
Computer Network Note 笔记

基于计算机网络笔记



ElegantLaTeX Program

Victory won't come to us unless we go to it.

作者: Miao

时间: July 12, 2023

邮箱: chenmiao.ku@gmail.com

版本: 0.10

目 录



1 计算机网络和因特网	5
1.1 什么是因特网	5
1.1.1 具体构成描述	5
1.1.2 服务描述	6
1.1.3 什么是协议	6
1.2 网络边缘	7
1.2.1 接入网	7
1.2.2 物理媒体	9
1.3 网络核心	11
1.3.1 分组交换	11
1.4 分组交换网中的时延、丢包和吞吐量	16
1.4.1 分组交换网中的时延概述	16
1.4.2 排队时延和丢包	18
1.4.3 端到端时延	19
1.4.4 计算机网络中的吞吐量	19
1.5 协议层次及其服务模型	20
1.5.1 分层的体系结构	20
1.5.2 封装	22
1.6 面对攻击的网络	23
1.6.1 通过因特网放入病毒	23
1.6.2 攻击服务器和网络基础设施	24
1.6.3 嗅探分组	24
1.6.4 伪装	25
2 应用层	26
2.1 应用层协议原理	26
2.1.1 网络应用程序体系结构	26
2.1.2 进程通信	27
2.1.3 可供应用程序使用的运输服务	28
2.1.4 因特网提供的运输服务	29
2.1.5 应用层协议	31

2.2	Web 和 HTTP	31
2.2.1	HTTP 概况	31
2.2.2	非持续连接和持续连接	32
2.2.3	HTTP 报文格式	34
2.2.4	用户与服务器的交互: cookie	37
2.2.5	Web 缓存	38
2.2.6	条件 GET 方法	39
2.3	因特网中的电子邮件	40
2.3.1	SMTP	41
2.3.2	与 HTTP 的对比	43
2.3.3	邮件报文格式	43
2.3.4	邮件访问协议	44
2.4	DNS: 因特网的目录服务	46
2.4.1	DNS 提供的服务	46
2.4.2	DNS 工作机原理	47
2.4.3	DNS 记录和报文	50
2.5	视频流和内容分发网	51
2.5.1	因特网视频	51
2.5.2	HTTP 流和 DASH	51
2.6	套接字编程: 生成网络应用	52
2.6.1	TCP 套接字编程	55
2.7	课后习题	59
3	运输层	60
3.1	概述和运输层服务	60
3.1.1	运输层和网络层的关系	61
3.1.2	因特网运输层概述	61
3.2	多路复用与多路分解	61
3.3	无连接运输: UDP	63
3.3.1	UDP 报文段结构	64
3.3.2	UDP 校验和	64
3.4	可靠数据传输原理	64
3.4.1	构造可靠数据传输协议	65
3.5	面向连接的运输: TCP	69
3.5.1	TCP 连接	69
3.5.2	TCP 报文结构	70
3.5.3	往返实践的估计与超时	72



3.5.4 可靠数据传输	74
3.5.5 流量控制	77
3.5.6 TCP 链接管理	78
3.6 拥塞控制原理	80
3.6.1 拥塞原因与代价	81
3.6.2 拥塞控制方法	82



第1章 计算机网络和因特网



1.1 什么是因特网

我们可以用两种方式来回答：

- 1) 描述因特网的具体构成，即构成因特网的基本硬件和软件组件
- 2) 根据为分布式应用提供服务的联网基础设施来描述因特网

1.1.1 具体构成描述

用因特网术语来说，所有这些设备都称为主机 (*host*) 或端系统 (*endsystem*)。端系统通过通信链路 (*communicationlink*) 和分组交换机 (*packetswitch*) 连接到一起。

通信链路，简单来说就是不同类型的物理媒介，例如光纤，铜线等，而链路的传输速率 (*transmissionrate*) 以比特/秒 (*bit/s*, 或 *bps*) 度量。当一台端系统要向另一台端系统发送数据时，发送端需要将数据进行分段，然后将每段数据加上首部字节 (很显然，这部分的知识点一开始难以理解，因为是属于更下层的知识)。因此，这样形成的信息包在计算机网络中被称为分组 (*packet*)。

而分组交换机就和其名字一样，它接受到达的分组，然后从一条出通信链路转发该分组。目前，最为常见的分组交换机为：路由器 (*route*) 和链路层交换机 (*link-layer switch*)。

- 路由器通常用于网络核心
- 链路层交换机通常接入网中

从发送端系统到接收端系统，一个分组所经历的一系列通信链路和分组交换机称为通过该网络的路径 (*oroute* 或 *path*)

端系统通过因特网提供商 (*InternetServiceProvider, IPS*) 接入因特网，每个 *IPS* 自身就是一个由多台分组交换机和多段通信链路组成的网络。

- 各 *IPS* 为端系统提供了各种不同类型的网络接入，同时 *IPS* 也为内容提供者提供因特网接入服务
- 较低层的 *IPS* 通过国家、国际的较高层 *IPS* 互联
- 较高层的 *IPS* 通过高速光纤链路互联的高速路由器组成

1.1.2 服务描述

前面我们辨识了构成因特网的许多部件，此处我们可以从一个完全不用的角度，即为应用程序提供服务的基础设施的角度来描述因特网。因为被描述的应用程序涉及多个互相交换数据的端系统，故它们被称为分布式应用程序 (*distributed application*)。

需要注意的是：因特网应用程序运行在端系统上，即它们并不运行在网络核心中的分组交换机中。尽管分组交换机能够加速端系统之间的数据交换，但它们并不在意作为数据的源或宿的应用程序。

为了使得运行在端上的应用程序能够向另一个端上发送信息，因此就有了套接字接口 (*socket interface*)，该接口规定了运行在一个端系统上的程序请求因特网基础设施向运行在另一个端系统上的特定目的地程序交付数据的方式。因特网套接字接口是一套发送程序必须遵循的规则集合。

1.1.3 什么是协议

理解计算机网络协议 (*protocol*) 最简单的方式显然是用人类本身来类比，因为实际上，我们人类无时无刻不在执行协议。

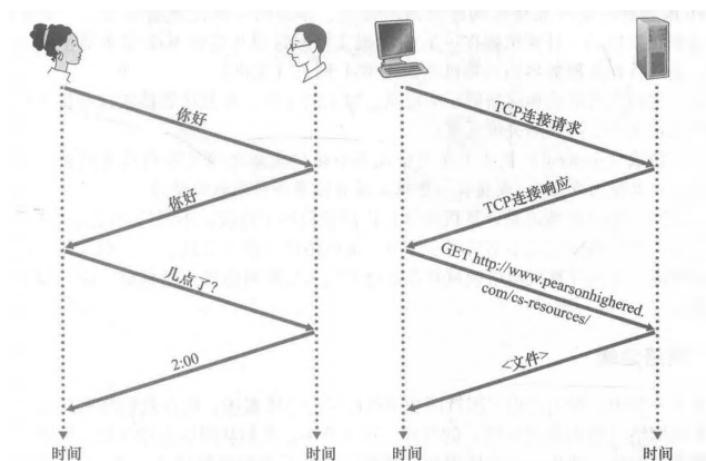


图 1.1: 人类活动类比协议

可以看见，人类协议（或者说好的行为方式）要求一方先发起问候，然后开始与另外一人的通信。那么在计算机中对应的便是，端系统对另一个端系统发起通信。注意：在我们人类协议中，有我们发送的特定报文，也有我们根据接收到的应答报文或其他事件（例如在某个给定的时间内没有回答采取的动作）。

那么反过来，如果是一个坏的行为方式（比如一上来就有人破口大骂），那么你也不想继续进行通信，也就是说，该协议就不能交互，从而无法完成工作。那么网络也是同理。

网络协议类似于人类协议，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：



- 1) 硬件实现的协议控制了两块网络接口卡间的“线上”的比特流
- 2) 拥塞控制协议控制了在发送方和接收方之间传输的分组发送的速率
- 3) 路由器中的协议决定了分组从源到目的地的路径
- n) ...

最终，我们可以得出协议的定义：协议 (*protocol*) 定义了在两个或多个通信实体之间交换的报文的格式和顺序，以及报文发送和/或接收一条报文或其他事件所采取的动作。

1.2 网络边缘

在计算机网络中，通常把与因特网相连的计算机和其他设备称为端系统，因为它位于因特网的边缘。

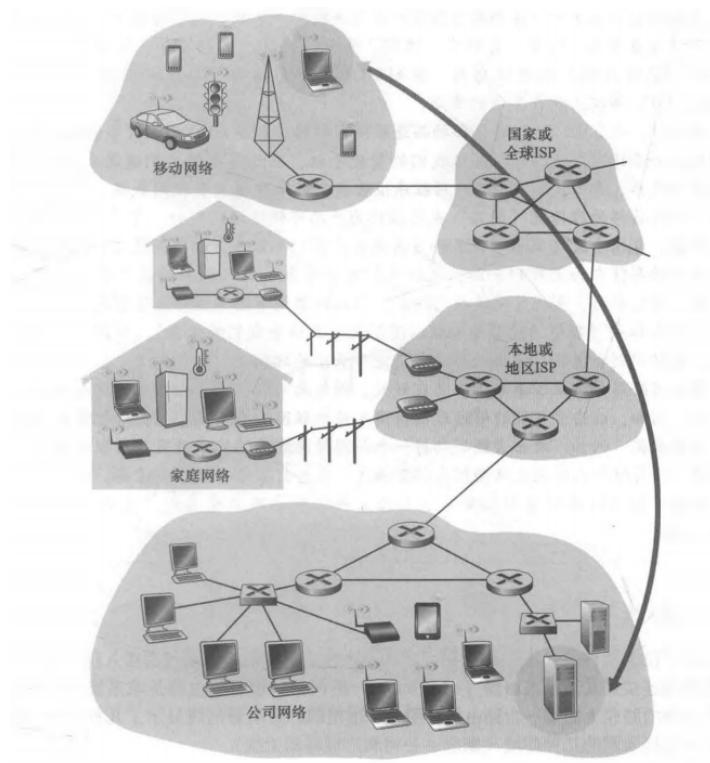


图 1.2: 端系统交互

1.2.1 接入网

接入网是指将端系统物理连接到其边缘路由器 (*edgerouter*) 的网络。边缘路由器是端系统到任何其他远程端系统的路径上的第一台路由器。



家庭接入：DSL、电缆、FTTH、拨号和卫星

如今，宽带入宅接入有两种最为流行的方式：数字用户线 (*Digital Subscriber Line, DSL*) 和电缆。

用户从运营商公司获取 DLS 接入，通过 DLS 调制解调器使用电话线与运营商的本地中心局 (*CO*) 中的数字用户接入复用器 (*DSLAM*) 交换数据。用户的 DSL 调制解调器获得数字数据后将其转换为高音频，通过电话线传输给本地中心局。

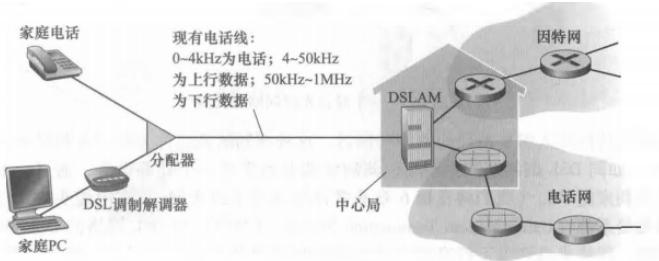


图 1.3: DSL 因特网接入

家庭电话线同时承载了数据和传统的电话信号，且用不同的频率进行编码。这种方法使得一根 DSL 线路能够看上去像三根独立的线路一样。：

- 高速下行信道，位于 50kHz 到 1MHz 频段
- 中速上行信道，位于 4kHz 到 50kHz 频段
- 普通的双向电话信道，位于 0 到 4kHz 频段

DSL 利用了电话公司现有的本地电话基础设施，而电缆因特网接入 (*cable Internet access*) 利用了有线电视公司现有的有线电视基础设施。如下图所示，光缆将电缆头端连接到地区枢纽，从这里使用传统的同轴电缆达到各家各户。因为在这个系统中应用了光纤和同轴电缆，因此被称为混合光纤同轴 (*Hybrid Fiber Coax, HFC*) 系统。

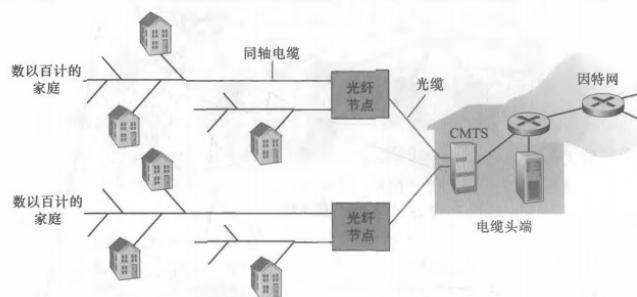


图 1.4: 混合光纤同轴接入网

电缆因特网接入需要特殊的电缆调制解调器 (*cablemodem*)。电缆调制解调器通常是一个外部设备，通过一个以太网端口连接到家庭 PC。在电缆头端，电缆调制解调器端接系统 (*CMTS*) 与 DSL 网络的 DSLAM 具有类似的功能。

电缆因特网接入的一个重要特征是共享广播媒体。特别是，由头端发送的每个分组向下行经每段链路到每个家庭；每个家庭发送的每个分组经上行信道向头端传输。这就是为什么我们在宿舍高峰期会感觉明显的网速变慢的原因。



另一种提供更为高速率的新兴技术是光纤到户 (*Fiber To The Home, FTTH*)，顾名思义，FTTH 直接从本地中心局到家庭提供了一条光纤路径。

企业 (和家庭) 接入：以太网和 WiFi

在公司和大学校园以及越来越多的家庭环境中，使用局域网 (LAN) 将端系统连接到边缘路由器。尽管有许多不同类型的局域网技术，但是以太网到目前为止是公司、大学和家庭网络中最为流行的接入技术。

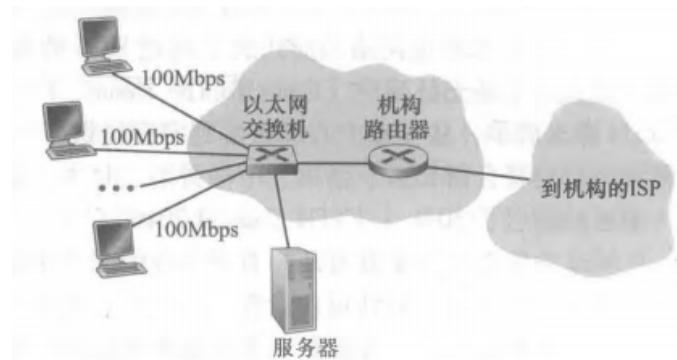


图 1.5: 以太网因特网接入

然而，越来越多的人从便携机、智能手机、平板电脑和其他物品无线接入因特网 (参见前面的插入内容“物联网”)。在无线 LAN 环境中，无线用户从/到一个接入点发送/接收分组，该接入点与企业网连接 (很可能使用了有线以太网)，企业网再与有线因特网相连。一个无线 LAN 用户通常必须位于接入点的几十米范围内。基于 IEEE 802.11 技术的无线 LAN 接入，更通俗地称为 WiFi，目前几乎无所不在。

广域无线接入：3G 和 LTE

iPhone 和安卓等设备越来越多地用来在移动中发信息、在社交网络中分享照片、观看视频和放音乐。与 WiFi 不同的是，一个用户仅需要位于基站的数万米 (而不是几十米) 范围内。

1.2.2 物理媒体

当我们描述上述技术时，我们也指出了所使用的物理媒体。在这一节，我们简要概述一下常在因特网中使用的传输媒体。

首先我们想象一下一个比特的短暂历程，该比特从一个端系统开始传输，通过一系列链路和路由器，到达另一个端系统。这个比特从源到目的地传输时，通过系列“发射器-接收器”对。

对于每个发射器-接收器对，通过跨越一种物理媒体 (physical medium) 传播电磁波或光脉冲来发送该比特。该物理媒体可具有多种形状和形式，并且对沿途的每个发射



器-接收器对而言不必具有相同的类型。物理媒体的例子包括双绞铜线、同轴电缆、多模光纤缆、陆地无线电频谱和卫星无线电频谱。

物理媒体分成两种类型：

- 导引型媒体 (*guidedmedia*)，电波沿着固体媒体前行，如铜线、双绞铜线或同轴电缆
- 非引导型媒体 (*unguidedmedia*)，点播在空气或外层空间中传播，例如在无线局域网或数字卫星频道中

物理链路(铜线、光缆等)的实际成本与其他网络成本相比通常是相当小的。特别是安装物理链路的劳动力成本能够比材料的成本高出几个数量级。

双绞铜线

最便宜并且最常用的导引型传输媒体是双绞铜线。双绞线由两根绝缘的铜线组成，每根大约 1mm 粗，以规则的螺旋状排列着。这两根线被绞合起来，以减少邻近类似的双绞线的电气干扰。通常许多双绞线捆扎在一起形成一根电缆，并在这些双绞线外面覆盖上保护性防护层。一对电线构成了一个通信链路。

无屏蔽双绞线 (*UnshieldedTwistedPair, UTP*) 常用在建筑物内的计算机网络中，即用于局域网 (LAN) 中。目前局域网中的双绞线的数据速率从 10Mbps 到 10Gbps 所能达到的数据传输速率取决于线的粗细以及传输方和接收方之间的距离。

双绞线最终已经作为高速 LAN 联网的主导性解决方案。

同轴电缆

与双绞线类似，同轴电缆由两个铜导体组成，但是这两个导体是同心的而不是并行的。借助于这种结构及特殊的绝缘体和保护层，同轴电缆能够达到较高的数据传输速率。同轴电缆在电缆电视系统中相当普遍。

同轴电缆能被用作导引型共享媒体 (*sharedmedium*)。特别是，许多端系统能够直接与该电缆相连，每个端系统都能接收由其他端系统发送的内容。

光纤

光纤是一种细而柔软的、能够导引光脉冲的媒体，每个脉冲表示一个比特。一根光纤能够支持极高的比特速率，高达数十甚至数百 Gbps。它们不受电磁干扰，长达 100km 的光缆信号衰减极低，并且很难窃听。这些特征使得光纤成为长途导引型传输媒体，特别是跨海链路。

陆地无线电通道

无线电通道承载电磁频谱中的信号。它不需要安装物理线路，并具有穿透墙壁、提供与移动用户的连接以及长距离承载信号的能力，因而成为一种有吸引力的媒体。



无线电信道的特性极大地依赖于传播环境和信号传输的距离。环境上的考虑取决于路径损耗和遮挡衰落(即当信号跨距离传播和绕过/通过阻碍物体时信号强度降低)、多径衰落(由于干扰对象的信号反射)以及干扰(由于其他传输或电磁信号)

陆地无线电信道能够大致分为三类:

- 1) 第一类运行在很短距离(如1米或2米)
- 2) 第二类运行在局域,通常跨越数十到几百米
- 3) 第三类运行在广域,跨越数万米

卫星无线电信道

一颗通信卫星连接地球上的两个或多个微波发射器/接收器,它们被称为地面站。该卫星在一个频段上接收传输,使用一个转发器(下面讨论)再生信号,并在另一个频率上发射信号。

通信中常使用两类卫星:

- 1) 同步卫星(*geostationary satellite*),同步卫星永久地停留在地球上方的相同点上。这种静止性是通过将卫星置于地球表面上方36000km的轨道上而取得的。
- 2) 近地轨道卫星(*Low-Earth Orbiting LEO*),近地轨道卫星放置得非常靠近地球,并且不是永久地停留在地球上方的一个点。

1.3 网络核心

网络核心是指由互联网因特网端系统的分组交换机和链路构成的网状网络。

1.3.1 分组交换

在网络应用中,端系统彼此交换报文。为了从源端系统向目的端系统发送一个报文,源将长报文划分为较小的数据块,称之为分组。

在源和目的地之间,每个分组都通过通信链路和分组交换机(packet switch)传送。(交换机主要有两类:路由器(router)和链路层交换机(link-layer switch)。)分组(数据包)以等于该链路最大传输速率的速度传输通过通信链路。

因此,如果某端系统或分组交换机经过一条链路发送一个 L 比特的分组,链路的传输速率为 R 比特/秒,则传输该分组的时间为 L/R 秒

存储转发传输

多数分组交换机在链路的输入端使用存储转发传输(store-and-forward transmission)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比特之前,必须接收到整个分组。



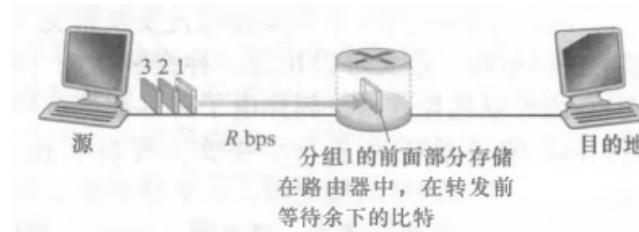


图 1.6: 存储转发传输

在图 1.6 所示的特定时刻，源已经传输了分组 1 的一部分，分组 1 的前沿已经到达了路由器。因为该路由器应用了存储转发机制，所以此时它还不能传输已经接收的比特，而是必须先缓存（即“存储”）该分组的比特。仅当路由器已经接收完了该分组的所有比特后，它才能开始向出链路传输（即“转发”）该分组。

对于一般情况（暂不考虑传播时延）：通过由 N 条速率均为 R 的链路组成的路径（意味着在源和目的地中由 $N - 1$ 台路由器），从源到目的地发送一个分组，我们看到的端对端时延是：

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

排队延时和分组丢失

对于每条相连的链路，该分组交换机具有一个输出缓存（output buffer，也称为输出队列（output queue）），它用于存储路由器准备发往那条链路的分组。该输出缓存在分组交换中起着重要的作用。

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他分组，该到达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（queuing delay）。

这些时延是变化的，变化的程度取决于网络的拥塞程度。又因为缓存空间的大小是有限的，因此可能存在一个到达的分组但是缓存已经充满的情况，此时就将出现分组丢失（丢包，packet loss）。

转发表和路由选择协议

路由器从与它相连的一条通信链路得到分组，然后向与它相连的另一条通信链路转发该分组。但是路由器怎样决定它应当向哪条链路进行转发呢？

在因特网中，每个端系统具有一个称为 IP 地址的地址。当源主机要向目的端系统发送一个分组时，源在该分组的首部包含了目的地的 IP 地址。当一个分组到达网络中的路由器时，路由器检查该分组的目的地址的一部分，并向一台相邻路由器转发该分组。



更特别的是，每台路由器具有一个转发表 (forwarding table)，用于将目的地址 (或目的地址的一部分) 映射成为输出链路。当某分组到达一台路由器时，路由器检查该地址，并用这个目的地址搜索其转发表，以发现适当的出链路。路由器则将分组导向该出链路。

