 (too many sources) 在以太快速率 (too fast) 发送太多数据 (too much data), 超过网络处理能力 \* , 导致拥塞。”
- 典型表现:
  - 路由器缓存队列排队时间变长 → \* 时延增大 \*;
  - 缓存溢出时丢包 → \* 丢包率上升 \*;
  - 丢包引发重传, 重传又加重负载 → \* 带宽浪费 \*, 有效吞吐下降。
- 重要区分: \* 拥塞控制 vs 流量控制 \*:
  - 拥塞控制 (congestion control):
    - 面向 \* 网络整体 \* (many senders, routers, links);
    - 目标: 防止/缓解网络内部某处 “太多流量挤在一起”。
  - 流量控制 (flow control):
    - 面向 \* 端系统一对发送方-接收方 \*;
    - 目标: 防止发送方太快, 导致接收方缓冲溢出。

拥塞控制: “太多发件人 + 太多邮件 + 邮局处理不过来”  
流量控制: “一个人写信太快, 对方拆信太慢, 家里信箱塞满”

### 1.2 拥塞的成因与代价

- 成因 (简化链条):
  1. 负载上升: 很多主机同时发送数据, 路由器需要处理更多分组。
  2. 队列堆积: 到达速率 > 出口链路传输速率 → 分组在路由器缓冲区排队。
  3. 缓冲溢出: 缓存用尽 → 新来的分组被直接丢弃。
  4. 重传加剧拥塞:
    - 端系统把丢包解释为 “需要重传”, 重传分组进一步增加负载。
- 结果:
  - 平均时延增大, 抖动增大;
  - 有效吞吐降低 (大量带宽浪费在多次发送同一分组上)。

### 1.3 拥塞控制的两大思路: 端到端 vs 网络辅助

#### 1.3.1 端到端拥塞控制 (End-to-end)

- 网络 \* 不提供显式拥塞反馈 \* (不告诉你现在有多拥塞);
- 发送端仅根据自己观察到的:
  - 丢包事件 (超时、重复 ACK);
  - RTT 增大 (队列变长);
  - 来“推测”网络是否拥塞, 并调整发送速率。
- 优点:
  - 不要求路由器/网络升级协议;
  - 完全由端系统实现;
- TCP 采用的就是这种思路的主流方案 (AIMD)。

#### 1.3.2 网络辅助拥塞控制 (Network-assisted)

- 路由器在发现自己拥塞时:
  - 显式向相关端系统发送拥塞信号;
  - 例如:
    - 标记数据包头部的特定位 (如 ECN);
    - 发送 ICMP 等控制报文;
    - 端系统根据该显式信号调整发送速率。
- 可携带的信息:
  - “有无拥塞”;
  - 拥塞程度;
  - 甚至直接给出 “建议发送速率”。

### 1.4 ECN: 显式拥塞通知 (Explicit Congestion Notification)

- ECN 是一种网络辅助拥塞控制的机制:
  - 在 IP 头 (ToS/DS 字段) 中使用 2 bit 的 ECN 字段:
    - =00: Not-ECT= (Non ECN-Capable Transport);
    - =01: ECT(1);
    - =10: ECT(0)= (ECN-Capable Transport);
    - =11: CE (Congestion Experienced)。
- 工作流程简化:

1. 发送方、接收方都支持 ECN, 连接建立时协商启用。
  2. 发送方在 IP 报文中设置 ECN=ECT, 表明 “本连接可处理显式拥塞通知”。
  3. 某个路由器检测到本端口拥塞:
    - 不丢包, 而是在经过的 ECT 报文中将 ECN 标记改为 CE (11)。
  4. 接收方拆出 IP 头, 看到 CE 标记:
    - 在 TCP ACK 段中设置 ECE 标志位, 通知发送方 “本连接路径上经历拥塞”。
  5. 发送方看到 ECE=1:
    - 按约定减小拥塞窗口 (cwnd), 即降低发送速率。
- 优点:
    - 不必以 “丢包” 为主要拥塞信号, 避免了额外重传;
    - 更 “柔和” 的拥塞反馈。

