计算机网络

Computer Networking

极夜酱

目录

Chapter 1 网络

1.1 因特网

1.1.1 因特网（Internet）

因特网是一个世界范围的计算机网络，它互联了遍及全世界数十亿的计算设备，所有这些设备都称为主机（host）或端系统（end system）。端系统通过通信链路（communication link）和分组交换机（packet switch）连接到一起，不同的链路能够以不同的速率传输数据，链路的传输速率（transmission rate）使用比特/秒（bps, bit/s）来度量。端系统通过因特网服务提供商（ISP, Internet Service Provider）接入因特网。

当一台端系统向另一台端系统发送数据时，发送端将数据分组，发送到目的端系统，在那里进行组装。一个分组所经历的一系列通信链路和分组交换机称为路径（route / path）。分组交换类似于现实中的货物运输，在出发地将货物分开并装上多辆卡车，每辆卡车独立通过公路运输，最后在目的地卸货并重新组装。

1.1.2 分布式应用程序（Distributed Application）

分布式应用程序涉及多个相互交换数据的端系统，例如即时通信、实时道路信息、视频会议、多人游戏等。分布式应用程序的核心问题在于一个端系统上的应用程序如何能够向运行在另一个端系统上的应用程序发送数据。

套接字接口（socket interface）规定了运行在一个端系统上的程序向运行在另一个端系统上的特定程序交付数据的方式。例如 Alice 要给 Bob 寄一封信，当然 Alice 不能只是写完这封信就把它丢出窗外。Alice 需要把信放入信封，在信封上根据指定格式写上收信人的全名、地址和邮政编码，信封上贴上邮票，再将信封投入信箱中。Alice 想要寄信就必须要遵守邮政服务制定的这一套规则。因此，发送数据的程序也必须遵守 socket 接口，才能向接收数据的程序发送数据。

1.1.3 协议（Protocol）

在两个人或两台设备之间进行通信时需要遵守一些协议，协议就是用于管理通信的一组规则。传输控制协议 TCP（Transmission Control Protocol）和网际协议 IP（Internet Protocol）是因特网中两个最为重要的协议，因特网的主要协议统称为 TCP/IP。

因特网标准（Internet standard）是经过充分测试的规约，只要是与因特网打交道，就会用到它们，并要服从于它们。因特网标准由 IETF（Internet Engineering Task Force）研发，IETF 的标准文档称为 RFC (Request For Comment)，目的是解决因特网先驱者们面临的网络和协议问题。它们定义了 TCP、IP、HTTP、SMTP 等协议，目前已经有将近 7000 个 RFC。

人类无时无刻都在执行协议，人类用约定好的交互方式互相交流。但是如果两人的交谈都不在同一频道上，那就不能好好沟通了。

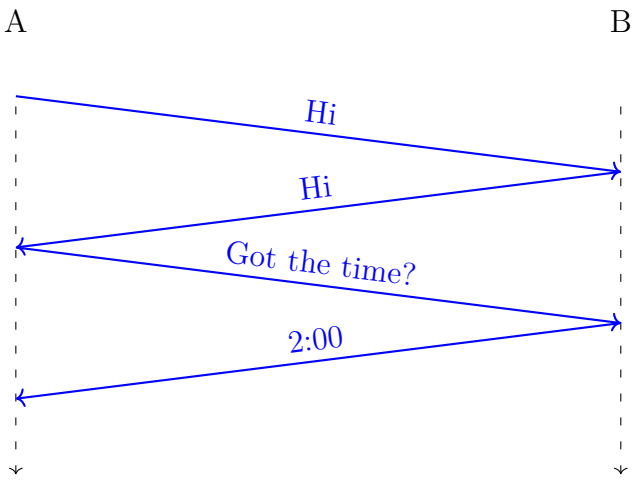

图 1.1: 人类协议

在因特网中，涉及两个或多个远程通信实体的所有活动都受协议的制约。例如在浏览器中输入 URL（Uniform Resource Locator）向一个 Web 服务器发出请求，首先你的计算机将向该 Web 服务器发送一条连接请求报文，并等待回答。Web 服务器接收到连接请求报文，并返回一条连接响应报文。在得知请求正常后，计

算机会发送一条要获取的网页名字的报文，最后 Web 服务器向计算机返回该网页。

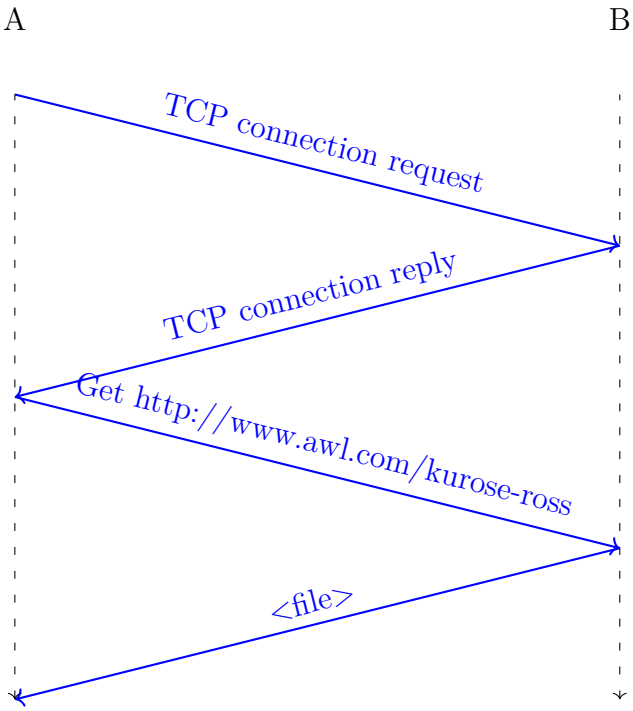

图 1.2: 网络协议

1.2 分组交换

1.2.1 存储转发传输（Store-and-Forward Transmission）

在网络应用中，端系统彼此交换报文（message），报文能够包含任何数据。报文从源端系统发送到目的端系统的过程中，长报文会被划分为较小的数据块，称为分组（packet），每个分组都通过通信链路和分组交换机传送。如果源端系统发送一个 L bits 分组，链路的传输速率为 R bits/sec，则传输该分组的时间为 $\frac{L}{R}$ 秒。

多数分组交换机在链路的输入端使用存储转发传输机制，存储转发传输是指在交换机能够开始向输出链路传输该分组的第一个 bit 之前，必须接收到整个分组。

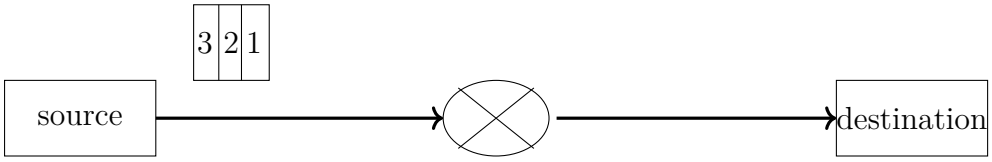

图 1.3: 存储转发

假设忽略传播时延（propagation delay），源端系统在时间 0 开始传输，路由器在时间 $\frac{L}{R}$ 刚好接收到整个分组，之后再向输出链路开始传输，在时间 $2\frac{L}{R}$ 整个分组被目的端系统接收。

因此，由 N 条速率为 R 的链路组成的路径（在源和目的地之间有 $N-1$ 台路由器）发送一个分组，端到端的时延为

$$d = N\frac{L}{R} \tag{1.1}$$

1.2.2 时延

当从一个节点到后继节点，一个分组在沿途的每个节点都经受了几种不同类型的时延，包括节点处理时延（nodal processing delay）、排队时延（queuing delay）、传输时延（transmission delay）、传播时延（propagation delay）。

处理时延包括了检查分组首部和决定分组去向所需要的时间，以及检查差错的时间。

分组交换机的每条链路都有一个输出缓存/输出队列（output buffer / output queue）。如果到达的分组需要传输到某条链路，但发现该链路正忙于传输其它分组，该分组必须在输出缓存中等待。因此，除了存储转发时延以外，分组还要承受输出缓存的排队时延。由于缓存空间的大小是有限的，到达的分组可能发现该缓存已被填满，这种情况下将出现丢包（packet loss），到达的分组或已经排队的分组将被丢弃。

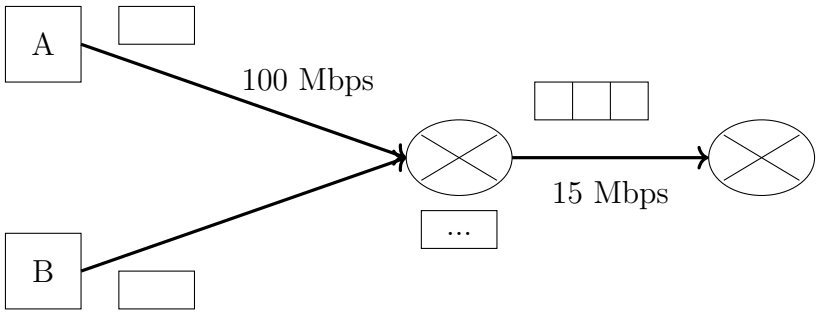

图 1.4: 排队时延

分组通常是以 FCFS（First-Come-First-Served）的方式传输，只有当所有已经到达的分组被传输之后，才能传输新到达的分组。传输时延 $\frac{L}{R}$ 就是将所有分组推向输出链路所需的时间。

传播时延是指从链路的起点到下一个路由器传播所需的时间。

假设 d_{proc}、d_{queue}、d_{trans} 和 d_{prop} 分别表示处理时延、排队时延、传输时延和传播时延，那么节点总时延为

$$d_{nodal} = d_{proc} + d_{queue} + d_{trans} + d_{prop} \tag{1.2}$$

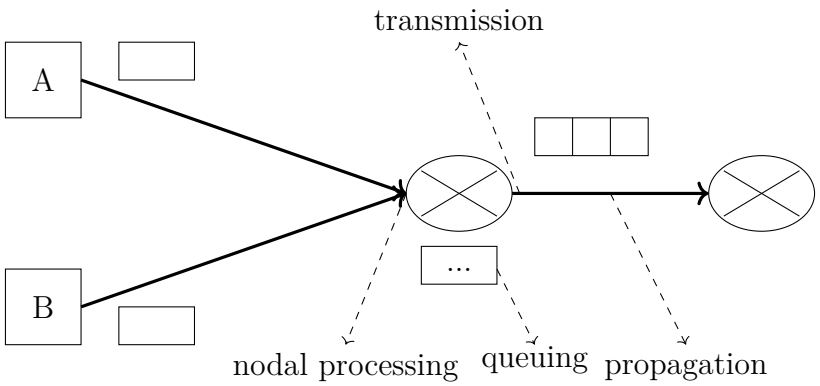

图 1.5: 时延

1.2.3 转发表（Forwarding Table）

路由器从输入链路获得分组，然后向输出链路发送分组，但是路由器怎样决定应当向哪一条链路发送呢?

因特网中每一个端系统都有一个 IP 地址，源在发送分组时在分组首部包含了目的地的 IP 地址。当分组到达每一个路由器时，路由器根据转发表，为其查询适当的输出链路。

端到端的选路过程类似于现实中的问路，每到一个地点都询问别人路线，逐步找到最终的目的地。在这个类比中，被问路的人就类似于路由器。

1.2.4 traceroute

traceroute 是诊断网络问题时常用的工具，它可以定位从源主机到目标主机之间经过了哪些路由器，以及到达各个路由器的耗时。当源主机向目标主机发送消息，发现消息无法送达。此时，可能是某个中间节点发生了问题，比如某个路由器因负载过高产生了丢包。通过 traceroute 可以定位网络包在是在哪个节点丢失的。

traceroute 将从源发送 N 个特殊的分组，当第 i 个路由器收到第 i 个分组时，该路由器会向源回送一个报文，记录了路由器的名字和地址。traceroute 会重复该

实验 3 次。

traceroute (Linux) / tracert (Windows)

```
1 tracert www.github.com
```

运行结果

```
1     2 ms     9 ms     2 ms  HS8145V [192.168.1.1]
2     5 ms    15 ms     5 ms  100.65.0.1
3     5 ms     5 ms     5 ms  124.74.22.41
4    25 ms     6 ms    17 ms  101.95.88.138
5     *        *        *     请求超时。
6     *        *        *     请求超时。
7     *        *        *     请求超时。
8    30 ms    32 ms    35 ms  106.38.244.146
9     *        *        *     请求超时。
10     *        *        *     请求超时。
11     *        *        *     请求超时。
12    30 ms    31 ms    31 ms  220.181.38.251
```

1.3 协议层

1.3.1 协议栈（protocol stack）

因特网是个极其复杂的系统，因此因特网的体系存在分层的组织结构。类似一次乘飞机的过程，首先需要购票、托运行李、登机，飞行到目的地后，需要离机、认领行李，如果对航班不满意，还可以在向票务机构投诉。

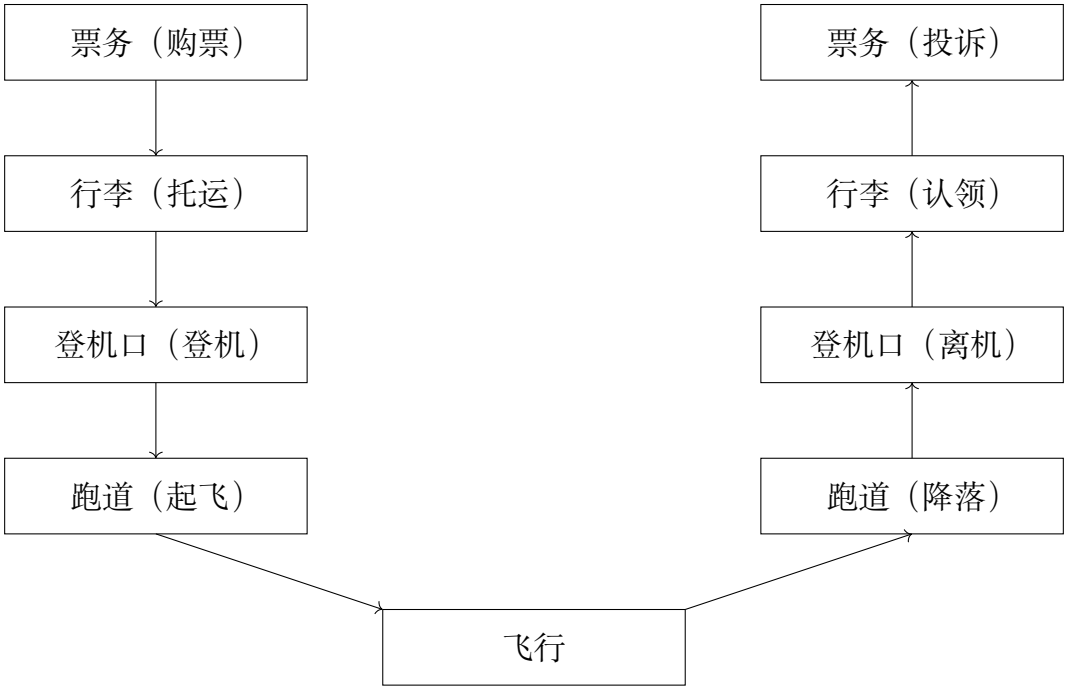

图 1.6: 航行流程

协议分层是为了使各层之间相互独立，每一层只专注于做一类事情。各层之间相互独立，各层之间不需要关心其它层是如何实现的，只需要知道自己如何调用下层提供好的功能就可以。同时协议分层提高了整体灵活性，每一层都可以使用最适合的技术来实现，只需要保证提供的功能以及暴露的接口的规则没有改变就行。

各层的所有协议被称为协议栈，TCP/IP 协议栈由 5 个层次组成，分为是物理层、链路层、网络层、传输层和应用层。

ISO（International Organization for Standard）为了更好地使网络应用更为普及，

推出了 OSI（Open System Interconnection）参考模型。但因为 OSI 七层模型出现地比 TCP/IP 五层模型晚，在 OSI 开始使用之前，TCP/IP 已经被广泛使用，最终 OSI 没有在实践中被广泛应用。

协议层	**功能**
应用层	提供网络服务操作接口
表示层	对要传输的数据进行处理
会话层	管理不同通讯节点之间的连接信息
传输层	建立不同节点之间的网络连接
网络层	将网络地址映射为 MAC 地址实现数据包转发
数据链路层	将要发送的数据包转为数据帧
物理层	利用物理设备实现数据的传输

