日期：24.11.5
姓名：谢志康
学号：22307110187
计算机网络：自顶向下方法（原书第八版） 中文版
第三章：传输层作业

P3. UDP 和 TCP 使用反码来计算检验和。假设你有下面 3 个 8 比特字节：01010011，01100110，01110100。这些 8 比特字节和的反码是多少？（注意到尽管 UDP 和 TCP 使用 16 比特的字来计算检验和，但对于这个问题，你应该考虑 8 比特和。）写出所有工作过程。UDP 为什么要用该和的反码，即为什么不直接使用该和呢？使用该反码方案，接收方如何检测出差错？1 比特的差错将可能检测不出来吗？2 比特的差错呢？

8 比特字节和：100101101 把第 9 位溢出回卷：00101110
反码：11010001
用反码原因：当接收方对所有接收到的数据（包括检验和）求和并取反时，如果没有出错，结果应该是全 1（即 `11111111`）。所以直接用反码算是保证和计算方式的对称性，便于接收方直接相加验证。
检测差错：接收方收到数据后，计算接收到的报文段的校验和，检查计算出的校验和是否等于校验和字段的值：如果不相等就检测到错误。即目标端：校验内容+校验和=全 1 则通过校验，否则没有通过。
1bit 差错是可以检测出来的，就是非全 1 的形式
但是 2bit 不一定能检测出来，例如两个 8 字节比特，最后两位：
正确情况：01　10
数据错误变异为：10　01
但是这俩 8bits 字节和是不变的，最后检测还是全 1。

P22. 考虑一个 GBN 协议，其发送方窗口为 4，序号范围为 1024。假设在时刻 t，接收方期待的下一个有序分组的序号是 k。假设媒介不会对报文重新排序。回答以下问题：
a. 在 t 时刻，发送方窗口内的报文序号可能是多少？论证你的回答。
b. 在 t 时刻，在当前传播回发送方的所有可能报文中，ACK 字段的所有可能值是多少？论证你的回答。

a. 接收方是按序确认的，发送方必须等到当前窗口内的分组都被确认才继续移动。所以接收方想要第 k 个分组表示 k 之前所有分组接收方一定已经接受并发送 ACK 了，但发送方不一定都接受到了 ACK，所以发送方窗口中左极限为：k-4（只要[k-4,k-1]的 ACK 信号全部到达，滑动窗口自然移到 k），右极限为：k+3（[k,k+3]已发送，但还没到达接收方）。所有情况如下所示：
   - (k-4,k-3,k-2,k-1) 发送方 ACK(k-4)~ACK(k-1)都没有收到；
   - (k-3,k-2,k-1,k) 发送方收到了 ACK(k-4)，ACK(k-3)~ACK(k-1)发送方还没有收到；
   - (k-2,k-1,k,k+1) 发送方收到了 ACK(k-3)，ACK(k-2)~ACK(k-1)都没有收到
   - (k-1,k,k+1,k+2) 发送方收到了 ACK(k-2)，ACK(k-1)没有收到
   - (k,k+1,k+2,k+3) 发送方收到了 ACK(k-1)，发送了 k，接收方还没收到 k
b. -(k-4,k-3,k-2,k-1)
   - (k-3,k-2,k-1)
   - (k-2,k-1)
   - (k-1)
   - ()

以上是最一般的情况，如果 |1024-k| < 4 那滑动窗口不会右移了，那发送方窗口内的报文序号一直是(1021,1022,1023,1024)。

P23. 考虑 GBN 协议和 SR 协议。假设序号空间的长度为 k，那么为了避免出现图 3-27 中的问题，对于这两种协议中的每一种，允许的发送方窗口最大为多少？

