当 A 发送一条消息时，它会连续使用相同的序列号重新发送一个消息，直到收到来自 B 的包含相同序列号的确认。发生这种情况时，A 补充（翻转）序列号并开始传输下一条消息。当 B 收到一个没有被破坏的序列号为 0 的消息时，它开始发送 ACK0 并一直这样做直到它收到一个有效的消息，然后开始发送 ACK1 等。

这意味着当 A 已经在发送序号为 1 的消息时，A 仍然可以收到 ACK0（反之亦然）。它将这种消息视为否定确认码（NAK）。最简单的行为是忽略它们并继续传输。协议可以通过发送虚假消息和序列号 1 进行初始化。序列号为 0 的第一个消息是一个真正的消息。

3.5 滑动窗口协议

在前面的协议中，数据帧只在一个方向上传输。而在大多数实际环境中，往往需要在两个方向上传输数据。实现全双工数据传输的一种方法是，使用两条独立的通信信道，每条都被用作单工数据信道（两条信道方向不同）。如果这样做，则我们拥有两条独立的物理线路，每条线路都有一个“前向”信道（用于数据）和一个“逆向”信道（用于确认）。在这两条线路中，逆向信道的带宽几乎完全被浪费了。实际上，用户付出了两条线路的费用，但是只使用了一条线路的容量。

使用同一条线路来传输两个方向上的数据是一种更好的做法，毕竟协议 2 和协议 3 已经在两个方向上传输帧了，而且逆向信道与前向信道具有同样的容量。接收方只要检查一下进来帧的头部中的 kind 域，就可以区别出该帧是从机器 A 到机器 B 的数据帧还是机器 A 到机器 B 的确认帧。

这个方案还可以进一步优化，当一个数据帧到达时，接收方并不是立即发送一个单独的控制帧，而是抑制一下自己并且开始等待，直到网络层传递给它下一个分组。然后，确认信息被附在往外发送的数据帧上（使用帧头中的 ack 域）。实际上，确认报文搭了下一个外发数据帧的便车。这种“将确认暂时延迟以便可以附到下一个外发数据帧”的技术称为捎带确认（piggy backing）。

与单独确认帧的方法相比较，使用捎带确认法更好地利用了信道的带宽。帧头中的 ack 域只占用几位的开销，而一个单独的帧则需要一个头、确认和校验和。而且，发送的帧越少，也意味着“帧到达”中断也越少，占用接收方的缓冲区可能也越少，这要取决于接收方的软件是如何实现的。在下一个要讨论的协议中，捎带域只占用帧头中的 1 位，它很少会占用许多位。

然而，捎带确认法也引入了一个在单独确认法中不曾出现过的复杂问题。为了捎带一个确认，数据链路层应该等待下一个分组多长时间？如果数据链路层等待的时间超过了发送方的超时周期，则该帧将会被重传，从而违背了确认机制的本意。如果数据链路层是一个先知者，能够预测将来，那么它就知道下一个网络层分组什么时候会到来，因此可以确定是继续等待下去，还是立即发送一个单独的确认帧，这取决于计划的等待时间是多长。当然，数据链路层不可能预测将来，所以它必须采用某种特殊的方法，比如等待一个固定的毫秒数。如果一个新的分组很快就到来了，则确认就可以被捎带上去。否则，如果在这段时间周期之前没有新的分组到来，则数据链路层只是简单地发送一个单独的确认帧。

接下去的三个协议都是双向协议，它们属于同一类，称为滑动窗口协议(sliding window protocol)，这三个协议在效率、复杂性和缓冲区需求等各个方面有所不同。如同所有的滑动窗口协议一样，在这三个协议中，每个外发的帧都包含一个序列号，其范围从 0 到某一个最大值。最大值通常是 2^n-1，这样序列号正好可以填入到一个 n 位的域中。停-等滑动窗口协议使用 $n=1$，限制序列号为 0 和 1，但是更加复杂的版本可以使用任意的 n。

所有滑动窗口协议的本质都是，在任何时刻，发送方总是维持着一组序列号，它们分别对应于允许它发送的帧，称这些帧落在发送窗口(sending window)之内。类似地，接收方也维持着一个接收窗口(receiving window)，对应于一组允许它接受的帧。发送方的窗口和接收方的窗口不必有同样的上下界，甚至也不必有同样的大小。在有些协议中，这两个窗口有固定的大小，但是在其他一些协议中，它们可以随着帧的发送和接收而增大或者缩小。

尽管这些协议使得数据链路层在发送和接收帧的顺序方面有了更多的自由度，但数据链路层协议将分组递交给网络层的次序必须与发送机器上分组被传递给数据链路层的次序相同。同样也不能改变的是：它必须按照发送的顺序递交所有的帧。

发送方窗口内的序列号代表了那些已经被发送，但是还没有被确认的帧，或者是那些可以被发送的帧。任何时候当有新的分组从网络层到来时，它被赋予下一个最高的序列号，并且窗口的上边界增 1。按照这种方法，该窗口持续地维持了一系列未被确认的帧，图 3.9 列举了一个例子。

