1. Transport-layer services 概览

1.1 传输层的任务与位置

- **目标**： 为位于不同主机上的 * 应用进程 *（application processes）提供一种 “看起来像直接互相通信” 的 * 逻辑通信 *（logical communication）服务。
- **运行位置**：
  - 传输协议（TCP/UDP）只在 * 端系统 *（end systems）中实现；
  - 中间路由器只看到 IP 数据报，不理解 TCP/UDP 的端口等上层语义。
- 典型协议：
  - =TCP=（Transmission Control Protocol）
  - =UDP=（User Datagram Protocol）

```
应用层: HTTP, DNS, FTP, ...
   ↑
传输层: TCP, UDP
   ↑
网络层: IP
   ↑
数据链路层 / 物理层: 以太网, Wi-Fi, ...
```

1.2 Transport vs. Network layer（层间职责）

- **Network layer (IP)**：
  - 提供 **host-to-host** 的逻辑通信；
  - 负责：寻址（IP 地址）、路径选择（routing）、在路由器之间转发数据报。
- **Transport layer (TCP/UDP)**：
  - 构建在 IP 之上；
  - 提供 **process-to-process** 的逻辑通信；
  - 负责：
    - 把应用消息切分成一个或多个 segment 并交给 IP；
    - 在接收端把多个 segment 重新组装成应用消息；
    - 根据 **端口号**把消息交付给正确的应用进程（multiplexing / demultiplexing）。

1.2.1 家庭类比（Household analogy）

- Houses ↔ Hosts（主机）
- Kids ↔ Processes（进程）
- Letters in envelopes ↔ App messages in segments（带封套的信 ↔ 有头部的报文）
- Ann & Bill（在家分信的人）↔ Transport protocol（负责 demux）
- Postal service ↔ Network-layer protocol（负责楼与楼之间的投递）

1.3 Transport Layer Actions：发送端与接收端

1.3.1 发送端（Sender side）

传输层在发送端从应用层接收一条应用消息，执行：

1. 接收应用层消息（application message）；
2. 决定报文段头部各字段的取值：
   - 源端口 / 目的端口；
   - 对于 TCP：序号、确认号、窗口等；
   - 校验和等其他控制字段；
3. 将消息封装成一个或多个 *segment*（传输层报文段）；
4. 将 segment 交给网络层，由 IP 再封装入 IP datagram 并发送。

```
App message
     ↓
[Transport layer @ sender]
  - choose header fields
  - create segments
     ↓
IP datagrams
```

1.3.2 接收端（Receiver side）

传输层在接收端，从 IP 收到携带 segment 的数据报后：

1. 从 IP 数据报中取出传输层 segment；
2. 检查头部字段（校验和、端口号、标志位等）；
3. 利用头部信息执行 *demultiplexing*：
   - 找到对应的 socket（即某个应用进程使用的端点）；
4. 将 payload 重新组装为完整的应用层消息；
5. 向上交给应用程序。

```
IP datagram
    ↓
[Transport layer @ receiver]
  - extract segment
  - check header
  - demux to proper socket
  - reassemble app message
    ↓
Application process
```

1.4 两大核心传输协议：TCP vs UDP

协议	可靠性	顺序性	连接	拥塞控制	流量控制	典型应用
UDP	不可靠（best-effort）	不保证按序	无连接	无	无	DNS，实时音视频，自定义协议等
TCP	可靠，无丢失/重复	保证按序	面向连接	有	有	HTTP，SMTP，FTP，Telnet，…

- 对两者 **共同点**的精确定义：
  - 都不提供：*delay guarantee*（时延保证）、*bandwidth guarantee*（带宽保证）；
  - 都依赖 IP 的“尽力而为”服务，只是在端系统通过机制来改善质量。

