TCP/IP UDP Socket Communication programing

TCP/IP/UDP Socket通讯开发实战

2016 深圳

陈超实战系列视频课程

课程大纲

- Linux 音频(alsa)，视频采集(v4l) 编码(x264)。
- Linux 服务器端Socket通讯开发。
- iOS客户端 Socket通讯开发。
- iOS 音视频播放(只提播放供库),详情见另外一个课程。

-iOS/Android/Linux

FFmpeg

QuickTime

主讲人:　陈 超

ubuntu 虚拟机

- 1 安装vmware-tools
- 2 安装开发环境 gcc
  - sudo apt-get update
  - sudo apt-get install vim
  - sudo apt-get install build-essential
  - qtcreator http://download.qt.io/archive/
  - sudo apt-get install libqt4-* libqml*

- QQ群： 541055300
- Email： chenchao_shenzhen@163.com

需要用到的库

- v4l (video for linux).
- sudo apt-get install libv4l-dev
- x264
- apt-get install libx264-dev
- alsa 音频
- sudo apt-get install libalsa-ocaml-dev

V4L

- video4linux(v4l)是一些视频系统，视频软件，音频软件的基础，经常使用在需要采集图像的场合，如视频监控，webcam,可视电话，经常应用在embedded linux中是linux嵌入式开发中经常使用的系统接口。它是linux内核提供给用户空间的编程接口，各种的视频和音频设备开发相应的驱动程序后，就可以通过v4l提供的系统API来控制视频和音频设备，也就是说v4l分为两层，底层为音视频设备在内核中的驱动，上层为系统提供的API，而对于我们来说需要的就是使用这些系统的API。

x264

- H.264是ITU（International Telecommunication Union，国际通信联盟）和MPEG（Motion Picture Experts Group，运动图像专家组）联合制定的视频编码标准。而x264是一个开源的H.264/MPEG-4 AVC视频编码函数库[1]，是最好的有损视频编码器。

x264 特点

- 8x8与4x4自适应空间域转换 自适应B帧选择
- B帧可作为参考帧/自由的帧顺序 CAVLC/CABAC熵编码 自定义精确的矩阵模板
- I帧：所有宏块格式（16x16, 8x8, 4x4, 以及有全部预测的PCM）
- P帧：所有的分割块（从16x16到4x4）
- B帧：分割块从16x16到8x8（包括skip/direct）
- 隔行扫描(MBAFF) 多个参考帧
- 码率控制：固定量化，固定质量，一次或者多次编码的平均码率，可选的VBV参数
- 场景变换检测 B帧时间域、空间域direct模式自适应选择
- 可在多个CPU平行编码
- 预测性的无损编码（x264似乎也是所有基于H.264标准的编码器中唯一实现这项的）
- 心理视觉优化，保留更多的细节（自适应量化，psy-RD，psy-trellis）
- 可用于手动调整码率分配的zones参数

ALSA

- ALSA是Advanced Linux Sound Architecture的缩写，高级Linux声音架构的简称,它在Linux操作系统上提供了音频和MIDI（Musical Instrument Digital Interface，音乐设备数字化接口）的支持

- ALSA是Advanced Linux Sound Architecture，高级Linux声音架构的简称,它在Linux操作系统上提供了音频和MIDI（Musical Instrument Digital Interface，音乐设备数字化接口）的支持。在2.6系列内核中，ALSA已经成为默认的声音子系统，用来替换2.4系列内核中的OSS（Open Sound System,开放声音系统）。[2]

- ALSA的主要特性包括：高效地支持从消费类入门级声卡到专业级音频设备所有类型的音频接口，完全模块化的设计， 支持对称多处理（SMP）和线程安全，对OSS的向后兼容，以及提供了用户空间的alsa-lib库来简化应用程序的开发。