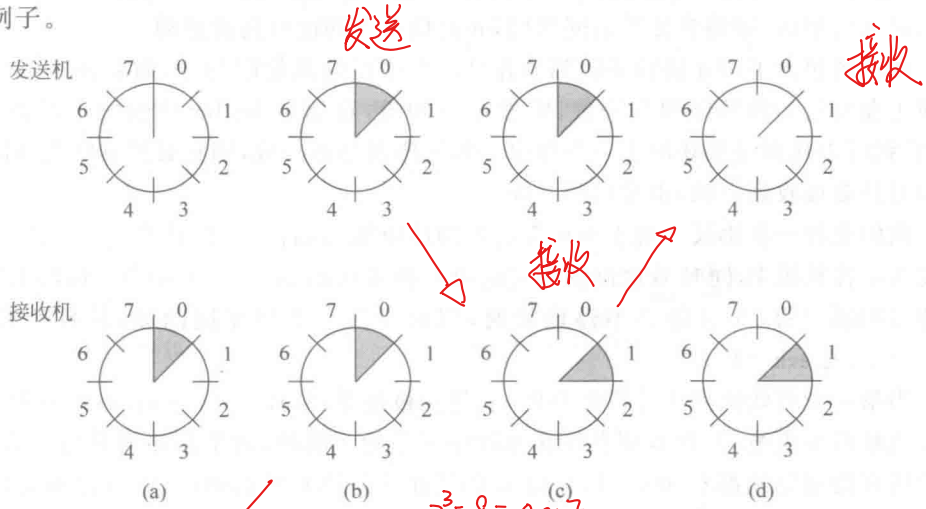

图 3.9 一个大小为 1、有 3 位序列号的滑动窗口

(a) 初始时；(b) 第一帧发送以后；(c) 第一帧接收以后；(d) 第一个确认收到以后

由于当前在发送方窗口内的帧最终有可能在传输过程中丢失或者被损坏，所以，发送方必须在内存中保存好所有这些帧，以便可能进行重传。因此，如果最大的窗口尺寸为 n，则发送方需要 n 个缓冲区来存放未被确认的帧。如果该窗口在某个时候真的达到了它的最大尺寸，则发送方的数据链路层必须强行关闭网络层，直到有一个缓冲区空闲出来为止。

接收方数据链路层的窗口对应于它可以接受的帧，任何落在窗口外面的帧都被丢弃，无需任何提示。当一个新接收到的帧的序列号等于窗口下边界时，接收方将它传递给网络层，并生成一个确认，然后将整个窗口向前移动 1 个位置。与发送方的窗口不同的是，接收方的窗口总是保持最初始的大小，需要注意的是，窗口的大小为 1 意味着数据链路层只按顺序接

受帧，但是对于大一点的窗口，这一条便不再成立。相反，网络层总是按照正确的顺序接收数据，与数据链路层的窗口大小没有关系。

图 3.9 显示了一个最大窗口尺寸为 1 的例子，刚开始时，没有帧要发送，所以发送方窗口的上下边界相等，但是随着时间的推移，状态的变化如该图所示。

1 1 位滑动窗口协议

在讨论一般的情形以前，先讨论最大窗口尺寸为 1 的滑动窗口协议。由于发送方在送出一帧以后，在发送下一帧之前要等待前一帧的确认，所以这样的 1 位滑动窗口协议使用了停-等的办法。

程序 3.5 描述了这样一个协议：与其他的协议一样，它也是从定义变量开始的。next_frame_to_send 指明了发送方正在发送的那一帧，类似地，frame_expected 指明了接收方正在等待的那一帧，这两个变量的取值只可能是 0 和 1。

在一般情况下，以两个数据链路层中的其中一个首先开始，并发送第一帧，换句话说，只有一个数据链路层程序应该在主循环外面包含 to_physical_layer 和 start_timer 过程调用，后面再讨论两个数据链路层同时启动这种较罕见的情形。初始启动的机器从它的网络层获取到第一个分组，然后根据该分组创建一帧，并将它发送出去。当这一帧（或者任何其他的帧）到达时，接收方的数据链路层检查该帧，看它是否为重复帧，如同协议 3 一样。如果这一帧正是所期望的，则将它传递给网络层，并且接收方的窗口向前滑动。

确认域包含了刚才最后接收到的并且没有错误的帧的序列号，如果该序列号与发送方当前正在发送的帧的序列号一致，则发送方知道，存储在 buffer 中的帧已经处理完毕，于是，它就可以从网络层获取下一个分组。如果序列号不一致，则它必须继续发送同一帧。无论何时只要接收到一帧，也要送回一帧。

我们来看一看协议 4 对于不正常情形的适应能力如何，假设计算机 A 试图将它的第 0 帧发送给计算机 B，同时 B 也试图将它的第 0 帧发送给 A。当 A 发送一帧给 B 时，由于 A 的超时间隔太短，A 可能会不停地超时，因而发送一系列相同的帧，并且所有这些帧的 seq＝0，以及 ack＝1。

当第一个有效帧到达计算机 B 时，它将会被接受，并且 frame_expected 设置为 1。所有的后续帧将被拒绝，因为 B 现在正在等待序列号为 1 的帧，而不是序列号为 0 的帧。而且，由于所有的重复帧都有 ack＝1，并且 B 仍然在等待第 0 帧的确认，所以它不会从网络层获取新的分组。

在每一个被拒绝的重复帧进来之后，B 向 A 发送一帧，其中包含 seq＝0 和 ack＝0。最后，这些帧中总会有一帧正确地到达 A，从而引起 A 开始发送下一个分组。丢帧或者提前超时的任何一种情况都不会导致该协议向网络层递交重复的分组，会导致漏掉一个分组或者锁死。

如果双方并发地发送一个初始分组，则会出现一种极为罕见的情形。这种同步的困难程度如图 3.10 所示。图(a)显示了该协议的正常操作。图(b)显示了这种罕见的情形。如果 B 在发送自己的帧之前先等待 A 的第一帧，则整个过程如图(a)所示。每一帧都被接受。然而，如果 A 和 B 同时开始通信，它们的第一帧有交叉，然后数据链路层进入到图(b)的状态。在图(a)中，每一帧到来后都会带给网络层一个新的分组，这里不会有重复帧。在图(b)中，即使没有传输错误，也会有一半的帧是重复的。有一方明显地提前开始也会发生类似的情形，实际上，如果发生多个提前超时，则每一帧有可能要发送三次或者更多。