## 2. TCP 拥塞控制: AIMD 与具体算法

### 2.1 AIMD 的核心思想与 sawtooth 行为

- TCP 拥塞控制的核心: AIMD (Additive Increase, Multiplicative Decrease)
  - 加性增 (AI): 线性探测更高带宽;
  - 乘性减 (MD): 在检测到拥塞时快速降速。
- 具体行为 (理想化):
  - 每个 RTT 内, 发送方把 cwnd 增加约 1 MSS (加性增)。
  - 一旦检测到“丢包”:
    - 通过 triple-duplicate ACK:  $cwnd \leftarrow cwnd / 2$
    - 通过 timeout:  $cwnd \leftarrow 1 \text{ MSS}$  (更激烈的退避)
  - 在长期运行中:
    - cwnd 呈现“锯齿”(sawtooth)形: 线性上升, 遇到丢包后乘性下降;
    - 这就是课件上画的“TCP sending rate vs time”的 sawtooth 曲线。
  - AIMD 之所以重要:
    - 在数学上已证明其有良好的网络稳定性与公平性:
      - 多个 TCP 连接的拥塞窗口会收敛到一种“相对公平”的共享带宽;
      - 分布式、异步执行也能保证整体稳定。

### 2.2 cwnd 与发送速率关系

- TCP 发送方限制未确认数据量:
- LastByteSent - LastByteAcked < cwnd
- 每个 RTT 大致可以发出 cwnd 字节:
    - 于是 理想速率约等于:
- TCP rate  $\approx cwnd / RTT$  (bytes/sec)
- cwnd 是一个由拥塞控制算法动态调整的变量:
    - 初始很小;
    - 根据 ACK 反馈逐渐增加;
    - 一旦检测到丢包事件就乘性减小。

### 2.3 慢启动 (Slow Start): 指数探测阶段

- 连接开始时:
  - 发送方对路径可用带宽几乎一无所知;
  - 若直接线性增加 cwnd, 会很慢;
  - 因此采用 \*slow start\*, 虽叫“慢启动”, 实际上是 \*指数上升\*。
- 规则:
  - 所以:
    - cwnd:  $1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow \dots$  MSS, 指数级增长;
    - 能在较少 RTT 内迅速探测到一个“合理的”速率水平。

初始化: cwnd = 1 MSS

每收到一个 ACK:  
 $cwnd \leftarrow cwnd + 1 \text{ MSS}$

若所有 ACK 在 1 RTT 内都到达:  
 每 RTT 结束时 cwnd roughly 翻倍

- 所以:
  - cwnd:  $1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow \dots$  MSS, 指数级增长;
  - 能在较少 RTT 内迅速探测到一个“合理的”速率水平。

### 2.4 从慢启动到拥塞避免 (Congestion Avoidance)

- 问题: 指数增长不能一直持续, 否则必然过快触顶导致严重拥塞。
- 思路:

- 设置一个阈值  $=ssthresh=(slow start threshold)$ :
  - $cwnd < ssthresh$  时  $\rightarrow$  使用慢启动 (指数增长);
  - $cwnd \geq ssthresh$  时  $\rightarrow$  切换到拥塞避免, 采用线性加性增 (AI)。
- 转换逻辑:
  - $ssthresh$  在发生丢包时更新:
    - 超时 (timeout) 发生:
      - $ssthresh \leftarrow cwnd/2$  (四舍五入)
      - $=cwnd \leftarrow 1 \text{ MSS}$  (重新慢启动)
    - 三次重复 ACK (triple dup ACK):
      - TCP Reno:  $ssthresh \leftarrow cwnd/2$
      - 进入快速重传/快速恢复 (下节)。

