第 3 章　数据链路层

在本章，我们将学习网络模型中的第二层（即数据链路层）的设计原则。学习内容涉及两台相邻机器实现可靠有效的完整信息块（称为帧）通信的一些算法，而不像物理层那样只关注单个比特传输。这里的相邻指两台机器通过一条通信信道连接起来，通信信道在概念上就像一条线路（比如同轴电缆、电话线或者无线信道）。信道像一条线路的本质特性使得信道上传递的比特顺序与发送顺序完全相同。

刚开始，你可能认为这个问题非常简单，似乎没有什么内容需要学习——机器 A 把比特放到线路上，然后机器 B 将这些比特取下来。不幸的是，通信线路偶尔会出错。而且，它们只有有限的数据传输率，并且在比特的发送时间和接收时间之间存在一个非零延迟。这些限制对数据传输的效率有非常重要的影响。通信所采用的协议必须考虑所有这些因素。这些协议正是本章的主题。

在介绍了数据链路层的关键设计问题之后，我们将通过考察错误的本质以及如何检测和纠正这些错误来开始数据链路层协议的学习。然后，我们将学习一系列复杂性逐步递增的协议，每个协议解决了本层中越来越多的问题。最后，我们将给出一些数据链路层协议的例子来结束本章。

3.1　数据链路层的设计问题

数据链路层使用物理层提供的服务在通信信道上发送和接收比特。它要完成一些功能，包括：

（1）向网络层提供一个定义良好的服务接口。

（2）处理传输错误。

（3）调节数据流，确保慢速的接收方不会被快速的发送方淹没。

为了实现这些目标，数据链路层从网络层获得数据包，然后将这些数据包封装成**帧**（frame）以便传输。每个帧包含一个帧头、一个有效载荷（用于存放数据包）以及一个帧尾，如图 3-1 所示。帧的管理构成了数据链路层工作的核心。在后面的章节中，我们将详细地讨论上面提到的这些问题。

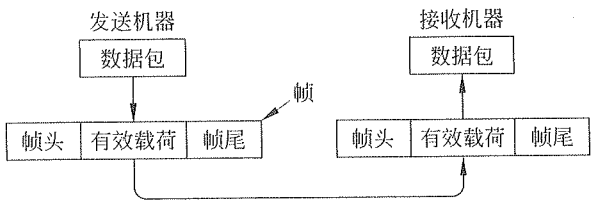


图 3-1　数据包和帧的关系

虽然本章明确讨论数据链路层及其协议，但是，我们在本章中学习的许多原理，比如错误控制和流量控制等同样可以在传输层和其他协议中寻觅到类似的踪迹。这是因为可靠性是网络的总目标，这个目标的实现需要各层次的紧密配合。实际上，在许多网络中，这些功能最常出现的地方是上层，数据链路层只要做很少的一点工作就已经“足够好”。然而，不管它们出现在哪里，原理是非常一致的。在数据链路层中，它们通常表现出最为简单和纯粹的形式，因此，数据链路层是详细学习这些原理的绝佳之地。

3.1.1 提供给网络层的服务

数据链路层的功能是为网络层提供服务。最主要的服务是将数据从源机器的网络层传输到目标机器的网络层。在源机器的网络层有一个实体（称为进程），它将一些比特交给数据链路层，要求传输到目标机器。数据链路层的任务就是将这些比特传输给目标机器，然后再进一步交付给网络层，如图3-2（a）所示。实际的传输过程则是沿着图3-2（b）所示的路径进行的，但很容易将这个过程想象成两个数据链路层的进程使用一个数据链路协议进行通信。基于这个原因，在本章中我们将隐式使用图3-2（a）的模型。

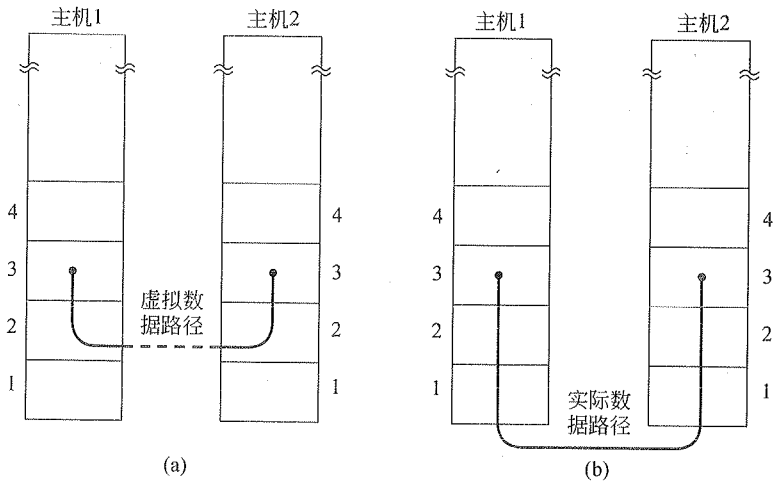


图 3-2
（a）虚拟通信；（b）实际通信

数据链路层可以设计成向上提供各种不同的服务。实际提供的服务因具体协议的不同而有所差异。一般情况下，数据链路层通常会提供以下3种可能的服务：

（1）无确认的无连接服务。

（2）有确认的无连接服务。

（3）有确认的有连接服务。

无确认的无连接服务是指源机器向目标机器发送独立的帧，目标机器并不对这些帧进行确认。以太网就是一个提供此类服务的数据链路层极好实例。采用这种服务，事先不需要建立逻辑连接，事后也不用释放逻辑连接。若由于线路的噪声而造成了某一帧的丢失，数据链路层并不试图去检测这样的丢帧情况，更不会去试图恢复丢失的帧。这类服务合适两种场合，第一种是错误率很低的场合，此时差错恢复过程可以留给上层来完成；第二种

是实时通信，比如语音传输，因为在实时通信中数据迟到比数据受损更糟糕。

迈向可靠性的下一步是有确认的无连接服务。当向网络层提供这种服务时，数据链路层仍然没有使用逻辑连接，但其发送的每一帧都需要单独确认。这样，发送方可知道一个帧是否已经正确地到达目的地。如果一个帧在指定的时间间隔内还没有到达，则发送方将再次发送该帧。这类服务尤其适用于不可靠的信道，比如无线系统。802.11（WiFi）就是此类服务的一个很好例子。

或许有一点值得强调，那就是在数据链路层提供确认只是一种优化手段，永远不应该成为一种需求。网络层总是可以发送一个数据包，然后等待该数据包被确认。如果在计时器超时之前，该数据包的确认还没有到来，那么发送方只要再次发送整个报文即可。这一策略的麻烦在于它可能导致传输的低效率。链路层对帧通常有严格的长度限制，这是由硬件所决定的；除此之外，还有传播延迟。但网络层并不清楚这些参数。网络层可能发出了一个很大的数据包，该数据包被拆分并封装到（比如说）10 个帧中，而且 20%的帧在传输中被丢失，那么这个数据包可能需要花很长的时间才能传到接收方。相反地，如果每个帧都单独确认和必要时重传，那么出现的差错就能更直接并且更快地被检测到。在可靠信道上，比如光纤，重量级数据链路协议的开销可能是不必要的；但在无线信道上，由于信道内在的不可靠性，这种开销还是非常值得的。

我们再回到有关服务的话题上，数据链路层向网络层提供的最复杂服务是面向连接的服务。采用这种服务，源机器和目标机器在传输任何数据之前要建立一个连接。连接上发送的每一帧都被编号，数据链路层确保发出的每个帧都会真正被接收方收到。它还保证每个帧只被接收一次，并且所有的帧都将按正确的顺序被接收。因此，面向连接的服务相当于为网络层进程提供了一个可靠的比特流。它适用于长距离且不可靠的链路，比如卫星信道或者长途电话电路。如果采用有确认的无连接服务，可以想象丢失了确认可能导致一个帧被收发多次，因而将浪费带宽。

当使用面向连接的服务时，数据传输必须经过三个不同的阶段。在第一个阶段，要建立连接，双方初始化各种变量和计数器，这些变量和计数器记录了哪些帧已经接收到，哪些帧还没有收到。在第二个阶段，才真正传输一个或者多个数据帧。在第三个也是最后一个阶段中，连接被释放，所有的变量、缓冲区以及其他用于维护该连接的资源也随之被释放。

3.1.2　成帧

为了向网络层提供服务，数据链路层必须使用物理层提供给它的服务。物理层所做的只是接收一个原始比特流，并试图将它传递给目标机器。如果信道上存在噪声，就像大多数无线链路和某些有线链路那样，物理层就会在它的信号中添加某种冗余，以便将误码率降到一定程度。然而，数据链路层接收到的比特流不能保证没有错误。某些比特的值可能已经发生变化，接收到的比特个数可能少于、等于或者多于发送的比特数量。检测错误和纠正错误（有必要的话）的工作正是数据链路层该做的。

对于数据链路层来说，通常的做法是将比特流拆分成多个离散的帧，为每个帧计算一个称为校验和的短令牌（本章后面将讨论校验和算法），并将该校验和放在帧中一起传输。当帧到达目标机器时，要重新计算该帧的校验和。如果新算出来的校验和与该帧中包含的

校验和不同，则数据链路层知道传输过程中产生了错误，它就会采取措施来处理错误（比如丢掉坏帧，可能还会发回一个错误报告）。

拆分比特流的实际工作比初看上去的要复杂得多。一个好的设计方案必须使接收方很容易发现一个新帧的开始，同时所使用的信道带宽要少。我们将考察下列4种方法：

（1）字节计数法。

（2）字节填充的标志字节法。

（3）比特填充的标志比特法。

（4）物理层编码违禁法。

第一种成帧方法利用头部中的一个字段来标识该帧中的字符数。当接收方的数据链路层看到字符计数值时，它就知道后面跟着多少个字节，因此也就知道了该帧在哪里结束。这项技术如图3-3（a）所示，其中4帧的大小分别为5、5、8和8个字节。

这个算法的问题在于计数值有可能因为一个传输错误而被弄混。例如，如果第2帧中的计数值5由于一个比特反转而变成了7，如图3-3（b）所示，则接收方就会失去同步，它再也不可能找到下一帧的正确起始位置。即使校验和不正确，接收方知道该帧已经被损坏，它仍然无法知道下一帧从哪里开始。在这种情况下，给发送方发回一个帧，要求重传也无济于事，因为接收方并不知道应该跳过多少个字节才能到达重传的开始处。正是由于这个原因，字节计数方法本身很少被使用。

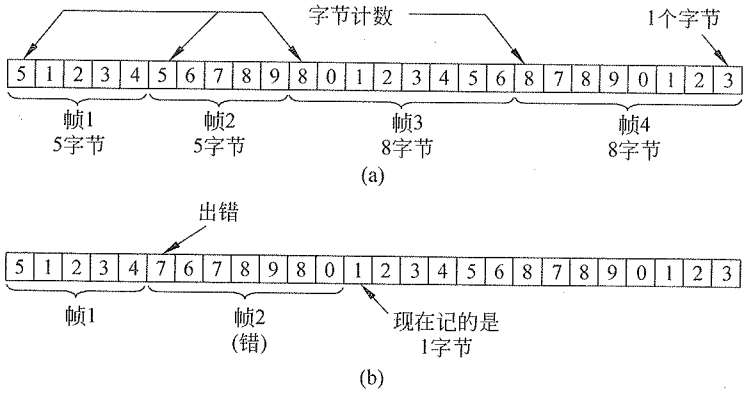


图3-3 字节流

（a）没有错误；（b）有一个错误

第二种成帧方法考虑到了出错之后的重新同步问题，它让每个帧用一些特殊的字节作为开始和结束。这些特殊字节通常都相同，称为**标志字节**（flag byte），作为帧的起始和结束分界符，如图3-4（a）中的FLAG所示。两个连续的标志字节代表了一帧的结束和下一帧的开始。因此，如果接收方丢失了同步，它只需搜索两个标志字节就能找到当前帧的结束和下一帧的开始位置。

然而，还有问题必须要解决。当标志字节出现在数据中时，尤其是当传输二进制数据（比如照片或歌曲）时，这种情景往往会严重干扰到帧的分界。有一种方法可以解决这个问题，发送方的数据链路层在数据中“偶尔”出现的每个标志字节的前面插入一个特殊的转义字节（ESC）。因此，只要看它数据中标志字节的前面有没有转义字节，就可以把作为帧分界符的标志字节与数据中出现的标志字节区分开来。接收方的数据链路层在将数据传递

给网络层之前必须删除转义字节。这种技术就称为**字节填充**（byte stuffing）。

当然，接下来的问题就是：如果转义字节也出现在数据中，那该怎么办？答案是同样用字节填充技术，即用一个转义字节来填充。在接收方，第一个转义字节被删除，留下紧跟在它后面的数据字节（或许是另一个转义字节或者标志字节）。图 3-4（b）给出了一些例子。在所有情况下，去掉填充字节之后递交给网络层的字节序列与原始的字节序列完全一致。我们仍然可以通过搜索两个标志字节来定位帧的边界，无须顾虑撤销转义字节的原意。

图 3-4 中描述的字节填充方案是 **PPP 协议**（Point-to-Point Protocol）使用的略微简化形式，该协议通常用在通信链路上传送数据包。我们将在本章后面讨论 PPP 协议。

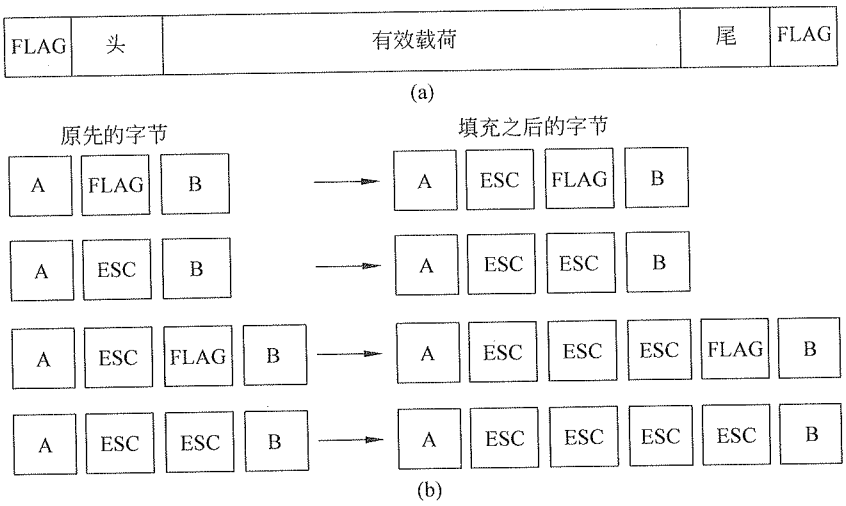


图 3-4

（a）由标志字节分界的帧；（b）字节填充之前和之后的字节序列示例

第三种区分比特流边界的方法考虑了字节填充的缺点，即只能使用 8 比特的字节。帧的划分可以在比特级完成，因而帧可以包含由任意大小单元（而不是只能以 8 比特为单元）组成的二进制比特数。这种方法是为曾经非常流行的 HDLC（高级数据链路控制）协议而开发的。每个帧的开始和结束由一个特殊的比特模式，01111110 或十六进制 0x7E 标记。这种模式是一个标志字节。每当发送方的数据链路层在数据中遇到连续五个 1，它便自动在输出的比特流中填入一个比特 0。这种比特填充类似于字节填充，在数据字段的标志字节之前插入一个转义字节到出境字符流中。比特填充还确保了转换的最小密度，这将有助于物理层保持同步。正是由于这个原因，USB（通用串行总线）采用了比特填充技术。

当接收方看到 5 个连续入境比特 1，并且后面紧跟一个比特 0，它就自动剔除（即删除）比特 0。比特填充和字节填充一样，对两台计算机上的网络层是完全透明的。如果用户数据中包含了标志模式 01111110，这个标志传输出去的是 011111010，但在接收方内存中存储还是 01111110。图 3-5 给出了一个比特填充的例子。

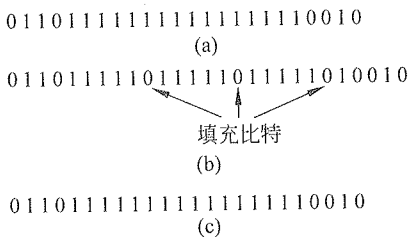


图 3-5　比特填充

（a）原始数据；（b）出现在线路上的数据；（c）存储在接收方内存中的数据

有了比特填充技术，两帧之间的边界可以由标志模式明确区分。因此，如果接收方失去了它的接收轨迹，它所要做的只是扫描输入比特流，找出其中的标志序列；因为这些标志只可能出现在帧的边界，而决不会出现在帧内的数据中。

采用比特填充和字节填充的一个副作用是一帧的长度现在要取决于它所携带的数据内容。例如，如果数据中没有标记字节，100 个字节或许被一个大约长为 100 字节的帧所携带。然而，如果数据完全由标志字节组成，每个标志字节都要被转义，那么最后发送出去帧的长度就变成大约 200 个字节。采用比特填充技术，帧的长度增幅大约为 12.5%，因为每个字节增加 1 个比特。

成帧的最后一种方法是一条使用物理层的捷径。我们在第 2 章看到比特编码成信号通常包括一些冗余比特，以便帮助接收器同步接收。这种冗余意味着一些信号将不会出现在常规数据中。例如，在 4B/5B 线性编码模式下，4 个数据位被映射成 5 个信号比特，通过这种方法确保线路上的信号有足够的跳变。这意味着 32 个可能的信号中有 16 个是不会被使用的。我们可以利用这些保留的信号来指示帧的开始和结束。实际上，我们使用“编码违法”来区分帧的边界。这种方案的优点在于，因为这些用作分界符的信号是保留不用的，所以很容易通过它们找到帧的开始和结束，而且不再需要填充数据。

许多数据链路协议为安全起见综合使用了这些方法。以太网和 802.11 使用了共同的分界模式，即用一个定义良好的比特模式来标识一帧的开始，该比特模式称为**前导码**（preamble）。这种定界模式可能很长（802.11 典型使用 72 位），目的是让接收方准备接收输入的数据包。前导码之后是头的长度字段（即计数），这个字段将被用来定位帧的结束处。