TCP/IP协议

- Transmission Control Protocol/Internet Protocol的简写，中译名为传输控制协议/因特网互联协议，又名网络通讯协议，是Internet最基本的协议、Internet国际互联网络的基础，由网络层的IP协议和传输层的TCP协议组成。TCP/IP 定义了电子设备如何连入因特网，以及数据如何在它们之间传输的标准。协议采用了4层的层级结构，每一层都呼叫它的下一层所提供的协议来完成自己的需求。通俗而言：TCP负责发现传输的问题，一有问题就发出信号，要求重新传输，直到所有数据安全正确地传输到目的地。而IP是给因特网的每一台联网设备规定一个地址。

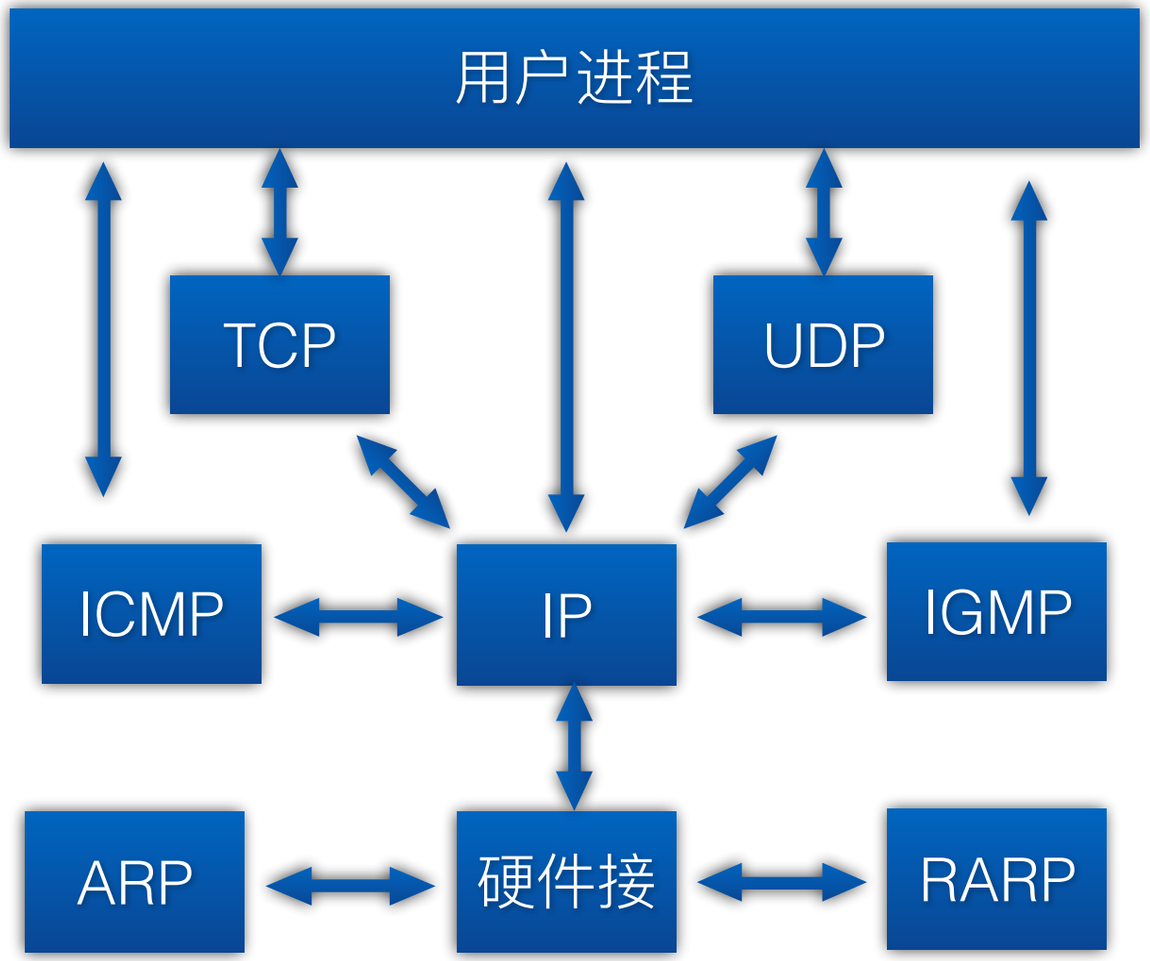
TCP/IP 协议栈

应用层

传输层

网络层

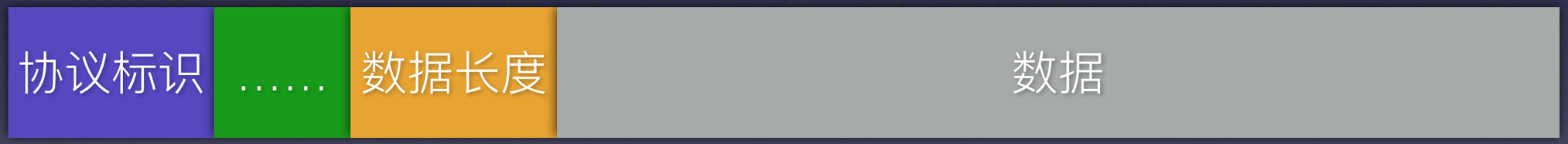
链路层



协议

以太网数据包

用户自定义协议数据包



TCP 服务

尽管TCP和UDP都使用相同的网络层(IP),TCP却向应用层提供与UDP完全不同的服务。 TCP提供一种面向连接的、可靠的字节流服务。