电路交换

通过网络链路和交换机移动数据有两种基本方法：电路交换 (circuit switching) 和分组交换 (packet switching)。

在电路交换网络中，在端系统间通信会话期间，预留了端系统间沿路径通信所需要的资源（缓存，链路传输速率）。在分组交换中，这些资源则不会预留，但是得承担其不得不等待接入通信线路的代价。

传统的电话网络是电路交换网络的例子。在发送方能够发送信息之前，该网络必须在发送方和接收方之间建立一条连接。这是一个名副其实的连接，因为此时沿着发送方和接收方之间路径上的交换机都将为该连接维护连接状态。

用电话的术语来说，该连接被称为一条电路 (circuit)。当网络创建这种电路时，它也在连接期间在该网络链路上预留了恒定的传输速率（表示为每条链路传输容量的一部分）。既然已经为该发送方-接收方连接预留了带宽，则发送方能够以确保的恒定速率向接收方传送数据。

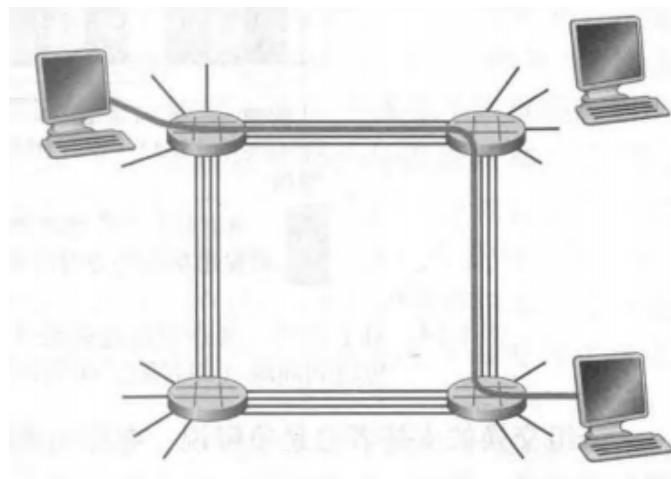


图 1.7: 简单电路交换网络

如上图，在这个网络中，用四条链路互联了四台电路交换机。当两台主机要通信时，该网络在两台主机之间创建一条专用的端到端链接 (*end-to-end*)。因为每条链路具有四条电路，对于由端到端链接所使用的每条链路而言，该连接在连接期间获得链路总传输总量的 $1/4$ 。

电路交换网络中的复用 链路中的电路是通过频分复用 (Frequency-Division Multiplexing, FDM) 或时分复用 (Time-Division Multiplexing, TDM) 来实现的。



对于 FDM，链路的频谱由跨越链路创建的所有连接共享。特别是，在连接期间链路为每条连接专用一个频段。FDM 通过将不同的信号调制到不同的频带上，然后将这些频带合并到一个复合信号中，实现了多路信号的同时传输。在 FDM 系统中，链路的频谱被划分为一系列不重叠的子带，每个子带用于传输一个独立的信号或连接。每个连接被分配一个特定的频带，该频带在整个连接的生命周期中都是专用的。

由于每个连接在链路上使用独占的频段，因此不同连接之间的信号不会相互干扰。这使得多个连接可以同时在链路上传输，并且可以实现频谱的高效利用。

需要注意的是，FDM 是一种静态频分复用技术，即连接在建立时会占用固定的频段，并且在连接的整个生命周期内保持不变。由于频谱资源是有限的，所以在设计 FDM 系统时，需要合理划分频带以满足所有连接的需求，并确保频段之间的不重叠，以避免干扰和交叉干扰。

对于于一条 TDM 链路，时间被划分为固定期间的帧，并且每个帧又被划分为固定数量的时隙。当网络跨越一条链路创建一条连接时，网络在每个帧中为该连接指定一个时隙。这些时隙专门由该连接单独使用，一个时隙（在每个帧内）可用于传输该连接的数据。

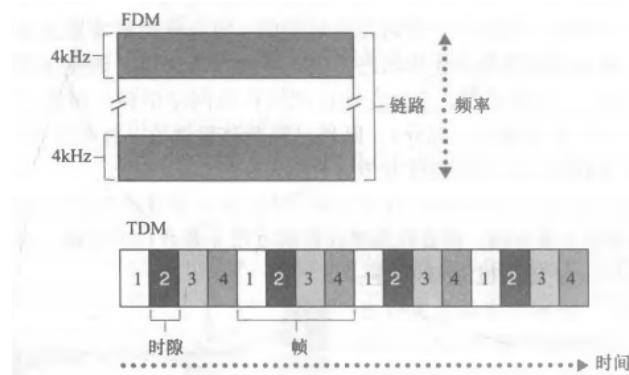


图 1.8: FDM 和 TDM 的不同演示

分组交换与电路交换的对比 分组交换的批评者经常争辩说，分组交换不适合实时服务（例如，电话和视频会议），因为它的端到端时延是可变的和不可预测的（主要是因为排队时延的变动和不可预测所致）。

分组交换的支持者却争辩道：

- 1) 它提供了比电路交换更好的带宽共享
- 2) 它比电路交换更简单、更有效，实现成本更低

其实有一个例子能够很好的证明分组交换会更加有效，只要同时活跃的用户不够多的情况下，分组交换总是优于电路交换，因为分组交换没有静默期这一情况。分组交换差不多总是提供了与电路交换相同的性能，并且允许在用户数量是其 3 倍时情况也是如此。

电路交换不考虑需求，而预先分配了传输链路的使用，这使得已分配而并不需要



的链路时间未被利用。另一方面，分组交换按需分配链路使用。链路传输能力将在所有需要在链路上传输分组的用户之间逐分组地被共享。

网络的网络

端系统经过一个接入 ISP 与因特网相连，该 ISP 不必是电信局或电缆局，如何解决因特网中数以万计的用户通过接入 ISP 互联是一个很大的问题。而要解决这个难题，接入 ISP 自身必须互联。通过创建网络的网络可以做到这一点。

网络结构 1 用单一的全球传输 ISP 互联所有接入 ISP。假想的全球传输 ISP 是一个由路由器和通信链路构成的网络，该网络不仅跨越全球，而且至少具有一台路由器靠近数十万接入 ISP 中的每一个。

为了有利可图，该全球 ISP 自然要向每一个接入 ISP 收费，故接入 ISP 被认为是客户 (customer)，而全球传输 ISP 被认为是提供商 (provider)。

网络结构 2 由数十万接入 ISP 和多个全球传输 ISP 组成，现在能够根据价格和服务因素在多个竞争的全球传输提供商之间进行选择。

网络结构 2 是一种两层的等级结构，其中全球传输提供商位于顶层，而接入 ISP 位于底层。这假设了全球传输 ISP 不仅能够接近每个接入 ISP，而且发现经济上也希望这样做。但是，世界上是没有哪一个 ISP 是无处不在的。

网络结构 3 在任何给定的区域，可能有一个区域 ISP(regional ISP)，区域中的接入 ISP 与之连接。每个区域 ISP 则与第一层 ISP(tier-1 ISP) 连接。第一层 ISP 类似于我们假想的全球传输 ISP，尽管它不是在世界上每个城市中都存在，但它确实存在。

在这个等级结构的每一层，都有客户-提供商关系。值得注意的是，第一层 ISP 不向任何人付费，因为它们位于该等级结构的顶部。这个多层次等级结构仍然仅仅是今天因特网的粗略近似。

网络结构 4 在等级化网络结构 3 上增加存在点 (Point of Presence, PoP)、多宿、对等和因特网交换点。PoP 存在于等级结构的所有层次，但底层 (接入 ISP) 等级除外。一个 PoP 只是提供商网络中的一台或多台路由器 (在相同位置) 群组，其中客户 ISP 能够与提供商 ISP 连接。对于要与提供商 PoP 连接的客户网络，它能从第三方电信提供商租用高速链路将它的路由器之一直接连接到位于该 PoP 的一台路由器。

客户 ISP 向它们的提供商 ISP 付费以获得全球因特网互联能力。客户 ISP 支付给提供商 ISP 的费用数额反映了它通过提供商交换的通信流量。为了减少这些费用，位于相同等级结构层次的邻近一对 ISP 能够对等 (peer)，也就是说，能够直接将它们的网络连到一起，使它们之间的所有流量经直接连接而不是通过上游的中间 ISP 传输。当两个 ISP 对等时，通常不进行结算，即任一个 ISP 不向其对等付费。



沿着这些相同路线，第三方公司能够创建一个因特网交换点 (Internet Exchange Point, TXP),IXF 是一个汇合点，多个 ISP 能够在这里一起对等。IXP 通常位于一个有自己的交换机的独立建筑物中。

网络结构 5 下图中显示了网络结构 5，它通过在网络结构 4 顶部增加内容提供商网络 (content provider network) 构建而成。

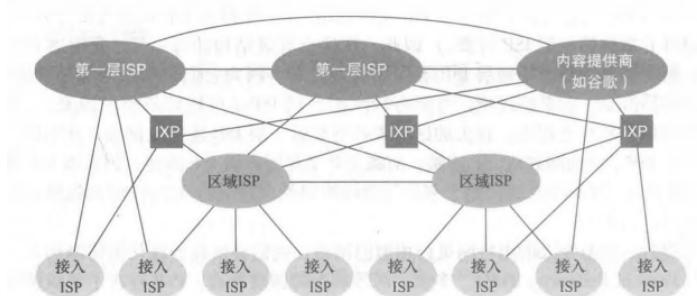


图 1.9: ISP 的互联

较低层的 ISP 与较高层的 ISP 相连，较高层 ISP 彼此互联。用户和内容提供商是较低层 ISP 的客户，较低层 ISP 是较高层 ISP 的客户。近年来，主要的内容提供商也已经创建自己的网络，直接在可能的地方与较低层 ISP 互联。

1.4 分组交换网中的时延、丢包和吞吐量

在理想情况下，我们希望因特网服务能够在任意两个端系统之间随心所欲地瞬间移动数据而没有任何数据丢失。但是，显然是一个极其困难的目标，计算机网络必定要限制在端系统之间的吞吐量 (每秒能够传送的数据量)，在端系统之间引入时延，而且实际上也会丢失分组。

1.4.1 分组交换网中的时延概述

分组从一台主机 (源) 出发，通过一系列路由器传输，在另一台主机 (目的地) 中结束它的历程。该分组在沿途的每个节点经受了几种不同类型的时延。这些时延最为重要的是节点处理时延 (nodal processing delay)、排队时延 (queuing delay)、传输时延 (transmission delay) 和传播时延 (propagation delay)，这些时延总体累加起来是节点总时延 (total nodal delay)。

时延的类型

作为源和目的地之间的端到端路由的一部分，一个分组从上游节点通过路由器 A 向路由器 B 发送。



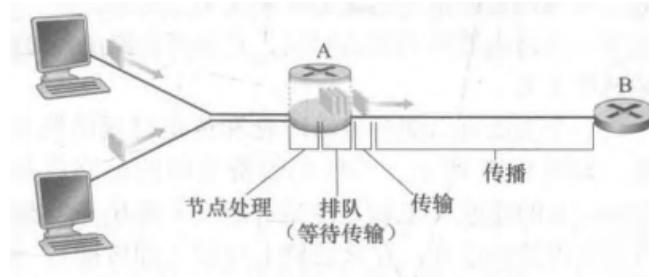


图 1.10: 路由器 A 上的节点时延

1) 处理时延

检查分组首部和决定将该分组导向何处所需要的时间是处理时延的一部分。

2) 排队时延

在队列中，当分组在链路上等待传输时，它经受排队时延。一个特定分组的排队时延长度将取决于先期到达的正在排队等待向链路传输的分组数量。如果该队列是空的，并且当前没有其他分组正在传输，则该分组的排队时延为 0。

3) 传输时延

假定分组以先到先服务方式传输——这在分组交换网中是常见的方式，仅当所有已经到达的分组被传输后，才能传输刚到达的分组。用 L 比特表示该分组的长度，用 R bps(即 b/s) 表示从路由器 A 到路由器 B 的链路传输速率。传输时延是 L/R 。

4) 传播时延

一旦一个比特被推向链路，该比特需要向路由器 B 传播。从该链路的起点到路由器 B 传播所需要的时间是传播时延。该比特以该链路的传播速率传播。该传播时延等于两台路由器之间的距离除以传播速率。即传播时延是其中 d 是路由器 A 和路由器 B 之间的距离， s 是该链路的传播速率

5) 传输时延与传播时延的比较

传输时延是路由器推出分组所需要的时间，它是分组长度和链路传输速率的函数，而与两台路由器之间的距离无关。

传播时延是一个比特从一台路由器传播到另一台路由器所需要的时间，它是两台路由器之间距离的函数，而与分组长度或链路传输速率无关。

我们可以用一个小例子来说明区别，假如有一个十辆车的车队，那么公路上行驶(即传播)，而每隔 100km 就有一个收费站(路由器)，而该车队必须整车队一起出发，因此收费站必须处理完十辆车后，车队开始出发。且无论该车队的第一辆汽车何时到达收费站，它在入口处等待，直到其他 9 辆汽车到达并整队依次前行。收费站将车队处理完之前(整个车队存储在收费站)，因此收费站处理就相当于传输时延，而到达下一个收费站就相当于传播时延。

如果令 d_{proc} , d_{queue} , d_{trans} , d_{prop} 分别表示处理时延、排队时延、传输时延和传播时延，则节点的总时延由下式给定：





图 1.11: 车队的类比

$$d_{total} = d_{proc} + d_{queue} + d_{trans} + d_{prop}$$

处理时延 d_{proc} 通常是微不足道的；然而，它对一台路由器的最大吞吐量有重要影响，最大吞吐量是一台路由器能够转发分组的最大速率。

1.4.2 排队时延和丢包

节点时延的最为复杂和有趣的成分是排队时延 d_{queue} 。与其他三项时延 (d_{proc} , d_{trans} , d_{prop}) 不同的是，排队时延对不同的分组可能是不同的。因此，当表征排队时延时，人们通常使用统计量来度量，如平均排队时延、排队时延的方差和排队时延超过某些特定值的概率。

什么时候排队时延大，什么时候又不大呢？该问题的答案很大程度取决于流量到达该队列的速率、链路的传输速率和到达流量的性质，即流量是周期性到达还是以突发形式到达。

我们假设令 a 表示分组到达队列的平均速率 (a 的单位是分组/秒，即 pkt/s)。前面讲过 R 是传输速率，即从队列中推出比特的速率 (以 bps 即 b/s 为单位)。为了简单起见，也假定所有分组都是由 L 比特组成的。则比特到达队列的平均速率是 $La \text{ bps}$ 。最后，假定该队列非常大，因此它基本能容纳无限数量的比特。比率 La/R 被称为流量强度 (traffic intensity)，它在估计排队时延的范围方面经常起着重要的作用。

如果 $La/R > 1$ ，则比特到达队列的平均速率超过从该队列传输出去的速率。因此：设计系统时流量强度不能大于 1。

如果 $La/R \leq 1$ ，此时到达流量的性质影响排队时延。如果分组周期性到达，即每 L/R 秒到达一个分组，则每个分组将到达一个空队列，不会有排队时延；若以突发形式到达，则可能有很大的平均排队时延。

通常，到达队列的过程是随机的，即到达并不遵循任何模式，分组之间的时间间隔是随机的。无论如何，随着流量强度接近 1，平均排队长度变得越来越长。平均排队时延与流量强度的定性关系如下图所示。

随着流量强度接近于 1，平均排队时延迅速增加。该强度的少量增加将导致时延大比例增加。



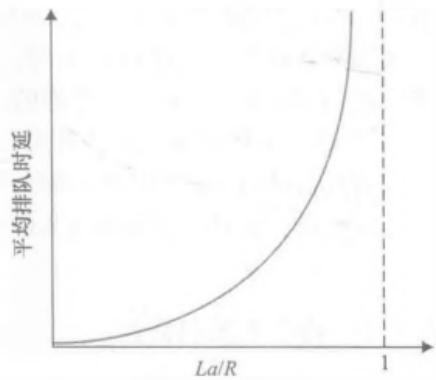


图 1.12: 平均排队时延与流量强度的关系

丢包

在现实中，一条链路前的队列只有有限的容量，尽管排队容量极大地依赖于路由器设计和成本。因为该排队容量是有限的，随着流量强度接近 1，排队时延并不真正趋向无穷大。相反，到达的分组将发现一个满的队列。由于没有地方存储这个分组，路由器将丢弃 (drop) 该分组，即该分组将会丢失 (lost)。

分组丢失的比例随着流量强度增加而增加。因此，一个节点的性能常常不仅根据时延来度量，而且根据丢包的概率来度量。

1.4.3 端到端时延

我们现在考虑从源到目的地的总时延。为了能够理解这个概念，假定在源主机和目的主机之间有 $N-1$ 台路由器。我们还要假设该网络此时是无拥塞的 (因此排队时延是微不足道的)，在每台路由器和源主机上的处理时延是 d_{pw} ，每台路由器和源主机的输出速率是 R bps，每条链路的传播时延是 d_{prop} 。节点时延累加起来，得到端到端时延：

$$d_{end-end} = N(d_{proc} + d_{trans} + d_{prop})$$

1.4.4 计算机网络中的吞吐量

假设我们从主机 A 到主机 B 跨越计算机网络传输一个大文件。在任何时间瞬间的瞬时吞吐量 (instantaneous throughput) 是主机 B 接收到该文件的速率 (以 bps 计)。如果该文件由 F 比特组成，主机 B 接收到所有 F 比特用去 T 秒，则文件传送的平均吞吐量 (average throughput) 是 F/T bps。

首先，我们考虑两个例子。令 R_s 表示服务器与路由器之间的链路速率； R_c 表示路由器与客户之间的链路速率。假定在整个网络中只有从该服务器到客户的比特在传送。



显然，这台服务器不能以快于 R_s bps 的速率通过其链路注入比特；这台路由器也不能以快于 R_c bps 的速率转发比特。如果 $R_s < R_c$ ，则在给定的吞吐量 R_s bps 的情况下，由该服务器注入的比特将顺畅地通过路由器“流动”，并以速率 R_s bps 到达客户。另一方面，如果 $R_c < R_s$ ，则该路由器将不能像接收速率那样快地转发比特。在这种情况下，比特将以速率 R_c 离开该路由器，从而得到端到端吞吐量 R_c 。

因此，对于这种简单的两链路网络，其吞吐量是 $\min\{R_c, R_s\}$ 这就是说，它是瓶颈链路 (bottleneck link) 的传输速率。因此，我们就近似的得到了传输一个 F 比特的大文件所需要的时间是 $F/\min\{R_c, R_s\}$ 。

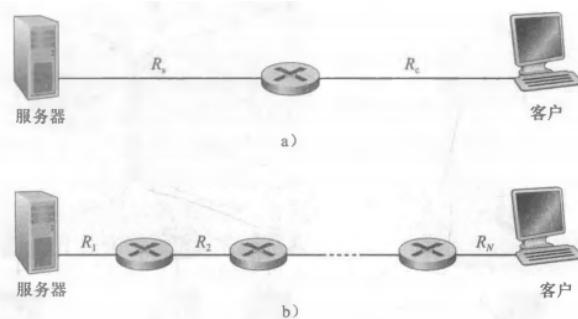


图 1.13: 一个文件从服务器传送到客户的吞吐量

对于一个在服务器和客户之间具有 N 条链路的网络，我们发现从服务器到客户的文件传输吞吐量是 $\min\{R_1, R_2, \dots, R_N\}$ ，这同样仍是沿着服务器和客户之间路径的瓶颈链路的速率。

吞吐量取决于数据流过的链路的传输速率。当没有其他干扰流量时，其吞吐量能够近似为沿着源和目的地之间路径的最小传输速率。同时，吞吐量不仅取决于沿着路径的传输速率，而且取决于干扰流量。特别是，如果许多其他的数据流也通过这条链路流动，一条具有高传输速率的链路仍然可能成为文件传输的瓶颈链路。

1.5 协议层次及其服务模型

1.5.1 分层的体系结构

利用分层的体系结构，我们可以讨论一个大而复杂的系统的定义良好的特定部分。这种简化本身由于提供模块化而具有很高价值，这使某层所提供的服务实现易于改变。只要该层对其上面的层提供相同的服务，并且使用来自下面层次的相同服务，当某层的实现变化时，该系统的其余部分保持不变。

协议分层

为了给网络协议的设计提供一个结构，网络设计者以分层 (layer) 的方式组织协议以及实现这些协议的网络硬件和软件。



我们关注某层向它的上一层提供的服务 (service)，即所谓一层的服务模型 (service model)。

一个协议层能够用软件、硬件或者两者结合的方式来实现。对于 HTTP 和 SMTP 这样的应用层协议总是在端系统中用软件实现，而物理层和数据链路层负责处理跨越特定链路的通信，因此在与给定链路相关联的网络接口卡中实现。网络层经常是硬件和软件实现的混合体。需要注意的是，一个第 n 层协议也分布在构成该网络的端系统、分组交换机和其他组件。

协议分层具有概念化和结构化的优点，这使得更新组件更为容易，但是，分层的一个潜在缺点是一层可能冗余较低层的功能。第二个潜在缺点是某层的功能可能需要仅在其他层才出现的信息，这又违反了层次分离的目标。

各层的所有协议被称为协议栈 (protocol stack)。因特网的协议栈由五个层次组成：物理层、链路层、网络层、运输层和应用层。



图 1.14: 因特网协议栈和 OSI 参考模型

应用层 应用层是网络应用程序及它们的应用层协议存留的地方。

应用层协议分布在多个端系统上，而一个端系统中的应用程序使用协议与另一个端系统中的应用程序交换信息分组。我们把这种位于应用层的信息分组称为报文 (message)。

运输层 因特网的运输层在应用程序端点之间传送应用层报文。

在因特网中，有两种运输协议，即 TCP 和 UDP，利用其中的任一个都能运输应用层报文。在本书中，我们把运输层的分组称为报文段。 (segment)。

网络层 因特网的网络层负责将称为数据报 (datagram) 的网络层分组从一台主机移动到另一台主机。在一台源主机中的因特网运输层协议 (TCP 或 UDP) 向网络层递交运



输层报文段和目的地址，就像你通过邮政服务寄信件时提供一个目的地址一样。

因特网的网络层包括著名的网际协议 IP，该协议定义了在数据报中的各个字段以及端系统和路由器如何作用于这些字段。IP 仅有一个，所有具有网络层的因特网组件必须运行 IP。因特网的网络层也包括决定路由的路由选择协议，它根据该路由将数据报从源传输到目的地。因特网具有许多路由选择协议。