程序 3.5　滑动窗口协议

```
/* Protocol 4 (sliding window) is bidirectional. */
#define MAX_SEQ 1                          /* must be 1 for protocol 4 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"
void protocol4 (void)
{
  seq_nr next_frame_to_send;               /* 0 or 1 only */
  seq_nr frame_expected;                   /* 0 or 1 only */
  frame r, s;                              /* scratch variables */
  packet buffer;                           /* current packet being sent */
  event_type event;

  next_frame_to_send = 0;                  /* next frame on the outbound stream */
  frame_expected = 0;                      /* frame expected next */
  from_network_layer(&buffer);             /* fetch a packet from the network layer */
  s.info = buffer;                         /* prepare to send the initial frame */
  s.seq = next_frame_to_send;              /* insert sequence number into frame */
  s.ack = 1 - frame_expected;              /* piggybacked ack */
  to_physical_layer(&s);                   /* transmit the frame */
  start_timer(s.seq);                      /* start the timer running */

  while (true) {
      wait_for_event(&event);              /* frame_arrival, cksum_err, or timeout */
      if (event == frame_arrival) {        /* a frame has arrived undamaged. */
          from_physical_layer(&r);         /* go get it */

          if (r.seq == frame_expected) {   /* handle inbound frame stream. */
              to_network_layer(&r.info);   /* pass packet to network layer */
              inc(frame_expected);         /* invert seq number expected next */
          }

          if (r.ack == next_frame_to_send) {  /* handle outbound frame stream. */
              stop_timer(r.ack);           /* turn the timer off */
              from_network_layer(&buffer); /* fetch new pkt from network layer */
              inc(next_frame_to_send);     /* invert senderís sequence number */
          }
      }
      s.info = buffer;                     /* construct outbound frame */
      s.seq = next_frame_to_send;          /* insert sequence number into it */
      s.ack = 1 - frame_expected;          /* seq number of last received frame */
      to_physical_layer(&s);               /* transmit a frame */
      start_timer(s.seq);                  /* start the timer running */
  }
}
```

图 3.10 中的记号为(seq,ack,分组号),星号表示网络层接受了一个分组。

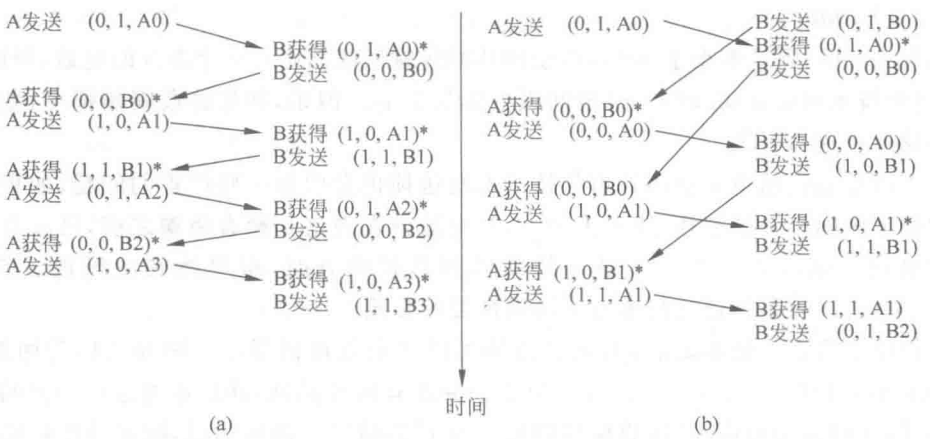

图 3.10　针对协议 4 的两种情形

(a) 正常情形；(b) 异常情形

2. 使用回退 n 帧技术的协议

之前我们一直默认：一个帧到达接收方所需要的传输时间加上确认帧回来的传输时间可以忽略不计。有时这种假设是明显有问题的，在这些情况下，过长的往返时间对于带宽的利用效率有严重的影响。举一个例子，考虑一个 50Kb/s 的卫星信道，它的往返传播延迟为 500ms。想象一下，在该信道上用协议 4 来发送一些 1000b 长度的帧。当 $t=0$ 时，发送方开始发送第一帧；当 $x=20$ms 时，该帧已经被完全发送出去了；当 $t=270$ms 时，该帧才完全到达接收方；当 $t=520$ms 时，确认帧才回到发送方，而这还是在最好情况下才可能做到（即在接收方没有停顿并且确认帧很短）。这意味着，在 96%的时间中（即 500/520）发送方是被阻塞的。换句话说，只有 4%的有效带宽可使用。很显然，从效率角度而言，过长的传输时间，高带宽和短的帧长度，是一种非常差的组合。

上面描述的问题可以看作是协议中以下规则的必然结果：协议要求发送方在发送下一帧之前必须等待前一帧的确认，如果放宽这一限制，就可以获得更好的带宽利用率。这个方案的基本思想是，允许发送方在阻塞之前发送多达 w 帧，而不是 1 帧。通过选择合适的 w 值，发送方在往返传输时间的这一段间隔内可以连续地发送帧，而不会填满窗口。在上面的例子中，w 至少应该为 26。发送方还像以前那样发送第 0 帧，直到 $t=520$ 时，它已经发送了 26 帧，这时，第 0 帧的确认刚好到达。因此，每隔 20ms 就会到达一个确认，所以发送方只要有必要，总是可以发送帧。在任何时候，总有 25 或者 26 个未被确认的帧在等待确认，换言之，发送方的最大窗口尺寸是 26。

任何时候，若带宽与往返延迟的乘积很大，则发送方就需要一个较大的窗口才可以。如果带宽很高，除非窗口真的非常大，即使对于一个并不很长的延迟，发送方也会很快用完它的窗口，如果延迟很长（比如通过一个地球同步卫星信道），更是如此。这两个因子的乘积基本上说明了这条管道的容量，发送方为了达到尖峰效率，需要利用这条管道来马不停蹄地发送数据。

这项技术称为管道化技术（pipelining），如果信道的容量是 bb/s，帧长为 1b，往返传输时间为 R(s)，则传输一帧所需要的时间为 $1/b$(s)。在一个数据帧的最后一位被发送出去之后，经过 $R/2$(s)的延迟之后该位到达接收方，再经过 $R/2$(s)的延迟之后确认帧回来，总共延迟为 R。在停-等协议中，这条线路有 $1/b$(s)是忙的，而 R(s)是空闲的，所以，线路的利用率为 $1/(1+bR)$。