面向连接意味着两个使用TCP的应用(通常是一个客户和一个服务器)在彼此交换数据之前必须先建立一个TCP连接。 这一过程与打电话很相似,先拨号振铃,等待对方摘机说“喂”,然后才说明是谁。在一个TCP连接中,仅有两方进行彼此通信。

TCP通过下列方式来提供可靠性:

- 应用数据被分割成TCP认为最适合发送的数据块。这和UDP完全不同,应用程序产生的数据报长度将保持不变。由TCP传递给IP的信息单位称为报文段或段(segment).
- 当TCP发出一个段后,它启动一个定时器,等待目的端确认收到这个报文段。如果不能及时收到一个确认,将重发这个报文段。(TCP协议中自适应的超时及重传策略)。
- 当TCP收到发自TCP连接另一端的数据,它将发送一个确认。这个确认不是立即发送,通常将推迟几分之一秒。
- TCP将保持它首部和数据的检验和。这是一个端到端的检验和,目的是检测数据在传输过程中的任何变化。如果收到段的检验和有差错,TCP将丢弃这个报文段和不确认收到此报文段(希望发端超时并重发)。
- 既然TCP报文段作为IP数据报来传输,而IP数据报的到达可能会失序,因此TCP报文段的到达也可能会失序。如果必要,TCP将对收到的数据进行重新排序,将收到的数据以正确的顺序交给应用层。
- 既然IP数据报会发生重复,TCP的接收端必须丢弃重复的数据。

