否出现了差错。实际上，计算检验和时，除了 UDP 报文段以外还包括了 IP 首部的一些字段。但是我们忽略这些细节，以便能从整体上看问题。下面我们将讨论检验和的计算。在 6.2 节中将描述差错检测的基本原理。长度字段指明了包括首部在内的 UDP 报文段长度（以字节为单位）。

3.3.2 UDP 检验和

UDP 检验和提供了差错检测功能。这就是说，检验和用于确定当 UDP 报文段从源到达目的地移动时，其中的比特是否发生了改变（例如，由于链路中的噪声干扰或者存储在路由器中时引入问题）。发送方的 UDP 对报文段中的所有 16 比特字的和进行反码运算，求和时遇到的任何溢出都被回卷。得到的结果被放在 UDP 报文段中的检验和字段。下面给出一个计算检验和的简单例子。在 RFC 1071 中可以找到有效实现的细节，还可在［Stone 1998；Stone 2000］中找到它处理真实数据的性能。举例来说，假定我们有下面 3 个 16 比特的字：

0110011001100000
0101010101010101
1000111100001100

这些 16 比特字的前两个之和是：

0110011001100000
0101010101010101
1011101110110101

再将上面的和与第三个字相加，得出：

1011101110110101
1000111100001100
0100101011000010

注意到最后一次加法有溢出，它要被回卷。反码运算就是将所有的 0 换成 1，所有的 1 转换成 0。因此，该和 0100101011000010 的反码运算结果是 1011010100111101，这就变为了检验和。在接收方，全部的 4 个 16 比特字（包括检验和）加在一起。如果该分组中没有引入差错，则显然在接收方处该和将是 1111111111111111。如果这些比特之一是 0，那么我们就知道该分组中已经出现了差错。

你可能想知道为什么 UDP 首先提供了检验和，就像许多链路层协议（包括流行的以太网协议）也提供了差错检测那样。其原因是不能保证源和目的之间的所有链路都提供差错检测；这就是说，也许这些链路中的一条可能使用没有差错检测的协议。此外，即使报文段经链路正确地传输，当报文段存储在某台路由器的内存中时，也可能引入比特差错。在既无法确保逐链路的可靠性，又无法确保内存中的差错检测的情况下，如果端到端数据传输服务要提供差错检测，UDP 就必须在端到端基础上在运输层提供差错检测。这是一个在系统设计中被称颂的**端到端原则**（end-end principle）的例子［Saltzer 1984］，该原则表述为因为某种功能（在此时为差错检测）必须基于端到端实现：“与在较高级别提供这些功能的代价相比，在较低级别上设置的功能可能是冗余的或几乎没有价值的。”

因为假定 IP 是可以运行在任何第二层协议之上的，运输层提供差错检测作为一种保险措施是非常有用的。虽然 UDP 提供差错检测，但它对差错恢复无能为力。UDP 的某种

实现只是丢弃受损的报文段；其他实现是将受损的报文段交给应用程序并给出警告。

至此结束了关于 UDP 的讨论。我们将很快看到 TCP 为应用提供了可靠数据传输及 UDP 所不能提供的其他服务。TCP 自然要比 UDP 复杂得多。然而，在讨论 TCP 之前，我们后退一步，先来讨论一下可靠数据传输的基本原理是有用的。

3.4　可靠数据传输原理

在本节中，我们在一般场景下考虑可靠数据传输的问题。因为可靠数据传输的实现问题不仅在运输层出现，也会在链路层以及应用层出现，这时讨论它是恰当的。因此，一般性问题对网络来说更为重要。如果的确要将所有网络中最为重要的“前 10 个”问题排名的话，可靠数据传输将是名列榜首的候选者。在下一节中，我们将学习 TCP，尤其要说明 TCP 所采用的许多原理，而这些正是我们打算描述的内容。

图 3-8 图示说明了我们学习可靠数据传输的框架。为上层实体提供的服务抽象是：数据可以通过一条可靠的信道进行传输。借助于可靠信道，传输数据比特就不会受到损坏（由 0 变为 1，或者相反）或丢失，而且所有数据都是按照其发送顺序进行交付。这恰好就是 TCP 向调用它的因特网应用所提供的服务模型。

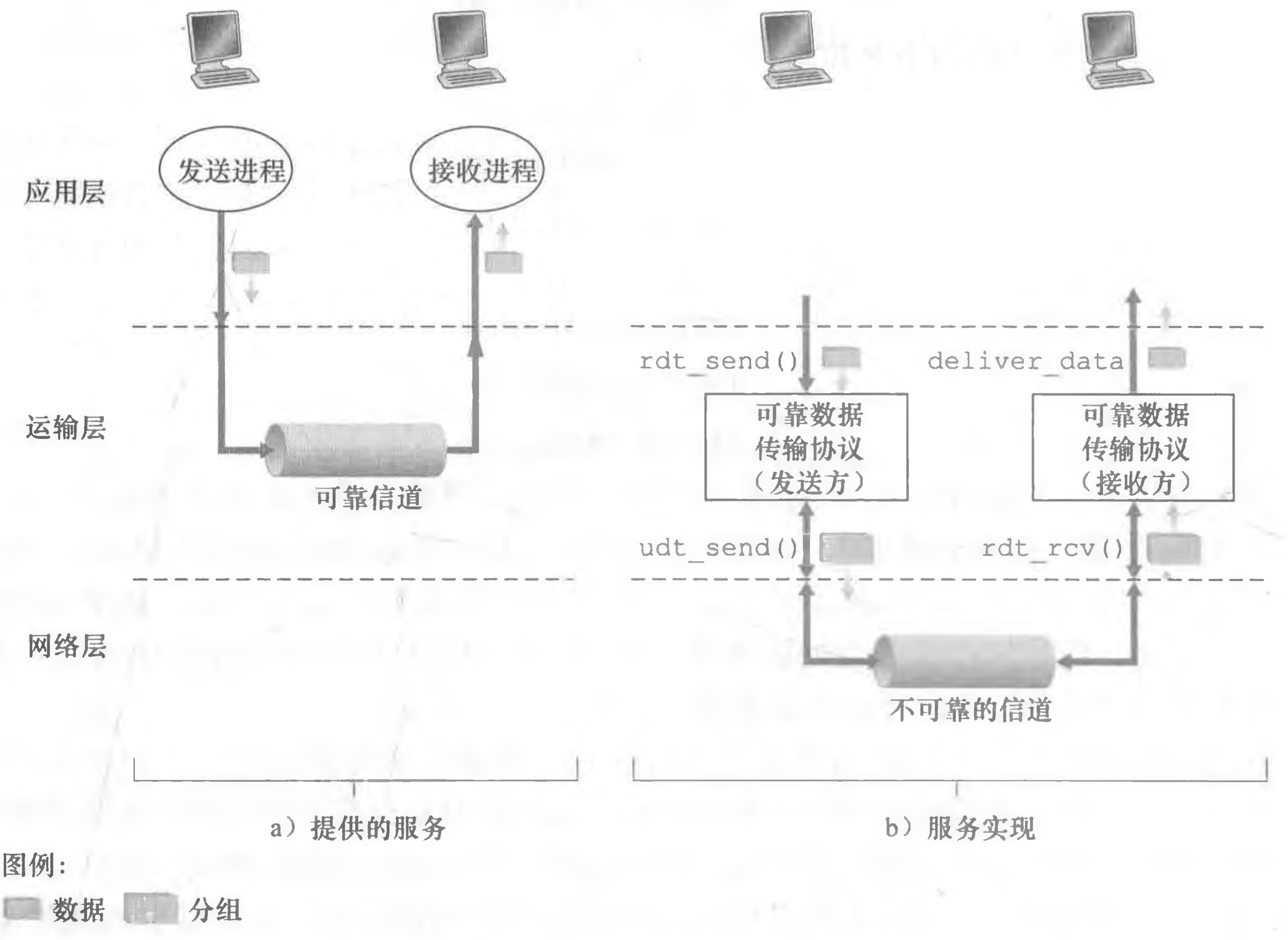


图 3-8　可靠数据传输：服务模型与服务实现

实现这种服务抽象是**可靠数据传输协议**（reliable data transfer protocol）的责任。由于可靠数据传输协议的下层协议也许是不可靠的，因此这是一项困难的任务。例如，TCP 是在不可靠的（IP）端到端网络层之上实现的可靠数据传输协议。更一般的情况是，两个可靠通信端点的下层可能是由一条物理链路（如在链路级数据传输协议的场合下）组成或是由一个全球互联网络（如在运输级协议的场合下）组成。然而，就我们的目的而言，我们

可将较低层直接视为不可靠的点对点信道。

在本节中，考虑到底层信道模型越来越复杂，我们将不断地开发一个可靠数据传输协议的发送方一侧和接收方一侧。例如，我们将考虑当底层信道能够损坏比特或丢失整个分组时，需要什么样的协议机制。这里贯穿我们讨论始终的一个假设是分组将以它们发送的次序进行交付，某些分组可能会丢失；这就是说，底层信道将不会对分组重排序。图 3-8b 图示说明了用于数据传输协议的接口。通过调用 rdt_send() 函数，上层可以调用数据传输协议的发送方。它将要发送的数据交付给位于接收方的较高层。（这里 rdt 表示可靠数据传输协议，_send 指示 rdt 的发送端正在被调用。开发任何协议的第一步就是要选择一个好的名字！）在接收端，当分组从信道的接收端到达时，将调用 rdt_rcv()。当 rdt 协议想要向较高层交付数据时，将通过调用 deliver_data() 来完成。后面，我们将使用术语“分组”而不用运输层的“报文段”。因为本节研讨的理论适用于一般的计算机网络，而不只是用于因特网运输层，所以这时采用通用术语“分组”也许更为合适。

