网络应用数据交换

P2P应用：原理与文件分发

- 纯P2P架构
- 文件分发：客户机/服务器 vs. P2P
  - CS
  - 为什么是这样的？不应该传送和发出难道是并行的？？？
  - P2P
- P2P文件分发典型例子：BitTorrent（bit激流）
- 思考题

P2P应用：索引技术

- P2P: 搜索信息
- 集中式索引
- 全分布非结构：洪泛式查询（Query flooding）
  - 如何查询？
- 层次式覆盖网络
- P2P案例应用：Skype（最成功的之一）
- 课后作业

Socket编程API

- 网络程序设计接口主要类型
- 理解应用编程接口API
- 几种典型的应用编程接口

Socket编程-Socket API概述

- Socket API
  - 如何标识socket？
- 对内的Socket抽象
- socket地址结构

Socket编程-Socket API函数

- Socket API函数（以WinSock为例）
  - WSAStartup
  - WSACleanup
  - socket
  - Socket面向TCP/IP的服务类型
  - closesocket（linux版为close）
  - bind（服务器）
  - listen（服务器）
  - connect（客户端，tcp/udp差异）
  - accept（服务器，tcp）
  - send, sendto（服务器客户机都用）
  - recv, recvfrom（服务器客户机都用）
  - setsockopt, getsockopt（不具体讲了）
- Socket API函数小结
  - win独占
  - 多平台通用
- 关于网络字节顺序
- 网络应用的Socket API(TCP)调用基本流程

Socket编程-客户端软件设计

- 解析服务器IP地址
- 解析服务器（熟知）端口号
- 解析协议号
- TCP客户端软件流程
- UDP客户端软件流程

网络应用数据交换

P2P应用：原理与文件分发

纯P2P架构

Peer-to-peer 没有服务器 任意端系统之间直接通信 节点阶段性接入Internet 节点可能更换IP地址

文件分发：客户机/服务器 vs. P2P

问题：从一个服务器向N个节点分发一个文件需要多长时间?

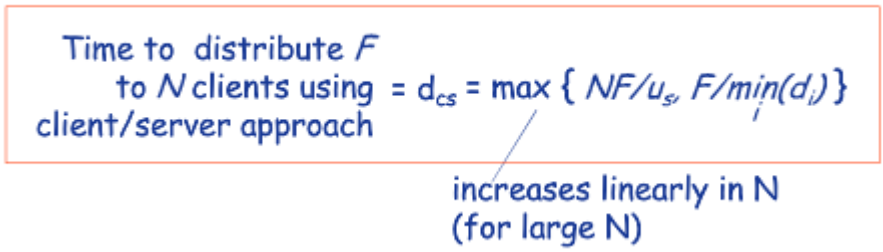
以下速度计算全部是建立在假设：因特网核心速度无限制;服务器客户端带宽全部被运用到文件传输。而且全部是一个下限！

us: 服务器上传带宽（upload） ui: 节点i的上传带宽 di: 节点i的下载带宽

CS

 服务器串行地发送N个副本 时间： NF/us 客户机i需要F/di时间下载

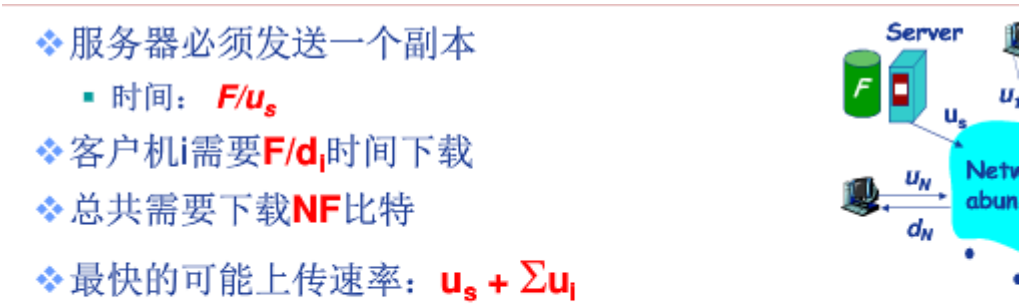
对cs，可以看到时间在N大的时候是和N呈线性的！



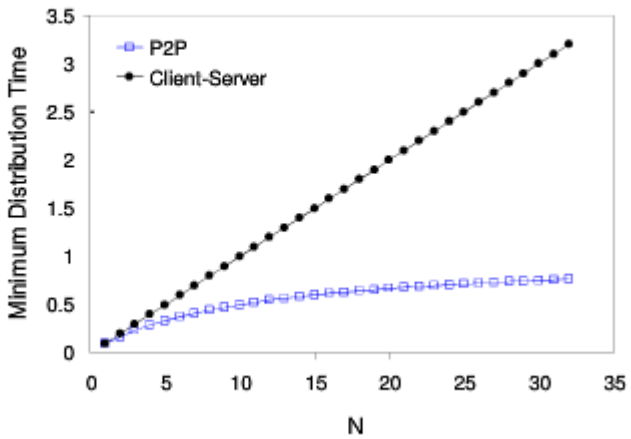
为什么是这样的？不应该传送和发出难道是并行的？？？

P2P

□服务器必须发送一个副本 □ 时间： F/us □客户机i需要F/di时间下载 □总共需要下载NF比特 □最快的可能上传速率：us + □ui

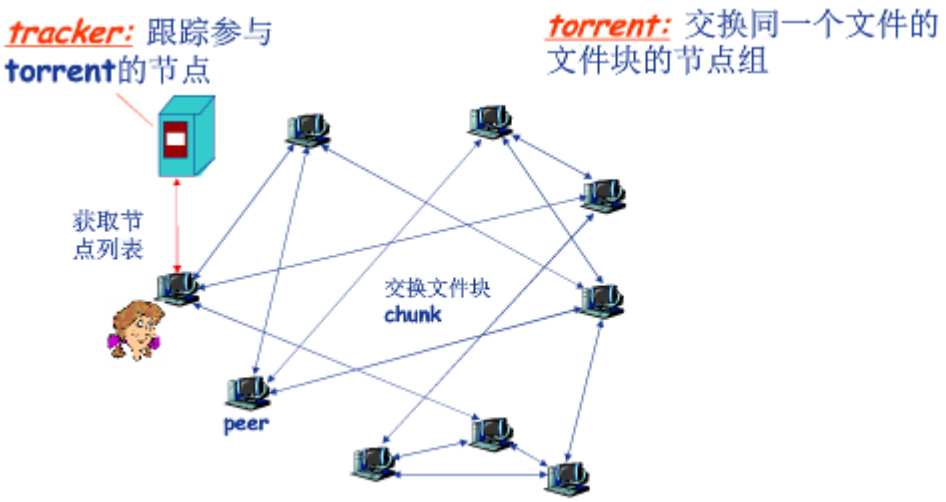


客户端上传速率= u， F/u = 1小时， u_s = 10u， $d_{min} \geq u_s$



这解释了p2p的自拓展性：对等方除了是bit的消费者还是重新分发者。

P2P文件分发典型例子：BitTorrent（bit激流）



tracker: 跟踪参与torrent的节点，每个torrent都有一个tracker

torrent: 交换同一个文件的文件块的节点组

- 文件划分为256KB的chunk（块）
- 节点加入torrent
  - 没有chunk，但是会逐渐积累
  - 向tracker注册以获得节点清单，与某些节点（“邻居”）建立连接
- 下载的同时，节点需要向其他节点上传chunk（从网络下载的或者是节点自身有的）
- 节点自由的加入或离开
- 一旦节点获得完整的文件，它可能（自私地）离开或（无私地）留下
- 获取chunk
  - 给定任一时刻，不同的节点持有文件的不同chunk集合
  - 节点(Alice)定期查询每个邻居所持有的chunk列表
  - 请求原则（请求获取缺失的chunk）
    - 稀缺优先的请求建立原则：先去拿自己没有的chunk中最稀少的
- 发（送原则）
  - tit-for-tat（以牙还牙）
    - Alice向4个邻居发送chunk：正在向Alice发送Chunk的邻居中速率最快的4个，每10秒重新评估top 4
      - **上传速率高，则能够找到更好的交易伙伴，从而更快地获取文件。**
    - 每30秒随机选择一个其他节点，向其发送chunk，则alice可能成为chunk的top4，从而建立新的连接。