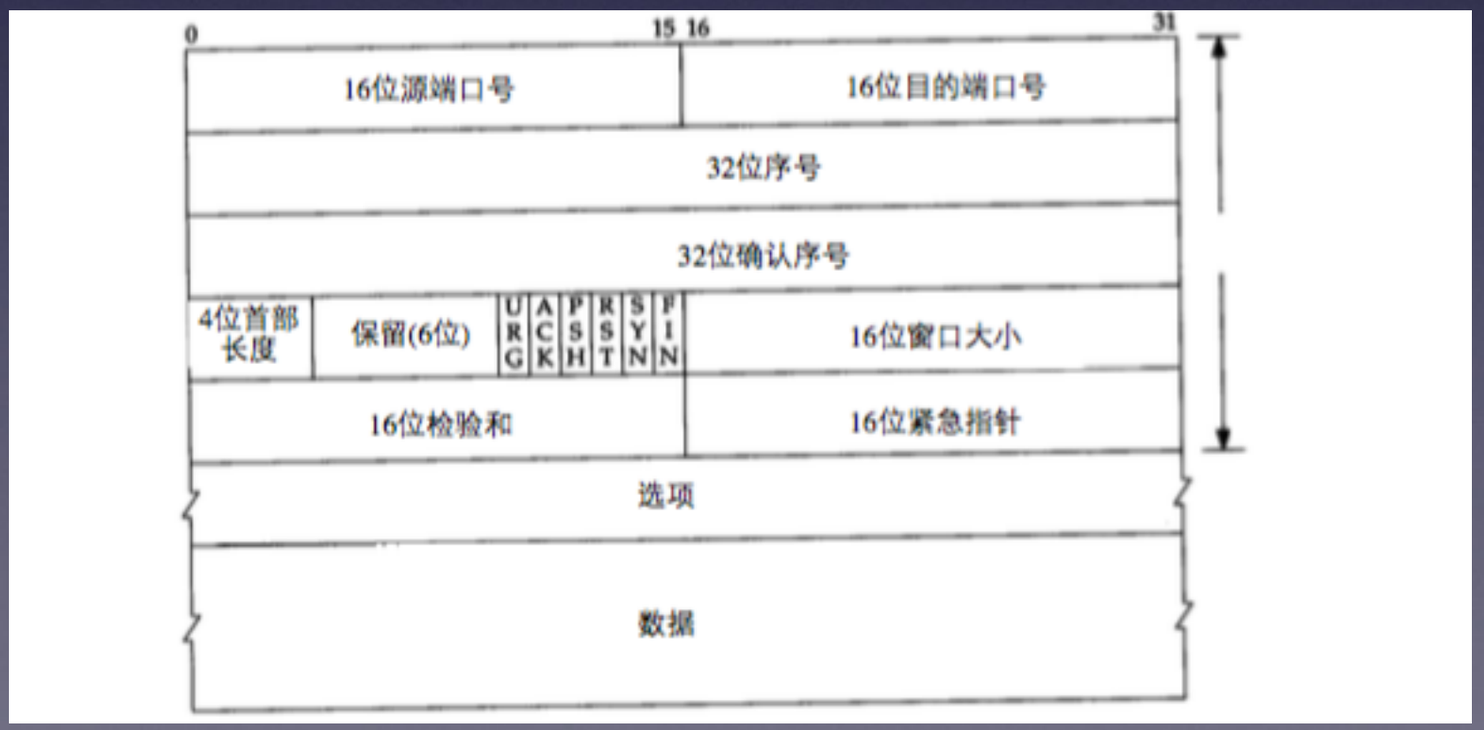
- TCP还能提供流量控制。TCP连接的每一方都有固定大小的缓冲空间。TCP的接收端只允许另一端发送接收端缓冲区所能接纳的数据。这将防止较快主机致使较慢主机的缓冲区溢出。

  两个应用程序通过TCP连接交换8bit字节构成的字节流。TCP不在字节流中插入记录标识符。我们将这称为字节流服务(bytestreamservice)。如果一方的应用程序先传10字节,又传20字节,再传50字节,连接的另一方将无法了解发方每次发送了多少字节。收方可以分4次接收这80个字节,每次接收20字节。一端将字节流放到TCP连接上,同样的字节流将出现在TCP连接的另一端。另外,TCP对字节流的内容不作任何解释。TCP不知道传输的数据字节流是二进制数据,还是ASCII字符、EBCDIC字符或者其他类型数据。对字节流的解释由TCP连接双方的应用层解释。这种对字节流的处理方式与Unix操作系统对文件的处理方式很相似。Unix的内核对一个应用读或写的内容不作任何解释,而是交给应用程序处理。对Unix的内核来说,它无法区分一个二进制文件与一个文本文件。

TCP的首部

TCP数据被封装在一个IP数据报中，

TCP首部的数据格式。如果不计任选字段,它通常是20个字节。



每个TCP段都包含源端和目的端的端口号,用于寻找发端和收端应用进程。这两个值加上IP首部中的源端IP地址和目的端IP地址唯一确定一个TCP连接。

有时,一个IP地址和一个端口号也称为一个插口(socket)。这个术语出现在最早的TCP规范(RFC 793)中,后来它也作为表示伯克利版的编程接口(参见1.15节)。插口对(socketpair)(包含客户IP地址、客户端口号、服务器IP地址和服务器端口号的四元组)可唯一确定互联网络中每个TCP连接的双方。