### 2.5 快速重传与快速恢复 (fast retransmit & fast recovery)

- 丢包检测方式有两种:
  1. 超时 (timeout): 比较粗糙, 时间长;
  2. 重复 ACK (duplicate ACK): 更快、更“聪明”。
- \* 快速重传 (Fast Retransmit) \*:
  - 当发送方收到 \*3 个额外重复 ACK\* (共 4 个相同 ACK) 时:
    - 认为 ACK 所指的下一个字节对应的段很可能丢失;
    - 不等超时, 立即重传该段 (失的那一段)。
- \* 快速恢复 (Fast Recovery, Reno 风格) \*:
  - 在三次重复 ACK 时:
    - $ssthresh \leftarrow cwnd / 2$
    - $=cwnd \leftarrow ssthresh + 3 \text{ MSS}$  (考虑队列中仍有 3 个重复 ACK 表示的段在飞)
    - 重传丢失的段;
    - 每收到一个额外 dup ACK:
      - $cwnd \leftarrow cwnd + 1 \text{ MSS}$  (认为还有未 ACK 的段被接收);
    - 一旦收到一个 新的 ACK (不再是重复 ACK):
      - $cwnd \leftarrow ssthresh$  (恢复到拥塞避免的线性阶段)。
  - 与超时相比:
    - 重复 ACK 的信号更快出现  $\rightarrow$  更快恢复;
    - 超时则意味着 RTT 估计严重偏大或网络反馈迟钝  $\rightarrow$  触发“更猛烈”的退避 ( $cwnd \rightarrow 1 \text{ MSS}$ )。

### 2.6 TCP 拥塞控制状态机 (Reno 逻辑总表)

可以用状态机角度理解 (简化版):

初始:

```
cwnd = 1 MSS
ssthresh = 64 KB (或其他初始值)
dupACKcount = 0
状态 = 慢启动
```

事件处理:

- 1) 新 ACK 到达:
  - 若状态 = 慢启动:
    - $cwnd += 1 \text{ MSS}$
    - 若  $cwnd \geq ssthresh \rightarrow$  状态转为 拥塞避免
  - 若状态 = 拥塞避免:
    - $cwnd += \text{MSS} * (\text{MSS} / cwnd) \approx$  每 RTT 增加 1 MSS
  - 若状态 = 快速恢复:
    - $cwnd = ssthresh$  ; 进入 拥塞避免
    - $dupACKcount = 0$
- 2) 重复 ACK 到达:
  - $dupACKcount++$
  - 若  $dupACKcount = 3$  且状态  $\neq$  快速恢复:
    - $ssthresh = cwnd / 2$
    - $cwnd = ssthresh + 3 \text{ MSS}$
    - 重传“丢失段”
    - 状态 = 快速恢复
  - 若  $dupACKcount > 3$  且状态 = 快速恢复:
    - $cwnd += 1 \text{ MSS}$  ; 发送新段 (若允许)
- 3) 超时:
  - $ssthresh = cwnd / 2$
  - $cwnd = 1 \text{ MSS}$
  - $dupACKcount = 0$

- 重传丢失段
- 状态 = 慢启动

## 3. TCP 吞吐量估算 (忽略慢启动)

### 3.1 理想化 AIMD 模型

- 假设:
  - 连接处于长期稳定状态：一直在“加性增 → 乘性减”的 sawtooth 循环；
  - 有足够的数据发送（饱和发送）；
  - 忽略慢启动阶段；
  - 丢包发生在窗口大小达到某个值 =  $W$  字节时。
- 在一个 sawtooth 周期内：
  - cwnd 从  $W/2$  线性增长到  $W$  然后丢包，之后被减半：
    - 平均窗口大小  $\approx (W/2 + W)/2 = 3W/4$ 。
  - 每个 RTT 内发送  $\approx$  cwnd 字节：
    - 平均速率  $\approx (\text{平均窗口大小})/\text{RTT}$ 。

### 3.2 平均 TCP 吞吐量公式（简化版）

- 因此平均 TCP 吞吐量近似为：

$$\text{avg TCP throughput} \approx (3/4 * W) / \text{RTT} \text{ bytes/s}$$

- 更严格的公式会进一步把丢包概率  $p$  引入 (Reno 模型)：
  - $\text{throughput} \approx K * \text{MSS} / (\text{RTT} * \sqrt{p})$  (课堂可能后面或在书中给出)
- 但本讲 PPT 的重点是直观理解：
  - 拥塞窗口越大、RTT 越小 → 平均吞吐越高；
  - 丢包事件越频繁 →  $W$  越小 → 吞吐越低。