如果 $1<bR$，则效率小于 50%，由于确认帧传输回来总是有一个非 0 的延迟，所以，原则上，管道化技术可以使得线路在这段间隔中也是忙的。但是，如果这段间隔很小，则没有必要将协议搞得这么复杂。

在不可靠通信信道上使用管道化技术来发送帧也会引起一些严重的问题，如果在一个很长的帧流中间有一帧损坏，或者丢失了，在发送方发现这一帧有错误之前，已经有大量的帧成功地到达接收方。当一个被损坏的帧到达接收方时，很显然它应该被丢弃掉，在图 3.11 中，可以看到管道化技术对于错误恢复的影响。

在使用了管道化技术之后，有两种方法可以用来处理错误。一种办法称为回退 n 帧，(go back n)，这种方法很简单，接收方只要丢弃所有后续的帧，并且不为这些丢弃的帧发送确认即可，这种策略对应于“接收窗口的尺寸为 1”的情形。换句话说，除了数据链路层必须要递交给网络层的下一帧以外，不接受其他帧。如果在定时器超时以前，发送方的窗口已经

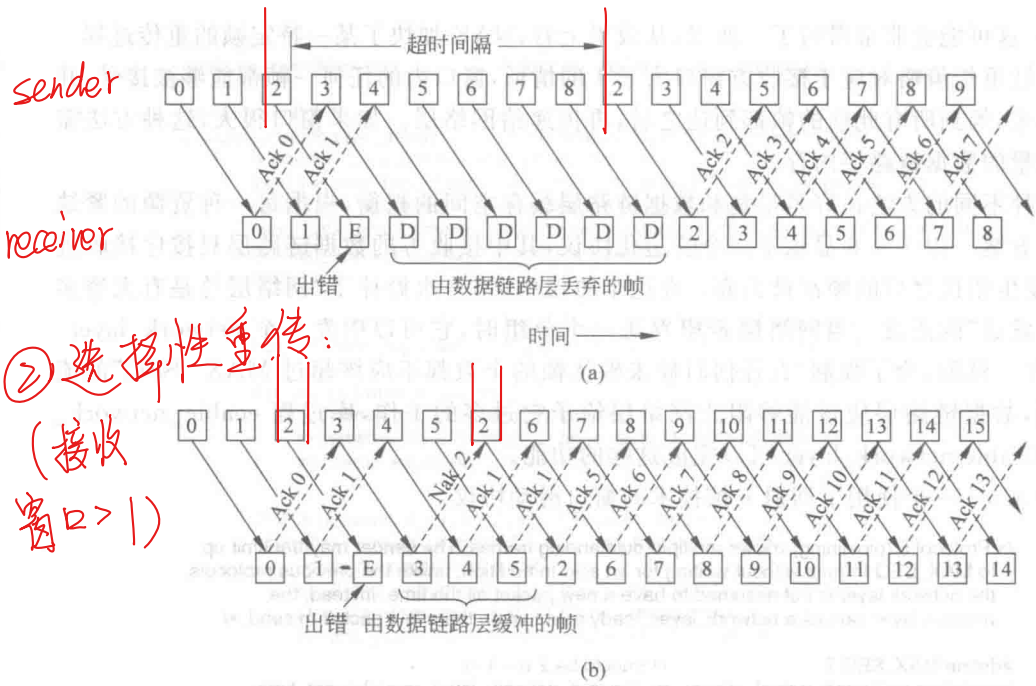

图 3.11 管道化技术与错误修复

(a) 接收方的窗口尺寸为 1；(b) 接收方的窗口尺寸较大

满了，则管道将空闲。最终，发送方将会超时，并且按照顺序重传所有未被确认的帧，从最初受损或者丢失的那一帧开始。如果错误率比较高，这种方法可能会浪费大量的带宽。

在图 3.11(a)中，可以看到回退 n 帧的情形，其中发送方的窗口比较大。第 0 帧和第 1 帧被正确地接收和确认，然而，第 2 帧损坏或者丢失了。但发送方不知道出现了问题，它继续发送后续的帧，直到第 2 帧的定时器过期为止。然后，它退回到第 2 帧，并从这里开始发送 2,3,4 帧等。

另一种处理错误的通用策略称为选择性重传(selective repeat)。当使用了这种策略以后，接收到的坏帧会被丢弃，坏帧后面的好帧继续被缓存起来。当发送方超时以后，它只重传最早的未被确认的那一帧。如果那一帧正确到达接收方，则接收方依次将它所缓存的帧递交给网络层。选择性重传策略通常也与以下的策略结合起来使用：当接收方检测到错误(例如，帧的校验和错误或者序列号不正确)时，它发送一个否定的确认(NAK，negative acknowledgement)。NAK 可以激发重传操作，而不需要等到相应的定时器过期，因此，NAK 可以提高性能。

在图 3.11(b)中，第 0 帧和第 1 帧被正确接收到，并得到确认，而第 2 帧则丢失了。当第 3 帧到达接收方时，那里的数据链路层注意到它已经错过了一帧，所以它针对第 2 帧会送回一个 NAK，但是将第 3 帧缓存起来。当第 4,5 帧到达之后，它们也被数据链路层缓存起来，不是被传递给网络层。第 2 帧的 NAK 达到发送方后，发送方立即重新发送第 2 帧。如图 3.11 中所示，当该帧到达接收方时，数据链路层现在有了第 2,3,4 和 5 帧，就可以将它们按照正确的顺序传递给网络层，也可以确认所有这些帧(从第 2 帧到第 5 帧)。如果 NAK 也丢失了，则发送方的第 2 帧定时器最终会超时，发送方就会重新发送第 2 帧(仅仅这一

帧),但是,这可能会非常滞后了。所以,从效果上看,NAK 加快了某一特定帧的重传过程。

选择性重传策略对应于接收方窗口大于 1 的情形,窗口内的任何一帧都能够被接受,并被缓存起来,等到所有此前的帧都到达之后,再传递给网络层。如果窗口很大,这种方法需要消耗大量的数据链路层内存。