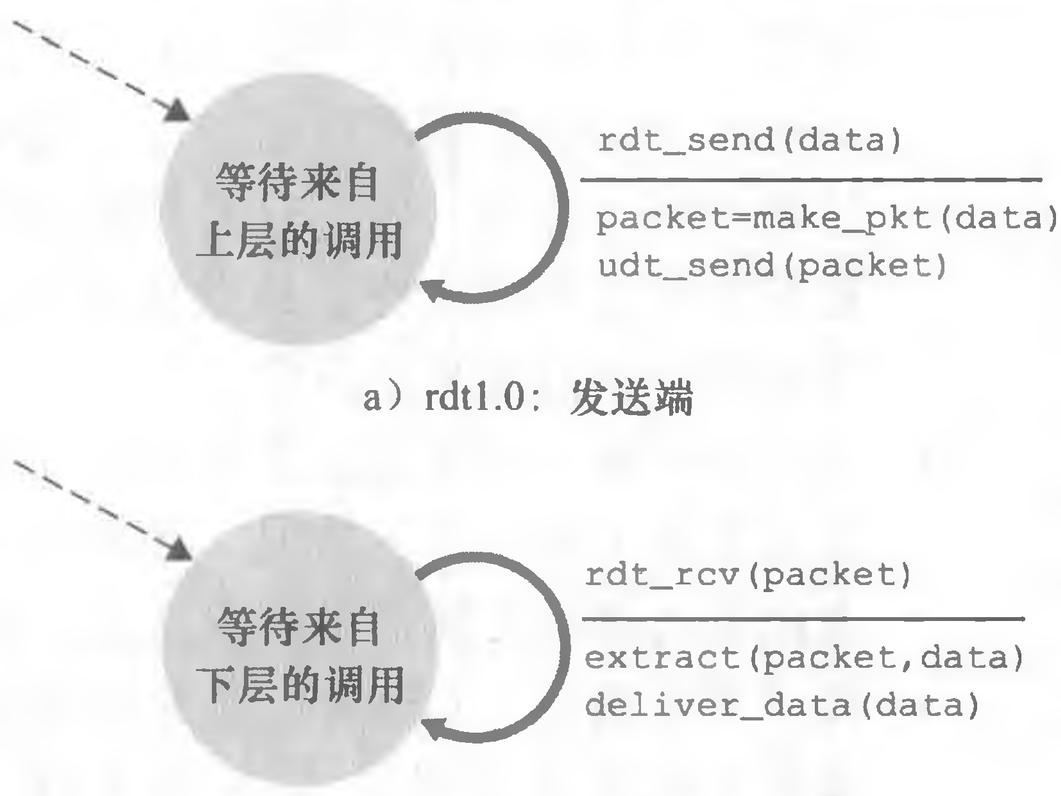
在本节中，我们仅考虑**单向数据传输**（unidirectional data transfer）的情况，即数据传输是从发送端到接收端的。可靠的**双向数据传输**（bidirectional data transfer）（即全双工数据传输）情况从概念上讲不会更难，但解释起来更为单调乏味。虽然我们只考虑单向数据传输，注意到下列事实是重要的，我们的协议也需要在发送端和接收端两个方向上传输分组，如图 3-8 所示。我们很快会看到，除了交换含有待传送的数据的分组之外，rdt 的发送端和接收端还需往返交换控制分组。rdt 的发送端和接收端都要通过调用 udt_send() 发送分组给对方（其中 udt 表示不可靠数据传输）。

3.4.1 构造可靠数据传输协议

我们现在一步步地研究一系列协议，它们一个比一个更为复杂，最后得到一个完美、可靠的数据传输协议。

1. 经完全可靠信道的可靠数据传输：rdt1.0

首先，我们考虑最简单的情况，即底层信道是完全可靠的。我们称该协议为 rdt1.0，该协议本身是简单的。图 3-9 显示了 rdt1.0 发送方和接收方的**有限状态机**（Finite-State Machine，FSM）的定义。图 3-9a 中的 FSM 定义了发送方的操作，图 3-9b 中的 FSM 定义了接收方的操作。注意到下列问题是重要的，发送方和接收方有各自的 FSM。图 3-9 中发送方和接收方的 FSM 每个都只有一个状态。FSM 描述图中的箭头指示了协议从一个状态变迁到另一个状态。（因为图 3-9 中的每个 FSM 都只有一个状态，因此变迁必定是从一个状态返回到自身；我们很快将看到更复杂的状态图。）引起变迁的事件显示在表示变迁的横线上方，事件发生时所采取的动作显示在横线下方。如果对一个事件没有动作，或没有就事件发生而采取了一个动作，我们将在横线上方或下方使用符号 Λ，以分别明确地表示缺少动作或事件。FSM 的初始状态用



a）rdt1.0：发送端

b）rdt1.0：接收端

图 3-9 rdt1.0：用于完全可靠信道的协议

虚线表示。尽管图3-9中的FSM只有一个状态，但马上我们就将看到多状态的FSM，因此标识每个FSM的初始状态是非常重要的。

rdt的发送端只通过rdt_send(data)事件接受来自较高层的数据，产生一个包含该数据的分组（经由make_pkt(data)动作），并将分组发送到信道中。实际上，rdt_send(data)事件是由较高层应用的过程调用产生的（例如，rdt_send()）。

在接收端，rdt通过rdt_rcv(packet)事件从底层信道接收一个分组，从分组中取出数据（经由extract(packet，data)动作），并将数据上传给较高层（通过deliver_data(data)动作）。实际上，rdt_rcv(packet)事件是由较低层协议的过程调用产生的（例如，rdt_rcv()）。

在这个简单的协议中，一个单元数据与一个分组没差别。而且，所有分组是从发送方流向接收方；有了完全可靠的信道，接收端就不需要提供任何反馈信息给发送方，因为不必担心出现差错！注意到我们也已经假定了接收方接收数据的速率能够与发送方发送数据的速率一样快。因此，接收方没有必要请求发送方慢一点！

2. 经具有比特差错信道的可靠数据传输：rdt2.0

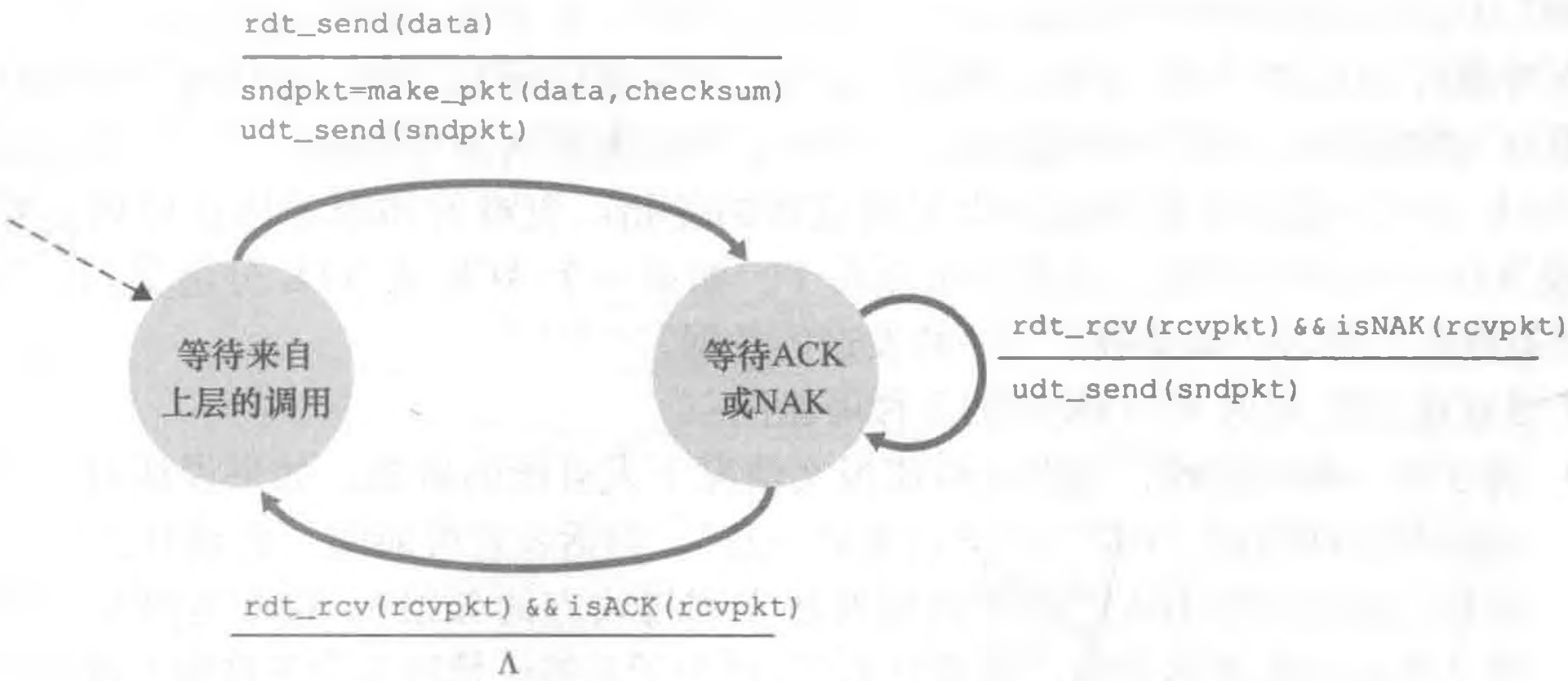
底层信道更为实际的模型是分组中的比特可能受损的模型。在分组的传输、传播或缓存的过程中，这种比特差错通常会出现在网络的物理部件中。我们眼下还将继续假定所有发送的分组（虽然有些比特可能受损）将按其发送的顺序被接收。