## 4. 传输层功能演进 & QUIC

### 4.1 传统 TCP/UDP 遇到的新挑战

- TCP/UDP 作为主力传输协议已经几十年，但新场景带来问题：

场景	主要挑战
长肥管道 (long fat pipes) 无线网络	大量 <i>in-flight</i> 分组，单次丢包会让整个流水线“塌陷”，恢复非常慢 丢包往往来自无线噪声/移动性，而 TCP 误认为是拥塞
超长 RTT 链路 (卫星等)	RTT 极大，超时与 ACK 往返全部放大
数据中心网络	对尾延迟极度敏感，突发拥塞对服务质量影响大
后台低优先级流量	希望有“低优先级 TCP”，不影响前台交互流程

- 在此背景下:
  - 出现了丰富的 TCP 变体 (CUBIC、BBR 等);
  - 以及一种趋势: \* 将部分传输层功能上移到应用层, 构建在 UDP 之上 \*。

## 4.2 QUIC: Quick UDP Internet Connections

- QUIC 是一个:
  - \* 应用层协议 \*;
  - 基于 UDP 之上;
  - 描述自己时强调:
    - 提供可靠数据传输、加密、安全、拥塞控制;
    - 在一条 QUIC 连接上复用多个应用层 stream;
    - 在 1-RTT, 甚至 0-RTT 内完成连接建立 + 安全协商 + 拥塞控制初始化。

### 4.2.1 HTTP/3 over QUIC vs HTTP/2 over TCP

- 传统:
  - HTTP/2 over TCP:
    - HTTP/2 多路复用多个逻辑流;
    - 但仍然在 **单条 TCP 连接** 上排队, 遇到丢包会 HOL (head-of-line) blocking:
      - 后续所有流数据都被前面那个丢包阻塞, 直到 TCP 重传完成。
- QUIC 设计:
  - HTTP/3 over UDP + QUIC:
    - 在 QUIC 之上多路复用多个 \* 应用级 stream\*;
    - 每个 stream 有自己的可靠性控制, 而共享一个统一的拥塞控制;
    - TCP 层面的 HOL 阻塞被避免:
      - 一个 stream 丢包不必阻塞其他 stream 的交付;
    - 安全 (类似 TLS) 与传输整合在 QUIC 中, 而非 “TCP + TLS 分层叠加”。

### 4.2.2 QUIC 与 TCP 的关系

- QUIC 在设计上借鉴了 TCP 中许多成熟思想:
  - 可靠数据传输: 序号、确认、重传;
  - 拥塞控制: loss-based/ delay-based control, AIMD 变种等;
  - 丢包检测: 超时 + “重复 ACK” 类似信号 (具体通过 QUIC 帧实现)。
- 但 QUIC 有更灵活的空间:
  - 由应用进程直接控制 (用户空间实现), 易于升级;
  - 不依赖 OS 升级内核 TCP 栈;
  - 可为不同应用 (Web、视频) 定制不同拥塞控制算法。

## 5. 网络层概览与服务模型

### 5.1 网络层的目标与角色

- 网络层总体目标:
  - 为上层 “传输分组 (datagram)” 提供从源主机到目标主机的传输服务。
- 在具体设备中的职责:
  - 端系统 (主机):
    - 发送端: 把传输层 segment 封装进 IP datagram, 下交链路层;
    - 接收端: 从 IP datagram 中拆出 segment, 交给传输层。
  - 路由器:
    - 在所有经过的 datagram 上执行 \* 转发 (forwarding) \*;
    - 查表决定 “下一跳” 或 “出口端口”。

### 5.2 网络层核心功能: 转发 vs 路由

- \*Forwarding (转发)\*:
  - 本地、逐路由器的功能;
  - 对每一个到达的 IP datagram:
    - 在路由器的 **转发表 (forwarding table)** 中查找;
    - 决定该 datagram 应该从哪个输出端口发出;
    - 将其交给对应链路层。
- \*Routing (路由)\*:
  - 全网范围的逻辑;
  - 计算从源到目的的路径 (route)。
- 比喻:
  - Routing = 规划自驾游路线 (规划从 A 到 B 经过哪些高速/匝道);
  - Forwarding = 开车在每个路口根据路牌左转/右转 (具体选择出口)。