3.1.3 差错控制

解决了如何标识每一帧的起始和结束位置之后，我们现在来看下一个问题：如何确保所有的帧最终都被传递给目标机器的网络层，并且保持正确的顺序。现在假设接收方可以知道它收到的帧包含了正确的或者错误的信息（我们将在 3.2 节考察用于检测和纠正传输错误的编码）。对于无确认的无连接服务，不管发出去的帧是否正确抵达目标机器，发送方只要把出境帧留存就可以了。但是对于可靠的、面向连接的服务，这样做肯定还远远不够。

确保可靠传递的常用方法是向发送方提供一些有关线路另一端状况的反馈信息。通常情况下，协议要求接收方发回一些特殊的控制帧，在这些控制帧中，对于它所接收到的帧进行肯定的或者否定的确认。如果发送方收到了关于某一帧的肯定确认，那么它就知道这帧已经安全地到达了。另一方面，否定的确认意味着传输过程中产生了错误，所以这帧必须重传。

更为复杂的是存在这样的可能性，有时候由于硬件的问题，一个帧被完全丢失了（比如一个突发噪声）。在这种情况下，接收方根本不会有任何反应，因为它没有根据做出反应。类似地，如果确认帧丢失，发送方也不知道该如何处理。显然，如果在一个协议中，发送方发出了一帧之后就等待肯定的或者否定的确认，那么，若由于硬件故障或通信信道出错等原因而丢失了某一帧，则发送方将永远等待下去。

这种可能性可以通过在数据链路层中引入计时器来解决。当发送方发出一帧时，通常

还要启动一个计时器。该计时器的超时值应该设置得足够长，以便保证在正常情况下该帧能够到达接收方，并且在接收方进行处理后再将确认返回到发送方。一般情况下，在计时器超时前，该帧应该被正确地接收，并且确认帧也被传了回来。这种情况下，计时器被取消。

然而，如果帧或者确认被丢失，则计时器将被触发，从而警告发送方存在一个潜在的问题。一种显然的解决方案是重新发送该帧。然而，当有的帧被发送了多次之后，可能会出现这样的危险：接收方将两次或者多次接收到同一帧，并且多次将它传递给网络层。为了避免发生这样的情形，一般有必要给发送出去的帧分配序号，这样接收方可以根据帧的序号来有效区分原始帧和重传帧。

管理好计时器和序号，以便保证每一帧最终都恰好一次地被传递给目标机器的网络层，这是数据链路层（以及上层）工作的重要组成部分。在本章后面，我们将通过一系列复杂性逐渐增加的例子，来考察如何做好计时器和序号的管理工作。

3.1.4　流量控制

在数据链路层（以及更高的各层）中，另一个重要的设计问题是如果发送方发送帧的速度超过了接收方能够接受这些帧的速度，发送方该如何处理。当发送方运行在一台高速并能量强大的计算机上，而接收方运行在一台慢速并且低端机器上时，这种情况很容易发生。一种常见的场景是一个智能手机向一个超强服务器请求一个 Web 页面。这就像突然打开了消防栓一样，大量的数据涌向可怜无助的手机，直到它被彻底淹没。即使传输过程不会出错，接收方也无法以数据到来的速度那样快地处理持续到来的帧，此时必然会丢弃一些帧。

很显然，必须要采取某种措施来阻止这种情况发生。常用的办法有两种。第一种方法是**基于反馈的流量控制**（feedback-based flow control），接收方给发送方返回信息，允许它发送更多的数据，或者至少告诉发送方自己的情况怎么样。第二种方法是**基于速率的流量控制**（rate-based flow control），使用这种方法的协议有一种内置的机制，它能限制发送方传输数据的速率，而无须利用接收方的反馈信息。

在本章，我们将学习基于反馈的流量控制方案，因为基于速率的方案仅在传输层（第 5 章）的一部分中可见，而基于反馈的方案则可同时出现在链路层和更高的层次。后者在近来较为常见，在这种情况下，链路层硬件设计的运行速度足够快到不会造成丢帧。例如，作为链路层硬件实现的**网络接口卡**（NIC，Network Interface Cards），有时声称能以“线速”运行，这意味着它们能以帧到达的速度来处理帧。因而，任何过载不再是链路层的问题，它们必须由高层来处理。

基于反馈的流控制方案有许多种，但是绝大多数使用了同样的基本原理。协议包含了许多定义良好的规则，这些规则规定了发送方什么时候可以发送下一帧。这些规则通常在没有得到接收方许可（隐式或者显式的）之前，禁止继续发送帧。例如，当建立一个连接时，接收方可能会这样说：“你现在可以给我发送 n 个帧，但是在发送完 n 个帧之后就别再发送，直到我告诉你可以继续发送。”稍后我们将讨论这些细节。

3.2 差错检测和纠正

正如我们在第2章中所看到的那样，通信信道有许多不同的特征。某些信道，例如电话系统中的光纤，其错误率很低，因而很少发生传输错误。但是其他信道，尤其是无线链路和老化的本地回路错误率高出光纤好几个数量级。对于这些信道而言，传输出错是常态。从性能的角度来看，这些错误不能在合理的成本开销内彻底解决。因此结论是传输错误非常普遍。我们必须知道如何处理传输错误。

网络设计者针对错误处理已经研究出两种基本策略。这两种策略都在发送的数据中加入冗余信息。一种策略是在每一个被发送的数据块中包含足够多的冗余信息，以便接收方能据此推断出被发送的数据是什么。另一种策略也是包含一些冗余信息，但这些信息只能让接收方推断出是否发生了错误（而推断不出哪个发生了错误），然后接收方可以请求发送方重传。前一种策略使用了**纠错码**（error-correcting code），后一种策略使用了**检错码**（error-detecting code）。使用纠错码的技术通常也称为**前向纠错**（FEC，Forward Error Correction）。

这里的每一项技术都占据着不同的生态位置。在高度可靠的信道上（比如光纤），较为合算的做法是使用检错码，当偶尔发生错误时只需重传整个数据块。然而，在错误发生很频繁的信道上（比如无线链路），更好的做法是在每一个数据块中加入足够的冗余信息，以便接收方能够计算出原始的数据块。FEC被用在有噪声的信道上，因为重传的数据块本身也可能像第一次传输那样出错。

这些编码的一个关键考虑是可能发生的错误类型。无论是纠错码还是检错码都无法处理所有可能的传输错误，因为提供保护措施的冗余比特很可能像数据比特一样出现错误（可危及它们的保护作用）。如果信道给予冗余比特的待遇好于数据比特那当然好，可惜事实并非如此。对信道而言，它们都是同样的比特。这意味着为了避免漏检错误，编码必须强大到足以应付预期的错误。

错误模型有两种。在第一种错误模型中，偶尔出现的极端热噪声快速淹没了信号，引起孤立的单个比特错误。在另一种错误模型中，传输错误往往呈现突发性而不以单个形式出现，这种错误源自物理过程，比如无线信道上的一个深衰落，或者有线信道上的瞬态电气干扰。

两种错误模型实际上造成的问题以及处理方式有不同的权衡。突发的错误相比单个比特错误有自己的优势，也有不足。从优势方面来看，计算机数据总是成块发送。假设数据块大小为1000个比特，误差率为每比特0.001。如果错误是独立的，大多数块将包含一个错误。但如果错误以100个比特的突发形式出现，则平均来说100块中只有一块会受到影响。突发错误的缺点在于当它们发生时比单个错误更难以纠正。

同时还存在着其他类型的错误。有时候，一个错误的位置可以获得，或许因为物理层接收到的一个模拟信号远离了0或1的预期值，因而可以宣布该比特被丢失。这种情形称为**擦除信道**（erasure channel）。擦除信道比那些把比特值翻转的信道更易于纠错，因为即使某个比特被丢失，至少我们还能知道哪个比特出了错。然而，我们往往不能从信道的擦

除性质上受益。

下面我们将同时考察纠错码和检错码。请记住两点。首先，我们在链路层讨论这些编码有客观因素，因为这里是面临数据可靠传输相关问题的首要地方。然而，这些编码方案被广泛使用的根本原因在于可靠性是整个系统所关注的问题。纠错码也会出现在物理层，特别是有噪声干扰的信道，同时还会出现在更高的层次，特别是实时流媒体应用和内容分发应用。检错码更是经常被用在链路层、网络层和传输层。

第二点要记住的是差错编码是应用的数学。除非特别熟悉伽罗瓦（Galois）领域或稀疏矩阵的性质，否则，应该从可靠的来源获得性质更优良的编码，而不是自己设计编码方案。事实上，这正是为何很多协议标准一次又一次采用了相同的编码方法。在下面的材料中，我们将学习一个简单的编码方法，然后再简要描述先进的编码方法。这样，我们可以从简单编码中来理解如何权衡，并且通过先进编码来讨论实际使用的编码方案。

3.2.1　纠错码

我们将考察以下 4 种不同的纠错编码：

（1）海明码。

（2）二进制卷积码。

（3）里德所罗门码。

（4）低密度奇偶校验码。

上述所有编码都将冗余信息加入到待发送的信息中。一帧由 m 个数据位（即信息）和 r 个冗余位（即校验）组成。在**块码**（block code）中，r 个校验位是作为与之相关的 m 个数据位的函数计算获得的，就好像在一张大表中找到 m 位数据对应的 r 校验位。在**系统码**（systematic code）中，直接发送 m 个数据位，然后发出 r 个校验位，而不是在发送前对它们进行编码。在**线性码**（line code）中，r 个校验位是作为 m 个数据位的线性函数被计算出来的。异或（XOR）或模 2 加是函数的流行选择，这意味着编码过程可以用诸如矩阵乘法或简单逻辑电路来完成。除非另有说明，我们在本节中考察的是线性码、系统块状码。

令数据块的总长度为 n（即 n=m+r）。我们将此描述为（n, m）码。一个包含了数据位和校验位的 n 位单元称为 n 位**码字**（codeword）。**码率**（code rate）或者简单地说速率，则定义为码字中不包含冗余部分所占的比例，或者用 m/n 表示。实际上这个码率变化很大。在一个有噪声信道上码率或许是 1/2，在这种情况下接收方所收到的信息中有一半是加入的冗余位；而在高品质的信道上码率接近 1，只有少数的校验位被添加到一个大块消息中。

为了理解发生传输错误后如何进行处理，有必要先来看一看错误到底是什么样的。给定两个被发送或接收的码字，比如说 10001001 和 10110001，完全可能确定这两个码字中有多少个对应位是不同的。此时，上述两个码字有 3 位不同。为确定有多少个不同位，只需 XOR 两个码字，并且计算结果中 1 的个数。例如：

$$\begin{array}{r} 10001001 \\ \underline{10110001} \\ 00111000 \end{array}$$

两个码字中不相同的位的个数称为**海明距离**（Hamming distance）（Hamming，1950）。

它的意义在于，如果两个码字的海明距离为 d，则需要 d 个 1 位错误才能将一个码字转变成另一个码字。

给定计算校验位的算法，完全可以构建一个完整的合法码字列表，然后从这个列表中找出两个具有最小海明距离的码字。这个距离就是整个编码的海明距离。

在大多数数据传输应用中，所有 2^m 种可能的数据报文都是合法的；但是，根据检验位的计算方法，并非所有 2^n 种可能的码字都会被用到。事实上，对于 r 校验位，可能的报文中只有很少一部分 $2^m/2^n$ 或 $1/2^r$ 是合法的码字。正是这种空间稀疏方法，即报文被嵌入到码字空间中，使得接收方能检测并纠正错误。

块码的检错和纠错特性跟它的海明距离有关。为了可靠地检测 d 个错误，需要一个距离为 d+1 的编码方案，因为在这样的编码方案中，d 个 1 位错误不可能将一个有效码字改变成另一个有效码字。当接收方看到一个无效码字时，它就知道发生了传输错误。类似地，为了纠正 d 个错误，需要一个距离为 2d+1 的编码方案，因为在这样的编码方案中，合法码字之间的距离足够远，即使发生了 d 位变化，结果也还是离它原来的码字最近。这意味着在不太可能有更多错误的假设下，可以唯一确定原来的码字，从而达到纠错的目的。

看一个纠错码实例，考虑一个只有下列 4 个有效码字的编码方案：

0000000000, 0000011111, 1111100000, 1111111111

该编码方案的距离是 5，这意味着它可以纠正 2 个错误或者检测双倍的错。如果接收到码字 0000000111 并且期望只有单个或者 2 个错误，则接收方知道原始的码字一定是 0000011111。然而，如果发生了三个错误，0000000000 变成了 0000000111，则以上编码就无法正确地纠正错误了。另外，如果我们期望所有这些错误都会发生，我们也可以检测出它们。只要没有收到合法的码字，就必然发生了错误。很明显在这个例子中，我们不能同时纠正 2 个错误和检测 4 个错误，因为这需要我们以两种不同的方式来解释接收到的码字。

在我们的例子中，解码的任务就是找出最接近接收码字的合法码字。不幸的是，大多数情况下全部码字都将作为候选被评估，这是一件非常耗时的搜索。相反，实际的代码被设计成允许使用快捷方式找出最有可能的原始码字。

设想我们要设计一种编码方案，每个码字有 m 个消息位和 r 个校验位，并且能够纠正所有的单个错误。对于 2^m 个合法消息，任何一个消息都对应有 n 个非法的码字，它们与该消息的距离为 1。这些非法的码字可以这样构成：将该消息对应的合法码字的 n 位，逐个取反，可以得到 n 个距离为 1 的非法码字。因此，每个 2^m 中的合法消息需要 n+1 个位模式来标识它们。由于总共只有 2^n 个位模式，所以，我们必须有 $(n+1)2^m \leqslant 2^n$。由 n=m+r，这个要求变成了

$$(m+r+1) \leqslant 2^r \qquad (3\text{-}1)$$

在给定 m 的情况下，这个条件给出了纠正单个错误所需要的校验位数的下界。

事实上这个理论下限可使用海明方法（1950）获得。在海明码中，码字的位被连续编号，从最左端的位开始，紧跟在右边的那位是 2，依次从左到右编号。2 的幂次方的位（1，2，4，8，16 等）是校验位，其余位（3，5，6，7，9 等）用来填充 m 个数据位。这种模式如图 3-6 所示的（11,7）海明码，其中 7 个数据位和 4 个校验位。每一个校验位强制进行模 2 加，或对某些位的集合，包括其本身进行偶（或奇）校验。一位可能被包括在几个校验位的计算中。若要查看在数据 k 位上的校验位，必须将 k 改写成 2 的幂之和。例

如，11 =1 +2 +8 和 29 =1 +4 +8 +16。校验某一位只需要检查那些覆盖了该位的校验位（例如，校验 1、2 和 8 位就可确定 11 位是否出错）。在这个例子中，我们采用偶校验计算 ASCII 字母“A”的校验和。

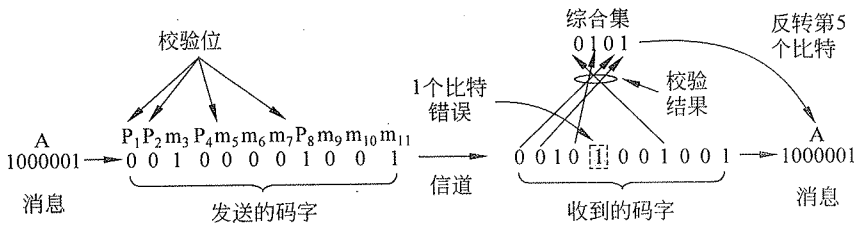


图 3-6　纠正单个错位的（11, 7）海明码示例

这种结构给出了海明距离为 3 的编码，意味着它可以纠正单个错误（或检测 2 个错误）。针对信息位和校验位小心编号的原因在解码的处理过程中表现得非常明显。当接收到一个码字，接收机重新计算其校验位，包括所收到的校验位，得到的计算结果我们称之为校验结果。如果校验位是正确的，对于偶校验和而言，校验结果应该是零。在这种情况下，码字才被认为是有效的，从而可以接收。

然而，如果校验结果不是全零，则意味着检测到了一个错误。校验结果的集合形成的错误综合集，可用来查明和纠正错误。在图 3-6 中，信道上发生了 1 位错误，因此分别针对 k =8，4，2，1 的校验结果是 0，1，0 和 1。由此得出的综合集为 0101 或 4 + 1 = 5。按照设计方案，这意味着第五位有误。把不正确的位（这可能是一个校验位或数据位）取反，并丢弃校验位就得到正确的消息——ASCII 字符“A”。

海明距离对理解块码是有价值的，而且海明码还被用在纠错存储器中。然而，大多数网络使用了更强大的编码。我们将考察的第二个编码是**卷积码**（convolutional code）。这是我们讨论的编码方法中唯一不属于块码的编码。在卷积码中，编码器处理一个输入位序列，并生成一个输出位序列。在块码中没有自然消息大小或编码边界。输出取决于当前的输入和以前的输入。也就是说，编码器有内存。决定当前输出的以前输入位数称为代码的**约束长度**（constraint length）。卷积码由它们的速率和约束长度来标识。

卷积码已被广泛应用于实际部署的网络中，例如，它已经成为 GSM 移动电话系统的一部分，在卫星通信和 802.11 中都得到应用。作为一个例子，图 3-7 给出了一个流行的卷积码。这个代码称为 NASA（美国航天局）卷积码，其 r=1/2 和 k=7。因为它是第一个被用在 1977 年的旅行者号航天飞行任务中的编码。从那以后，它被随意重用于许多其他地方，例如，已成为 802.11 的一部分。

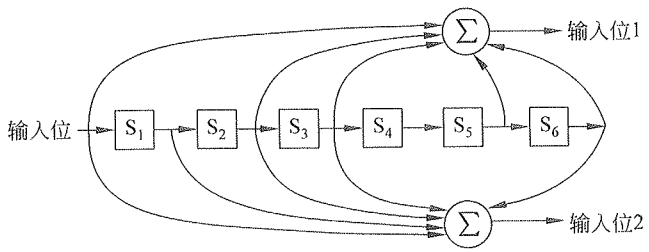


图 3-7　应用于 802.11 的 NASA 二进制卷积码

在图 3-7 中，左边每个输入位产生右边的两个输出位，输出位是输入位和内部状态的 XOR 和。由于它处理的是比特位并执行线性运算，因此是二进制的线性卷积码。又因为 1 个输入产生 2 个输出，因此码率为 1/2。这里的输出位不是简单的输入位，从而它不属于系统码。