思考题

BitTorrent技术对网络性能有哪些潜在的危害？

这个欠考虑。

> BT三大指控：高温、重复读写、扇区断块。

> Bittorrent下载是宽带时代新兴的P2P交换文件模式，各用户之间共享资源，互相当种子和中继站，俗称BT下载。由于每个用户的下载和上传几乎是同时进行，因此下载的速度非常快。不过，开发BT的人因为缺乏对维护硬盘的考虑，使用了很差的HASH算法,它会将下载的数据直接写进硬盘(不像FlashGet等下载工具可以调整缓存，到指定的数据量后才写入硬盘)，因此造成硬盘损害，提早结束硬盘的寿命。
>
> 此外，BT下载事先要申请硬盘空间，在下载较大的文件的时候，一般会有2~3分钟时间整个系统优先权全部被申请空间的任务占用，其他任务反应极慢。有些人为了充分利用带宽，还会同时进行几个BT下载任务，此时就非常容易出现由于磁盘占用率过高而导致的死机故障。
>
> 因为BT对硬盘的重复读写动作会产生高温，令硬盘的温度升高，直接影响硬盘的寿命。而当下载人数愈多，同一时间读取你的硬盘的人亦愈多，硬盘大量进行重复读写的动作，加速消耗。基于对硬盘工作原理的分析可以知道，硬盘的磁头寿命是有限的，频繁的读写会加快磁头臂及磁头电机的磨损，频繁的读写磁盘某个区域更会使该区温度升高，将影响该区磁介质的稳定性还会导至读写错误，高温还会使该区因热膨涨而使磁头和碟面更近了（正常情况下磁头和碟面只有几个微米，高温膨胀会让磁头更靠近碟面），而且也会影响薄膜式磁头的数据读取灵敏度，会使晶体振荡器的时钟主频发生改变，还会造成硬盘电路元件失灵。任务繁多也会导至ide硬盘过早损坏，由于ide硬盘自身的不足，过多任务请求是会使寻道失败率上升导至磁头频繁复位（复位就是磁头回复到 0磁道，以便重新寻道）加速磁头臂及磁头电机磨损。因此有些人形容，BT就像把单边燃烧的柴枝折开两、三段一起燃烧，大量的读写动作会大大加速硬盘的消耗，燃烧硬盘的生命。
>
> 其次，同时因为下载太多东西，使扇区的编排混乱，读写数据时要在不同扇区中读取，增加读写次数，加速硬盘消耗。 2、对网络带宽的损害
>
> 当前，以BitTorrent(以下简称BT)为代表的P2P下载软件流量占用了宽带接入的大量带宽，据统计已经超过了50%。这对于以太网接入等共享带宽的宽带接入方式提出了很大的挑战，大量的使接入层交换机的端口长期工作在线速状态，严重影响了用户使用正常的Web、E-mail以及视频点播等业务，并可能造成重要数据无法及时传输而给企业带来损失。因此，运营商、企业用户以及教育等行业的用户都有对这类流量进行限制的要求。BT将会占用太多的网络资源，从而有可能在接入网、传输网、骨干网等不同层面形成瓶颈，造成资源紧张，这似乎也是目前运营商包括网通、长宽等封掉BT端口的最大理由。
>
> 3、助长了病毒的传播
>
> 2005年11月17日，公安部公共信息网络安全监察处许剑卓处长在天津AVAR2005大会上做了《中国网络犯罪现状》的报告，报告指出，通过计算机病毒和木马进行的黑客行为是计算机网络犯罪的主要根源。调查情况表明，计算机病毒除了通过常规的电子邮件等途径传播外，目前网络上盛行的P2P软件成为计算机病毒和木马传播的主要途径。这些病毒和木马对企业的安全形成巨大的挑战。 4、可能面临着版权侵害的风险
>
> Fred Lawrence是一个美国普通老人，今年67岁，因为自己孙子的缘故惹来了美国电影协会（MPAA）的大麻烦。Lawrence的孙子通过iMesh P2P服务在家中的电脑下载并分享了4部电影，美国电影协会通过IP地址找到了他和他的电脑，并以侵犯版权为由要求老人为此在18个月中付出4000美元的罚金……； 现在国内外都在严厉打击盗版，不排除版权作者或机构通过各种网络跟踪技术来找到非法进行P2P下载的用户，并提起诉讼或者其他赔偿要求；如果企业员工进行了这些行为，可能由此对企业的形象造成极大负面影响，并可能使得企业遭受其他损失。此外，员工可能通过BT等下载一些色情、反动、暴力的等违法的信息，这些信息可能被公安机关检测到，由此可能给员工和企业带来法律风险。

P2P应用：索引技术

P2P: 搜索信息

- P2P系统的索引：信息到节点位置(IP地址+端口号)的映射

- 文件共享(电驴)
  - 利用索引**动态跟踪、存储**节点所共享的文件的位置
  - 节点需要告诉索引它拥有哪些文件
  - 节点搜索索引，从而获知能够得到哪些文件
- 即时消息(QQ)
  - 索引负责将用户名映射到位置
  - 当用户开启IM应用时，需要通知索引它的位置
  - 节点检索索引，确定用户的IP地址

集中式索引

- Napster最早采用这种设计
  - 1) 节点加入时，通知中央服务器：
    - IP地址
    - 内容
  - 2) Alice查找“Hey Jude”
  - 3) Alice从Bob处请求文件
- 集中式索引的问题
  - 单点失效问题
  - 性能瓶颈
  - 版权问题（集中目录容易被发现版权问题）

全分布非结构：洪泛式查询（Query flooding）

- 完全分布式架构
- 每个节点对它共享的文件进行索引，且只对它共享的文件进行索引

如何查询？

由覆盖网络(overlay network): Graph来组织查询和连接

- 节点X与Y之间如果有TCP连接，那么构成一个边
- 所有的活动节点和边构成覆盖网络
  - 边：虚拟链路
- 节点一般邻居数少于10个？？？

在覆盖网络上进行广播来完成查询。

- 查询消息通过已有的TCP连接发送
- 节点转发查询消息
- 如果查询命中，则利用反向路径发回查询节点

- 问题
  - 信息泛滥，带宽压力，无效的垃圾信息

- 网络形成之初，难以形成网络，而成为各个独立分支，造成信息缺失。需要特殊手段处理

层次式覆盖网络

- □ 介于集中式索引和洪泛查询之间的方法
- □ 每个节点或者是一个超级节点，或者被分配一个超级节点
  - □ 节点和超级节点间维持TCP连接（集中式索引）
  - □ 某些超级节点对之间维持TCP连接（超级节点之间是洪泛式）
- □ 超级节点负责跟踪子节点的内容