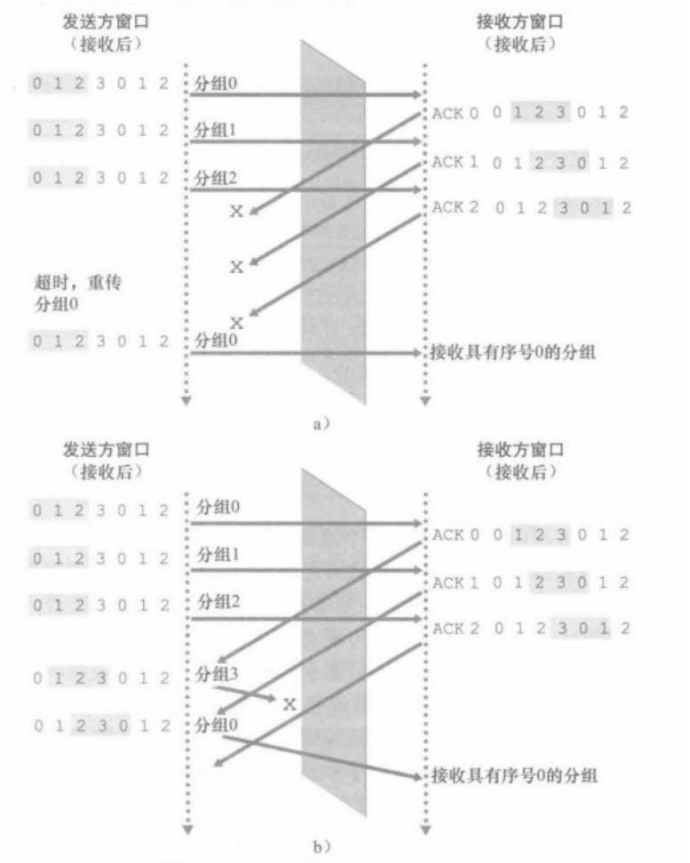

图 3-27 SR 接收方窗口太大的困境：是一个新分组还是一次重传

要避免让接受者窗口的最前端(也就是具有最高序列号的那个)与发送窗口的最尾端(发送窗口中的具有最低序列号的那个)交叠在同一个序列号空间中。最极限的情况就是差一个产生交叠，也就是接收方窗口的 min 刚好为发送方窗口的 max+1. 即序列号空间长度必须至少是窗口长度的两倍 k≥2w。所以发送方窗口最大为 k//2 （向下取整）

P26. 考虑从主机 A 向主机 B 传输 L 字节的大文件，假设 MSS 为 536 字节。

a. 为了使得 TCP 序号不至于用完，L 的最大值是多少？前面讲过 TCP 的序号字段为 4 字节。

b. 对于你在（a）中得到的 L，传输此文件要用多长时间？假定运输层、网络层和数据链路层首部总共为 66 字节，并加在每个报文段上，然后经 155Mbps 链路发送得到的分组。忽略流量控制和拥塞控制，使主机 A 能够一个接一个和连续不断地发送这些报文段。

a. TCP 序号字段为 4 字节（32 位），因此 TCP 序号的范围为：0 到 2^32 -1
L 的最大值就是 2^32，不能超过 TCP 序号范围了。

b. MSS = 536，总报文段数：L/536 = 8012999。
而一个报文段总大小 = 536 + 66 = 602 bytes
传输速率为 155 * 10^6 / 8 bytes/s 因此一个报文段时间为：31.071 * 10^-6
所以总时间约为 249s

P27. 主机 A 和 B 经一条 TCP 连接通信，并且主机 B 已经收到了来自 A 的最长为 126 字节的所有字节。假定主机 A 随后向主机 B 发送两个紧接着的报文段。第一个和第二个报文段分别包含了 80 字节和 40 字节的数据。在第一个报文段中，序号是 127，源端口号是 302，目的地端口号是 80。无论何时主机 B 接收到来自主机 A 的报文段，它都会发送确认。

a. 在从主机 A 发往 B 的第二个报文段中，序号、源端口号和目的端口号各是什么？

b. 如果第一个报文段在第二个报文段之前到达，在第一个到达报文段的确认中，确认号、源端口号和目的端口号各是什么？

c. 如果第二个报文段在第一个报文段之前到达，在第一个到达报文段的确认中，确认号是什么？

d. 假定由 A 发送的两个报文段按序到达 B。第一个确认丢失了而第二个确认在第一个超时间隔之后到达。画出时序图，显示这些报文段和发送的所有其他报文段和确认。（假设没有其他分组丢失。）对于图上每个报文段，标出序号和数据的字节数量；对于你增加的每个应答，标出确认号。

B 已收到 A 的最长 126 bytes 的所有 byte，随后 80bytes、40bytes

Id: 127　src_port: 302　des_port: 80

a. 上一个序号 127，80bytes，下一个紧接着上一个，也就是 127+80-1+1 = 207
源端口和目的端口不会变的。
所以：id: 207　src_port: 302　des_port: 80

b. 确认号：20 源端口号：80 目的端口号：302 （确认，ack 反着传回去）

c. 如果第二个报文段（序号为 207）先到达，由于第一个报文段（127-206）尚未到达，主机 B 会认为中间有缺失的数据。按照 TCP 的标准行为，主机 B 会发送一个重复的 ACK，要求序号 127 的数据。所以确认号是 127