内部状态保存在 6 个内存寄存器中。每当输入一位寄存器的值就右移一位。例如，如果输入序列为 111，初始状态是全零，则在输入第一、第二和第三位后从左到右的内部状态变化成 100000、110000 和 111000。对应的输出位分别是 11、10 和 01。这个过程需要 7 次移位才能完全清空输入，从而不影响输出。因此，该卷积码的约束长度是 k= 7。

卷积码的解码过程是针对一个输入位序列，找出最有可能产生观察到的输出位序列（包括任何错误）。对于较小值的 k，一种广泛使用的算法是由 Viterbi 开发的（Forney，1973）。该算法逐个检查观察到的序列，记住每一步和输入序列的每个可能内部状态，即输入序列产生观察序列可能产生的错误。最终其中那个具有最少错误的输入序列就是最有可能的消息。

卷积码实际上已经非常流行，它之所以很容易被采纳的一个因素在于解码 0 或 1 的不确定性。例如，假设–1 V 表示逻辑 0，+1 V 表示逻辑 1，接收方可能接收到的 2 位分别是 0.9 V 和–0.1 V。卷积码不是简单地将这些信号绝对映射成逻辑 1 和 0，而是把 0.9 V 看成“很可能是 1”，把–0.1 V 看成“很可能是 0”，从而最终获得正确的整个序列。Viterbi 算法的扩展适用于这些不确定因素，因而能提供更强的纠错功能。这种带有一位不确定性的工作方法称为**软判决解码**（soft-decision decoding）。相反，在执行纠错之前就决定了每个位是 0 或 1 的工作方法称为**硬判决解码**（hard-decision decoding）。

我们将描述的第三种纠错码是**里德所罗门码**（Reed-Solomon code）。像海明码一样，里德所罗门码是线性块码，而且往往也是系统码。但与海明码不同的是，里德所罗门码对 m 位符号进行操作，而不是针对单个位处理。当然，这里需要更多的数学参与，因此我们将用类比的方式来描述该编码方案。

里德所罗门码基于这样的事实：每一个 n 次多项式是由 n+1 点唯一确定的。例如，一条具有 ax+b 形式的线由两个点所决定。同一条线上的额外点都是冗余的，这将有助于纠错。可以想象有两个数据点代表了一条线，并且我们给这两个数据点额外加上两个校验点，该两个校验点选自同一条线。如果收到的其中一个点出现错误，我们仍然可以通过接收点的拟合线来恢复这个数据点。三个点将处在同一条直线上，而出错的那个点不在这条线上。只要找到这条线，我们就可以纠正错误。

里德所罗门码实际上被定义成一个在有限域上操作的多项式，但工作方式相同。对于 m 位符号而言，码字长 2^m-1 个符号。一种流行的选择是 m=8，这样符号就是字节。因此，一个码字为 255 个字节长。（255，233）码被广泛使用，它在 233 个数据符号上增加了 32 个冗余符号。带有纠错功能的解码算法由 Berlekamp 和 Massey 开发，它能有效执行中等长度的解码（Massy，1969）。

里德所罗门码得到广泛应用的原因还在于其强大的纠错性能，尤其针对突发错误。它们被用在 DSL、线缆上的数据通信、卫星通信、最无处不在的 CD、DVD 和蓝光光盘。因为它们基于 m 位符号，因此一位错误和 m 位突发错误都只是作为一个出错的符号来处理。当加入 2t 个冗余符号后，里德所罗门码能够纠正传输符号中的任意 t 个错误。这意味着，

例如在（255，233）中，由于有 32 个冗余符号，因此可以纠正多达 16 个符号错误。因为符号是连续的，并且每个 8 位长，所以可以纠正高达 128 位的突发错误。如果错误模型是擦除的（例如，一张 CD 消除了一些符号划痕）则情况更好。在这种情况下，高达 2t 个错误都可以得到更正。

里德所罗门码通常与其他编码结合在一起使用，如卷积码。这种想法的依据在于：卷积码在处理孤立的比特错误时很有效，但当接收到的比特流中有太多的错误（和突发错误类似），卷积码就无法处理了。在卷积码内加入里德所罗门码，因为里德所罗门码可以横扫突发错误，因此两者的结合就能将纠错任务完成得非常好，综合起来的编码模式对单个错误和突发错误都有良好的保障作用。

我们考察的最后一个纠错码是**低密度奇偶校验码**（LDPC，Low-Density Parity Check）。LDPC 码是线性块码，由 Robert Gallagher 在他的博士论文中首次提出（Gallagher，1962）。像大多数学位论文一样，它们很快被人遗忘，直到 1995 年计算能力的进步使得它们重新走向实际应用。

LDPC 码中的每个输出位由一小部分的输入位形成。这样使得编码可以用一个 1 的密度很低的矩阵来表示，这也是编码名称的由来。接收到的码字通过一个近似算法解码获得，该算法通过迭代不断改进接收到的数据与合法码字的最佳匹配。如此来纠正错误。

LDPC 码比较适用于大块数据，而且具有出色的纠错能力，因而性能优于其他许多编码（包括我们曾经考察过的那些）。正是基于这个原因，它们迅速被新的协议所采纳，成为数字视频广播、万兆以太网、电力线网络，以及最新版本 802.11 标准的一部分。我们希望它们出现在未来的更多网络中。

3.2.2　检错码

纠错编码被广泛应用于无线链路。众所周知，相比光纤，无线链路嘈杂不堪而且容易出错，如果没有纠错码，将很难从该链路获得任何信息。然而，光纤或高品质铜线的错误率要低得多，因此对于偶尔出现的错误采用差错检测和重传的处理方式通常更加有效。

我们将考察 3 种检错码。这些检错码都是线性的系统块码：

（1）奇偶。

（2）校验和。

（3）循环冗余校验（CRC）。

为了看清楚检错码如何比纠错码更有效，考虑第一个检错码——把单个奇偶校验位附加到数据中。奇偶位的选择原则是使得码字中比特 1 的数目是偶数（或奇数）。这样处理等同于对数据位进行模 2 加或异或操作来获得奇偶位。例如，当以偶校验方式发送 1011010 时，在数据末尾添加一位成为 10110100。若采用奇校验方式发送 1011010 时，则结果为 10110101。具有单个校验位的编码具有码距 2，因为任何 1 位错误都将使得码字的奇偶校验码出错。这意味着奇偶码可以检测出 1 位错误。

考虑这样的信道，其上发生的错误都是孤立的，并且每个比特的出错率是 10^{-6}。这看似微小的错误率，对长距离的线缆已经是最好的条件了，但对错误检测却是一个挑战。典型局域网链路的误码率大约为 10^{-10}。令数据块大小为 1000 位。为了提供相应的纠错功能，

我们根据等式（3-1）可知需要10个校验位。因此1兆大的数据块将需要10 000个校验位。但是，如果只为检测出该块数据中是否存在1位错误，每块数据仅一个校验位就足够了。如此一来，每1000个数据块中才有一块出现错误，只需要重传额外的一块（1001位）即可修复错误。每1兆数据用于错误检测和重传的总开销只有2001位，相比海明码需要的10 000位显然效率高得多。

这种校验方案的困难在于单个校验位只能可靠地检测出1位错误。如果数据块因一个长的突发错误造成严重乱码，那么这种错误被检测出来的概率只有0.5，显然这是令人难以接受的。如果发送的每个数据块作为一个n位宽和k位高的长方形矩阵来处理，则检测出错误的概率可望得到很大提高。现在，如果我们为每一行计算和发送一个校验位，只要每一行最多只有一个错误发生，那么我们就能可靠地检测出k位错误。

然而，我们还可以做点其他事情来提高针对突发错误的更好保护：改变计算校验位的次序，即以不同于数据位发送的次序来计算校验位。这种处理方式就是所谓的交错校验（interleaving）。在这种情况下，我们将为n列中的每列计算校验位，按k行发送全部的数据位，发送次序是从上到下发送每一行，行内数据位通常按从左到右的次序发送。在最后一行，发送n个校验位。这种传输顺序如图3-8所示，其中n=7和k=7。

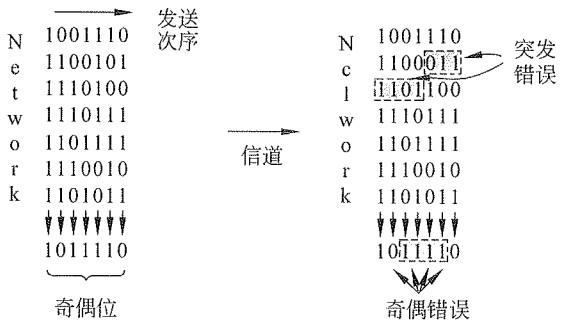


图3-8 检测一个突发错误的交错奇偶校验位

交错校验是一种将检测（或纠正）单个错误的编码转换成能检测（或纠正）突发错误的通用技术。在图3-8中，当发生一个长度为n=7的突发错误，出错的位恰好分散在不同的列（突发错误并不隐含着所有的位都出错；它只意味着至少第一位和最后一位错误。如图3-8所示4位错误分布在7位范围内）。n列中的每一列至多只有一位受到影响，因此这些列中的校验位将能检测到该错误。这种方法对于nk长度的数据块使用了n个校验位就能检测出一个长度小于等于n的突发错误。

然而，一个长度为n+1的突发错误将被遗漏，如果第一位和最后一位反转，所有其他位都正确，这样的错误无法检测出来。如果数据块被一个长突发错误或多个短突发错误所扰乱，该n列中任何一列有正确校验位的概率偶尔有0.5，所以一个不该接受而被接收的坏块的概率是2^{-n}。

第二类检错码是校验和，与一组奇偶校验位密切相关。**校验和**（checksum）这个词通常用来指与信息相关的一组校验位，不管这些校验位是如何计算出来的。一组奇偶校验位是校验和的一个例子。然而，还有其他种类的校验和，强大的校验和基础是对消息中的数据位进行求和计算。校验和通常放置在消息的末尾，作为求和功能的补充。这样一来，通

过对整个接收到的码字（包含了数据位和校验和）进行求和计算就能检测出错误。如果计算结果是零，则没有检测出错误。

校验和的一个例子是 16 位的 Internet 校验和，作为 IP 协议的一部分用在所有 Internet 数据包中（Braden 等，1988）。该校验和是按 16 位字计算得出的消息位总和。由于此方法针对字而不是像奇偶校验那样针对位进行操作，因此奇偶校验没能检测出的错误此时仍然影响着校验和，从而能被检测出来。例如，如果两个字的最低位都从 0 错误地变成了 1，其上的奇偶校验将无法检测到这个错误。但是，两个 1 增加到 16 位校验和中将产生不同的结果。这个错误就能被检测出来。

Internet 校验和是以补码运算而非 2^{16} 模加运送得到的。在补码运算中，负数是其正数的按位补。现代计算机采用双补算法（two's complement），负数是该数的补码加一。在双补计算机中，一个数的补码等价于模 2^{16} 加，并且任何高序位溢出被放回到低序位上。该算法为校验和位提供了更均匀的数据覆盖面。否则，两个高序位可能因相加、溢出而被丢失，从而对校验和没有任何作用。这种双补算法还有另一个好处，0 的补码有两种表示方法：全 0 和全 1。这就允许用一个值（例如，全 0）表示没有校验和而不需要用额外一个字段作特别的说明。

几十年来，人们一直有这样的假设，即计算校验和所针对的帧包含了随机位。校验和算法的所有分析都是基于这样的假设而进行。但是由（Partridge 等，1995）检查的真实数据表明这种假设是十分错误的。因此，在某些情况下未发现的错误比以前想象得更为常见。

尤其是 Internet 校验和，它有效而简单；但在某些情况下提供的保护很弱，正是因为它只是一个简单的和。它检测不出 0 数据的增加或删除，也检测不出消息中被掉换的那部分，而且对两个数据包拼接起来的消息只有弱保护作用。这些错误在随机过程中似乎不太可能发生，但恰恰是一种能发生在有缺陷硬件上的错误。

一个更好的选择是 Fletcher 校验和（Fletcher，1982）。它包括一个位置组件，将数据和其位置的乘积添加到总和中。这样能对数据位置的变化提供更强大的检测作用。

尽管前面两种方案对高层而言有时候已足够，但实际上，链路层广泛使用的是第三种更具检错能力的方法：**循环冗余校验码**（CRC，Cyclic Redundancy Check），也称为**多项式编码**（polynomial code）。多项式编码的基本思想是：将位串看成是系数为 0 或 1 的多项式。一个 k 位帧看作是一个 k-1 次多项式的系数列表，该多项式共有 k 项，从 x^{k-1} 到 x^0。这样的多项式认为是 k-1 阶多项式。高次（最左边）位是 x^{k-1} 项的系数，接下来的位是 x^{k-2} 项的系数，依此类推。例如，110001 有 6 位，因此代表了一个有 6 项的多项式，其系数分别为 1、1、0、0、0 和 1：即 $1x^5+1x^4+0x^3+0x^2+0x^1+1x^0$。

多项式的算术运算遵守代数域理论规则，以 2 为模来完成。加法没有进位，减法没有借位。加法和减法都等同于异或。例如：

10011011	00110011	11110000	01010101
−11001010	+11001101	−10100110	−10101111
01010001	11111110	01010110	11111010

长除法与二进制中的除法运算一样，只不过减法按模 2 进行。如果被除数与除数有一样多的位，则该除数要“进入到”被除数中。

使用多项式编码时，发送方和接收方必须预先商定一个**生成多项式**（generator

polynomial）G(x)。生成多项式的最高位和最低位系数必须是1。假设一帧有m位，它对应于多项式M(x)，为了计算它的CRC，该帧必须比生成多项式长。基本思想是在帧的尾部附加一个校验和，使得附加之后的帧所对应的多项式能够被G(x)除尽。当接收方收到了带校验和的帧之后，它试着用G(x)去除它。如果有余数的话，则表明传输过程中有错误。

计算CRC的算法如下：

（1）假设G(x)的阶为r。在帧的低位端加上r个0位，使得该帧现在包含m+r位，对应多项式为$x^rM(x)$。

（2）利用模2除法，用对应于G(x)的位串去除对应于$x^rM(x)$的位串。

（3）利用模2减法，从对应于$x^rM(x)$的位串中减去余数（总是小于等于r位）。结果就是将被传输的带校验和的帧。它的多项式不妨设为T(x)。

图3-9显示了采用生成多项式为$G(x)=x^4+x+1$计算帧1101011111校验和的情形。

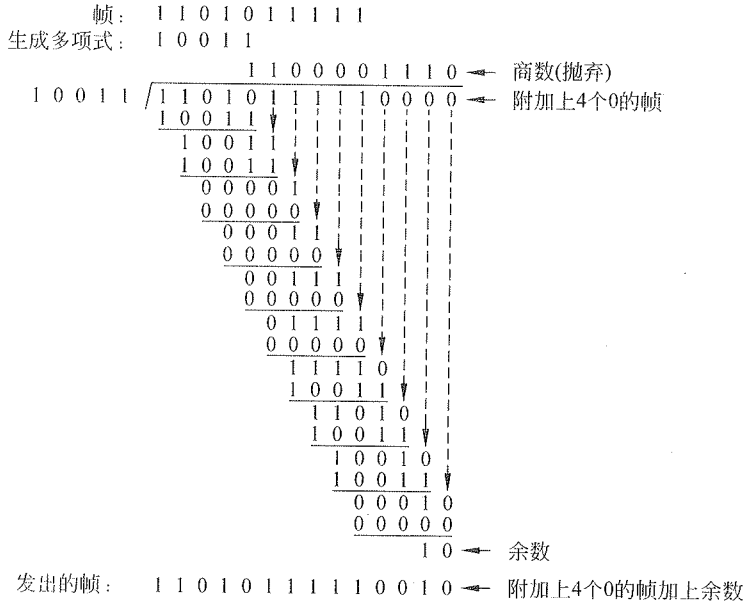


图3-9 CRC计算示例

显然，T(x)可以被G(x)除尽（模2）。在任何一种除法中，如果将被除数减掉余数，则剩下的差值一定可以被除数除尽。例如，在十进制中，如果210 278被10 941除，则余数为2399。从210 278中减去2399，得到207 879，它可以被10 941除尽。

现在我们来分析这种方法的能力。什么样的错误能被检测出来？想象一下在传输过程中发生了一个错误，因此接收方收到的不是T(x)，而是T(x)+E(x)，其中E(x)中对应的每位1都变反了。如果E(x)中有k个1，则表明发生了k个1位错误。单个突发错误可以这样描述：初始位是1，然后是0和1的混合，最后一位也是1，所有其他位都是0。

接收方在收到了带校验和的帧之后，用G(x)来除它；也就是说，接收方计算[T(x) + E(x)]/G(x)。T(x)/G(x)是0，因此计算结果简化为E(x)/G(x)。如果错误恰好发生在作为因子的G(x)多项式中，那么将无法被检测到；所有其他的错误都能够检测出来。

如果只有一位发生错误，即$E(x) = x^i$，这里i决定了错误发生在哪一位上。如果G(x)

包含两项或者更多项，则它永远也不会除尽 E(x)，所以，所有的一位错误都将被检测到。

如果有两个独立的一位错误，则 $E(x) = x^i+x^j$，这里 i>j。换一种写法，E(x)可以写成 $E(x) = x^j(x^{i-j}+1)$。如果我们假定 G(x)不能被 x 除尽，则所有的双位错误都能够检测出来的充分条件是：对于任何小于等于 i-j 最大值（即小于等于最大帧长）的 k 值，G(x)都不能除尽 x^k+1。简单地说，低阶的多项式可以保护长帧。例如，对于任何 k<32 768，$x^{15}+x^{14}+1$ 都不能除尽 x^k+1。