在研发一种经这种信道进行可靠通信的协议之前，首先考虑一下人们会怎样处理这类情形。考虑一下你自己是怎样通过电话口述一条长报文的。在通常情况下，报文接收者在听到、理解并记下每句话后可能会说“OK”。如果报文接收者听到一句含糊不清的话时，他可能要求你重复那句容易误解的话。这种口述报文协议使用了**肯定确认**（positive acknowledgment）（“OK”）与**否定确认**（negative acknowledgment）（“请重复一遍”）。这些控制报文使得接收方可以让发送方知道哪些内容被正确接收，哪些内容接收有误并因此需要重复。在计算机网络环境中，基于这样重传机制的可靠数据传输协议称为**自动重传请求**（Automatic Repeat reQuest，ARQ）**协议**。

重要的是，ARQ协议中还需要另外三种协议功能来处理存在比特差错的情况：

- *差错检测*。首先，需要一种机制以使接收方检测到何时出现了比特差错。前一节讲到，UDP使用因特网检验和字段正是为了这个目的。在第5章中，我们将更详细地学习差错检测和纠错技术。这些技术使接收方可以检测并可能纠正分组中的比特差错。此刻，我们只需知道这些技术要求有额外的比特（除了待发送的初始数据比特之外的比特）从发送方发送到接收方；这些比特将被汇集在rdt2.0数据分组的分组检验和字段中。
- *接收方反馈*。因为发送方和接收方通常在不同端系统上执行，可能相隔数千英里，发送方要了解接收方情况（此时为分组是否被正确接收）的唯一途径就是让接收方提供明确的反馈信息给发送方。在口述报文情况下回答的“肯定确认”（ACK）和“否定确认”（NAK）就是这种反馈的例子。类似地，我们的rdt2.0协议将从接收方向发送方回送ACK与NAK分组。理论上，这些分组只需要一个比特长；如用0表示NAK，用1表示ACK。
- *重传*。接收方收到有差错的分组时，发送方将重传该分组文。

图3-10说明了表示rdt2.0的FSM，该数据传输协议采用了差错检测、肯定确认与否定确认。



a）rdt2.0: 发送端

b）rdt2.0: 接收端

图 3-10 rdt2.0：用于具有比特差错信道的协议

rdt2.0 的发送端有两个状态。在最左边的状态中，发送端协议正等待来自上层传下来的数据。当 rdt_send(data)事件出现时，发送方将产生一个包含待发送数据的分组（sndpkt），带有检验和（例如，就像在 3.3.2 节讨论的对 UDP 报文段使用的方法），然后经由 udt_send(sndpkt)操作发送该分组。在最右边的状态中，发送方协议等待来自接收方的 ACK 或 NAK 分组。如果收到一个 ACK 分组（图 3-10 中符号 rdt_rcv(rcvpkt)&& isACK(rcvpkt)对应该事件），则发送方知道最近发送的分组已被正确接收，因此协议返回到等待来自上层的数据的状态。如果收到一个 NAK 分组，该协议重传上一个分组并等待接收方为响应重传分组而回送的 ACK 和 NAK。注意到下列事实很重要：当发送方处于等待 ACK 或 NAK 的状态时，它不能从上层获得更多的数据；这就是说，rdt_send()事件不可能出现；仅当接收到 ACK 并离开该状态时才能发生这样的事件。因此，发送方将不会发送一块新数据，除非发送方确信接收方已正确接收当前分组。由于这种行为，rdt2.0 这样的协议被称为**停等**（stop-and-wait）协议。

rdt2.0 接收方的 FSM 仍然只有单一状态。当分组到达时，接收方要么回答一个 ACK，要么回答一个 NAK，这取决于收到的分组是否受损。在图 3-10 中，符号 rdt_rcv(rcvpkt)&& corrupt(rcvpkt)对应于收到一个分组并发现有错的事件。

rdt2.0 协议看起来似乎可以运行了，但遗憾的是，它存在一个致命的缺陷。尤其是我们没有考虑到 ACK 或 NAK 分组受损的可能性！（在继续研究之前，你应该考虑怎样解决该问题。）遗憾的是，我们细小的疏忽并非像它看起来那么无关紧要。至少，我们需要在 ACK/NAK 分组中添加检验和比特以检测这样的差错。更难的问题是协议应该怎样纠正 ACK 或 NAK 分组中的差错。这里的难点在于，如果一个 ACK 或 NAK 分组受损，发送方无法知道接收方是否正确接收了上一块发送的数据。

考虑处理受损 ACK 和 NAK 时的 3 种可能性：

- 对于第一种可能性，考虑在口述报文情况下人可能的做法。如果说话者不理解来自接收方回答的“OK”或“请重复一遍”，说话者将可能问“你说什么?”（因此在我们的协议中引入了一种新型发送方到接收方的分组）。接收方则将复述其回答。但是如果说话者的“你说什么?”产生了差错，情况又会怎样呢？接收者不明白那句混淆的话是口述内容的一部分还是一个要求重复上次回答的请求，很可能回一句“你说什么?”。于是，该回答可能含糊不清了。显然，我们走上了一条困难重重之路。
- 第二种可能性是增加足够的检验和比特，使发送方不仅可以检测差错，还可恢复差错。对于会产生差错但不丢失分组的信道，这就可以直接解决问题。
- 第三种方法是，当发送方收到含糊不清的 ACK 或 NAK 分组时，只需重传当前数据分组即可。然而，这种方法在发送方到接收方的信道中引入了**冗余分组**（duplicate packet）。冗余分组的根本困难在于接收方不知道它上次所发送的 ACK 或 NAK 是否被发送方正确地收到。因此它无法事先知道接收到的分组是新的还是一次重传！

解决这个新问题的一个简单方法（几乎所有现有的数据传输协议中，包括 TCP，都采用了这种方法）是在数据分组中添加一新字段，让发送方对其数据分组编号，即将发送数据分组的**序号**（sequence number）放在该字段。于是，接收方只需要检查序号即可确定收到的分组是否一次重传。对于停等协议这种简单情况，1 比特序号就足够了，因为它可让接收方知道发送方是否正在重传前一个发送分组（接收到的分组序号与最近收到的分组序号相同），或是一个新分组（序号变化了，用模 2 运算“前向”移动）。因为目前我们假定信道不丢分组，ACK 和 NAK 分组本身不需要指明它们要确认的分组序号。发送方知道所接收到的 ACK 和 NAK 分组（无论是否是含糊不清的）是为响应其最近发送的数据分组而生成的。

图 3-11 和图 3-12 给出了对 rdt2.1 的 FSM 描述，这是 rdt2.0 的修订版。rdt2.1 的发送方和接收方 FSM 的状态数都是以前的两倍。这是因为协议状态此时必须反映出目前（由发送方）正发送的分组或（在接收方）希望接收的分组的序号是 0 还是 1。值得注意的是，发送或期望接收 0 号分组的状态中的动作与发送或期望接收 1 号分组的状态中的动作是相似的；唯一的不同是序号处理的方法不同。

协议 rdt2.1 使用了从接收方到发送方的肯定确认和否定确认。当接收到失序的分组时，接收方对所接收的分组发送一个肯定确认。如果收到受损的分组，则接收方将发送一个否定确认。如果不发送 NAK，而是对上次正确接收的分组发送一个 ACK，我们也能实现与 NAK 一样的效果。发送方接收到对同一个分组的两个 ACK（即接收**冗余 ACK**（duplicate ACK））后，就知道接收方没有正确接收到跟在被确认两次的分组后面的分组。rdt2.2 是在有比特差错信道上实现的一个无 NAK 的可靠数据传输协议，如图 3-13 和

图 3-14所示。rdt2. 1 和 rdt2. 2 之间的细微变化在于，接收方此时必须包括由一个 ACK 报文所确认的分组序号（这可以通过在接收方 FSM 中，在 make_pkt() 中包括参数 ACK 0 或 ACK 1 来实现），发送方此时必须检查接收到的 ACK 报文中被确认的分组序号（这可通过在发送方 FSM 中，在 isACK() 中包括参数 0 或 1 来实现）。