### 5.3 数据平面 vs 控制平面

- 数据平面 (data plane):
  - 本地、按包、在路由器级别执行;
  - 决定单个到达的 datagram 如何被转发;
  - 通常在硬件中实现 (如 ASIC、TCAM), 纳秒级。
- 控制平面 (control plane):
  - 网络范围逻辑;
  - 对路由、转发表的内容进行计算与更新;
  - 典型方式:
    - 传统: 每台路由器运行路由协议 (RIP, OSPF, BGP 等), 分布式协同;
    - SDN (Software-Defined Networking): 集中控制器计算转发表, 统一下发。

#### 5.3.1 Per-router control plane

- 每个路由器都运行路由算法的一部分:
  - 接收/发送路由消息;
  - 独立计算自己的路由表;
  - 根据网络状态变化 (链路上/下、代价变化) 更新路由信息。

#### 5.3.2 SDN 控制平面

- SDN 架构下:
  - 控制平面从路由器中 “抽离” 出来, 集中在远程控制器上;
  - 控制器获取全局拓扑与状态;
  - 统一计算 “匹配-动作 (match + action) ” 规则;
  - 把这些规则以 **转发表** 的形式下发到每个交换机/路由器。
- 好处:
  - 逻辑集中控制, 便于管理与编程;
  - 路由策略、ACL、QoS 等都可由控制器统一更新。

### 5.4 网络服务模型: best-effort vs QoS

- 问题: 网络层能提供什么样的 “服务质量” (QoS)?
- 面向单个 datagram 的服务示例:
  - 保证交付 (guaranteed delivery);
  - 保证在某个最大时延内交付 (如 <40ms)。
- 面向一个 datagram 流的服务示例:
  - 有序交付 (in-order delivery);
  - 最小带宽保证;
  - 对包间隔变化 (jitter) 的限制。
- 对比不同网络体系结构的服务模型 (简化):

架构	服务模型	带宽保证	丢包	顺序	时间
Internet	Best effort	无	可能	不保	不保
ATM	Constant Bit Rate	固定速率	极低	保序	提供延迟保证
ATM	Available Bit Rate	最小保证	可能	保序	可能有约束
IntServ	Guaranteed	保证最小	几乎无	保序	提供严格时间保证
DiffServ	Differentiated	尝试提供	视分类	视分类	视分类

- 互联网真实采用的是 \*best-effort service model\*:
  - 不保证交付、不保证时延、不保证顺序、也不保证带宽；
  - 但其简单性 + 上层（传输层、应用层）配合，使其实用性极强。

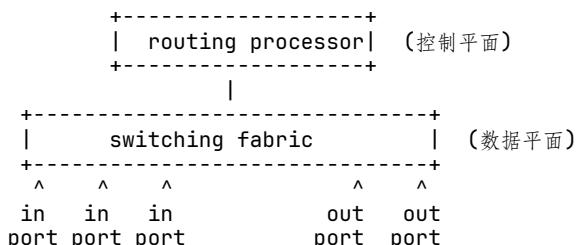
## 5.5 对 best-effort 的反思

- 优点:
  - 机制简单 → 协议栈与设备易于实现、扩展和部署；
  - 加带宽 (overprovisioning) 即可在统计意义上让大部分时间“够用”：
    - 语音、视频等实时应用通常在今天的互联网表现“够好”；
  - 大规模分布式应用（CDN、云数据中心）通过在“离用户更近的地方”提供服务来减少路径时延；
  - 传输层的拥塞控制（如 TCP）使得“弹性服务”（elastic services）能适应网络条件。
- 结论:
  - 虽然从理论上 QoS 保证更优雅，但 best-effort 的成功难以否定；
  - 在现实中，经常是“传输层 + 应用层 + 架构设计”共同实现“足够好”的 QoS。

# 6. 路由器体系结构与高速转发

## 6.1 路由器宏观结构

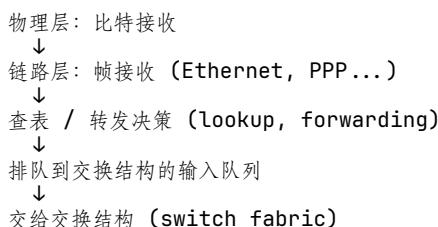
- 通用路由器结构：



- 数据平面（硬件）：
  - 高速交换结构（switching fabric）；
  - 输入端口（input ports）与输出端口（output ports）。
- 控制平面（软件）：
  - 路由处理器（routing processor）运行路由协议，维护转发表。