2. Multiplexing & Demultiplexing：多路复用与分用

2.1 核心问题：一台主机上很多进程，如何区分？

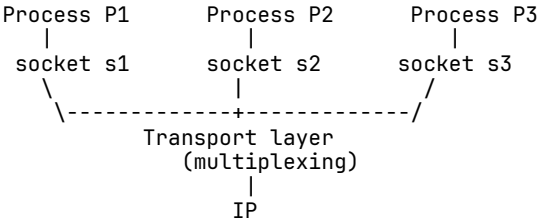
- 一台主机上可能同时运行：
  - 多个浏览器进程（甚至多个标签页）；
  - 聊天软件、视频软件、游戏客户端……
- 这些进程都在通过网络收发数据：
  - 收到一个 segment 时：* 究竟交给哪一个进程？*
  - 发送时：* 怎样把不同进程的数据混合走同一条 IP 通道？*

这就是 * 多路复用 *（multiplexing）和 * 分用 *（demultiplexing）的职责。

2.2 定义：Multiplexing vs Demultiplexing

2.2.1 Multiplexing（发送端）

- * 多路复用 as sender*：
  - 传输层从多个 socket 接收来自不同进程的数据；
  - 为每条数据增加合适的传输层头部（包含端口号等信息）；
  - 将生成的多个 segments 交给网络层，共同共享同一条 IP 通道。



2.2.2 Demultiplexing（接收端）

- * 分用 as receiver*：
  - 主机从 IP 层收到一个又一个数据报，其中每个都包含一个 segment；
  - 传输层从 segment 的头部读取：
    - 源 IP / 源端口；
    - 目的 IP / 目的端口；
  - 根据这些字段找到正确的 socket；
  - 将 payload 提交给相关联的应用进程。

2.3 socket 与端口号：分用的“地址”

- *socket*：应用进程与传输层交互的接口（抽象端点）。
- 每个 socket 至少绑定一个：
  - 本机 IP 地址（可为 ANY/0.0.0.0）；
  - 本地端口号（port number）。
- * 端口号范围 *（记忆用，不一定考试要求）：
  - 0-1023: well-known ports（HTTP 80, HTTPS 443, DNS 53, etc.）
  - 1024-49151: registered ports
  - 49152-65535: ephemeral ports（通常由 OS 临时分配给客户端进程）

在分用时，操作系统根据“IP + 端口”组合来判定 segment 属于哪个 socket。

2.4 “How demultiplexing works”精确过程

当接收主机 H 收到一个 IP 数据报时：

1. IP 层检查目的 IP 地址 =? H 的某个本地地址；
2. 若匹配，则把携带的传输层 segment 交给对应的 TCP 或 UDP 模块；
3. 传输层模块读取 segment 头部中的：
   - source IP address
   - destination IP address
   - source port number
   - destination port number
4. 利用这些信息，在本机 socket 表中查找匹配项：
   - 对 UDP：通常只看“本地端口 + 本地 IP”；
   - 对 TCP：使用完整的 4-tuple。
5. 找到 socket 后，将 payload 交给对应进程。

2.5 UDP: Connectionless demultiplexing（无连接分用）

2.5.1 基本规则

- UDP socket 由“本机地址 + 本地端口号”标识（host-local port number）。
- 典型创建方式（伪代码）：

```
Socket = socket(AF_INET, SOCK_DGRAM);
Socket.bind(("", 12345));  // 绑定本地 UDP 端口 12345
```

- 当主机收到任何 **目的端口 = 12345** 的 UDP segment 时：
  - 只要目的 IP 是本机，它就会被交给 **同一个** socket；
  - 无论来源 IP/端口各不相同。

2.5.2 “同一端口，多来源”的效果

- 结果是：一个 UDP 服务器进程，可以通过 **一个 socket** 同时与多个客户端通信：

```
客户端 A: (IP_A, port_A1) → 服务器: (IP_S, 12345)
客户端 B: (IP_B, port_B2) → 服务器: (IP_S, 12345)
客户端 C: (IP_C, port_C3) → 服务器: (IP_S, 12345)
```