表 1.1: OSI 七层模型

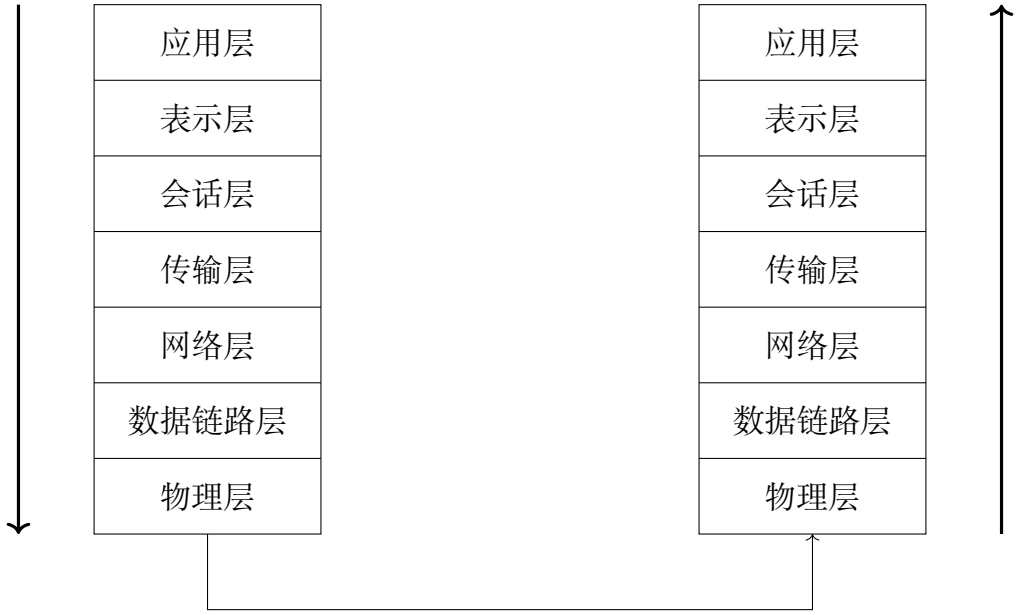

图 1.7: 数据传输

1.3.2 封装（Encapsulation）

在发送主机端，应用层报文（application-layer message）被传送给传输层，传输层收取到报文并附加首部信息，形成传输层报文段（transport-layer segment），被添加的首部将被接收端的传输层使用。首部信息包括了交付应用程序信息、差错检测位信息等。

传输层继续向网络层传递该报文段，网络层再添加首部信息，如源和目的端系统地址等，形成了网络层数据报（network-layer datagram）。

该数据报接下来被传递给链路层，链路层添加其首部信息，形成链路层帧（link-layer frame）。

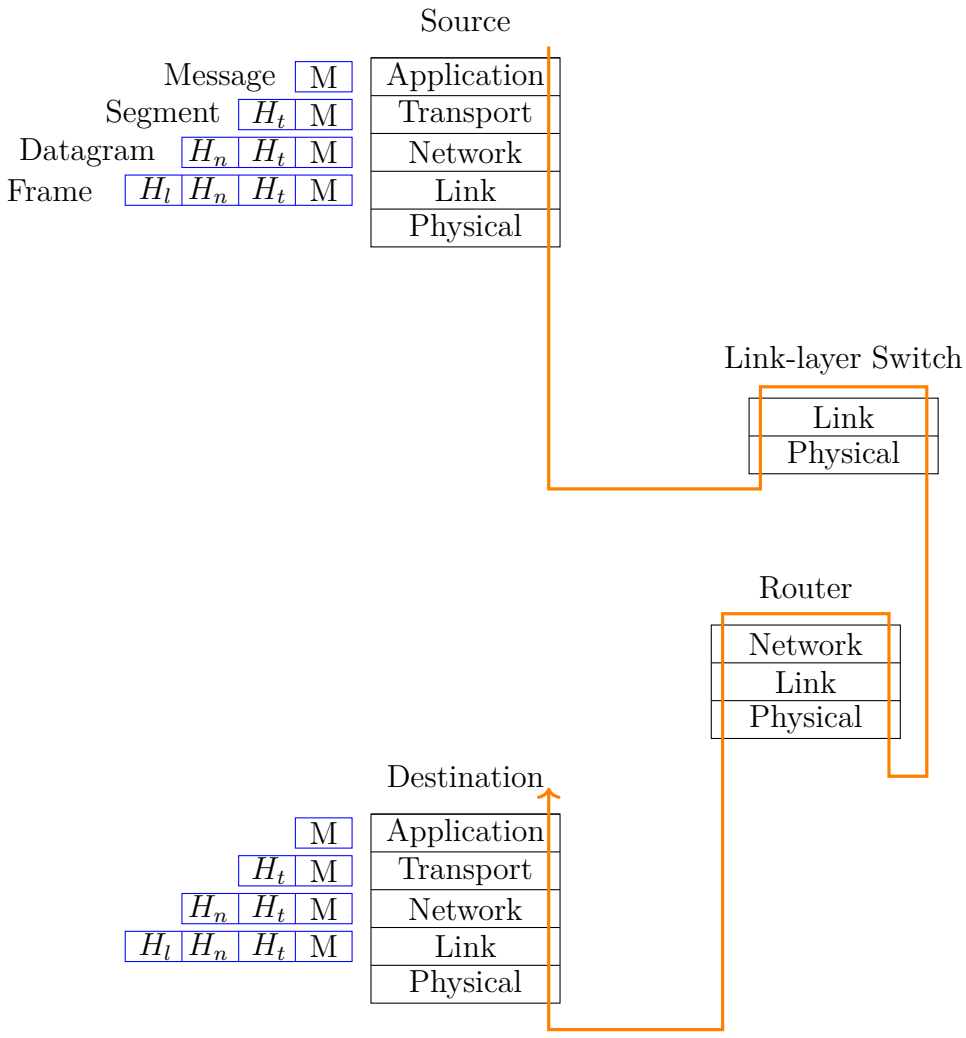

图 1.8: 封装

1.4 网络安全

1.4.1 网络攻击

2000 年 2 月 7 日，化名 MafiaBoy 的 15 岁加拿大少年攻击了 Yahoo、Amazon 和 eBay，使这些网站瘫痪达数小时，造成了超过 12 亿美元的损失，在股市中造成混乱。MafiaBoy 后来被透露为一名名叫 Michael Calce 的高中生，通过入侵几所大学的网络并利用其服务器进行 DDoS 攻击。这次袭击直接导致了今天许多网络犯罪法的制定。现在 MafiaBoy 已经金盆洗手，从事网络安全行业工作。

2017 年 1 月，University of Albert 内 20 个教室和实验室的 300 台电脑被安装恶意软件，导致近 3000 名学生的用户信息被盗。

2017 年 4 月，一名 Laurentian University 的 CS 学生为了证明学校的系统容易受到攻击，从而入侵了学校系统，导致近 2000 名学生的个人记录（包括密码、电话号码、成绩等）被泄漏。

因特网最初的设计理念是让一群相互信任的用户连接到一个透明的网络上，因此安全性并没有太过必要。然而如今的网络不再是完全透明和相互信任的，因此网络安全领域需要研究如何攻击计算机网络、如何防御免受攻击和如何设计能够更好地避免攻击的体系。

1.4.2 恶意软件

多数的恶意软件（malware）通过自我复制（self-replicating）传播，一旦设备感染了恶意软件，就有可能导致文件丢失、隐私泄漏等。恶意软件能够以病毒（virus）或蠕虫（worm）的形式传播。病毒是利用用户交互来感染用户设备的，例如包含恶意代码的电子邮件附件，如果用户无意打开附件，就会在其设备上运行恶意软件。另一种蠕虫则无需任何用户交互，例如用户在使用某些比较脆弱的网络应用程序时，该应用可能用因特网接受恶意软件并运行。

1.4.3 DoS 攻击

另一种通过攻击服务器和网络基础设施的威胁称为拒绝服务攻击（DoS, Denial-of-Service）。例如泛洪攻击，攻击者通过向目标主机发送大量的分组，堵塞目标的接入链路，使得合法分组无法到达服务器。但是如果服务器的接入速率非常大的话，单一的攻击源无法产生足够大的流量来伤害服务器。因此攻击者可以通过控制多个源向目标发送大量流量，这种方法称为分布式 DoS（DDoS, Distributed DoS）。

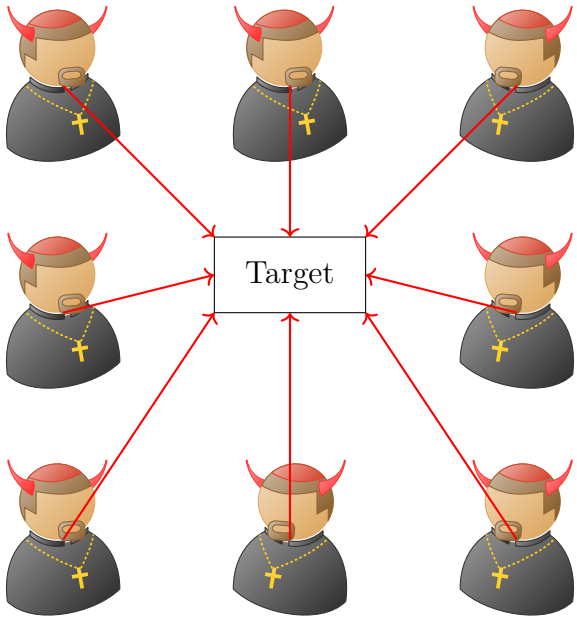

图 1.9: DDoS

1.4.4 IP 欺骗（IP Spoofing）

IP 欺骗是指伪造源 IP 地址，以便冒充其它系统或发件人的身份，从而冒充另外一台机器与服务器打交道。IP 欺骗会造成目标系统受到攻击却无法确认攻击源，或者取得目标系统的信任以便获取机密信息。IP 欺骗的防范，一方面需要目标设备采取更强有力的认证措施，不仅仅根据源 IP 就信任来访者，还需要更多的认证手段。

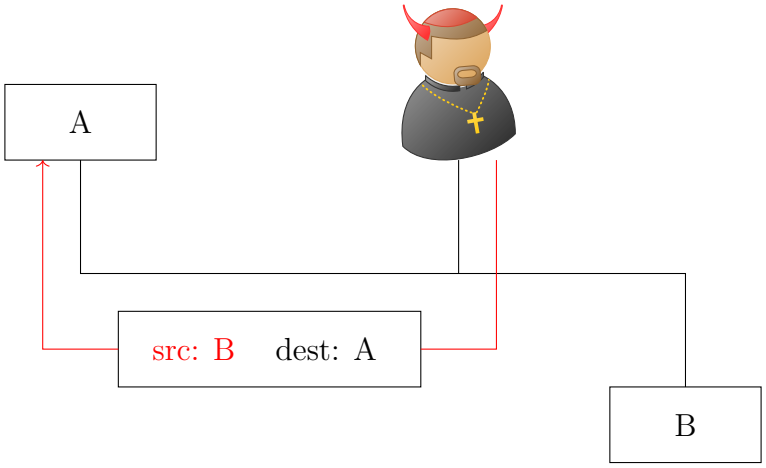

图 1.10: IP 欺骗

Chapter 2 应用层

2.1 应用层协议

2.1.1 网络应用

网络应用程序是指能够运行在不同端系统并通过网络彼此通信的程序。例如在 Web 应用程序中，有两个互相通信的程序，一个是运行在用户主机上的浏览器程序，另一个是运行在 Web 服务器主机上的服务器程序。

应用程序体系结构包括 Client-Server 和 P2P（Peer-to-Peer）两种体系结构。在 Client-Server 中，服务器主机一直运行处理客户端的请求。因为服务器需要有一个固定的的 IP 地址，因此客户端能够通过 IP 地址发送分组与其联系。著名的 Client-Server 应用程序包括 Web、FTP、Telnet 和电子邮件等。在 Client-Server 应用中，经常会出现一台单独的服务器主机不堪重负，跟不上所有客户请求的情况。例如搜索引擎（如 Google、Bing、Baidu）、电商（如 Amazon、eBay、Alibaba）、电子邮件（如 Gmail、Yahoo!）、社交媒体（如 Facebook、Instagram、Twitter、WeChat）都分步了多个数据中心，每数据中心都有数十万台服务器，必须要持续供电和维护。而在 P2P 中，主机之间的通信不必通过专门的服务器，很多流量密集型的应用都是 P2P 体系结构的，包括对等方协助下载加速器（如迅雷）、文件共享（如 BitTorrent）。

2.1.2 进程通信

在同一个端系统中，多个进程可以通过进程间通信的机制相互通信，然而在不同的端系统（可能有不同的操作系统）中，需要通过网络交换报文（message）。进程通过套接字（socket）接口向网络发送或接收报文。

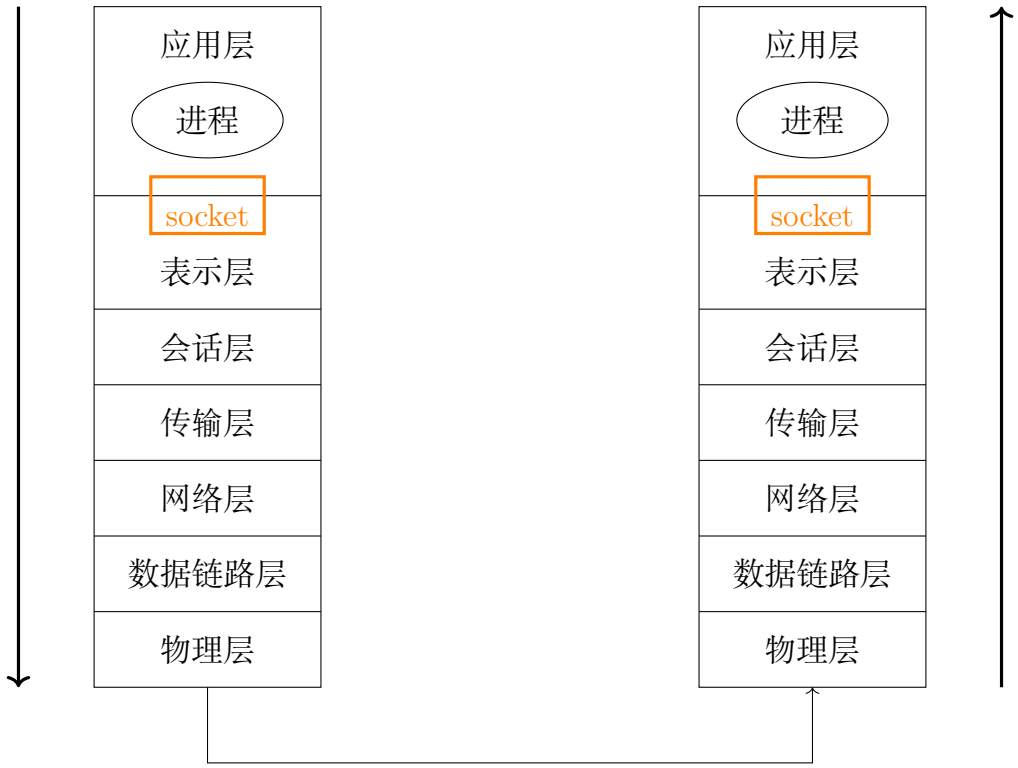

图 2.1: socket

在进程发送分组的过程中，必须要标识 IP 地址和端口号（port number）才能将分组发送给另一主机的进程。其中 IP 地址用于标识主机，端口号用于指定运行在目的主机上的接收进程。由于一台主机上会运行多个应用程序，因此端口号是不可或缺的信息。一些著名的应用已经被分配了特定的端口号，例如 Web 服务器使用端口号 80、邮件服务器（SMTP 协议）进程使用端口号 25。

发送端在使用 socket 时必须选择一种传输层协议，不同的协议会提供不同的服务。

一个传输层协议可以通过四个方面进行分类：

1. 可靠数据传输（reliable data transfer）：分组在网络传输中可能会因溢出或损坏等原因丢失，对于电子邮件、文件传输和金融相关的应用来说，数据丢失会造成灾难性的后果，这种情况下就必须采用可靠数据传输。对于一些可以容忍丢失（loss-tolerant）的应用，例如多媒体音视频，它们能够承受一定量的数据丢失，这只会造成小干扰，而非致命性的问题。

2. 吞吐量（throughput）：传输层协议能够以特定的速率提供服务。

3. 定时（timing）：传输层协议提供定时保证。例如在网络电话或多人游戏中，较长的时间延迟会出现不自然的停顿或失去真实感。

4. 安全性（security）：传输层协议保证数据安全。例如在发送端将数据加密，并在接收端解密数据，以防在传输被中途被窃听。

应用	**数据丢失**	**吞吐量**	**时间敏感**
文件传输	不允许丢失	弹性	不
电子邮件	不允许丢失	弹性	不
网络电话	容忍丢失	few kbps $\sim$ 1 Mbps	100 ms
视频会议	容忍丢失	10 kbps $\sim$ 5 Mbps	100 ms
交互式游戏	容忍丢失	few kbps $\sim$ 10 kbps	100 ms