P2P案例应用：Skype（最成功的之一）

□ 通话时是P2P的：用户/节点对之间直接通信。但是索引阶段是层次式覆盖网络架构

□ 私有应用层协议

□ **采用层次式覆盖网络架构**

□ **索引负责维护用户名与IP地址间的映射**

□ **索引分布在超级节点上**

课后作业

查阅Skype应用的相关资料，就其架构、协议、算法等撰写一篇调研报告，长度在5000字以上。

Socket编程API

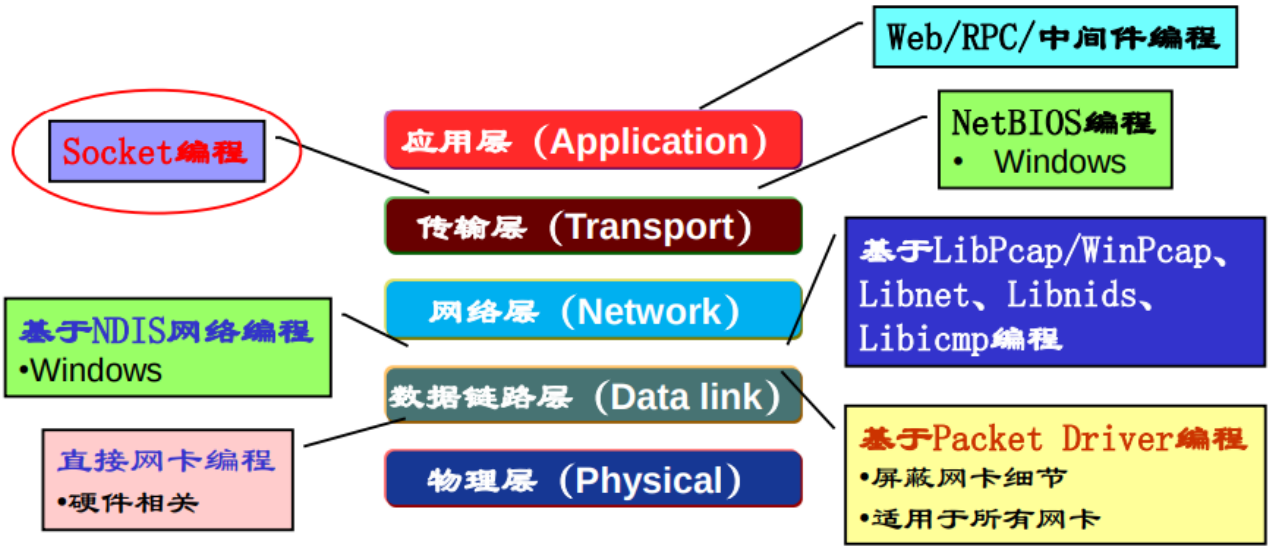
网络程序设计接口主要类型

开发网络应用程序关联的API类型。

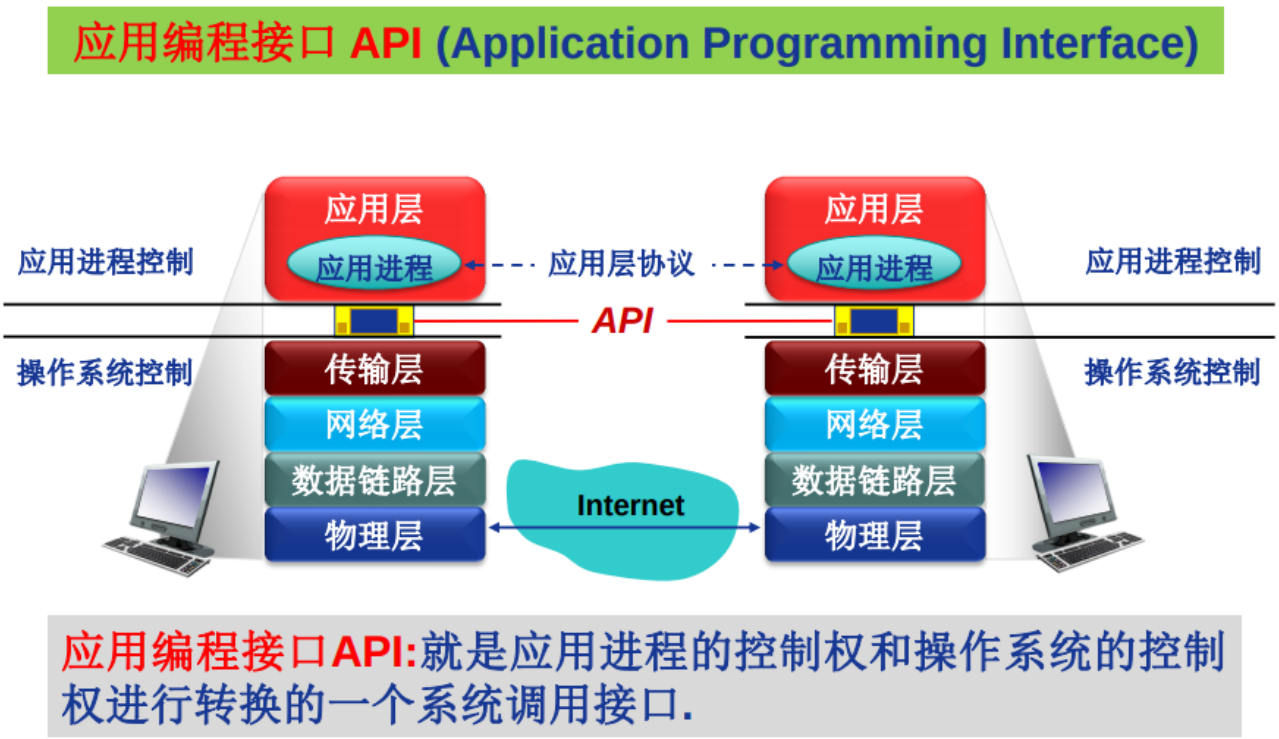
按5层结构观察，除物理层外，所有层次，包括应用层本身，都能提供网络程序设计的api。

- 直接面向网卡编程-大部分不需要也难以掌握。**网卡是数据链路层**
- 网卡之上的，数据链路层的编程，屏蔽网卡细节，适用所有网卡。

- 特定操作系统的开发api。
- 基于库的。
- socket：应用层、传输层之间的。**属于传输层**



理解应用编程接口API



- 应用层协议组构应用进程之间的逻辑连接。
- API通常是从传输层开始封装。

几种典型的应用编程接口

- Berkeley UNIX 操作系统定义了一种 API，称为**套接字接口(**socket interface)，简称套接字（socket）。
- 微软公司在其操作系统中采用了套接字接口 API，形成了一个稍有不同的 API，并称之为Windows Socket Interface，**WINSOCK**。
- AT&T（美国电话电报公司）为其 UNIX 系统 V 定义了一种 API，简写为 **TLI** (Transport Layer Interface)。
    - **UNIX**，一种计算机操作系统，具有多任务、多用户的特征。于1969年，在美国AT&T公司的贝尔实验室开发类UNIX（UNIX-like）

Socket编程-Socket API概述

Socket API

抽象通信机制。是一种门面模式，为应用层封装传输层协议，为应用层提供抽象链路。

- 最初设计
    - 面向BSD UNIX-Berkley
    - 面向TCP/IP协议栈接口
- 目前
    - Internet网络应用最典型的API接口，事实上的工业标准
    - 绝大多数操作系统都支持
- 通信模型
    - 面向客户/服务器（C/S），**p2p也使用，p2p微观上也是cs**
- **由进程或者操作系统创建**
    - 本质上是操作系统提供api，进程调用该api通知操作系统创建。