这两种不同的方法正好是带宽和数据链路层缓存空间的权衡,根据每一种资源的紧缺程度选定方案。程序 3.6 显示了一个管道化协议,其中接收方的数据链路层只按序接收进来的帧,发生错误之后的帧都被丢弃。在这个协议中,第一次抛掉了"网络层总是有无穷多的分组要发送"的假设。当网络层希望发送一个分组时,它可以引发一个 network_layer_ready 事件。然而,为了强制"在任何时候未确认帧的个数都不应该超过 MAX_ SEQ"的流控制规则,数据链路层应该能够阻止网络层给予它过多的工作,库过程 enable_network_layer 和 disable-network_layer 可以完成这样的功能。

程序 3.6 一个使用了回退 n 帧技术的窗口滑动协议

```
/* Protocol 5 (pipelining) allows multiple outstanding frames. The sender may transmit up
   to MAX_SEQ frames without waiting for an ack. In addition, unlike the previous protocols,
   the network layer is not assumed to have a new packet all the time. Instead, the
   network layer causes a network_layer_ready event when there is a packet to send. */

#define MAX_SEQ 7                       /* should be 2^n - 1 */
typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready} event_type;
#include "protocol.h"

static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
/* Return true if a <=b < c circularly; false otherwise. */
  if (((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a)))
      return(true);
   else
      return(false);
}

static void send_data(seq_nr frame_nr, seq_nr frame_expected, packet buffer[ ])
{
/* Construct and send a data frame. */
  frame s;                              /* scratch variable */

  s.info = buffer[frame_nr];            /* insert packet into frame */
  s.seq = frame_nr;                     /* insert sequence number into frame */
  s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);/* piggyback ack */
  to_physical_layer(&s);                /* transmit the frame */
  start_timer(frame_nr);                /* start the timer running */
}

void protocol5(void)
{
  seq_nr next_frame_to_send;            /* MAX_SEQ > 1; used for outbound stream */
  seq_nr ack_expected;                  /* oldest frame as yet unacknowledged */
  seq_nr frame_expected;                /* next frame expected on inbound stream */
  frame r;                              /* scratch variable */
  packet buffer[MAX_SEQ + 1];           /* buffers for the outbound stream */
  seq_nr nbuffered;                     /* # output buffers currently in use */
  seq_nr i;                             /* used to index into the buffer array */
  event_type event;

  enable_network_layer();               /* allow network_layer_ready events */
  ack_expected = 0;                     /* next ack expected inbound */
  next_frame_to_send = 0;               /* next frame going out */
  frame_expected = 0;                   /* number of frame expected inbound */
  nbuffered = 0;                        /* initially no packets are buffered */
```

```
while (true) {
  wait_for_event(&event);              /* four possibilities: see event_type above */

  switch(event) {
    case network_layer_ready:          /* the network layer has a packet to send */
         /* Accept, save, and transmit a new frame. */
         from_network_layer(&buffer[next_frame_to_send]); /* fetch new packet */
         nbuffered = nbuffered + 1;  /* expand the sender's window */
         send_data(next_frame_to_send, frame_expected, buffer);/* transmit the frame */
         inc(next_frame_to_send);    /* advance sender's upper window edge */
         break;

    case frame_arrival:                /* a data or control frame has arrived */
         from_physical_layer(&r);      /* get incoming frame from physical layer */

         if (r.seq == frame_expected) {
              /* Frames are accepted only in order. */
              to_network_layer(&r.info);  /* pass packet to network layer */
              inc(frame_expected);  /* advance lower edge of receiver's window */
         }
         /* Ack n implies n - 1, n - 2, etc.  Check for this. */
         while (between(ack_expected, r.ack, next_frame_to_send)) {
              /* Handle piggybacked ack. */
              nbuffered = nbuffered    1; /* one frame fewer buffered */
              stop_timer(ack_expected); /* frame arrived intact; stop timer */
              inc(ack_expected);      /* contract sender's window */
         }
         break;

    case cksum_err: break;             /* just ignore bad frames */

    case timeout:                      /* trouble; retransmit all outstanding frames */
         next_frame_to_send = ack_expected;    /* start retransmitting here */
         for (i = 1; i <= nbuffered; i++) {
              send_data(next_frame_to_send, frame_expected, buffer);/* resend 1 frame */
              inc(next_frame_to_send);  /* prepare to send the next one */
         }

  }

  if (nbuffered < MAX_SEQ)
         enable_network_layer();
  else
         disable_network_layer();
 }
}
```

注意，尽管不同序列号的个数是 MAX_SEQ－1，分别为 0，1，2，…，MAX_SEQ，但是，在任何时候，未确认帧的最大数量是 MAX_SEQ，而不是 MAX_SER＋1。为了看清楚为什么这个限制是必要的，请考虑下面 MAX_SEQ＝7 的场景。

（1）发送方发送第 0～7 帧。

（2）第 7 帧的捎带确认最终被送回到发送方。

（3）发送方送出另外 8 帧，其序列号仍然是 0～7。

（4）现在第 7 帧的另一个捎带确认也回来了。

问题来了：属于第二批的 8 帧全部成功到达了吗？或者这 8 帧全部丢失了（把出错之后丢弃的帧也算作丢失了）？在这两种情况下，接收方都会发送第 7 帧的确认，而发送方无从分辨。因此，未确认帧的最大数目必须为 MAX_SEQ。

虽然协议 5 后到来的帧没有缓存出错，但是它也没有因此摆脱缓存的问题。由于发送方可能在将来的某个时刻重传所有未被确认的帧，所以，它必须把已经送出的帧保留一段时

间，直到它知道接收方已经接受了这些帧。当第 n 帧的确认到来时，第 $n-1$ 帧、第 $n-2$ 帧等也都自动被确认了。当有些先前捎带确认的帧被丢失或者受损之后，这个特性显得尤为重要。当任何一个确认到来时，数据链路层都要检查看是否可以释放一些缓冲区。如果释放掉一些缓冲区（即窗口中又有了一些可以利用的空间），则原来被阻塞的网络层现在又可以激发更多的 network_layer_ready 事件。

