CAN201 – Week 5: Transport Layer (2)

Pipelined reliable transport & TCP

InubashiriLix (Github)

0. Roadmap

- 本讲两大块：
  1. *Pipelined communication*：在 rdt3.0 基础上，引入流水线 ARQ 协议
     - Go-Back-N（GBN）
     - Selective Repeat（SR）
  2. *TCP: connection-oriented transport*：面向连接的传输层协议
     - 段结构、序号与 ACK
     - RTT 估计与自适应超时
     - 可靠传输、快速重传
     - 流量控制（flow control）与窗口扩展
     - 连接建立（三次握手）与关闭（四次挥手）

1. Motivation: 从 rdt3.0 到 pipelining

1.1 rdt3.0 = Stop-and-Wait ARQ 的性能问题

- rdt3.0 的关键特征：
  - 任何时刻 **最多只有一个**未确认的报文在 “飞”（in-flight）。
  - 每发完一个报文（seq=0/1）后，发送方必须 * 停止 *，等待 ACK 或超时。
- 链路利用率（sender utilization）：
  - 定义：发送方真正“在链路上发数据”的时间占总时间的比例。
  - 对于 stop-and-wait：
    - 每轮只发一个长度为 L 的报文，发送时间 =L/R=；
    - 然后等待约一个 RTT（往返时延）才能确认再发下一包。
    - 利用率：
      U_sender = (L/R) / (RTT + L/R)
- 课件例子：
  - L = 8000 bits, R = 1 Gbps → L/R = 8 μs；
  - RTT ≈ 30 ms；
  - 则：
    U_sender ≈ 0.008 ms / (30.008 ms) ≈ 2.7 × 10^-4 ≈ 0.027%
  - 99.97% 时间在 “干等 ACK”，几乎没有利用高带宽链路。

1.2 如何提升利用率：流水线（pipelining）

- * 思想 *：不要“发一包等一包”，而是允许发送方同时发出多个未确认报文。
- 关键变化：
  1. 发送方允许多个 in-flight 报文：
     - 相当于在链路上 “排队” 的报文数量增加；
     - 管道中同时装入更多比特，提高带宽利用率。
  2. 必须增大 * 序列号空间 *：
     - 需要能区分多个尚未确认的报文；
     - k bit 序号 → 2^k 个不同序号。
  3. 发送端和接收端都需要 * 缓存 *：
     - 发送端缓存未确认报文，用于重传；
     - 接收端可能需要缓存 “乱序到达” 的报文用于 SR。

1.3 利用率的提升（N 包流水线）

- 若允许最多 N 个报文同时在飞：
  - 一轮中可以连续发送 N 个报文，总发送时间 ≈ N·L/R；
  - 然后等待一个 RTT 以收到这些报文的 ACK。
- 利用率近似为：

  U_sender ≈ (N · L/R) / (RTT + N · L/R)
- PPT 示例：N = 3 时，利用率提高约 3 倍：
  - U_sender(3) = (3L/R) / (RTT + 3L/R)
    ≈ 0.024 ms / (30.024 ms)
    ≈ 0.0008 (> 0.00027)

2. Pipelined ARQ: Go-Back-N vs Selective Repeat

2.1 两种典型流水线重传协议

- 在流水线场景下，需要对 “丢包 / 出错 / 乱序” 时如何重传作出策略设计。
- 两种经典方法：

协议	发送端窗口（最多 N 未 ACK）	接收端 ACK 方式	接收端对乱序包处理	定时器策略	重
Go-Back-N	有，大小为 N	* 累计 ACK*（cumulative ACK）	通常丢弃或可选缓存	1 个：最老未 ACK 报文的计时器	超
SelectiveRpt	有，大小为 N	各报文单独 ACK（per-pkt ACK）	必须缓存	概念上每个未 ACK 报文一个	超