尽管网络层包括了网际协议和一些路由选择协议，但通常把它简单地称为 IP 层，这反映了 IP 是将因特网连接在一起的黏合剂这样的事实。

链路层 因特网的网络层通过源和目的地之间的一系列路由器路由数据报。为了将分组从一个节点（主机或路由器）移动到路径上的下一个节点，网络层必须依靠该链路层的服务。

由链路层提供的服务取决于应用于该链路的特定链路层协议。因为数据报从源到目的地传送通常需要经过几条链路，一个数据报可能被沿途不同链路上的不同链路层协议处理。网络层将受到来自每个不同的链路层协议的不同服务。在本书中，我们把链路层分组称为帧（frame）。

物理层 链路层的任务是将整个帧从一个网络元素移动到邻近的网络元素，而物理层的任务是将该帧中的一个个比特从一个节点移动到下一个节点。在这层中的协议仍然是链路相关的，并且进一步与该链路（例如，双绞铜线、单模光纤）的实际传输媒体相关。

OSI 模型

因特网协议栈不是唯一的协议栈，是在 20 世纪 70 年代后期，国际标准化组织（ISO）提出计算机网络围绕 7 层来组织，称为开放系统互连（OSI）模型。

OSI 参考模型的 7 层是：应用层、表示层、会话层、运输层、网络层、数据链路层和物理层。这些层次中，5 层的功能大致与它们名字类似的因特网对应层的功能相同。所以，我们来考虑 OSI 参考模型中附加的两个层，即表示层和会话层。表示层的作用是使通信的应用程序能够解释交换数据的含义。这些服务包括数据压缩和数据加密（它们是自解释的）以及数据描述（这使得应用程序不必担心在各台计算机中表示/存储的内部格式不同的问题）。会话层提供了数据交换的定界和同步功能，包括了建立检查点和恢复方案的方法。

1.5.2 封装

下图显示了这样一条物理路径：数据从发送端系统的协议栈向下，沿着中间的链路层交换机和路由器的协议栈上上下下，然后向上到达接收端系统的协议栈。路由器和链路层交换机都是分组交换机。与端系统类似，路由器和链路层交换机以多层次的



方式组织它们的网络硬件和软件。而路由器和链路层交换机并不实现协议栈中的所有层次。

图中也说明了一个重要概念：封装（encapsulation）。在发送主机端，一个应用层报文（application-layer message）（图中的 M ）被传送给运输层。在最简单的情况下，运输层收取到报文并附上附加信息（所谓运输层首部信息，图中的 H_t ，该首部将被接收端的运输层使用。应用层报文和运输层首部信息一道构成了运输层报文段（transport layer segment）。

运输层报文段因此封装了应用层报文。附加的信息也许包括了下列信息：允许接收端运输层向上向适当的应用程序交付报文的信息；差错检测位信息，该信息让接收方能够判断报文中的比特是否在途中已被改变。

运输层则向网络层传递该报文段，网络层增加了如源和目的端系统地址等网络层首部信息（图中的 H_n ，生成了网络层数据报（network-layer datagram）。该数据报接下来被传递给链路层，链路层（自然而然地）增加它自己的链路层首部信息并生成链路层帧（link layer frame）。

所以我们看到，在每一层，一个分组具有两种类型的字段：首部字段和有效载荷字段（payload field）。有效载荷通常是来自上一层的分组。

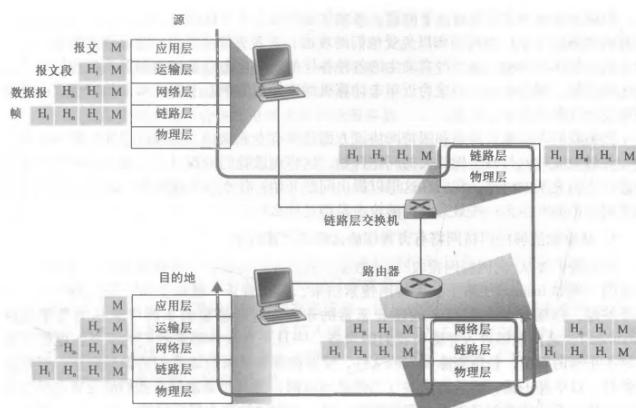


图 1.15: 主机、路由器和链路交换机

1.6 面对攻击的网络

1.6.1 通过因特网放入病毒

因为我们要从/向因特网接收/发送数据，所以我们将设备与因特网相连。不幸的是，伴随好的东西而来的还有恶意的东西，这些恶意的东西可统称为恶意软件（malware），它们能够进入并感染我们的设备。我们的受害主机也可能成为数以千计的类似受害设备网络中的一员，它们被统称为僵尸网络 (botnet)。



至今为止的多数恶意软件是自我复制 (self-replicating) 的：一旦它感染了一台主机，就会从那台主机寻求进入因特网上的其他主机，从而形成新的感染主机，再寻求进入更多的主机。

病毒 (virus) 是一种需要某种形式的用户交互来感染用户设备的恶意软件。

蠕虫 (worm) 是一种无须任何明显用户交互就能进入设备的恶意软件。

1.6.2 攻击服务器和网络基础设施

另一种宽泛类型的安全性威胁称为拒绝服务攻击 (Denial-of Service (DoS) attack)。顾名思义，DoS 攻击使得网络、主机或其他基础设施部分不能由合法用户使用。

大多数因特网 DoS 攻击属于下列三种类型之一：

- 1) 弱点攻击。这涉及向一台目标主机上运行的易受攻击的应用程序或操作系统发送制作精细的报文。如果适当顺序的多个分组发送给一个易受攻击的应用程序或操作系统，该服务器可能停止运行，或者更糟糕的是主机可能崩溃。
- 2) 带宽洪泛。攻击者向目标主机发送大量的分组，分组数量之多使得目标的接入链路变得拥塞，使得合法的分组无法到达服务器。
- 3) 连接洪泛。攻击者在目标主机中创建大量的半开或全开 TCP 连接。该主机因这些伪造的连接而陷入困境，并停止接受合法的连接。

下图中显示的分布式 DoS (Distributed DoS, DDoS) 中，攻击者控制多个源并让每个源向目标猛烈发送流量。使用这种方法，遍及所有受控源的聚合流量速率需要大约 R 的能力来使该服务陷入瘫痪。DDoS 攻击充分利用由数以千计的受害主机组成的僵尸网络，这在今天是屡见不鲜的。

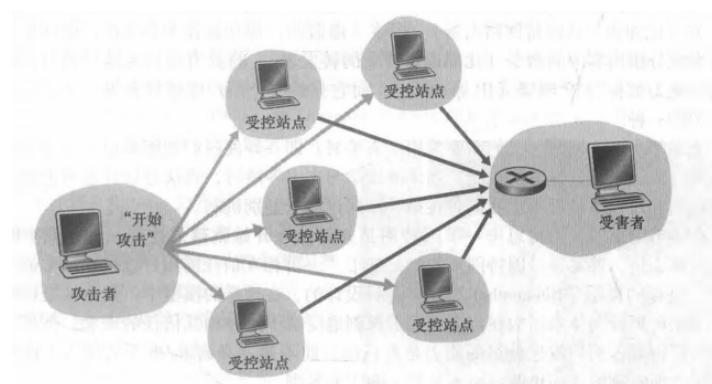


图 1.16: 分布式拒绝服务攻击

1.6.3 嗅探分组

无所不在的因特网接入极为便利并让移动用户方便地使用令人惊奇的新应用程序的同时，也产生了严重的安全脆弱性——在无线传输设备的附近放置一台被动的接收机，该接收机就能得到传输的每个分组的副本！这些分组包含了各种敏感信息，包



括口令、社会保险号、商业秘密和隐秘的个人信息。记录每个流经的分组副本的被动接收机被称为分组嗅探器 (packet sniffer)。

嗅探器也能够部署在有线环境中。在有线的广播环境中，如在许多以太网 LAN 中，分组嗅探器能够获得经该 LAN 发送的所有分组。

因为分组嗅探器是被动的，也就是说它们不向信道中注入分组，所以难以检测到它们。因此，当我们向无线信道发送分组时，我们必须接受这样的可能性，即某些坏家伙可能记录了我们的分组的副本。

1.6.4 伪装

生成具有任意源地址、分组内容和目的地址的分组，然后将这个人工制作的分组传输到因特网中，因特网将忠实地将该分组转发到目的地，这一切都极为容易。将具有虚假源地址的分组注入因特网的能力被称为 IP 哄骗 (IP spoofing)，而它只是一个用户能够冒充另一个用户的许多方式中的一种。

为了解决这个问题，我们需要采用端点鉴别，即一种使我们能够确信一个报文源自我们认为它应当来自的地方的机制。



第 2 章 应用层



网络应用是计算机网络存在的理由

2.1 应用层协议原理

研发网络应用程序的核心是写出能够运行在不同的端系统和通过网络彼此通信的程序。

当研发新应用程序时，你需要编写将在多台端系统上运行的软件。重要的是，你不需要写在网络核心设备如路由器或链路层交换机上运行的软件。即使你要为网络核心设备写应用程序软件，你也不能做到这一点。网络核心设备并不在应用层上起作用，而仅在较低层起作用，特别是在网络层及下面层次起作用。这种基本设计，即将应用软件限制在端系统的方法，促进了大量的网络应用程序的迅速研发和部署。

2.1.1 网络应用程序体系结构

当进行软件编码之前，应当对应用程序有一个宽泛的体系结构计划。从应用程序研发者的角度看，网络体系结构是固定的，并为应用程序提供了特定的服务集合。在另一方面，应用程序体系结构 (application architecture) 由应用程序研发者设计，规定了如何在各种端系统上组织该应用程序。

客户-服务器体系结构

在客户-服务器体系结构 (client-server architecture) 中，有一个总是打开的主机称为服务器，它服务于来自许多其他称为客户的主机的请求。值得注意的是利用客户-服务器体系结构，客户相互之间不直接通信。

客户-服务器体系结构的另一个特征是该服务器具有固定的、周知的地址，该地址称为 IP 地址。因为该服务器具有固定的、周知的地址，并且因为该服务器总是打开的，客户总是能够通过向该服务器的 IP 地址发送分组来与其联系。

在一个客户-服务器应用中，常常会出现一台单独的服务器主机跟不上它所有客户请求的情况。为此，配备大量主机的数据中心 (data center) 常被用于创建强大的虚拟服务器。

P2P 体系结构

在一个 P2P 体系结构 (P2P architecture) 中，对位于数据中心的专用服务器有最小的(或者没有)依赖。相反，应用程序在间断连接的主机对之间使用直接通信，这些主机对被称为对等方。因为这种对等方通信不必通过专门的服务器，该体系结构被称为对等方到对等方的。

许多目前流行的、流量密集型应用都是 P2P 体系结构的。需要提及的是，某些应用具有混合的体系结构，它结合了客户-服务器和 P2P 的元素。

P2P 体系结构的最引人入胜的特性之一是它们的自扩展性 (self-scalability)。例如，在一个 P2P 文件共享应用中，尽管每个对等方都由于请求文件产生工作负载，但每个对等方通过向其他对等方分发文件也为系统增加服务能力。

2.1.2 进程通信

用操作系统的术语来说，进行通信的实际上是进程 (process) 而不是程序。一个进程可以被认为是运行在端系统中的一个程序。当多个进程运行在相同的端系统上时，它们使用进程间通信机制相互通信。进程间通信的规则由端系统上的操作系统确定。

此处，我们并不特别关注同一台主机上的进程间的通信，而关注运行在不同端系统(可能具有不同的操作系统)上的进程间的通信。

在两个不同端系统上的进程，通过跨越计算机网络交换报文 (message) 而相互通信。发送进程生成并向网络中发送报文；接收进程接收这些报文并可能通过回送报文进行响应。

客户和服务器进程

网络应用程序由成对的进程组成，这些进程通过网络相互发送报文。对每对通信进程，我们通常将这两个进程之一标识为客户 (client)，而另一个进程标识为服务器 (serve)。

在一对进程之间的通信会话场景中，发起通信(即在该会话开始时发起与其他进程的联系)的进程被标识为客户，在会话开始时等待联系的进程是服务器。

进程与计算机网络之间的接口

多数应用程序是由通信进程对组成，每对中的两个进程互相发送报文。进程通过一个称为套接字 (socket) 的软件接口向网络发送报文和从网络接收报文。

下图显示了两个经过因特网通信的进程之间的套接字通信(图中假定由该进程使用的下面运输层协议是因特网的 TCP 协议)。如该图所示，套接字是同一台主机内应用层与运输层之间的接口。由于该套接字是建立网络应用程序的可编程接口，因此套接字也称为应用程序和网络之间的应用程序编程接口 (Application Programming Interface,



API)。应用程序开发者可以控制套接字在应用层端的一切，但是对该套接字的运输层端几乎没有控制权。

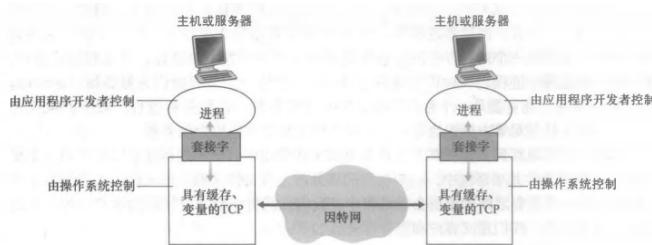


图 2.1: 应用程序间的套接字通信

应用程序开发者对于运输层的控制仅限于：①选择运输层协议；②也许能设定几个运输层参数，如最大缓存和最大报文段长度等。

进程寻址

在因特网中，主机由其 IP 地址 (IP address) 标识。此时，我们只要知道 IP 地址是一个 32 比特的量且它能够唯一地标识该主机就够了。除了知道报文发送目的地的主机地址外，发送进程还必须指定运行在接收主机上的接收进程 (更具体地说，接收套接字)。因为一般而言一台主机能够运行许多网络应用，这些信息是需要的。目的地端口号 (port number) 用于这个目的。

2.1.3 可供应用程序使用的运输服务

包括因特网在内的很多网络提供了不止一种运输层协议。当开发一个应用时，必须选择一种可用的运输层协议。如何做出这种选择呢？最可能的方式是，通过研究这些可用的运输层协议所提供的服务，选择一个最能为你的应用需求提供恰当服务的协议。

我们大体能够从四个方面对应用程序服务要求进行分类：可靠数据传输、吞吐量、定时和安全性。

可靠数据传输

分组在计算机网络中可能丢失。在一些应用中，数据丢失可能会造成灾难性的后果。

因此，为了支持这些应用，必须做一些工作以确保由应用程序的一端发送的数据正确、完全地交付给该应用程序的另一端。如果一个协议提供了这样的确保数据交付服务，就认为提供了可靠数据传输 (reliable data transfer)。

运输层协议能够潜在地向应用程序提供的一个重要服务是进程到进程的可靠数据传输。当一个运输协议提供这种服务时，发送进程只要将其数据传递进套接字，就可以完全相信该数据将能无差错地到达接收进程。



吞吐量

在沿着一条网络路径上的两个进程之间的通信会话场景中，可用吞吐量就是发送进程能够向接收进程交付比特的速率。

运输层协议能够以某种特定的速率提供确保的可用吞吐量。使用这种服务，该应用程序能够请求 r 比特/秒的确保吞吐量，并且该运输协议能够确保可用吞吐量总是为至少 r 比特/秒。这样的确保吞吐量的服务将对许多应用程序有吸引力。

具有吞吐量要求的应用程序被称为带宽敏感的应用 (bandwidth-sensitive application)。带宽敏感的应用具有特定的吞吐量要求，而弹性应用 (elastic application) 能够根据当时可用的带宽或多或少地利用可供使用的吞吐量。

定时

运输层协议也能提供定时保证。如同具有吞吐量保证那样，定时保证能够以多种形式实现。一个保证的例子如：发送方注入进套接字中的每个比特到达接收方的套接字不迟于 100ms。

安全性

运输协议能够为应用程序提供一种或多种安全性服务。这种服务将在发送和接收进程之间提供机密性，以防该数据以某种方式在这两个进程之间被观察到。运输协议还能提供除了机密性以外的其他安全性服务，包括数据完整性和端点鉴别。

2.1.4 因特网提供的运输服务

我们已经考虑了计算机网络能够提供的通用运输服务。现在我们要更为具体地考察由因特网提供的运输服务类型。因特网 (更一般的是 TCP/IP 网络) 为应用程序提供两个运输层协议，即 UDP 和 TCP。

TCP 服务

TCP 服务模型包括面向连接服务和可靠数据传输服务。当某个应用程序调用 TCP 作为其运输协议时，该应用程序就能获得来自 TCP 的这两种服务。

1) 面向连接的服务

在应用层数据报文开始流动之前，TCP 让客户和服务器互相交换运输层控制信息。这个所谓的握手过程提醒客户和服务器，让它们为大量分组的到来做好准备。在握手阶段后，一个 TCP 连接 (TCP connection) 就在两个进程的套接字之间建立了。这条连接是全双工的，即连接双方的进程可以在此连接上同时进行报文收发。当应用程序结束报文发送时，必须拆除该连接。



表 2.1: 选择的网络应用的要求

应用	数据丢失	带宽	时间敏感
文件传输	不能丢失	弹性	不
电子邮件	不能丢失	弹性	不
Web 文档	不能丢失	弹性 (几 kbps)	不
因特网电话/视频会议	容忍丢失	音频 (几 kbps 1 Mbps)	是, 100ms
		视频 (10 kbps 5 Mbps)	
流式存储音频/视频	容忍丢失	同上	是, 几秒
交互式游戏	容忍丢失	几 kbps 10 kbps	是, 100ms
只能手机讯息	不能丢失	弹性	是和不是

2) 可靠的数据传送服务

通信进程能够依靠 TCP, 无差错、按适当顺序交付所有发送的数据。当应用程序的一端将字节流传进套接字时, 它能够依靠 TCP 将相同的字节流交付给接收方的套接字, 而没有字节的丢失和冗余。

3) 拥塞控制机制

这种服务不一定能为通信进程带来直接好处, 但能为因特网带来整体好处。当发送方和接收方之间的网络出现拥塞时, TCP 的拥塞控制机制会抑制发送进程 (客户或服务器)。

UDP 服务

UDP 是一种不提供不必要的轻量级运输协议, 它仅提供最小服务。UDP 是无连接的, 因此在两个进程通信前没有握手过程。UDP 协议提供一种不可靠数据传送服务, 也就是说, 当进程将一个报文发送进 UDP 套接字时, UDP 协议并不保证该报文将到达接收进程。不仅如此, 到达接收进程的报文也可能是乱序到达的。

UDP 没有包括拥塞控制机制, 所以 UDP 的发送端可以用它选定的任何速率向其下层 (网络层) 注入数据。(然而, 值得注意的是实际端到端吞吐量可能小于该速率, 这可能是因为中间链路的带宽受限或因为拥塞而造成的。)

因特网运输协议所不提供的服务

在我们对 TCP 和 UDP 的简要描述中, 明显地漏掉了对吞吐量或定时保证的讨论, 即这些服务目前的因特网运输协议并没有提供。

下表指出了一些流行的因特网应用所使用的运输协议。



表 2.2: 流行的因特网应用及其应用层协议与支撑的运输协议

应用	应用层协议	支撑的运输协议
电子邮件	SMTP[RFC 5321]	TCP
远程终端访问	Telnet[RFC 854]	TCP
Web	HTTP[RFC 2616]	TCP
文件传输	FTP[RFC 959]	TCP
流式多媒体	HTTP(如 YouTube)	TCP
因特网电话	SIP[RFC 3261]、RTP[RFC 3550] 或专用的(如 Skype)	UDP 或 TCP

2.1.5 应用层协议

应用层协议 (application-layer protocol) 定义了运行在不同端系统上的应用程序进程如何相互传递报文。特别是应用层协议定义了：

- 1) 交换的报文类型，例如请求报文和响应报文
- 2) 各种报文类型的语法，如报文中的各个字段及这些字段是如何描述的
- 3) 字段的语义，即这些字段中的信息的含义
- 4) 确定一个进程何时以及如何发送报文，对报文进行响应的规则

区分网络应用和应用层协议是很重要的。应用层协议只是网络应用的一部分(尽管从我们的角度看，它是应用非常重要的一部分)。

2.2 Web 和 HTTP

Web 是一个引起公众注意的因特网应用，它极大地改变了人们与工作环境内外交流的方式。它将因特网从只是很多数据网之一的地位提升为仅有的一个数据网。

2.2.1 HTTP 概况

Web 的应用层协议是超文本传输协议 (HyperText Transfer Protocol, HTTP)，它是 Web 的核心。HTTP 由两个程序实现：一个客户程序和一个服务器程序。客户程序和服务器程序运行在不同的端系统中，通过交换 HTTP 报文进行会话。HTTP 定义了这些报文的结构以及客户和服务器进行报文交换的方式。

Web 页面 (Webpage)(也叫文档) 是由对象组成的。一个对象 (object) 只是一个文件，诸如一个 HTML 文件、一个 JPEG 图形、一个 Java 小程序或一个视频片段这样的文件，且它们可通过一个 URL 地址寻址。多数 Web 页面含有一个 HTML 基本文件 (base HTML file) 以及几个引用对象。

HTML 基本文件通过对对象的 URL 地址引用页面中的其他对象。每个 URL 地址由两部分组成：存放对象的服务器主机名和对象的路径名。Web 服务器 (Webserver) 实



现了 HTTP 的服务器端，它用于存储 Web 对象，每个对象由 URL 寻址。

HTTP 定义了 Web 客户向 Web 服务器请求 Web 页面的方式，以及服务器向客户传送 Web 页面的方式。如下图所示，客户与服务器交互的过程。



图 2.2: HTTP 请求-响应行为

HTTP 使用 TCP 作为它的支撑运输协议 (而不是在 UDP 上运行)。HTTP 客户首先发起一个与服务器的 TCP 连接。一旦连接建立，该浏览器和服务器进程就可以通过套接字接口访问 TCP。

客户向它的套接字接口发送 HTTP 请求报文并从它的套接字接口接收 HTTP 响应报文。类似地，服务器从它的套接字接口接收 HTTP 请求报文并向它的套接字接口发送 HTTP 响应报文。一旦客户向它的套接字接口发送了一个请求报文，该报文就脱离了客户控制并进入 TCP 的控制。

这里我们看到了分层体系结构最大的优点，即 *HTTP* 协议不用担心数据丢失，也不关注 *TCP* 从网络的数据丢失和乱序故障中恢复的细节。那是 *TCP* 以及协议栈较低层协议的工作。

服务器向客户发送被请求的文件，而不存储任何关于该客户的状态信息。假如某个特定的客户在短短的几秒内两次请求同一个对象，服务器并不会因为刚刚为该客户提供了该对象就不再做出反应，而是重新发送该对象，就像服务器已经完全忘记不久之前所做过的事一样。因为 *HTTP* 服务器并不保存关于客户的任何信息，所以我们说 *HTTP* 是一个无状态协议 (*stateless protocol*)。