如果有奇数个位发生了错误，则 E(x)包含奇数项（比如 x^5+x^2+1，但不能是 x^2+1）。有意思的是，在模 2 系统中，没有一个奇数项多项式包含 x+1 因子。因此，以 x+1 作为 G(x)的一个因子，我们就可以捕捉到所有包含奇数个位变反的错误情形。

最后，也是最重要的，带 r 个校验位的多项式编码可以检测到所有长度小于等于 r 的突发错误。长度为 k 的突发错误可以用 $x^i(x^{k-1}+x^{k-2}+\cdots+1)$来表示，这里 i 决定了突发错误的位置离帧的最右端的距离有多远。如果 G(x)包含一个 x^0 项，则它不可能有 x^i 因子，所以，如果括号内表达式的阶小于 G(x)的阶，则余数永远不可能为 0。

如果突发错误的长度为 r+1，则当且仅当突发错误与 G(x)一致时，错误多项式除以 G(x)的余数才为 0。根据突发错误的定义，第一位和最后一位必须为 1，所以它是否与 G(x)匹配取决于其他 r-1 个中间位。如果所有的组合被认为是等概率的话，则这样一个不正确的帧被当做有效帧接收的概率是 $1/2^{r-1}$。

同样可以证明，当一个长度大于 r+1 位的突发错误发生时，或者几个短突发错误发生时，一个坏帧被当做有效帧通过检测的概率为 $1/2^r$，这里假设所有的位模式都是等概率的。

一些特殊的多项式已经成为国际标准，其中一个被 IEEE 802 用在以太网示例中，多项式为：

$$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x^1+1$$

除了其他优良特性外，该多项式还能检测到长度小于等于 32 的所有突发错误，以及影响到奇数位的全部突发错误。20 世纪 80 年代以后它得到广泛应用。但是，这并不意味着它是最好的选择。（Castagnoli 等，1993）和（Koopman，2002）采用穷尽计算搜索，发现了最好的 CRC。这些 CRC 针对典型消息长度获得的海明距离为 6，而 IEEE 标准的 CRC-32 海明距离只有 4。

虽然计算 CRC 所需的运算看似很复杂，但在硬件上通过简单的移位寄存器电路很容易计算和验证 CRC（Pdterson&Brown，1961）。实际上，这样的硬件几乎一直被使用着。数十种网络标准采用了不同的 CRC，包括几乎所有的局域网（如以太网、802.11）和点到点链接（例如，SONET 上的数据包）。

3.3　基本数据链路层协议

为了引入协议主题，我们先从 3 个复杂性逐渐增加的协议开始。对此感兴趣的读者，可以通过 Web 网站（参见前言）获得这些协议以及后面介绍的一些协议的模拟器。在考察这些协议之前，先明确一些有关底层通信模型的基本假设是完全有必要的。

首先，我们假设物理层、数据链路层和网络层都是独立的进程，它们通过来回传递消

息进行通信。图 3-10 给出了一个通用实现。物理层进程和某些数据链路层进程运行在一个称为网络接口卡（NIC，Network Interface Card）的专用硬件上；链路层进程的其他部分和网络层进程作为操作系统的一部分运行在主 CPU 上，链路层进程的软件通常以设备驱动器的形式存在。然而，其他的实现方案也是有可能的（比如，把 3 个进程下载到一个称为网络加速器的专用硬件上运行，或者 3 个进程按照软件定义的速率运行在主 CPU）。实际上，首选的实现方式随着技术权衡每 10 年都会发生变化。无论如何，将这 3 层作为独立的进程来讨论有助于使概念更加清晰，同时也可以强调每一层的独立性。

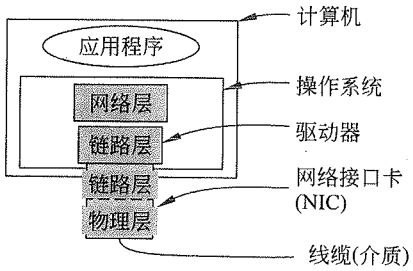


图 3-10 物理层、数据链路层和网络层的实现

另一个关键的假设是：机器 A 希望用一个可靠的、面向连接的服务向机器 B 发送一个长数据流。以后我们再考虑 B 同时向 A 发送数据的情形。假定 A 要发送的数据总是已经准备好，不必等待这些数据被生成出来。或者说，当 A 的数据链路层请求数据时，网络层总能够立即满足数据链路层的要求（这个限制后面也将被去掉）。

我们还假设机器不会崩溃。也就是说，这些协议只处理通信错误，不处理因为机器崩溃和重新启动而引起的问题。

在涉及数据链路层时，通过接口从网络层传递到数据链路层的数据包是纯粹的数据，它的每一位都将被递交到目标机器的网络层。目标机器的网络层可能会将数据包的一部分解释为一个头，这不属于数据链路层的考虑范围。

当数据链路层接收到一个数据包，它就在数据包前后增加一个数据链路层头和尾，由此把数据包封装到一个帧中（见图 3-1）。因此，一个帧由一个内嵌的数据包、一些控制信息（在头中）和一个校验和（在尾部）组成。然后，帧被传输到另一台机器上的数据链路层。我们假设有一个现成的代码库，其中过程 to_physical_layer 发送一帧，from_physical_layer 接收一帧。这些过程负责计算和附加校验和，并检查校验和是否正确（这部分工作通常由硬件完成），所以我们无须关心本节数据链路层协议的这部分内容。例如，它们或许会用到上节讨论的 CRC 算法。

刚开始时，接收方什么也不做。它只是静静地等待着某些事情的发生。在本章给出的示例协议中，我们用过程调用 wait_for_event（&event）标示数据链路层正在等待事情发生。只有当确实发生了什么事情（比如，到达了一个帧），该过程才返回。过程返回时，变量 event 说明究竟发生了什么事件。对于不同的协议，可能的事件集合也是不同的，每个协议都需要单独定义和描述事件集合。请注意，在一个更加实际的环境中，数据链路层不会像我们所建议的那样在一个严格的循环中等待事件，而是会接收一个中断；中断将使它终止当前的工作，转而处理入境帧。然而，为了简便起见，我们将忽略数据链路层内部所有并发进行的活动细节，假定它全部时间都在处理一个信道。

当一帧到达接收方，校验和被重新计算。如果计算出的帧校验和不正确（即发生了传输错误），则数据链路层会收到通知（event = cksum_err）。如果到达的帧没有受到任何损坏，数据链路层也会接到通知（event=frame_arrival），因此它可以利用 from_physical_layer 得到

该帧，并对其进行处理。只要接收方的数据链路层获得了一个完好无损的帧，它就检查头部的控制信息；如果一切都没有问题，它就将内嵌的数据包传递给网络层。无论在什么情况下，帧头部分的信息都不会被交给网络层。

为什么网络层永远得不到任何帧头的信息，理由非常简单，那就是要保持网络层和数据链路层的完全分离。只要网络层对数据链路协议和帧格式一无所知，那么当数据链路协议和帧格式发生变化时，网络层软件可以不作任何改变。每当一块新 NIC 安装在计算机上时这种情况就会发生。在网络层和数据链路层之间提供一个严格的接口可以大大地简化任务设计，因为不同层上的通信协议可以独立地发展。

图 3-11 给出了后面要讨论的许多协议公用的一些声明（C 语言）。这里定义了 5 个数据结构：boolean、seq_nr、packet、frame_kind 和 frame。boolean 是一个枚举类型，可以取值 true 和 false。seq_nr 是一个小整数，用来对帧进行编号，以便我们可以区分不同的帧。这些序号从 0 开始，一直到（含）MAX_SEQ，所以，每个需要用到序号的协议都要定义它。packet 是同一台机器上网络层和数据链路层之间，或者不同机器上的网络层对等实体

```
#define MAX_PKT 1024                        /*determines packet size in bytes */

typedef enum {false, true} boolean;         /* boolean type */
typedef unsigned int seq_nr;                /* sequence or ack numbers */
typedef struct {unsigned char data          /* packet definition */
  [MAX_PKT];} packet;
typedef enum {data, ack, nak}               /* frame_kind definition */
  frame_kind;


typedef struct{                             /* frames are transported in this layer */
  frame_kind kind;                          /* what kind of frame is it? */
  seq_nr seq;                               /* sequence number */
  seq_nr ack;                               /* acknowledgement number */
  packet info;                              /* the network layer packet */
} frame;

/* Wait for an event to happen; return its type in event. */
void wait_for_event(event_type *event);
/* Fetch a packet from the network layer for transmission on the channel. */
void from_network_layer(packet *p);
/* Deliver information from an inbound frame to the network layer. */
void to_network_layer(packet *p);
/* Go get an inbound frame from the physical layer and copy it to r. */
void from_physical_layer(frame *r);
/* Pass the frame to the physical layer for transmission. */
void to_physical_layer(frame *s);
/* Start the clock running and enable the timeout event. */
void start_timer(seq_nr k);
/* Stop the clock and disable the timeout event. */
void stop_timer(seq_nr k);
/* Start an auxiliary timer and enable the ack_timeout event. */
void start_ack_timer(void);
/* Stop the auxiliary timer and disable the ack_timeout event. */
void stop_ack_timer(void);
/* Allow the network layer to cause a network_layer_ready event. */
void enable_network_layer(void);
/* Forbid the network layer from causing a network_layer_ready event. */
void disable_network_layer(void);
/* Macro inc is expanded in-line: increment k circularly. */
#define inc(k) if (k < MAX_SEQ) k = k + 1; else k = 0
```

图 3-11　协议需要遵守的某些定义。这些定义放置在 protocol.h 文件中

之间交换的信息单元。在我们的模型中，它总是包含 MAX_PKT 个字节数据，但在实际环境中，它的长度应该是可变的。

一个帧由 4 个字段组成：kind、seq、ack 和 info，其中前 3 个包含了控制信息，最后一个可能包含了要被传输的实际数据。这些控制字段合起来称为**帧头**（frame header）。

kind 字段指出了帧中是否有数据，因为有些协议需要区分只有控制信息的帧和同时包含控制信息和数据信息的帧。seq 和 ack 分别用作序号和确认，后面还会详细描述它们的用法。数据帧的 info 字段包含了一个数据包；控制帧的 info 字段没有用处。一个更加实际的协议实现将会使用一个变长的 info 字段，而对于控制帧则完全忽略。

再次强调的是理解数据包和帧之间的关系非常重要。网络层从传输层获得一个报文，然后在该报文上增加一个网络层头，由此创建了一个数据包。该数据包被传递给数据链路层，然后被放到输出帧的 info 字段中。当该帧到达目标机器时，数据链路层从帧中提取出数据包，将数据包传递给网络层。在这样的工作方式中，网络层就好像机器一样可以直接交换数据包。

图 3-11 还列出了许多过程。这些库程序的细节与具体实现有关，它们的内部工作机制不是我们下面讨论需要关心的。前面已经提过，wait_for_event 是一个严格的循环过程，它等待事情发生。过程 to_network_layer 和 from_network_layer 被数据链路层用来向网络层传递数据包或者从网络层接收数据包。注意，from_physical_layer 和 to_physical_layer 在数据链路层和物理层之间传递帧。换句话说，to_network_layer 和 from_network_layer 处理第二层和第三层之间的接口，而 from_physical_layer 和 to_physical_layer 则处理第一层和第二层之间的接口。

在大多数协议中，我们假设信道是不可靠的，并且偶尔会丢失整个帧。为了能够从这种灾难中恢复过来，发送方的数据链路层每当发出一帧，就要启动一个内部计时器或者时钟。如果在预设的时间间隔内没有收到应答，则时钟超时，数据链路层会收到一个中断信号。

在我们的协议中，这个过程是这样实现的：让过程 wait_for_event 返回 event = timeout。过程 start_timer 和 stop_timer 分别打开和关闭计时器。只有当计时器在运行并且调用 stop_timer 之前，超时事件才有可能发生。在计时器运行的同时，允许显式地调用 start_timer；这样的调用只是重置时钟，等到再经过一个完整的时钟间隔之后引发下一次超时事件（除非它再次被重置，或者被关闭）。

过程 start_ack_timer 和 stop_ack_timer 控制一个辅助计时器，该定时器被用于在特定条件下产生确认。

过程 enable_network_layer 和 disable_network_layer 用在较为复杂的协议中。在这样的协议中，我们不再假设网络层总是有数据要发送。当数据链路层启动网络层后，它就允许网络层在有数据包要发送时中断自己。我们用 event=network_layer_ready 来表示这种情况。当网络层被禁用后，它不会引发这样的事件发生。通过非常谨慎地设置何时启用网络层以及何时禁用网络层，数据链路层就能有效避免网络层用大量的数据包把自己淹没，即耗尽所有的缓存空间。

帧序号总是落在从 0 到 MAX_SEQ（含）的范围内，不同协议的 MAX_SEQ 可以不相同。通常有必要对序号按循环加 1 处理（即 MAX_SEQ 之后是 0）。宏 inc 可以执行这项序

号递增任务。之所以把这项工作定义成宏是因为在帧处理的关键路径上都要用到该参数。正如后面我们将会看到，限制网络性能的因素通常在于协议处理过程，所以把这种简单操作定义成宏并不会影响代码的可读性，却能够提高性能。

图 3-11 中的声明是我们下面将要讨论的每个协议的一部分。为了节省空间和方便引用，它们被提取出来列在一起；但从概念上讲，它们应该与协议本身合并在一起。在 C 语言中，合并的方法是，把这些定义放在一个特殊的头文件中，这里是 protocol.h 文件，然后在协议文件中使用 C 预处理器的#include 设施将这些定义包含进来。

3.3.1　一个乌托邦式的单工协议

作为第一个例子，我们来考虑一个简单得不能再简单的协议，它不需要考虑任何出错的情况。在这个协议中，数据只能单向传输。发送方和接收方的网络层总是处于准备就绪状态。数据处理的时间忽略不计。可用的缓存空间无穷大。最强的一个条件是数据链路层之间的通信信道永远不会损坏帧或者丢失帧。这个完全不现实的协议我们给它一个昵称“乌托邦”（Utopia），我们将以此作为构建后续协议的基本结构。图 3-12 给出了这个协议的实现。

```
/* Protocol 1 (Utopia) provides for data transmission in one direction only,
   from sender to receiver. The communication channel is assumed to be error
   free and the receiver is assumed to be able to process all the input
   infinitely quickly.Consequently, the sender just sits in a loop pumping
   data out onto the line as fast as it can. */
typedef enum {frame_arrival} event_type;
#include "protocol.h"
void sender1(void)
{
   frame s;                              /* buffer for an outbound frame */
   packet buffer;                        /* buffer for an outbound packet */
   while (true) {
     from_network_layer(&buffer); /* go get something to send */
     s.info = buffer;                    /* copy it into s for transmission */
     to_physical_layer(&s);               /* send it on its way */
}                                        /* Tomorrow, and tomorrow, and tomorrow,
                                         Creeps in this petty pace from day to day
                                         To the last syllable of recorded time.
                                          - Macbeth, V, v */
}
void receiver1(void)
{
   frame r;
   event_type event;                     /* filled in by wait, but not used here */
   while (true) {
     wait_for_event(&event);             /* only possibility is frame_arrival */
     from_physical_layer(&r);            /* go get the inbound frame */
     to_network_layer(&r.info);          /* pass the data to the network layer */
   }
}
```

图 3-12　一个乌托邦式的单工协议

协议由两个单独的过程组成：一个发送过程和一个接收过程。发送过程运行在源机器的数据链路层上；接收过程运行在目标机器的数据链路层上。这里没有用到序号和确认，所以不需要 MAX_SEQ。唯一可能的事件类型是 frame_arrival（即到达了一个完好无损

的帧）。

发送过程是一个无限的 while 循环，它尽可能快速地把数据放到线路上。循环体由三个动作组成：从（总是就绪的）网络层获取一个数据包、利用变量 s 构造一个出境帧，然后通过物理层发送该帧。这个协议只用到了帧结构中的 info 字段，因为其他字段都跟差错控制或者流量控制有关，而这里并没有差错控制或者流量控制方面的限制。

接收过程同样很简单。开始时，它等待某些事情的发生，这里唯一可能的事件是到达了一个未损坏的帧。到达了一个帧后，过程 wait_for_event 返回，其中的参数 event 被设置成 frame_arrival（这里被忽略）。调用 from_physical_layer 将新到达的帧从硬件缓冲区中删除，并且放到变量 r 中，以便接收方的代码可以访问该帧。最后，该帧的数据部分被传递给网络层，数据链路层返回继续等待下一帧的到来，即实际上它把自己挂起来，直到下一帧到来为止。

乌托邦协议是不现实的，因为它不处理任何流量控制或纠错工作。其处理过程接近于无确认的无连接服务，必须依赖更高层次来解决上述这些问题，即使无确认的无连接的服务也要做一些差错检测的工作。

3.3.2 无错信道上的单工停-等式协议

现在我们将处理这样的问题：发送方以高于接收方能处理到达帧的速度发送帧，导致接收方被淹没。这种情形实际上很容易出现，因此协议是否能够防止它非常重要。然而，我们仍然假设通信信道不会出错，并且数据流量还是单工的。

一种解决办法是建立足够强大的接收器，使其强大到能处理一个接着一个帧组成的连续流（或等价于把数据链路层定义成足够慢，慢到接收器的处理速度完全跟得上）。它必须有足够大的缓冲区和以线速率运行的处理能力，而且必须能足够快地把所接收到的帧传递给网络层帧。然而，这是一个最坏情况下的解决方案。它需要专用的硬件，而且如果该链路的利用率十分低还可能浪费资源。此外，它只是把发送方太快这个问题转移到了其他地方，在这种情况下就是网络层。

这个问题的更一般化解决方案是让接收方给发送方提供反馈信息。接收方将数据包传递给网络层之后给发送方返回一个小的哑帧，实际上这一帧的作用是给发送方一个许可，允许它发送下一帧。发送方在发出一帧之后，根据协议要求，它必须等待一段时间直到短哑帧（即确认）到达。这种延缓就是流量控制协议的一个简单例子。

发送方发送一帧，等待对方确认到达后才能继续发送，这样的协议称为**停-等式协议**（stop-and-wait）。图 3-13 给出了一个单工停-等式协议的例子。