- 都是基于流水线、序号、ACK/重复 ACK 的 ARQ 方案。

3. Go-Back-N（GBN）协议

3.1 发送端：滑动窗口（sliding window）

- 发送端维护：
  - 窗口大小 = N；
  - 两个关键指针：
    - send_base：当前窗口中最早未被 ACK 的报文序号；
    - next_seq_num：下一个可发送报文的序号。
- 规则：
  - 若 =$next_{seqnum}$ < $send_{base}$ + N=：
    - 可以发送新的报文，seq = $next_{seqnum}$；
    - $next_{seqnum}$++；
    - 若此时没有计时器在跑，为 $send_{base}$ 报文启动定时器。
  - 收到一个累计 ACK(n)：
    - 意味着“序号 ≤ n 的报文都已正确收到”；
    - 更新 =$send_{base}$ = n+1=，窗口整体前移；
    - 若仍有未 ACK 报文 → 重启计时器（针对新的 $send_{base}$）。
- 定时器：
  - 只为“最老未确认报文”维护一个计时器；
  - 若计时器超时（超时对应 seq = $send_{base}$）：
    - * 重发所有窗口内未 ACK 的报文 *（从 $send_{base}$ 到 $next_{seqnum}$-1）；
    - 重启计时器。

3.2 接收端：累计 ACK 与乱序处理

- 接收端维护变量 =rcv_{base}=：当前“期望的下一个 in-order 序号”。
- 对接收端行为的一个简单实现（课件思路）：
  - 收到 **按序**报文（seq == rcv_{base} 且无误）：
    - 将报文交付上层；
    - rcv_{base}++；
    - 连续检查缓存中是否有 rcv_{base} 后续序号（如果实现了缓存）；
    - **发送 ACK(r 最后一个 in-order 序号)** —— 即累积 ACK。
  - 收到 **乱序**报文（seq > rcv_{base}，或有缺口）：
    - 可以选择：
      - * 丢弃 *：仅重新发送 ACK(rcv_{base}-1)；
      - 或者 * 缓存 *：等待缺失报文到达，再一次性交付。
    - 标准 GBN 教科书版本通常“不缓存”，直接丢弃。
  - 行为特点：
    - 始终 ACK 当前最高连续收到的序号；
    - 可能产生重复 ACK（duplicate ACK）：缺口存在时，每收一个乱序包都重发同一 ACK。
- 接收端只需记住 =rcv_{base}=（若不缓存）即可，状态简单。

3.3 GBN 示意时间线（配合 PPT）

- 示例：窗口大小 N=4，发送 0,1,2,3,4,5…，其中报文 2 在途中丢失：
  - 0、1 被正确收到并 ACK；
  - 3、4、5 先到达，因为 2 未到，接收端要么丢弃它们、要么缓存但 ACK 仍只到 1；
  - 发送端迟迟收不到 ACK≥2，等待计时器为报文 2 超时；
  - 超时后，发送端重发 2,3,4,5；
  - 接收端按序接收 2、3、4、5，逐个交付并 ACK。
- 代价：丢一个包会导致整个窗口之后的报文都被重新发送（即使其中很多已经成功到达）。

4. Selective Repeat（SR）协议

4.1 基本思想

- * 目标 *：避免 GBN 中“一个包丢了，窗口内后面所有包都白发一遍”的浪费。
- 关键点：
  - 接收端对每个正确收到的报文 * 单独 ACK*；
  - 接收端 * 必须缓存乱序报文 *，以便恢复 in-order 交付；
  - 发送端每个未确认报文有（概念上）一个单独计时器，超时报文单独重传；
  - 双方都维护一个大小为 N 的 * 窗口 *，限制可用序号范围。

4.2 发送端：SR window

- 维护：
  - =$send_{base}$=：当前发送窗口的起点；
  - =$next_{seqnum}$=：下一个可用序号；
  - 窗口范围：[$send_{base}$, $send_{base}$ + N - 1]。
- 行为：
  - 当上层来数据，若 $next_{seqnum}$ 落在窗口内：
    - 发送 seq=$next_{seqnum}$ 的报文；
    - 启动该报文的计时器；
    - $next_{seqnum}$++。
  - 收到 ACK(n)：
    - 若 n 在当前发送窗口内：
      - 标记报文 n 已确认；
      - 若 n 是当前窗口内最小未确认序号：
        - 将 $send_{base}$ 前移到下一个未确认报文的序号；
        - （窗口可能连续前移多个位置）。
  - 定时器超时 timeout(n)：
    - 只重传报文 n；
    - 重启报文 n 的计时器。