## 6.2 输入端口功能

- 输入端口处理流程：



- 查表与转发：
  - 使用到达分组头部字段（通常是 \* 目的 IP 地址 \*）；
  - 在 转发表中查找对应输出端口（“匹配 + 动作”）；
  - 优化目标：在“线速”（line rate）下完成查找。
- 目的地地址转发的两种形式：
  - 传统：\*destination-based forwarding\*：
    - 只根据目的 IP 地址决定转发；
  - 广义：\*generalized forwarding\*：
    - 可以根据任意头部字段（5 元组、标签等）决定动作。

## 6.3 目的地址转发与最长前缀匹配 (Longest Prefix Match)

- 转发表例子（目的地址范围映射到接口）：

Dest Address Range	Link Interface
200.23.16.0 - 200.23.23.255	0
200.23.24.0 - 200.23.24.255	1
200.23.25.0 - 200.23.31.255	2
otherwise	3

- 实际中，路由条目通常使用“前缀/掩码”（CIDR）描述：
  - 如：200.23.16.0/20、200.23.24.0/24 等。
- \* 最长前缀匹配规则 \*：
  - 查表时，对一个给定目的地址，可能匹配多条前缀；
  - 选择 \* 匹配前缀最长 \*（最具体）的那条；
  - 原因：路由聚合后需要通过更长前缀表达更精细的路由策略。
- 硬件实现：
  - 常用 ternary CAM (TCAM)：
    - 可以在一个时钟周期内完成匹配（支持 0/1/“任意”三态）。

## 6.4 交换结构 (switching fabric) 的类型

- 目标：在路由器内部把分组从输入端口高速转发到输出端口。
- 典型实现方式：

1. \* 通过内存 (switching via memory) \*:
  - 类似传统计算机体系结构：
    - 输入端口把分组复制到共享内存；
    - 路由处理器从内存读取，再写入相应输出端口缓冲；
  - 受限于内存带宽（每个分组两次总线访问），已不适合高端设备。
2. \* 通过总线 (switching via bus) \*:
  - 所有输入端口通过一条共享总线连接到所有输出端口；
  - 分组在总线上广播，输出端口根据目标接口接收；
  - 总线带宽成为瓶颈（bus contention）；
  - 适合中低速路由器。
3. \* 通过互连网络 (switching via interconnection network) \*:
  - 例如 crossbar、Clos 网络、多级交换结构：
    - 多个“小交换单元”组成一个大的  $n \times n$  交换网络；
  - 利用并行性：
    - 将 datagram 切分为固定大小的 cell；
    - 在多个 plane/stage 上并行交换；
    - 出口再重组为完整 datagram；
  - 可进一步采用多平面（multi-plane）结构，扩展到 Tbps 级。

## 6.5 输入端口排队与 HOL 阻塞

- 如果交换结构总处理速率小于所有输入端口到达速率之和：
  - 输入端口会排队；
  - 输入队列过长 → 队列时延增加 → 可能丢包（缓冲溢出）。
- \*Head-of-the-Line (HOL) blocking\*：
  - 若队列头部的分组因目的输出端口冲突不能进入交换结构；
  - 队列后面的分组即使本可以前往其他空闲输出端口，也被挡在后面；
  - 导致系统吞吐下降。

## 6.6 输出端口排队、缓存管理与调度

### 6.6.1 输出端口排队与缓存管理

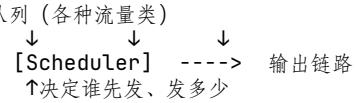
- 当从交换结构到达输出端口的速率 > 输出链路发送速率：
  - 必须在输出端口缓存（buffer）分组；
  - 过载时缓存溢出导致丢包。
- 缓存管理策略（buffer management）：

- \*tail drop\*: 缓存满时直接丢弃新到分组;
- \*priority drop\*: 按优先级丢弃;
- \*标记 (marking)\*:
  - 对分组打标记 (如 ECN、RED 随机早期丢弃/标记), 提前向端系统发出拥塞信号。