虽然这个例子中的数据流量是单工的，即只是从发送方传到接收方，但是帧可以在两个方向上传送。因此，两个数据链路层之间的通信信道必须具备双向传输信息的能力。然而，这个协议限定了流量的严格交替关系：首先发送方发送一帧，然后接收方发送一帧；接着发送方发送另一帧，然后接收方发送另一帧，以此类推。这里采用一个半双工的物理信道就足够了。

就像在协议 1 中那样，发送方首先从网络层获取一个数据包，用它构造一帧，然后发送出去。但现在，与协议 1 不同的是，发送方在开始下一轮循环从网络层获取下一个数据

```
/* Protocol 2 (Stop-and-wait) also provides for a one-directional flow of data
   from sender to receiver. The communication channel is once again assumed to
   be error free, as in protocol 1. However, this time the receiver has only a
   finite buffer capacity and a finite processing speed, so the protocol must
   explicitly prevent the sender from flooding the receiver with data faster
   than it can be handled. */

typedef enum {frame_arrival} event_type;
#include "protocol.h"
void sender2(void)
{
   frame s;                              /* buffer for an outbound frame */
   packet buffer;                        /* buffer for an outbound packet */
   event_type event;                     /* frame_arrival is the only possibility */
   while (true) {
      from_network_layer(&buffer);       /* go get something to send */
      s.info = buffer;                   /* copy it into s for transmission */
      to_physical_layer(&s);             /* bye-bye little frame */
      wait_for_event(&event);            /* do not proceed until given the go ahead */
   }
}
void receiver2(void)
{
   frame r, s;                           /* buffers for frames */
   event_type event;                     /* frame_arrival is the only possibility */
   while (true) {
      wait_for_event(&event);            /* only possibility is frame_arrival */
      from_physical_layer(&r);           /* go get the inbound frame */
      to_network_layer(&r.info);         /* pass the data to the network layer */
      to_physical_layer(&s);             /* send a dummy frame to awaken sender */
   }
}
```

图 3-13　一个单工停等式协议

包之前必须等待，直到确认帧到来。发送方的数据链路层甚至根本不检查接收到的帧，因为对它而言只有一种可能性，即入境帧总是来自接收方的确认。

在 receiver1 和 receiver2 之间的唯一区别在于 receiver2 将数据包递交给网络层之后，在进入下一轮等待循环之前，要先给发送方返回一个确认帧。对于发送方来说，确认帧的到来比帧内包含什么内容更重要，因此接收方根本不需要往确认帧中填充任何特别的信息。

3.3.3　有错信道上的单工停-等式协议

现在让我们来考虑比较常规的情形，即通信信道可能会出错。帧可能会被损坏，也可能完全被丢失。然而，我们假设，如果一帧在传输过程中被损坏，则接收方硬件在计算校验和时能检测出来。如果一帧被损坏了之后校验和仍然是正确的（这种情况不太可能会出现），那么这个协议（以及所有其他的协议）将会失败（即给网络层递交了一个不正确的数据包）。

粗看起来，协议 2 稍作修改就能应付这项工作：增加一个计时器。发送方发出一帧，接收方只有在正确接收到数据之后才返回一个确认帧。如果到达接收方的是一个已经损坏的帧，则它将被丢弃。经过一段时间之后发送方将超时，于是它再次发送该帧。这个过程将不断重复，直至该帧最终完好无损地到达接收方。

这个方案有一个致命的缺陷。在继续往下读之前，请仔细想一想这个问题，并努力找

出哪里可能会出错。

为了看清楚哪里可能出错，必须牢记：数据链路层的目标是在两个网络层进程之间提供无差错的、透明的通信。机器A上的网络层将一系列数据包交给它的数据链路层，而它的数据链路层则必须保证这些数据包将丝毫不差地由机器B上的数据链路层递交给它的网络层。尤其是机器B上的网络层不可能知道数据包是否被丢失，或者被复制了多份，所以数据链路层必须保证不可能出现没有形式的传输错误会导致一个数据包被多次递交给了网络层，然而这似乎不太可能。

考虑下面的场景。

（1）机器A的网络层将数据包1交给它的数据链路层。机器B正确地接收到该数据包，并且将它传递给机器B上的网络层。机器B给机器A送回一个确认帧。

（2）确认帧完全丢失了，它永远也不可能到达机器A。如果信道出错后只丢数据帧而不丢控制帧，则问题就大大简化了；但遗憾的是，信道对这两种帧并没有区别对待。

（3）机器A上的数据链路层最终超时。由于它没有收到确认，所以它（不正确地）认为自己发的数据帧丢失了，或者被损坏了，于是它再次发送一个包含数据包1的帧。

（4）这个重复的帧也完好无损地到达机器B上的数据链路层，并且不知不觉中被传给了网络层。如果机器A正在给机器B发送一个文件，那么文件中的一部分内容将会被重复发送（即机器B得到的文件副本是不正确的，而且错误没有被检测出来）。换句话说，该协议失败了。

显然，对于接收方来说，它需要有一种办法能够区分到达的帧是第一次发来的新帧，还是被重传的老帧。为了做到这一点，很显然的做法是让发送方在它所发送的每个帧的头部放上一个序号。然后，接收方可以检查它所接收到的每个帧的序号，由此判断这是个新帧还是应该被丢弃的重复帧。

因为协议必须正确，并且处于链路效率的考虑包含在帧头中的序号字段可能很小，于是问题出现了：序号至少需要用多少位表示？帧头可以提供1位、数位，1个字节或多个字节作为序号，具体多少位由链路层协议自己确定。最重要的一点在于帧携带的序号必须足够大到保证协议能正确工作，否则它就不那么像一个协议。

在这个协议中，唯一不明确的地方出现在一帧（序号为m）和它的直接后续帧（序号为m+1）之间。如果m号帧被丢失，或者被损坏，接收方将不会对其进行确认，从而发送方将会不停地发送该帧。一旦该帧被正确地接收到了，接收方将给发送方返回一个确认。正是在这里可能出现潜在的问题。依据确认帧是否正确地返回发送方，发送方决定发送m号帧还是m+1号帧。

在发送方，触发其发送m+1号的事件是m号帧的确认帧按时返回了。但是这种情况隐含着m–1号帧已经被接收方正确地接收，而且它的确认帧也已经正确地被发送方接收。否则发送方就不会开始发送m号帧，更不用说发送m+1号帧了。所以，唯一不明确的地方在于一帧和它的前一帧或者和它的后一帧之间，而不在于它的前一帧和它的后一帧两者之间。

因此，一位序号（0或者1）就足以解决问题。在任何一个时刻，接收方期望下一个特定的序号。当包含正确序号的帧到来时，它被接受下来并且被传递给网络层。然后，接收方期待的下一个的序号模2增1（即0变成1，1变成0）。任何一个到达的帧，如果包含了错误序号都将作为重复帧而遭到拒绝。不过，最后一个有效的确认要被重复，以便发送方

最终发现已经被接收的那个帧。

图 3-14 给出了这类协议的一个例子。如果在一个协议中，发送方在前移到下一个数据之前必须等待一个肯定确认，这样的协议称为**自动重复请求**（ARQ，Automatic Repeat reQuest）或**带有重传的肯定确认**（PAR，Positive Acknowledgement with Retransmission）。与协议 2 类似，这类协议也只在一个方向上传输数据。

```
/* Protocol 3 (PAR) allows unidirectional data flow over an unreliable channel. */
  #define MAX_SEQ1                        /* must be 1 for protocol 3 */
  typedef enum {frame_arrival, cksum_err, timeout} event_type;
  #include "protocol.h"
void sender3(void)
{
  seq_nr next_frame_to_send;              /* seq number of next outgoing frame */
  frame s;                                /* scratch variable */
  packet buffer;                          /* buffer for an outbound packet */
  event_type event;
  next_frame_to_send = 0;                 /* initialize outbound sequence numbers */
  from_network_layer(&buffer);            /* fetch first packet */
  while (true) {
    s.info = buffer;                      /* construct a frame for transmission */
    s.seq = next_frame_to_send;           /* insert sequence number in frame */
    to_physical_layer(&s);                /* send it on its way */
    start_timer(s.seq);                   /* if answer takes too long, time out */
    wait_for_event(&event);               /* frame_arrival, cksum_err, timeout */
    if (event == frame_arrival) {
      from_physical_layer(&s);            /* get the acknowledgement */
      if (s.ack == next_frame_to_send) {
        stop_timer(s.ack);                /* turn the timer off */
        from_network_layer(&buffer);      /* get the next one to send */
        inc(next_frame_to_send);          /* invert next_frame_to_send */
      }
    }
  }
}
void receiver3(void)
{
  seq_nr frame_expected;
  frame r, s;
  event_type event;
  frame_expected = 0;
  while (true) {
  wait_for_event(&event);                 /* possibilities: frame_arrival, cksum_err */
  if (event == frame_arrival){            /* a valid frame has arrived */
    from_physical_layer(&r);              /* go get the newly arrived frame */
    if (r.seq == frame_expected){         /*this is what we have been waiting for */
      to_network_layer(&r.info);          /* pass the data to the network layer */
      inc(frame_expected);                /* next time expect the other sequence nr */
    }
   s.ack = 1 - frame_expected;
```

图 3-14　带有重传机制的肯定确认协议

协议 3 与其前任协议的不同之处在于，当发送方和接收方的数据链路层处于等待状态时，两者都用一个变量记录下了有关的值。发送方在 next_frame_to_send 中记录了下一个要发送的帧的序号；接收方则在 frame_expected 中记录了下一个期望接收的序号。每个协议在进入无限循环之前都有一个简短的初始化阶段。

发送方在发出一帧后启动计时器。如果计时器已经在运行，则将它重置，以便等待另一个完整的超时时间间隔。在选择超时值时，应该保证它足够长，确保帧到达接收方，按

照最坏的情形被接收方所处理，然后确认帧被返回发送方所需要的全部操作时间。只有当这段时间间隔过去之后，发送方才可以假定原先的数据帧或者它的确认帧已经被丢失，于是重发原先的数据帧。如果超时间隔设置得太短，发送方将会发送一些不必要的帧。虽然这些额外的帧不会影响协议的正确性，但是会严重损害协议的性能。

发送方在送出一帧并启动了计时器后，它就等待着相关事情的发生。此时只有3种可能的情况：确认帧完好无损地返回、确认帧受到损伤蹒跚而至或者计时器超时。如果到达了一个有效的确认帧，则发送方从它的网络层获取下一个数据包，并把它放入缓冲区覆盖掉原来的数据包，同时它还会递增帧的序号。如果到达了一个受损的确认帧，或者计时器超时，则缓冲区和序号都不作任何改变，以便重传原来的帧。无论如何，在这3种情况下，缓冲区里的内容被发送出去（下一个数据包或者重复数据包）。

当一个有效帧到达接收方时，接收方首先检查它的序号，确定是否为重复数据包。如果不是，则接受该数据包并将它传递给网络层，然后生成一个确认帧。重复帧和受损帧都不会被传递给网络层，但它们的到来会导致最后一个正确接收到的数据帧的确认被重复发送，返回给发送方，以便发送方做出前进到下一帧或重发那个受损帧的决策。

3.4 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同时传输数据。实现全双工数据传输的一种办法是运行前面协议的两个实例，每个实例使用一条独立的链路进行单工数据传输（在不同的方向上）。因此，每条链路由一个“前向”信道（用于数据）和一个“逆向”信道（用于确认）组成。两种情况下的逆向信道带宽几乎完全被浪费了。

一种更好的做法是使用同一条链路来传输两个方向上的数据。毕竟，协议2和协议3已经在两个方向上传输帧，而且逆向信道与前向信道具有同样的容量。在这种模型中，从机器A到机器B的数据帧可以与从机器A到机器B的确认帧混合在一起。接收方只要检查入境帧头部的kind字段，就可以区别出该帧是数据帧还是确认帧。

尽管让数据帧和控制帧在同一条链路上交错传输是对前面提到的两条独立物理链路方案的一种改进，但仍然有更进一步改进的可能。当到达一个数据帧时，接收方并不是立即发送一个单独的控制帧，而是抑制自己并开始等待，直到网络层传递给它下一个要发送的数据包。然后，确认信息被附加在往外发送的数据帧上（使用帧头的ack字段）。实际上，确认信息搭了下一个出境数据帧的便车。这种暂时延缓确认以便将确认信息搭载在下一个出境数据帧上的技术就称为**捎带确认**（piggybacking）。

与单独发确认帧的方法相比，使用捎带确认的最主要好处是更好地利用了信道的可用带宽。帧头的ack字段只占用很少几位，而一个单独的帧则需要一个帧头、确认信息和校验和。而且，发送的帧越少，也意味着接收方的处理负担越轻。在下面讨论的协议中，捎带字段只占用帧头中的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，数据链路层应该等网络层传递给它下一个数据包，要等多长时间？如果数据链路层

等待的时间超过了发送方的超时间隔，那么该帧将会被重传，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，能够预测未来，那么它就知道下一个网络层数据包什么时候会到来，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自组织方法，比如等待一个固定的毫秒数。如果一个新的数据包很快就到来，那么确认就可立即被捎带回去。否则的话，如果在这段间隔超时之前没有新的数据包到来，数据链路层必须发送一个单独的确认帧。

接下去的 3 个协议都是双向协议，它们同属于一类称为**滑动窗口**（sliding window）的协议。这 3 个协议在效率、复杂性和缓冲区需求等各个方面有所不同，本节后面将会逐个讨论。如同所有的滑动窗口协议一样，在这 3 个协议中，任何一个出境帧都包含一个序号，范围从 0 到某个最大值。序号的最大值通常是 2^n-1，这样序号正好可以填入到一个 n 位的字段中。停-等式滑动窗口协议使用 n=1，限制了序号只能是 0 和 1，但是更加复杂的协议版本可以使用任意的 n。

所有滑动窗口协议的本质是在任何时刻发送方总是维持着一组序号，分别对应于允许它发送的帧。我们称这些帧落在**发送窗口**（sending window）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，但是我们绝对不能降低基本需求，即数据链路层协议将数据包递交给网络层的次序必须与发送机器上数据包从网络层被传递给数据链路层的次序相同。我们同样也不能改变这样的需求：物理通信信道就像一根“线”，也就是说，它必须按照发送的顺序递交所有的帧。

发送方窗口内的序号代表了那些可以被发送的帧，或者那些已经被发送但还没有被确认的帧。任何时候当有新的数据包从网络层到来时，它被赋予窗口中的下一个最高序号，并且窗口的上边界前移一格。当收到一个确认时，窗口的下边界也前移一格。按照这种方法发送窗口持续地维持了一系列未被确认的帧。图 3-15 显示了一个例子。

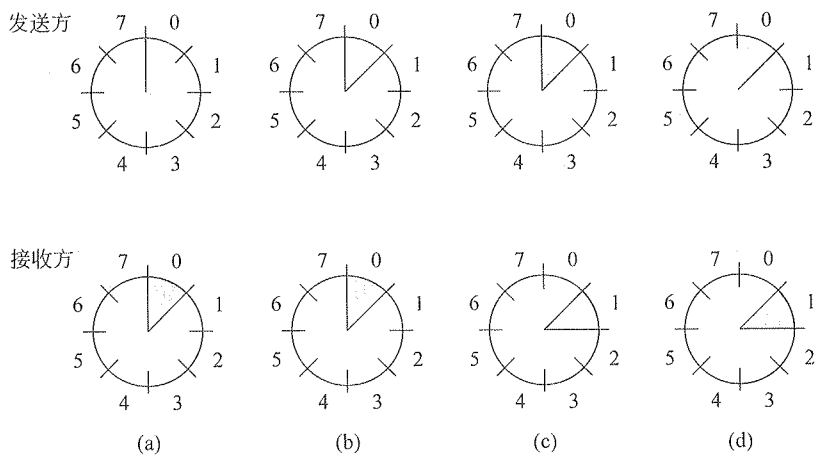


图 3-15　大小为 1 的滑动窗口，序号 3 位

（a）初始化；（b）第一帧发出之后；（c）第一帧被接收之后；（d）第一个确认被接收之后

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所有这些帧，以便满足可能的重传需要。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。如果窗口在某个时候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。任何落在窗口内的帧被放入接收方的缓冲区。当收到一个帧，而且其序号等于窗口下边界时，接收方将它传递给网络层，并将整个窗口向前移动 1 个位置。任何落在窗口外面的帧都将被丢弃。在所有情况下，接收方都要生成一个确认并返回给发送方，以便发送方知道该如何处理。请注意，窗口大小为 1 意味着数据链路层只能按顺序接受帧，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受数据，跟数据链路层的窗口大小没有关系。

图 3-15 显示了一个最大窗口尺寸为 1 的例子。起初，没有帧发出，所以发送方窗口上界和下界相同；但随着时间的推移，窗口的变化进展如图所示。与发送方窗口不同，接收方窗口始终保持着它的初始大小，窗口的旋转意味着下一帧被接受并被传递到网络层。

3.4.1 1 位滑动窗口协议

在讨论一般情形以前，我们先来考察一个窗口尺寸为 1 的滑动窗口协议。由于发送方在发出一帧以后，必须等待前一帧的确认到来才能发送下一帧，所以这样的协议使用了停-等式办法。

图 3-16 描述了这样一个协议。跟其他协议一样，它也是从定义变量开始的。next_frame_to_send 指明了发送方试图发送的那一帧。类似地，frame_expected 指明了接收方等待接收的那一帧。两种情况下，0 和 1 是唯一的可能。

在一般情况下，两个数据链路层中的某一个首先开始，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。初始启动的机器从它的网络层获取第一个数据包，然后根据该数据包创建一帧，并将它发送出去。当该帧（或者任何其他帧）到达目的地，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。如果这帧正是接收方所期望的，则将它传递给网络层，并且接收方的窗口向前滑动。

确认字段包含了最后接收到的正确帧的序号。如果该序号与发送方当前试图发送的帧的序号一致，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数据包。如果序号不一致，则它必须继续发送同一帧。无论什么时候只要接收到一帧，就要返回一帧。