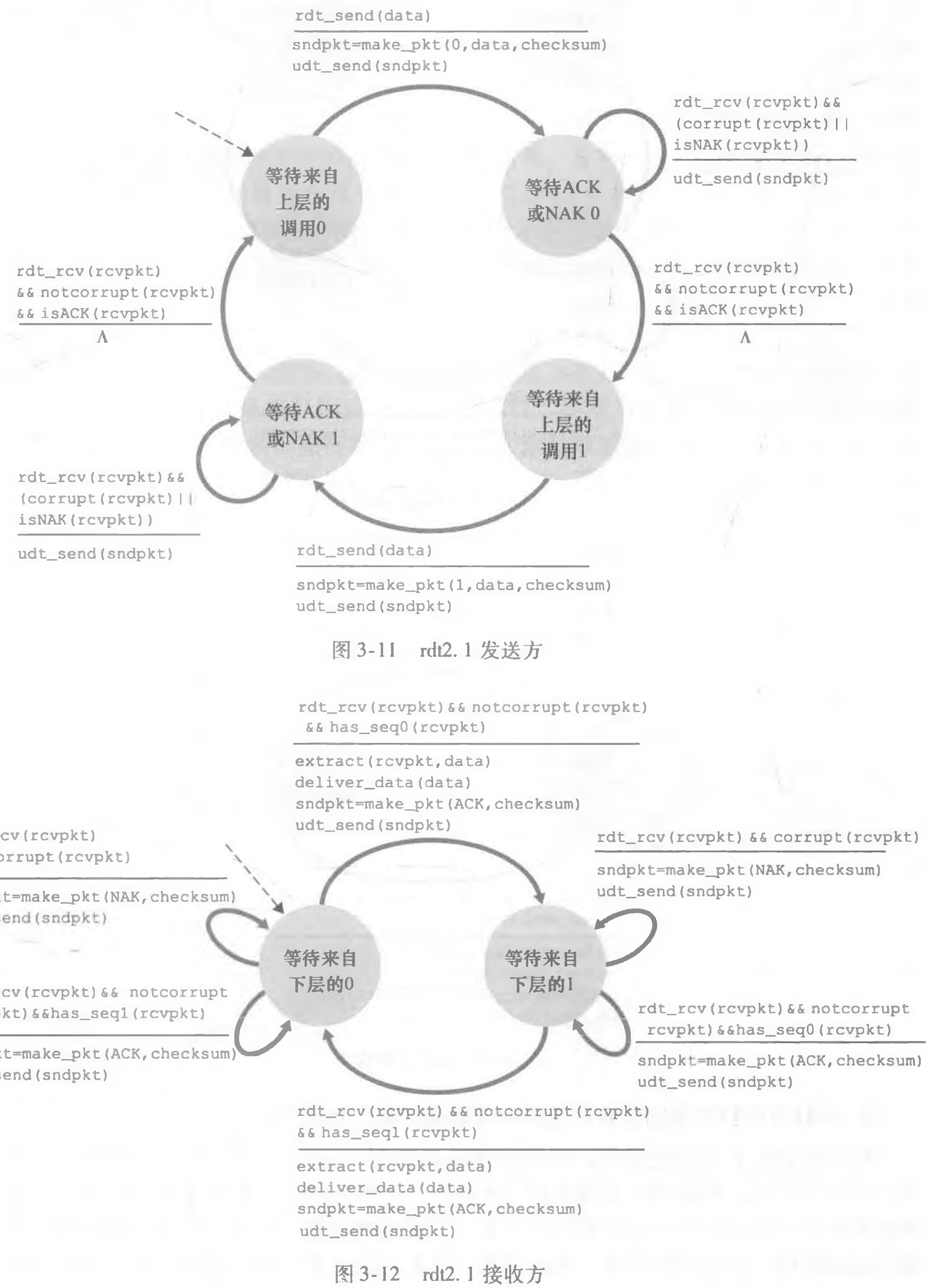


图 3-11 rdt2. 1 发送方

图 3-12 rdt2. 1 接收方

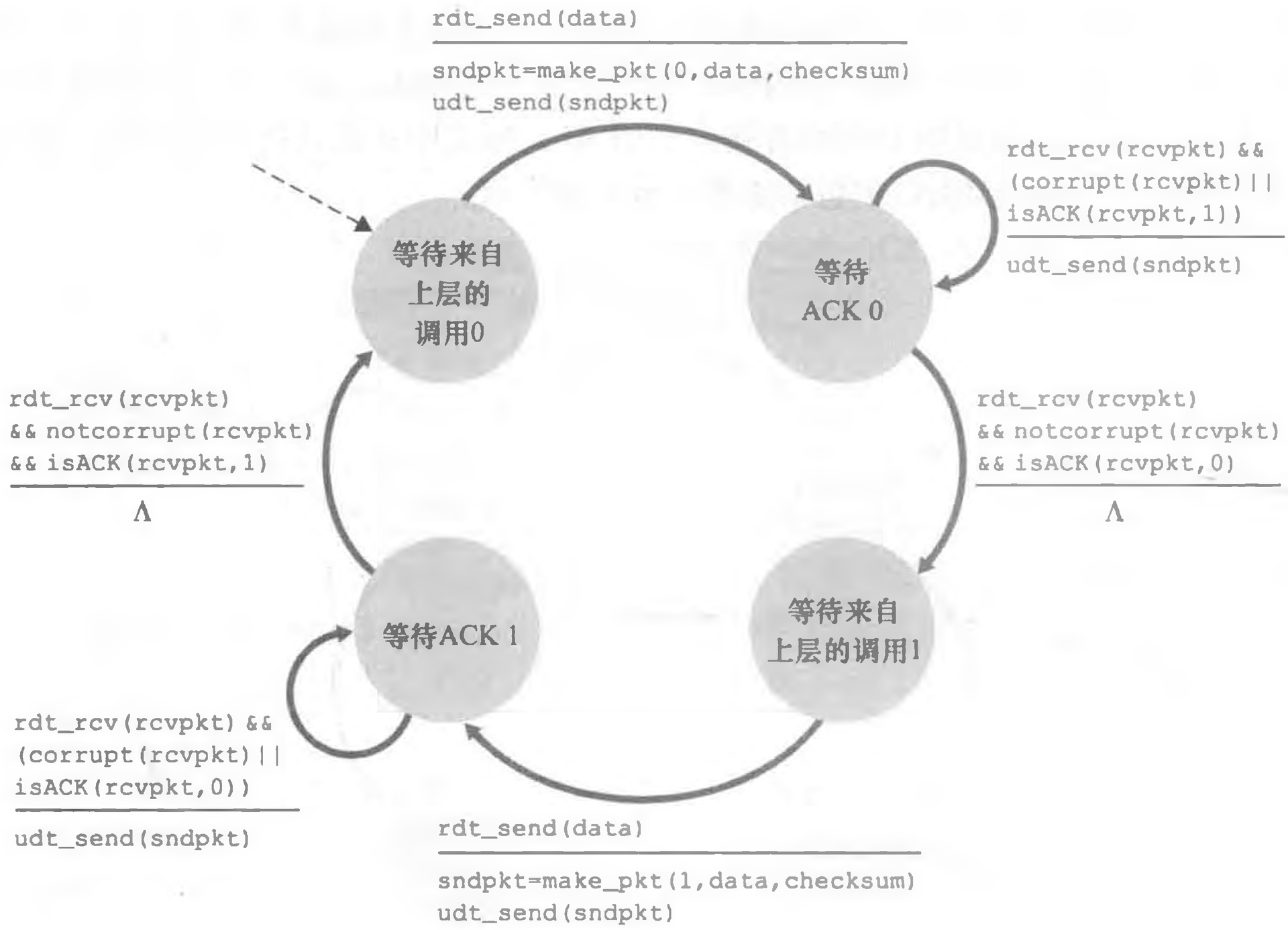


图 3-13　rdt2.2 发送方

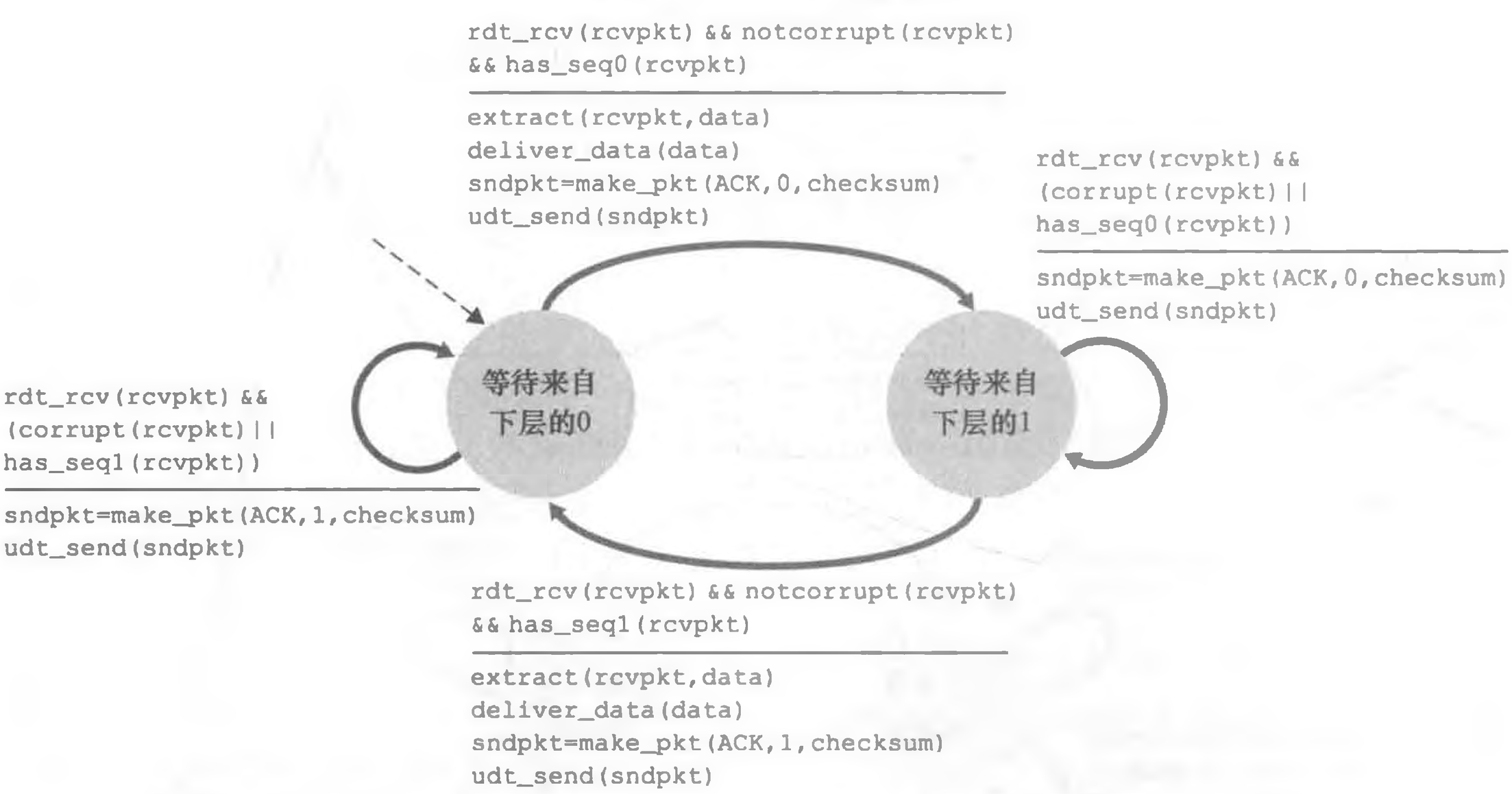


图 3-14　rdt2.2 接收方

3. 经具有比特差错的丢包信道的可靠数据传输：rdt3.0

现在假定除了比特受损外，底层信道还会丢包，这在今天的计算机网络（包括因特网）中并不罕见。协议现在必须处理另外两个关注的问题：怎样检测丢包以及发生丢包后该做些什么。在 rdt2.2 中已经研发的技术，如使用检验和、序号、ACK 分组和重传等，使我们能给出后一个问题的答案。为解决第一个关注的问题，还需增加一种新的协议机制。