对于这个协议，假设链路上总是有反向的流量可以捎带确认，如果没有这样的流量，则这些确认报文就不会被送回来。协议 4 并不需要这样的假设，因为它每次接收到一帧就送回一帧，即使它刚刚已经送出了这一帧也是如此。将在下一个协议中介绍一种非常巧妙的方法来解决这个单向流量的问题。

因为协议 5 有多个未被确认的帧，所以逻辑上它需要多个定时器，即每一个未被确认的帧都需要一个定时器。每一帧的超时是独立的，相互之间没有关系。用软件很容易模拟所有这些定时器：只需使用一个硬件时钟，它可以周期性地引发中断。所有未发生的超时事件构成了一个链表，链表中的每个节点描述了一个定时器离过期还有多少个时钟滴答，该定时器为哪一帧计时，以及一个指针指向下一个节点。

为了演示如何实现多个定时器，请参照图 3.12(a)中的例子，假设每 100ms 时钟滴答一次，刚开始时，实际的时间为 10：00：00.0；有三个超时事件，分别定在 10：00：00.5，10：00：01.3 和 10：00：01.9。每次硬件时钟滴答到来时，实际的时间被更新，链表头上的滴答计数器减 1。当滴答计数器变成 0 时，就引发一个超时事件，并将该节点从链表中移除，如图 3.12(b)所示。虽然这种实现方式也要求在调用 start_timer 或 stop_timer 时扫描该链表，但是，在每次滴答中断时它并不要求更多的工作。在协议 5 中，start_timer 和 stop_timer 这两个过程都带一个参数，表示正在对哪一帧计时。

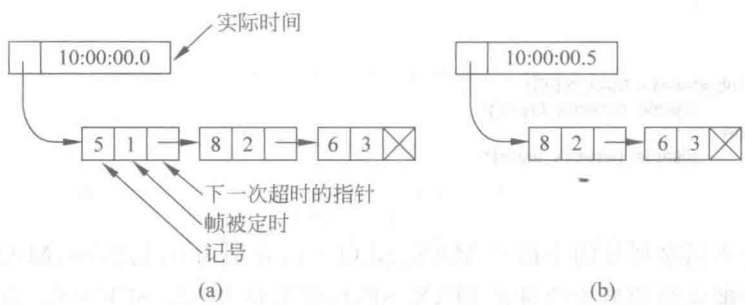

图 3.12 用软件来模拟多个定时器

3. 使用选择性重传的协议

如果错误发生率比较低，选用协议 5 更加合适，但如果线路质量很差，在重传的帧上要浪费很多带宽。另一种处理错误的策略是，对于坏帧或者丢失帧后面的帧，接收方也可以接受并缓存起来。这样的协议不会因为前面的帧被损坏或者丢失，而丢弃后续的帧。

在这个协议中，发送方和接收方都维持了一个窗口，窗口内部包含了那些可以接受的序列号。发送方的窗口大小从 0 开始，以后可以增大到某一个预设的最大值 MAX_SEQ。而接收方的窗口总是固定大小的，其大小等于 MAX_SEQ。接收方为其窗口内的每一个序列号保留于一个缓冲区。与每一个缓冲区相关联的还有一位(arrived)，用来指明该缓冲区是

满的还是空的。当任何一帧到达时，接收方通过 between 函数检查它的序列号，看是否落在窗口内。如果确实落在窗口内，并且以前没有接收过这一帧，则接受该帧，并且保存起来。不管这一帧是否包含了网络层所期望的下一个分组，这个过程是肯定要执行的。当然，该帧一定会被保存在数据链路层中，并且，直到所有序列号比它小的那些帧都已经按照正确的顺序递交给网络层之后，它才会被传递给网络层。程序 3.7 列举了一个使用该算法的协议。