现在我们来看协议 4 对于不正常情形的适应能力怎么样。假设计算机 A 试图将它的 0 号帧发送给计算机 B，同时 B 也试图将它的 0 号帧发送给 A。假定 A 发出一帧给 B，但是 A 的超时间隔设置得有点短。因此，A 可能会不停地超时而重发一系列相同的帧，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，它会被接受，并且 frame_expected 设置为 1。所有到达的后续帧都将遭到拒绝，因为 B 现在等待的是序号为 1 的帧，而不是序号为 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 0 号帧的确认，所以它不会从网络

```
/* Protocol 4 (Sliding window) is bidirectional. */
  #define MAX_SEQ1                        /* 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 sender's 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 3-16　一个 1 位滑动窗口协议

层获取新的数据包。

在每一个被拒绝的重复帧到达后，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，引起 A 开始发送下一个数据包。丢失的帧或者过早超时的任何一种混合出错情况都不会导致该协议向网络层递交重复的数据包、漏掉一个数据包或者死锁。协议正确！

然而，为了显示协议交换是多么的细微，我们注意到双方同时发送一个初始数据包时出现的一种极为罕见的情形。这种同步的困难程度如图 3-17 所示。图 3-17（a）显示了该协议的正常操作过程。图 3-17（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 3-17（a）所示，每一帧都将被接受。

然而，如果 A 和 B 同时发起通信，则它们的第一帧就会交错，数据链路层进入图 3-17（b）描述的状态。在图 3-17（a）中，每一帧到来后都带给网络层一个新的数据包，这里没有任何重复；在图 3-17（b）中，即使没有传输错误，也会有一半的帧是重复的。类似的情形同样发生在过早超时的情况下，即使有一方明显地首先开始传输也会发生这样的情形。实际上，

如果发生多个过早超时，则每一帧都有可能被发送三次或者更多次，严重浪费了宝贵带宽。

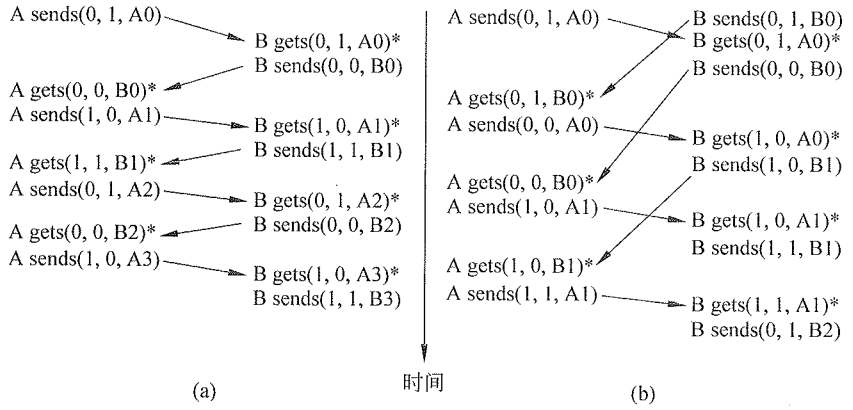


图 3-17 协议 4 的两种场景

（a）正常情况；（b）异常情况

记号表示（序号，确认，包号），星号表示网络层接受了一个包

3.4.2 回退 N 协议

到现在为止，我们一直有这样的默契假设，即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。有时候，这种假设明显是不正确的。在这些情形下，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 50 kbps 的卫星信道，它的往返传播延迟为 500 毫秒。我们想象一下，在该信道上用协议 4 来发送长度为 1000 位的帧。在 t=0 时刻，发送方开始发送第一帧；在 t=20 ms 时，该帧被完全发送出去；直到 t=270 毫秒时该帧才完全到达接收方；在 t=520 毫秒时，确认帧才回到发送方，而且所有这一切还是在最好情况下才可能发生（即在接收方没有停顿并且确认帧很短）。这意味着，发送方在 500/520 或 96%的时间内是被阻塞的。换句话说，只有 4%的有效带宽被利用了。很显然，站在效率的角度，长发送时间、高带宽和短帧这三者组合在一起就是一种灾难。

这里描述的问题可以看作是这种规则的必然结果，即发送方在发送下一帧之前必须等待前一帧的确认。如果我们放松这一限制，则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 个帧，而不是一个帧。通过选择足够大的 w 值，发送方就可以连续发送帧，因为在发送窗口被填满之前前面帧的确认就返回了，因而防止了发送方进入阻塞。

为了找到一个合适的 w 值，我们需要知道在一帧从发送方传播到接收方期间信道上能容纳多少个帧。这种容量由比特/秒的带宽乘以单向传送时间所决定，或数据链路层有责任以链路的**带宽-延迟乘积**（bandwidth-delay product）序列把数据包传递给网络层。我们可以将这个数量拆分成一帧的比特数，从而用帧的数量来表示。我们将这个数值称为 BD。因此，w 应设置为 2BD+1。如果考虑发送方连续发送帧并且在往返时间内收到一个确认，那么两倍的带宽-延时就是发送方可以连续发送的帧的个数；“+1”是因为必须接收完整个帧之后确认帧才会被发出。

在上述例子中，具有 50 kbps 的带宽和 250 毫秒的单向传输时间，带宽-延迟乘积为

12.5 kb 或 12.5 个长度为 1000 位的帧。因此，2BD+1 是 26 帧。假设发送方还和以前一样开始发送 0 号帧，并且每隔 20 毫秒发送一个新帧。到 t=520 时，它已经发送了 26 帧，这时 0 号帧的确认刚好返回。此后，每隔 20 毫秒就会到达一个确认，因此必要时发送方总是可以发送帧。从那时起，25 或 26 个未被确认的帧将始终在旅途中。换言之，发送方的最大窗口尺寸是 26。

对于较小尺寸的窗口，链路的利用率将小于 100%，因为发送方时常会被阻塞住。我们可以将链路利用率表示成发送方未被阻塞的时间比例：

$$\text{链路利用率} \leqslant \frac{w}{1+2BD}$$

这个值是个上限，因为它不容许有任何帧的处理时间，并且视确认帧的长度为零（因为它通常很短）。该方程显示无论带宽-延迟乘积多大发送方的窗口 w 一定要大。如果延迟很高，发送方将迅速耗尽窗口，即使对于卫星通信那样的中等带宽；如果带宽很高，即使对于普通延迟发送方也将很快耗尽其窗口，除非它有一个很大的发送窗口（例如，一个具有 1 毫秒延迟的 1 Gbps 链路能容纳 1 Mb）。停-等式协议的 w=1，如果延迟传播甚至只有一帧时间，协议效率都将低于 50%。

保持多个帧同时在传送的技术是**管道化**（pipelining）的一个例子。在一个不可靠的通信信道上像管道一样传送帧会引起一些严重的问题。首先，位于某个数据流中间的一个帧被损坏或丢失，会发生什么事情？在发送方发现问题之前大量的后续帧已经发出，并且即将到达接收方。当损坏的那个帧到达接收方时，显然它应该被丢弃，但接收方该如何处理所有那些后续到达的正确帧呢？请记住，接收方的数据链路层有责任按正确的顺序把数据包传递给网络层。

有两种基本办法可用来处理管道化传输中出现的错误，图 3-18 显示了这两种技术。

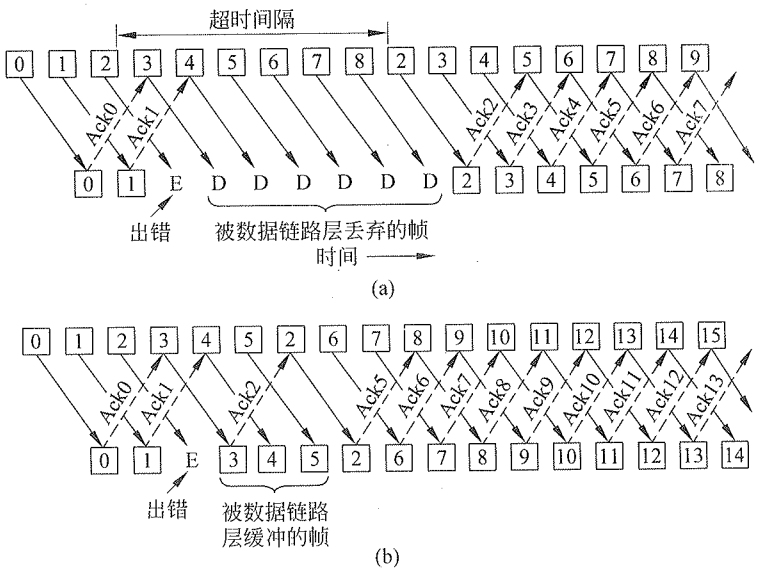


图 3-18　管道化以及差错恢复

（a）接收方窗口为 1 时错误的影响；（b）接收方窗口很大时错误的影响

一种选择办法称为**回退 n**（go-back-n），接收方只需简单丢弃所有到达的后续帧，而且

针对这些丢弃的帧不返回确认。这种策略对应于接收窗口大小为 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，它拒绝接受任何帧。如果在计时器超时以前，发送方的窗口已被填满，则管道将变为空闲。最终，发送方将超时，并且按照顺序重传所有未被确认的帧，从那个受损或者被丢失的帧开始。如果信道的错误率很高，这种方法会浪费大量的带宽。

在图 3-18（b）中，我们可以看到回退 n 帧的情形，其中接收方的窗口比较大。0 号帧和 1 号帧被正确地接收和确认。然而，2 号帧被损坏或者丢失。发送方并有意识到出现了问题，它继续发送后续的帧，直到 2 号帧的计时器超时。然后，它退回到 2 号帧，从这里重新发送 2 号、3 号、4 号帧等，一切从头再来。

针对管道化发送帧发生的错误，另一种通用的处理策略称为**选择重传**（selective repeat）。使用这种策略，接收方将收到的坏帧丢弃，但接受并缓存坏帧后面的所有好帧。当发送方超时，它只重传那个最早的未被确认的帧。如果该重传的帧正确到达接收方，接收方就可按序将它缓存的所有帧递交给网络层。选择重传对应的接收方窗口大于 1。如果窗口很大，则这种方法对数据链路层的内存需求很大。

选择重传策略通常跟否定策略结合起来一起使用，即当接收方检测到错误（例如，帧的校验和错误或者序号不正确），它就发送一个**否定确认**（NAK，negative acknowledgement）。NAK 可以触发该帧的重传操作，而不需要等到相应的计时器超时，因此协议性能得以提高。

在图 3-18（b）中，0 号帧和 1 号帧被正确接收，并得到确认；2 号帧丢失了。当 3 号帧到达接收方时，那里的数据链路层注意到自己错过了一帧，所以它针对错失的 2 号帧返回一个 NAK，但是将第 3 帧缓存起来。当 4 号和 5 号帧到达之后，它们也被数据链路层缓存起来，而没有被传递给网络层。2 号帧的 NAK 抵达发送方后，发送方立即重传 2 号帧。当该帧到达接收方时，数据链路层现在有了 2 号、3 号、4 号和 5 号帧，于是就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 2 号帧到 5 号帧），如图中所示。如果 NAK 被丢失，则发送方的 2 号帧计时器最终超时，发送方就会重新发送 2 号帧（仅仅这一帧），但是，这可能已经过了相当长一段时间。

这两种不同方法恰好是带宽使用效率和数据链路层缓存空间之间的权衡。根据资源的紧缺程度，选择使用其中一种方法。图 3-19 显示了一个回退 n 协议的示例，其中接收方的

```
/* Protocol 5 (Go-back-n) allows multiple outstanding frames. The sender may
   transmit up to MAX_SEQ frames without waiting for an ack. In addition,
   unlike in 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
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[ ])
```

图 3-19 一个采用了回退 n 机制的滑动窗口协议

```
{
/* 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;                     /* number of 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 frame */
      inc(next_frame_to_send);          /* prepare to send the next one */
      }
    }
    if (nbuffered < MAX_SEQ)
      enable_network_layer();
    else
      disable_network_layer();
  }
}
```

图 3-19 （续）

数据链路层按序接收入境帧，发生错误之后的所有入境帧都将被丢弃。在这个协议中，我们第一次抛掉了网络层总是有无穷多的数据包要发送的假设。当网络层希望发送一个数据包时，它触发一个 network_layer_ready 事件。然而，为了使得流量控制机制可以限制任何时候的发送窗口大小或者未确认的出境帧个数，数据链路层必须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。

任何时候，可以发送的帧的最大个数不能等同于序号空间的大小。对回退 n 协议，可发送的帧最多为 MAX_SEQ 个，即使存在 MAX_SEQ+1 个不同的序号（分别为 0、1、2、…、MAX_SEQ）。我们马上就看到对于下一个选择重传协议，这种限制更严格。为了看清楚为什么必须有这个限制，请考虑下面 MAX_SEQ=7 的场景：

（1）发送方发送 0～7 号共 8 个帧。

（2）7 号帧的捎带确认返回到发送方。

（3）发送方发送另外 8 个帧，其序列号仍然是 0～7。

（4）现在 7 号帧的另一个捎带确认也返回了。

问题出现了：属于第二批的 8 个帧全部成功到达了呢，还是这 8 个帧全部丢失了（把出错之后丢弃的帧也算作丢失）？在这两种情况下，接收方都会发送针对 7 号帧的确认，但发送方却无从分辨。由于这个原因，未确认帧的最大数目必须限制为不能超过 MAX_SEQ。

虽然协议 5 没有缓存出错后到来的帧，但是它也没有因此完全摆脱缓存问题。由于发送方可能在将来的某个时刻要重传所有未被确认的帧，所以，它必须把已经发送出去的帧一直保留，直到它能肯定接收方已经接受了这些帧。当 n 号帧的确认到达，n–1 号帧、n–2 号帧等都会自动被确认。这种类型的确认称为**累计确认**（cumulative acknowledgement）。当先前一些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。每当到达任何一个确认，数据链路层都要检查看是否可以释放出一些缓冲区。如果能释放出一些缓冲区（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又可以激发更多的 network_layer_ready 事件了。

对于这个协议，我们假设链路上总是有反向的数据流量可以捎带确认。协议 4 不需要这样的假设，因为它每次接收到一帧就送回一帧，即使它刚刚发送出一帧也必须再发一个帧。在下一个协议中，我们将用一种非常巧妙的办法来解决这个单向流量问题。

因为协议 5 有多个未被确认的帧，所以逻辑上它需要多个计时器，即每一个未被确认的帧都需要一个计时器。每一帧的超时是独立的，相互之间没有关系。然而，用软件很容易模拟所有这些计时器：只需使用一个硬件时钟，它周期性地引发中断。所有未发生的超时事件构成了一个链表，链表中的每个节点包含了离超时还有多少个时钟滴答、超时对应的那个帧，以及一个指向下一个节点的指针。

为了示范如何实现多个计时器，请考虑图 3-20（a）中的例子。假设每 1 毫秒时钟滴答一次。初始时，实际时间为 10:00:00.000；有三个超时事件，分别定在 10:00:00.005、10:00:00.013 和 10:00:019。每当硬件时钟滴答一次，实际的时间就被更新，链表头上的滴答计数器被减 1。当滴答计数器变成 0 的时候，就引发一个超时事件，并将该节点从链表中移除，如图 3-20（b）所示。虽然这种实现方式要求在调用 start_timer 或 stop_timer 时扫描链表，但在每次滴答中断时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，表示针对哪一帧计时。

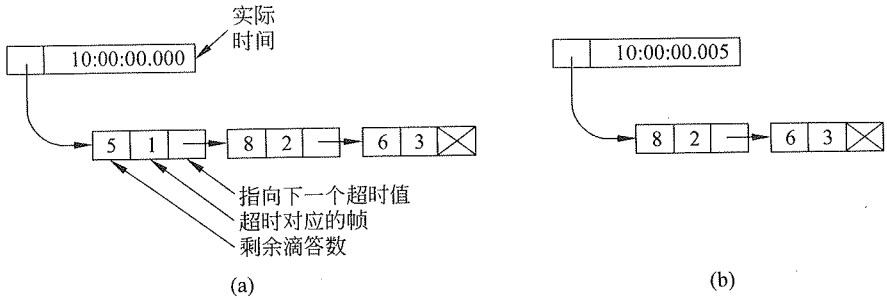


图 3-20　软件模拟多个计时器

（a）队列中的计时器；（b）第一个计时器超时后的情形

3.4.3　选择重传协议

如果错误很少发生，则回退 n 协议可以工作得很好；但是，如果线路质量很差，那么重传的帧要浪费大量带宽。另一种处理错误的策略是选择重传协议，允许接收方接受并缓存坏帧或者丢失帧后面的所有帧。

在这个协议中，发送方和接收方各自维持一个窗口，该窗口分别包含可发送或已发送但未被确认的和可接受的序号。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。相反，接收方的窗口总是固定不变，其大小等于预先设定的最大值。接收方为其窗口内的每个序号保留一个缓冲区。与每个缓冲区相关联的还有一个标志位（arrived），用来指明该缓冲区是满的还是空的。每当到达一帧，接收方通过 between 函数检查它的序号，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该帧，则接受该帧，并且保存在缓冲区。不管这帧是否包含了网络层所期望的下一个数据包，这个过程肯定要执行。当然，该帧只能被保存在数据链路层中，直到所有序号比它小的那些帧都已经按序递交给网络层之后，它才能被传递给网络层。图 3-21 显示了一个使用该算法的协议。

```
/*Protocol 6 (Selective repeat) 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_SEQ7                        /* 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 protocol 5, 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 */
```

图 3-21　一个采用了选择重传机制的滑动窗口协议