### 6.6.2 分组调度 (packet scheduling)

- 调度问题: 输出口有多个队列 (或一个队列多分类), \* 选择谁先发 \*?
- 常见调度策略:
  1. FCFS / FIFO (First-Come, First-Served):
    - 按分组到达队列的先后顺序发送;
    - 简单但无法区分流量类型。
  2. 优先级调度 (Priority Scheduling):
    - 按某种规则 (如 DSCP、端口号) 把流量分到不同优先级队列;
    - 调度器总是先从最高优先级有分组的队列中发送;
    - 在一个优先级内部可用 FCFS 排序。
    - 问题: 低优先级流量可能被长期 “饿死”。
  3. 轮询 (Round Robin, RR):
    - 将流量按类划分, 分别排队;
    - 输出链路轮流从各队列中取一个分组发送;
    - 各类在长期内拥有大致相同的机会。
  4. 加权公平队列 (Weighted Fair Queuing, WFQ):
    - RR 的加权版本:
      - 每个 class  $i$  有权重  $w_i$ ;
      - 在一个轮次中, 该队列获得  $w_i / \sum_j w_j$  的服务份额 (带宽)。
    - 作用:
      - 实现 per-class 的 \* 最小带宽保证 \*;
      - 用于 QoS 场景 (语音/视频 vs 背景流量)。

调度抽象:



## 7. 互联网网络层实例：IP、ICMP、路由协议

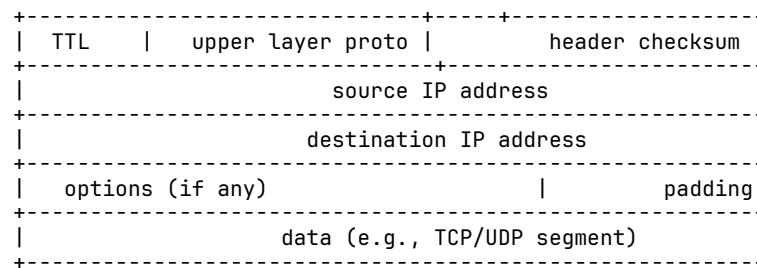
### 7.1 网络层协议族

- 在 IP 网络中, 网络层主要包括:
  1. \*IP 协议 (Internet Protocol)\*:
    - 地址 (IP addressing)、子网划分;
    - 报文格式 (datagram format);
    - 分片与重组;
    - 处理规则: TTL 递减、转发表查找等。
  2. \* 路由协议 \*:
    - 负责路径选择, 计算转发表:
      - 内部网关协议 (IGP): RIP、OSPF 等;
      - 外部网关协议 (EGP): BGP。
  3. \*ICMP 协议 \*:
    - Internet Control Message Protocol;
    - 用于错误报告 (如目的不可达)、诊断 (如 ping, traceroute);
    - 也用于某些路由器 “信号” (如重定向)。

### 7.2 IP 数据报 (datagram) 格式

- IP 头部主要字段 (IPv4):

0	4	8	12	16	20
version	IHL	Tos	total length (bytes)		
	identification		flags	fragment offset	



- 字段含义:
  - version: IP 协议版本 (4 或 6);
  - IHL (Internet Header Length): 头长;
  - ToS / DSCP / ECN:
    - Type of Service / Differentiated Services;
    - 包括前面讲的 ECN 标记;
  - total length: 整个 IP 数据报长度 (头 + 数据);
  - identification / flags / fragment offset:
    - 用于分片与重组;
  - TTL (Time To Live):
    - 每过一跳减 1, 减到 0 丢弃, 避免环路中无限循环;
  - upper layer protocol:
    - 指示负载是 TCP (=6) 还是 UDP (=17) 等;
  - header checksum:
    - 仅覆盖 IP 头部分;
  - options:
    - 例如记录路由 (record route)、时间戳、严格/松散源路由等;
    - 现代 Internet 中使用极少。
- 与 TCP 头叠加后的开销:
  - 常规 TCP + IP: 20 (TCP) + 20 (IP) = 40 字节 (不含 options)。