表 2.1: 常见应用传输服务需求

2.1.3 TCP / UDP

TCP （Transmission Control Protocol）的特点包括：

- 面向连接服务（connection-oriented service）：在应用层数据包开始发送之前，TCP 让客户端和服务器之间互相交换传输层控制信息，让它们为分组的到来做好准备。在此之后，两个进程的 socket 就能建立起 TCP 连接，并可以发送报文了。在发送结束后，该 TCP 连接会被拆除。

- 可靠数据传输：TCP 确保了通信进程交付的数据无差错、不丢失、不重复、不乱序。

- 拥塞控制（congestion control）

UDP（User Datagram Protocol）的特点包括：

- 无连接（connectionless）

- 不可靠数据传输：UDP 不保证报文能够到达接收端，同时报文也有可能是乱序到达的。

- 无拥塞控制

应用	**传输协议**
文件传输	TCP
电子邮件	TCP
网络电话	UDP
视频会议	UDP
交互式游戏	UDP

表 2.2: 常见应用传输协议

2.2 HTTP

2.2.1 HTTP （HyperText Transfer Protocol）

一个 Web 页面是由对象（object）组成的，一个对象就是一个文件，例如 HTML 文件、JPEG 图片、JavaScript 文件、CSS 样式文件等。如果一个 Web 页面包含 1 个 HTML 文件和 5 个 JPEG 图片，那么这个 Web 页面就有 6 个对象。

每一个对象都可以通过 URL（Uniform Resource Locator）寻址，URL 地址由存放对象的服务器主机名（host name）和路径名（path name）组成。例如对于 http://www.someSchool.edu/someDepartment/picture.gif 而言，其中主机名就是 www.someSchool.edu，路径名是/someDepartment/picture.gif。

HTTP 定义在 RFC 1945、RFC 7230 和 RFC 7540 中。HTTP 使用 TCP 作为它的支撑传输协议，客户端首先发起 TCP 连接，连接建立后，客户进程可以向服务器进程发送 HTTP 请求报文，服务器进程可以向客户进程发送 HTTP 响应报文。

HTTP 是一个无状态协议（stateless protocol），服务器不能存储任何关于客户的状态信息。例如某个客户在短时间内两次请求同一个对象，服务器并不会因为第一次已经向客户提供了该对象而不再作出响应，而是再次重新发送对象。

2.2.2 HTTP 请求报文

HTTP 请求报文由第一行的请求行（request line）和后续的首部行（header line）组成，每行由一个回车（carriage return）和换行（line feed）结束，在首部行之后再附加一个只包含回车换行的空行。

HTTP 请求报文

```
1 GET /somedir/page.html HTTP/1.1\r\n
2 Host: www.someschool.edu\r\n
```

```
Connection: close\r\n
User-agent: Mozilla/5.0\r\n
Accept-language: fr\r\n
\r\n
[entity body]
```

请求行包含 3 个字段：

- 方法：包括 GET、POST、HEAD、PUT 和 DELETE。
- URL：带有请求对象的标识。
- HTTP 版本

首部行中 Host 指明了对象所在的主机。Connection: close 表示浏览器告诉服务器不要使用持续连接，而是要求服务器在发送完对象后就关闭此连接。User-agent 指明了用户代理，即向服务器发送请求的浏览器类型，例如 Mozilla/5.0。服务器可以通过此信息向用户代理发送不同版本的对象。Accept-language 表示用户想要获取对象的语言版本，如果服务器不存在的话，就会发送一个默认版本。

使用 GET 方法时，实体部分（entity body）为空。而使用 POST 方法时，实体部分可以用于包含用户在表单中填写的输入值。HEAD 方法与 GET 类似，服务器会响应，但并不返回请求对象。因此 HEAD 方法常用于调试跟踪。PUT 方法允许用户向服务器上传对象。DELETE 方法允许用户删除服务器上的对象。

2.2.3 HTTP 响应报文

HTTP 响应报文由状态行（status line）、首部行（header line）和实体部分组成。

HTTP 响应报文

```
HTTP/1.1 200 OK\r\n
Connection: close\r\n
Date: Tue, 18 Aug 2015 15:44:04 GMT\r\n
Server: Apache/2.2.3 (CentOS)\r\n
```

```
Last-Modified: Tue, 18 Aug 2015 15:11:03 GMT\r\n
Content-Length: 6821\r\n
Content-Type: text/html\r\n
\r\n
(data data data data data ...)
```

状态行包括了协议版本、状态码和对应状态信息。首部行中 Connection: close 用于告诉客户，发送完报文后将会关闭 TCP 连接。Date 用于表示服务器生成并发送该报文的日期时间。Last-Modified 用于表示对象创建或最后修改的日期时间。Content-Length 用于表示被发送对象的字节数。Content-Type 用于表示对象的类型。

状态码	**含义**
200 OK	请求成功
204 No Content	无内容
301 Moved Permanently	永久性重定向，资源被分配了新 URL
400 Bad Request	请求语法错误，服务器无法理解
403 Forbidden	拒绝执行请求
404 Not Found	无法找到资源
500 Internal Server Error	服务器内部错误
503 Service Unavailable	由于超载或系统维护，暂时无法处理请求
505 HTTP Version not supported	不支持请求的 HTTP 协议的版本

表 2.3: 常见 HTTP 状态码

2.3 Cookie

2.3.1 Cookie

HTTP 服务器是无状态的，然而一些 Web 站点通常希望能够识别用户，从而记住用户信息或限制用户访问。目前 Cookie 广泛用于记录用户登录信息，这样下次访问时可以不再输入用户名和密码了。当然这种方便也存在用户信息泄密的问题，尤其在多个用户公用一台电脑时很容易出现这样的情况。

例如用户首次访问 Amazon，HTTP 响应报文中会包含 Set-cookies 识别码，浏览器会将识别码添加到所管理的文件中。当用户继续浏览 Amazon 时，每一个 HTTP 请求浏览器都会从 cookie 文件中查询该网站的识别码，并添加到 HTTP 请求报文中。Amazon 也可以通过 cookie 来维护用户希望购买的商品信息，并推荐个性化产品。

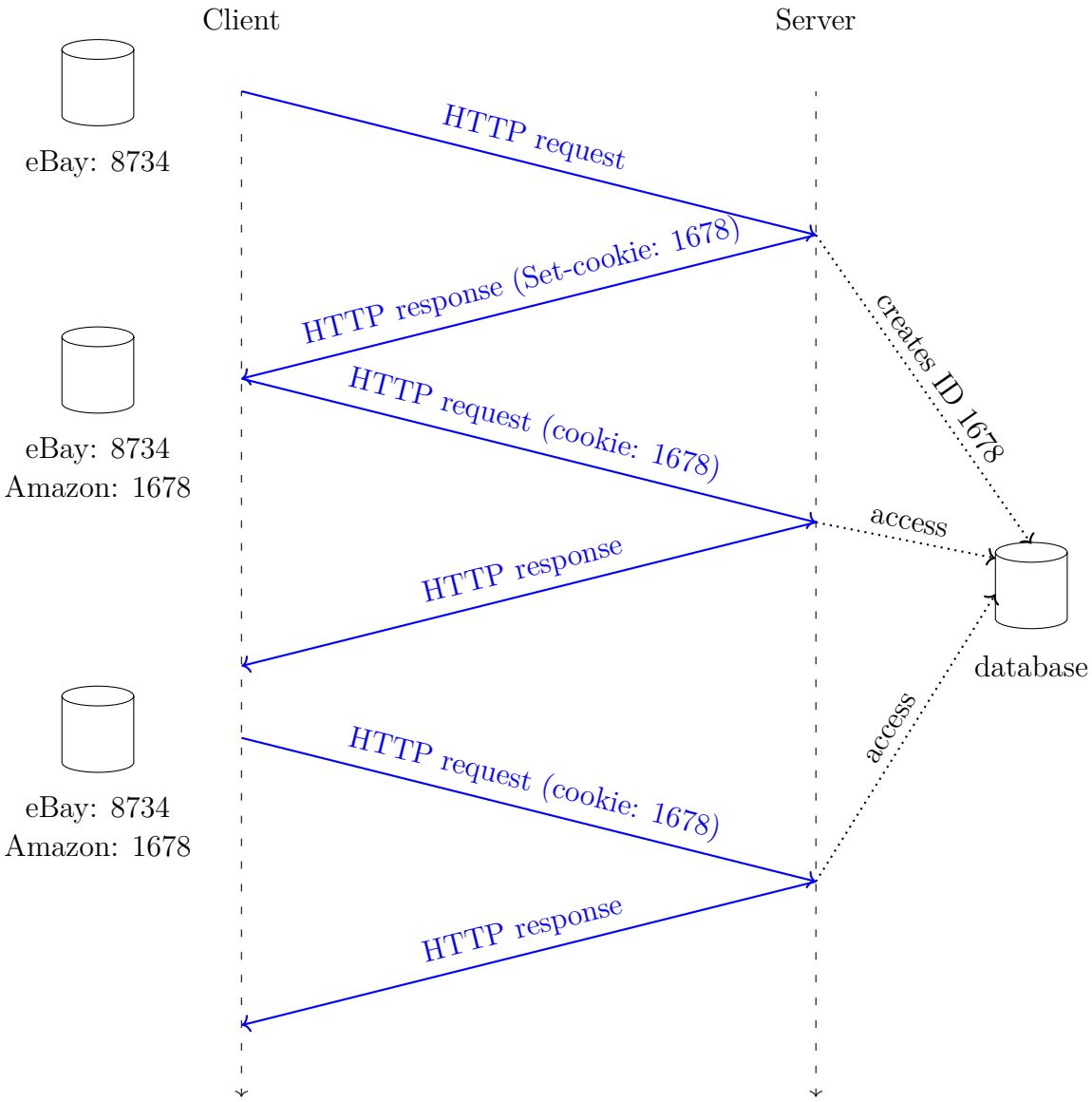
Client
Server
eBay: 8734
HTTP request
HTTP response (Set-cookie: 1678)
creates ID 1678
eBay: 8734
Amazon: 1678
HTTP request (cookie: 1678)
access
HTTP response
database
eBay: 8734
Amazon: 1678
HTTP request (cookie: 1678)
access
HTTP response

图 2.2: Cookie

2.4 Email

2.4.1 SMTP（Simple Mail Transfer Protocol）

电子邮件系统由用户代理（user agent）、邮件服务器（mail server）和 SMTP 组成，其中例如 Outlook、Apple Mail、Gmail 就是用户代理。当 Alice 发送邮件时，她的用户代理就会向邮件服务器发送邮件，此时邮件被放在邮件服务器的待发报文队列中。当 Bob 要查看邮件时，他的用户代理就会其邮件服务器中获取该报文。

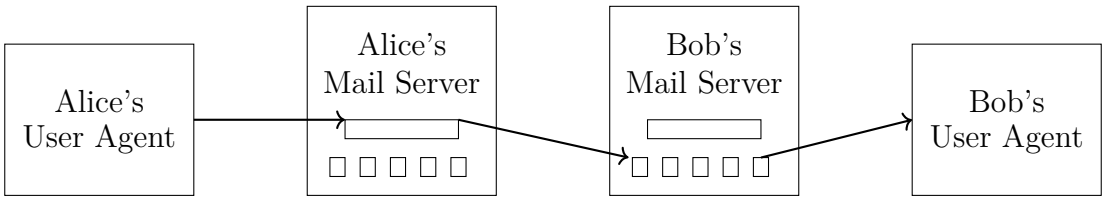

图 2.3: SMTP

SMTP 是电子邮件中使用的应用层协议，它使用 TCP 可靠数据传输服务，使用 25 号端口。

SMTP 使用命令和应答对报文进行传输：

命令	**描述**
HELO	与服务器确认信息
AUTH	登录验证
MAIL FROM	发件人信息
RCPT TO	收信人地址
DATA	输入邮件基本信息
FROM	发信人显示信息
TO	服务器收件人显示信息
SUBJECT	邮件标题
QUIT	断开链接

表 2.4: SMTP 命令

返回码	含义
220	服务就绪
250	请求动作成功完成
235	认证通过
221	处理中
354	发送开始
500	指令错误
550	命令无法执行

表 2.5: SMTP 应答返回码

当用户调用用户代理查看邮件报文时就需要用到邮件访问协议。流行的邮件访问协议有邮局协议版本 3（POP3）、因特网邮件访问协议（IMAP, Internet Mail Access Protocol）和 HTTP。

2.5 DNS

2.5.1 DNS（Domain Name System）

人类可以用很多种方式来表示，比如姓名、身份证号、社保号等，但是人类会更乐意使用容易记忆的姓名而不是社保号。想象一下人们之间这样说话：“你好，我叫 132-67-9875”。

主机可以使用主机名(hostname)进行表示,例如 www.google.com、www.facebook.com，这些名字便于记忆。同时主机也可以使用 IP 地址进行标识，一个 IP 地址由 4 个取值在 0 ∼ 255 的字节组成，通常以点分十进制表示，如 121.7.106.83。

路由器更喜欢定长的、有层次结构的 IP 地址来标记主机，因此域名系统 DNS 的作用就是进行主机名到 IP 地址的转换。

类似电话本的原理，每一个名字都对应了一个单独的电话号码。DNS 最简单的设计就是只使用一个 DNS 服务器，其中包含所有的映射（mapping），客户端可直接将查询发往该 DNS 服务器。

这种集中式的设计存在几个问题：

- 单点故障（single point of failure）：如果该 DNS 服务器故障，整个因特网随之瘫痪。
- 通信容量（traffic volume）：单个 DNS 需要处理所有 DNS 查询。
- 远距离集中式数据库（distant centralized database）：单个 DNS 不可能与所有的客户端都相邻，一些地区需要跨越半个地球，造成严重的时延。
- 维护（maintainance）：中央数据库庞大，需要频繁为新添加的主机更新记录。

2.5.2 分布式层次数据库

为了处理扩展性的问题，DNS 使用了大量的 DNS 服务器分布在全世界，并以层次方式组织。

DNS 服务器分层 3 类：

1. 根 DNS 服务器
2. 顶级域（TLD, Top-Level Domain）DNS 服务器
3. 权威（authoritative）DNS 服务器

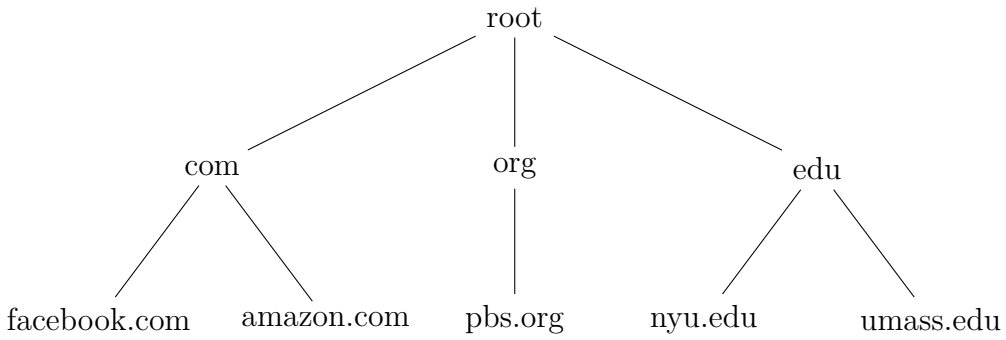

图 2.4: DNS 服务器

例如一个 DNS 客户需要获取 www.amazon.com 的 IP 地址。客户首先与根服务器之一联系，它将返回顶级域名 com 的 TLD 服务器的 IP 地址。接着客户与 TLD 服务器之一联系，它将为 amazon.com 返回权威服务器的 IP 地址。最后客户与权威服务器之一联系，它将为 www.amazon.com 返回 IP 地址。

每个 ISP 都有一个本地 DNS 服务器（local DNS server），它并不严格属于 DNS 的层级体系。当主机进行 DNS 查询时，查询会被发送到本地 DNS 服务器，再由该本地 DNS 服务器作为代理（proxy），将查询转发给层级 DNS 服务器。