```
  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 */
```

图 3-21 （续）

```
    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();
 }
}
```

图 3-21 （续）

非顺序接收引发了一些特殊问题，这些问题对于那些按顺接受帧的协议是不用考虑的。我们用一个例子就很容易说明麻烦之处。假设我们用 3 位序号，那么发送方允许连续发送 7 个帧，然后开始等待确认。刚开始时，发送方和接收方的窗口如图 3-22（a）所示。现在发送方发出 0～6 号帧。接收方的窗口允许它接受任何序号落在 0～6（含）之间的帧。这 7 个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 7、0、1、2、3、4 或 5 号帧，如图 3-22（b）所示。所有这 7 个缓冲区都标记为空。

此时，灾难降临了，闪电击中了电线杆子，所有的确认都被摧毁。协议应该不管灾难是否发生都能正确工作。最终发送方超时，并且重发 0 号帧。当这帧到达接收方时，接收方检查它的序号，看是否落在窗口中。不幸的是，如图 3-22（b）所示，0 号帧落在新窗口中，所以它被当作新帧接受了。接收方同样返回（捎带）6 号帧确认，因为 0～6 号帧都已经接收到了。

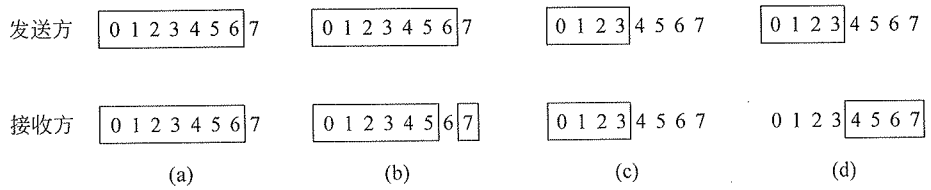


图　3-22

（a）窗口大小为 7 的初始情形；（b）发出 7 个帧并且接收 7 帧，但尚未确认；
（c）窗口大小为 4 的初始情形；（d）发出 4 个帧并且接收 4 帧，但尚未确认

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，所以它向前移动发送窗口，并立即发送 7、0、1、2、3、4 和 5 号帧。7 号帧将被接收方接收，并且它的数据包直接传递给网络层。紧接着，接收方的数据链路层进行检查，看它是否已经有一个有效的 0 号帧，它发现确实已经有了（即前面重发的 0 号帧），然后把内嵌的数据包作为新的数据包传递给网络层。因此，网络层得到了一个不正确的数据包。协议失败！

这个问题的本质在于：当接收方向前移动它的窗口后，新的有效序号范围与老的序号范围有重叠。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）。可怜的接收方根本无法区分这两种情形。

解决这个难题的办法是确保接收方向前移动窗口之后，新窗口与老窗口的序号没有重叠。为了保证没有重叠，窗口的最大尺寸应该不超过序号空间的一半，如图 3-22（c）和图 3-22（d）所做的那样。如果用 3 位来表示序号，则序号范围为 0～7。在任何时候，应该只有 4 个未被确认的帧。按照这种方法，如果接收方已经接收了 0～3 号帧，并且向前移动了窗口，以便允许接收第 4～7 帧，那么，它可以明确地区分出后续的帧是重传帧（序号为 0～3），还是新帧（序列号为 4～7）。一般来说，协议 6 的窗口尺寸为（MAX_SEQ+1）/2。

一个有意思的问题是：接收方必须拥有多少个缓冲区？无论如何，接收方不可能接受序号低于窗口下界的帧，也不可能接受序号高于窗口上界的帧。因此，所需要的缓冲区的数量等于窗口的大小，而不是序号的范围。在前面的 3 位序号例子中，只需要 4 个缓冲区就够了，编号为 0～3。当 i 号的帧到达时，它被放在（i mod 4）号缓冲区中。请注意，虽然 i 和（i+4）mod 4 在“竞争”同一个缓冲区，但它们永远不会同时落在窗口内，因为如果那样的话，则窗口的尺寸至少为 5。

出于同样的原因，需要的计时器数量等同于缓冲区的数量，而不是序号空间的大小。实际上，每个缓冲区都有一个相关联的计时器。当计时器超时，缓冲区的内容就要被重传。

协议 6 还放宽了隐含假设，即信道的负载很重。我们在协议 5 中作了这个假设，因为协议要依赖反方向上的数据帧来捎带确认信息。如果逆向流量很轻，确认会被延缓很长一段时间，这将导致问题。在极端的情况下，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，那么当发送方的窗口达到最大值后协议将被阻塞。

为了不受该假设的约束，当一个按正常次序发送的数据帧到达接收方之后，接收方通过 start_ack_timer 启动一个辅助的计时器。如果在计时器超时之前，没有出现需要发送的反向流量，则发送一个单独的确认帧。由该辅助计时器超时而导致的中断称为 ack_timeout 事件。在这样的处理方式下，缺少可以捎带确认的反向数据帧不再是一个问题，因此那些只存在单向数据流的场合依然可以使用该协议。只要一个辅助计时器就可以正常工作，如果该计时器在运行时又调用了 start_ack_timer，则该次调用不起作用。计时器不会被重置或者被扩展，因为它的作用是提供确认的某种最小速率。

辅助计时器的超时间隔应该明显短于与数据帧关联的计时器间隔，这点非常关键。这是确保下列情况的必要条件：即一个被正确接收的帧应该尽早地被确认，使得该帧的重传计时器不会超时因而不会重传该帧。

协议 6 采用了比协议 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，它就给发送方返回一个否定确认（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。在两种情况下，接收方要特别注意：接收到一个受损的帧，或者到达的帧并非是自己所期望的（可能出现丢帧错误）。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，则不会有实质性的伤害，因为发送方最终会超时，无论如何它都会重传丢失的帧。如果一个 NAK 被发送出去之后丢失了，而接收方又收到一个错误的帧，则 no_nak 将为 true，并且辅助计时器将被启动。当该辅助计时器超时后，一个 ACK 帧将被发送出去，以便使发送方重新同步到接收方的当前状态。

在某些情形下，从一帧被发出去开始算起，到该帧经传播抵达目的地、在那里被处理，

然后它的确认被传回来，整个过程所需要的时间（几乎）是个常数。在这些情形下，发送方可以把它的计时器调得“紧”一些，让它恰好略微大于正常情况下从发送一帧到接收到其确认之间的时间间隔。此时 NAC 的作用就微乎其微了。

然而，在其他一些情况下这段时间的变化非常大。例如，如果反向流量零零散散，那么如果刚好有逆向流量则接收到帧延缓发送确认之前的这段时间会比较短；反之，如果没有逆向流量，则这段时间会很长。因此发送方必须做出选择，要么将计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等）。两种选择都会浪费带宽。一般来说，当确认间隔的标准偏差与间隔本身相比非常大的时候，计时器应该设置得“松”一点。然后，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由哪一帧引发超时。在协议 5 中，引发超时的帧总是 ack_expected 指定的帧，因为它总是最早的那个帧。在协议 6 中，没有一种很具体的办法来确定谁来引发超时。假定已经发送了 0～4 号帧，这意味着未确认帧的列表是 01234，按照时间先后顺序排列。现在想象这样的情形：0 号帧超时，5 号帧（新帧）被发送出去，1 号帧超时，2 号帧超时，6 号帧（又一新帧）被发送出去。这时候，未确认帧的列表是 3405126，也是按照从最老的帧到最新的帧顺序排列。如果所有入境流量（即那些包含确认的帧）一下子全部被丢失，那么这 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，我们没有显示计时器的管理过程。相反，我们只是假设在超时变量 oldest_frame 设置好之后，它就能指出哪一帧超时。

3.5　数据链路协议实例

在单个建筑物内，局域网被广泛用于互连主机，但大多数广域网的基础设施是以点到点方式建设的。我们将在第 4 章重点关注局域网，这里要考察的是那些出现在 Internet 两种常见情形下的数据链路协议，这些协议主要用在点到点的线路上。第一种情形是通过广域网中的 SONET 光纤链路发送数据包。例如，这些链路被广泛用于连接一个 ISP 网络中位于不同位置的路由器。

第二种情形是运行在 Internet 边缘的电话网络本地回路上的 ADSL 链路。这些链接把成千上百万的个人和企业连接到 Internet 上。

针对上述这些使用场景 Internet 需要点到点链路，还会用到拨号调制解调器、租用线路和线缆调制解调器等。所谓**点到点协议**（PPP，Point-to-Point Protocol）的标准协议就是使用这些链路来发送数据包。PPP 由 RFC1661 定义，并在 RFC1662 中得到进一步的阐述（Simpson，1994a，1994b）。SONET 和 ADSL 链路都采用了 PPP，但在使用方式上有所不同。

3.5.1　SONET 上的数据包

2.6.4 章节涉及的 SONET 是物理层协议，它最常被用在广域网的光纤链路上，这些光纤链路构成了通信网络的骨干网，其中包括电话系统。它提供了一个以定义良好速度运行

的比特流，比如 2.4 Gbps 的 OC-48 链路。比特流被组织成固定大小字节的有效载荷，不论是否有用户数据需要发送，每隔 125 微秒要发出一个比特流。

为了在这些链路上承载数据包，需要某种成帧机制，以便将偶尔出现的数据包从传输它们的连续比特流中区分出来。运行在 IP 路由器上的 PPP 就提供了这种运行机制，如图 3-23 所示。

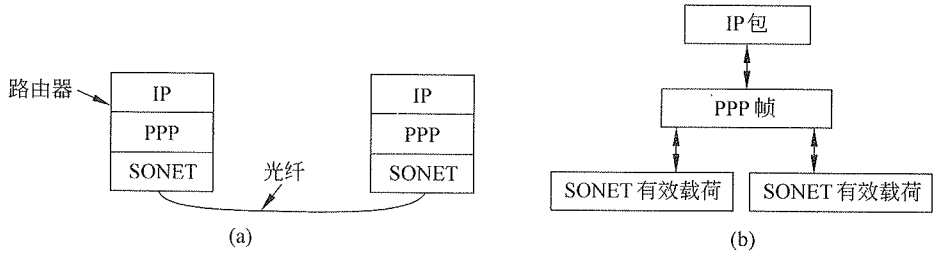


图 3-23 SONET 之上的数据包
（a）协议栈；（b）帧关系

PPP 功能包括处理错误检测链路的配置、支持多种协议、允许身份认证等。它是一个早期简化协议的改进，那个协议称为**串行线路 Internet 协议**（SLIP，Serial Line Internet Protocol）。伴随着一组广泛选项，PPP 提供了 3 个主要特性：

（1）一种成帧方法。它可以毫无歧义地区分出一帧的结束和下一帧的开始。

（2）一个链路控制协议。它可用于启动线路、测试线路、协商参数，以及当线路不再需要时温和地关闭线路。该协议称为**链路控制协议**（LCP，Link Control Protocol）。

（3）一种协商网络层选项的方式。协商方式独立于网络层协议。所选择的方法是针对每一种支持的网络层都有一个不同的**网络控制协议**（NCP，Network Control Protocol）。

因为没有必要重新发明轮子，所以 PPP 帧格式的选择酷似 HDLC 帧格式。HDLC 是**高级数据链路控制协议**（High-level Data Link Control），是一个早期被广泛使用的家庭协议实例。

PPP 和 HDLC 之间的主要区别在于：PPP 是面向字节而不是面向比特的。特别是 PPP 使用字节填充技术，所有帧的长度均是字节的整数倍。HDLC 协议则使用比特填充技术，允许帧的长度不是字节的倍数，例如 30.25 字节。

然而，实际上还有第二个主要区别。HDLC 协议提供了可靠的数据传输，所采用的方式正是我们已熟悉的滑动窗口、确认和超时机制等。PPP 也可以在诸如无线网络等嘈杂的环境里提供可靠传输，具体细节由 RFC1663 定义。然而，实际上很少这样做。相反，Internet 几乎都是采用一种“无编号模式”来提供无连接无确认的服务。

PPP 帧格式如图 3-24 所示。所有的 PPP 帧都从标准的 HDLC 标志字节 0x7E（01111110）开始。标志字节如果出现在 Payload 字段，则要用转义字节 0x7D 去填充；然后将紧跟在后面的那个字节与 0x20 进行 XOR 操作，如此转义使得第 6 位比特反转。例如 0x7D 0x5E 是标志字节 0x7E 的转义序列。这意味着只需要简单扫描 0x7E 就能找出帧的开始和结束之处，因为这个字节不可能出现在其他地方。接收到一个帧后要去掉填充字节。具体做法是，扫描搜索 0x7D，发现后立即删除；然后用 0x20 对紧跟在后面的那个字节进行 XOR 操作。此外，两个帧之间只需要一个标志字节。当链路上没有帧在发送时，可以用多个标志字节填充。

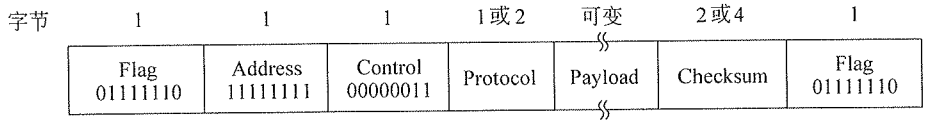


图 3-24　无编号模式操作下的完整 PPP 帧格式

紧跟在帧开始处标记字节后面出现的是 Address（地址）字段。这个字段总是被设置为二进制值 11111111，表示所有站点都应该接受该帧。使用这个值可避免如何为数据链路层分配地址这样的问题。

Address 字段后面是 Control（控制）字段，其默认值是 00000011。此值表示一个无编号帧。

因为 Address 和 Control 字段总是取默认配置的常数，因此 LCP 提供了某种必要的机制，允许通信双方就是否省略这两个字段进行协商，去掉的话可以为每帧节省 2 个字节的空间。

PPP 的第四个字段是 Protocol（协议）字段。它的任务是通告 Payload 字段中包含了什么类型的数据包。以 0 开始的编码定义为 IP 版本 4、IP 版本 6 以及其他可能用到的网络层协议，比如 IPX 和 AppleTalk。以 1 开始的编码被用于 PPP 配置协议，包括 LCP 和针对每个网络层协议而设置的不同 NCP。Protocol 字段的默认大小为 2 个字节，但它可以通过 LCP 协商减少到 1 个字节。协议设计师们也许过于谨慎了，认为有一天可能使用超过 256 个协议。

Payload（有效载荷）字段是可变长度的，最高可达某个协商的最大值。如果在链路建立时没有通过 LCP 协议进行协商，则采用默认长度 1500 字节。如果需要，在有效载荷后填充字节以便满足帧的长度要求。

Payload 字段后是 Checksum（校验和）字段，它通常占 2 个字节，但可以协商使用 4 个字节的校验和。事实上，4 字节的校验和与 32 位的 CRC 相同，其生成多项式就是 3.3.2 章节末尾给出的那个多项式。2 字节的校验和也是一个工业标准 CRC。

PPP 是一个成帧机制，它可以在多种类型的物理层上承载多种协议的数据包。为了在 SONET 上使用 PPP，RFC2615 列出了一些可用的选择（Malis 和 Simpson，1999）。因为这是检测物理层、数据链路层和网络层传输错误的主要手段，因此采用了 4 字节的校验和。它还建议不要压缩 Address、Control 和 Protocol 字段，因为 SONET 链路的运行速度已经达到很高了。

PPP 还有一个不同寻常的特点。PPP 的有效载荷在插入到 SONET 的有效载荷之前先进行扰码操作（正如 2.5.1 所描述的那样）。用长伪随机序列对有效载荷进行扰码 XOR 之后再传送出去。问题在于为了保持同步 SONET 比特流，必须频繁地进行比特跳变。这些跳变很自然地与语音信号的变化结合在一起，但在数据通信中用户选择发送的信息和数据包可能包含一长串的 0。有了扰频技术，用户因为发送一长串 0 而导致问题的可能性降到极低。

在通过 SONET 线路发送 PPP 帧之前，必须建立和配置 PPP 链路。PPP 链路的启动、使用和关闭的一系列阶段如图 3-25 所示。

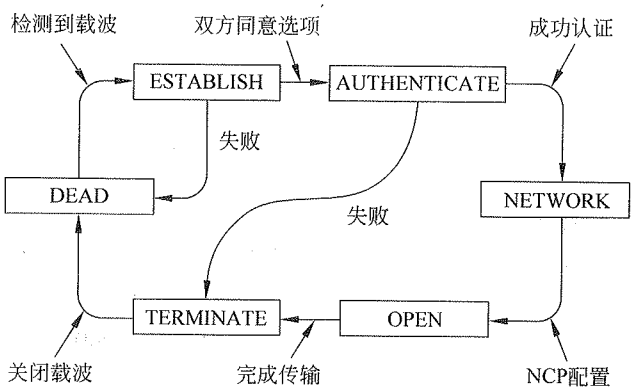


图 3-25 PPP 链路建立到释放的状态转换图

链路的初始状态为 DEAD（死），这意味着不存在物理层连接。当物理连接被建立起来，链路转移到 ESTABLISH（建立）状态。此时，PPP 对等实体交换一系列的 LCP 报文进行上面所说的那些 PPP 选项的协商，这些 LCP 报文放在 PPP 帧的 Payload 字段。初始发起连接的实体提出自己的选项请求，对等实体可以部分或者全部接受，甚至全部拒绝，同时它也可以提出自己的选项要求。

如果 LCP 选项协商成功，链路状态进入 AUTHENTICATE（认证）状态。现在，如果需要，双方可以互相检查对方的身份。如果认证成功，则链路进入 NETWORK（网络）状态，通过发送一系列的 NCP 包来配置网络层参数。NCP 协议很难一概而论，因为每个协议特定于实际采用的某个网络层协议，允许执行针对特定于该网络层协议的配置请求。例如，对于 IP 协议而言，为链路的两端分配 IP 地址是最重要的可能操作。

一旦进入 OPEN（打开）状态，双方就可以进行数据传输。正是在这个状态下，IP 数据包被承载在 PPP 帧中通过 SONET 线路传输。当完成数据传输后，链路进入 TERMINATE（终止）状态；当物理层连接被舍弃后从这里回到 DEAD 状态。

3.5.2 对称数字用户线

ADSL 以 Mbps 速率将百万计的家庭用户连接到 Internet 上，并且使用的是与普通老式电话服务相同的本地回路。在 2.5.3 节我们介绍了如何把一种称为 DSL 调制解调器的设备安置在家庭一端。它通过本地回路发出的数据比特到达一个称为 DSLAM（DSL 接入复用器，发音为“dee-slam”）的设备，该设备安置在电话公司端局。现在，我们将更详细地探讨如何在 ADSL 链路上承载数据包。