程序 3.7 使用选择性重传的窗口滑动协议

```
/* Protocol 6 (nonsequential receive) accepts frames out of order, but passes packets to the
   network layer in order. Associated with each outstanding frame is a timer. When the timer
   expires, only that frame is retransmitted, not all the outstanding frames, as in protocol 5. */

#define MAX_SEQ 7                                   /* should be 2^n - 1 */
#define NR_BUFS ((MAX_SEQ + 1)/2)
typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready, ack_timeout} event_type;
#include "protocol.h"
boolean no_nak = true;                              /* no nak has been sent yet */
seq_nr oldest_frame = MAX- SEQ + 1;                 /* initial value is only for the simulator */

static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
/* Same as between in protocol5, but shorter and more obscure. */
  return ((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a));
}

static void send_frame(frame_kind fk, seq_nr frame_nr, seq_nr frame_expected, packet buffer[])
{
/* Construct and send a data, ack, or nak frame. */
  frame s;                                          /* scratch variable */

  s.kind = fk;                                      /* kind == data, ack, or nak */
  if (fk == data) s.info = buffer[frame_nr % NR_BUFS];
  s.seq = frame_nr;                                 /* only meaningful for data frames */
  s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
  if (fk == nak) no_nak = false;                    /* one nak per frame, please */
  to_physical_layer(&s);                            /* transmit the frame */
  if (fk == data) start_timer(frame_nr % NR_BUFS);
  stop_ack_timer();                                 /* no need for separate ack frame */
}

void protocol6(void)
{
  seq_nr ack_expected;                              /* lower edge of sender's window */
  seq_nr next_frame_to_send;                        /* upper edge of sender's window + 1 */
  seq_nr frame_expected;                            /* lower edge of receiver's window */
  seq_nr too_far;                                   /* upper edge of receiver's window + 1 */
  int i;                                            /* index into buffer pool */
  frame r;                                          /* scratch variable */
  packet out_buf[NR_BUFS];                          /* buffers for the outbound stream */
  packet in_buf[NR_BUFS];                           /* buffers for the inbound stream */
  boolean arrived[NR_BUFS];                         /* inbound bit map */
  seq_nr nbuffered;                                 /* how many output buffers currently used */
  event_type event;

  enable_network_layer();                           /* initialize */
  ack_expected = 0;                                 /* next ack expected on the inbound stream */
  next_frame_to_send = 0;                           /* number of next outgoing frame */
  frame_expected = 0;
  too_far = NR_BUFS;
  nbuffered = 0;                                    /* initially no packets are buffered */
  for (i = 0; i < NR_BUFS; i++) arrived[i] = false;

  while (true) {
    wait_for_event(&event);                         /* five possibilities: see event_type above */
    switch(event) {
      case network_layer_ready:                     /* accept, save, and transmit a new frame */
           nbuffered = nbuffered + 1;               /* expand the window */
           from_network_layer(&out_buf[next_frame_to_send % NR_BUFS]); /* fetch new packet */
           send_frame(data, next_frame_to_send, frame_expected, out_buf);/* transmit the frame */
```

```
            inc(next_frame_to_send);                /* advance upper window edge */
            break;
        case frame_arrival:                         /* a data or control frame has arrived */
            from_physical_layer(&r);                /* fetch incoming frame from physical layer */
            if (r.kind == data) {
                /* An undamaged frame has arrived. */
                if ((r.seq != frame_expected) && no_nak)
                   send_frame(nak, 0, frame_expected, out_buf); else start_ack_timer();
                if (between(frame_expected, r.seq, too_far) && (arrived[r.seq%NR_BUFS] == false)) {
                    /* Frames may be accepted in any order. */
                    arrived[r.seq % NR_BUFS] = true;      /* mark buffer as full */
                    in_buf[r.seq % NR_BUFS] = r.info;     /* insert data into buffer */
                    while (arrived[frame_expected % NR_BUFS]) {
                        /* Pass frames and advance window. */
                        to_network_layer(&in_buf[frame_expected % NR_BUFS]);
                        no_nak = true;
                        arrived[frame_expected % NR_BUFS] = false;
                        inc(frame_expected);      /* advance lower edge of receiver's window */
                        inc(too_far);             /* advance upper edge of receiver's window */
                        start_ack_timer();        /* to see if a separate ack is needed */
                    }
                }
            }

            if((r.kind==nak) && between(ack_expected,(r.ack+1)%(MAX_SEQ+1),next frame to send))
                send_frame(data, (r.ack+1) % (MAX_SEQ + 1), frame_expected, out_buf);

            while (between(ack_expected, r.ack, next_frame_to_send)) {
                nbuffered = nbuffered   1;            /* handle piggybacked ack */
                stop_timer(ack_expected % NR_BUFS);   /* frame arrived intact */
                inc(ack_expected);                    /* advance lower edge of sender's window */
            }
            break;
        case cksum_err:
            if (no_nak) send_frame(nak, 0, frame_expected, out_buf);/* damaged frame */
            break;
        case timeout:
            send_frame(data, oldest_frame, frame_expected, out_buf);/* we timed out */
            break;
        case ack_timeout:
            send_frame(ack,0,frame_expected, out_buf);     /* ack timer expired; send ack */
        }
        if (nbuffered < NR_BUFS) enable_network_layer(); else disable_network_layer();
    }
}
```

非顺序接收带来了一些特殊的问题，可以从这个例子中反映出：假设使用 3 位的序列号，那么，发送方允许连续发送 7 帧，然后开始等待确认。刚开始时，发送方和接收方的窗口如图 3.13(a)所示。现在发送方送出第 0～6 帧，接收方的窗口允许它接受任何序列号在 0～6(含)之间的帧。这 7 帧全部正确地到达了，所以接收方对它们进行确认，并且向前移动它的窗口，允许接收第 7，0，1，2，3，4 或 5 帧，如图 3.13(b)所示，所有这 7 个缓冲区都标记为空。

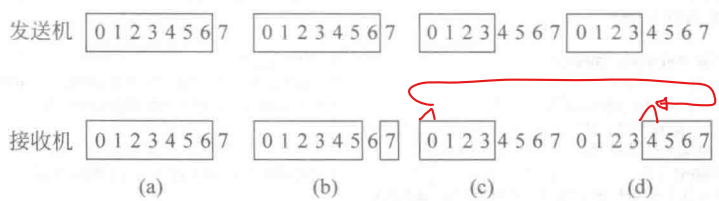

图 3.13　发送方和接收方的窗口

(a) 窗口大小为 7 的初始状态；(b) 7 帧都已送出并接收，但是均未被确认；

(c) 窗口大小为 4 的初始状态；(d) 4 帧都已送出并接收，但是均未被确认

此时，灾难降临了，所有的确认都被毁掉了。发送方最终超时了，并且重发第 0 帧。当这一帧到达接收方时，接收方检查它的序列号，看是否落在它的窗口中。不幸的是，如图 3.13(b)所示，第 0 帧落在新的窗口中，所以它被接收了。接收方送回第 6 帧的捎带确认，因为第 0～6 帧都已经接收到了。

此时发送方很高兴地得知，所有它发出去的帧都已经正确地到达了，所以它向前移动它的窗口，并立即发送第 7、0、1、2、3、4 和 5 帧。第 7 帧将被接收方接受，并且它的分组直接传递给网络层。紧接着，接收方的数据链路层进行检查，看它是否已经有了一个有效的第 0 帧，它发现确实已经有了(即前面重发的第 0 帧)，然后把内嵌的分组传递给网络层。因此，网络层得到了一个不正确的分组，从而导致协议失败。

这个问题的本质是，当接收方向前移动了它的窗口之后，新的有效序列号范围与老的范围有重叠。因此，后续的一批帧可能是重复的帧(如果所有的确认都丢失了)，也可能是新的帧(如果所有的确认都接收到了)，接收方将无法区分这两种情形。