d. 两报文段按顺序到达主机 B，第一个确认丢失，第二个确认在第一个超时后到达。
主机 A 发送第一个报文段（序号 127，80 字节）。
主机 B 接收该报文段，发送确认（确认号 207），此确认丢失。
主机 A 发送第二个报文段（序号 207，40 字节）。
主机 B 接收第二个报文段，并再次发送确认（确认号 247）以确认收到所有数据。
主机 A 接收第二次确认（确认号 247），数据传输完成。
时序图：

```
主机 A                          主机 B
------                          ------
发送: Seq=127, Len=80  ----->  接收: Seq=127, Len=80
                                发送: ACK=207  ----->  (确认丢失)
发送: Seq=207, Len=40  ----->  接收: Seq=207, Len=40
                                发送: ACK=247  ----->
接收: ACK=247
```

P29. 在 3.5.6 节中讨论了 SYN cookie。

a. 服务器在 SYNACK 中使用一个特殊的初始序号，这为什么是必要的？

b. 假定某攻击者得知一台目标主机使用了 SYN cookie。该攻击者能够通过直接向目标发送一个 ACK 分组创建半开或全开连接吗？为什么？

c. 假设某攻击者收集了由服务器发送的大量初始序号。该攻击者通过发送具有初始序号的 ACK，能够引起服务器产生许多全开连接吗？为什么？

a. 在使用 SYN cookie 时，服务器不为每一个进入的 SYN 请求分配资源来维护连接状态，而是通过计算得出一个特殊的初始序号，将客户端的连接信息（如源 IP、源端口、目的 IP、目的端口和时间戳等）编码在这个序号中。当客户端返回 ACK 时，服务器可以通过解析该序号来验证连接是否是合法请求。这样，服务器无需在接收到 SYN 时保存连接状

态，而仅在客户端返回 ACK 时检查序号是否合法。如果序号不合法，服务器将丢弃该连接，避免资源消耗。

b. 不能。服务器不会将这 ACK 报文视为有效连接的建立。因为这 ACK 报文的序不会 SYN cookie 相匹配。

c. 不能。服务器使用 SYN cookie 的初始序号是根据特定算法生成的，每次连接的初始序号是不同的。生成 SYN cookie 的过程通常包括客户端的连接信息和服务器的私钥或时间戳,攻击者无法预测或重现这些序号,所以无法成功地伪造合法 ACK 报文来打开连接。

P33. 在 3.5.3 节中，我们讨论了 TCP 的往返时间的估计。TCP 避免测量重传报文段的 SampleRTT，对此你有何看法？

细节上有需要注意的地方：比如要避免误导性的 RTT，重传报文段通常是因为先前的报文段在网络中丢失或延迟超时才触发的。此时，网络的拥塞程度或路径状态可能与原始传输时有所不同，这时候就重传的 RTT 可能偏大或偏小，不能代表正常的往返时间。不能纳入这部分去统计 SampleRTT。TCP 用平滑算法更新 EstimatedRTT。如果重传的这部分 RTT 被计算在内，可能会导致估计的不稳定，使得 RTO 时长忽高忽低。这种波动会引起过多的无效重传，进一步增加网络负载和拥塞，反而不利于连接的性能。

P36. 在 3.5.4 节中，我们看到 TCP 直到收到 3 个冗余 ACK 才执行快速重传。你对 TCP 设计者没有选择在收到对报文段的第一个冗余 ACK 后就快速重传有何看法？

合理。这其实是在平衡丢包检测的敏感度与网络负载之间的关系。因为只有 TCP 接收方收到了序号大于它本身期待序号的分组时，才会发送冗余的 ACK。在实际网络中，报文段可能因网速轻微抖动或延迟而暂时无序到达接收端。这样的无序到达会触发单个冗余 ACK，但这并不一定表示报文段丢失。试想如果 TCP 在接收到第一个冗余 ACK 时就立即重传，很可能会导致许多不必要的重传，反而大额增加了网络的负担。因此，等待 3 个冗余 ACK，可以将这些轻微的乱序现象与真正的丢包区分开来。连着 3 此就大概率是真的丢包了。

式来进行 ssthresh 设置吗？

P39. 考虑图 3-46b。如果 λ'_{in} 增加超过了 $R/2$，λ_{out} 能够增加超过 $R/3$ 吗？试解释之。现在考虑图 3-46c。假定一个分组从路由器到接收方平均转发两次，如果 λ'_{in} 增加超过 $R/2$，λ_{out} 能够增加超过 $R/4$ 吗？试解释之。