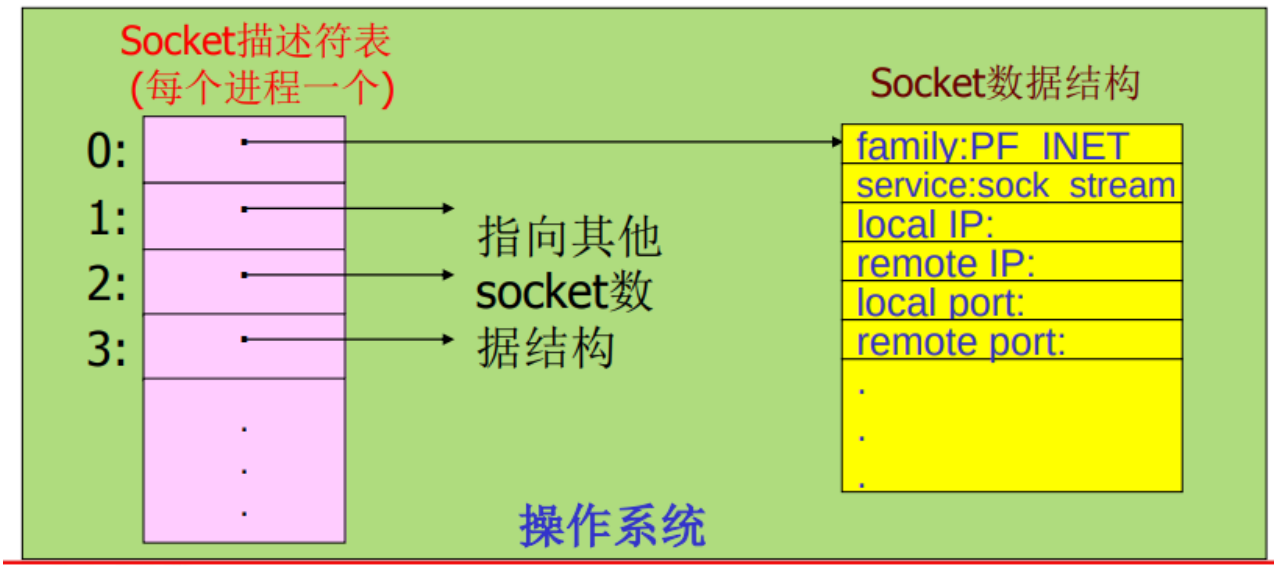
如何标识socket？

- 对外
    - 标识通信端点：
        - IP地址+端口号（**16位整数端口号，0-65535,即16位二进制无符号数**）
- **对内**
    - **操作系统/进程**如何管理套接字（对内）？
        - 套接字描述符（socket descriptor）
            - **小整数**
            - 当应用程序要创建一个套接字时，操作系统就返回一个小整数作为描述符，应用程序则使用这个描述符来引用该套接字。

对内的Socket抽象

- 类似于文件的抽象，像文件一样管理socket

- 当应用进程创建套接字时，**操作系统**分配一个**数据结构存储该套接字相关信息**
- 进程调用api通知操作系统创建套接字，该函数由操作系统返回套接字描述符给进程。
- 都是通过该socket描述符来引用、访问套接字。
- 每一个进程都管理一个soket描述符表，管理其创建的socket，这个表类似一个结构体指针数组，每个指针指向一个socket数据结构。
- **由系统使用来提供api。**



socket地址结构

❖已定义结构*sockaddr_in*:

```
struct sockaddr_in
{
  u_char sin_len;          /*地址长度                    */
  u_char sin_family;       /*地址族(TCP/IP：AF_INET)     */
  u_short sin_port;        /*端口号                      */
  struct in_addr sin_addr; /*IP地址                      */
  char sin_zero[8];        /*未用(置0)                   */
}
```

- **sin：socket internet**
- IP地址、本地端口号这两个必需。
- socket提供多协议支持，不仅仅是TCP/IP
  - 地址族：表述所使用的传输层协议
  - AF_INEF：TCP/IP使用的地址族
    - 只需知道，windows下tcpip要用的地址族是AF_INEF就够了
  - sin_zero是为了让sockaddr与sockaddr_in两个数据结构保持大小相同而保留的空字节。

- 使用TCP/IP协议簇的网络应用程序声明端点地址变量时，使用结构sockaddr_in

> sockaddr与sockaddr_in：
>
> 注释中标明了属性的含义及其字节大小，这两个结构体一样大，都是16个字节，而且都有family属性，不同的是：
>
> sockaddr用其余14个字节来表示sa_data，而sockaddr_in把14个字节拆分成sin_port, sin_addr和sin_zero
>
> 分别表示端口、ip地址。sin_zero用来填充字节使sockaddr_in和sockaddr保持一样大小。
>
> sockaddr和sockaddr_in包含的数据都是一样的，但他们在使用上有区别：
>
> 程序员不应操作sockaddr，sockaddr是给操作系统用的
>
> 程序员应使用sockaddr_in来表示地址，sockaddr_in区分了地址和端口，使用更方便。

思考：为何这里要维护地址长度？

Socket编程-Socket API函数

Socket API函数（以WinSock为例）

winsock实现机制是动态连接库，所以要初始化、释放动态连接库，使用api开始和结束要分别调用

- WSAStartup（初始化Windows Sockets API）
- WSACleanup（释放所使用的WindowsSockets DLL）
- **wsa表示Windows Sockets API**

WSAStartup

使用Socket的应用程序在使用Socket之前**必须首先**调用WSAStartup函数

```
int WSAStartup(WORD wVersionRequested, LPWSADATA lpWSAData);
```

- WORD wVersionRequested
  - 指明程序请求使用的WinSock版本，其中高位字节指明副版本、低位字节指明主版本
  - 十六进制整数，例如0x102表示2.1版（**WORD表示双字节**）
- LPWSADATA lpWSAData
  - 返回实际的WinSock的版本信息，指向WSADATA结构的指针
- 例：使用2.1版本的WinSock的程序代码段

```
int WSAStartup(WORD wVersionRequested, LPWSADATA lpWSAData);
wVersionRequested = MAKEWORD( 2, 1 );
err = WSAStartup( wVersionRequested, &wsaData );
```

WSACleanup

应用程序在完成对请求的Socket库的使用，最后要调用WSACleanup函数。

```
int WSACleanup (void);
```

- 解除与Socket动态库的绑定。
- 释放Socket库所占用的系统资源。

下面的不带wsa的api是多系统通用的，上面的wsa api是win专用的

socket

创建套接字

```
sd = socket(protofamily,type,proto);
```

- sd
  - 操作系统返回的套接字描述符，应用层使用该描述符操作、引用套接字。
- protofamily协议族（说明面向哪个协议族）
  - protofamily = PF_INET（TCP/IP）
- type套接字类型（每种协议族不同，下面的例子是）
  - type = SOCK_STREAM,SOCK_DGRAM or SOCK_RAW（TCP/IP）
- proto(协议号，访问的是哪种协议):0表示缺省，可以使用对应数字表示所选协议族和套接字类型的支持的协议号
- 例：创建一个流套接字的代码段

```
struct protoent *p;
p=getprotobyname("tcp");
SOCKET sd=socket(PF_INET,SOCK_STREAM,p->p_proto);
```

思考：为何这里拥PF——INEF？

Socket面向TCP/IP的服务类型

stream（传输流，使用tcp）：可靠、面向连接、字节流传输、点对点（一个连接只能连接两点）

dgram（datagram数据报，使用udp）：不可靠、无连接、数据报传输

raw（raw：生的，这里指不加传输层处理的原始套接字）

closesocket（linux版为close）

关闭一个描述符为sd的套接字。

```
int closesocket(SOCKET sd);
```

- 如果多个**进**程共享一个套接字，有计数。
  - 调用closesocket将把调用进程的描述符表中的引用删除，然后该socket的数据结构的reference counting减1，减至0才释放该socket数据结构空间。
- 一个进程中的多**线**程对一个套接字的使用无计数