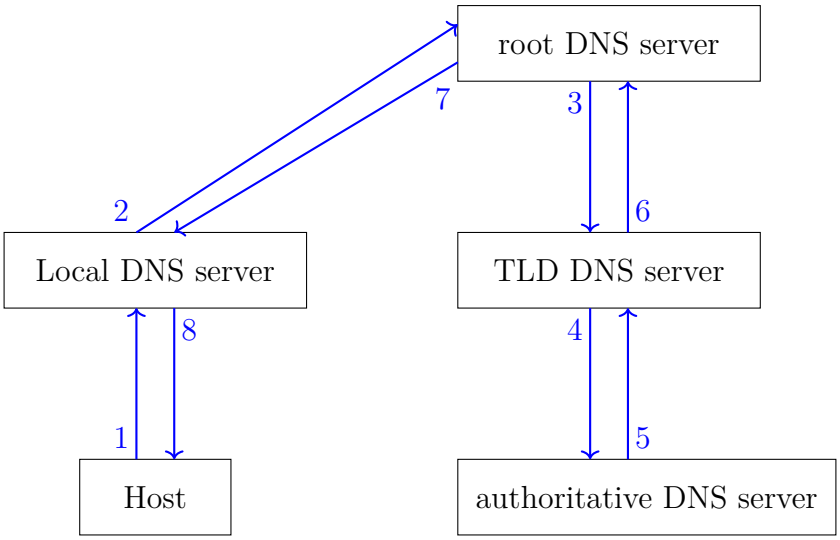

图 2.5: DNS 递归查询

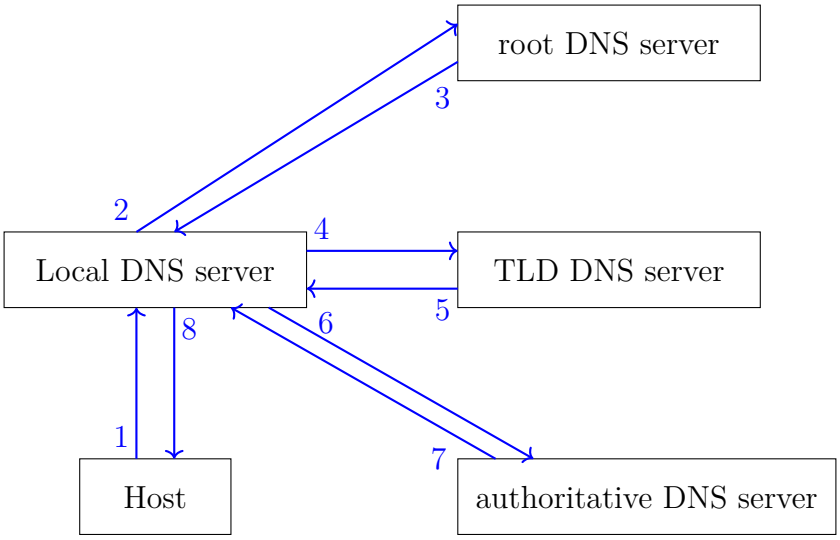

图 2.6: DNS 迭代查询

为了减少时延并减少传输 DNS 报文的数量，DNS 使用了 DNS 缓存（DNS caching）。只要 DNS 服务器获得了 IP 映射，就将其缓存，因此根 DNS 服务器不会再被经常访问。然而 IP 映射并不是永久的，DNS 服务器在一段时间后将会丢弃缓存信息。

2.6 P2P

2.6.1 P2P（Peer-to-Peer）

P2P 体系结构中，其中每个 peer 节点都能够帮助服务器来分发文件。也就是说，当一个 peer 节点接收到文件数据时，它可以利用自己的上载能力重新将数据分发给其它 peer 节点。

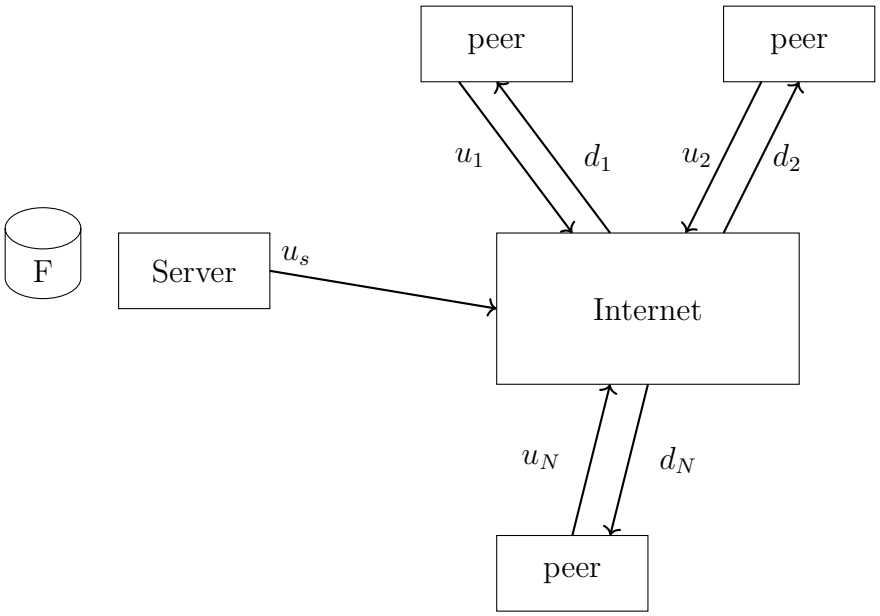

图 2.7: P2P

2.6.2 BitTorrent

BitTorrent 是一种用于文件分发的 P2P 协议，其中参与一个特定文件分发的所有 peer 的集合被称为洪流（torrent）。在一个 torrent 中的 peer 彼此下载等长度的文件块（chunk），块长度通常为 256KB。

当一个 peer 节点一开始加入一个 torrent 时，它没有文件块。随着时间的推移，它将累积越来越多的文件块。当它下载文件块时，也为其它 peer 节点上载了多个文件块。peer 节点一但获得了整个文件，它可以自私地离开 torrent，也可以大公无私地留在 torrent 中并继续向其它 peer 节点上载文件块。

例如，当一个 peer 节点 Alice 加入 torrent 时，追踪服务器随机选择一些 peer 节点，并将这些 peer 节点的 IP 地址发送给 Alice。Alice 持有这些 peer 节点的列表，试着与该列表上的多个 peer 节点创建并行的 TCP 连接。这里称所有与 Alice 成功地创建 TCP 连接的 peer 节点为邻近 peer 节点（neighboring peers）。

随着时间的推移，其中的一些 peer 节点可能离开，而其它 peer 节点可能试着与 Alice 创建 TCP 连接。因此，邻近 peer 节点会随着时间而改变。

在任何时刻，每个 peer 节点都拥有来自某文件块的子集，且不同的 peer 节点具有不同的文件块子集。Alice 周期性地询问每个邻近 peer 节点它们所具有的块列表，并对当前还没有的块发出请求。

Alice 使用一种称为最稀缺优先（rarest first）的策略，其思路是根据她没有的块从她的邻居中找到拷贝数量最少的那些块，并优先请求这些稀缺块。因此，最稀缺的块会更迅速地分发，其目标是均衡每个块在 torrent 中的拷贝数量。

2.7 视频流

2.7.1 视频流（Video Streaming）

视频是由一系列的图像，以一种恒定的速率（如每秒 24 张或 30 张图像）来展现的。一幅未压缩、数字编码的图像由像素阵列（array of pixels）组成，其中每个像素都是由一些比特编码来表示亮度和颜色。

视频的一个重要特征就是能够被压缩，因而可用比特率（bit rate）来权衡视频质量。视频有各种各样的压缩算法，对应不同的比特率。视频信息之所以存在大量可以被压缩的空间，是因为其中本身就存在大量的数据冗余。

如果分析一个视频里的每一帧，我们会看到有许多区域是相互关联的。因此在传输比特时，可以传输颜色值及其重复次数，而不是传输多个相同的颜色值。

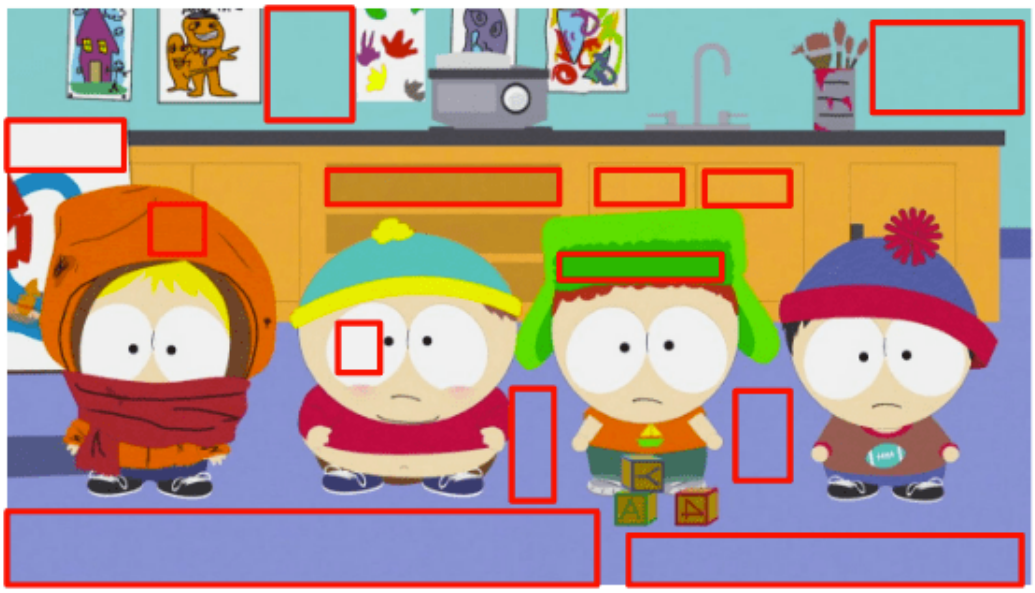

图 2.8: 空间冗余

另一种是时间冗余，对于连续的两帧，尝试花费较少的数据量进行编码，只传输两帧之间发生改变的地方，而不用全部传输。

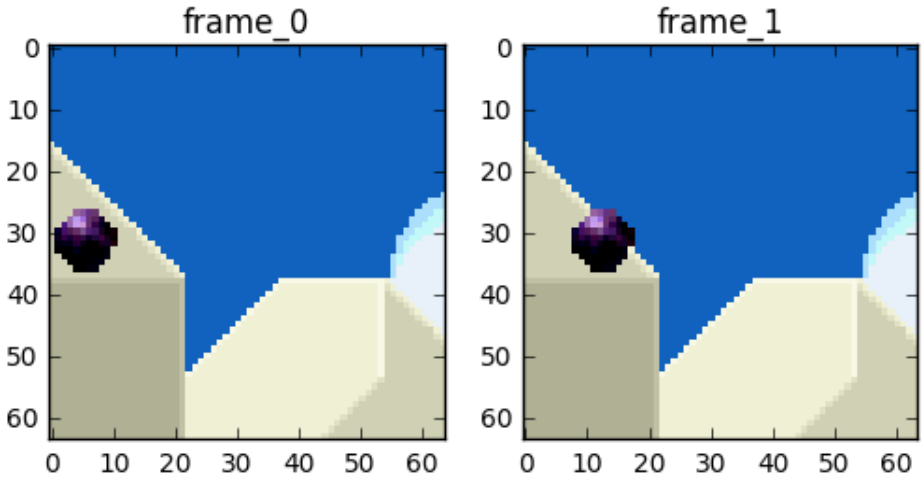

图 2.9: 时间冗余

2.7.2 DASH（Dynamic Adaptive Streaming over HTTP）

在 HTTP 流中，视频只是存储在 HTTP 服务器中的一个普通文件，每个文件都有一个特定的 URL。但是因为如此，所有客户接收到的视频编码版本也相同，显然这并没有考虑用户的带宽问题。

DASH，即经 HTTP 的动态适应流，解决了这个问题。在 DASH 流中，将视频编码成几个不同的版本，每个版本都具有不同的比特率，每个版本都有属于自己的 URL。DASH 在初始时能够根据用户的带宽选择一个合适的版本，同时也允许用户自定义更换版本。

2.7.3 CDN（Content Distribution Network）

服务器一定都是有物理位置的，因此离服务器越远的地方访问服务器就会越慢。况且很多中小型网站的服务器本身使用的带宽就小，一旦同时访问的人一多，访问网站依然会很慢。另一个问题就是服务器都会有一定几率遇到宕机的问题，服务器出了问题就导致无法打开网站。

内容分发网 CDN 的出现就可以解决这个问题。它与电商的仓库管理方式非常类似，电商刚开始的时候，买家购买一样物品后，商家都只能从仓库所在地发货，

所以物流时间需要 3~10 天不等。后来电商和物流公司想了个招，在全国各个区域都建设仓储中心，先把货物放到这些仓储中心，这样买家不管在哪个地方，直接从最近的仓库发货，这样就能减少物流时间。

CDN 管理分布在多个地理位置上的服务器，在它的服务器上存储视频（包括其它 Web 内容）的副本，并且将每个用户请求定向到一个能提供最好用户体验的 CDN 位置。

2.8 Socket 编程

2.8.1 Socket

Socket（套接字）是对 TCP/IP 网络协议进行的包装（协议的抽象应用），它本身最大的特点是提供了不同进行之间的数据通讯操作。所有的网络协议的组成是非常繁琐的，如果所有的开发者去研究具体的通讯协议会对开发带来很大的难度，所以在不同的编程语言内部就会考虑对一些网络的协议进行包装。

Socket 主要是针对两种协议的包装：

- 传输控制协议 TCP（Transmission Control Protocol）：采用有状态的通讯机制进行传输，在通讯时会通过三次握手机制保证与一个指定节点的数据传输的可靠性，在通讯完毕后会通过四次挥手的机制关闭连接。由于在每次数据通讯前都需要消耗大量的时间进行连接控制，所以执行性能较低，且系统资源占用较大。
- 用户数据报协议 UDP（User Datagram Protocol）：采用无状态的通讯机制进行传输，没有了 TCP 中复杂的握手与挥手处理机制，这样就节约了大量的系统资源，同时数据传输性能较高。但是由于不保存单个节点的连接状态，所以发送的数据不一定可以被全部接收。UDP 不需要连接就可以直接发送数据，并且多个接收端都可以接收同样的消息，所以使用 UDP 适合于广播操作。

2.8.2 字节序（Endianness）

字节序是指多字节数据在计算机内存中存储或者网络传输时各字节的存储顺序。计算机有两种储存数据的方式，分别是大端字节序（big endian）和小端字节序（little endian）。

例如存储两个十六进制数 0x12345678 和 0x11223344，小端字节序把值的低位存在低地址，而大端字节序把值的高位存在低地址。

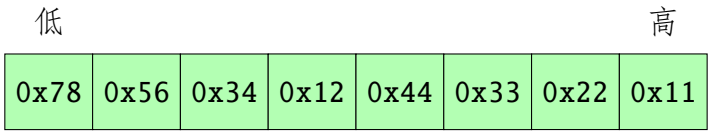

图 2.10: 小端字节序

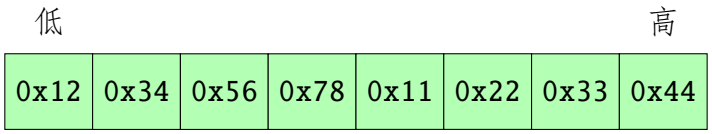

图 2.11: 大端字节序

由此引发了计算机界的大端与小端之争，不同的 CPU 厂商并没有达成一致。在网络通信中，RFC1700 规定使用大端字节序作为网络字节序。因此不使用大端的计算机在发送数据时必须将自己的主机字节序转换为网络字节序，接收数据时需要转换成自己的主机字节序，这样就与 CPU 和操作系统无关了。

htons() 和 htonl() 可用于将主机字节序转换为网络字节序，ntohs() 和 ntohl() 可用于将网络字节序转换为主机字节序。

2.8.3 TCP

Socket 是在应用层和传输层之间的一个抽象层，它把 TCP/IP 层复杂的操作抽象为几个简单的接口供应用层调用已实现进程在网络中通信。

Socket 起源于 UNIX 一切皆文件的哲学思想，服务器和客户端各自维护一个 Socket，在建立连接后，可以向自己 Socket 写入内容供对方读取或者读取对方内容，通讯结束时关闭 Socket。