有很多可能的方法用于解决丢包问题（在本章结尾的习题中研究了几种其他方法）。

这里，我们让发送方负责检测和恢复丢包工作。假定发送方传输一个数据分组，该分组或者接收方对该分组的 ACK 发生了丢失。在这两种情况下，发送方都收不到应当到来的接收方的响应。如果发送方愿意等待足够长的时间以便确定分组已丢失，则它只需重传该数据分组即可。你应该相信该协议确实有效。

但是发送方需要等待多久才能确定已丢失了某些东西呢？很明显发送方至少需要等待这样长的时间：即发送方与接收方之间的一个往返时延（可能会包括在中间路由器的缓冲时延）加上接收方处理一个分组所需的时间。在很多网络中，最坏情况下的最大时延是很难估算的，确定的因素非常少。此外，理想的协议应尽可能快地从丢包中恢复出来；等待一个最坏情况的时延可能意味着要等待一段较长的时间，直到启动差错恢复为止。因此实践中采取的方法是发送方明智地选择一个时间值，以判定可能发生了丢包（尽管不能确保）。如果在这个时间内没有收到 ACK，则重传该分组。注意到如果一个分组经历了一个特别大的时延，发送方可能会重传该分组，即使该数据分组及其 ACK 都没有丢失。这就在发送方到接收方的信道中引入了**冗余数据分组**（duplicate data packet）的可能性。幸运的是，rdt2.2 协议已经有足够的功能（即序号）来处理冗余分组情况。

从发送方的观点来看，重传是一种万能灵药。发送方不知道是一个数据分组丢失，还是一个 ACK 丢失，或者只是该分组或 ACK 过度延时。在所有这些情况下，动作是同样的：重传。为了实现基于时间的重传机制，需要一个**倒计数定时器**（countdown timer），在一个给定的时间量过期后，可中断发送方。因此，发送方需要能做到：①每次发送一个分组（包括第一次分组和重传分组）时，便启动一个定时器。②响应定时器中断（采取适当的动作）。③终止定时器。

图 3-15 给出了 rdt3.0 的发送方 FSM，这是一个在可能出错和丢包的信道上可靠传

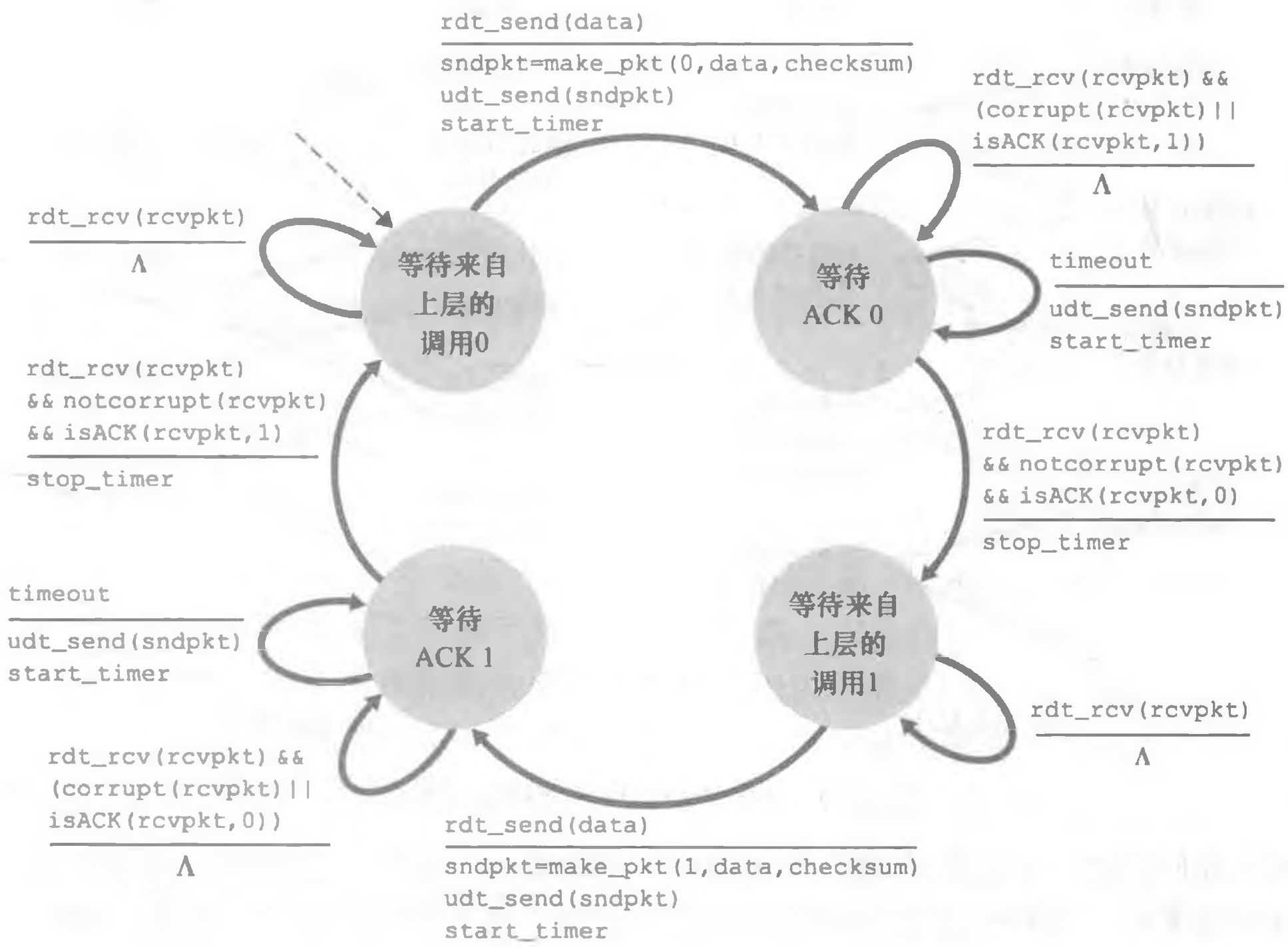


图 3-15 rdt3.0 发送方

输数据的协议；在课后习题中，将请你提供 rdt3.0 的接收方 FSM。图 3-16 显示了在没有丢包和延迟分组情况下协议运作的情况，以及它是如何处理数据分组丢失的。在图 3-16 中，时间从图的顶部朝底部移动；注意到一个分组的接收时间必定迟于一个分组的发送时间，这是因为发送时延与传播时延之故。在图 3-16b ~ d 中，发送方括号部分表明了定时器的设置时刻以及随后的超时。本章后面的习题探讨了该协议几个更细微的方面。因为分组序号在 0 和 1 之间交替，因此 rdt3.0 有时被称为**比特交替协议**（alternating-bit protocol）。