    - 如果进程中的一个线程调用closesocket将一个套接字关闭，则该进程的描述符表中无该socket数据结构的引用，该进程中的其他线程也将不能访问该套接字。
    - **描述符表是一个进程一个，而不是一个线程一个！**
- 返回值
    - 0：成功
    - SOCKET_ERROR：失败

bind（服务器）

绑定（填写）套接字的本地端点地址（IP地址+16进制端口号）

```
int bind(sd,localaddr,addrlen);
```

- 参数:
    - 套接字描述符（sd）
    - 端点地址（localaddr）
        - 结构sockaddr_in
- 客户程序一般不必调用bind函数，因为这个工作一般是由操作系统来完成的。
- 服务器端需要调用
    - 绑定设置熟知端口号：80 25 等等
    - IP地址？服务器运行主机的ip地址可以吗？
    - 绑定问题：服务器应该绑定哪个地址？
    - 解决方案
        - 地址通配符：INADDR_ANY，把ip地址附成该值，表示主机上任意一个有效ip地址都是可以来访问该socket。
        - 服务器应该绑定INADDR_ANY：端口号，而是saddrlen不具体IP：端口号
        - INADDR_ANY一般是0.0.0.0

思考：为什么要加地址长度？localaddr里面已经包含长度

listen（服务器）

置流套接字处于监听状态，listen只用于服务器端，仅面向tcp连接的流类型。

```
int listen(sd,queuesize);
```

- 设置连接请求缓存队列大小（queuesize）
- 返回值：
    - 0：成功
    - SOCKET_ERROR：失败

connect（客户端，tcp/udp差异）

连接建立不等同数据请求发送，还需要send

```
connect(sd,saddr,saddrlen);
```

- 客户程序调用connect函数来使本地套接字（sd）与特定计算机的特定端口（saddr）的套接字（服务）进行连接
- 仅用于客户端
- 可用于TCP客户端也可以用于UDP客户端
    - 用于TCP客户端：建立TCP连接
    - 用于UDP客户端：只是简单的本地指定了服务器端点地址，在之后发送数据报和接受数据时使用该地址。

accept（服务器，tcp）

```
newsock = accept(sd,caddr,caddrlen);
```

- 服务程序调用accept函数从处于监听状态的流套接字sd的客户连接请求队列中取出排在**最前的一个**客户请求，并且**创建一个新的套接字**来与客户套接字创建连接通道。
- 注意这里接**受的是建立连接的请求，而不是对数据的请求**，后者由recv, recvfrom负责，数据由send, sendto发送。
- **仅用于TCP套接字**
- **仅用于服务器**
- 利用新创建的套接字
- 使用新套接字（newsock）与客户通信，why？
    - tcp是点对点的，socket2socket的，单对单的，如果不这么做，就不能并发的提供服务。这里把接受服务和提供服务的socket区分开了。

send, sendto（服务器客户机都用）

```
send(sd,*buf,len,flags);
sendto(sd,*buf,len,flags,destaddr,addrlen);
```

- send函数TCP套接字（客户与服务器）或调用了connect函数的UDP客户端套接字
- sendto函数用于UDP服务器端套接字与未调用connect函数的UDP客户端套接字

recv, recvfrom（服务器客户机都用）

```
recv(sd,*buffer,len,flags);
recvfrom(sd,*buf,len,flags,senderaddr,saddrlen);
```

- recv函数从TCP连接的另一端接收数据，或者从调用了connect函数的UDP客户端套接字接收服务器发来的数据。
- recvfrom函数用于从UDP服务器端套接字与未调用connect函数的UDP客户端套接字接收对端数据。

setsockopt, getsockopt（不具体讲了）

```
int setsockopt(int sd, int level, int optname, *optval, int optlen);
int getsockopt(int sd, int level, int optname,*optval, socklen_t *optlen);
```

- setsockopt()函数用来设置套接字sd的选项参数
- getsockopt()函数用于获取任意类型、任意状态套接口的选项当前值，并把结果存入optval

Socket API函数小结

win独占

- WSAStartup: 初始化socket库(仅对WinSock)
- WSACleanup: 清楚/终止socket库的使用 (仅对WinSock)

多平台通用

按照服务器、客户机、tcp、udp去理解

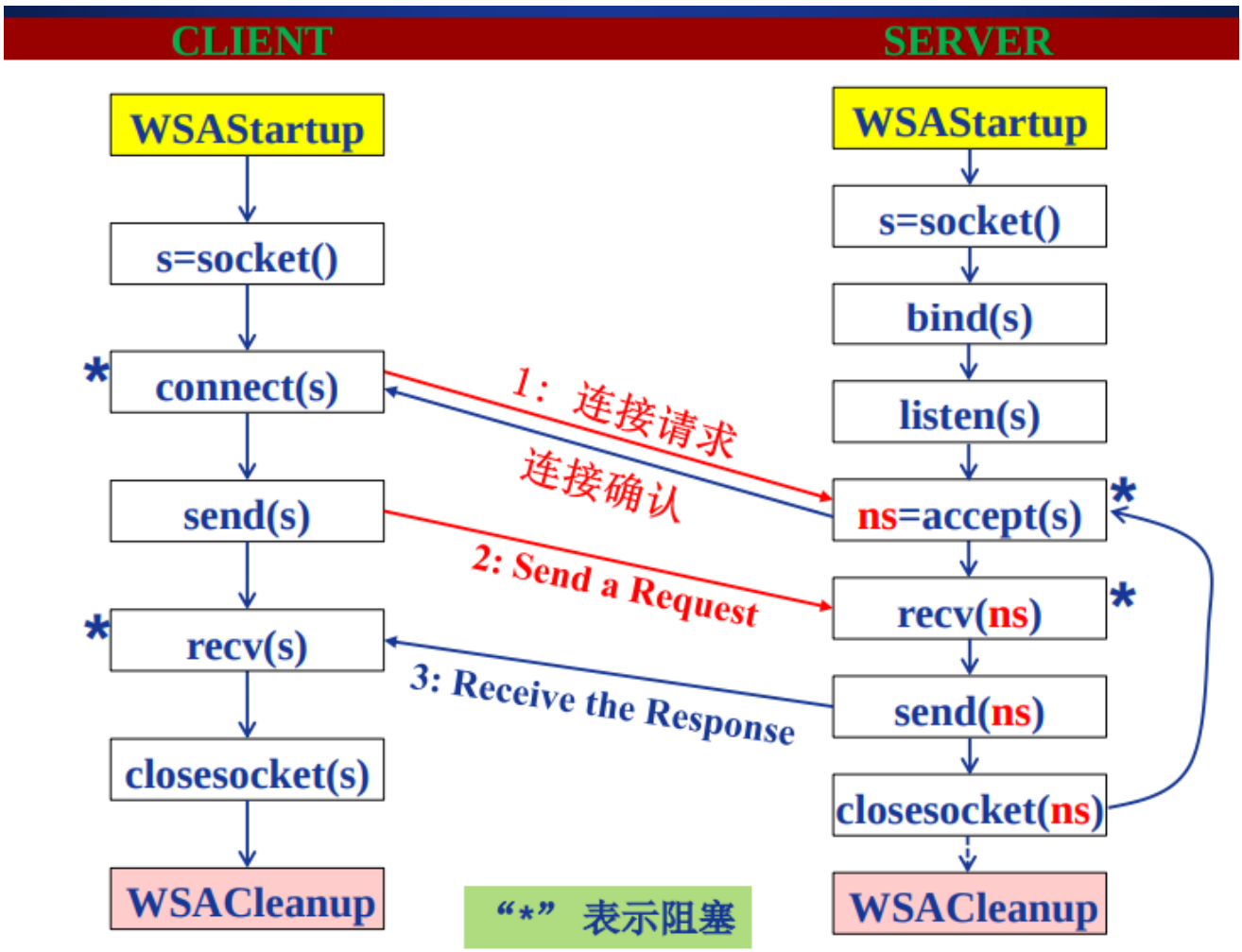
- socket: 创建套接字
- connect:"连接"远端服务器 (仅用于客户端)
- closesocket: 释放/关闭套接字
- bind: 绑定套接字的本地IP地址和端口号（通常客户端不需要）
- listen: 置服务器端TCP套接字为监听模式，并设置队列大小 (仅用于服务器端TCP套接字)
- accept: 接受/提取一个连接请求，创建新套接字，通过新套接 (仅用于服务器端的TCP套接字)
- recv: 接收数据（用于TCP套接字或连接模式的客户端UDP套接字）
- recvfrom: 接收数据报（用于非连接模式的UDP套接字）
- send: 发送数据（用于TCP套接字或连接模式的客户端UDP套接字）
- sendto:发送数据报（用于非连接模式的UDP套接字）
- setsockopt: 设置套接字选项参数（用的时候查）
- getsockopt: 获取套接字选项参数（用的时候查）