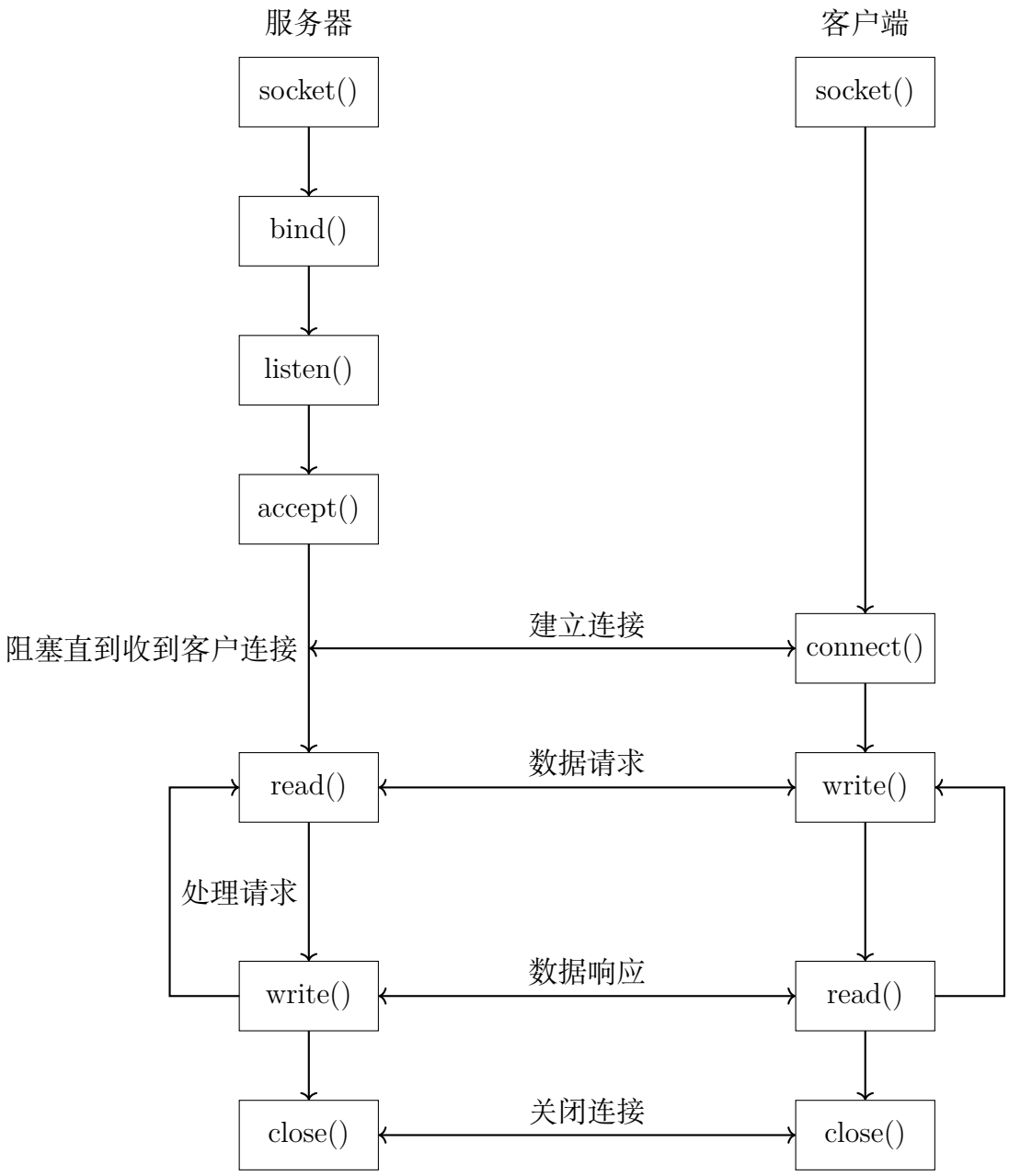

图 2.12: TCP

socket()

- IP 地址类型：
  - AF_INET（IPv4）
  - AF_INET6（IPv6）
- 数据传输方式/套接字类型：

- SOCK_STREAM（流格式套接字/面向连接的套接字）
- SOCK_DGRAM（数据报套接字/无连接的套接字）

bind()

将 Socket 和本地地址绑定，地址包括 IP 和端口号。

127.0.0.1 用于表示本地地址。

端口号的范围为 0~65535，其中 0~1023 为知名端口（well known ports），用于绑定一些常用服务。

listen()

监听，将接收到的客户端连接放入队列。

accept()

从队列获取请求，返回客户端 Socket 描述符。当已完成连接队列的下一个完成连接是空, 那么 accept() 将被阻塞。

read() / write()

双向通信。read() 负责从 Socket 中读取内容，write() 负责将内容写入 Socket 中。

close()

关闭 Socket，终止 TCP/UDP 连接。

TCP

tcp_server.py

```python
import socket

SERVER_HOST = "127.0.0.1"
SERVER_PORT = 8080

def main():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.bind((SERVER_HOST, SERVER_PORT))
    sock.listen()
    print("【服务端】服务器启动完毕。")
    print("【服务端】等待客户端连接...")

    # 当有客户端连接后，获取客户端的socket和地址
    client, client_addr = sock.accept()
    print("【服务端】客户端%s:%s连接到服务器。" % client_addr)

    # 持续接收和响应信息
    while True:
        # 接收客户端发送的数据
        data = client.recv(100).decode("UTF8")
        print("【服务端】接收数据：%s" % data)
        if data == "exit":
            client.send("exit".encode("UTF8"))
            break
        else:
            client.send(data.encode("UTF8"))

    sock.close()

if __name__ == "__main__":
    main()
```

tcp_client.py

```python
import socket

SERVER_HOST = "127.0.0.1"
```

```
4  SERVER_PORT = 8080
5
6  def main():
7      sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
8      sock.connect((SERVER_HOST, SERVER_PORT))
9
10     # 客户端持续与服务端交互
11     while True:
12         msg = input("【客户端】输入数据：")
13         sock.send(msg.encode("UTF8"))
14         reply = sock.recv(100).decode("UTF8")
15         if reply == "exit":
16             break
17         else:
18             print("【服务端】" + reply)
19
20     sock.close()
21
22 if __name__ == "__main__":
23     main()
```

2.8.4 UDP

UDP 是尽最大能力进行传输，但是并不能保证可靠性，丢包并不会重发。TCP 的可靠性是因为一系列的机制保证可靠性。两种协议并没有优略之分，要区分不同的场景来区分。

例如进行文件传输，不能有数据丢失，TCP 协议就更合适，而进行实时视频通话，UDP 会根据恒定的速率进行发送，这样的情况容许部分数据的丢失去追求更好的实时性，所以 UDP 更合适。

UDP

udp_server.py

```
1  import socket
```

```
2
3  SERVER_HOST = "127.0.0.1"
4  SERVER_PORT = 8080
5
6  def main():
7      sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
8      sock.bind((SERVER_HOST, SERVER_PORT))
9      print("【服务端】服务器启动完毕。")
10     print("【服务端】等待客户端连接...")
11
12     # 持续接收和响应信息
13     while True:
14         # 接收客户端发送的数据
15         data, client_addr = sock.recvfrom(100)
16         print("【服务端】客户端%s:%s： " % client_addr, end="： ")
17         data = data.decode("UTF8")
18         print(data)
19         if data == "exit":
20             sock.sendto("exit".encode("UTF8"))
21             break
22         else:
23             sock.sendto(data.encode("UTF8"), client_addr)
24
25     sock.close()
26
27 if __name__ == "__main__":
28     main()
```

udp_client.py

```
1  import socket
2
3  SERVER_HOST = "127.0.0.1"
4  SERVER_PORT = 8080
5
6  def main():
7      sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
8
```

```
9       # 客户端持续与服务端交互
10      while True:
11          msg = input("【客户端】输入数据：")
12          sock.sendto(msg.encode("UTF8"), (SERVER_HOST, SERVER_PORT))
13          reply, addr = sock.recvfrom(100)
14          reply = reply.decode("UTF8")
15          if reply == "exit":
16              break
17          else:
18              print("【服务端】" + reply)
19
20      sock.close()
21
22  if __name__ == "__main__":
23      main()
```

Chapter 3 传输层

3.1 多路复用与多路分解

3.1.1 传输层

传输层协议为运行在不同主机上的应用进程之间提供了逻辑通信。在发送端，传输层将应用进程的报文添加传输层首部形成传输层分组，称为报文段（segment），这个过程被称为多路复用（multiplexing）。在接收端，网络层从数据报中提取传输层报文段，并交付给传输层，传输层处理报文段，将数据交付给应用进程，这个过程被称为多路分解（demultiplexing）。

应用层可以使用 UDP 和 TCP 这两种截然不同的传输层协议。其中 UDP 提供了一种不可靠、无连接的服务，因此，UDP 不能保证一个进程发送的数据能够完整无缺地到达目的进程。而 TCP 提供了一种可靠的、面向连接的服务，通过使用流量控制、序号、确认和定时器，TCP 确保正确地、按序地将数据交付给接收进程。

3.1.2 无连接的多路复用/多路分解

一个 UDP 的 socket 是由一个二元组进行标识的，该二元组包含了目的 IP 地址和目的端口号。如果两个 UDP 报文段来自不同的源 IP 地址或源端口号，但是具有相同的目的 IP 和目的端口号，那么这两个报文段将通过相同的 socket 被发送到相同的目的进程。

使用 UDP 时，当 A 给 B 发送的报文段中，源端口号是用作返回地址的一部分。当 B 回发一个报文段给 A 时，就需要从 A 到 B 的报文段中取值。

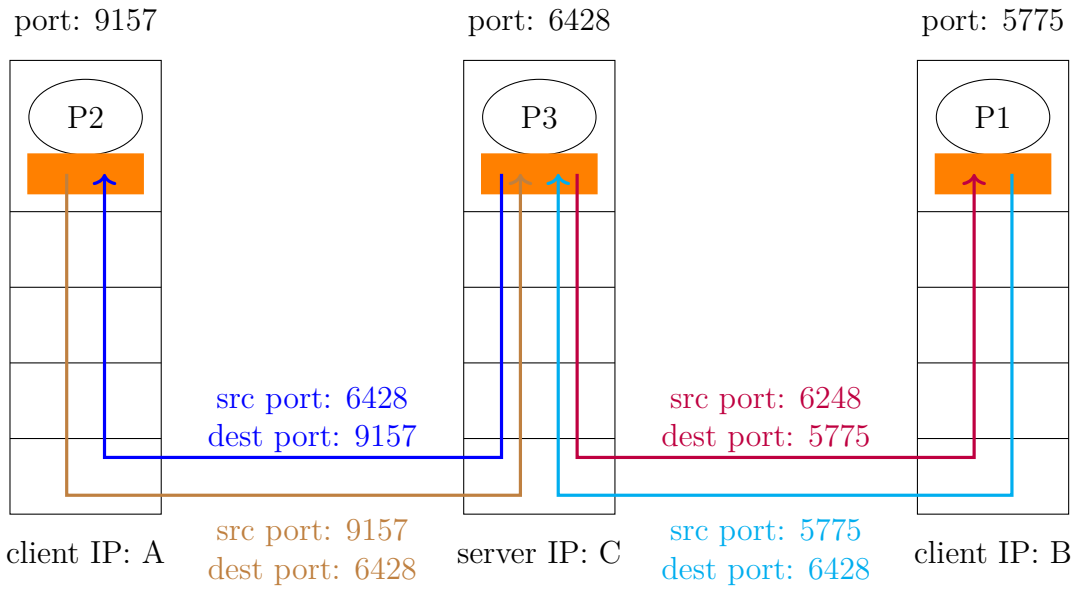

图 3.1: 无连接的多路复用与多路分解

3.1.3 面向连接的多路复用与多路分解

TCP 的 socket 是由一个四元组来标识的，其中包括了源 IP 地址、源端口号、目的 IP 地址和目的端口号。与 UDP 不同的是，两个具有不同源 IP 地址或源端口号的 TCP 报文段将被发送到两个不同的 socket。

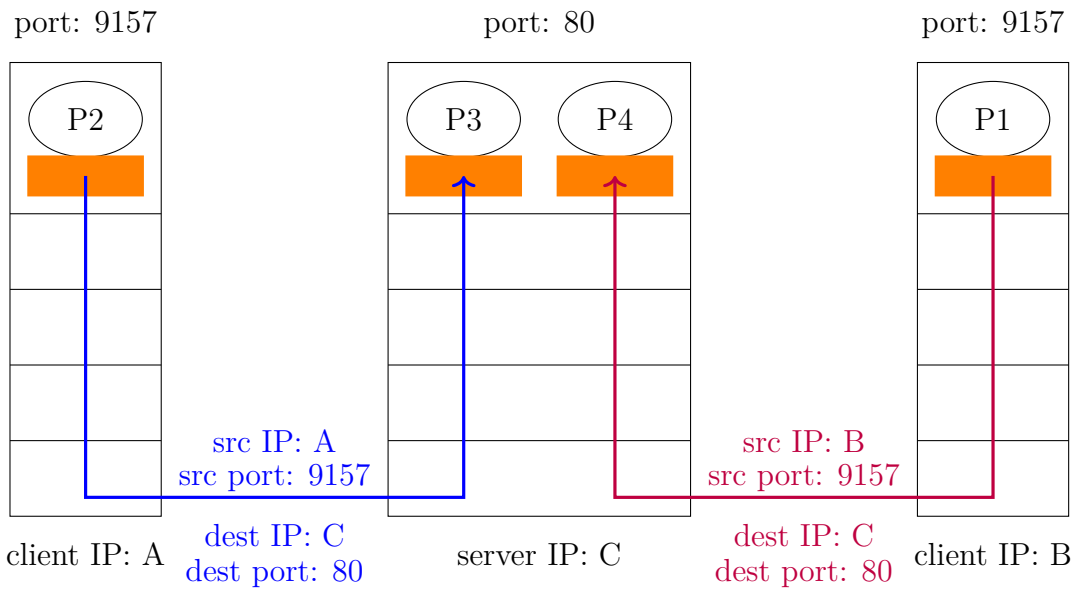

图 3.2: 面向连接的多路复用与多路分解

3.2 无连接传输 UDP

3.2.1 无连接传输 UDP

使用 UDP 时，在发送报文段之前，发送方和接收方的传输层实体之间没有握手。正因为如此，UDP 被称为是无连接的。

UDP 尽最大努力将数据包交付到目的主机，但不保证可靠性和顺序，也不保证带宽及延迟要求。UDP 相较于 TCP 的优势包括无需连接建立、无连接状态、分组首部开销小。

source port #	dest port #
length	checksum
application data (message)	

表 3.1: UDP 报文段结构

UDP 首部只有 4 个字段，每个字段由 2 个字节组成，通过端口号可以将应用数据交给运行在目的端系统中的相应进程。长度字段指示了 UDP 报文段中的字节数（包括首部）。检验和可以用来检查该报文段中是否出现了差错。

3.2.2 校验和（Checksum）

当 UDP 报文段从源到达目的地的过程中，其中的 bit 有可能会受到噪声干扰或在路由器中存储而发生改变。UDP 检验和提供了差错检测的功能。发送方对 UDP 报文段中所有内容都当作 16 位整数进行求和，在求和时遇到的溢出都需要被回卷（wraparound），再对和进行求反，得到的结果被放在 UDP 报文段中的检验和字段。

例如两个 16 位的整数相加：

$$
\begin{array}{rr}
 & 1110011001100110 \\
+ & 1101010101010101 \\
\hline
= & 11011101110111011
\end{array}
$$

将溢出位进行回卷：

$$
\begin{array}{rr}
 & 1011101110111011 \\
+ & 1 \\
\hline
= & 1011101110111100
\end{array}
$$

计算反码得到校验和 0100010001000011。

接收方收到报文段后，将所有 16 位整数相加（包括检验和）。如果该分组中没有差错，则计算得到的和将是 1111111111111111，否则说明分组中有差错。

当检验和错误的时候，该分组一定错误，将会被丢弃。但是当检验和没有错误时，并不能保证分组是完全正确的。虽然 UDP 提供差错检测，但它对差错恢复无能无力。

3.3 可靠数据传输

3.3.1 可靠数据传输（RDT, Reliable Data Transfer）

由于可靠数据传输协议的下层协议也许是不可靠的，信道的不可靠特性决定了可靠数据传输协议的复杂性，例如 TCP 就是在不可靠的端到端网络层之上实现的可靠数据传输协议。