P40. 考虑图 3-61。假设 TCP Reno 是一个经历如上所示行为的协议，回答下列问题。在各种情况中，简要地论证你的回答。

a. 指出 TCP 慢启动运行时的时间间隔。

b. 指出 TCP 拥塞避免运行时的时间间隔。

c. 在第 16 个传输轮回之后，报文段的丢失是根据 3 个冗余 ACK 还是根据超时检测出来的？

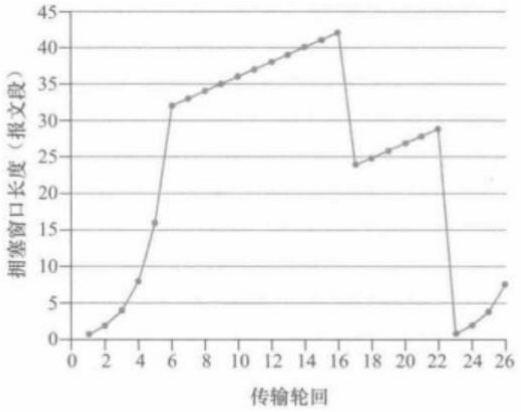

图 3-61　TCP 窗口长度作为时间的函数

d. 在第 22 个传输轮回之后，报文段的丢失是根据 3 个冗余 ACK 还是根据超时检测出来的？

e. 在第 1 个传输轮回里，ssthresh 的初始值设置为多少？

f. 在第 18 个传输轮回里，ssthresh 的值设置为多少？

g. 在第 24 个传输轮回里，ssthresh 的值设置为多少？

h. 在哪个传输轮回内发送第 70 个报文段？

i. 假定在第 26 个传输轮回后，通过收到 3 个冗余 ACK 检测出有分组丢失，拥塞的窗口长度和 ssthresh 的值应当是多少？

j. 假定使用 TCP Tahoe（而不是 TCP Reno），并假定在第 16 个传输轮回收到 3 个冗余 ACK。在第 19 个传输轮回，ssthresh 和拥塞窗口长度是什么？

k. 再次假设使用 TCP Tahoe，在第 22 个传输轮回有一个超时事件。从第 17 个传输轮回到第 22 个传输轮回（包括这两个传输轮回），一共发送了多少分组？

a. 1-6　23-26

b. 6-16　17-22

c. 3 个冗余 ACK 重复确认

d. 超时

e. 2

f. 24 （那一段上升最前面最低的那个值）

g. 14

h. 7 （第六周期传过去大约 33 个）

i. ssthresh 是 4，窗口长度是 7

j. 拥塞窗口长度为 4，ssthresh 为 21

k. 1+2+4+8+16+21（不会超过 ssthresh） = 52 个

P42. 在 3.5.4 节中，我们讨论了在发生超时事件后将超时间隔加倍。为什么除了这种加倍超时间隔机制外，TCP 还需要基于窗口的拥塞控制机制（如在 3.7 节中学习的那种机制）呢？

能做到更加高效的流量管理和拥塞控制，加倍超时算是被动的，而基于窗口调整是主动行为。他们可以互补、应对不同的网络拥塞：超时间隔加倍主要是被动应对重传延迟，而窗口控制机制则是主动管理数据发送量。当网络检测到严重拥塞（如超时事件）时，TCP 先加倍超时间隔减少重传频率，并且通过减小窗口避免进一步加重拥塞。

P47. 回想 TCP 吞吐量的宏观描述。在连接速率从 W/(2RTT) 变化到 W/RTT 的周期内，只丢失了一个分组（在该周期的结束）。

a. 证明其丢包率（分组丢失的比率）等于：

$$L = \text{丢包率} = \frac{1}{\frac{3}{8}W^2 + \frac{3}{4}W}$$

b. 如果一条连接的丢包率为 L，使用上面的结果，则它的平均速率近似由下式给出：

$$\text{平均速率} \approx \frac{1.22\text{MSS}}{\text{RTT}\sqrt{L}}$$

a. 在一个周期内，拥塞窗口从 W/2 增加到 W，这是一个线性增长的过程。
在一个 RTT 时间内，窗口增加 1 个 MSS，因此窗口大小增长所需的 RTT 数是从 W/2 增加到 W 的时间，即 W/2 个 RTT。分组总数就是从 W/2 增加到 W 的窗口大小的平均值乘以所需的 RTT 数：
均值为(W/2+W)/2 = 3W/4，因此分组总数为 3W 方/8 + 3W/4。
在此周期结束时发生一次丢包，因此丢包率即为上式。

b. 周期内速率呈线性增长，因此平均速率为 3W/4RTT，W*MSS 是传输的字节速。
由 a，W2 远大于 W 把分母后项舍去，W 约等于 1.22/根号 L，因此最后算的上式。