2.2.2 非持续连接和持续连接

在许多因特网应用程序中，客户和服务器在一个相当长的时间范围内通信，其中客户发出一系列请求并且服务器对每个请求进行响应。依据应用程序以及该应用程序的使用方式，这一系列请求可以以规则的间隔周期性地或者间断性地一个接一个发出。当这种客户-服务器的交互是经 *TCP* 进行的。



应用程序的研制者就需要做一个重要决定，即每个请求/响应对是经一个单独的 TCP 连接发送，还是所有的请求及其响应经相同的 TCP 连接发送呢？采用前一种方法，该应用程序被称为使用非持续连接 (non-persistent connection)；采用后一种方法，该应用程序被称为使用持续连接 (persistent connection)。

采用非持续连接的 HTTP

我们看看在非持续连接情况下，从服务器向客户传送一个 Web 页面的步骤。假设该页面含有一个 HTML 基本文件和 10 个 JPEG 图形，并且这 11 个对象位于同一台服务器上。进一步假设该 HTML 文件的 URL 为：http：“www. someSchool.edu/someDepartment/home, indexo 我们看看发生了什么情况：

- 1) HTTP 客户进程在端口号 80 发起一个到服务器 www.someSchool.edu 的 TCP 连接，该端口号是 HTTP 的默认端口。在客户和服务器上分别有一个套接字与该连接相关联。
- 2) HTTP 客户经它的套接字向该服务器发送一个 HTTP 请求报文。请求报文中包含了路径名/someDepartment/home.index（后面我们会详细讨论 HTTP 报文）。
- 3) HTTP 服务器进程经它的套接字接收该请求报文，从其存储器（RAM 或磁盘）中检索出对象 www.someSchool.edu/someDepartment/home.index，在一个 HTTP 响应报文中封装对象，并通过其套接字向客户发送响应报文。
- 4) HTTP 服务器进程通知 TCP 断开该 TCP 连接。（但是直到 TCP 确认客户已经完整地收到响应报文为止，它才会实际中断连接。）
- 5) HTTP 客户接收响应报文，TCP 连接关闭。该报文指出封装的对象是一个 HTML 文件，客户从响应报文中提取出该文件，检查该 HTML 文件，得到对 10 个 JPEG 图形的引用
- 6) 对每个引用的 JPEG 图形对象重复前 4 个步骤

上面的步骤举例说明了非持续连接的使用，其中每个 TCP 连接在服务器发送一个对象后关闭，即该连接并不为其他的对象而持续下来。值得注意的是每个 TCP 连接只传输一个请求报文和一个响应报文。因此在本例中，当用户请求该 Web 页面时，要产生 11 个 TCP 连接。

我们给出往返时间 (Round Trip Time, RTF) 的定义，该时间是指一个短分组从客户到服务器然后再返回客户所花费的时间。RTT 包括分组传播时延、分组在中间路由器和交换机上的排队时延以及分组处理时延。

采用持续连接的 HTTP

非持续连接有一些缺点：

第一，必须为每一个请求的对象建立和维护一个全新的连接。对于每个这样的连接，在客户和服务器中都要分配 TCP 的缓冲区和保持 TCP 变量，这给 Web 服务器带



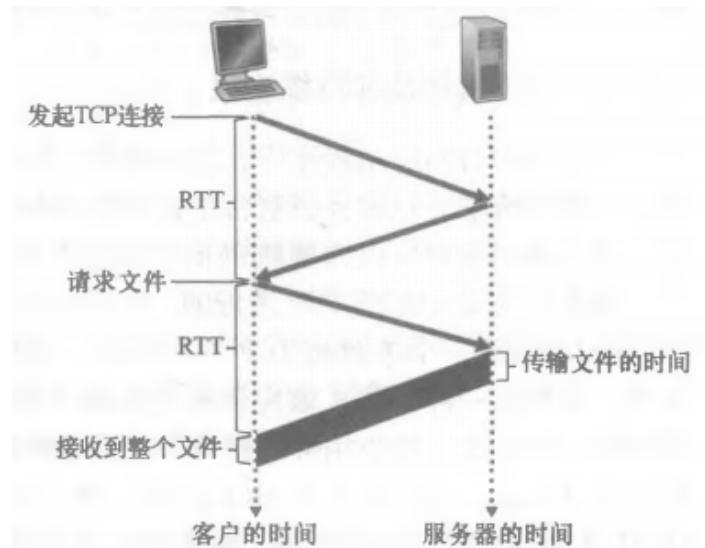


图 2.3: 请求并接收一个 HTML 文件所需的时间估算

来了严重的负担，因为一台 Web 服务器可能同时服务于数以百计不同的客户的请求。

第二，就像我们刚描述的那样，每一个对象经受两倍 RTT 的交付时延，即一个 RTT 用于创建 TCP，另一个 RTT 用于请求和接收一个对象。

在采用 HTTP 1.1 持续连接的情况下，服务器在发送响应后保持该 TCP 连接打开。在相同的客户与服务器之间，后续的请求和响应报文能够通过相同的连接进行传送。一般来说，如果一条连接经过一定时间间隔（一个可配置的超时间隔）仍未被使用，HTTP 服务器就关闭该连接。HTTP 的默认模式是使用带流水线的持续连接。

2.2.3 HTTP 报文格式

HTTP 规范 [RFC 1945; RFC 2616; RFC 7540] 包含了对 HTTP 报文格式的定义。HTTP 报文有两种：请求报文和响应报文。

HTTP 请求报文

```

1 GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
3 Connection: close
User-agent: Mozilla/5.0
5 Accept-language: fr

```

观察这个简单的请求报文，其中有五行信息，每一行的结尾都有一个结束符号（操作系统不同所对应的结束符号不同，在这里以 '\r\n' 为准）。



HTTP 请求报文的第一行叫做请求行 (request line), 其后继的行叫做首部行 (header line)。请求行有 3 个字段: 方法字段、URL 字段和 HTTP 版本字段。

方法字段可以取几种不同的值, 包括 GET、POST、HEAD、PUT 和 DELETE。绝大部分的 HTTP 请求报文使用 GET 方法。当浏览器请求一个对象时, 使用 GET 方法, 在 URL 字段带有请求对象的标识。

现在我们看看本例的首部行。首部行 Host: www.someschool.edu 指明了对象所在的主机。该首部行提供的信息是 Web 代理高速缓存所要求的。

通过包含 Connection: close 首部行, 该浏览器告诉服务器不要麻烦地使用持续连接, 它要求服务器在发送完被请求的对象后就关闭这条连接。

User-agent: 首部行用来指明用户代理, 即向服务器发送请求的浏览器的类型。这里浏览器类型是 Mozilla/5.0, 即 Firefox 浏览器。这个首部行是有用的, 因为服务器可以有效地为不同类型的用户代理实际发送相同对象的不同版本。(每个版本都由相同的 URL 寻址。)

最后, Accept-language: 首部行表示用户想得到该对象的法语版本(如果服务器中有这样的对象的话); 否则, 服务器应当发送它的默认版本。Accept-language: 首部行仅是 HTTP 中可用的众多内容协商首部之一。

接下来我们来看看请求报文的通用格式:

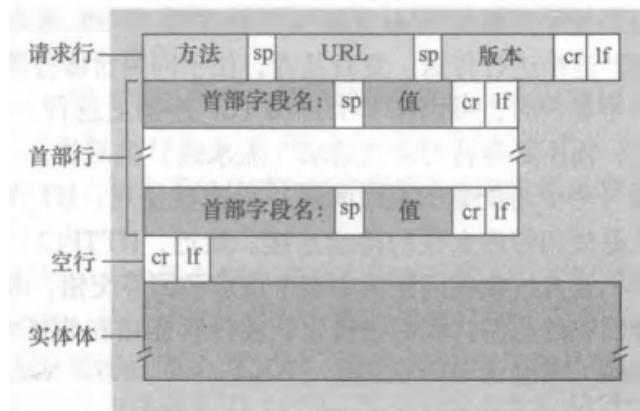


图 2.4: HTTP 请求报文的通用格式

我们发现, 在空行后有一个“实体体”(entity body)。使用 GET 方法时实体体为空, 而使用 POST 方法时才使用该实体体。当用户提交表单时, HTTP 客户常常使用 POST 方法。如果方法字段的值为 POST 时, 则实体体中包含的就是用户在表单字段中的输入值。

当然, 值得注意的是, HTML 表单经常使用 GET 方法, 并在(表单字段中)所请求的 URL 中包括输入的数据。例如, 一个表单使用 GET 方法, 它有两个字段, 分别填写的是“monkeys”和“bananas”, 这样, 该 URL 结构为 www.somesite.com/animalsearch?monkeys&bananas。

HEAD 方法类似于 GET 方法。当服务器收到一个使用 HEAD 方法的请求时, 将会用一个 HTTP 报文进行响应, 但是并不返回请求对象。应用程序开发者常用 HEAD



方法进行调试跟踪。PUT 方法常与 Web 发行工具联合使用，它允许用户上传对象到指定的 Web 服务器上指定的路径(目录)。PUT 方法也被那些需要向 Web 服务器上传对象的应用程序使用。DELETE 方法允许用户或者应用程序删除 Web 服务器上的对象。

HTTP 响应报文

下面我们提供了一条典型的 HTTP 响应报文。

```
1 HTTP/1.1 200 OK
Connection: close
3 Date: Tue, 18 Aug 2015 15:44:04 GMT
Server: Apache/2.2.3 (CentOS)
5 Last-Modified: Tue, 18 Aug 2015 15:11:03 GMT
Content-Length: 6821
7 Content-Type: text/html
(data data data data data ...)
```

我们仔细看一下这个响应报文。它有三个部分：一个初始状态行(status line), 6个首部行(headerline), 然后是实体体(entity body)。实体体部分是报文的主要部分，即它包含了所请求的对象本身(表示为 data data data data data...)。状态行有3个字段：协议版本字段、状态码和相应状态信息。

我们现在来看看首部行。

服务器用 Connection: close 首部行告诉客户，发送完报文后将关闭该 TCP 连接。

Date: 首部行指示服务器产生并发送该响应报文的日期和时间。值得一提的是，这个时间不是指对象创建或者最后修改的时间，而是服务器从它的文件系统中检索到该对象，将该对象插入响应报文，并发送该响应报文的时间。

Server: 首部行指示该报文是由一台 ApacheWeb 服务器产生的，它类似于 HTTP 请求报文中的 User-agent: 首部行。

Last-Modified: 首部行指示了对象创建或者最后修改的日期和时间。Last-Modified: 首部行对既可能在本地客户也可能在网络缓存服务器上的对象缓存来说非常重要。我们将很快详细地讨论缓存服务器(也叫代理服务器)。

Content Length: 首部行指示了被发送对象中的字节数。

Content-Type: 首部行指示了实体体中的对象是 HTML 文本。(该对象类型应该正式地由 Content-Type: 首部行而不是用文件扩展名来指示。)

接下来我们看一下响应报文的通用格式

我们补充说明一下状态码和它们对应的短语。状态码及其相应的短语指示了请求的结果。一些常见的状态码和相关的短语包括：

- 1) 200 OK: 请求成功，信息在返回的响应报文中



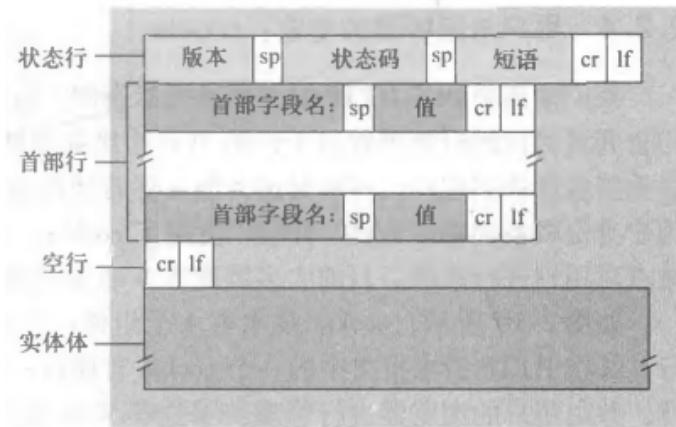


图 2.5: HTTP 响应报文的通用格式

- 2) 301 Moved Permanently: 请求的对象已经被永久转移了，新的 URL 定义在响应报文的 Location: 首部行中。客户软件将自动获取新的 URL
- 3) 400 Bad Request: 一个通用差错代码，指示该请求不能被服务器理解
- 4) 404 Not Found: 被请求的文档不在服务器上
- 5) 505 HTTP Version Not Supported: 服务器不支持请求报文使用的 HTTP 协议版本

2.2.4 用户与服务器的交互: cookie

我们前面提到了 HTTP 服务器是无状态的。这简化了服务器的设计，并且允许工程师们去开发可以同时处理数以千计的 TCP 连接的高性能 Web 服务器。

然而一个 Web 站点通常希望能够识别用户，可能是因为服务器希望限制用户的访问，或者因为它希望把内容与用户身份联系起来。为此，HTTP 使用了 cookie。cookie 在 [RFC 6265] 中定义，它允许站点对用户进行跟踪。目前大多数商务 Web 站点都使用了 cookie。

cookie 技术有 4 个组件：①在 HTTP 响应报文中的一个 cookie 首部行；②在 HTTP 请求报文中的一个 cookie 首部行；③在用户端系统中保留有一个 cookie 文件，并由用户的浏览器进行管理；④位于 Web 站点的一个后端数据库。

假设我们在以前访问过一个网站站点，那么在第一次建立连接的时候，该 Web 站点会产生一个唯一的 cookie 标识码以此来作为索引在后端数据库中产生一个表项，在后续访问该网站时，首部的响应报文就会包含：

Set-cookie: 1678

因此，当浏览器收到了该 HTTP 响应报文，就会将之前用户所注册的信息活动反馈过来。



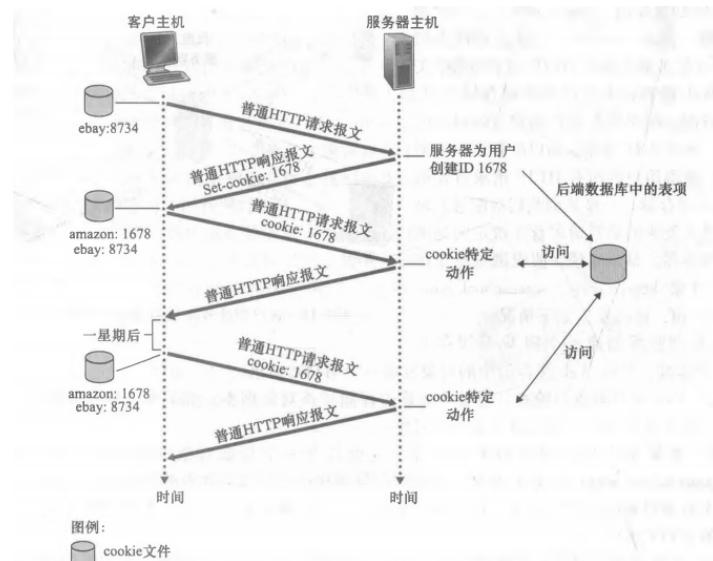


图 2.6: 用 cookie 跟踪用户状态

2.2.5 Web 缓存

Web 缓存器 (Web cache) 也叫代理服务器 (proxy server), 它是能够代表初始 Web 服务器来满足 HTTP 请求的网络实体。Web 缓存器有自己的磁盘存储空间, 并在存储空间中保存最近请求过的对象的副本。

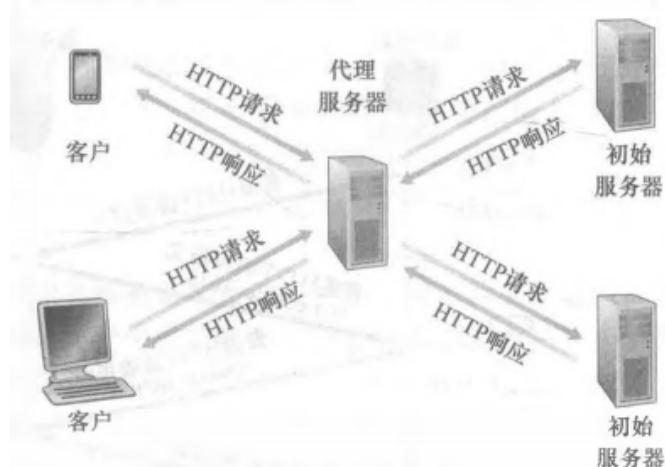


图 2.7: 客户通过 Web 缓存器请求对象

可以配置用户的浏览器, 使得用户的所有 HTTP 请求首先指向 Web 缓存器。一旦某浏览器被配置, 每个对某对象的浏览器请求首先被定向到该 Web 缓存器。举例来说, 假设浏览器正在请求对象 <http://www.someschool.edu/campus.gif>, 将会发生如下情况:

- 1) 浏览器创建一个到 Web 缓存器的 TCP 连接, 并向 Web 缓存器中的对象发送一个 HTTP 请求。



- 2) Web 缓存器进行检查，看看本地是否存储了该对象副本。如果有，Web 缓存器就向客户浏览器用 HTTP 响应报文返回该对象。
- 3) 如果 Web 缓存器中没有该对象，它就打开一个与该对象的初始服务器（即 WWW.someschool.edu）的 TCP 连接。Web 缓存器则在这个缓存器到服务器的 TCP 连接上发送一个对该对象的 HTTP 请求。在收到该请求后，初始服务器向该 Web 缓存器发送具有该对象的 HTTP 响应。
- 4) 当 Web 缓存器接收到该对象时，它在本地存储空间存储一份副本，并向客户的浏览器用 HTTP 响应报文发送该副本（通过现有的客户浏览器和 Web 缓存器之间的 TCP 连接）。

在因特网上部署 Web 缓存器有两个原因。首先，Web 缓存器可以大大减少对客户请求的响应时间，特别是当客户与初始服务器之间的瓶颈带宽远低于客户与 Web 缓存器之间的瓶颈带宽时更是如此。其次，Web 缓存器能够大大减少一个机构的接入链路到因特网的通信量。

2.2.6 条件 GET 方法

尽管高速缓存能减少用户感受到的响应时间，但也引入了一个新的问题，即存放在缓存器中的对象副本可能是陈旧的。

幸运的是，HTTP 协议有一种机制，允许缓存器证实它的对象是最新的。这种机制就是条件 GET(conditional GET) 方法。如果：① 请求报文使用 GET 方法；并且② 请求报文中包含一个“*If Modified-Since:*”首部行。那么，这个 HTTP 请求报文就是一个条件 GET 请求报文。

为了演示，我们给出以下的例子：

首先，一个代理缓存器 (proxy cache) 代表一个请求浏览器，向某 Web 服务器发送一个请求报文：

```
1 GET /fruit/kiwi.gif HTTP/1.1
Host: www.exotiquecuisine.com
```

其次，该 Web 服务器向缓存器发送具有被请求的对象的响应报文：

```
HTTP/1.1 200 OK
2 Date: Sat, 3 Oct 2015 15:39:29
Server: Apache/1.3.0 (Unix)
4 Last-Modified: Wed, 9 Sep 2015 09:23:24
Content-Type: image/gif
6 (data data data data data ...)
```



该缓存器在将对象转发到请求的浏览器的同时，也在本地缓存了该对象。重要的是，缓存器在存储该对象时也存储了最后修改日期。由于在过去的一个星期中位于 Web 服务器上的该对象可能已经被修改了，该缓存器通过发送一个条件 GET 执行最新检查。具体来说，该缓存器发送：

```
1 GET /fruit/kiwi.gif HTTP/1.1
2 Host: www.exotiquecuisine.com
If-modified-since: Wed, 9 Sep 2015 09:23:24
```

该条件 GET 报文告诉服务器，仅当自指定日期之后该对象被修改过，才发送该对象。假设该对象自 2015 年 9 月 9H09: 23: 24 后没有被修改。接下来的第四步，Web 服务器向该缓存器发送一个响应报文：

```
1 HTTP/1.1 304 Not Modified
Date: Sat, 10 Oct 2015 15:39:29
3 Server: Apache/1.3.0 (Unix)
(empty entity body)
```

我们看到，作为对该条件 GET 方法的响应，该 Web 服务器仍发送一个响应报文，但并没有在该响应报文中包含所请求的对象。包含该对象只会浪费带宽，并增加用户感受到的响应时间，特别是如果该对象很大的时候更是如此。值得注意的是在最后的响应报文中，状态行中为 304 Not Modified，它告诉缓存器可以使用该对象，能向请求的浏览器转发它（该代理缓存器）缓存的该对象副本。

2.3 因特网中的电子邮件

与普通邮件一样，电子邮件是一种异步通信媒介，即当人们方便时就可以收发邮件，不必与他人的计划进行协调。与普通邮件相比，电子邮件更为快速并且易于分发，而且价格便宜。现代电子邮件具有许多强大的特性，包括具有附件、超链接、HTML 格式文本和图片的报文。

下图展示了因特网电子邮件系统的总体情况。从该图中我们可以看到它有 3 个主要组成部分：用户代理 (user agent)、邮件服务器 (mail server) 和简单邮件传输协议 (Simple Mail Transfer Protocol, SMTP)。

邮件服务器形成了电子邮件体系结构的核心。每个接收方在其中的某个邮件服务器上有一个邮箱 (mailbox)。

一个典型的邮件发送过程是：从发送方的用户代理开始，传输到发送方的邮件服务器，再传输到接收方的邮件服务器，然后在这里被分发到接收方的邮箱中。



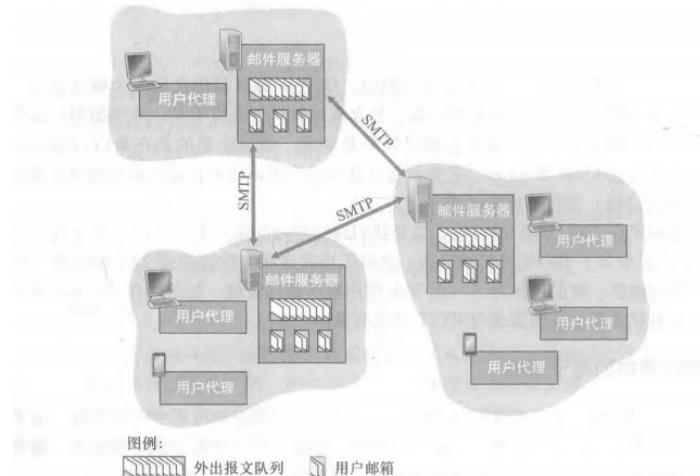


图 2.8: 电子邮件系统的总体描述

如果发送方的服务器不能将邮件交付给接收方的服务器, 发送方的邮件服务器在一个报文队列 (message queue) 中保持该报文并在以后尝试再次发送。通常每 30 分钟左右进行一次尝试; 如果几天后仍不能成功, 服务器就删除该报文并以电子邮件的形式通知发送方。