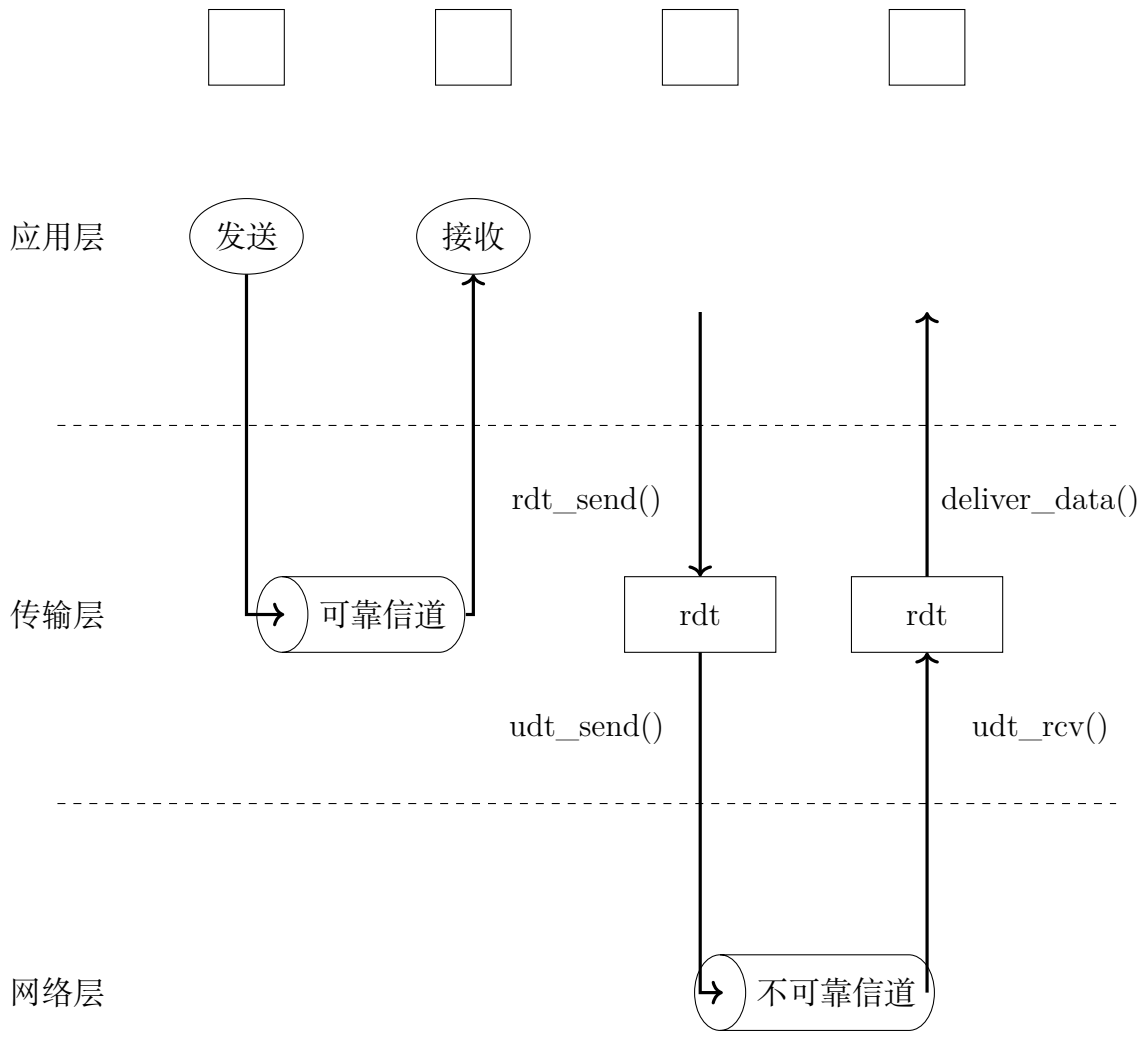

图 3.3: 可靠数据传输

3.3.2 rdt 1.0：经完全可靠信道的可靠数据传输

考虑最简单的情况，假设底层信道是完全可靠的（不会出错、不会丢失）。有限状态机（FSM, Finite State Machine）可以用于描述发送方和接收方的操作。

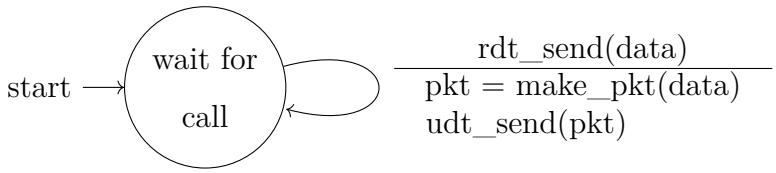

图 3.4: rdt 1.0 发送端

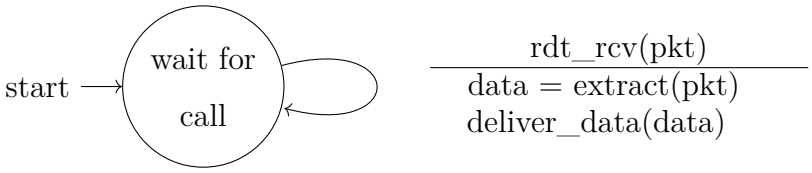

图 3.5: rdt 1.0 接收端

3.3.3 rdt 2.0：经具有比特差错信道的可靠数据传输

在分组的传输、传播或缓存的过程中，分组中的比特可能会受损。类似于打电话，当接听电话的人听到并理解一句话时会说“OK”，但如果没听清，就会要求对方再说一遍。

rdt 2.0 增加了差错检验、接收方反馈和重传机制。当接收方接收到正确的报文时，就给对方一个肯定确认（ACK, positive acknowledgment），告诉他“没问题”。当接收方检测到报文错误时，就需要给对方一个否定确认（NAK, negative acknowledgment），告诉对方“你给我发的不对，重新给我发一份新的吧”。

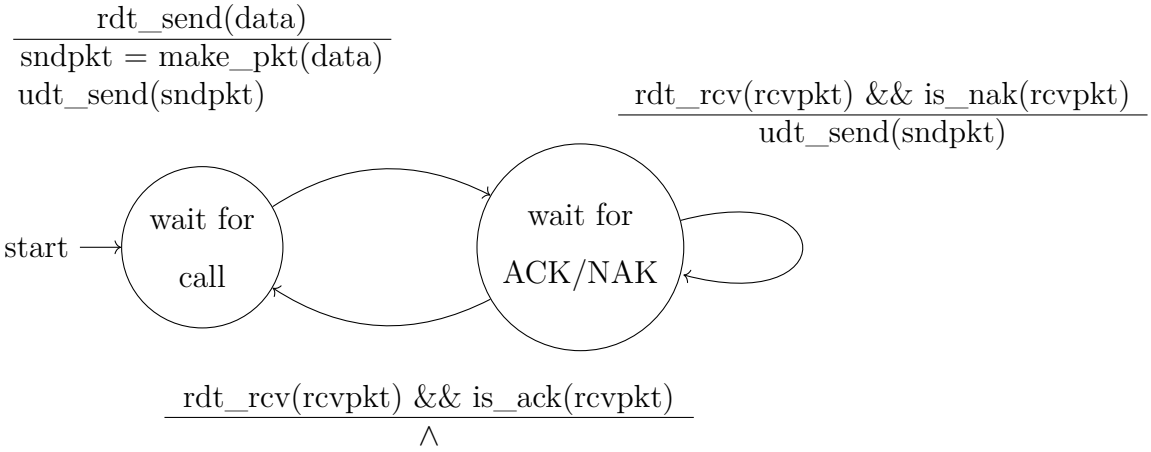

图 3.6: rdt 2.0 发送端

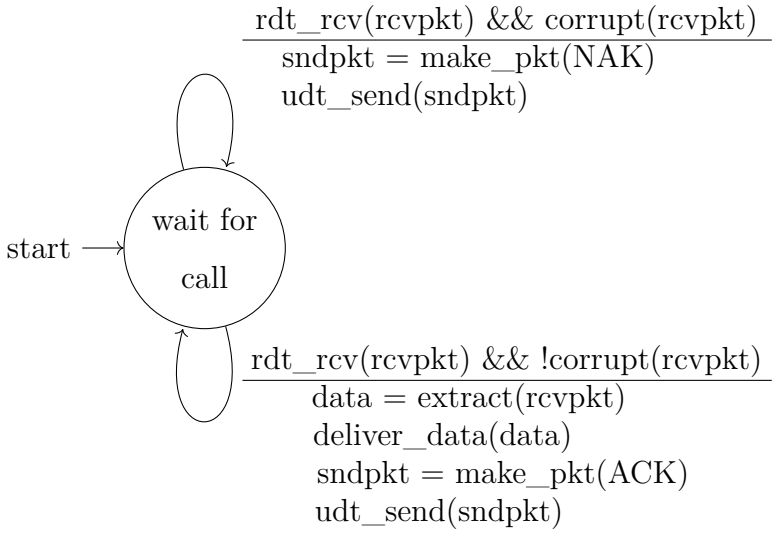

图 3.7: rdt 2.0 接收端

发送方在将分组发送后，等待来自接收方的 ACK 或 NAK。如果收到 ACK，则代表接收方正确地接收了分组，那么就回到初始状态继续等待上层的数据。如果收到 NAK，那就表示接收方没有收到正确的分组，需要进行重传并且继续处于等待 ACK 或 NAK 的状态。像这种只有当发送方接收到 ACK 后才能够继续发送新的报文的协议，被称为停等协议（stop-and-wait）。

rdt 2.0 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——ACK 或 NAK 也存在受损的可能性！

当发送方收到的是一个受损的 ACK 或 NAK 时，如果发送方简单地选择直接重发分组会导致接收方收到重复的分组。

3.3.4 rdt 2.1：带序号消息协议

发送方可以为分组添加序号，接收方只需检查需要就可确定收到的分组是否重复。对于停等协议而言，序号只需要使用 0 和 1 就可以了。

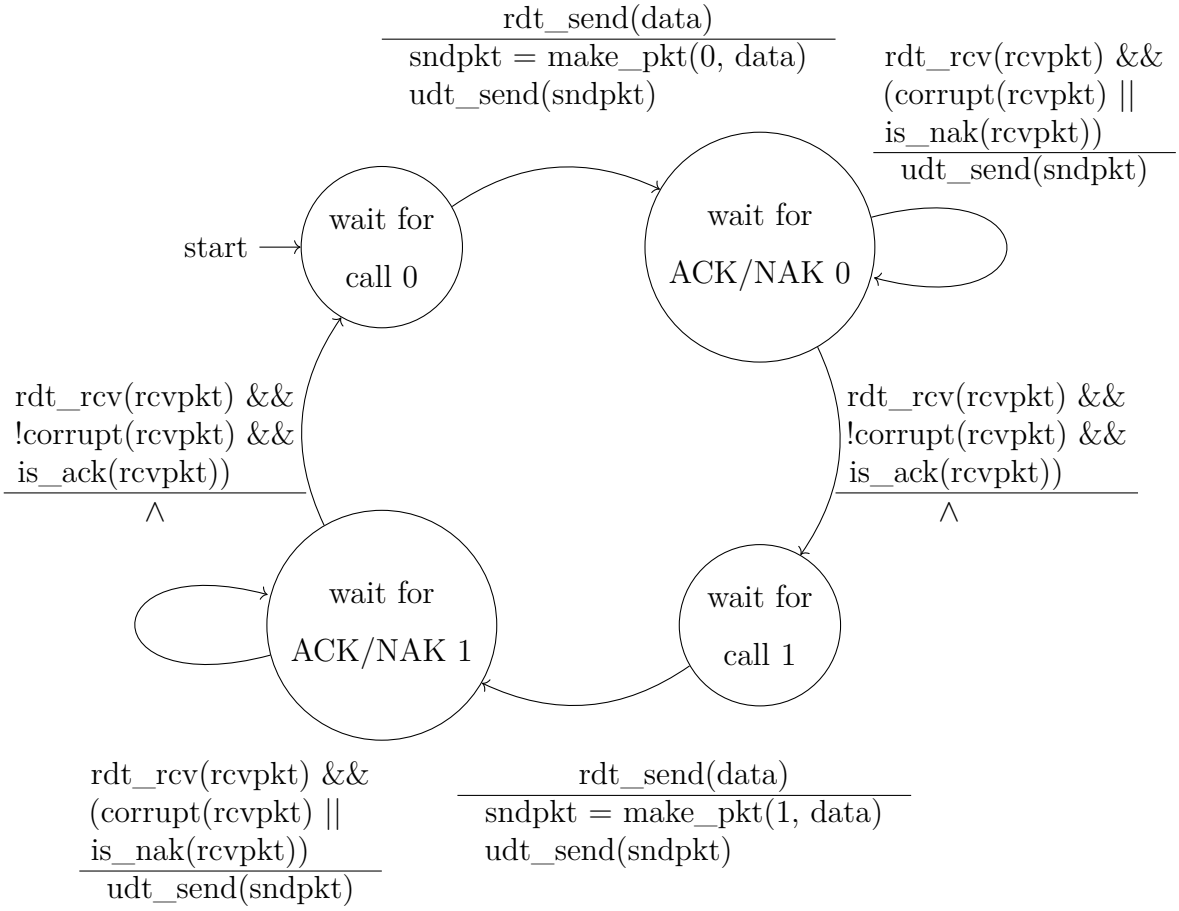

图 3.8: rdt 2.1 发送端

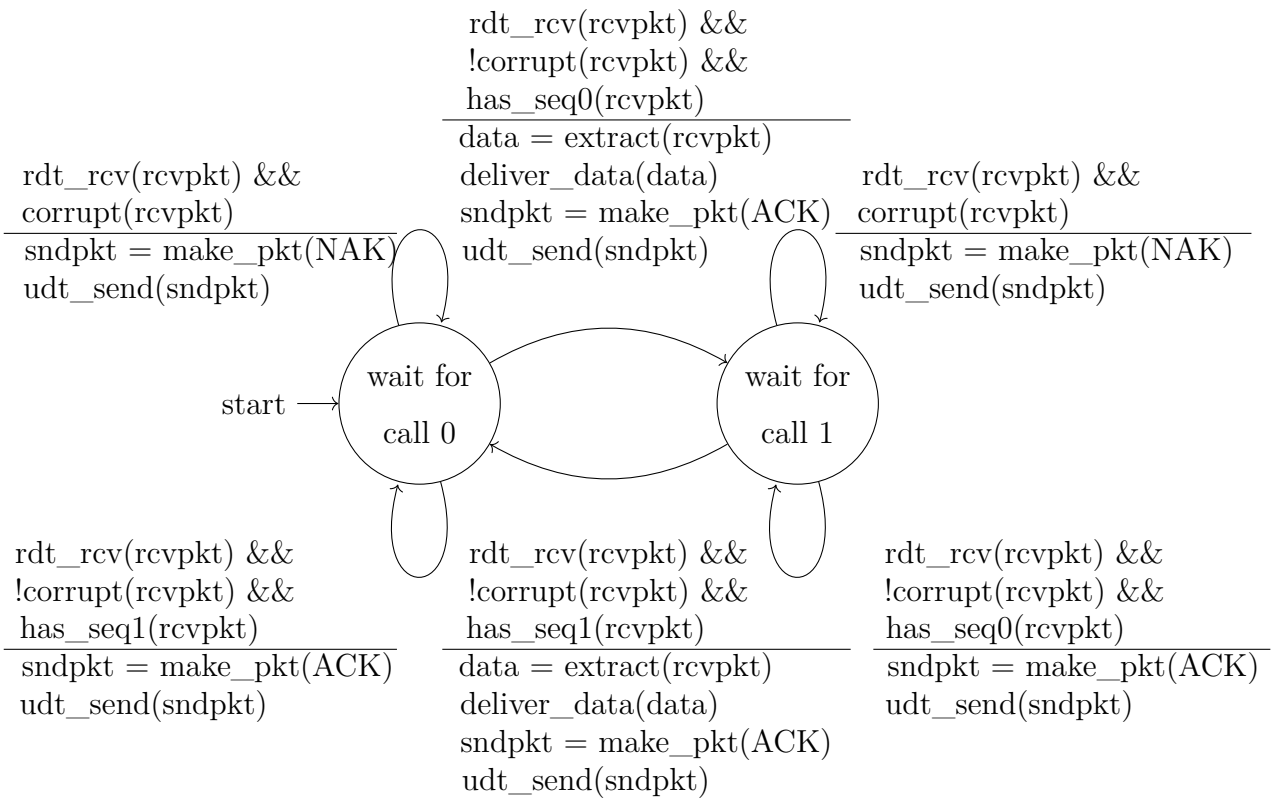

图 3.9: rdt 2.1 接收端

3.3.5 rdt 2.2：无 NAK 消息协议

rdt 2.2 与同 rdt 2.1 的功能相同，但是只使用 ACK。接收方每次 ACK 最后一个被正确接收的分组，当发送方收到重复的 ACK 之后，采用与 NAK 相同的处理动作，将分组重传给接收方。