4.3 接收端：SR window

- 接收端维护：
  - =rcv_{base}=：当前期望的最低未到序号；
  - 接收窗口范围：[rcv_{base}, rcv_{base} + N - 1]。
- 行为：
  - 收到 seq=n 的报文时：
    - 若 n ∈ [rcv_{base}, rcv_{base}+N-1]（当前窗口内）：
      - 若报文未损坏且尚未接收：
        - 缓存该报文；
        - 发送 ACK(n)；
      - 若 n == rcv_{base}：
        - 此报文可立即交付上层；
        - 同时检查缓存中 rcv_{base}+1, rcv_{base}+2 … 是否都已收到，若有，则连续交付；
        - rcv_{base} 更新为“下一个尚未收到的序号”，窗口整体前移。
    - 若 n ∈ [rcv_{base}-N, rcv_{base}-1]（已经从旧窗口滑出，是“旧报文”）：
      - 不再交付上层；
      - 但必须重新发送 ACK(n)(防止发送端以为 ACK 丢了)。
    - 若 n 落在上述两范围之外（太新、超出窗口）：
      - 直接丢弃，不 ACK。

4.4 SR 示例流程（对比 GBN）

- 与之前 GBN 示例类似：报文 2 丢失，0,1,3,4,5 到达：
  - 接收端 SR 策略：
    - 收到 0 → 交付、ACK(0)，rcv_{base}=1；

        - 收到 1 → 交付、ACK(1)，rcv_{base}=2；
        - 收到 3 → 缓存、ACK(3)（rcv_{base}=2 仍未移动）；
        - 收到 4 → 缓存、ACK(4)；
        - 收到 5 → 缓存、ACK(5)；
      - 发送端只在 2 超时后重发报文 2；
      - 接收端收到 2：
        - 交付 2；
        - 同时把缓存的 3、4、5 一次性交付；
        - 发送 ACK(2)。
- 问题：当 ACK(2) 最后到达发送端时，窗口如何前移？
  - $send_{base}$ 会前移到“下一个未 ACK 的报文”，若 3,4,5 早已 ACK，则可一次前移至 6。

4.5 SR 的“困境”：序号空间 vs 窗口大小

4.5.1 场景描述（base-4，窗口大小 3）

- 序号空间：0,1,2,3（mod 4）。
- 发送窗口大小 N = 3。
- 通过 PPT 场景 (a)/(b) 可以看到：
  - 若窗口大小过大，当旧报文重传到达时，接收端可能把它**误认为**新窗口中的报文。
  - 关键原因：
    - 序号是循环使用的；
    - 接收端无法“看到”发送端窗口位置，只能通过 seq# 判断是否在接收窗口内；
    - 在某些情况下，同一个 seq# 对应的“旧窗口位置”和“新窗口位置”都落在接收窗口范围里，从而出现二义性。

4.5.2 避免歧义的约束

- 为避免上述歧义，经典结论：
  - 对 k bit 序号（序号空间大小 = 2^k）：
    - SR 协议的窗口大小必须满足：
      N ≤ 2^(k-1)
    - 即窗口大小最多为序号空间的一半。
- 直观理由：
  - 任意时刻，发送端窗口中最多有 N 个“活跃报文序号”；
  - 同时接收端窗口中也最多有 N 个“可接受新报文序号”；
  - 如果 N > $2^{(k-1)}$，则发送端“旧窗口”和“新窗口”之间会出现重叠，导致接收端无法区分“旧包重传”与“新包第一次到达”。