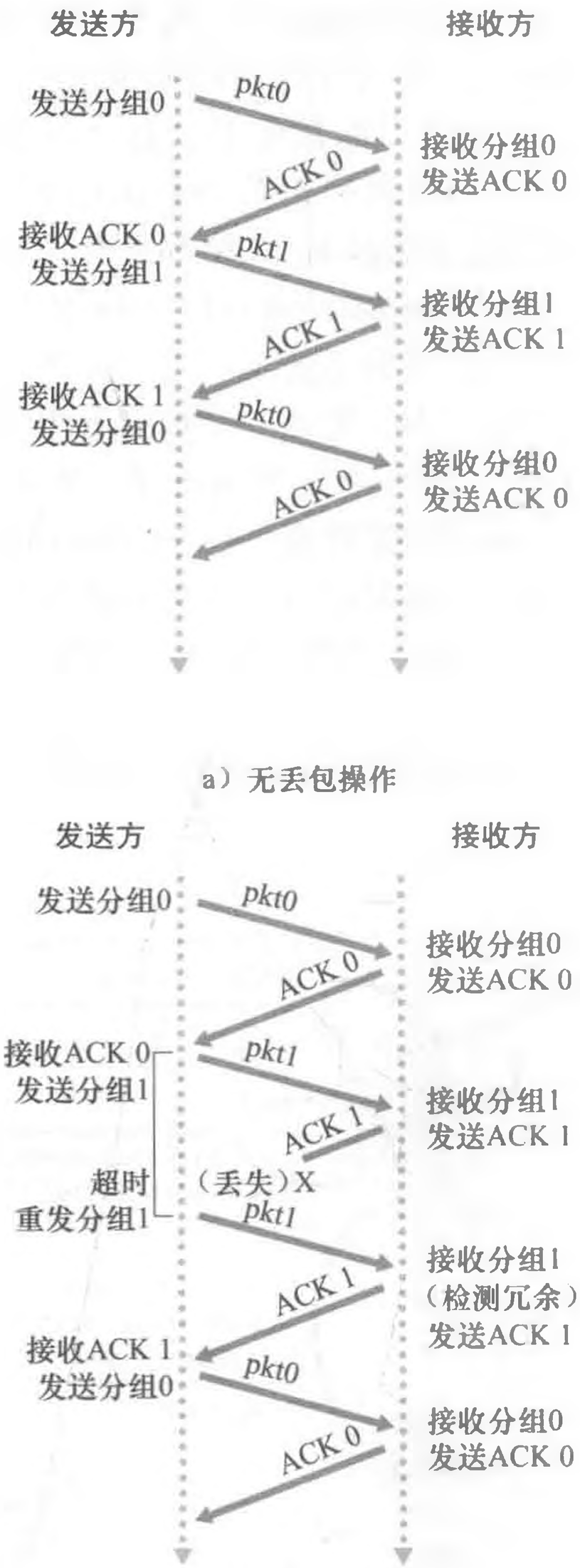


图 3-16　rdt3.0 的运行，比特交替协议

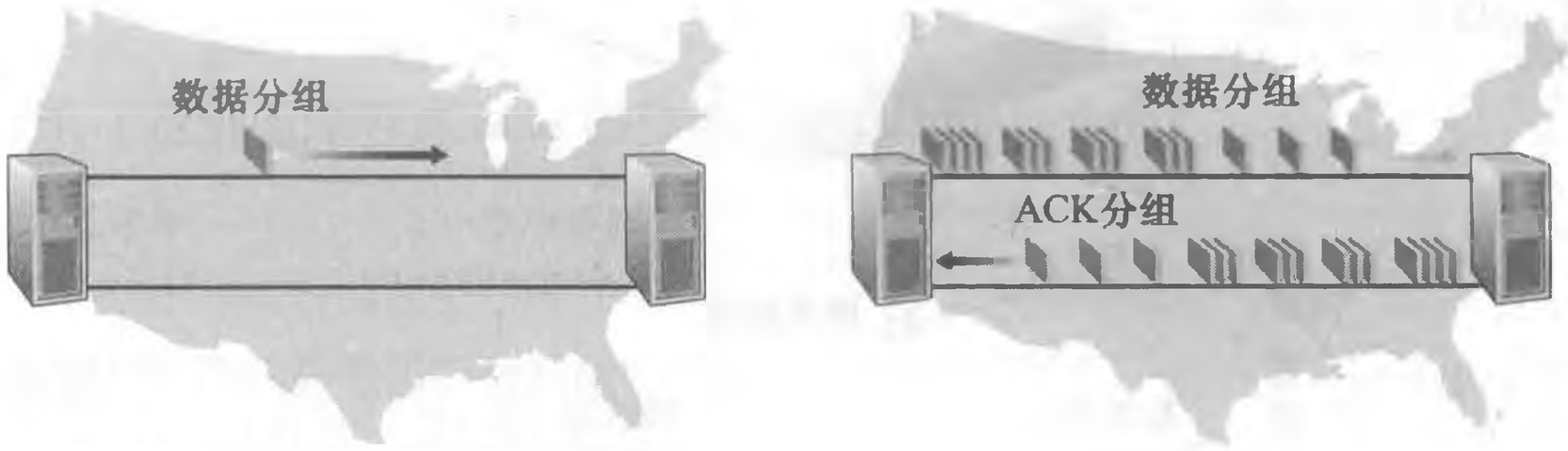
现在我们归纳一下数据传输协议的要点。在检验和、序号、定时器、肯定和否定确认分组这些技术中，每种机制都在协议的运行中起到了必不可少的作用。至此，我们得到了一个可靠数据传输协议！

3.4.2 流水线可靠数据传输协议

rdt3.0 是一个功能正确的协议，但并非人人都对它的性能满意，特别是在今天的高速网络中更是如此。rdt3.0 性能问题的核心在于它是一个停等协议。

为了评价该停等行为对性能的影响，可考虑一种具有两台主机的理想化场合，一台主机位于美国西海岸，另一台位于美国东海岸，如图 3-17 所示。在这两个端系统之间的光速往返传播时延 RTT 大约为 30 毫秒。假定彼此通过一条发送速率 R 为 1Gbps（每秒 10^9 比特）的信道相连。包括首部字段和数据的分组长 L 为 1000 字节（8000 比特），发送一个分组进入 1Gbps 链路实际所需时间是：

$$t_{\text{trans}} = \frac{L}{R} = \frac{8000\text{bit/pkt}}{10^9\text{bit/s}} = 8\mu\text{s/pkt}$$



a）一个运行中的停等协议　　b）一个运行中的流水线协议

图 3-17 停等协议与流水线协议

图 3-18a 显示了对于该停等协议，如果发送方在 $t=0$ 时刻开始发送分组，则在 $t=L/R=8\mu s$ 后，最后 1 比特数据进入了发送端信道。该分组经过 15ms 的穿越国家的旅途后到达接收端，该分组的最后 1 比特在时刻 $t=\text{RTT}/2+L/R=15.008\text{ms}$ 时到达接收方。为了简化起见，假设 ACK 分组很小（以便我们可以忽略其发送时间），接收方一旦收到一个数据分组的最后 1 比特后立即发送 ACK，ACK 在时刻 $t=\text{RTT}+L/R=30.008\text{ms}$ 时在发送方出现。此时，发送方可以发送下一个报文。因此，在 30.008ms 内，发送方的发送只用了 0.008ms。如果我们定义发送方（或信道）的**利用率**（utilization）为：发送方实际忙于将发送比特送进信道的那部分时间与发送时间之比，图 3-18a 中的分析表明了停等协议有着非常低的发送方利用率 U_{sender}：

$$U_{\text{sender}} = \frac{L/R}{\text{RTT}+L/R} = \frac{0.008}{30.008} = 0.000\,27$$

这就是说，发送方只有万分之 2.7 时间是忙的。从其他角度来看，发送方在 30.008ms 内只能发送 1000 字节，有效的吞吐量仅为 267kbps，即使有 1Gbps 的链路可用也是如此！想象一个不幸的网络经理购买了一条千兆比容量的链路，但他仅能得到 267kbps 吞吐量的情况！这是一个形象的网络协议限制底层网络硬件所提供的能力的图例。而且，我们还忽略了在发送方和接收方的底层协议处理时间，以及可能出现在发送方与接收方之间的任何中间路由器上的处理与排队时延。考虑到这些因素，将进一步增加时延，使其性能更糟糕。

这种特殊的性能问题的一个简单解决方法是：不以停等方式运行，允许发送方发送多个分组而无须等待确认，如在图 3-17b 图示的那样。图 3-18b 显示了如果发送方可以在等待确认之前发送 3 个报文，其利用率也基本上提高 3 倍。因为许多从发送方向接收方输送

的分组可以被看成是填充到一条流水线中，故这种技术被称为**流水线**（pipelining）。流水线技术对可靠数据传输协议可带来如下影响：

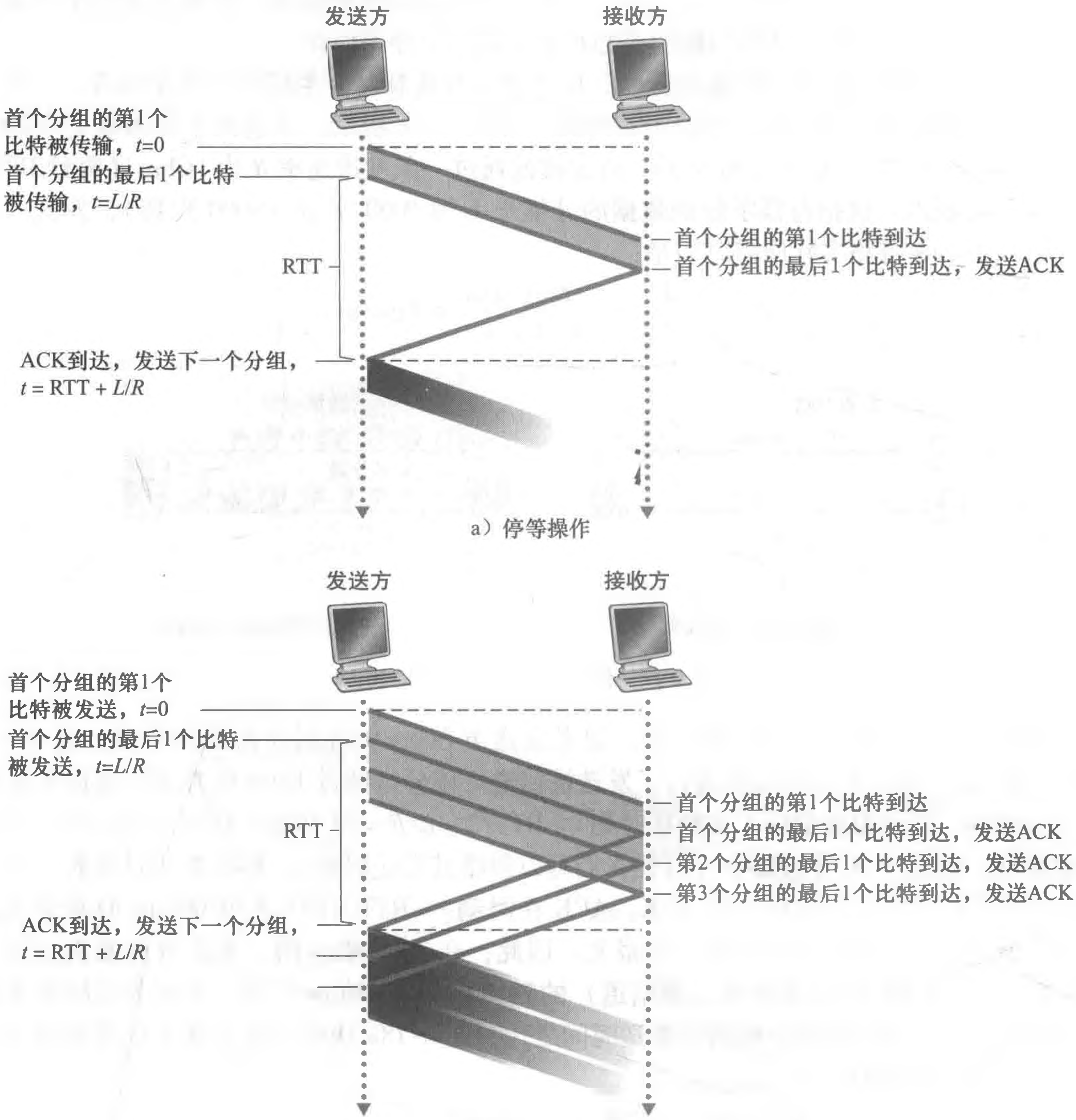


图 3-18　停等和流水线发送

- 必须增加序号范围，因为每个输送中的分组（不计算重传的）必须有一个唯一的序号，而且也许有多个在输送中的未确认报文。
- 协议的发送方和接收方两端也许不得不缓存多个分组。发送方最低限度应当能缓冲那些已发送但没有确认的分组。如下面讨论的那样，接收方或许也需要缓存那些已正确接收的分组。
- 所需序号范围和对缓冲的要求取决于数据传输协议如何处理丢失、损坏及延时过大的分组。解决流水线的差错恢复有两种基本方法是：**回退 *N* 步**（Go-Back-N，GBN）和**选择重传**（Selective Repeat，SR）。