所有这些 segment 最终都进入服务器上绑定 12345 的那一个 UDP socket。
应用层可以从 recvfrom() 中读取对方的 (src IP, src port)，自己区分是

- 这是“*connectionless demultiplexing*”：
  - 没有所谓“连接状态”，所有来自不同对端、发往同一端口的 segment 都共用一条“逻辑入口”。

2.6 TCP: Connection-oriented de-multiplexing（面向连接分用）

2.6.1 4-tuple 标识一条 TCP 连接

- TCP 为了区分不同连接，引入完整的四元组：

(src IP, src port, dest IP, dest port)

- 一个 TCP *socket 实例 *（即一个连接）由这四个值唯一标识。
- 接收端在分用时 * 必须同时匹配这四个字段 *，才能找到正确的连接。

2.6.2 Web 服务器示例

假设：

- Web 服务器 B 使用 TCP 端口 80 提供 HTTP 服务；
- 客户端 A、C 都来访问 B。

连接 1：

- 源 IP = IP_A，源端口 = 9157；
- 目的 IP = IP_B，目的端口 = 80；

连接 2：

- 源 IP = IP_C，源端口 = 5775；
- 目的 IP = IP_B，目的端口 = 80；

虽然 **目的 IP+ 端口**都是 (IP_B, 80)，但两个连接的 4-tuple 不同。因此服务器 B 会为每个连接维护一个独立的 TCP socket（含独立的发送/接收缓冲区、序列号、窗口大小等状态）。

```
Client A                          Server B (HTTP)
---------                         ----------------
src: A,9157  → dest: B,80   =>  socket 1: (A,9157,B,80)

Client C                          Server B (HTTP)
---------                         ----------------
src: C,5775  → dest: B,80   =>  socket 2: (C,5775,B,80)
```

- Web 服务器因此可以同时处理很多客户端：
  - 每个客户端对应一个独立的 TCP 连接(甚至非持久 HTTP：每个请求一个连接)。

2.6.3 对比 UDP 的分用方式

特性	UDP 分用	TCP 分用
socket 标识	(本机 IP, 本地端口)	(src IP, src port, dest IP, dest port)
收到 segment 时匹配字段	主要看目标端口（+ 目标 IP）	四元组全部匹配
一个服务器端口能否支撑多个 client	能；所有 client 共用一个 socket	能；每个 client 拿到一个独立连接 socket
是否存在“连接状态”	无；每个报文独立	有；需要维护连接状态（序列号、窗口、计时器等）

2.7 总结：服务与分用是如何衔接的?

- **传输层服务**决定了 segment 必须携带哪些头部字段:
  - 为了实现可靠/有序（TCP）或简单无连接（UDP）;
  - 为了在 host 内部识别“是谁发、发给谁”（端口号）。
- * 多路复用（发送端）*:
  - 把来自多个应用进程的数据，包装成带有端口信息的 segment;
  - 将这些 segment 混合后交给 IP，复用同一条 host-to-host 通道。
- * 分用（接收端）*:
  - 使用 IP 地址 + 端口号（UDP）或 4-tuple（TCP）从混合流中“拆出来”每个应用的专属数据流;
  - 保证每个进程看到的，只是属于自己的那条逻辑连接（或无连接消息流）。

从上往下看整条链路:

```
[Process P on Host A]
  ⇳  (via socket, port a)
[Transport layer (TCP/UDP)]
  ⇳  (segments inside IP datagrams)
[Network layer (IP): host A ⇔ host B]
  ⇳
[Transport layer on Host B: demux by (IP,port)]
  ⇳
[Correct Process Q on Host B]
```