ADSL 使用的协议和设备的概貌如图 3-26 所示。不同的网络可以部署不同的协议，所以我们选择了最流行的场景。在家里，比如 PC 机那样的计算机使用以太网链路把 IP 数据包发送到 DSL 调制解调器；然后 DSL 调制解调器通过本地回路把 IP 数据包发送到 DSLAM，发送数据包所用的协议就是我们将要学习的协议。在 DSLAM 设备（或连接到它路由器，视具体实施而定）上 IP 数据包被提取出来，并被注入到 ISP 网络，因此它们最终能到达 Internet 上的任何目的地。

在图 3-26 中，ADSL 链路之上的协议底部是 ADSL 物理层。它们基于称为正交频分复

用（也称为离散多音）的数字调制方案，就像我们在 2.5.3 节看到的那样。接近协议栈顶部，恰好位于 IP 网络层正下方的是 PPP。该协议与我们刚刚考察过在 SONET 上传输数据包的 PPP 相同。它以同样的方式来建立和配置链路，以及运载 IP 数据包。

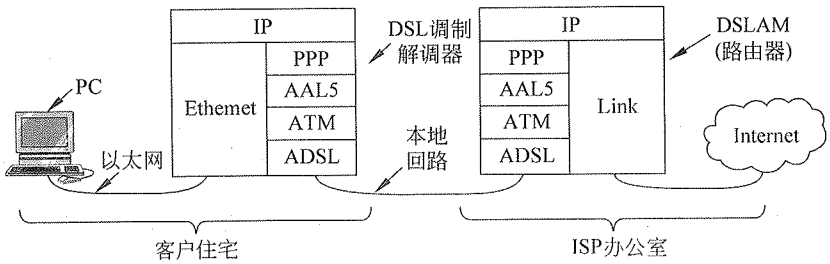


图 3-26　ADSL 协议栈

在 ADSL 和 PPP 两者之间是的 ATM 和 AAL5。这些是我们以前从未见过的新协议。**异步传输模式**（ATM，Asynchronous Transfer Mode）早在 20 世纪 90 年代初就被设计了出来，并且以令人难以置信的炒作方式公布于众。它承诺的网络技术可以解决世界上的电信问题，它把语音、数据、有线电视、电报、信鸽、串起来的罐头、击鼓，以及其他所有一切合并成一个综合系统，声称能为每个人做每件事。但事实并非如此。在很大程度上，ATM 的问题类似于我们介绍的 OSI 协议，也就是说，错误的时机、错误的技术、错误的实施以及错误的政治。不过，ATM 的命运远好于 OSI。虽然它并没有在世界各地流行起来，但它仍然被广泛应用在合适的领域，包括诸如 DSL 那样的宽带接入线路以及电话网络内部的广域网链路。

ATM 是一种链路层，它的传输基于固定长度的信息**信元**（cell）。其名称中的“异步”意味着这些信元并不总是以连续的方式发送，这与 SONET 在同步线路上发送的方式不同。只有当出现需要运载的信息时，才发送信元。ATM 是一种面向连接的技术。每个信元在它的头部带有**虚电路**（virtual circuit）标识符，交换设备根据此标识符沿着连接建立的路径转发信元。

每个信元 53 字节长，由一个 48 字节的有效载荷以及一个 5 字节的头组成。利用这些小信元，ATM 可以为不同用户灵活划分物理层链路的带宽。这种能力对网络应用非常有用，例如，在同一条链路上发送语音信息和数据信息，不会因为出现很长的数据包而导致声音样本值的延迟变化过大。信元长度的与众不同选择（例如，相比之下，更自然的选择是 2 的幂次方）恰好说明 ATM 的设计是多么的政治化。选择有效载荷的长度为 48 字节正是解决欧洲和美国分歧的一个妥协，欧洲希望信元取 32 字节长，而美国方面则希望信元取 64 字节长。有关 ATM 的简要概述请参考（Siu 和 Jain，1995）。

为了在 ATM 网络上发送数据，需要将数据映射成一系列的信元。这个映射由 ATM 适配层完成，映射过程称为**分段**（segmentation）和**重组**（reassembly）。针对不同的服务定义了几个适配层，从周期性的声音样本到数据包数据。其中一个主要用于数据包数据的适配层是 **ATM 适应层 5**（AAL5，ATM Adaptation Layer 5）。

一个 AAL5 帧如图 3-27 所示。除了帧头，它还有一个帧尾，给出了帧的长度和用于错误检测的 4 字节 CRC。很自然，这里的 CRC 与 PPP 和诸如以太网那样的 IEEE 802 局域网使用的相同。（Wang 和 Crowcroft，1992）已表明该 CRC 强大到足以检测出非传统错误（诸

如信元重新排序）。除了有效载荷外，AAL5 帧还需要被填充。填充的目的是使得帧的总长度是 48 字节的倍数，以便帧被均匀地划分成多个信元。这里不需要地址，因为每个信元携带的虚电路标识将引导它到达正确的目的地。

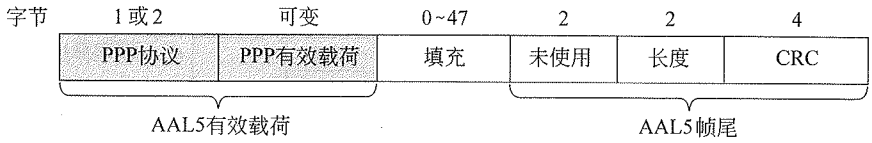


图 3-27 运载 PPP 数据的 AAL5 帧

现在，我们已经简单描述了 ATM，但仅涉及在 ADSL 的情况下 PPP 如何使用 ATM。这项工作由另一个称为 ATM 上的 PPP（PPPoA，PPP over ATM）标准完成。这个标准不是真正意义上的协议（所以没出现在图 3-26 中），但详细说明了 PPP 和 AAL5 帧如何工作的，该标准由 RFC2364 描述（Gross 等，1998）。

只有 PPP 协议和有效载荷字段被放置在 AAL5 的有效载荷，如图 3-27 所示。协议字段告诉远端 DSLAM 有效载荷里包含的是一个 IP 数据包，或是另一种协议的数据包（比如 LCP 协议）。远端当然知道信元包含了 PPP 信息，因为 ATM 虚电路就是为这个目的而建立的。

在 AAL5 帧内，PPP 的成帧功能并不是必需的，因为在这里起不到任何作用；ATM 和 AAL5 早已提供了成帧的功能。更多的成帧考虑将毫无价值。同样，PPP 的 CRC 也没有必要，因为 AAL5 已经包括了相同的 CRC。这个错误检测机制补充了 ADSL 的物理层功能，ADSL 物理层采用 Reed-Solomon 码来检测错误，同时还用 1 字节的 CRC 来检测遗漏的任何错误。该方案具有比在 SONET 上发送数据包更为复杂的错误恢复机制，因为 ADSL 是一种非常嘈杂的信道。

3.6 本章总结

数据链路层的任务是将物理层提供的原始比特流转换成由网络层使用的帧流。链路层为这样的帧流提供不同程度的可靠性，范围从无连接无确认的服务到可靠的面向连接服务不等。

链路层采用的成帧方法各种各样，包括字节计数、字节填充和比特填充。数据链路协议提供了差错控制机制来检测或纠正传输受损的帧，以及重新传输丢失的帧。为了防止快速发送方淹没慢速接收方，数据链路协议还提供了流量控制机制。滑动窗口机制被广泛用来以一种简单方式集成差错控制和流量控制两大机制。当窗口大小为 1 个数据包时，则协议是停-等式的。

纠错和检错码使用不同的数学技术把冗余信息添加到消息中。卷积码和里德所罗门码被广泛用于纠错，低密度校验码越来越受到欢迎。实际使用的检错码包括循环冗余校验和校验和两种。所有这些编码方法不仅可被应用在链路层，而且也可用在物理层和更高的层次。

我们考察了一系列协议，这些协议在更现实的假设下通过确认和重传，或者 ARQ（自

动重复请求）机制为上层提供了一个可靠的链路层。从一个无错误的环境开始，即接收方可以处理传送给它的任何帧，我们引出了流量控制，然后是带有序号的差错控制和停-等式算法。然后，我们使用滑动窗口算法允许双向通信，并引出捎带确认的概念。最后给出两个协议把多个帧的传输管道化，以此来防止发送方被一个有着漫长传播延迟的链路所阻塞。接收方可以丢弃所有乱序的帧，或者为了获得更大的带宽效率而缓冲这些乱序帧，并且给发送方反馈否定确认。前一种策略是回退 n 协议，后一种策略是选择重传协议。

Internet 使用 PPP 作为点到点线路上的主要数据链路协议。PPP 协议提供了无连接的无确认服务，使用标志字节区分帧的边界，至于错误检测则采用 CRC。该协议通常被用在运载数据包的一系列链路上，包括广域网中的 SONET 链路和家庭 ADSL 链路。

习　题

1. 一个上层数据包被分成 10 个帧，每一帧有 80%的机会无损地到达目的地。如果数据链路协议没有提供错误控制，试问，该报文平均需要发送多少次才能完整地到达接收方？
2. 数据链路协议使用了下面的字符编码：

   A: 01000111；　B: 11100011；　FLAG: 01111110；　ESC: 11100000

   为了传输一个包含 4 个字符的帧：A B ESC FLAG，试问使用下面的成帧方法时所发送的比特序列（用二进制表达）是什么？

   （a）字节计数。

   （b）字节填充的标志字节。

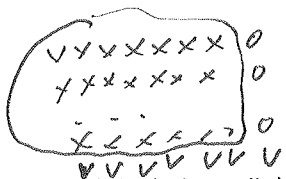
   （c）比特填充的首尾标志字节。
3. 一个数据流中出现了这样的数据段：A B ESC C ESC FLAG FLAG D，假设采用本章介绍的字节填充算法，试问经过填充之后的输出是什么？
4. 试问字节填充算法的最大开销是多少？
5. 你的一个同学 Scrooge 指出每一帧的结束处用一个标志字节而下一帧的开始处又用另一个标志字节，这种做法非常浪费空间。用一个标志字节也可以完成同样的任务，这样可以节省下来一个字节。试问你同意这种观点吗？
6. 需要在数据链路层上被发送一个比特串：0111101111101111110。试问，经过比特填充之后实际被发送出去的是什么？
7. 试问在什么样的环境下，一个开环协议（比如海明码）有可能比本章通篇所讨论的反馈类协议更加适合？
8. 为了提供比单个奇偶位更强的可靠性，一种检错编码方案如下：用一个奇偶位来检查所有奇数序号的位，用另一个奇偶位来检查所有偶数序号的位。请问这种编码方案的海明距离是多少？
9. 假设使用海明码来传输 16 位的报文。试问，需要多少个校验位才能确保接收方能同时检测并纠正单个比特错误？对于报文 1101001100110101，试给出传输的比特模式。假设在海明码中使用了偶校验。
10. 接收方收到一个 12 位的海明码，其 16 进制值为 0xE4F。试问该码的原始值是多少？

假设至多发生了一位错。

11. 检测错误的一种方法是按 n 行、每行 k 位来传输数据，并且在每行和每列加上奇偶位。其中最右下角是一个校验其所在行和列的奇偶位。试问这种方案能检测出所有的 1 位错吗？2 位错误呢？3 位错误呢？请说明这种方案无法检测出某些 4 位错误。
12. 假设数据以块形式传输，每块大小 1000 比特。试问，在什么样的最大错误率下，错误检测和重传机制（每块 1 个校验位）比使用海明码更好？假设比特错误相互独立，并且在重传过程中不会发生比特错误。
13. 一个具有 n 行 k 列的块使用水平和垂直奇偶校验位进行差错检测。假设正好有 4 位由于传输错误被反转。请推导出该错误无法被检测出来的概率表达式。
14. 利用如图 3-7 所示的卷积码，当输入序列是 10101010（从左到右）并且内部状态初始化为全零时，试问，输出序列是什么？
15. 假设使用 Internet 校验和（4 位字）来发送一个消息 1001110010100011。试问校验和的值是什么？
16. x^7+x^5+1 被生成多项式 x^3+1 除，试问，所得余数是什么？
17. 使用本章介绍的标准 CRC 方法传输比特流 10011101。生成多项式为 x^3+1。试问实际传输的位串是什么？假设左边开始的第三个比特在传输过程中变反了。请说明这个错误可以在接收方被检测出来。给出一个该比特流传输错误的实例，使得接收方无法检测出该错误。
18. 发送一个长度为 1024 位的消息，其中包含 992 个数据位和 32 位 CRC 校验位。CRC 计算采用了 IEEE 802 标准，即 32 阶的 CRC 多项式。对于下面每种情况，说明在信息传输中出现的错误能否被接收方检测出来：
    （a）只有一位错误。
    （b）有 2 个孤立的一位错误。
    （c）有 18 个孤立的一位错误。
    （d）有 47 个孤立的一位错误。
    （e）有一个长度为 24 位的突发错误。
    （f）有一个长度为 35 位的突发错误。
19. 在 3.3.3 节讨论 ARQ 协议时，概述了一种场景，由于确认帧的丢失导致接收方接收了两个相同的帧。试问，如果不会出现丢帧（消息或者确认），接收方是否还有可能收到同一帧的多个副本？
20. 考虑一个具有 4 kbps 速率和 20 毫秒传输延迟的信道。试问帧的大小在什么范围内，停-等式协议才能获得至少 50%的效率？
21. 在协议 3 中，当发送方的计时器已经在运行时，它还有可能启动该计时器吗？如果可能，试问这种情况是如何发生的？如果不可能，试问为什么不可能？
22. 使用协议 5 在一条 3000 千米长的 T1 中继线上传输 64 字节的帧。如果信号的传播速度为 6 微秒/千米，试问序号应该有多少位？
23. 想象一个滑动窗口协议，它的序号占用的位数相当多，使得序号几乎永远不会回转。试问 4 个窗口边界和窗口大小之间必须满足什么样的关系？假设这里窗口的大小固定不变，并且发送方和接收方的窗口大小相同。

24. 在协议 5 中，如果 between 过程检查的条件是 a≤b≤c，而不是 a≤b<c，试问这对于协议的正确性和效率有影响吗？请解释你的答案。
25. 在协议 6 中，当一个数据帧到达时，需要检查它的序号是否不同于期望的序号，并且 no_nak 为真。如果这两个条件都成立，则发送一个 NAK；否则，启动辅助定时器。假定 else 子句被省略掉，试问这种改变会影响协议的正确性吗？
26. 假设将协议 6 中接近尾部的内含三条语句的 while 循环去掉，试问这样会影响协议的正确性吗？或者只是仅仅影响了协议的性能？请解释你的答案。
27. 地球到一个遥远行星的距离大约是 9×10^{10} 米。如果采用停-等式协议在一条 64 Mbps 的点到点链路上传输帧，试问信道的利用率是多少？假设帧的大小为 32 KB，光的速度是 3×10^{8} m/s。
28. 在前面的问题中，假设用滑动窗口协议来代替停-等式协议。试问多大的发送窗口才能使得链路利用率为 100%？发送方和接收方的协议处理时间可以忽略不计。
29. 在协议 6 中，frame_arrival 代码中有一部分是用来处理 NAK 的。如果入境帧是一个 NAK，并且另一个条件也满足，则这部分代码会被调用。请给出一个场景，在此场景下另一个条件非常关键。
30. 考虑在一条 1 Mbps 的完美线路（即无差错）上使用协议 6。帧的最大长度为 1000 位。每过 1 秒产生一个新数据包。超时间隔为 10 毫秒。如果取消特殊的确认计时器，那么就会发生不必要的超时事件。试问，平均报文要被传输多少次？
31. 在协议 6 中，MAX_SEQ=2^n-1。虽然这种情况显然是希望尽可能利用头部空间，但我们无法证明这个条件是基本的。试问，当 MAX_SEQ=4 时协议也能够正确工作吗？
32. 利用地球同步卫星在一个 1 Mbps 的信道上发送长度为 1000 位的帧，该信道的传播延迟为 270 毫秒。确认总是被捎带在数据帧中。帧头非常短，序号使用了 3 位。试问，在下面的协议中，可获得的最大信道利用率是多少？

    （a）停等式？
    （b）协议 5？
    （c）协议 6？



33. 在一个负载很重的 50 kbps 卫星信道上使用协议 6，数据帧包含长度为 40 位的头和长度为 3960 位的数据，试问浪费的带宽开销（帧头和重传）占多少比例？假设从地球到卫星的信号传播时间为 270 毫秒。ACK 帧永远不会发生。NAK 帧长 40 位。数据帧的错误率是 1%，NAK 帧的错误率忽略不计。序号占 8 位。
34. 考虑在一个无错的 64kbps 卫星信道上单向发送 512 字节长的数据帧，来自另一个方向反馈的确认帧非常短。对于窗口大小为 1、7、15 和 127 的情形，试问最大的吞吐量分别是多少？从地球到卫星的传播时间为 270 毫秒。
35. 在一条 100 千米长的线缆上运行 T1 数据速率。线缆的传播速度是真空中光速的 2/3。试问线缆中可以容纳多少位？
36. PPP 使用字节填充而不是比特填充，这样做的目的是防止有效载荷字段偶尔出现的标志字节造成的混乱。请至少给出一个理由说明 PPP 为什么这么做。
37. 试问，使用 PPP 发送一个 IP 数据包的最低开销是多少？如果只计算 PPP 自身引入的开销，而不计 IP 头开销，试问最大开销又是多少？

38. 在本地回路上使用ADSL协议栈来发送一个长为100个字节的IP数据包。试问一共发出去多少个ATM信元？请简要描述这些信元的内容。

39. 本实验练习的目标是用本章描述的标准CRC算法实现一个错误检测机制。编写两个程序：generator和verifier。generator程序从标准输入读取一行ASCII文本，该文本包含由0和1组成的n位消息。第二行是个k位多项式，也是以ASCII码表示。程序输出到标准输出设备上的是一行ASCII码，由n+k个0和1组成，表示被发送的消息。然后，它输出多项式，就像它输入的那样。verifier程序读取generator程序的输出，并输出一条消息指示正确与否。最后，再写一个程序alter，它根据参数（从最左边开始1的比特数）反转第一行中的比特1，但正确复制两行中的其余部分。通过键入：

```
generator <file | verifier
```

你应该能看到正确的消息，但键入：

```
generator <file | alter arg | verifier
```

你只能得到错误的消息。