3.4.3 回退 N 步

在**回退 N 步**（GBN）协议中，允许发送方发送多个分组（当有多个分组可用时）而不需等待确认，但它也受限于在流水线中未确认的分组数不能超过某个最大允许数 N。在本节中我们较为详细地描述 GBN。但在继续阅读之前，建议你操作本书配套 Web 网站上的 GBN Java 小程序（这是一个非常好的 Java 程序）。

图 3-19 显示了发送方看到的 GBN 协议的序号范围。如果我们将基序号（base）定义为最早未确认分组的序号，将下一个序号（nextseqnum）定义为最小的未使用序号（即下一个待发分组的序号），则可将序号范围分割成 4 段。在［0，base－1］段内的序号对应于已经发送并被确认的分组。［base，nextseqnum－1］段内对应已经发送但未被确认的分组。［nextseqnum，base＋N－1］段内的序号能用于那些要被立即发送的分组，如果有数据来自上层的话。最后，大于或等于 base＋N 的序号是不能使用的，直到当前流水线中未被确认的分组（特别是序号为 base 的分组）已得到确认为止。

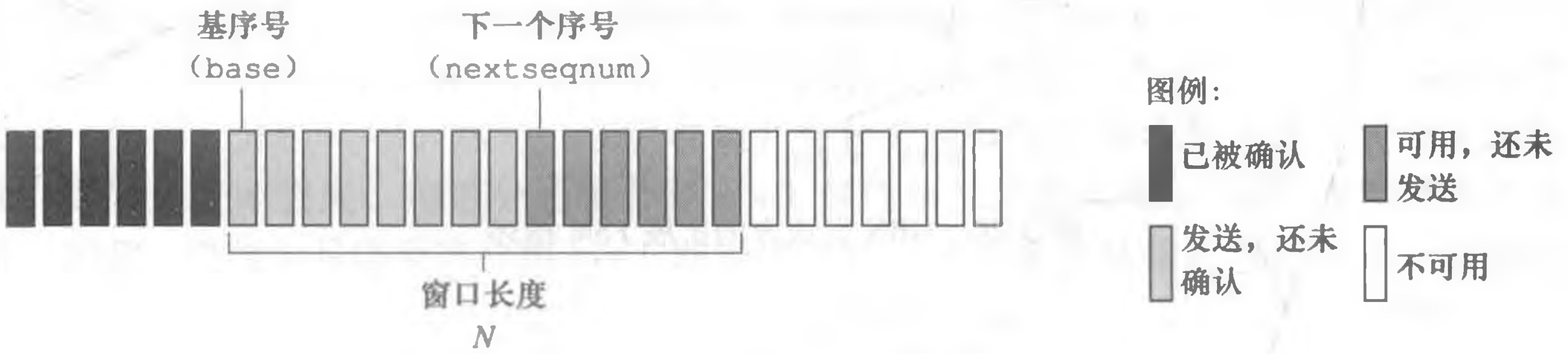


图 3-19 在 GBN 中发送方看到的序号

如图 3-19 所提示的那样，那些已被发送但还未被确认的分组的许可序号范围可以被看成是一个在序号范围内长度为 N 的窗口。随着协议的运行，该窗口在序号空间向前滑动。因此，N 常被称为**窗口长度**（window size），GBN 协议也常被称为**滑动窗口协议**（sliding-window protocol）。你也许想知道，我们为什么先要限制这些被发送的、未被确认的分组的数目为 N 呢？为什么不允许这些分组为无限制的数目呢？我们将在 3.5 节看到，流量控制是对发送方施加限制的原因之一。我们将在 3.7 节学习 TCP 拥塞控制时分析另一个原因。

在实践中，一个分组的序号承载在分组首部的一个固定长度的字段中。如果分组序号字段的比特数是 k，则该序号范围是［0，2^k-1］。在一个有限的序号范围内，所有涉及序号的运算必须使用模 2^k 运算。（即序号空间可被看作是一个长度为 2^k 的环，其中序号 2^k-1 紧接着序号 0。）前面讲过，rdt3.0 有一个 1 比特的序号，序号范围是［0，1］。在本章末的几道习题中探讨了一个有限的序号范围所产生的结果。我们将在 3.5 节看到，TCP 有一个 32 比特的序号字段，其中的 TCP 序号是按字节流中的字节进行计数的，而不是按分组计数。

图 3-20 和图 3-21 给出了一个基于 ACK、无 NAK 的 GBN 协议的发送方和接收方这两端的扩展 FSM 描述。我们称该 FSM 描述为扩展 FSM，是因为我们已经增加了变量（类似于编程语言中的变量）base 和 nextseqnum，还增加了对这些变量的操作以及与这些变量有关的条件动作。注意到该扩展的 FSM 规约现在变得有点像编程语言规约。［Bochman 1984］对 FSM 扩展技术提供了一个很好的综述，也提供了用于定义协议的其他基于编程语言的技术。