关于网络字节顺序

osi7层模型才有表示层来兼容字节顺序，5层中是没有的，需要协议辅助完成该功能。

- TCP/IP定义了标准的用于协议头中的二进制整数表示：网络字节顺序（network byte order）
- 某些Socket API函数的参数需要存储为网络字节顺序而不是本地字节顺序（如IP地址、端口号等）。
- **可以实现本地字节顺序与网络字节顺序间转换的函数**
  - htons: 本地字节顺序→网络字节顺序(16bits)（host2net s）
  - ntohs: 网络字节顺序→本地字节顺序(16bits)
  - htonl: 本地字节顺序→网络字节顺序(32bits)
  - ntohl: 网络字节顺序→本地字节顺序(32bits)

网络应用的Socket API(TCP)调用基本流程



注意，这张图并不完整，注意区分tcp udp

listen并不阻塞，listen只是开启listen状态。

recv、accept是真正对连接的反馈需要循环开启，也是后续的操作的前提，所以要阻塞accept、recv。

阻塞过程，表示当函数未成功则一直等待。

左右两边都有2个阻塞函数。

ns：newsocket

注意两边close的区别，服务器只是关闭了ns。

Socket编程-客户端软件设计

socket服务器按上述流程要：选择服务器ip、端口号，网络字节转化，选择协议。

具体实现上要求：4位ip/域名：服务名转化为32位ip：端口号，协议名转化为服务号

解析服务器IP地址

- 客户端可能使用域名（如:study.163.com）或IP地址（如：123.58.180.121）标识服务器

- IP协议需要使用32位二进制IP地址
- 需要将域名或IP地址转换为32位IP地址
  - 函数inet_addr( ) 实现点分十进制IP地址到32位IP地址转换
    - 如果正确执行将返回一个无符号长整数型数。如果传入的字符串不是一个合法的IP地址，将返回INADDR_NONE。
    - 已经是网络字节顺序
  - 函数gethostbyname( ) 实现域名到32位IP地址转换
    - 返回一个指向结构hostent 的指针,该结构中包含32位ip地址
    - 已经是网络字节顺序

解析服务器（熟知）端口号

- 客户端可能不使用端口号而是使用服务名（如HTTP）标识服务器端口
- 需要将服务名转换为标准的熟知端口号
- 函数getservbyname( )
  - •返回一个指向结构servent的指针，该结构包含端口号

解析协议号

- socket使用协议号来标识协议
- 客户端可能使用协议名（如:TCP）指定协议，需要将协议名转换为协议号（如：6）
- 函数getprotobyname ( ) 实现协议名到协议号的转换
  - 返回一个指向结构protoent的指针

TCP客户端软件流程

1. 确定服务器IP地址与端口号
2. 创建套接字
3. 分配本地端点地址（IP地址+端口号）（**显示的创建socket的同时，系统自动完成本地端点地址分配**）
4. 连接服务器（套接字）
5. 遵循应用层协议进行通信
6. 关闭/释放连接

UDP客户端软件流程

1. 确定服务器IP地址与端口号
2. 创建套接字
3. 分配本地端点地址（IP地址+端口号）
4. 指定服务器端点地址，构造UDP数据报并发送（注意，这里并没有通知服务器，而是直接发送）
5. 遵循应用层协议进行通信
6. 关闭/释放套接字

客户端软件的实现- connectsock()

设计一个connectsock过程封装底层代码，这部分代码在udp和tcp中都可能用到。

```
/* consock.cpp - connectsock */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <winsock.h>
#ifndef INADDR_NONE
#define INADDR_NONE 0xffffffff
#endif /* INADDR_NONE */
void errexit(const char *, ...);
/*------------------------------------------------------------
* connectsock - allocate & connect a socket using TCP or UDP
*------------------------------------------------------------
*/

//transport指的是所用协议名称
SOCKET connectsock(const char *host, const char *service, const char
*transport )
{
struct hostent *phe; /* pointer to host information entry */
struct servent *pse; /* pointer to service information entry */
struct protoent *ppe; /* pointer to protocol information entry */
struct sockaddr_in sin;/* an Internet endpoint address */
int s, type; /* socket descriptor and socket type */
memset(&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;



/* Map service name to port number */
if ( pse = getservbyname(service, transport) )
sin.sin_port = pse->s_port;
else if ( (sin.sin_port = htons((u_short)atoi(service))) == 0 )
errexit("can't get \"%s\" service entry\n", service);
 /* Map host name to IP address, allowing for dotted decimal */
if ( phe = gethostbyname(host) )
memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);
else if ( (sin.sin_addr.s_addr = inet_addr(host))==INADDR_NONE)
errexit("can't get \"%s\" host entry\n", host);
 /* Map protocol name to protocol number */
if ( (ppe = getprotobyname(transport)) == 0)
errexit("can't get \"%s\" protocol entry\n", transport);



/* Use protocol to choose a socket type */
if (strcmp(transport, "udp") == 0)
type = SOCK_DGRAM;
else
type = SOCK_STREAM;
 /* Allocate a socket */
s = socket(PF_INET, type, ppe->p_proto);
if (s == INVALID_SOCKET)
errexit("can't create socket: %d\n", GetLastError());
```

```
 /* Connect the socket */
if (connect(s, (struct sockaddr *)&sin, sizeof(sin))==SOCKET_ERROR)
errexit("can't connect to %s.%s: %d\n", host, service,
GetLastError());
return s;
}
```

客户端软件的实现-UDP客户端

设计connectUDP过程用于创建连接模式客户端UDP套接字

```
/* conUDP.cpp - connectUDP */
#include <winsock.h>
SOCKET connectsock(const char *, const char *, const char *);
/*------------------------------------------------------
 * connectUDP - connect to a specified UDP service
 * on a specified host
 *------------------------------------------------------
*/
SOCKET connectUDP(const char *host, const char *service )
{
return connectsock(host, service, "udp");
}
```

客户端软件的实现-TCP客户端

设计connectTCP过程，用于创建客户端TCP套接字

```
/* conTCP.cpp - connectTCP */
#include <winsock.h>
SOCKET connectsock(const char *, const char *, const char *);
/*------------------------------------------------------
 * connectTCP - connect to a specified TCP service
 * on a specified host
 *------------------------------------------------------
 */
SOCKET connectTCP(const char *host, const char *service )
{
return connectsock( host, service, "tcp");
}
```

客户端软件的实现-异常处理

```
/* errexit.cpp - errexit */
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <winsock.h>
/*------------------------------------------------------------
```

```
* errexit - print an error message and exit
*------------------------------------------------------------
*/
/*VARARGS1*/
void errexit(const char *format, ...)
{ va_list args;
va_start(args, format);
vfprintf(stderr, format, args);
va_end(args);
WSACleanup();
exit(1);}
```