5. TCP 概览：connection-oriented transport

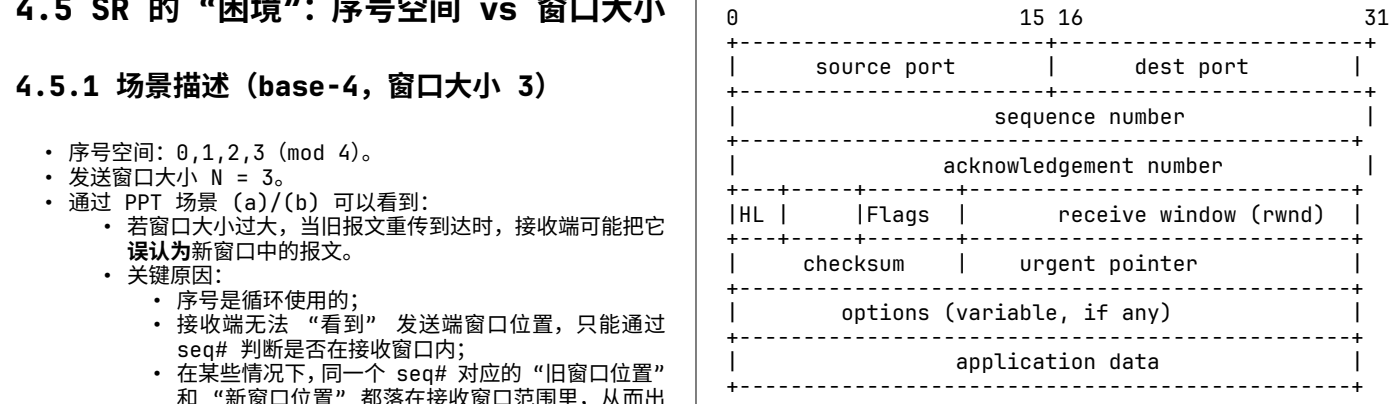
5.1 服务模型与特性

- TCP 是 Internet 中的主力传输协议，其关键特性：
  1. *Point-to-point*：
     - 每条连接只有一个发送端和一个接收端（逻辑上；方向可双向）。
  2. * 可靠、有序、无边界的字节流 *(reliable, in-order byte stream)：
     - TCP 向应用提供的是“连续的字节流”，不保留“报文边界”。
  3. *Pipelined*：
     - 内部使用窗口机制（结合流量控制与拥塞控制）实现流水线数据传输。
  4. *Full duplex*：
     - 双向数据流在同一条连接上同时存在；
     - 双方既是发送方又是接收方。
  5. *Connection-oriented*：
     - 连接建立前需三次握手（3-way handshake）初始化状态；
     - 连接关闭时有明确的终止过程。
  6. *Flow-controlled*：
     - 通过接收窗口（receive window，rwnd）避免淹没接收方缓冲区。
  7. *Congestion-controlled*（后续讲）：避免过度占用网络资源。

5.2 TCP segment 结构 vs UDP

5.2.1 TCP 头部字段概览

- TCP 段 = TCP header + data（payload）：



  - 关键字段说明：
    - *source port, dest port*：端口号，用于 (de)mux；
    - *sequence number*：
      - 此段数据中第 **一个字节**在 TCP 字节流中的序号；
      - 注意：TCP * 按字节编号 *，不是按段编号。
    - *acknowledgement number*：
      - ACK 位（flag A）为 1 时有效；
      - 表示“期望从对端收到的下一个字节序号”；
      - 因此是 * 累计 ACK*。
    - *HL（header length）*（即 Data Offset）：TCP 头部长度，以 4B 为单位。
    - *Flags*：
      - URG, ACK, PSH, RST, SYN, FIN 等：
        - SYN：连接建立；
        - FIN：连接关闭；
        - RST：异常复位；
        - ACK：ACK 字段有效；
        - URG, PSH：紧急数据/“推送”数据（实践中较少用）。
    - *receive window（rwnd）*：
      - 接收方通告的“还能再接收多少字节”，用于流量控制。
    - *checksum*：
      - 采用与 UDP 相同的 Internet checksum。
    - *urgent pointer*：
      - 与 URG 标志配合，标记紧急数据结束位置（很少使用）。
    - *options*：
      - 如 MSS、窗口扩大（window scale）、时间戳等。

5.2.2 与 UDP 报文对比

- UDP 仅有 8B 头：
  - src port, dest port, length, checksum。
- TCP 因提供更多服务（可靠、流控、拥塞控制等），头部显著更复杂。

6. TCP 序号与 ACK 语义

6.1 字节流编号

- TCP 把应用数据视作连续字节流，对 **每个字节**赋予一个序号。

- 一个 TCP 段包含若干字节：
  - 段头中的 **sequence number** = 此段 **第一个字节**的序号。
- ACK 是 **累计**形式：
  - ack = k 表示“已经正确收到所有字节 < k，下一字节期望为 k”。

6.2 发送窗口中的四个区域（概念）

- 对发送方来说，可以将序号空间分为四段（如 PPT 图）：
  - 已发送且已 ACK；
  - 已发送但尚未 ACK（in-flight，窗口内数据）；
  - 未发送但在窗口内（可立即发送）；
  - 不在窗口内（暂时不可用）。
- 窗口大小由：
  - 流量控制（rwnd）；
  - 拥塞控制（cwnd，后续讲）
  - 共同决定，实际可发送数据量为：min(rwnd, cwnd)。