### 7.3 ToS / DSCP / ECN 简记

- ToS 早期用来表达优先级、延迟敏感度等;
- 现代使用 DSCP (Differentiated Services Code Point):
  - 提供多种服务类 (EF、AF 等), 用于 DiffServ QoS 架构;
- ECN bits:
  - 用于前面讲的显式拥塞通知。

## 8. IP 分片与重组 (Fragmentation & Reassembly)

### 8.1 MTU 与分片动机

- 每条链路都有自己的最大传输单元 (MTU):
  - 例如: 以太网典型 MTU = 1500 bytes;
  - 不同链路技术 → 不同 MTU。
- 若 IP 数据报长度 > 出口链路 MTU:
  - 必须把一个大 IP 数据报切成多个小数据报 (fragment) 在网络中传输;
  - 重组 (reassembly) 只在最终目的地主机进行, 中间路由器只负责分片。

### 8.2 分片相关字段与示例

- 相关字段:
  - identification: 同一原始数据报的所有分片共享同一 ID;
  - flags:
    - MF1 (More Fragments): 是否还有后续分片;
    - DF (Don't Fragment): 禁止分片标志;
    - fragment offset:
      - 当前分片在原始数据中的偏移 (以 8 字节为单位)。
- 示例 (4000 字节 IP 数据报, MTU=1500):
  - identification: 0x00000000
  - flags: 0x00000000
  - fragment offset: 0x00000000

- 每个 IP 数据报头假设 20 bytes → 每个分片数据部分  $\leq 1480$  bytes;
- 4000 字节数据需要切成 3 个分片：
  - Fragment 1:
    - length = 1500 ( $=20 + 1480$ );
    - offset = 0;
    - MF = 1 (还有后续)。
  - Fragment 2:
    - length = 1500;
    - offset =  $1480/8 = 185$ ;
    - MF = 1.
  - Fragment 3:
    - 剩余数据长度 =  $4000 - 1480 \times 2 = 1040$ ;
    - length =  $20 + 1040 = 1060$ ;
    - offset =  $(1480 \times 2)/8 = 370$ ;
    - MF = 0 (最后一个分片)。
- 目的主机根据：
  - (source IP, dest IP, identification) 找到属于同一原始报文的分片；
  - 根据 fragment offset 放回正确位置；
  - 当所有分片到齐（且 MF=0 的分片收到）后，重组成原始数据报。

4000B datagram (ID=x)

↓ MTU=1500

Fragments:

F1: ID=x, offset=0, MF=1, length=1500  
 F2: ID=x, offset=185, MF=1, length=1500  
 F3: ID=x, offset=370, MF=0, length=1040

- 实际网络中：
  - 频繁分片会严重影响性能（多个分片丢任意一个就要重传整个上层报文）；
  - 现代传输协议常会尽量控制 segment 大小（例如 Path MTU Discovery），避免路径中间分片。

## 9. 全讲总结：从 TCP 拥塞控制到网络层实现

- 传输层：
  - TCP 拥塞控制基于 AIMD：
    - 慢启动（快速探测可用带宽）；
    - 拥塞避免（线性探测）；
    - 丢包判据（超时 / 重复 ACK）；
    - 快速重传 / 快速恢复；
  - 吞吐量  $\approx (3/4 \cdot W)/RTT \rightarrow$  拥塞窗口与 RTT 共同决定性能；
  - 为新型场景引入 QUIC 等新协议，将可靠性/安全/拥塞控制上移到应用层、基于 UDP 实现。
- 网络层：
  - 功能：为端系统之间的 datagram 提供 best-effort 传输服务；
  - 数据平面：在每个路由器完成高速转发；
  - 控制平面：通过传统路由协议或 SDN 控制器计算并下发转发表；
  - 服务模型：互联网采用 best effort，但通过上层机制与大带宽“堆起来”，提供现实中足够好的体验；
  - 路由器体系结构：
    - 输入端口查表与排队；
    - 交换结构（内存/总线/互连网络）；
    - 输出端口队列与调度（FCFS、优先级、RR、WFQ）；
  - 互联网实例：IP 协议、路由协议、ICMP 协议及其协同；
  - IP 报文格式、ToS/DSCP/ECN、以及分片/重组机制。

整体逻辑链：