例1：访问DAYTIME服务的客户端（TCP）

DAYTIME服务 获取日期和时间 双协议服务（TCP、 UDP），端口号13 TCP版利用TCP连接请求触发服务（**不需要发送任何数据，只需要发送连接建立请求**） UDP版需要**客户端发送一个数据报**

```
/* TCPdtc.cpp - main, TCPdaytime */
#include <stdlib.h>
#include <stdio.h>
#include <winsock.h>
void TCPdaytime(const char *, const char *);
void errexit(const char *, ...);
SOCKET connectTCP(const char *, const char *);
#define LINELEN 128
#define WSVERS MAKEWORD(2, 0)
/*------------------------------------------------------------
* main - TCP client for DAYTIME service
*------------------------------------------------------------
*/
int main(int argc, char *argv[])
{
char *host = "localhost"; /* host to use if none supplied */
char *service = "daytime"; /* default service port */
WSADATA wsadata;
switch (argc) {
case 1:
host = "localhost";
break;
case 3:
service = argv[2];
/* FALL THROUGH */
case 2:
host = argv[1];
break;
default:
fprintf(stderr, "usage: TCPdaytime [host [port]]\n");
exit(1);
}
if (WSAStartup(WSVERS, &wsadata) != 0)
errexit("WSAStartup failed\n");
TCPdaytime(host, service);
```

```
WSACleanup();
return 0; /* exit */
}
/*--------------------------------------------------------
* TCPdaytime - invoke Daytime on specified host and print results
*--------------------------------------------------------
*/
void TCPdaytime(const char *host, const char *service)
{
char buf[LINELEN+1]; /* buffer for one line of text */
SOCKET s; /* socket descriptor */
int cc; /* recv character count */
 s = connectTCP(host, service);
cc = recv(s, buf, LINELEN, 0);
while( cc != SOCKET_ERROR && cc > 0)
{
buf[cc] = '\0'; /* ensure null-termination */
(void) fputs(buf, stdout);
cc = recv(s, buf, LINELEN, 0);
}
closesocket(s);
}
```

例2：访问DAYTIME服务的客户端（UDP）

```
/* UDPdtc.cpp - main, UDPdaytime */
#include <stdlib.h>
#include <stdio.h>
#include <winsock.h>
void UDPdaytime(const char *, const char *);
void errexit(const char *, ...);
SOCKET connectUDP(const char *, const char *);
#define LINELEN 128
#define WSVERS MAKEWORD(2, 0)
#define MSG “what daytime is it?\n"
/*--------------------------------------------------------
* main - UDP client for DAYTIME service
*--------------------------------------------------------
*/
int main(int argc, char *argv[])
{
char *host = "localhost"; /* host to use if none supplied */
char *service = "daytime"; /* default service port */
WSADATA wsadata;
switch (argc) {
case 1:
host = "localhost";
break;
case 3:
service = argv[2];
/* FALL THROUGH */
case 2:
```

```
host = argv[1];
break;
default:
fprintf(stderr, "usage: UDPdaytime [host [port]]\n");
exit(1);
}
if (WSAStartup(WSVERS, &wsadata) != 0)
errexit("WSAStartup failed\n");
UDPdaytime(host, service);
WSACleanup();
return 0; /* exit */
}
/*------------------------------------------------------
* UDPdaytime - invoke Daytime on specified host and print results
*------------------------------------------------------
*/
void UDPdaytime(const char *host, const char *service)
{
char buf[LINELEN+1]; /* buffer for one line of text */
SOCKET s; /* socket descriptor */
int n; /* recv character count */
s = connectUDP(host, service);
(void) send(s, MSG, strlen(MSG), 0);
/* Read the daytime */
n = recv(s, buf, LINELEN, 0);
if (n == SOCKET_ERROR)
errexit("recv failed: recv() error %d\n", GetLastError());
else
{
buf[cc] = '\0'; /* ensure null-termination */
(void) fputs(buf, stdout);
}
closesocket(s);
return 0; /* exit */
}
```

Socket编程-服务器软件设计

4种类型基本服务器

循环：同时处理一个用户的请求

并发：同时处理多个用户请求

无连接：基于udp

面向连接：基于tcp

1. 循环无连接(Iterative connectionless)服务器
2. 循环面向连接(Iterative connection-oriented)服务器

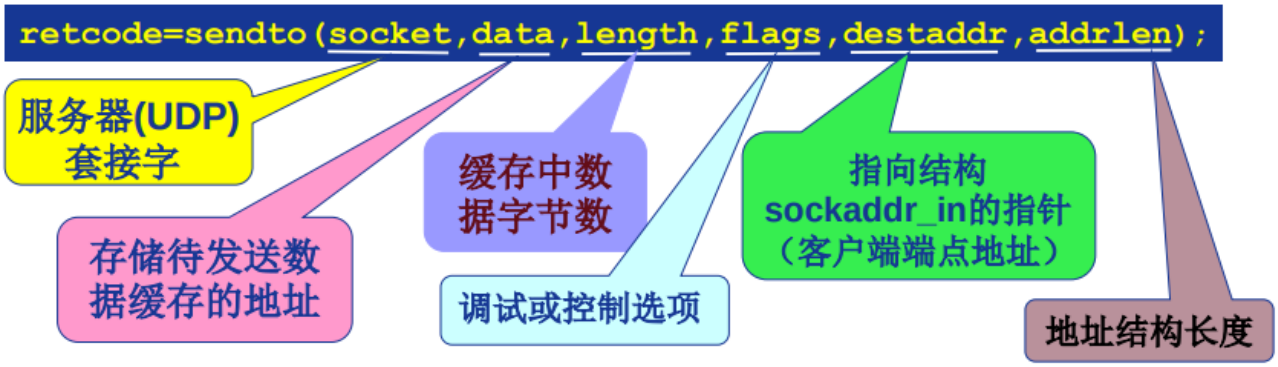
3. 并发无连接(Concurrent connectionless)服务器
4. 并发面向连接(Concurrent connection-oriented)服务器

循环无连接服务器基本流程

1. 创建套接字
2. 绑定端点地址（**INADDR_ANY+端口号**）
3. 反复接收来自客户端的请求
4. 遵循应用层协议，构造响应报文，发送给客户

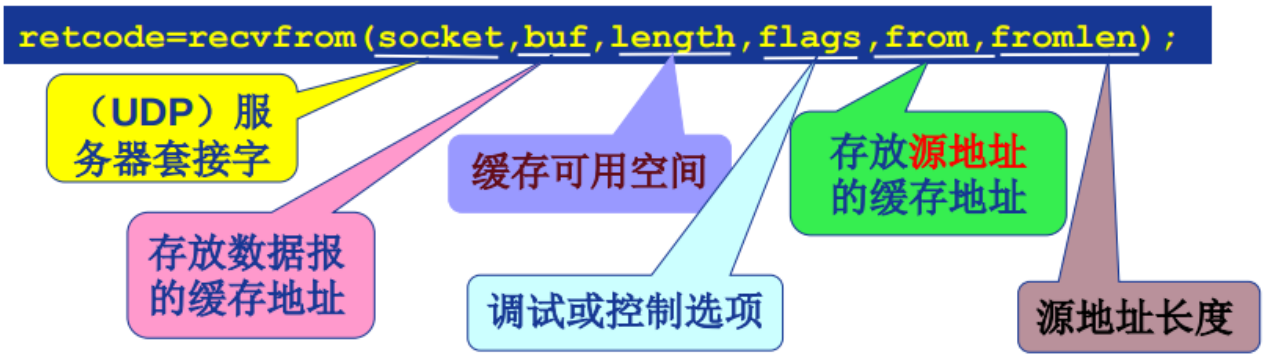
数据发送

 服务器端不能使用connect()函数，connect()是客户端专用 无连接服务器使用sendto()函数发送数据报



获取客户端点地址

 调用recvfrom()函数接收数据时，自动提取客户进程端点地址



循环面向连接服务器基本流程

1. 创建（主）套接字，并绑定熟知端口号；
2. 设置（主）套接字为被动监听模式，准备用于服务器；
3. 调用accept()函数接收下一个连接请求（**通过主套接字**），创建**新套接字**用于与该客户建立连接；（**主套接字要用与接受连接申请**）
4. 遵循应用层协议，反复接收客户请求，构造并发送响应(通过新套接字)；

5. 完成为特定客户服务后，关闭与该客户之间的连接，返回步骤3.