3. Connectionless transport: UDP

3.1 UDP 的核心特性与使用场景

- * 定义 *: UDP（User Datagram Protocol）是运行在 IP 之上的一种 **无连接、尽力而为**的传输层协议。
- * 主要特性 *:
  - =Connectionless=: 发送前 **不需要**建立连接，没有三次握手;
  - =Best effort=:
    - 可能丢包;
    - 可能乱序;
    - 可能重复;
    - 不做重传、不保证到达。
  - =Message-oriented=: 应用交给 UDP 的一块数据就是一个 datagram，接收时也是“按块”收。
  - = 独立处理 =: 每个 UDP segment（报文段）被当成独立的“数据报”，互相之间无状态关联。
- * 典型使用 *:
  - 实时 / 流式多媒体: 语音通话、视频直播等（宁可少量丢包，也要低延迟）;
  - DNS 查询;
  - HTTP/3 的底层（QUIC 基于 UDP 自己实现可靠性和拥塞控制）;
  - 其他应用自带可靠性机制的场景（应用在上层做重传、纠错）。

3.2 UDP: Transport Layer Actions(动作流程)

3.2.1 UDP 发送端（sender）

以 PPT 中 SNMP client → SNMP server 为例:

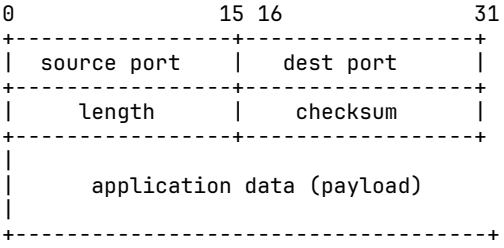
1. 应用层（SNMP 客户端）调用 socket 发送一条应用消息;
2. UDP 在发送端:
   - 接收应用层消息（=SNMP msg=）;
   - 决定 UDP 头部字段:
     - 源端口号（source port）;
     - 目的端口号（dest port）;
     - 长度（length）;
     - 校验和（checksum）;
   - 构造 UDP segment:
     - =header（8B）+ data（payload)=;
   - 把 UDP segment 交给 IP 层,由 IP 封装成 datagram 发送。

3.2.2 UDP 接收端（receiver）

1. IP 层把收到的 IP datagram 交给 UDP;
2. UDP 从 datagram 中取出 UDP segment;
3. UDP 执行:
   - 校验和检查（checksum）;
   - 根据 **目的端口号**找到正确的 UDP socket（demulti-plexing）;
   - 将 payload（SNMP msg）交给对应应用进程。

3.3 UDP segment header 结构

UDP 报文段固定 8 字节头部，后面跟应用数据:



- =source port=: 发送端端口号;
- =dest port=: 接收端端口号;
- =length=: 整个 UDP 报文长度（头部 + 数据），单位字节;
- =checksum=: 覆盖 UDP 头 + 数据 + 伪首部的 16 位校验和。

3.4 为什么要 UDP?（优势）

- * 无连接 *:
  - 不需要三次握手，减少时延;
  - 短小请求（例如 DNS）不必为建立连接付出额外开销。
- * 简单、无状态 *:
  - 发送端、接收端不维护复杂的连接状态（序列号、窗口、计时器等）;
  - 实现容易，资源占用更少;
  - 对于简单控制协议（SNMP 等）特别合适。
- * 头部开销小 *:
  - UDP 头部只有 8 字节，相比 TCP 最少 20 字节更轻量。
- * 无拥塞控制 *:
  - 协议层面不会主动 “减速”，应用可以按自己需要 “猛发”;
  - 适用于需要自己控制速率或使用专网的场景（但在公共 Internet 上需要小心，不要搞成 DDoS)。

3.5 UDP 校验和（checksum）与 Internet checksum

3.5.1 目标：检测比特错误

- 目的: 检测在链路传输过程中发生的比特翻转（bit errors);
- 基本思想: 发送端和接收端对同一块数据做同样的 “加和 + 取反” 运算，结果应一致。

3.5.2 发送端计算过程（概念版）

1. 把下面几部分视为一串 16 位整数:

    - “伪首部”（pseudo header：源/目的 IP 等，防止错误路由影响）；
    - UDP 头部（source port, dest port, length, checksum 字段把 checksum 位置先置 0）；
    - UDP 数据部分。
2. 对所有 16 位字进行 * 一补和 *(one’s complement sum)：
    - 普通二进制加法；
    - 如果最高位产生进位（溢出），把这 1 再加回低 16 位（wraparound）。
3. 得到的和再做一补（bitwise NOT），结果写入 checksum 字段。

3.5.3 接收端校验过程

1. 同样把 “伪首部 + UDP 头 + 数据” 分成 16 位整数；
2. 先把 checksum 字段也包括进去做一补和；
3. 检查和的结果是否等于全 1（0xFFFF）：
    - 是：认为 “无检测到错误”（但不能保证绝对没错）；
    - 否：认为报文损坏，丢弃或交给上层做错误处理。

3.5.4 一补和小例子（逻辑）

假设有两个 16 位数：

```
  1100110011001100
+ 1101010110101010
--------------------
  1101110111010110   (若产生进位，则回卷加到低 16 位)
  ...
  结果取反 => checksum
```

- 精确的逐位演示 PPT 已经给了若干张 slide，这里记住要点：
    - =wraparound carry=：最高位的进位要加回去；
    - 最终 checksum 是 “和的一补”;(接收端：和 + checksum ⇒ 全 1 才算通过)。

4. Principles of reliable data transfer（可靠数据传输原理）

4.1 服务抽象 vs 协议实现

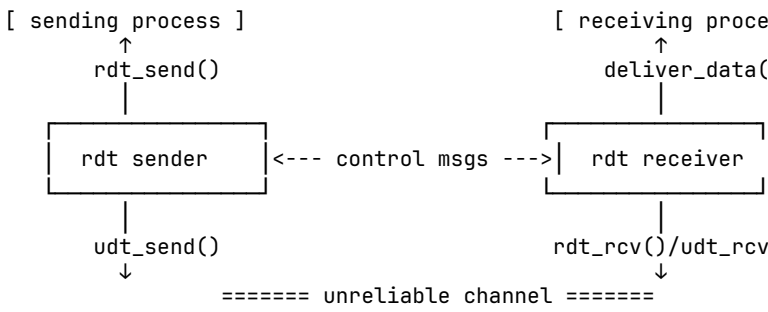
4.1.1 服务抽象（上层视角）

- 上层应用希望看到的是一个 “* 可靠信道 *”：
    - 发送进程交给传输层的数据 = 接收进程毫无差错地、按顺序收到的数据；
    - 不丢、不乱、不重复。

```
[ sending process ]  --(reliable channel)-->  [ receiving process ]
          ↑                                          ↑
      application                                application
          ↓                                          ↓
      transport                                  transport
```

4.1.2 底层现实（实现视角）

- 实际上传输发生在一个 **不可靠的底层信道**上：
    - 可能丢包；
    - 可能比特出错；
    - 可能乱序；
    - 可能重复。
- 为了对上层 “伪装” 出一条可靠信道，需要：
    - 在发送端和接收端实现一个 * 可靠数据传输协议 rdt*；
    - 做好序号、确认、重传、缓存、定时器等机制。



- 协议复杂度 **强烈依赖**底层信道 “糟糕程度”：
    - 仅有 bit errors?
    - 还会丢包?
    - 还会乱序? ……

4.2 rdt 协议接口（API）

- =rdt_{send}(data)=：上层调用，向对端发送数据；
- =udt_{send}(pkt)=：rdt 在发送端调用，把打包好的 packet 交给不可靠信道；
- =rdt_{rcv}(pkt)=：当底层信道在接收端交来一个 packet 时，由 rdt 调用；
- =$deliver_{data}$(data)=：rdt 在接收端调用，把正确的数据交给上层应用。