解决这个难题的方法是，确保接收方向前移动窗口之后，新的窗口与老的窗口之间没有重叠。为了保证没有重叠，最大的窗口尺寸应该不超过序列号范围的一半，如图 3.13(c)和图 3.13(d)所示。例如，如果用 4 位来表达序列号，则序列号的范围为 0～15。在任何时候，应该只有 8 个未被确认的帧处于等待状态。按照这种方法，如果接收方已经接受了 0～7 帧，并且向前移动了窗口，以便允许接收第 8～15 帧，这样它可以明确地区分出后续的帧是重传帧(序列号为 0～7)，还是新帧(序列号为 8～15)。一般地，协议 6 的窗口尺寸为(MAX_SEQ+1)/2，因此，对于 3 位序列号，窗口尺寸为 4。

一个有意思的问题是：接收方必须有多少个缓冲区？无论如何，接收方不可能接受序列号低于窗口下边界的帧，也不可能接受序列号高于上边界的帧。因此，所需要的缓冲区的数量等于窗口的尺寸，而不是序列号的范围。在上面的 4 位序列号的例子中，只需要 8 个缓冲区就够了，编号为 0～7。当第 1 帧到达时，它被放在(i mod 8)号缓冲区中。请注意，虽然 i 和(i+8) mod“竞争”同一个缓冲区，但是它们永远不会同时在窗口内，因为如果那样，窗口的尺寸至少为 9。

出于同样的原因，所需要的定时器的数量也等于缓冲区的数量，而不是序列号空间的大小。实际上，每个缓冲区都有一个相关联的定时器，当定时器超时时，缓冲区的内容就要被重传在协议 5 中，有一个隐含的设定：信道的负载很繁重。当一帧到达时，接收方不立即发送确认，而是把该帧的确认放在下一个往外发的数据帧中捎带回去。如果反向的流量很轻，则确认信息将会滞留很长时间。如果在一个方向上有很大的流量，而另一个方向上根本没有流量，则只有 MAX_SEQ 个分组被发送出去，然后协议就阻塞了，这就是为什么必须要假设总是有反向流量。

在协议 6 中，这个问题被修正了。当一个按正常次序发送的数据帧到达之后，接收方通过 start_ack_timer 启动一个辅助的定时器。如果在定时器到期之前，没有出现反向的流量，则发送一个单独的确认帧。由于该辅助定时器而导致的中断称为 ack_timeout 事件。利用这种方式，即使单向的数据流也没有问题了，缺少可以捎带确认的反向数据帧不再是一个障碍了。只需要一个辅助定时器就可以了，如果该定时器正在运行时 start_ack_timer 又被调用了，则它被重置为一个完整的确认超时间隔。

与辅助定时器关联的超时间隔应该明显短于与数据帧关联的定时器间隔，这是非常关

键的。这个条件是必要的，因为它应该保证一个正确接收的帧能够尽早地被确认，从而该帧的重传定时器还没有过期，所以还没有重传该帧。

协议 6 使用了比协议 5 更加有效的策略来处理错误，当接收方有理由怀疑出现了错误时，它就给发送方送回一个否定的确认(NAK)帧。这样的帧实际上是一个重传请求，在 NAK 中指定了要重传的帧，接收方应该怀疑接收到一个受损的帧，或者到达的帧并非是自己所期望的(可能有丢帧错误)。为了避免多次请求重传同一个丢失帧，接收方应该记录下对于某一帧是否已经发送过 NAK 在协议 6 中，如果对于 frame_expected 还没有发送过 NAK，则变量 no_nak 为 true。如果 NAK 被损坏了或者丢失了，不会有实质性的伤害，因为发送方最终会超时，无论如何它会重传丢失的帧。如果一个 NAK 被发送出去之后丢了，而接收方又收到一个错误的帧，则 no_nak 将为 true，并且辅助定时器将被启动。当辅助定时器超时后，一个 NAK 帧将被发送出去，以便将发送方重新同步到接收方的当前状态。

在有些情况下，从一帧被发送出去开始，该帧到达目的地，再在那里被处理，然后它的确认被送回来，这整个过程所需要的时间近似为常数。在这样的情况下，发送方可以调整它的定时器，让它略微大于正常情况下从发送一帧到接收到它的确认之间的时间间隔。然而，如果这段时间的变化非常大，则发送方必须做出选择，要么将定时器的间隔设置得比较小(其风险是不必要的重传)，要么将它设置得比较大(发生错误之后长时间地空闲)。

这两种选择都会浪费带宽，如果反向的流量比较稀少，则确认之前的时间将非常不规则，当有反向流量时这段时间非常短，当没有反向流量时这段时间非常长。一般地，当确认间隔的标准偏差与间隔本身相比非常小时，定时器可以设置得“紧”一点，这时 NAK 并不很有用，否则，定时器应该设置得“松”一点，从而避免不必要的重传，但是 NAK 可以加快丢失帧或者损坏帧的重传速度。

与超时和 NAK 紧密相关的一个问题是确定哪一帧引发了超时，在协议 5 中，它总是 ack_expected 的帧，因为它总是最早的帧。在协议 6 中，没有一种很具体的方法来确定谁引发了超时。假定已经发送了第 0～4 帧，这意味着未确认帧是按照时间的先后顺序来排列的，列表为 01234。现在请想象这样的情形：第 0 帧超时了，第 5 帧(新帧)被发送出去了，第 1 帧超时了，第 2 帧超时了，第 6 帧(又一新帧)也被发送出去了。这时候，未确认帧的列表是 3405126，也是按照发送的时间先后顺序排列。如果所有进来的流量(即那些包含确认的帧)被丢失一段时间，则这 7 个未确认的帧将会依次超时。

为了避免使该例子过于复杂，我们没有显示定时器的管理过程。只是假设在超时的时候 oldest_frame 变量已经设置好了，它指示出哪一帧超时了。

3.6 协议验证

在本节中，将学习一些模型和技术来描述和验证这些协议。

1. 有限状态机模型

在许多协议模型中用到的一个关键概念是有限状态机(finite state machine)，利用这项技术，每个协议机(protocol machine，即发送方或接收方)在任何一个时刻，总是处于一种特定的状态。它的状态是由所有变量的值组成的，其中也包括程序计数器。