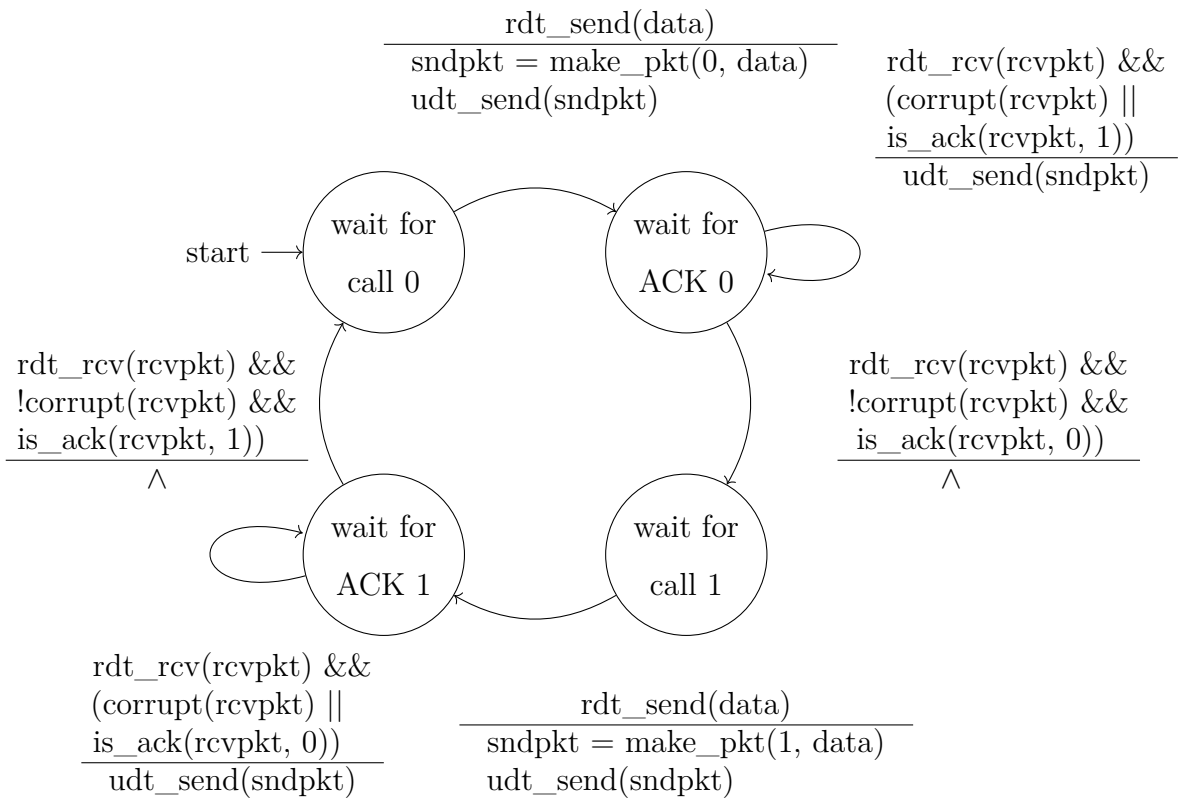

图 3.10: rdt 2.2 发送端

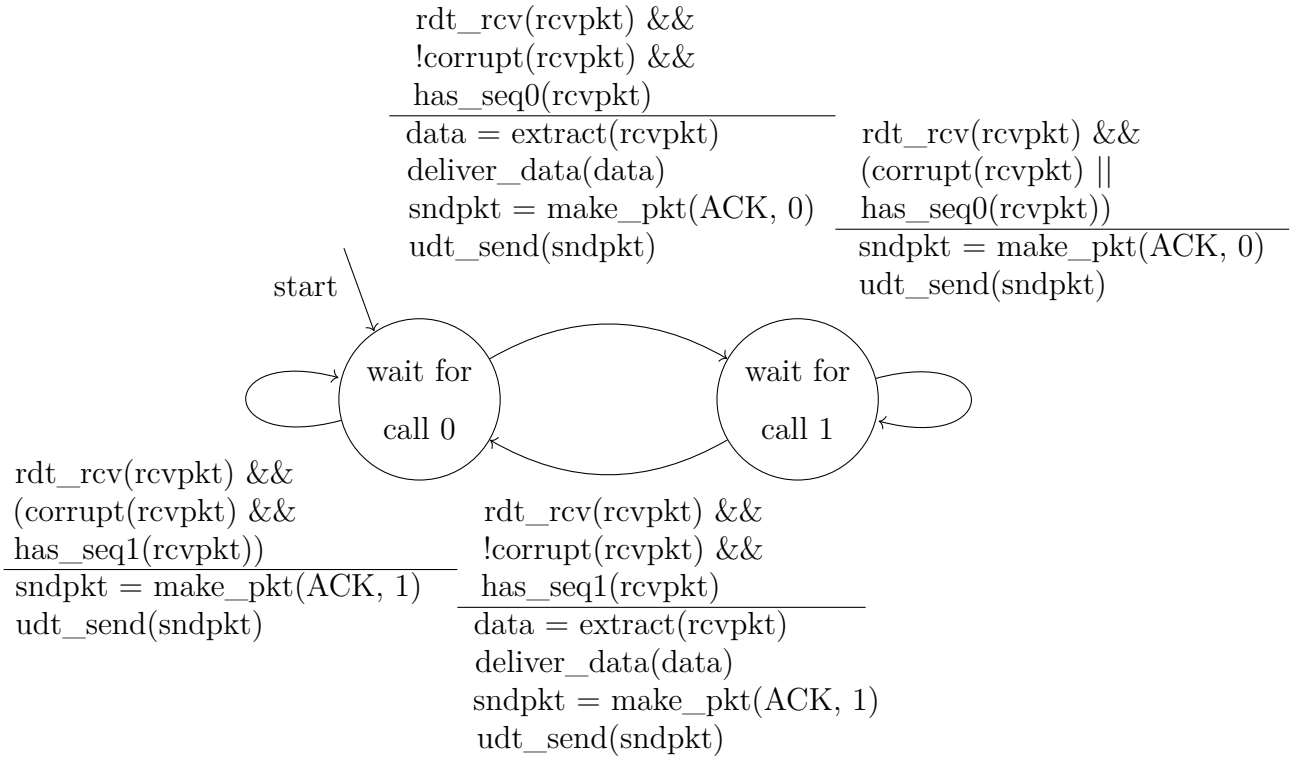

图 3.11: rdt 2.2 接收端

3.3.6 rdt 3.0：经具有比特差错的丢包信道的可靠数据传输

假设现在信道既可能发生错误，也可能会丢失分组。如果发送端发送的数据在半路就丢失了，那么发送方就会无休止的等待，接收方没有收到数据，也会处于等待的状态。

因此发送方需要一个倒计时定时器（countdown timer），如果发送方在规定时间内没有收到 ACK，就将该分组重传。另一种情况是，如果发送的分组并没有丢失，只是延误到达了，这样也会触发定时器重发。但是接收方可以根据分组的序号丢弃重复的分组。

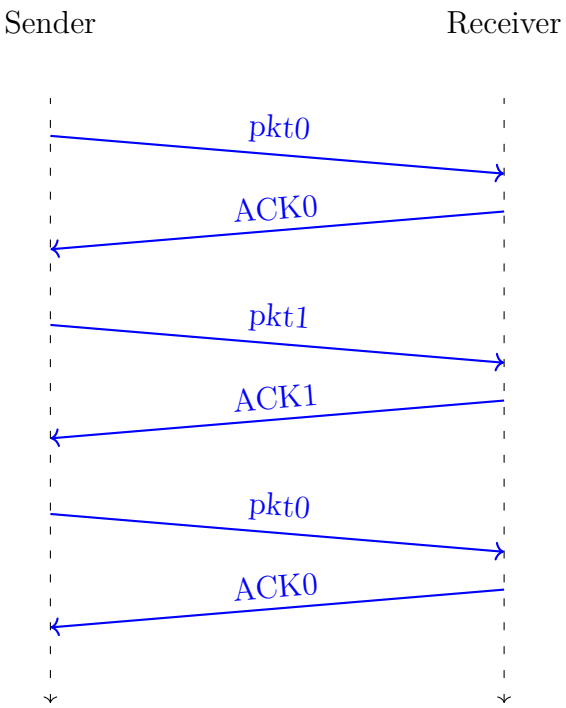

图 3.12: 无丢包

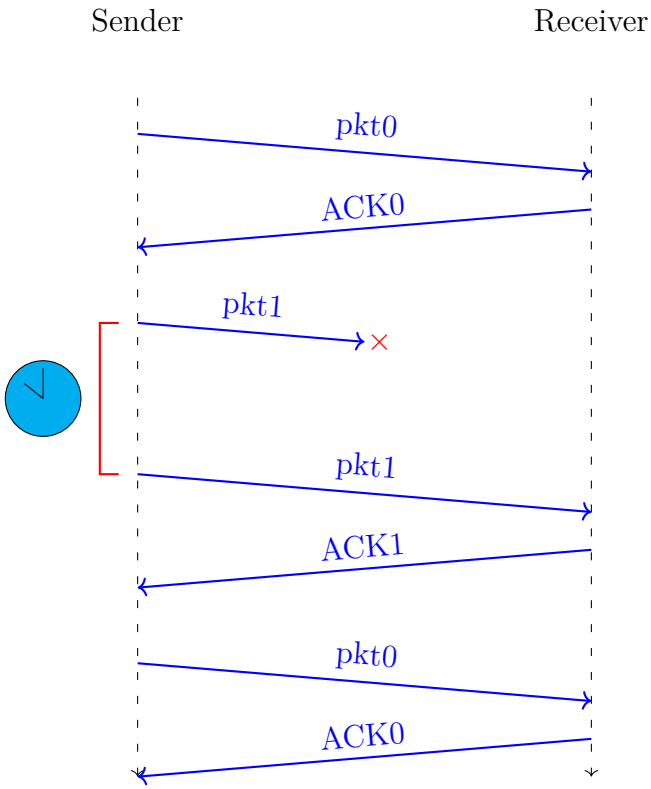

图 3.13: 分组丢失

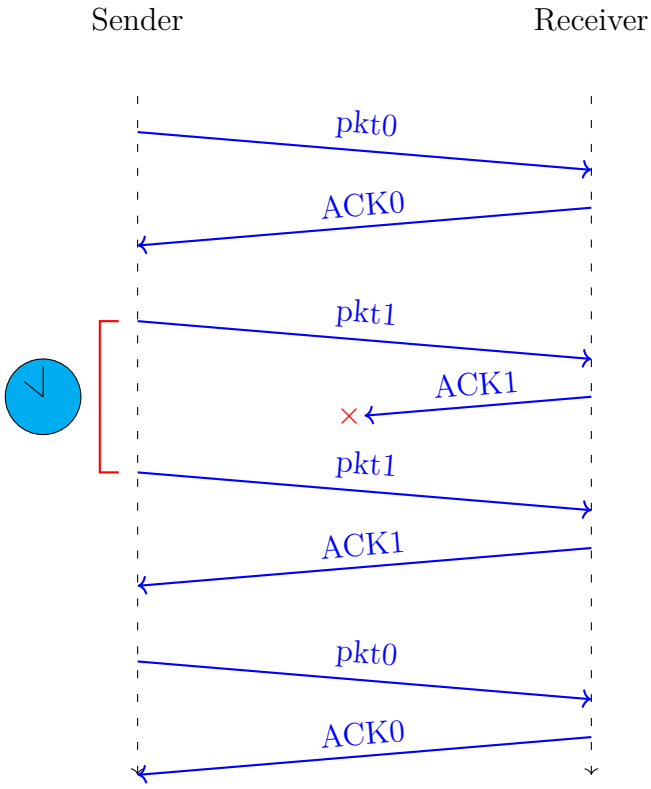

图 3.14: ACK 丢失

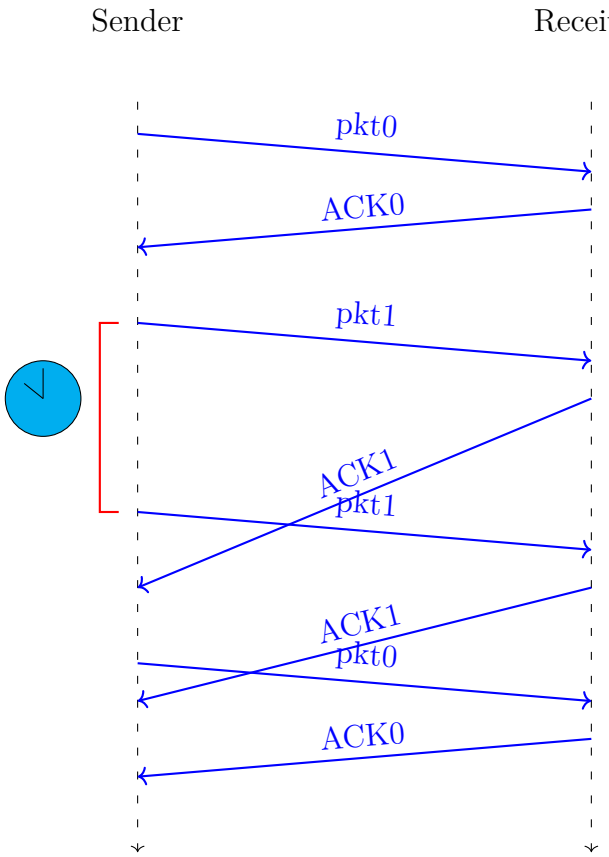

图 3.15: 过早超时

rdt 3.0 能够正确工作，但是性能很差。因为停等协议，导致了发送方的利用率很低。

3.4 流水线

3.4.1 流水线（Pipelining）

为了提高发送方的利用率，可以不以停等方式运行，而是采用流水线协议，允许发送方在收到 ACK 之前连续发送多个分组。

使用流水线协议，就需要更大的序列号范围，因为每个输送中的分组必须有一个唯一的序号。同时发送方和接收方需要更大的存储空间以缓存分组。

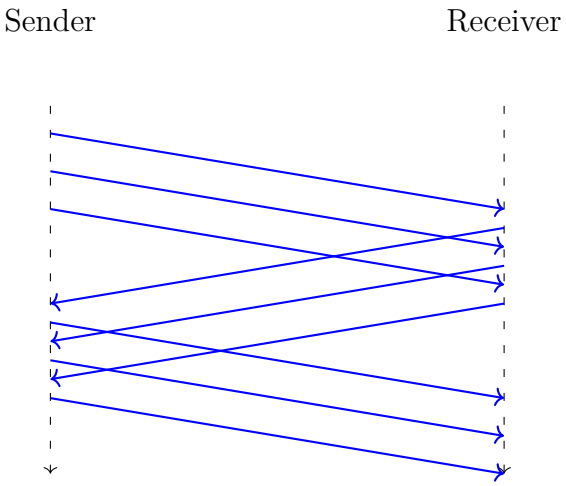

图 3.16: 流水线

解决流水线的差错恢复有两种基本方法：

1. Go-Back-N
2. 选择重传（Selective Repeat）

3.4.2 Go-Back-N (GBN)

在 GBN 协议中，发送方允许发送多个分组，而不需等待确认。在 GBN 协议中，发送方的分组被分为 4 段。

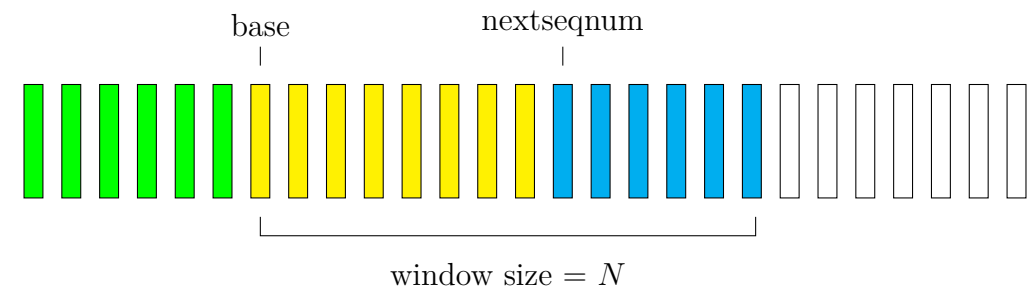

图 3.17: GBN 滑动窗口

其中，$[0, base-1]$ 范围内的序号表示已经发送并被确认的分组，$[base, nextseqnum-1]$ 范围内的序号表示已经发送但未被确认的分组，$[nextseqnum, base+N-1]$ 范围内的序号表示等待被发送的分组，最后，大于等于 $base+N$ 的序号是不可用的。

随着协议的运行，窗口会在序号空间向前滑动，因此，GBN 协议也常被称为滑动窗口协议（sliding-window protocol）。对于接收方，它的接收窗口为 1，因此接收方将会丢弃所有失序的分组。

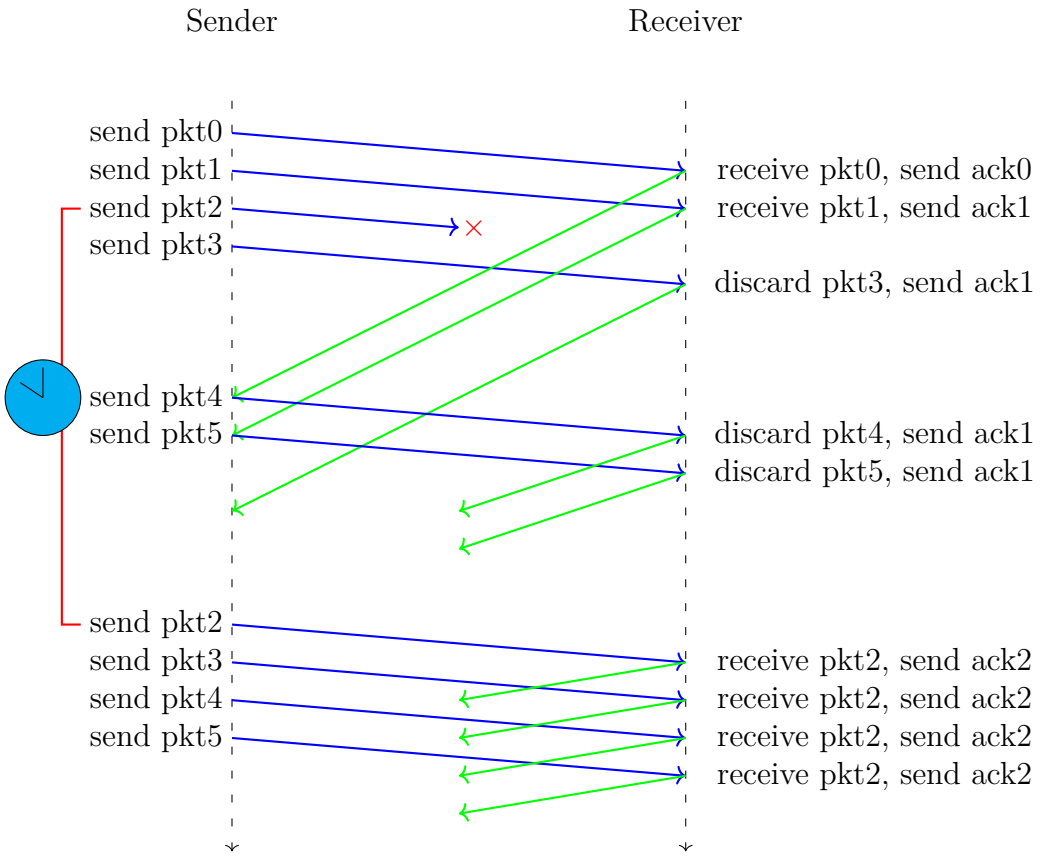

图 3.18: GBN

3.4.3 选择重传

在 GBN 中，单个分组的差错就能够引起大量分组的重传，其中有很多分组其实是没有必要重传的。选择重传协议只需让发送方重传有差错的分组，而对于在窗口内已经接收的分组，将会把失序的分组缓存，直到重新接收到先前丢失的分组，这时才可以将这一批分组按序交付给上层。