SMTP 是因特网电子邮件中主要的应用层协议。它使用 TCP 可靠数据传输服务, 从发送方的邮件服务器向接收方的邮件服务器发送邮件。像大多数应用层协议一样, SMTP 也有两个部分: 运行在发送方邮件服务器的客户端和运行在接收方邮件服务器的服务器端。每台邮件服务器上既运行 SMTP 的客户端也运行 SMTP 的服务器端。当一个邮件服务器向其他邮件服务器发送邮件时, 它就表现为 SMTP 的客户; 当邮件服务器从其他邮件服务器上接收邮件时, 它就表现为一个 SMTP 的服务器。

2.3.1 SMTP

SMTP 是因特网电子邮件的核心。如前所述, SMTP 用于从发送方的邮件服务器发送报文到接收方的邮件服务器。SMTP 问世的时间比 HTTP 要长得多。尽管电子邮件应用在因特网上的独特地位可以证明 SMTP 有着众多非常出色的性质, 但它所具有的某种陈旧特征表明它仍然是一种继承的技术。

在 SMTP 协议中, 限制了所有邮件报文的体部分 (不只是其首部) 只能采用简单的 7 比特 ASCII 表示。为了描述 SMTP 的基本操作, 我们来模拟一种场景。假设 Alice 想给 Bob 发送一封简单的 ASCII 报文。

- 1) Alice 调用她的邮件代理程序并提供 Bob 的邮件地址 (例如 `bob@someschool.edu`), 撰写报文, 然后指示用户代理发送该报文
- 2) Alice 的用户代理把报文发给她的邮件服务器, 在那里该报文被放在报文队列中
- 3) 运行在 Alice 的邮件服务器上的 SMTP 客户端发现了报文队列中的这个报文, 它就创建一个到运行在 Bob 的邮件服务器上的 SMTP 服务器的 TCP 连接



- 4) 在经过一些初始 SMTP 握手后，SMTP 客户通过该 TCP 连接发送 Alice 的报文
- 5) 在 Bob 的邮件服务器上，SMTP 的服务器端接收该报文。Bob 的邮件服务器然后将该报文放入 Bob 的邮箱中
- 6) 在 Bob 方便的时候，他调用用户代理阅读该报文

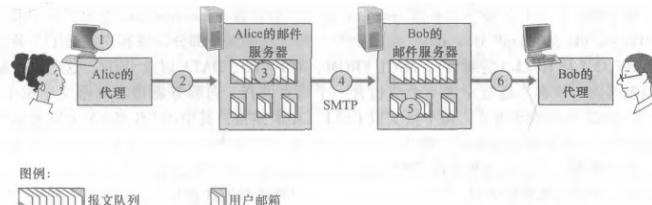


图 2.9：一个简单的 SMTP 邮件传输流程

需要注意的是：SMTP 一般不使用中间邮件服务器发送邮件，即使这两个邮件服务器位于地球的两端也是这样。假设 Alice 的邮件服务器在中国香港，而 Bob 的服务器在美国圣路易斯，那么这个 TCP 连接也是从香港服务器到圣路易斯服务器之间的直接相连。特别是，如果 Bob 的邮件服务器没有开机，该报文会保留在 Alice 的邮件服务器上并等待进行新的尝试，这意味着邮件并不在中间的某个邮件服务器存留。

接下来我们分析一个 SMTP 客户 (C) 和 SMTP 服务器 (S) 之间的交换报文文本的例子，客户的主机名为 crepes.fr，服务器的主机名为 hamburger.du

```

S: 220 hamburger.edu
2 C: HELO crepes.fr
S: 250 Hello crepes.fr, pleased to meet you
4 C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr ... Sender ok
6 C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
8 C: DATA
S: 354 Enter mail, end with ". " on a line by itself
10 C: Do you like ketchup?
C: How about pickles?
12 C: .
S: 250 Message accepted for delivery
14 C: QUIT
S: 221 hamburger.edu closing connection
  
```

第一行，表示服务器已经准备好与客户端建立连接，第二行则是客户端对服务器表明自己的身份。该客户发送了 5 条命令：HELO (是 HELLO 的缩写)、MAIL FROM、RCPTTO、DATA 以及 QUIT。这些命令都是自解释的。该客户通过发送一个只包含一



个句点的行，向服务器指示该报文结束了。(按照 ASCII 码的表示方法，每个报文以 CRLF.CRLF 结束，其中的 CR 和 LF 分别表示回车和换行。) 服务器对每条命令做出回答，其中每个回答含有一个回答码和一些(可选的)英文解释。

2.3.2 与 HTTP 的对比

这两个协议都用于从一台主机向另一台主机传送文件：HTTP 从 Web 服务器向 Web 客户(通常是一个浏览器)传送文件(也称为对象)；SMTP 从一个邮件服务器向另一个邮件服务器传送文件(即电子邮件报文)。

当然的，两者也有一些重要的区别：

- 1) HTTP 主要是一个拉协议(pull protocol)，即在方便的时候，某些人在 Web 服务器上装载信息，用户使用 HTTP 从该服务器拉取这些信息。特别是 TCP 连接是由想接收文件的机器发起的。另一方面，SMTP 基本上是一个推协议(push protocol)，即发送邮件服务器把文件推向接收邮件服务器。特别是，这个 TCP 连接是由要发送该文件的机器发起的
- 2) SMTP 要求每个报文(包括它们的体)采用 7 比特 ASCII 码格式。如果某报文包含了非 7 比特 ASCII 字符(如具有重音的法文字符)或二进制数据(如图形文件)，则该报文必须按照 7 比特 ASCII 码进行编码。HTTP 数据则不受这种限制
- 3) 是如何处理一个既包含文本又包含图形(也可能是其他媒体类型)的文档。HTTP 把每个对象封装到它自己的 HTTP 响应报文中，而 SMTP 则把所有报文对象放在一个报文之中。

2.3.3 邮件报文格式

首部行和该报文的体用空行(即回车换行)进行分隔。RFC 5322 定义了邮件首部行和它们的语义解释的精确格式。如同 HTTP 协议，每个首部行包含了可读的文本，是由关键词后跟冒号及其值组成的。某些关键词是必需的，另一些则是可选的。每个首部必须含有一个 From: 首部行和一个 To: 首部行；一个首部也许包含一个 Subject: 首部行以及其他可选的首部行。重要的是注意到下列事实：这些首部行不同于我们在之前所学到的 SMTP 命令(即使那里包含了某些相同的词汇，如 from 和 to)。那节中的命令是 SMTP 握手协议的一部分；本节中考察的首部行则是邮件报文自身的一部分。

一个典型的报文首部看起来如下：

```
1 From: alice@crepes.fr
2 To: bob@hamburger.edu
3 Subject: Searching for the meaning of life.
```



在报文首部之后，紧接着一个空白行，然后是以 ASCII 格式表示的报文体。你应该用 Telnet 向邮件服务器发送包含一些首部行的报文，包括 Subject: 首部行

2.3.4 邮件访问协议

一旦 SMTP 将邮件报文从 Alice 的邮件服务器交付给 Bob 的邮件服务器，该报文就被放入了 Bob 的邮箱中。在今天，邮件访问使用了一种客户-服务器体系结构，即典型的用户通过在用户端系统上运行的客户程序来阅读电子邮件。

如果接收方在本地 PC 上运行用户代理软件库，那么在他的本地 PC 上放置一个邮件服务器也是自然而然的事情。但是，为了能够及时接收可能在任何时候到达的新邮件，他的 PC 必须总是不间断地运行着并一直保持在线。这对于许多因特网用户而言是不现实的。

现在我们来看以下通信流程：

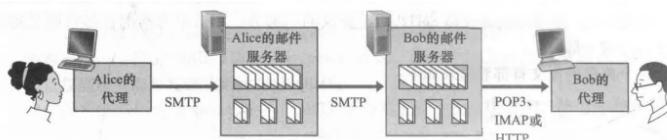


图 2.10：电子邮件协议及其通信协议

我们可以看见，像 Bob 这样的接收方，是如何通过运行其本地 PC 上的用户代理，获得位于他的某 ISP 的邮件服务器上的邮件呢？值得注意的是 Bob 的用户代理不能使用 SMTP 得到报文，因为取报文是一个拉操作，而 SMTP 协议是一个推协议。通过引入一个特殊的邮件访问协议来解决这个难题，该协议将 Bob 邮件服务器上的报文传送给他的本地 PC。目前有一些流行的邮件访问协议，包括第三版的邮局协议 (PostOffice Protocol—Version 3, POP3)、因特网邮件访问协议 (Internet Mail Access Protocol, IMAP) 以及 HTTP。

SMTP 用来将邮件从发送方的邮件服务器传输到接收方的邮件服务器；SMTP 也用来将邮件从发送方的用户代理传送到发送方的邮件服务器。如 POP3 这样的邮件访问协议用来将邮件从接收方的邮件服务器传送到接收方的用户代理

POP3

POP3 是一个极为简单的邮件访问协议，由 RFC 1939 进行定义。当用户代理（客户）打开了一个到邮件服务器（服务器）端口 110 上的 TCP 连接后，POP3 就开始工作了。随着建立 TCP 连接，POP3 按照三个阶段进行工作：特许 (authorization)、事务处理以及更新。

- 1) 在第一个阶段即特许阶段，用户代理发送（以明文形式）用户名和口令以鉴别用户。



- 2) 在第二个阶段即事务处理阶段，用户代理取回报文；同时在这个阶段用户代理还能进行如下操作，对报文做删除标记，取消报文删除标记，以及获取邮件的统计信息。
- 3) 在第三个阶段即更新阶段，它出现在客户发出了 quit 命令之后，目的是结束该 POP3 会话；这时，该邮件服务器删除那些被标记为删除的报文。

在 POP3 的事务处理过程中，用户代理发出一些命令，服务器对每个命令做出回答。回答可能有两种：+OK(有时后面还跟有服务器到客户的数据)，被服务器用来指示前面的命令是正常的；-ERR，被服务器用来指示前面的命令出现了某些差错。

特许阶段有两个主要的命令：user <user name> 和 pass <password>。

```
1 telnet mailserver 110
+OK POP3 server ready
3 user bob
+OK
5 pass hungry
+OK user successfully logged on
```

现在我们来看一下事务处理过程。使用 POP3 的用户代理通常被用户配置为“下载并删除”或者“下载并保留”方式。POP3 用户代理发出的命令序列取决于用户代理程序被配置为这两种工作方式的哪一种。使用下载并删除方式，用户代理发出 list、retr 和 dele 命令。举例来说，假设用户在他(她)的邮箱里有两个报文。在下面的对话中,C: (代表客户) 是用户代理，S: (代表服务器) 是邮件服务器。事务处理过程将类似于如下过程：

```
C: list
2 S: 1 498
S: 2 912
4 S: .
C: retr 1
6 S: (blah blah ...
S: .....
8 S: blah)
S: .
10 C: dele 1
C: retr 2
12 S: (blah blah ...
S: .....
14 S: blah)
S: .
```



```

16 C: dele 2
C: quit
18 S: +OK POP3 server signing off

```

用户代理首先请求邮件服务器列出所有存储的报文的长度。接着用户代理从邮件服务器收回并删除每封邮件。注意到在特许阶段以后，用户代理仅使用四个命令 list、retr. dele 和 quit，这些命令的语法定义在 RFC 1939 中。在处理 quit 命令后，POP3 服务器进入更新阶段，从用户的邮箱中删除邮件 1 和 2。

在用户代理与邮件服务器之间的 POP3 会话期间，该 POP3 服务器保留了一些状态信息；特别是记录了哪些用户报文被标记为删除了。然而，POP3 服务器并不在 POP3 会话过程中携带状态信息。会话中不包括状态信息大大简化了 POP3 服务的实现。

IMAP

不想看了。。。

2.4 DNS: 因特网的目录服务

因特网上的主机和人类一样，可以使用多种方式进行标识。主机的一种标识方法是用它的主机名 (hostname)。然而，主机名几乎没有提供（即使有也很少）关于主机在因特网中位置的信息。况且，因为主机名可能由不定长的字母数字组成，路由器难以处理。由于这些原因，主机也可以使用所谓 IP 地址 (IP address) 进行标识。

2.4.1 DNS 提供的服务

我们刚刚看到了识别主机有两种方式，通过主机名或者 IP 地址。人们喜欢便于记忆的主机名标识方式，而路由器则喜欢定长的、有着层次结构的 IP 地址。为了折中这些不同的偏好，我们需要一种能进行主机名到 IP 地址转换的目录服务。这就是域名系统 (Domain Name System, DNS) 的主要任务。

DNS 是：①一个由分层的 DNS 服务器 (DNS server) 实现的分布式数据库；②一个使得主机能够查询分布式数据库的应用层协议。DNS 服务器通常是运行 BIND (Berkeley Internet Name Domain) 软件 [BIND 2012] 的 UNIX 机器。DNS 协议运行在 UDP 之上，使用 53 号端口。

DNS 通常是由其他应用层协议所使用的，包括 HTTP、SMTP 和 FTP，将用户提供的主机名解析为 IP 地址。举一个例子，考虑运行在某用户主机上的一个浏览器（即一个 HTTP 客户）请求 URL www.someschool.edu/index.html 页面时会发生什么现象。为了



使用户的一主机能够将一个 HTTP 请求报文发送到 Web 服务器 www.someschool.edu, 该用户主机必须获得 www.someschool.edu 的 IP 地址。其做法如下：

- 1) 同一台用户主机上运行着 DNS 应用的客户端
- 2) 浏览器从上述 URL 中抽出主机名 www.someschool.edu, 并将这台主机名传给 DNS 应用的客户端
- 3) DNS 客户向 DNS 服务器发送一个包含主机名的请求
- 4) DNS 客户最终会收到一份回答报文, 其中含有对应于该主机名的 IP 地址
- 5) 一旦浏览器接收到来自 DNS 的该 IP 地址, 它能够向位于该 IP 地址 80 端口的 HTTP 服务器进程发起一个 TCP 连接。

除了进行主机名到 IP 地址的转换外, DNS 还提供了一些重要的服务:

- 1) 主机别名 (host aliasing)。有着复杂主机名的主机能拥有一个或者多个别名

例如, 一台名为 relay1.west-coast.enterprise.com 的主机, 可能还有两个别名为 enterprise.com 和 www.enterprise.com。在这种情况下, relay1.west-coast.enterprise.com 也称为规范主机名 (canonical hostname)。主机别名 (当存在时) 比主机规范名更加容易记忆。应用程序可以调用 DNS 来获得主机别名对应的规范主机名以及主机的 IP 地址

- 2) 邮件服务器别名 (mail server aliasing)。显而易见, 人们也非常希望电子邮件地址好记忆。

电子邮件应用程序可以调用 DNS, 对提供的主机名别名进行解析, 以获得该主机的规范主机名及其 IP 地址。

- 3) 负载分配 (load distribution)。DNS 也用于在冗余的服务器 (如冗余的 Web 服务器等) 之间进行负载分配。

繁忙的站点 (如 cnn.com) 被冗余分布在多台服务器上, 每台服务器均运行在不同的端系统上, 每个都有着不同的 IP 地址。由于这些冗余的 Web 服务器, 一个 IP 地址集合因此与同一个规范主机名相联系。DNS 数据库中存储着这些 IP 地址集合。当客户对映射到某地址集合的名字发出一个 DNS 请求时, 该服务器用 IP 地址的整个集合进行响应, 但在每个回答中循环这些地址次序。因为客户通常总是向 IP 地址排在最前面的服务器发送 HTTP 请求报文, 所以 DNS 就在所有这些冗余的 Web 服务器之间循环分配了负载。

2.4.2 DNS 工作机原理

假设运行在用户主机上的某些应用程序 (如 Web 浏览器或邮件阅读器) 需要将主机名转换为 IP 地址。这些应用程序将调用 DNS 的客户端, 并指明需要被转换的主机名 (在很多基于 UNIX 的机器上, 应用程序为了执行这种转换需要调用函数 gethostbyname())。用户主机上的 DNS 接收到后, 向网络中发送一个 DNS 查询报文。所有的 DNS 请求和回答报文使用 UDP 数据报经端口 53 发送。经过若干毫秒到若干秒的时延



后，用户主机上的 DNS 接收到一个提供所希望映射的 DNS 回答报文。这个映射结果则被传递到调用 DNS 的应用程序。因此，从用户主机上调用应用程序的角度看，DNS 是一个提供简单、直接的转换服务的黑盒子。

DNS 的一种简单设计是在因特网上只使用一个 DNS 服务器，该服务器包含所有的映射。在这种集中式设计中，客户直接将所有查询直接发往单一的 DNS 服务器，同时该 DNS 服务器直接对所有的查询客户做出响应。但是显然的，这种模型并不适用于今天的因特网：

- 1) 单点故障 (a single point of failure)。如果该 DNS 服务器崩溃，整个因特网随之瘫痪！
- 2) 通信容量 (traffic volume)。单个 DNS 服务器不得不处理所有的 DNS 查询 (用于为上亿台主机产生的所有 HTTP 请求报文和电子邮件报文服务)。
- 3) 远距离的集中式数据库 (distant centralized database)。单个 DNS 服务器不可能“邻近”所有查询客户。
- 4) 维护 (maintenance)。单个 DNS 服务器将不得不为所有的因特网主机保留记录。这不仅将使这个中央数据库庞大，而且它还不得不为解决每个新添加的主机而频繁更新

分布式、层次数据库

为了处理扩展性问题，DNS 使用了大量的 DNS 服务器，它们以层次方式组织，并且分布在全世界范围内。没有一台 DNS 服务器拥有因特网上所有主机的映射。

大致说来，有 3 种类型的 DNS 服务器：根 DNS 服务器、顶级域 (Top Level Domain, TLD) DNS 服务器和权威 DNS 服务器。这些服务器以下图中所示的层次结构组织起来。



图 2.11: 部分 DNS 服务器的层次结构

- 1) 根 DNS 服务器。有 400 多个根名字服务器遍及全世界。这些根名字服务器由 13 个不同的组织管理。根名字服务器提供 TLD 服务器的 IP 地址
- 2) 顶级域 (DNS) 服务器。对于每个顶级域 (如 com、org、net、edu 和 gov) 和所有国家的顶级域 (如 uk、fr、ca 和 jp)，都有 TLD 服务器 (或服务器集群)。TLD 服务器提供了权威 DNS 服务器的 IP 地址
- 3) 权威 DNS 服务器。在因特网上具有公共可访问主机 (如 Web 服务器和邮件服务器) 的每个组织机构必须提供公共可访问的 DNS 记录，这些记录将这些主机的名字映射为 IP 地址。



除了上述三种 DBS 服务器，还有另一类重要的 DNS 服务器，称为本地 DNS 服务器 (local DNS server)。严格说来，一个本地 DNS 服务器并不属于该服务器的层次结构，但它对 DNS 层次结构是至关重要的。

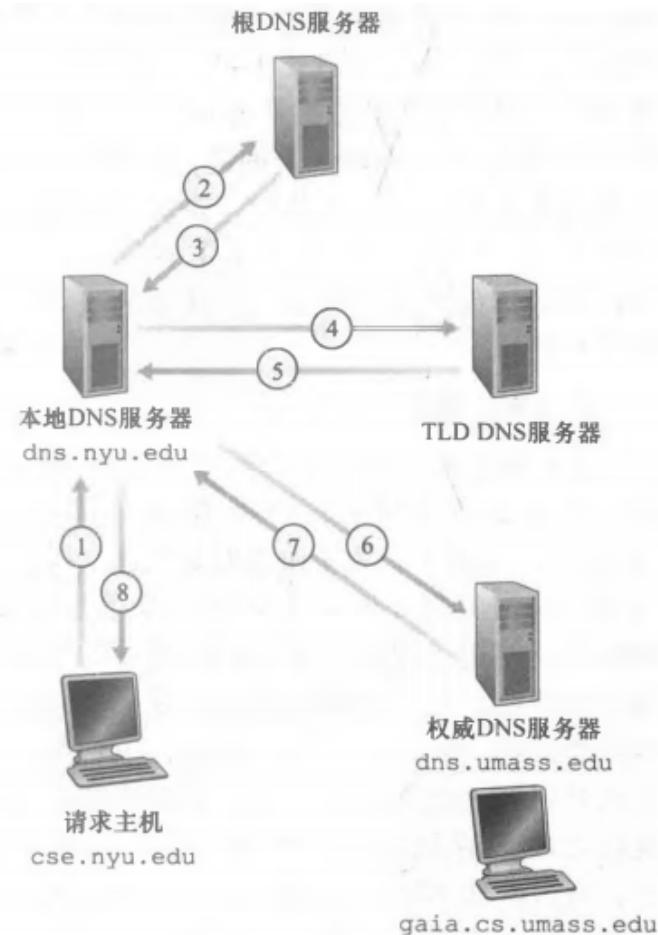


图 2.12: 各种 DNS 服务器的交互

首先第一步，请求主机将 gaia.cs.umass.edu 包装成报文发送给本地服务器，本地服务器将该报文转发给根 DNS 服务器，根 DNS 服务器注意到.edu 信息，并向本地服务器发送负责.edu 的 TLD 的 IP 地址列表。

然后，本地服务器通过得到的 IP 地址列表向 TLD 服务器发送查询报文，当 TLD 服务器查询到.umass.edu 前缀，并用权威 DNS 服务器的 IP 地址进行响应，最终本地服务器直接向权威服务器发送查询报文，权威服务器使用 gaia.cs.umass.edu 的 IP 地址进行响应。

上述所示的例子利用了递归查询 (recursive query) 和迭代查询 (iterative query)。从 cse.nyu.edu 到 dns.nyu.edu 发出的查询是递归查询，因为该查询以自己的名义请求 dns.nyu.edu 来获得该映射。而后继的 3 个查询是迭代查询，因为所有的回答都是直接返回给 dns.nyu.edu。



DNS 缓存

实际上，为了改善时延性能并减少在因特网上到处传输的 DNS 报文数量，DNS 广泛使用了缓存技术。DNS 缓存的原理非常简单。在一个请求链中，当某 DNS 服务器接收一个 DNS 回答（例如，包含某主机名到 IP 地址的映射）时，它能将映射缓存在本地存储器中。由于主机和主机名与 IP 地址间的映射并不是永久的，DNS 服务器在一段时间后（通常设置为两天）将丢弃缓存的信息。

2.4.3 DNS 记录和报文

共同实现 DNS 分布式数据库的所有 DNS 服务器存储了资源记录（Resource Record,RR），RR 提供了主机名到 IP 地址的映射。每个 DNS 回答报文包含了一条或多条资源记录。

资源记录是一个包含了下列字段的 4 元组：

$$(Name, Value, Type, TTL)$$

TTL 是该记录的生存时间，它决定了资源记录应当从缓存中删除的时间。在下面给出的记录例子中，我们忽略掉 TTL 字段。Name 和 Value 的值取决于 Type：