并发无连接服务器基本流程

这里123指的是第一步第二步

主线程1: 创建套接字，并绑定熟知端口号； 主线程2: 反复调用recvfrom()函数，接收下一个客户请求并创建**新线程处理（为了并发）**该客户响应； 子线程1: 接收一个特定请求； 子线程2: 依据应用层协议构造响应报文，并调用sendto()发送； 子线程3: 退出(一个子线程处理一个请求后即终止)。

并发面向连接服务器基本流程

这里123指的是第一步第二步

主线程1: 创建（主）套接字，并绑定熟知端口号；

主线程2: 设置（主）套接字为被动监听模式，准备用于服务器；

主线程3: 反复调用**accept()函数**接收下一个连接请求（通过主套接字），并创**建一个新子线程处理**该客户响应；

子线程1: 接收一个客户的服务请求（通过新**创建的套接字**）；

子线程2: 遵循应用层协议与特定客户进行交互；

子线程3: 关闭/释放连接并退出（线程终止）.

服务器的实现

实现的一种设计，可以不这么设计。

 设计一个底层过程隐藏底层代码：passivesock() 两个高层过程分别用于创建服务器端UDP套接字和TCP套接字（调用passivesock()函数）： passiveUDP() passiveTCP()

服务器的实现-passivesock()

```
/* passsock.cpp - passivesock */
#include <stdlib.h>
#include <string.h>
#include <winsock.h>
void errexit(const char *, ...);
/*------------------------------------------------------------------------
* passivesock - allocate & bind a server socket using TCP or UDP
*------------------------------------------------------------------------
*/
SOCKET passivesock(const char *service, const char *transport, int qlen)
{
struct servent *pse; /* pointer to service information entry */
struct protoent *ppe; /* pointer to protocol information entry */
struct sockaddr_in sin;/* an Internet endpoint address */
SOCKET s; /* socket descriptor */
int type; /* socket type (SOCK_STREAM, SOCK_DGRAM)*/
memset(&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
```

```
sin.sin_addr.s_addr = INADDR_ANY;
 /* Map service name to port number */
if ( pse = getservbyname(service, transport) )
sin.sin_port = (u_short)pse->s_port;
else if ( (sin.sin_port = htons((u_short)atoi(service))) == 0 )
errexit("can't get \"%s\" service entry\n", service);
/* Map protocol name to protocol number */
if ( (ppe = getprotobyname(transport)) == 0)
errexit("can't get \"%s\" protocol entry\n", transport);
 /* Use protocol to choose a socket type */
if (strcmp(transport, "udp") == 0)
type = SOCK_DGRAM;
else
type = SOCK_STREAM;
 /* Allocate a socket */
s = socket(PF_INET, type, ppe->p_proto);
if (s == INVALID_SOCKET)
errexit("can't create socket: %d\n", GetLastError());
 /* Bind the socket */
if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) == SOCKET_ERROR)
errexit("can't bind to %s port: %d\n", service,
GetLastError());
if (type == SOCK_STREAM && listen(s, qlen) == SOCKET_ERROR)
errexit("can't listen on %s port: %d\n", service,
GetLastError());
return s;}


```

服务器的实现-passiveUDP()

```
/* passUDP.cpp - passiveUDP */
#include <winsock.h>
SOCKET passivesock(const char *, const char *, int);
/*------------------------------------------------------------------------
---
* passiveUDP - create a passive socket for use in a UDP server
*------------------------------------------------------------------------
--
*/
SOCKET passiveUDP(const char *service)
{
return passivesock(service, "udp", 0);
}
```

服务器的实现-passiveTCP()

```
/* passTCP.cpp - passiveTCP */
#include <winsock.h>
SOCKET passivesock(const char *, const char *, int);
/*------------------------------------------------------------------------
--
* passiveTCP - create a passive socket for use in a TCP server
*------------------------------------------------------------------------
-
*/
SOCKET passiveTCP(const char *service, int qlen)
{
return passivesock(service, "tcp", qlen);
```

例1：无连接循环DAYTIME服务器

```
/* UDPdtd.cpp - main, UDPdaytimed */
#include <stdlib.h>
#include <winsock.h>
#include <time.h>
void errexit(const char *, ...);
SOCKET passiveUDP(const char *);
#define WSVERS MAKEWORD(2, 0)
/*------------------------------------------------------------------------
* main - Iterative UDP server for DAYTIME service
*------------------------------------------------------------------------
*/
void main(int argc, char *argv[])
{
struct sockaddr_in fsin; /* the from address of a client */
char *service = "daytime"; /* service name or port number */
SOCKET sock; /* socket */
int alen; /* from-address length */
char * pts; /* pointer to time string */
time_t now; /* current time */
WSADATA wsadata;
switch (argc)
{
 case 1:
break;
 case 2:
service = argv[1];
break;
 default:
errexit("usage: UDPdaytimed [port]\n");
}
if (WSAStartup(WSVERS, &wsadata) != 0)
errexit("WSAStartup failed\n");
sock = passiveUDP(service);
while (1)
{
alen = sizeof(struct sockaddr);
```

```
if (recvfrom(sock, buf, sizeof(buf), 0,
 (struct sockaddr *)&fsin, &alen) == SOCKET_ERROR)
errexit("recvfrom: error %d\n", GetLastError());
(void) time(&now);
pts = ctime(&now);
(void) sendto(sock, pts, strlen(pts), 0,
(struct sockaddr *)&fsin, sizeof(fsin));
}
return 1; /* not reached */
}
```

例2：面向连接并发DAYTIME服务器

```
/* TCPdtd.cpp - main, TCPdaytimed */
#include <stdlib.h>
#include <winsock.h>
#include <process.h>
#include <time.h>
void errexit(const char *, ...);
void TCPdaytimed(SOCKET);
SOCKET passiveTCP(const char *, int);
#define QLEN 5
#define WSVERS MAKEWORD(2, 0)
/*------------------------------------------------------------------------
* main - Concurrent TCP server for DAYTIME service
*------------------------------------------------------------------------
*/
void main(int argc, char *argv[])
{
struct sockaddr_in fsin; /* the from address of a client */
char *service = "daytime"; /* service name or port number*/
SOCKET msock, ssock; /* master & slave sockets */
int alen; /* from-address length */
WSADATA wsadata;
switch (argc) {
case1:
break;
case2:
service = argv[1];
break;
default:
errexit("usage: TCPdaytimed [port]\n");
}
if (WSAStartup(WSVERS, &wsadata) != 0)
errexit("WSAStartup failed\n");
msock = passiveTCP(service, QLEN);
while (1) {
alen = sizeof(struct sockaddr);
ssock = accept(msock, (struct sockaddr *)&fsin, &alen);
if (ssock == INVALID_SOCKET)
errexit("accept failed: error number %d\n",
GetLastError());
```

```
if (_beginthread((void (*)(void *)) TCPdaytimed, 0,
 (void *)ssock) < 0) {
errexit("_beginthread: %s\n", strerror(errno));
}
}
return 1; /* not reached */
}
/*------------------------------------------------------------------------
* TCPdaytimed - do TCP DAYTIME protocol
*------------------------------------------------------------------------
*/
void TCPdaytimed(SOCKET fd)
{
char * pts; /* pointer to time string */
time_t now; /* current time */
(void) time(&now);
pts = ctime(&now);
(void) send(fd, pts, strlen(pts), 0);
(void) closesocket(fd);
}
```