6.3 Telnet 示例（回显单个字符）

- 情形：
  - 用户在 A 上通过 Telnet 向 B 输入单个字符’C’；
  - TCP 的传输过程：
    1. A→B：包含’C’ 的数据段，seq=42，ACK=79；
    2. B 收到后回显’C’ 给 A，对 A 的数据做 ACK：
       - B→A：seq=79，ACK=43，data=’C’；
    3. A 收到回显后，可能再发送下一个序号的字符。
- 重点：
  - ACK 号总是“下一个期望字节”的序号；
  - seq 号按字节流递增，与应用层消息边界无直接关系。

7. TCP RTT 估计与自适应超时

7.1 为什么要估计 RTT?

- TCP 重传基于 * 超时 *：
  - 超时时间太短：
    - 会频繁产生 “误判丢包” 的超时，从而不必要地重传；
  - 超时时间太长：
    - 真实丢包时恢复缓慢，连接性能下降。
- 因此需要一个对实时 RTT 的平滑估计，使超时既不过短也不过长。

7.2 SampleRTT 与 EstimatedRTT

- 对于每个 TCP 段（未重传过的），发送端可以测量：
  - SampleRTT = 从段发送到收到对应 ACK 所用的时间。
- 由于 SampleRTT 波动较大，TCP 使用 * 指数加权移动平均 *（EWMA）：

```
EstimatedRTT_n = (1 - α) * EstimatedRTT_{n-1}
                 + α * SampleRTT_n
```

  - α 为平滑因子，典型取值 α = 0.125；
  - 最近一次样本权重为 α，历史样本权重随时间指数衰减。

7.3 RTT 偏差 DevRTT 与 TimeoutInterval

- 为了设置 “安全裕量”，需要估计 SampleRTT 与 EstimatedRTT 的偏差：

```
DevRTT_n = (1 - β) * DevRTT_{n-1}
         + β * |SampleRTT_n - EstimatedRTT_n|
```

  - β 通常取 0.25。
- 超时间隔设置为：

```
TimeoutInterval = EstimatedRTT + 4 * DevRTT
```

  - 当 RTT 波动大（DevRTT 大）时，TimeoutInterval 自动增加，避免误报；
  - 当 RTT 波动小且稳定时，TimeoutInterval 更接近 EstimatedRTT，反应更快。

8. TCP 可靠数据传输机制（rdt over IP）

8.1 相对于通用 rdt 的特点

- 建立在不可靠的 IP 之上，TCP 实现 rdt 时采用：
  - Pipelined segments：使用窗口机制；
  - Cumulative ACKs：累计确认；
  - 单一重传计时器（针对最早未 ACK 报文）；
  - 重传触发事件：
    - timeout；
    - duplicate ACKs（快速重传）。

8.2 简化 TCP 发送端逻辑（先忽略拥塞/流量控制）

- 事件 1：上层交付数据
  1. 将字节写入发送缓存；
  2. 若窗口允许，构造一个或多个报文段：
     - 填写 seq #（对应首字节序号）；
     - 发送；
  3. 若计时器尚未运行，为最早未 ACK 的报文启动定时器。
- 事件 2：收到 ACK
  1. 若 ACK 号 > $send_{base}$：
     - 更新 $send_{base}$（最早未 ACK 序号）；
     - 将已被 ACK 的字节从发送缓冲区移除；
  2. 若仍有未 ACK 报文：
     - 重启计时器（针对新的 $send_{base}$）；
  3. 否则：
     - 停止计时器。
- 事件 3：定时器超时
  1. 重传“最早未 ACK 报文”（$send_{base}$ 对应段）；
  2. 重启计时器。

8.3 TCP 接收端 ACK 生成规则（RFC 5681）

- 按课件表格整理（假设不考虑 SACK）：

事件（接收端）	动作（ACK 策略）
收到 **按序**段（恰好为期望 seq），且之前没有挂起的 ACK	延迟 ACK：最多等待 500ms 以期望下一个段；若无新段到达，再发送 ACK。
收到 **按序**段，且已有一个延迟 ACK 挂起	立即发送一个累计 ACK，确认两个已按序段。
收到 **乱序**段（seq 大于期望值，出现 gap）	立即发送一个重复 ACK，ACK 期望的下一个字节（gap 起点），提示有 segment 缺失。
收到一个段，恰好填补了缺口的全部或一部分	若该段起始位置是 gap 的低端，则立即发送 ACK（此时可能连续交付多段数据）。