- 1) 如果 Type = A，则 Name 是主机名，Value 是该主机名对应的 IP 地址。因此，一条类型为 A 的资源记录提供了标准的主机名到 IP 地址的映射。
- 2) 如果 Type = NS，则 Name 是个域（如 foo.com），而 Value 是个知道如何获得该域中主机 IP 地址的权威 DNS 服务器的主机名。这个记录用于沿着查询链来路由 DNS 查询。
- 3) 如果 Type = NAME，则 Value 是另 9 名为 Name 的主机对应的规范主机名。该记录能够向查询的主机提供一个主机名对应的规范主机名。
- 4) 如果 Type = MX，则 Value 是个别名为 Name 的邮件服务器的规范主机名。MX 记录允许邮件服务器主机名具有简单的别名。值得注意的是，通过使用 MX 记录，一个公司的邮件服务器和其他服务器（如它的 Web 服务器）可以使用相同的别名。为了获得邮件服务器的规范主机名，DNS 客户应当请求一条 MX 记录；而为了获得其他服务器的规范主机名，DNS 客户应当请求 CNAME 记录。

DNS 报文

DNS 只有两种报文：DNS 查询和回答报文。并且，查询和回答报文有着相同的格式：

对于 DNS 报文中个字段的语义如下：

- 1) 受不了了，，，以后有精力再看吧。。



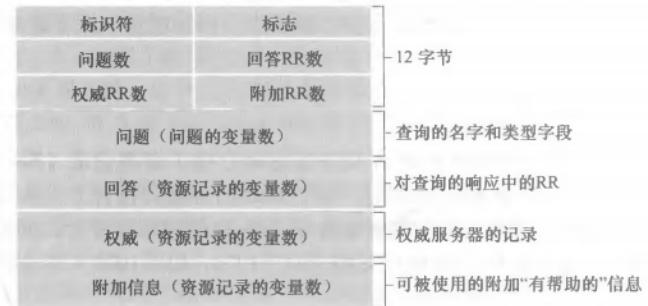


图 2.13: DNS 报文格式

2.5 视频流和内容分发网

2.5.1 因特网视频

在流式存储视频应用中，基础的媒体是预先录制的视频，例如电影、电视节目、录制好的体育事件或录制好的用户生成的视频。这些预先录制好的视频放置在服务器上，用户按需向这些服务器发送请求来观看视频。

从网络的观点看，也许视频最为突出的特征是它的高比特率。压缩的因特网视频的比特率范围通常从用于低质量视频的 100kbps, 到用于流式高分辨率电影的超过 3Mbps, 再到用于 4K 流式展望的超过 10Mbps。这能够转换为巨大的流量和存储，特别是对高端视频。例如，单一 2Mbps 视频在 67 分钟期间将耗费 1GB 的存储和流量。到目前为止，对流式视频的最为重要的性能度量是平均端到端吞吐量。为了提供连续不断的布局，网络必须为流式应用提供平均吞吐量，这个流式应用至少与压缩视频的比特率一样大

2.5.2 HTTP 流和 DASH

在 HTTP 流中，视频只是存储在 HTTP 服务器中作为一个普通的文件，每个文件有一个特定的 URL。

当用户要看该视频时，客户与服务器创建一个 TCP 连接并发送对该 URL 的 HTTP GET 请求。服务器则以底层网络协议和流量条件允许的尽可能快的速率，在一个 HTTP 响应报文中发送该视频文件。在客户一侧，字节被收集在客户应用缓存中。一旦该缓存中的字节数量超过预先设定的门限，客户应用程序就开始播放，特别是，流式视频应用程序周期性地从客户应用程序缓存中抓取帧，对这些帧解压缩并且在用户屏幕上展现。因此，流式视频应用接收到视频就进行播放，同时缓存该视频后面部分的帧。

尽管 HTTP 流在实践中已经得到广泛部署，但它具有严重缺陷，即所有客户接收到相同编码的视频，尽管对不同的客户或者对于相同客户的不同时间而言，客户可用的带宽大小有很大不同。这导致了一种新型基于 HTTP 的流的研发，它常常被称为经 HTTP 的动态适应性流 (Dynamic Adaptive Streaming over HTTP, DASH)。在 DASH 中，



视频编码为几个不同的版本，其中每个版本具有不同的比特率，对应于不同的质量水平。客户动态地请求来自不同版本且长度为几秒的视频段数据块。

DASH 允许客户使用不同的以太网接入速率流式播放具有不同编码速率的视频。使用 DASH 后，每个视频版本存储在 HTTP 服务器中，每个版本都有一个不同的 URL。HTTP 服务器也有一个告示文件 (manifest file)，为每个版本提供了一个 URL 及其比特率。客户首先请求该告示文件并且得知各种各样的版本。然后客户通过在 HTTP GET 请求报文中对每块指定一个 URL 和一个字节范围，一次选择一块。在下载块的同时，客户也测量接收带宽并运行一个速率决定算法来选择下次请求的块。

2.6 套接字编程：生成网络应用

UDP 套接字编程

现在我们仔细观察使用 UDP 套接字的两个通信进程之间的交互。在发送进程能够将数据分组推出套接字之门之前，当使用 UDP 时，必须先将目的地址附在该分组之上。在该分组传过发送方的套接字之后，因特网将使用该目的地址通过因特网为该分组选路到接收进程的套接字。当分组到达接收套接字时，接收进程将通过该套接字取回分组，然后检查分组的内容并采取适当的动作。

目的主机的 IP 地址是目的地址的一部分。通过在分组中包括目的地的 IP 地址，因特网中的路由器将能够通过因特网将分组选路到目的主机。但是因为一台主机可能运行许多网络应用进程，每个进程具有一个或多个套接字，所以在目的主机指定特定的套接字也是必要的。当生成一个套接字时，就为它分配一个称为端口号 (port number) 的标识符。因此，如你所期待的，分组的目的地址也包括该套接字的端口号。总的来说，发送进程为分组附上目的地址，该目的地址是由目的主机的 IP 地址和目的地套接字的端口号组成的。此外，如我们很快将看到的那样，发送方的源地址也是由源主机的 IP 地址和源套接字的端口号组成，该源地址也要附在分组之上。然而，将源地址附在分组之上通常并不是由 UDP 应用程序代码所为，而是由底层操作系统自动完成的。

- 1) 客户从其键盘读取一行字符 (数据) 并将该数据向服务器发送。
- 2) 服务器接收该数据并将这些字符转换为大写。
- 3) 服务器将修改的数据发送给客户。
- 4) 客户接收修改的数据并在其监视器上将该行显示出来

```

#include <cstring>
2 #include <iostream>
#include <netinet/in.h>
4 #include <string>
#include <sys/socket.h>
6 #include <arpa/inet.h>

```



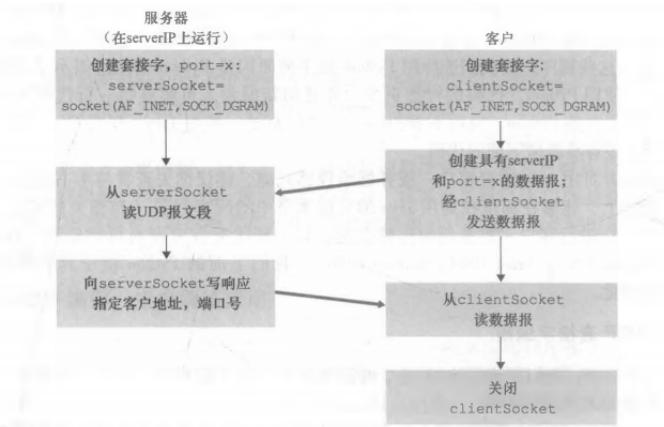


图 2.14: 使用 UDP 的客户-服务器应用

```

8 const std::string SERVER_ADDR = "127.0.0.1";
9 const int PORT = 8081;
10 const size_t BUFFER_SIZE = 1024;

12 int main() {
13     int sock_fd;
14     char buffer[BUFFER_SIZE] = {};
15     struct sockaddr_in server_addr;
16
17     if ((sock_fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
18         std::cerr << "socket fail" << std::endl;
19         exit(-1);
20     }

22     memset(&server_addr, 0, sizeof(server_addr));
23     server_addr.sin_family = AF_INET;
24     server_addr.sin_addr.s_addr = inet_addr(SERVER_ADDR.c_str());
25     server_addr.sin_port = htons(PORT);

26     socklen_t server_len = sizeof(sockaddr);
27     while (true) {
28         std::cout << "[CLIENT]: ";
29         gets(buffer);
30         sendto(sock_fd, (const char*)buffer, sizeof(buffer), 0,
31               (const struct sockaddr*)&server_addr, server_len);

32         memset(buffer, 0, BUFFER_SIZE);
33         ssize_t len = recvfrom(sock_fd, (char*)buffer, BUFFER_SIZE,

```

```

    0, ( struct sockaddr*)&server_addr , &server_len);
    if (len < 0) {
36        std :: cerr << "recv fail" << std :: endl;
        exit(-1);
    }
    std :: cout << "[CLIENT RECV]: " << buffer << "\n";
40 }
return 0;
42 }
```

Listing 2.1: udp client code

```

1 #include <cstring>
2 #include <iostream>
3 #include <arpa/inet.h>
4 #include <netinet/in.h>
5 #include <sys/socket.h>
6 #include <string>

8 const size_t BUFFER_SIZE = 1024;
const int PORT = 8081;
10
11 int main(int, char**){
12     int sock_fd;
13     sockaddr_in server_addr, client_addr;
14     char buffer[BUFFER_SIZE];

16     if ((sock_fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
17         std :: cerr << "socket fail" << std :: endl;
18         exit(-1);
19     }

20     memset(&server_addr, 0, sizeof server_addr);
21     memset(&client_addr, 0, sizeof client_addr);

24     server_addr.sin_family = AF_INET;
25     server_addr.sin_addr.s_addr = INADDR_ANY;
26     server_addr.sin_port = htons(PORT);

28     if (bind(sock_fd, (const sockaddr*)&server_addr, sizeof
```



```

server_addr) < 00) {
    std::cerr << "bind fail" << std::endl;
30   exit(-1);
}
32
socklen_t client_len = sizeof(client_addr);
34
while (true) {
36     memset(buffer, 0, sizeof buffer);
37     ssize_t len = recvfrom(sock_fd, (char*)buffer, BUFFER_SIZE,
38     0, (sockaddr*)&client_addr, &client_len);
39     if (len < 0) {
40         exit(-1);
41     }
42     buffer[len] = '\0';
43     std::cout << "[SERVER RECV]: " << buffer << "\n";
44     for (char& c : buffer) {
45         if (islower(c)) {
46             c = toupper(c);
47         }
48     }
49     sendto(sock_fd, (const char*)buffer, sizeof buffer, 0,
50     (const sockaddr*)&client_addr, client_len);
51 }
52 return 0;
}

```

Listing 2.2: udp server code

2.6.1 TCP 套接字编程

与 UDP 不同，TCP 是一个面向连接的协议。这意味着在客户和服务器能够开始互相发送数据之前，它们先要握手和创建一个 TCP 连接。TCP 连接的一端与客户套接字相联系，另一端与服务器套接字相联系。当创建该 TCP 连接时，我们将其与客户套接字地址 (IP 地址和端口号) 和服务器套接字地址 (IP 地址和端口号) 关联起来。使用创建的 TCP 连接，当一侧要向另一侧发送数据时，它只需经过其套接字将数据丢进 TCP 连接。这与 UDP 不同，UDP 服务器在将分组丢进套接字之前必须为其附上一个目的地地址。

```
1 #include <cstring>
```



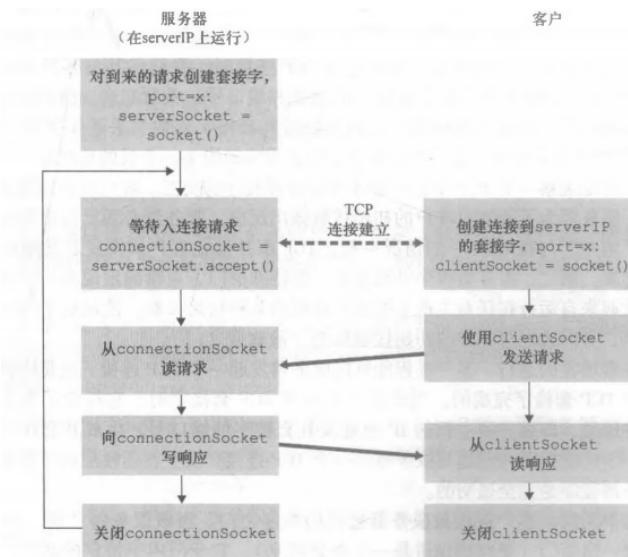


图 2.15: 使用 TCP 的客户-服务器应用

```

1 #include <iostream>
3 #include <arpa/inet.h>
4 #include <netinet/in.h>
5 #include <sys/socket.h>
6 #include <string>
7 #include <unistd.h>

9 const size_t BUFFER_SIZE = 1024;
10 const int PORT = 8081;

11 int main(int, char**){
12     int sock_fd, client_sock;
13     sockaddr_in server_addr, client_addr;
14     char buffer[BUFFER_SIZE];

17     if ((sock_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
18         std::cerr << "socket fail" << std::endl;
19         exit(-1);
20     }

21     memset(&server_addr, 0, sizeof server_addr);
22     memset(&client_addr, 0, sizeof client_addr);

25     server_addr.sin_family = AF_INET;
26     server_addr.sin_addr.s_addr = INADDR_ANY;

```



```

27     server_addr.sin_port = htons(PORT);

29     if (bind(sock_fd, (const sockaddr*)&server_addr, sizeof
server_addr) < 0) {
30         std::cerr << "bind fail" << std::endl;
31         exit(-1);
32     }

33     if (listen(sock_fd, 5) < 0) {
34         std::cerr << "listen fail" << std::endl;
35         exit(-1);
36     }

37     socklen_t client_len = sizeof(client_addr);

38     clinet_sock = accept(sock_fd, (sockaddr*)&client_addr, &
client_len);

39     while (true) {
40         memset(buffer, 0, sizeof buffer);
41         ssize_t len = read(clinet_sock, buffer, BUFFER_SIZE);
42         if (len < 0) {
43             exit(-1);
44         }
45         buffer[len] = '\0';
46         std::cout << "[SERVER RECV]: " << buffer << "\n";
47         for (char& c : buffer) {
48             if (islower(c)) {
49                 c = toupper(c);
50             }
51         }
52         send(clinet_sock, buffer, sizeof buffer, 0);
53     }
54     return 0;
55 }
```

Listing 2.3: tcp server code

```

1 #include <cstring>
2 #include <iostream>
```



```
3 #include <netinet/in.h>
4 #include <string>
5 #include <sys/socket.h>
6 #include <arpa/inet.h>
7 #include <unistd.h>

9 const std::string SERVER_ADDR = "127.0.0.1";
10 const int PORT = 8081;
11 const size_t BUFFER_SIZE = 1024;

13 int main() {
14     int sock_fd;
15     char buffer[BUFFER_SIZE] = {};
16     struct sockaddr_in server_addr;
17
18     if ((sock_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
19         std::cerr << "socket fail" << std::endl;
20         exit(-1);
21     }
22
23     memset(&server_addr, 0, sizeof server_addr);
24     server_addr.sin_family = AF_INET;
25     server_addr.sin_addr.s_addr = inet_addr(SERVER_ADDR.c_str());
26     server_addr.sin_port = htons(PORT);
27
28     if (connect(sock_fd, (sockaddr*)&server_addr, sizeof
29     server_addr) < 0) {
30         std::cerr << "connect fail" << std::endl;
31         exit(-1);
32     }
33
34     socklen_t server_len = sizeof(sockaddr);
35     while (true) {
36         std::cout << "[CLIENT]: ";
37         gets(buffer);
38         send(sock_fd, buffer, sizeof buffer, 0);
39
40         memset(buffer, 0, BUFFER_SIZE);
41         ssize_t len = read(sock_fd, buffer, BUFFER_SIZE);
```

```
41     if (len < 0) {
42         std::cerr << "recv fail" << std::endl;
43         exit(-1);
44     }
45     std::cout << "[CLIENT RECV]: " << buffer << "\n";
46 }
47 return 0;
```

Listing 2.4: tcp client code

2.7 课后习题

有时间，再做！下次一定！



第3章 运输层



运输层位于应用层和网络层之间，是分层的网络体系结构的重要部分。该层为运行在不同主机上的应用进程提供直接的通信服务起着至关重要的作用。

3.1 概述和运输层服务

运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 (logic communication) 功能。从应用程序的角度看，通过逻辑通信，运行不同进程的主机好像直接相连一样；

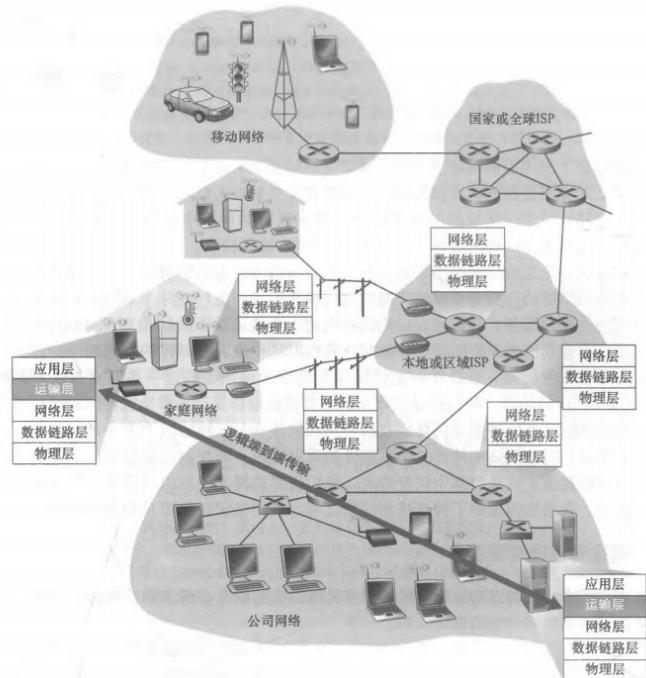


图 3.1: 运输层在应用程序进程间提供逻辑的并非物理的通信

如上图所示，运输层协议是在端系统中而不是在路由器中实现的。在发送端，运输层将从发送应用程序进程接收到的报文转换成运输层分组，用因特网术语来讲该分组称为运输层报文段 (segment)。

实现的方法 (可能) 是将应用报文划分为较小的块，并为每块加上一个运输层首部以生成运输层报文段。然后，在发送端系统中，运输层将这些报文段传递给网络层，网路层将其封装成网络层分组 (即数据报) 并向目的地发送。

3.1.1 运输层和网络层的关系

网络层提供了主机之间的逻辑通信，而运输层为运行在不同主机上的进程之间提供了逻辑通信。

运输层协议只工作在端系统中。在端系统中，运输层协议将来自应用进程的报文移动到网络边缘(即网络层)，反过来也是一样，但对有关这些报文在网络核心如何移动并不作任何规定。

运输协议能够提供的服务常常受制于底层网络层协议的服务模型。如果网络层协议无法为主机之间发送的运输层报文段提供时延或带宽保证的话，运输层协议也就无法为进程之间发送的应用程序报文提供时延或带宽保证。

3.1.2 因特网运输层概述

因特网为应用层提供了两种截然不同的可用运输层协议。这些协议一种是 UDP(用户数据报协议)，它为调用它的应用程序提供了一种不可靠、无连接的服务。另一种是 TCP(传输控制协议)，它为调用它的应用程序提供了一种可靠的、面向连接的服务。

为了简化术语，我们将运输层分组称为报文段 (segment)。然而，因特网文献(如 RFC 文档)也将 TCP 的运输层分组称为报文段，而常将 UDP 的分组称为数据报 (*data gram*)。而这类因特网文献也将网络层分组称为数据报！在此处，我们将 TCP 和 UDP 的分组统称为报文段。

因特网网络层协议有一个名字叫 IP，即网际协议。IP 为主机之间提供了逻辑通信。IP 的服务模型是尽力而为交付服务 (best-effort delivery service)。

这意味着 IP 尽它“最大的努力”在通信的主机之间交付报文段，但它并不做任何确保。特别是，它不确保报文段的交付，不保证报文段的按序交付，不保证报文段中数据的完整性。由于这些原因，IP 被称为不可靠服务 (unreliable service)。在此处，我们只需要记住每台主机只有一个 IP 地址。

UDP 和 TCP 最基本的责任是，将两个端系统间 IP 的交付服务扩展为运行在端系统上的两个进程之间的交付服务。将主机间交付扩展到进程间交付被称为运输层的多路复用 (transport-layer multiplexing) 与多路分解 (demultiplexing)。

UDP 和 TCP 还可以通过在其报文段首部中包括差错检查字段而提供完整性检查。进程到进程的数据交付和差错检查是两种最低限度的运输层服务，也是 UDP 所能提供的仅有的两种服务。特别是，与 IP一样，UDP 也是一种不可靠的服务，即不能保证一个进程所发送的数据能够完整无缺地(或全部！)到达目的进程。

3.2 多路复用与多路分解

多路复用与多路分解服务是所有计算机网络都需要的。



在目的主机，运输层从紧邻其下的网络层接收报文段。运输层负责将这些报文段中的数据交付给在主机上运行的适当应用程序进程。

一个进程(作为网络应用的一部分)有一个或多个套接字(socket)，它相当于从网络向进程传递数据和从进程向网络传递数据的门户。因此，如下所示，在接收主机中的运输层实际上并没有直接将数据交付给进程，而是将数据交给了一个中间的套接字。由于在任一时刻，在接收主机上可能有不止一个套接字，所以每个套接字都有唯一的标识符。

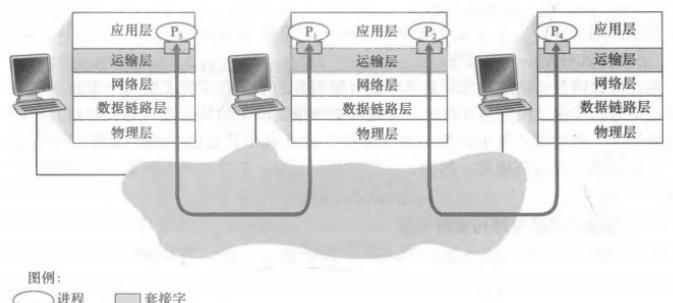


图 3.2：运输层的多路复用与多路分解

现在我们考虑接收主机怎样将一个到达的运输层报文段定向到适当的套接字。为此目的，每个运输层报文段中具有几个字段。在接收端，运输层检查这些字段，标识出接收套接字，进而将报文段定向到该套接字。

将运输层报文段中的数据交付到正确的套接字的工作称为多路分解(demultiplexing)。

在源主机从不同套接字中收集数据块，并为每个数据块封装上首部信息(这将在以后用于分解)从而生成报文段，然后将报文段传递到网络层，所有这些工作称为多路复用(multiplexing)。