这是后面所有 rdtX.Y FSM 的 “外部接口框架”。

4.3 rdt1.0：完美信道上的可靠传输

4.3.1 假设

- 底层信道是 **完美**的：
    - 不丢包；
    - 不出错；
    - 不乱序。
- 因此：发送什么就到什么，不需要额外机制。

4.3.2 协议行为（FSM 思想）

- 发送端：
    - 拿到上层数据 → 打包成 packet → 调用 udt_{send} → 回到 “等待上层调用” 状态。
- 接收端：
    - 从底层拿到 packet → 直接把数据部分交给上层 $deliver_{data}$ → 回到 “等待底层来包”。

没有 ACK / NAK，没有序列号，协议极其简单。

4.4 rdt2.0：信道可能出现 bit errors（但不丢包）

4.4.1 新问题

- 信道可能翻转某些 bit：packet 到达但内容可能损坏；
- 需要检测错误，并在错误时重传。

4.4.2 新增机制

- * 错误检测 *：
    - 通过 checksum 检查（类似 UDP 的思路）；
- * 接收方反馈 *：
    - ACK（Acknowledgement）：确认某个包正确收到；

    - NAK（Negative ACK）：当前包有错误。
- * 发送方策略 *：
    - 收到 NAK 或检测到 packet 损坏 → 重传当前包。

4.4.3 rdt2.0 的 FSM 要点

- 发送端：
    - 状态 “等待上层调用”：接到数据，打包发送，进入 “等待 ACK/NAK” 状态；
    - 状态 “等待 ACK/NAK”：
        - 收到未损坏的 ACK → 回到 “等待上层调用”；
        - 收到 NAK 或损坏的控制包 → 重发上一个数据包。
- 接收端：
    - 收到未损坏的数据包 → 交付上层 → 发送 ACK；
    - 收到损坏的数据包 → 发送 NAK（不交付）。

4.5 rdt2.0 的致命缺陷 & rdt2.1 的改进

4.5.1 缺陷：ACK/NAK 自己也可能被损坏

- 若 ACK/NAK 在路上被 bit error 损坏，发送方的情况：
    - 它 **不知道**接收方到底有没有收到/交付这个包；
    - 如果简单 “再发一次数据包”，接收方可能会收到 * 重复包 *：
        - 上层可能被交付两遍同样的数据。

4.5.2 解决思路：引入 序列号与 “重复检测”

- 每个数据包带一个 seq#（序列号）；
- 接收方维护 “期望下一个收到的 seq#”：
    - 如果收到 seq# 正确且校验和正确 → 交付上层 + 发送 ACK；
    - 如果收到 seq# 重复或损坏 → 不交付上层，只回复适当的 ACK/NAK。
- 只要发送方 “在不确定时重发当前包”，重复包也不会再交付给上层。

4.5.3 rdt2.1：仅考虑 bit errors，无丢包

- 采用 1-bit 序列号（0 / 1）足够：
    - stop-and-wait 场景中，任何时刻只有 “当前一个包在飞”，交替使用 0/1 即可；
    - 不会混淆 “这是更早的某个旧包还是当前包”。
- 发送端状态：
    - “等待上层数据，期望序号 0”；
    - “等待 ACK/NAK for seq 0”；
    - “等待上层数据，期望序号 1”；
    - “等待 ACK/NAK for seq 1”。
- 接收端状态：
    - “期望 seq 0 的数据包”；
    - “期望 seq 1 的数据包”。

关键点：> **接收端状态 = 当前期望的 seq#；发送端状态 = 当前已发送且待确认的 seq#。**

4.6 rdt2.2：NAK-free（仅用 ACK）

4.6.1 设计动机

- 实际实现里，ACK/NAK 两种控制类型有点多余；
- 可以仅使用 ACK，并通过 “* 重复 ACK*” 来表达 NAK 的语义：
    - 对重复包，接收端 ** 再次发送上一次的 ACK**。