- 这些规则共同作用：
  - 正常情况下减少 ACK 数量（延迟 ACK）；
  - 丢失或乱序时快速产生重复 ACK，为“快速重传”提供信号。

9. TCP 重传与快速重传（fast retransmit）

9.1 超时重传场景

- 场景 1：ACK 丢失
  - A 发送 seq=92(8B 数据)给 B,B 收到并发送 ACK=100；
  - ACK 在途中丢失，A 超时后重发 seq=92；
  - B 收到重复数据：
    - 识别为已收到的字节范围；
    - 不重复交付上层，只再次发送 ACK=100；
  - A 收到 ACK=100 后前移 $send_{base}$，继续发送后续数据。
- 场景 2：数据段丢失
  - A：92→8B，100→20B 等；
  - 若 100 的段丢失，B 对 92 段 ACK=100；
  - A 若超时重传 100 段，B 收到后 ACK=120；
  - 通过 **累计 ACK**，“覆盖”了早前未送达的 ACK。

9.2 快速重传（Fast Retransmit）

- 仅靠超时重传会带来较大延迟：
  - 若某个段丢失，而后面的段仍成功到达：
    - 接收端对每个后续段发送重复 ACK（同一个 ACK 号）；
  - 在超时时间到达前,发送端已经观察到连续多个重复 ACK。
- *Fast retransmit 策略 *：
  - 当发送端收到 * 三个额外的重复 ACK*（即总共 4 个同 ACK 值）时：
    - 认为“ACK 所指的下一个字节对应的段很可能丢失”；
    - 立即重传对应最小未 ACK 段，而 * 不必等到超时 *。
- 效果：
  - 显著加快丢包恢复速度；
  - 减少对 RTT 估计与 timeout 设置的依赖。

10. TCP 流量控制（flow control）

10.1 问题：接收缓存溢出

- TCP 接收端在 OS 内维护 * 接收缓存 *（RcvBuffer）：
  - IP 层把到达的 TCP payload 写入这一缓存；
  - 应用进程从 socket 中读出数据（消耗）。
- 若网络层到达速率 > 应用读取速率：
  - 缓存可能被填满；
  - 新到数据只能丢弃或导致错误。
- * 目标 *：让发送方不要超出接收方的“消化能力”。

10.2 接收窗口 rwnd 的机制

- 接收端通过 TCP 头部字段 =rwnd=（receive window）向发送端通告自身剩余缓冲空间：
  - RcvBuffer：接收缓存总大小（可通过 socket 选项配置或内核自动调整）；
  - buffered data：当前已占用的字节。
  - rwnd = RcvBuffer - buffered data。
- * 发送端约束 *：
  - 保证：
    LastByteSent - LastByteAcked ⩽ rwnd
  - 即飞行中的未确认数据量不超过接收方宣告的 rwnd。
- 效果：
  - 确保接收端缓存不会被溢出；
  - 是端到端的 * 流量控制 *，与网络拥塞无关（这是 congestion control 的范畴）。

10.3 窗口扩展（Window Scaling）

- TCP 头部中的 rwnd 字段只有 16 bit：
  - 最大值 = 65535 字节 ≈ 64 KB；
  - 在高速长 RTT 网络（“长肥管道”）中远远不够。
- 通过 *Window Scale Option*：
  - 在连接建立时（SYN 包中的 option）协商一个扩展因子 WS；
  - 实际窗口大小 = rwnd * 2^{WS}；
  - 例如：
    - rwnd = 6379，WS=6 → 实际窗口 ≈ 6379 × 64 ≈ 408,256 字节。
- 这样既兼容旧协议（保持 16 bit 字段），又能支持更大窗口。

11. TCP 连接管理：建立与关闭

11.1 “Handshake”的目的

- 在交换数据之前，双方需要：
  1. 确认对方 **在线且愿意**建立连接；
  2. 交换初始参数：
     - 双方初始序号（Initial Sequence Number, ISN）；
     - 窗口大小、选项（如 MSS、窗口扩展等）。
- 若只采用简单两次握手（two-way handshake），会因：
  - 报文重传、重排序、延迟；
  - 而导致“半开连接”或“旧连接报文被当成新连接的一部分”等问题。