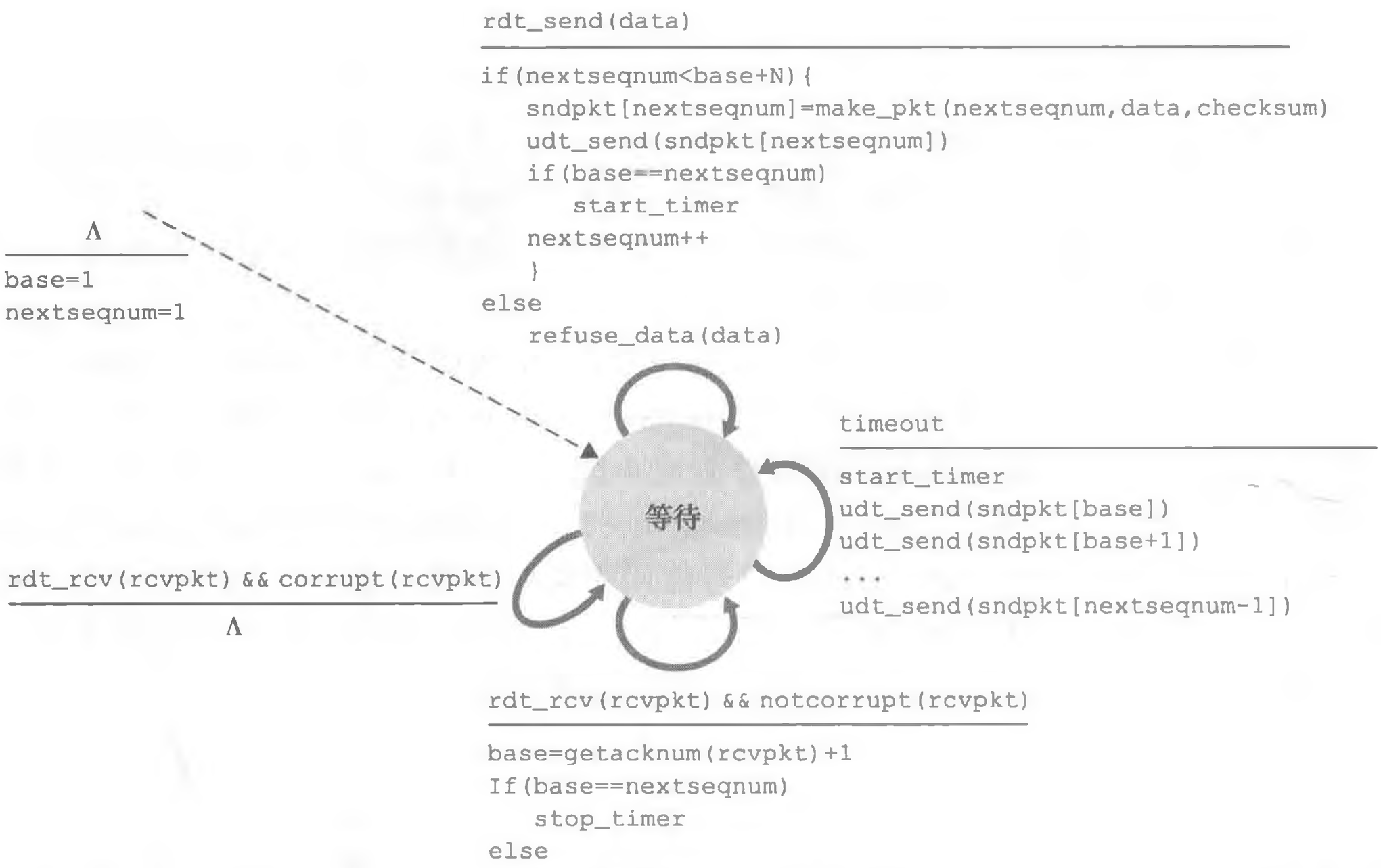


图 3-20 GBN 发送方的扩展 FSM 描述

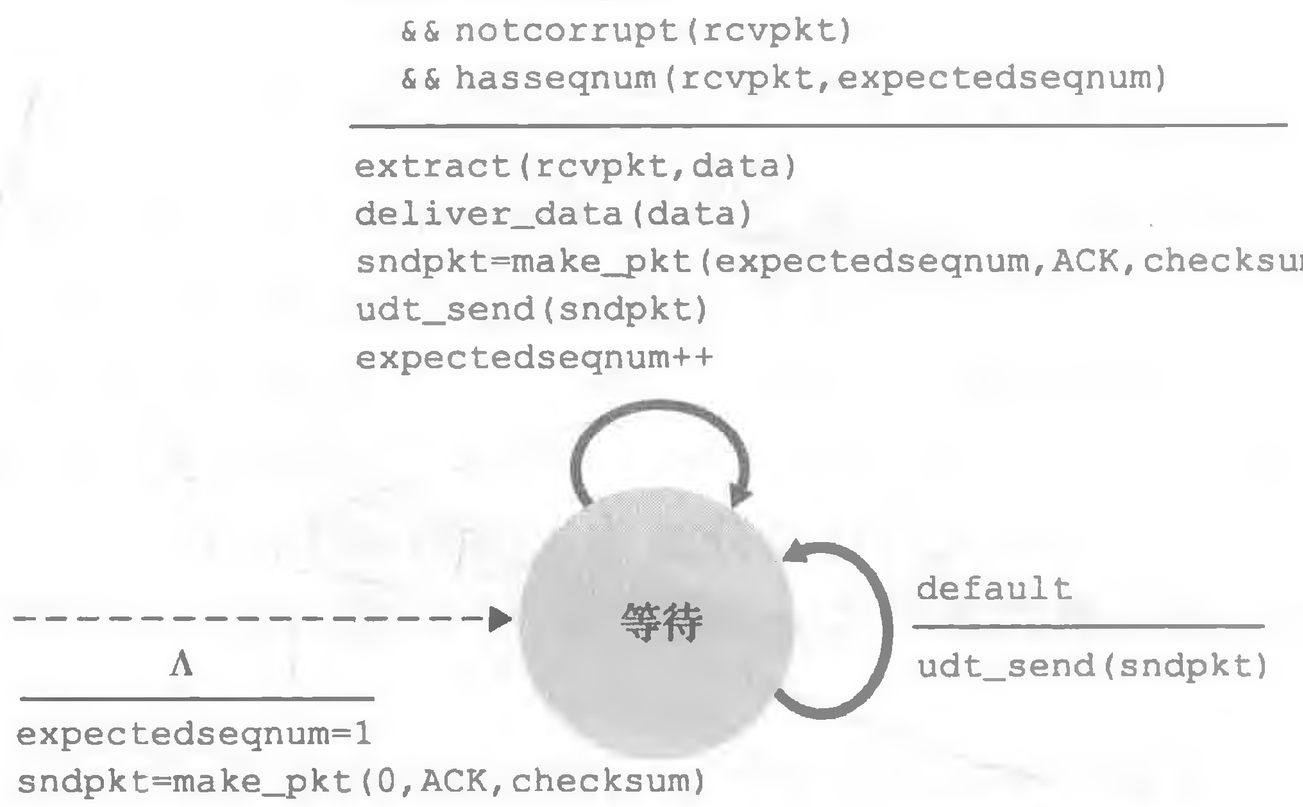


图 3-21 GBN 接收方的扩展 FSM 描述

GBN 发送方必须响应三种类型的事件：

- 上层的调用。当上层调用 rdt_send()时，发送方首先检查发送窗口是否已满，即是否有 N 个已发送但未被确认的分组。如果窗口未满，则产生一个分组并将其发送，并相应地更新变量。如果窗口已满，发送方只需将数据返回给上层，隐式地指示上层该窗口已满。然后上层可能会过一会儿再试。在实际实现中，发送方更可能缓存（并不立刻发送）这些数据，或者使用同步机制（如一个信号量或标志）允许上层在仅当窗口不满时才调用 rdt_send()。
- 收到一个 ACK。在 GBN 协议中，对序号为 n 的分组的确认采取**累积确认**（cumulative acknowledgment）的方式，表明接收方已正确接收到序号为 n 的以前且包括 n 在内的所有分组。稍后讨论 GBN 接收方一端时，我们将再次研究这个主题。

- 超时事件。协议的名字“回退 N 步”来源于出现丢失和时延过长分组时发送方的行为。就像在停等协议中那样，定时器将再次用于恢复数据或确认分组的丢失。如果出现超时，发送方重传所有已发送但还未被确认过的分组。图 3-20 中的发送方仅使用一个定时器，它可被当作是最早的已发送但未被确认的分组所使用的定时器。如果收到一个 ACK，但仍有已发送但未被确认的分组，则定时器被重新启动。如果没有已发送但未被确认的分组，停止该定时器。

在 GBN 中，接收方的动作也很简单。如果一个序号为 n 的分组被正确接收到，并且按序（即上次交付给上层的数据是序号为 $n-1$ 的分组），则接收方为分组 n 发送一个 ACK，并将该分组中的数据部分交付到上层。在所有其他情况下，接收方丢弃该分组，并为最近按序接收的分组重新发送 ACK。注意到因为一次交付给上层一个分组，如果分组 k 已接收并交付，则所有序号比 k 小的分组也已经交付。因此，使用累积确认是 GBN 一个自然的选择。

在 GBN 协议中，接收方丢弃所有失序分组。尽管丢弃一个正确接收（但失序）的分组有点愚蠢和浪费，但这样做是有理由的。前面讲过，接收方必须按序将数据交付给上层。假定现在期望接收分组 n，而分组 $n+1$ 却到了。因为数据必须按序交付，接收方可能缓存（保存）分组 $n+1$，然后，在它收到并交付分组 n 后，再将该分组交付到上层。然而，如果分组 n 丢失，则该分组及分组 $n+1$ 最终将在发送方根据 GBN 重传规则而被重传。因此，接收方只需丢弃分组 $n+1$ 即可。这种方法的优点是接收缓存简单，即接收方不需要缓存任何失序分组。因此，虽然发送方必须维护窗口的上下边界及 nextseqnum 在该窗口中的位置，但是接收方需要维护的唯一信息就是下一个按序接收的分组的序号。该值保存在 expectedseqnum 变量中，如图 3-21 中接收方 FSM 所示。当然，丢弃一个正确接收的分组的缺点是随后对该分组的重传也许会丢失或出错，因此甚至需要更多的重传。

图 3-22 给出了窗口长度为 4 个分组的 GBN 协议的运行情况。因为该窗口长度的限制，发送方发送分组 0～3，然后在继续发送之前，必须等待直到一个或多个分组被确认。当接收到每一个连续的 ACK（例如 ACK 0 和 ACK 1）时，该窗口便向前滑动，发送方便可以发送新的分组（分别是分组 4 和分组 5）。在接收方，分组 2 丢失，因此分组 3、4 和 5 被发现是失序分组并被丢弃。

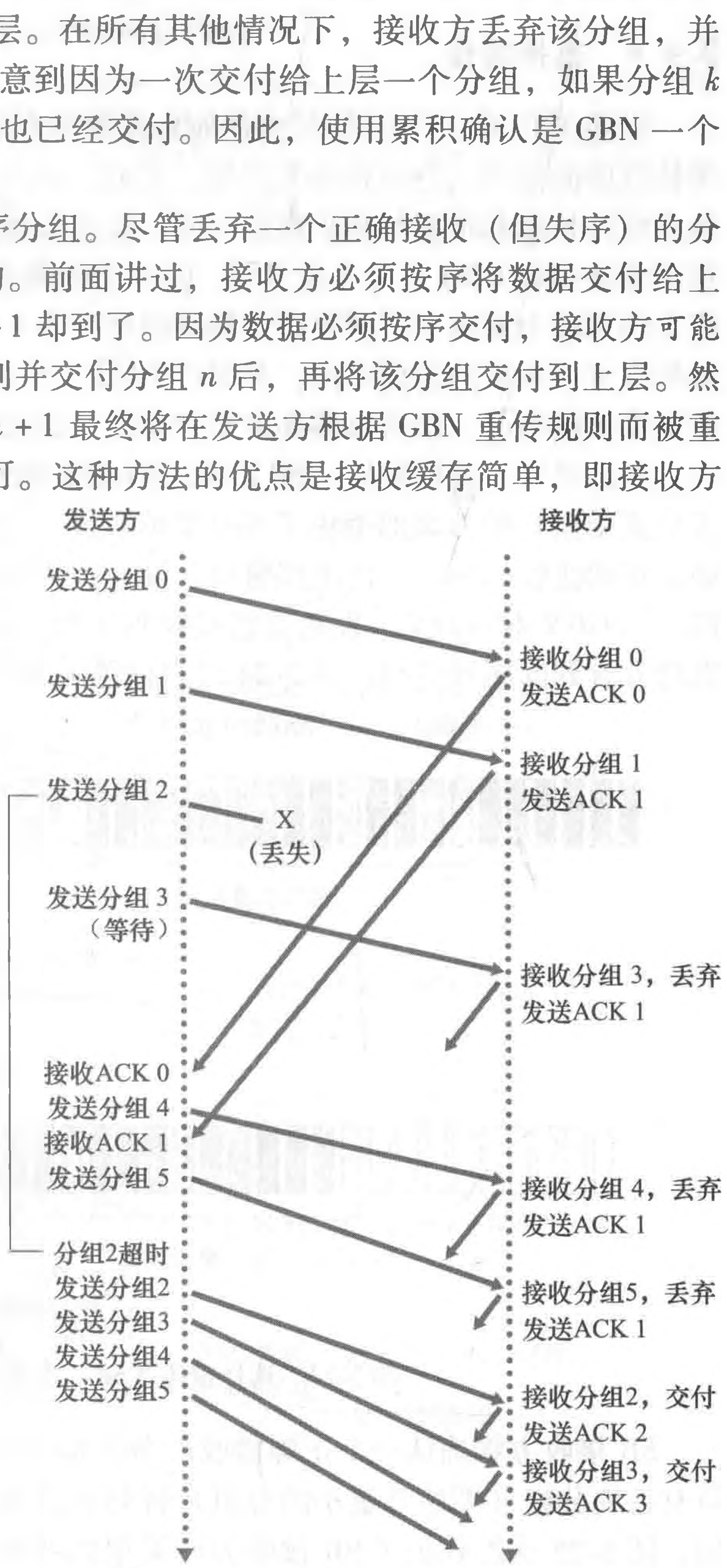


图 3-22 运行中的 GBN

在结束对 GBN 的讨论之前，需要提请注意的是，在协议栈中实现该协议可能与图 3-20 中的扩展 FSM 有相似的