P58. 在这个习题中，我们考虑由 TCP 慢启动阶段引入的时延。考虑一个客户和一个 Web 服务器直接连接到速率 R 的一条链路。假定该客户要取回一个对象，其长度正好等于 $15S$，其中 S 是最大段长度（MSS）。客户和服务器之间的往返时间表示为 RTT（假设为常数）。忽略协议首部，确定在下列情况下取回该对象的时间（包括 TCP 连接创建）：

a. $4S/R > S/R + \text{RTT} > 2S/R$
b. $S/R + \text{RTT} > 4S/R$
c. $S/R > \text{RTT}$

初始化连接阶段从客户端视角来看，可以清晰地理解整个连接过程。初始化连接需要经历两个 RTT：第一个 RTT：客户端发送请求报文（报文①），服务器响应连接请求（报文②）。第二个 RTT：客户端接收到服务器的确认报文后，回复服务器（报文③），服务器再确认客户端的回复（报文④）。

因此，从客户端发出请求（报文①）到接收到服务器的最终确认（报文④）之间，总共需要 2 × RTT。这里假设是客户端请求服务器的数据，因此，服务器需要在接收到（报文③）之后才会开始传输数据包。

在服务器确认连接建立后，便可以开始将数据传送到网络中，传输的延时是 S/RS/RS/R。在慢启动阶段，服务器最初仅能发送一个分组，必须等待客户端对该分组的确认才能继续。以客户端视角来看，从收到服务器的最终确认报文（报文④）到收到第一个数据分组的确认，耗时为 S/R+RTTS/R + RTTS/R+RTT。因此，从客户端最初发送请求（即报文①）

到此刻，整体耗时为 3RTT+S/R3 \, RTT + S/R3RTT+S/R。

之后，服务器便能发送两个分组。由于 RTT>S/RRTT > S/RRTT>S/R，服务器发送第三个分组时，第二个分组的确认还未收到，因此服务器必须等待第二个分组的确认，而非第三个分组的传输完成。这表明服务器是被 ACK 确认的返回时间所限制，而非传输速度。
在慢启动过程中，服务器每收到一个 ACK，窗口便会扩展 1。每个成功的 ACK 会推动窗口移动，增加两个空位，允许继续发送新的分组。
从客户端视角，客户端在收到第一个分组的确认到接收到第三个分组之间，耗时 S/R+RTTS/R + RTTS/R+RTT。
接着，服务器可以依次发送 4 个、8 个分组。以窗口大小为 4 为例，发送第一个分组耗时 S/RS/RS/R，而第一个分组的确认往返则耗时 RTTRTTRTT。由于 S/R+RTT<4S/RS/R + RTT < 4S/RS/R+RTT<4S/R，在第一个分组往返的过程中，其余分组还在传输中。同时，第一个分组的返回确认会增加窗口并推动窗口移动，因此服务器可以发送下一个分组。
从客户端的视角，从收到第三个分组到接收完所有分组，耗时为 12×S/R12 \times S/R12×S/R。总耗时为：
2×RTT+(S/R+RTT)+(S/R+RTT)+12×S/R=4×RTT+14×S/R2 \times RTT + (S/R + RTT) + (S/R + RTT) + 12 \times S/R = 4 \times RTT + 14 \times S/R2×RTT+(S/R+RTT)+(S/R+RTT)+12×S/R=4×RTT+14×S/R

同理，若 S/R+RTT>4S/RS/R + RTT > 4S/RS/R+RTT>4S/R，则在窗口大小为 4 时，第一个分组的确认还未返回，因此服务器需等待确认返回，受 RTT 限制。窗口大小增至 8 时，限制消失。总耗时则为：
2×RTT+(S/R+RTT)+(S/R+RTT)+(S/R+RTT)+8×S/R=5×RTT+11×S/R2 \times RTT + (S/R + RTT) + (S/R + RTT) + (S/R + RTT) + 8 \times S/R = 5 \times RTT + 11 \times S/R2×RTT+(S/R+RTT)+(S/R+RTT)+(S/R+RTT)+8×S/R=5×RTT+11×S/R

同理，从第二次开始每次发送均不受 RTT 限制。总耗时为：
2×RTT+(S/R+RTT)+(2+4+8)×S/R=3×RTT+15×S/R2 \times RTT + (S/R + RTT) + (2 + 4 + 8) \times S/R = 3 \times RTT + 15 \times S/R2×RTT+(S/R+RTT)+(2+4+8)×S/R=3×RTT+15×S/R