运输层多路复用要求：

- 1) 套接字有唯一标识符
- 2) 每个报文段有特殊字段来指示该报文段所要交付到的套接字

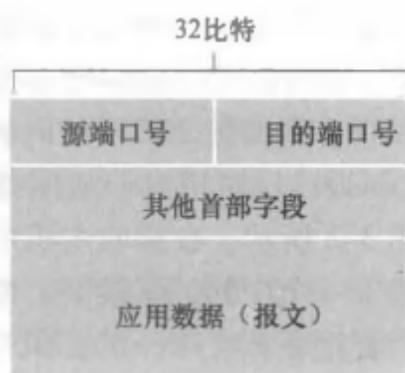


图 3.3：运输层报文段中源与目的端口字段



这些特殊字段是源端口号字段 (source port number field) 和目的端口号字段 (destination port number field)。

端口号是一个 16 比特的数，其大小在 0~65535 之间。0~1023 范围的端口号称为周知端口号 (well-known port number)，是受限制的。

现在应该清楚运输层是怎样能够实现分解服务的了：在主机上的每个套接字能够分配一个端口号，当报文段到达主机时，运输层检查报文段中的目的端口号，并将其定向到相应的套接字。然后报文段中的数据通过套接字进入其所连接的进程。

无连接的多路复用与多路分解

一个 *UDP* 套接字是由一个二元组全面标识的，该二元组包含一个目的 *IP* 地址和一个目的端口号。因此，如果两个 *UDP* 报文段有不同的源 *IP* 地址和/或源端口号，但具有相同的目的 *IP* 地址和目的端口号，那么这两个报文段将通过相同的目的套接字被定向到相同的目的进程。

面向连接的多路复用和多路分解

TCP 套接字是由一个四元组 (源 *IP* 地址, 源端口号, 目的 *IP* 地址, 目的端口号) 来标识的。因此，当一个 *TCP* 报文段从网络到达一台主机时，该主机使用全部 4 个值来将报文段定向 (分解) 到相应的套接字。特别与 *UDP* 不同的是，两个具有不同源 *IP* 地址或源端口号的到达 *TCP* 报文段将被定向到两个不同的套接字，除非 *TCP* 报文段携带了初始创建连接的请求。

3.3 无连接运输: UDP

由 [RFC 768] 定义的 *UDP* 只是做了运输协议能够做的最少工作。除了复用/分解功能及少量的差错检测外，它几乎没有对 *IP* 增加别的东西。

有许多应用更适合用 *UDP*，原因主要以下几点：

- 1) 关于发送什么数据以及何时发送的应用层控制更为精细。

采用 *UDP* 时，只要应用进程将数据传递给 *UDP*，*UDP* 就会将此数据打包进 *UDP* 报文段并立即将其传递给网络层。

- 2) 无须连接建立。

UDP 不会引入建立连接的时延。这可能是 *DNS* 运行在 *UDP* 之上而不是运行在 *TCP* 之上的主要原因。

- 3) 无连接状态。

TCP 需要在端系统中维护连接状态。此连接状态包括接收和发送缓存、拥塞控制参数以及序号与确认号的参数。*UDP* 不维护连接状态，也不跟踪这些参数。

- 4) 分组首部开销小。



每个 TCP 报文段都有 20 字节的首部开销，而 UDP 仅有 8 字节的开销。

3.3.1 UDP 报文段结构

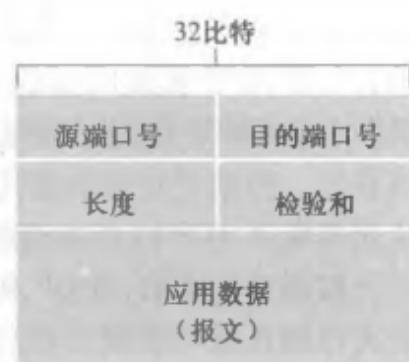


图 3.4: UDP 报文段结构

UDP 首部只有 4 个字段，每个字段由两个字节组成。通过端口号可以使目的主机将应用数据交给运行在目的端系统中的相应进程（即执行分解功能）。长度字段指示了在 UDP 报文段中的字节数（首部加数据）。因为数据字段的长度在一个 UDP 段中不同于在另一个段中，故需要一个明确的长度。接收方使用检验和来检查在该报文段中是否出现了差错。

3.3.2 UDP 校验和

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。

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。

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-to-end principle) 的例子。

3.4 可靠数据传输原理

下图说明了我们的学习框架。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。



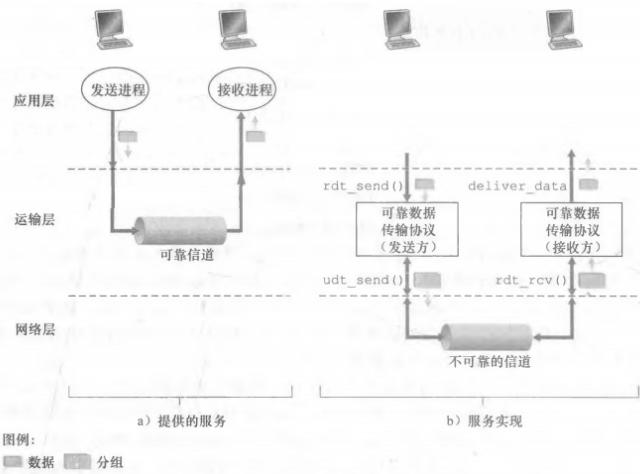


图 3.5: 可靠数据传输：服务模型与服务实现

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。

3.4.1 构造可靠数据传输协议

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。

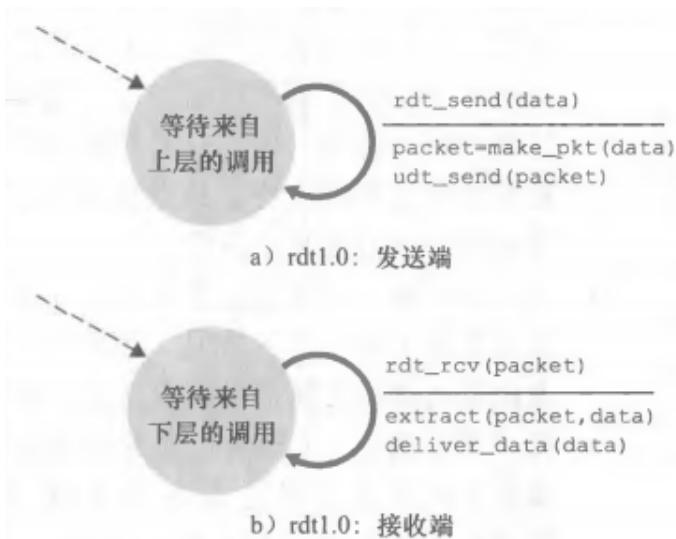


图 3.6: rdt1.0: 用于完全可靠信道的协议

上图显示了 rdt1.0 发送方和接收方的有限状态机 (Finite-State Machine, FSM) 的定义。

发送方和接收方有各自的 FSM。上图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，或没有事件发生而采取了一个动作，我们将横线上方或下方使用符号 \wedge ，以分别明确地表示缺少动作或事件。FSM 初始状态用虚线表示。

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。

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。一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。

ARQ 协议中需要另外三种协议来处理存在比特差错的情况：

- 1) 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。

- 2) 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。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。

- 3) 重传。

接收方收到有差错的分组时，发送方将重传该分组文。

下图表示 rdt2.0 的 FSM，该数据协议采用了差错检测、肯定确认与否定确认。

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，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。



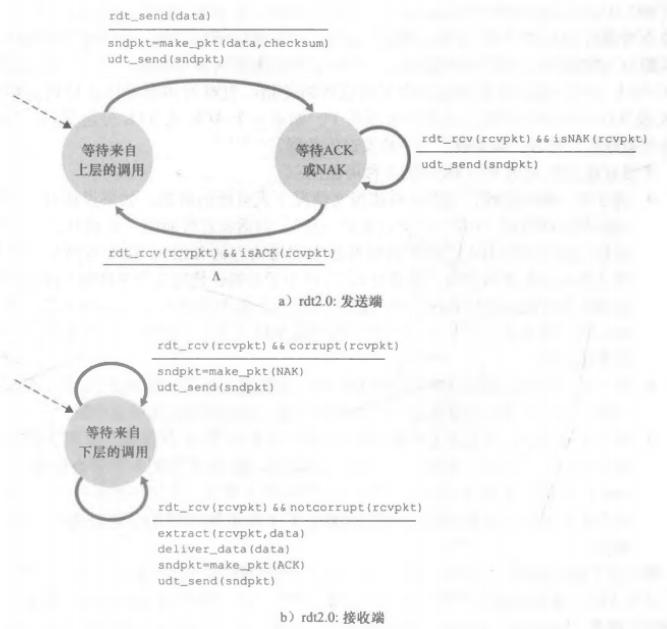


图 3.7: rdt2.0: 用于具有比特差错信道的协议

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，*rdcsend()* 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 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！

考虑处理 *ACK* 和 *NAK* 时的三种可能性：

- 1) 第一种可能性，接收者不明白发起者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
- 2) 第二种可能性，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
- 3) 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。

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。

剩下的 rdt2.2, rdt3.0 后面再说吧。。不想看了



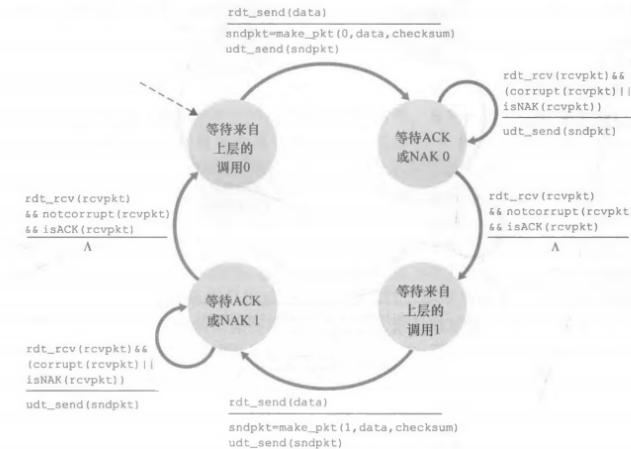


图 3.8: rdt2.1 发送方

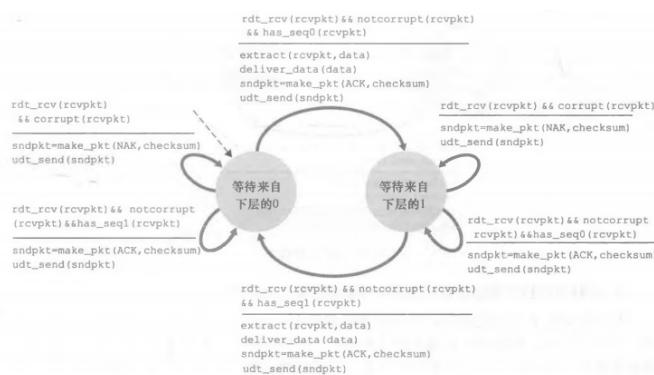


图 3.9: rdt2.1 接收方



3.5 面向连接的运输: TCP

3.5.1 TCP 连接

TCP 被称为是面向连接的 (connection-oriented), 这是因为在一个应用进程可以开始向另一个应用进程发送数据之前, 这两个进程必须先相互“握手”, 即它们必须相互发送某些预备报文段, 以建立确保数据传输的参数。作为 TCP 连接建立的一部分, 连接的双方都将初始化与 TCP 连接相关的许多 TCP 状态变量

TCP 连接也总是点对点 (point-to-point) 的, 即在单个发送方与单个接收方之间的连接。所谓“多播”(参见本书的在线补充材料), 即在一次发送操作中, 从一个发送方将数据传送给多个接收方, 这种情况对 TCP 来说是不可能的。

一旦建立起一条 TCP 连接, 两个应用进程之间就可以相互发送数据了。客户进程通过套接字 (该进程之门) 传递数据流。数据一旦通过该门, 它就由客户中运行的 TCP 控制了。

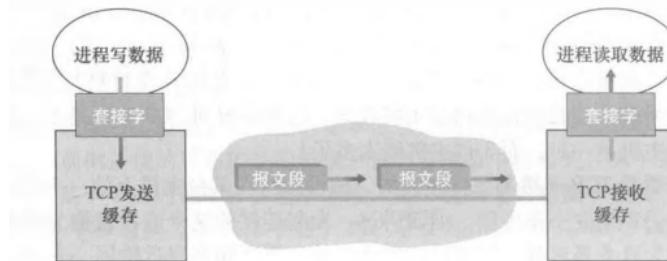


图 3.10: TCP 发送缓存和接收缓存

TCP 将这些数据引导到该连接的发送缓存 (send buffer) 里, 发送缓存是发起三次握手期间设置的缓存之一。接下来 TCP 就会不时从发送缓存里取出一块数据, 并将数据传递到网络层。

TCP 可从缓存中取出并放入报文段中的数据数量受限于最大报文段长度 (Maximum Segment Size, MSS)。MSS 通常根据最初确定的由本地发送主机发送的最大链路层帧长度 (即所谓的最大传输单元 (Maximum Transmission Unit, MTU)) 来设置。

设置该 MSS 要保证一个 TCP 报文段 (当封装在一个 IP 数据报中) 加上 TCP/IP 首部长度 (通常 40 字节) 将适合单个链路层帧。以太网和 PPP 链路层协议都具有 1500 字节的 MTU, 因此 MSS 的典型值为 1460 字节。已经提出了多种发现路径 MTU 的方法, 并基于路径 MTU 值设置 MSS(路径 MTU 是指能在从源到目的地的所有链路上发送的最大链路层帧 [RFC 1191])。注意到 MSS 是指在报文段里应用层数据的最大长度, 而不是指包括首部的 TCP 报文段的最大长度。



3.5.2 TCP 报文结构

TCP 报文段由首部字段和一个数据字段组成。数据字段包含一块应用数据。如前所述, MSS 限制了报文段数据字段的最大长度。当 TCP 发送一个大文件, 例如某 Web 页面上的一个图像时, TCP 通常是将该文件划分成长度为 MSS 的若干块(最后一块除外, 它通常小于 MSS)。

下图显示了 TCP 报文段的结构:



图 3.11: TCP 报文段结构

除了源端口号和目的端口号外, 还包括:

- 1) 32 比特的序号字段 (sequence number field) 和 32 比特的确认号字段 (acknowledgment number field)

这些字段被 TCP 发送方和接收方用来实现可靠数据传输服务

- 2) 16 比特的接收窗口字段 (receive window field)

该字段用于流量控制。我们很快就会看到, 该字段用于指示接收方愿意接受的字节数量。

- 3) 4 比特的首部长度字段 (header length field)

该字段指示了以 32 比特的字为单位的 TCP 首部长度。由于 TCP 选项字段的原因, TCP 首部的长度是可变的。(通常, 选项字段为空, 所以 TCP 首部的典型长度是 20 字节。)

- 4) 可选与变长的选项字段 (options field)

该字段用于发送方与接收方协商最大报文段长度 (MSS) 时, 或在高速网络环境下用作窗口调节因子时使用。首部字段中还定义了一个时间戳选项。

- 5) 6 比特的标志字段 (flag field)



[a] ACK 比特用于指示确认字段中的值是有效的，即该报文段包括一个对已被成功接收报文段的确认。

[b] RST、SYN 和 FIN 比特用于连接建立和拆除

[c] 在明确拥塞通告中使用了 CWR 和 ECE 比特

[d] 当 PSH 比特被置位时，就指示接收方应立即将数据交给上层

[e] URG 比特用来指示报文段里存在着被发送端的上层实体置为“紧急”的数据。紧急数据的最后一个字节由 16 比特的紧急数据指针字段 (urgent data pointer field) 指出。当紧急数据存在并给出指向紧急数据尾指针的时候，TCP 必须通知接收端的上层实体。

在实践中，PSH、URG 和紧急数据指针并没有使用。为了完整性起见，我们才提到这些字段。

序号和确认号

TCP 报文段首部中两个最重要的字段是序号字段和确认号字段。这两个字段是 TCP 可靠传输服务的关键部分。

TCP 把数据看成一个无结构的、有序的字节流。

假设主机 A 上的一个进程想通过一条 TCP 连接向主机 B 上的一个进程发送一个数据流。主机 A 中的 TCP 将隐式地对数据流中的每一个字节编号。假定数据流由一个包含 500000 字节的文件组成，其 MSS 为 1000 字节，数据流的首字节编号是 0。如下图所示，该 TCP 将为该数据流构建 500 个报文段。给第一个报文段分配序号 0, 第二个报文段分配序号 1000, 第三个报文段分配序号 2000, 以此类推。

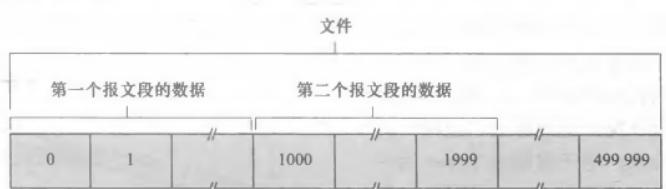


图 3.12: 文件数据划分成 TCP 报文段

现在来讨论确认号。主机 A 填充进报文段的确认号是主机 A 期望从主机 B 收到的下一字节的序号。假如主机 A 已经收到了来自主机 B 的 0 535 的所有字节，那么如果想要接收后续的字节，主机 A 应该往主机 B 发送 536 的确认号。

那么，TCP 中如果数据的包有丢失 (比如主机 A 收到了 0 535 与 900 1000 的报文段)，很显然，主机 A 没有接收到 536 899 的报文段。因此，主机 A 需要重新构建主机 B 的数据流，因此 A 到 B 的下一个报文段将在确认号中包含 536。因为 TCP 只确认该流中第一个丢失字节为止的字节，因此 TCP 被称为提供累计确认 (*cumulative acknowledgment*)。

最后，我们来考虑：当主机在一条 TCP 连接中收到失序报文段时该怎么办？



- 1) 接收方立即丢弃失序报文段
 - 2) 接收方保留失序的字节，并等待缺少的字节以填补该间隔
- 显然，第二种选择对网络贷款更为有效，是实践中采用的方法。

Telnet: 序号和确认号的一个学习案例

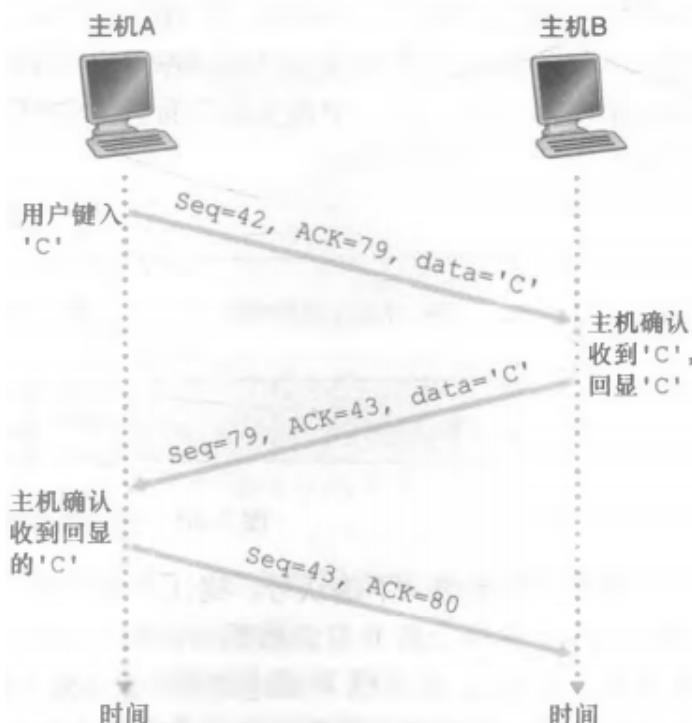


图 3.13: 一个经 TCP 的简单 Telnet 应用的确认号和序号

假设 A 发起会话 (因此, 主机 A 被标记为客户端)。假设客户和服务器的起始序号分别为 42 和 79, 一个报文段的序号就是该报文段数据字段首字节的序号。确认号就是主机正在等待的数据的下一个字节序号。

因此, 客户发送的第一个报文段序号为 42, 请求服务器的确认号为 79。

第二个报文段由服务器发往客户。首先为该服务器收到的数据提供一个确认, 因此在确认号中填入了 43, 然后回显了字符'C', 因此在 data 段中填入了字符'C'。

值得注意的是, 对客户到服务器的数据的确认被装载在一个承载服务器到客户的报文段中; 这种确认被称为是被捎带 (*piggybacked*) 在服务器到客户的报文段中的。

第三个报文段是从客户端发往服务器。其唯一目的是确认已从服务器收到的数据。因此确认号中填入的 80.