11.2 两次握手的典型问题

- 场景 1：半开连接（half-open connection）
  - 客户端发送连接请求 $req_{conn}(x)$ 后崩溃；
  - 服务器重传 $req_{conn}(x)$ 的响应或迟到版本；
  - 最终服务器认为连接已 ESTAB，但客户端已经不存在。
- 场景 2：旧连接数据被新连接误收
  - 旧连接中某个数据段 data(x+1) 延迟在网络中；
  - 新连接恰好使用相同 ISN x；
  - data(x+1) 抵达时，服务器认为这是新连接的数据并接受。
- 本质问题：
  - 由于传输时延和重传机制，两个“逻辑上不同的连接”在报文层面不易区分。

11.3 三次握手（TCP 3-way handshake）

- 三次握手过程（简化）：

```
1. 客户端 -> 服务器: SYN, seq = x
   - 客户端选择自己的初始序号 x
   - 状态: CLOSED → SYN-SENT

2. 服务器 -> 客户端: SYN-ACK, seq = y, ack = x+1
   - 服务器选择自己的初始序号 y
   - 用 ACK 确认收到了客户端的 SYN(x)
   - 状态: LISTEN → SYN-RCVD

3. 客户端 -> 服务器: ACK, seq = x+1, ack = y+1
   - 客户端确认服务器的 SYN(y)
   - 双方均进入 ESTABLISHED 状态
   - 该 ACK 段可以携带数据（客户端到服务器的首批数据）
```

- 三次握手的效果：
  - 双方都显式确认了对方的 ISN；
  - 延迟/重传的旧报文不容易被误认成新连接的一部分；
  - 避免了两次握手下的“半开连接”和“旧数据注入”问题。

11.4 连接关闭：四次挥手（four-segment close）

- TCP 是 * 双向字节流 *，因此连接关闭本质上是双方各自关闭一个方向：
  - 任一端都可以主动发起关闭流程。
- 标准关闭流程（假设客户端先发起）：

1. 客户端：发送 FIN，seq = x；
   - 状态：ESTAB → FIN_{WAIT1}；
   - 表示“客户端不再发送数据，但仍可接收数据”。
2. 服务器：收到 FIN，发送 ACK，ack = x+1；
   - 状态：ESTAB → $CLOSE_{WAIT}$；
   - 通知应用“对端已关闭发送方向”，本端仍可继续发送数据。
3. 服务器：当应用结束发送后，发送自己的 FIN，seq = y；
   - 状态：$CLOSE_{WAIT}$ → $LAST_{ACK}$。
4. 客户端：收到 FIN(y)，发送 ACK，ack = y+1；
   - 状态：FIN_{WAIT1}/2 → $TIME_{WAIT}$ → 经过 2MSL 后 → CLOSED。
   - 2MSL（两倍最大报文生存时间）用于：
     - 确保迟到的 FIN/ACK 不会残留在网络中；
     - 确保对端若未收到最后 ACK，重传 FIN 时，本端仍有状态可做正确回应。

- 服务器在收到 ACK 之后：$LAST_{ACK}$ → CLOSED。

11.5 小结：三次握手 vs 四次挥手

- 建立连接：
  - 需要三次握手 → 同时完成：
    - 双方 ISN 交换；
    - 双向通信状态建立；
    - 避免旧报文干扰。
- 关闭连接：
  - 是双向独立的“关闭发送方向”：
    - 所以需要双方各自发送一个 FIN，并对对方 FIN 发 ACK；
    - 在一般情况下表现为“四次挥手”。

12. 全讲小结：从 rdt3.0 到 TCP

- rdt3.0（stop-and-wait）给出了可靠传输的最基本思想，但效率极低。
- 引入流水线（pipelining）后：
  - 通过 GBN/SR 扩展到多包并行，但需要更大序号空间与复杂的窗口管理。
- TCP 在真实 Internet 上实现可靠传输：
  - 使用类似 SR/GBN 的思想（累计 ACK + 重传 + 窗口）；
  - 结合 RTT 估计与自适应超时、自适应 ACK 策略与快速重传；
  - 通过流量控制（rwnd）保证接收端不被淹没；
  - 通过三次握手、四次挥手管理连接生命周期。
- 后续课程在此基础上进一步引入：
  - 拥塞控制算法（慢启动、拥塞避免、快恢复等），
  - 完成对 TCP 行为的整体刻画。