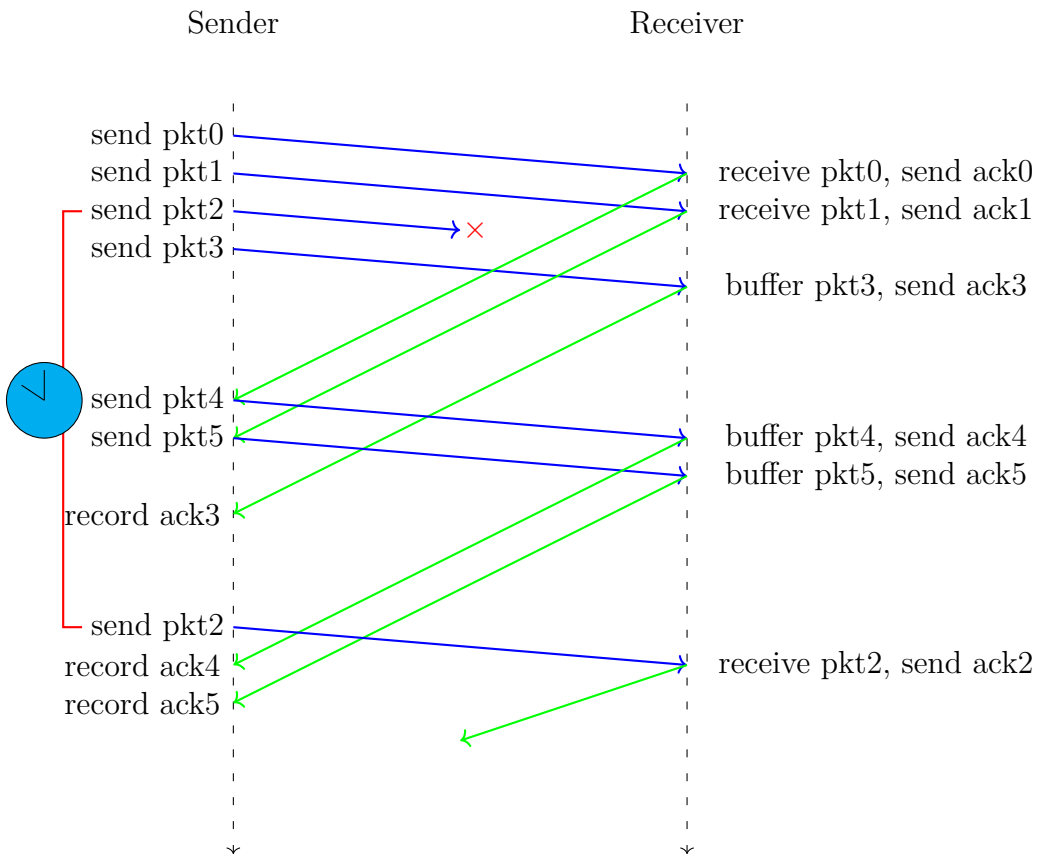

图 3.19: 选择重传

3.5 面向连接传输 TCP

3.5.1 TCP 报文段结构

<table>
<tr><td colspan="5">源端口号</td><td colspan="6">目的端口号</td></tr>
<tr><td colspan="11">Sequence number</td></tr>
<tr><td colspan="11">Acknowledgment number</td></tr>
<tr><td>首部长度</td><td>Unused</td><td>C</td><td>E</td><td>U</td><td>A</td><td>P</td><td>R</td><td>S</td><td>F</td><td>接收窗口</td></tr>
<tr><td colspan="5">校验和</td><td colspan="6">紧急数据指针</td></tr>
<tr><td colspan="11">选项</td></tr>
<tr><td colspan="11">数据</td></tr>
</table>

表 3.2: TCP 报文段结构

TCP 报文段首部主要包括:

- 源端口号/目的端口号: 用于多路复用/分解上层应用的数据。
- 检验和: 检测差错。
- 序号/确认号: 用于实现可靠数据传输。
- 接收窗口: 用于流量控制，指示接收方愿意接受的字节数。
- 首部长度: 由于 TCP 选项字段，首部长度是可变的。通常选项字段为空，所以 TCP 首部的典型长度为 20 字节。
- ACK: 指示确认字段中的值是有效的。
- RST/SYN/FIN: 用于连接建立和拆除。

3.5.2 序号与确认号

TCP 连接的双方都可以随机选择初始序号，这样可以降低上一条 TCP 连接残留的报文段被误认为是新连接的报文段的可能性（如果这两个连接恰好使用了同一个端口）。

主机 A 的确认号是主机 A 期望从主机 B 收到的下一字节的序号。因为 TCP 只确认该流中至第一个丢失字节为止的字节，所以 TCP 被称为提供累计确认。

Telnet 是一个用于远程登录的应用层协议，它运行在 TCP 之上。假设主机 A 发起一个与主机 B 的 Telnet 会话，主机 A 的用于键入的每个字符都会被发送给主机 B，主机 B 将回送每个字符的副本给主机 A，并将这些字符显示在屏幕上，以确保发送的字符得到了处理。

假设主机 A 和 B 的初始序号为 42 和 79，当用户输入一个字符后，这个字符会在网络中传输两次。

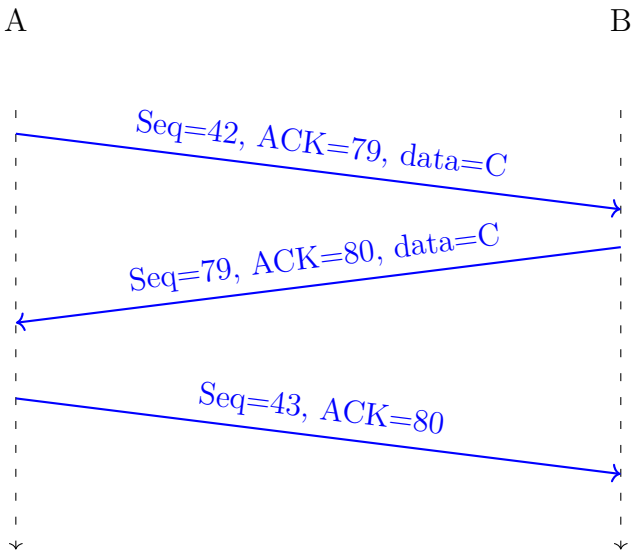

图 3.20: Telnet 序号与确认号

3.5.3 可靠数据传输

TCP 是建立在不可靠的、尽力而为的 IP 服务上的可靠数据传输协议，因此 TCP 采用了超时重传机制，确保了接收端收到的数据是有序无损的。

第一种情况是，假设主机 A 给主机 B 发送一个报文段，序号是 92，包含 8 字节数据。在发出该报文段后，主机 A 需要等待来自主机 B 的确认号为 100 的报文段。虽然主机 B 收到了 A 的报文段，但是 B 发往 A 的确认报文丢失了。在这

种情况下，超时事件就会发生。主机 A 会重传相同的报文段，当主机 B 收到重复报文将会将其丢弃。

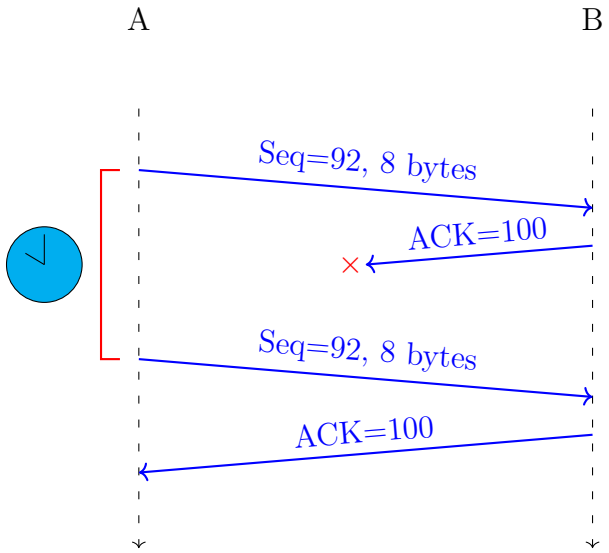

图 3.21: 确认丢失

第二种情况是，假设主机 A 连续发送了两个报文段，第一个报文段序号是 92，包含 8 字节数据，第二个报文段序号是 100，包含 20 字节数据。

这两个报文段都完好无损地到达主机 B，并且主机 B 分别发送了确认号 100 和 120，但是这两个确认报文都没有在超时之前到达。

当超时事件发生时，主机 A 将会重传序号为 92 的第一个报文段。此时，如果第二个报文段的 ACK 在新的超时之前到达，那么第二个报文段就不再会被重传。

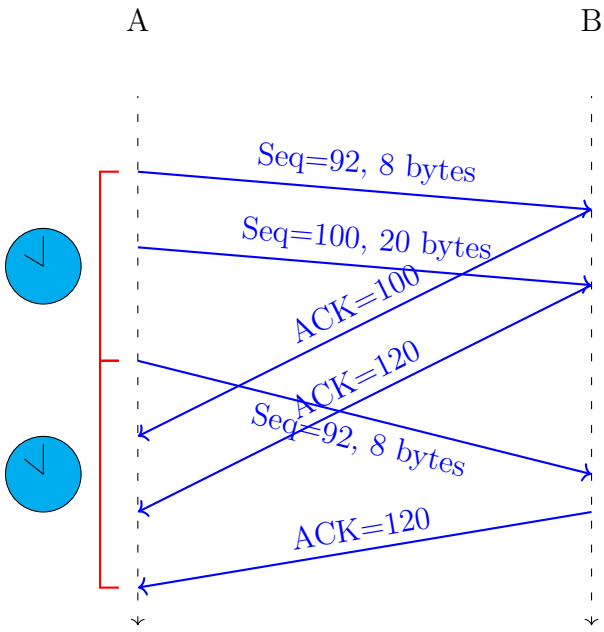

图 3.22: 过早超时

第三种情况是，主机 A 发送两个报文段，其中第一个报文段的确认丢失，但是在超时事件发生之前，主机 A 收到了第二个报文段的确认。由于累计确认，主机知道主机 B 已经收到了之前所有的报文段，因此不会再重传。

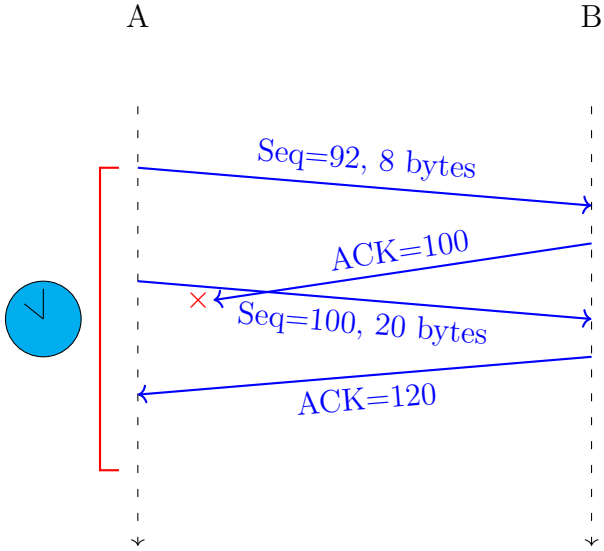

图 3.23: 累计确认

3.5.4 快速重传（Fast Retransmit）

超时触发重传存在一个问题，就是超时周期可能相对较长。当一个报文段丢失时，发送方就被迫延迟重传丢失的分组，因而增加了端到端时延。

通常，发送方可在超时事件发生之前，通过注意冗余的 ACK 可以检测到丢包情况。因为发送方发送大量的报文段，如果一个报文段丢失，就很可能引起许多冗余 ACK。

如果发送方一旦收到 3 个冗余 ACK，就认为该报文段已经丢失，TCP 就会执行快速重传，即在该报文段超时之前就进行重传。

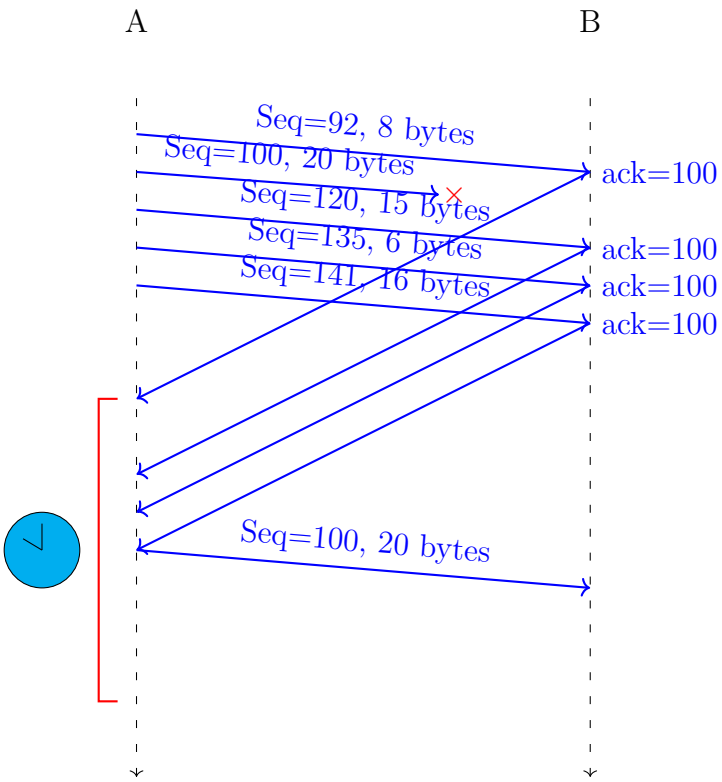

图 3.24: 快速重传

3.5.5 TCP 连接管理

TCP 的运输连接包括连接建立、数据传送和连接释放三个阶段。运输连接管理就是对连接建立以及连接释放过程的管控，使其能正常运行。

连接建立的过程被称为三次握手（three-way handshake）：

1. 第一次握手：客户端的应用进程主动打开，并向服务端发出请求报文段。其首部中，SYN=1、seq=x。

2. 第二次握手：服务器应用进程被动打开，若同意客户端的请求，则发回确认报文。其首部中，SYN=1、ACK=1、ack=x+1、seq=y。

3. 第三次握手：客户端收到确认报文后，通知上层应用进程连接已建立，并向服务器发出确认报文。其首部，ACK=1、ack=y+1。当服务器收到客户端的确认报文后，也通知其上层应用进程连接已建立。

连接释放的过程被称为四次挥手：

1. 第一次挥手：数据传输结束后，客户端应用进程发出连接释放报文段，并停止发送数据。其首部，FIN=1、seq=u。

2. 第二次挥手：服务器收到连接释放报文段后，发出确认报文。其首部，ack=u+1、seq=v。此时本次连接就进入了半关闭状态，客户端不再向服务器发送数据，而服务器仍会继续发送。

3. 第三次挥手：若服务器已经没有要向客户端发送的数据，其应用进程就通知服务器释放 TCP 连接。这个阶段服务器所发出的最后一个报文的首部中，FIN=1、ACK=1、seq=w、ack=u+1。

4. 第四次挥手：客户端收到连接释放报文段后，必须发出确认，ACK=1、seq=u+1、ack=w+1。再经过 2 倍最长报文段寿命 MSL（Maximum Segment Lifetime），本次 TCP 连接真正结束。