4.6.2 工作方式

- 接收端：
    - 如果收到 “期望的 seq# 且未损坏”：交付上层，并发送 “ACK(seq#)”；
    - 如果收到 “损坏的包或重复 seq#”：不交付上层，发送 “ACK(上一个正确包的 seq#)”。
- 发送端：
    - 在等待 ACK 时：
        - 收到 “ACK(当前包的 seq#)” → 认为已经被正确接收，切换到下一个 seq#；
        - 收到 “ACK(另一个 seq#)” 或校验错误 → 视为 NAK，重发当前包。

这样用一个 ACK 类型就实现了跟 rdt2.1 一样的功能。

4.7 rdt3.0：既有 bit errors 又可能丢包

4.7.1 新问题：丢包

- 底层信道不仅可能 bit error，还可能：
    - 丢数据包；
    - 丢 ACK；
    - ACK 延迟到达（“看起来像是丢了”）。

4.7.2 新增机制：定时器（timeout）

- 发送端在发送数据包同时启动一个 * 倒计时定时器 *：
    - 若在 timer 超时前收到期望的 ACK → 停止定时器，发送下一个数据；
    - 若超时仍未收到 ACK → 假定数据包或 ACK 丢失，* 重传当前数据包 * 并重启定时器。

4.7.3 仍然依赖序列号 & ACK

- 底层仍基于 rdt2.2 的思想：
    - 有 0/1 序列号；
    - 有 ACK(seq#)；
    - 接收端通过检查 seq# + checksum 来检测 “新包/旧包/损坏包”。

4.7.4 三种典型场景

1. * 无错误无丢包 *：
    - 发送方发包 → 收 ACK → 立即发下一个；
2. * 数据包丢失 *：
    - 发送方发数据 → 接收方啥也没收到；
    - 发送方等不到 ACK → timeout → 重发数据；
3. *ACK 丢失或延迟 *：
    - 接收方其实收到了数据并发送 ACK，但 ACK 在路上丢了 / 延迟；
    - 发送方 timeout 重发相同数据；
    - 接收方看到是 “重复包”（seq# 相同）：
        - 不再交付上层，仅重新发送 ACK。

4.7.5 rdt3.0 的本质：stop-and-wait ARQ

- 这是一个典型的 **停止-等待**自动重传协议（Stop-and-Wait ARQ）：
    - 任意时刻最多只有 1 个未确认的数据包在飞；
    - 可靠性靠：
        - 校验和（detect errors）；
        - 序列号（detect duplicates）；
        - ACK / 重传（recover from loss / errors）；
        - 定时器（detect loss）。

4.8 性能分析：利用率为什么“惨不忍睹”?

4.8.1 链路利用率 Usender

- 定义：发送方真正“在传数据”的时间占总时间的比例。

假设：

- 链路带宽：1 Gbps；
- 单向传播时延：15 ms → RTT ≈ 30 ms；
- 每个数据包：8,000 bits（约 1 KB）；

传输一个包的发送时间：

```
D_trans = L / R = 8000 bits / (10^9 bits/s) = 8 μs
```

一轮“发包 + 等 ACK”的时间：

```
T_round = RTT + D_trans ≈ 30 ms + 0.008 ms ≈ 30.008 ms
```

利用率：

```
U_sender = D_trans / T_round ≈ 0.008 ms / 30.008 ms ≈ 2.7 × 10^-4
```

也就是：** 只有 0.027% 的时间在真正传输数据，剩下 99.973% 都在空等 ACK**。

4.8.2 结论 & 预告

- rdt3.0 在“长 RTT + 高速链路 + 小包”的情况下效率极低；
- 根本原因：*stop-and-wait* 模式下管道里“同时在路上的数据太少”；
- 下一步自然要引入：
  - *pipeline*（流水线）：一次发多个未确认的数据包；
  - 例如 GBN（Go-Back-N）、Selective Repeat 等协议。

Question（讲义结尾提出的问题）：

如何提高利用率？ —— 答案：使用 pipelined reliable data transfer 协议。