3.5.3 往返实践的估计与超时

TCP 采用超时/重传机制来处理报文段的丢失问题。



估计往返时间

TCP 通过如下方式估计往返时间: 报文段的样本 RTT(表示为 SampleRTT) 就是从某报文段被发出(即交给 IP)到对该报文段的确认被收到之间的时间量。

在任意时刻, 仅为一个已发送的但目前尚未被确认的报文段估计 SampleRTT, 从而产生一个接近每个 RTT 的新 SampleRTT 值。另外, TCP 决不为已被重传的报文段计算 SampleRTT; 它仅为传输一次的报文段测量 SampleRTT。

由于路由器的拥塞和端系统负载的变化, 这些报文段的 SampleRTT 值会随之波动。由于这种波动, 任何给定的 SampleRTT 值也许都是非典型的。因此, 为了估计一个典型的 RTT, 自然要采取某种对 SampleRTT 取平均的办法。TCP 维持一个 SampleRTT 均值(称为 EstimatedRTT)。一旦获得一个新 SampleRTT 时, TCP 就会根据下列公式来更新 EstimatedRTT:

$$\text{EstimatedRTT} = (1 - \alpha) \cdot \text{EstimatedRTT} + \alpha \cdot \text{SampleRTT}$$

即, EstimatedRTT 的新值由以前的值与 SampleRTT 新值加权组合而来, $\alpha = 0.125$

$$\text{EstimatedRTT} = 0.875 \cdot \text{EstimatedRTT} + 0.125 \cdot \text{SampleRTT}$$

值得注意的是, EstimatedRTT 是一个 SampleRTT 值的加权平均值。

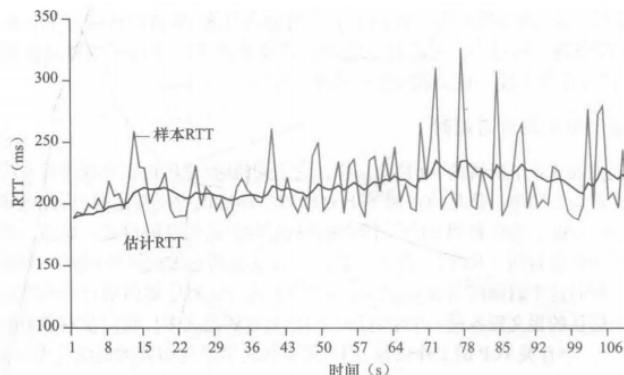


图 3.14: RTT 样本与 RTT 估计

[RFC 6298] 定义了 RTT 偏差 DevRTT, 用于估算 SampleRTT 一般会偏离 EstimatedRTT 的程度:

$$\text{DevRTT} = (1 - \beta) \cdot \text{DevRTT} + \beta \cdot |\text{SampleRTT} - \text{EstimatedRTT}|$$

注意到 DevRTT 是一个 SampleRTT 与 EstimatedRTT 之间差值的 EWMA¹。如果 SampleRTT 值波动较小, 那么 DevRTT 的值就会很小; 另一方面, 如果波动很大, 那

¹从统计学观点讲, 这种平均被称为指数加权移动平均 (Exponential Weighted Moving Average, EWMA)。在 EWMA 中的“指数”一词看起来是指一个给定的 SampleRTT 的权值在更新的过程中呈指数型快速衰减。



么 DevRTT 的值就会很大。 β 的推荐值为 0.25。

设置和管理重传超时间隔

假设已经给出了 EstimatedRTT 值和 DevRTT 值，超时间隔应该大于等于 EstimatedRTT，否则，将造成不必要的重传。但是超时间隔也不应该比 EstimatedRTT 大太多，否则当报文段丢失时，TCP 不能很快地重传该报文段，导致数据传输时延大。

$$\text{Timeoutinterval} = \text{EstimatedRTT} + 4 \cdot \text{DevRTT}$$

推荐的初始 Timeoutinterval 值为 1 秒 [RFC 6298]。同时，当出现超时后，TimeoutInterval 值将加倍，以免即将被确认的后继报文段过早出现超时。然而，只要收到报文段并更新 EstimatedRTT，就使用上述公式再次计算 TimeoutInterval。

3.5.4 可靠数据传输

因特网的网络层服务 (IP 服务) 是不可靠的。IP 不保证数据报的交付，不保证数据报的按序交付，也不保证数据报中数据的完整性。

TCP 在 IP 不可靠的尽力而为服务之上创建了一种可靠数据传输服务 (reliable data transfer service)。TCP 的可靠数据传输服务确保一个进程从其接收缓存中读出的数据流是无损坏、无间隙、非冗余和按序的数据流；即该字节流与连接的另一方端系统发送出的字节流是完全相同。

我们将以两个递增的步骤来讨论 TCP 是如何提供可靠数据传输的。我们先给出一个 TCP 发送方的高度简化的描述，该发送方只用超时来恢复报文段的丢失；然后再给出一个更全面的描述，该描述中除了使用超时机制外，还使用冗余确认技术。在接下来的讨论中，我们假定数据仅向一个方向发送，即从主机 A 到主机 B，且主机 A 在发送一个大文件。

```

1 /*
2 * 假设发送方不受TCP流量和拥塞控制的限制，来自上层数据的长度小于MSS,
3 * 且数据传送只在一个
4 * 方向进行。
5 */
6 NextSeqNum=InitialSeqNumber
7 SendBase=InitialSeqNumber

8 loop (永远) {
9     switch (事件)
10    事件：从上面应用程序接收到数据e
11        生成具有序号NextSeqNum的TCP报文段

```



```

12      if (定时器当前没有运行)
          启动定时器
14      向IP传递报文段
15      NextSeqNum = NextSeqNum + length(data)
16      break;
事件: 定时器超时
18      重传具有最小序号但仍未应答的报文段
19      启动定时器
20      break;
事件: 收到ACK, 具有ACK字段值y
21      if (y > SendBase) {
          SendBase = y
22      if (当前仍无任何应答报文段) {
          启动定时器
23      }
24      }
25      break;
26  }
27  }
28 }

} /*结束永远循环*/

```

Listing 3.1: 简化的 TCP 发送方

我们看到在 TCP 发送方有 3 个与发送和重传有关的主要事件：从上层应用程序接收数据；定时器超时和收到 ACK。

一旦第一个主要事件发生，TCP 从应用程序接收数据，将数据封装在一个报文段中，并把该报文段交给 IP。

还要注意到如果定时器还没有为某些其他报文段而运行，则当报文段被传给 IP 时，TCP 就启动该定时器。(将定时器想象为与最早的未被确认的报文段相关联是有帮助的)。该定时器的过期间隔是 Timeoutinterval¹

第二个主要事件是超时。TCP 通过重传引起超时的报文段来响应超时事件。然后 TCP 重启定时器。

TCP 发送方必须处理的第三个主要事件是，到达一个来自接收方的确认报文段(ACK)(更确切地说，是一个包含了有效 ACK 字段值的报文段)

超时间隔加倍

在大多数 TCP 实现中所做的一些修改。首先关注的是在定时器时限过期后超时间隔的长度。在这种修改中，每当超时事件发生时，如前所述，TCP 重传具有最小序号的还未被确认的报文段。只是每次 TCP 重传时都会将下一次的超时间隔设为先前值的两倍，而不是用从 EstimatedRTT 和 DevRTT 推算出的值

¹由 EstimatedRTT 和 DevRTT 计算得出



快速重传

超时触发重传存在的问题之一是超时周期可能相对较长。当一个报文段丢失时，这种长超时周期迫使发送方延迟重传丢失的分组，因而增加了端到端时延。

发送方通常可在超时事件发生之前通过注意所谓冗余 ACK 来较好地检测到丢包情况。冗余 ACK(*duplicate ACK*) 就是再次确认某个报文段的 ACK，而发送方先前已经收到对该报文段的确认。

事件	TCP 接收方动作
具有所期望序号的按序报文段到达。所有在期望序号及以前的数据都已经被确认	延迟的 ACK。对另一个按序报文段的到达最多等待 500ms。如果下一个按序报文段在这个时间间隔内没有到达，则发送一个 ACK
具有所期望序号的按序报文段到达。另—个按序报文段等待 ACK 传输	立即发送单个累积 ACK，以确认两个按序报文段
比期望序号大的失序报文段到达。检测出间隔	立即发送冗余 ACK，指示下一个期待字节的序号（其为间隔的低端的序号）
能部分或完全填充接收数据间隔的报文段到达	倘若该报文段起始于间隔的低端，则立即发送 ACK

图 3.15: 产生 TCP ACK 的建议 [RFC 5681]

因为发送方经常一个接一个地发送大量的报文段，如果一个报文段丢失，就很可能引起许多一个接一个的冗余 ACK。如果 TCP 发送方接收到对相同数据的 3 个冗余 ACK，它把这当作一种指示，说明跟在这个已被确认过 3 次的报文段之后的报文段已经丢失。

一旦收到 3 个冗余 ACK，TCP 就执行快速重传 (fast retransmit)[RFC 5681]，即在该报文段的定时器过期之前重传丢失的报文段。对于采用快速重传的 TCP，可用下列代码片段代替 ACK 收到事件：

```

1 事件：收到ACK，具有ACK字段值y
2     if (y > SendBase) {
3         SendBase=y
4         if (当前仍无任何应答报文段)
5             启动定时器
6     }
7     else /* 快对已经确认的报文段的一个冗余ACK */
8         对y收到的冗余ACK数加1
9         if (对y == 3收到的冗余ACK数)
10            /* TCP快速重传 */
11            重新发送具有序号y的报文段
12
13    break;

```

Listing 3.2: 针对快速重传的机制

3.5.5 流量控制

一条 *TCP* 连接的每一侧主机都为该连接设置了接收缓存。当该 *TCP* 连接收到正确、按序的字节后，它就将数据放入接收缓存。相关联的应用进程会从该缓存中读取数据，但不必是数据刚一到达就立即读取。

TCP 为它的应用程序提供了流量控制服务 (flow control service) 以消除发送方使接收方缓存溢出的可能性。流量控制因此是一个速度匹配服务，即发送方的发送速率与接收方应用程序的读取速率相匹配。

TCP 发送方也可能因为 IP 网络的拥塞而被遏制；这种形式的发送方的控制被称为拥塞控制 (congestion control)。

值得注意的是：使流量控制和拥塞控制采取的动作非常相似 (对发送方的遏制)，但是它们显然是针对完全不同的原因而采取的措施。

TCP 通过让发送方维护一个称为接收窗口 (receive window) 的变量来提供流量控制。通俗地说，接收窗口用于给发送方一个指示——该接收方还有多少可用的缓存空间。因为 *TCP* 是全双工通信，在连接两端的发送方都各自维护一个接收窗口。

我们定义以下变量：

- RcvBuffer: 来表示接收缓存大小
- LastByteRead: 主机 B 上的应用进程从缓存读出的数据流的最后一个字节的编号
- LastByteRcvd: 从网络中到达的并且已放入主机 B 接收缓存中的数据流的最后一个字节的编号

由于 *TCP* 不允许已分配的缓存溢出，下式必须成立：

$$\text{LastByteRcvd} - \text{LastByteRead} = \text{RcvBuffer}$$

接收窗口用 rwnd 表示，根据缓存可用空间的数量来设置：

$$\text{rwnd} = \text{RcvBuffer} - [\text{LastByteRcvd} - \text{LastByteRead}]$$



图 3.16: 接收窗口和接收缓存

3.5.6 TCP 链接管理

TCP 链接：

- 1) 客户端向服务器发送特殊 TCP 报文(也就是 SYN 报文段), 此时 SYN 报文段被置 1。然后随机选择初始序号 (*client_isn*)
- 2) 当 SYN 报文段的 IP 数据包到达服务器主机, 则提取出 SYN, 服务器端发送允许链接的报文段¹(此时, SYN 同样包含, 确认号字段为 *client_isn* + 1, 最后服务器选择自己的初始序号 (*server_isn*))
- 3) 在收到 SYNACK 报文段后, 客户给该链接分配缓存和变量。同时, 向服务器发送链接确认的报文 (SYN 置 0)

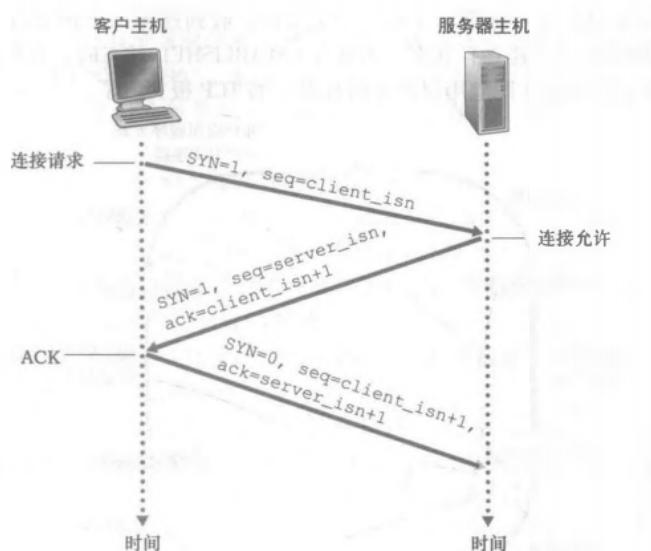


图 3.17: TCP 三次握手：报文段交换

当客户端发送关闭连接命令：

- 1) 客户端发送特殊 TCP 报文段(也就是 FIN 报文段)
- 2) 服务器接收到 FIN 后, 发送确认 ACK 报文段
- 3) 服务器发送 FIN 终止报文段
- 4) 客户端发送 ACK 确认报文段

在一个 TCP 连接的生命周期内, 运行在每台主机中的 TCP 协议在各种 TCP 状态 (TCP state) 之间变迁。

客户 TCP 开始时处于 CLOSED(关闭) 状态。客户的应用程序发起一个新的 TCP 连接。这引起客户中的 TCP 向服务器中的 TCP 发送一个 SYN 报文段。在发送过 SYN 报文段后, 客户 TCP 进入了 SYN_SENT 状态。

当客户 TCP 处在 SYN_SENT 状态时, 它等待来自服务器 TCP 的对客户所发报文段进行确认且 SYN 比特被置为 1 的一个报文段。收到这样一个报文段之后, 客户 TCP 进入 ESTABLISHED(已建立) 状态。

¹允许链接的报文段被称为 SYNACK 报文段 (SYNACK segment)



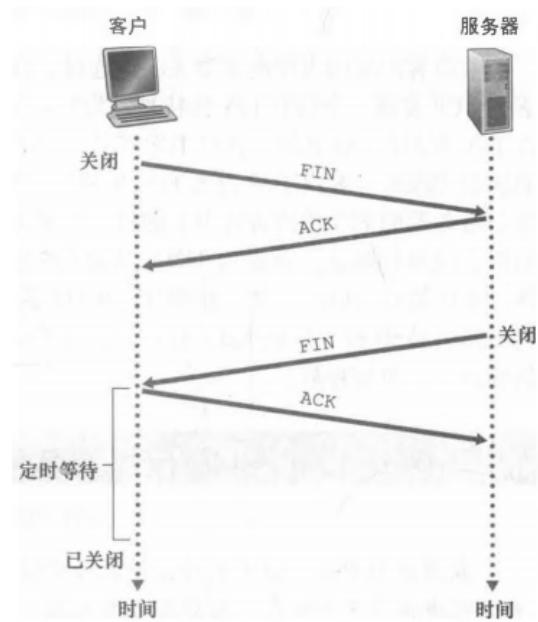


图 3.18: TCP 关闭连接

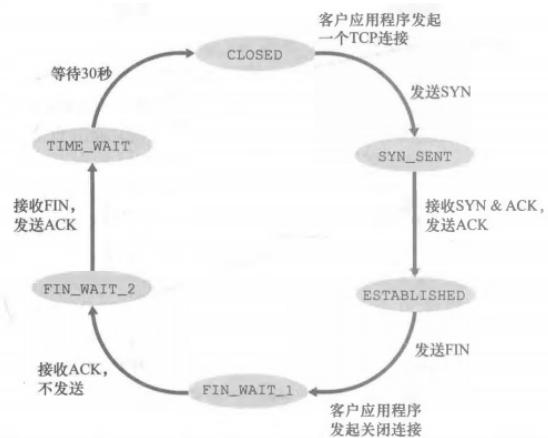


图 3.19: 客户 TCP 经历的典型 TCP 状态序列

当处在 ESTABLISHED 状态时，TCP 客户就能发送和接收包含有效载荷数据（即应用层产生的数据）的 TCP 报文段了。

假设客户应用程序决定要关闭该连接。这引起客户 TCP 发送一个带有 FIN 比特被置为 1 的 TCP 报文段，并进入 FIN_WAIT_1 状态。

当处在 FIN_WAIT_1 状态时，客户 TCP 等待一个来自服务器的带有确认的 TCP 报文段。当它收到该报文段时，客户 TCP 进入 FIN_WAIT_2 状态。

当处在 FIN_WAIT_2 状态时，客户等待来自服务器的 FIN 比特被置为 1 的另一个报文段；当收到该报文段后，客户 TCP 对服务器的报文段进行确认，并进入 TIME_WAIT 状态。

假定 ACK 丢失，TIME_WAIT 状态使 TCP 客户重传最后的确认报文。在 TIME_WAIT 状态中所消耗的时间是与具体实现有关的，而典型的值是 30 秒、1 分钟或 2 分钟。经过等待后，连接就正式关闭，客户端所有资源（包括端口号）将被释放。

对于服务器 TCP 需要经历的流程：

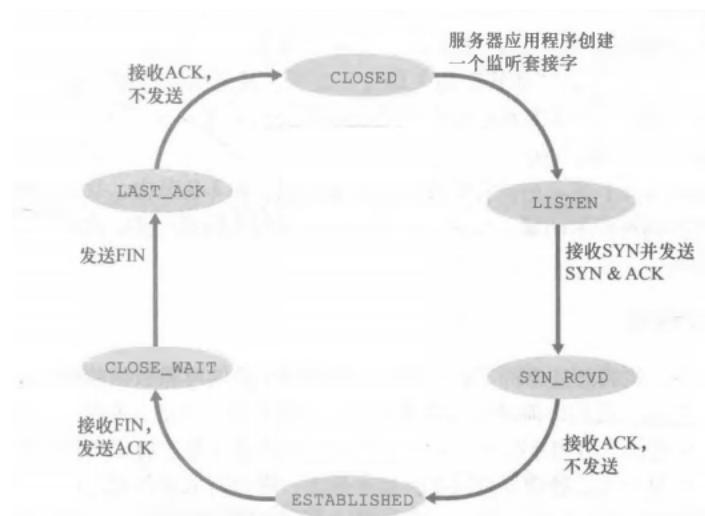


图 3.20：服务器端 TCP 经历的典型 TCP 状态序列

3.6 拥塞控制原理

在实践中，这种丢包一般是当网络变得拥塞时由于路由器缓存溢出引起的。分组重传因此作为网络拥塞的征兆（某个特定的运输层报文段的丢失）来对待，但是却无法处理导致网络拥塞的原因，因为有太多的源想以过高的速率发送数据。为了处理网络拥塞原因，需要一些机制以在面临网络拥塞时遏制发送方



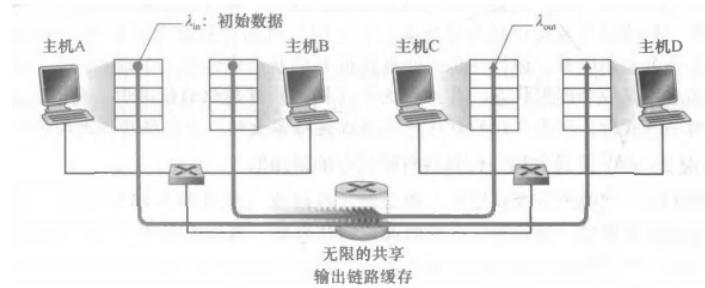


图 3.21: 拥塞情况 1: 两条连接共享具有无限大缓存的单条路由

3.6.1 拥塞原因与代价

情况一：两个发送方和一台具有无穷大缓存的路由器

假设主机 A 以 λ_{in} 字节/秒的平均速率发送数据到连接中，也就意味着主机 A 向路由器提供流量的速率是 λ_{in} 字节/秒。假设 B 也是如此，在一段容量为 R 的共享式输出链路上传输。因此，我们假设路由器有无限大的缓存空间。

下图展示了主机 A 的连接性能

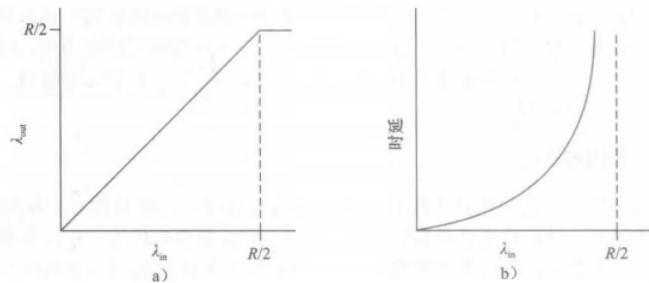


图 3.22: 拥塞情况 1: 吞吐量、时延与主机发送速率的函数关系

可以看见，在发送速率为 $0 \sim R/2$ 之间时，接收方的吞吐量等于发送方的发送速率；然而当越过 $R/2$ 时，吞吐量只能到达 $R/2$ ，无论如何也不会超过。

并且，在无限接近 $R/2$ 时，平均时延会趋向于无穷大，因此：即当分组的到达速率接近链路容量时，分组经历巨大的排队时延

情况 2：两个发送方和一台具有有限缓存的路由器

在情况一的基础上进行修正：从无限缓存变为有限。当分组到达一个已满的缓存时会被丢弃。其次，我们假定每条连接都是可靠的。如果一个包含有运输层报文段的分组在路由器中被丢弃，那么它终将被发送方重传。

我们再次以 λ_{in} 字节/秒表示应用程序将初始数据发送到套接字中的速率。运输层向网络中发送报文段（含有初始数据或重传数据）的速率用 λ'_{in} ；字节/秒表示。 λ'_{in} 有时被称为网络的供给载荷 (offered load)。

在情况 2 下实现的性能强烈地依赖于重传的方式。



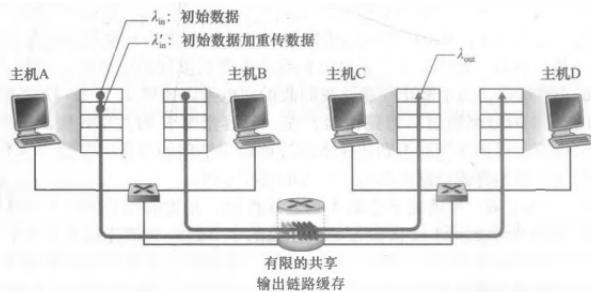


图 3.23: 情况 2: (有重传的) 两台主机与一台拥有有限缓存的路由器

首先考虑一种假设：主机 A 能够确定路由器中的缓存是否有空闲，因此只会在空闲时发送分组。这种情况下不会产生丢包并且 $\lambda_{in} = \lambda'_{in}$ ，连接的吞吐量就等于 λ_{in} 。如下图中图 a 所示

于是考虑一种更为真实的情况：发送方仅当确认了一个分组丢失才会重传。由图 b 所示，发送方必须执行重传以补偿因为缓存溢出而丢弃（丢失）的分组

最后，我们考虑：发送方也许会提前发送超时并重传在队列中已被推迟但未丢失的分组。如图 c 所示，发送方在遇到大时延时所进行的不必要重传会引起路由器利用其链路带宽来转发不必要的分组副本

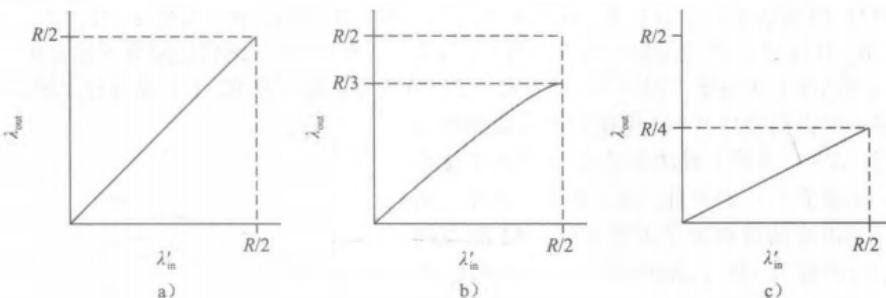


图 3.24: 具有有限缓存时情况 2 的性能

3.6.2 拥塞控制方法

在最为宽泛的级别上，我们可根据网络层是否为运输层拥塞控制提供了显式帮助，来区分拥塞控制方法

1) 端到端拥塞控制。

在该方法中，网络层并没有为运输层拥塞控制提供显示支持。

2) 网络辅助的拥塞控制

在该方法中，路由器向发送方提供关于网络中拥塞状态的显示反馈信息