序号用来标识从TCP发端向TCP收端发送的数据字节流,它表示在这个报文段中的的第一个数据字节。如果将字节流看作在两个应用程序间的单向流动,则TCP用序号对每个字节进行计数。序号是32bit的无符号数,序号到达$2^{32}-1$后又从0开始。

当建立一个新的连接时,SYN标志变1。序号字段包含由这个主机选择的该连接的初始序号ISN(InitialSequenceNumber)。该主机要发送数据的第一个字节序号为这个ISN加1,因为SYN标志消耗了一个序号(将在下章详细介绍如何建立和终止连接,届时我们将看到FIN标志也要占用一个序号)。

既然每个传输的字节都被计数,确认序号包含发送确认的一端所期望收到的下一个序号。因此,确认序号应当是上次已成功收到数据字节序号加1。只有ACK标志(下面介绍)为1时确认序号字段才有效。

发送ACK无需任何代价,因为32bit的确认序号字段和ACK标志一样,总是TCP首部的一部分。因此,我们看到一旦一个连接建立起来,这个字段总是被设置,ACK标志也总是被设置为1。

TCP为应用层提供全双工服务。这意味数据能在两个方向上独立地进行传输。因此,连接的每一端必须保持每个方向上的传输数据序号。

TCP可以表述为一个没有选择确认或否认的滑动窗口协议(滑动窗口协议用于数据传输 将在20.3节介绍)。我们说TCP缺少选择确认是因为TCP首部中的确认序号表示发方已成功收到字节,但还不包含确认序号所指的字节。当前还无法对数据流中选定的部分进行确认。例如,如果1~1024字节已经成功收到,下一报文段中包含序号从2049~3072的字节,收端并不能确认这个新的报文段。它所能做的就是发回一个确认序号为1025的ACK。它也无法对一个 报文段进行否认。例如,如果收到包含1025~2048字节的报文段,但它的检验和错,TCP接收端所能做的就是发回一个确认序号为1025的ACK。在21.7节我们将看到重复的确认如何帮助确定分组已经丢失。

首部长度给出首部中 32 bit字的数目。需要这个值是因为任选字段的长度是可变的。这个 字段占4 bit ,因此TCP最多有60字节的首部。然而,没有任选字段,正常的长度是20字节。

在TCP首部中有6个标志比特。它们中的多个可同时被设置为1。 我们在这儿简单介绍它 们的用法,在随后的章节中有更详细的介绍。

URG ACK PSH RST SYN FIN

紧急指针(urgentpointer)有效(见20.8节)。 确认序号有效。 接收方应该尽快将这个报文段交给应用层。
重建连接。 同步序号用来发起一个连接。这个标志和下一个标志将在第 发端完成发送任务。

TCP的流量控制由连接的每一端通过声明的窗口大小来提供。窗口大小为字节数,起始于确认序号字段指明的值,这个值是接收端正期望接收的字节。窗口大小是一个16bit字段,因而窗口大小最大为65535字节。

检验和覆盖了整个的TCP报文段:TCP首部和TCP数据。这是一个强制性的字段,一定是由发端计算和存储,并由收端进行验证。TCP检验和的计算和UDP检验和的计算相似。

只有当URG标志置1时紧急指针才有效。紧急指针是一个正的偏移量,和序号字段中的值相加表示紧急数据最后一个字节的序号。TCP的紧急方式是发送端向另一端发送紧急数据的一种方式。

最常见的可选字段是最长报文大小,又称为MSS (Maximum Segment Size)。每个连接方通常都在通信的第一个报文段(为建立连接而设置SYN标志的那个段)中指明这个选项。它指明本端所能接收的最大长度的报文段。

TCP报文段中的数据部分是可选的。如果一方没有数据要发送,也使用没有任何数据的首部来确认收到的数据。在处理超时的许多情况中,也会发送不带任何数据的报文段。

TCP的建立和终止

TCP是一个面向连接的协议。无论哪一方向另一方发送数据之前,都必须先在双方之间建立一条连接。本章将详细讨论一个TCP连接是如何建立的以及通信结束后是如何终止的。

这种两端间连接的建立与无连接协议如UDP不同。使用UDP向另一端发送数据报时,无需任何预先的握手。

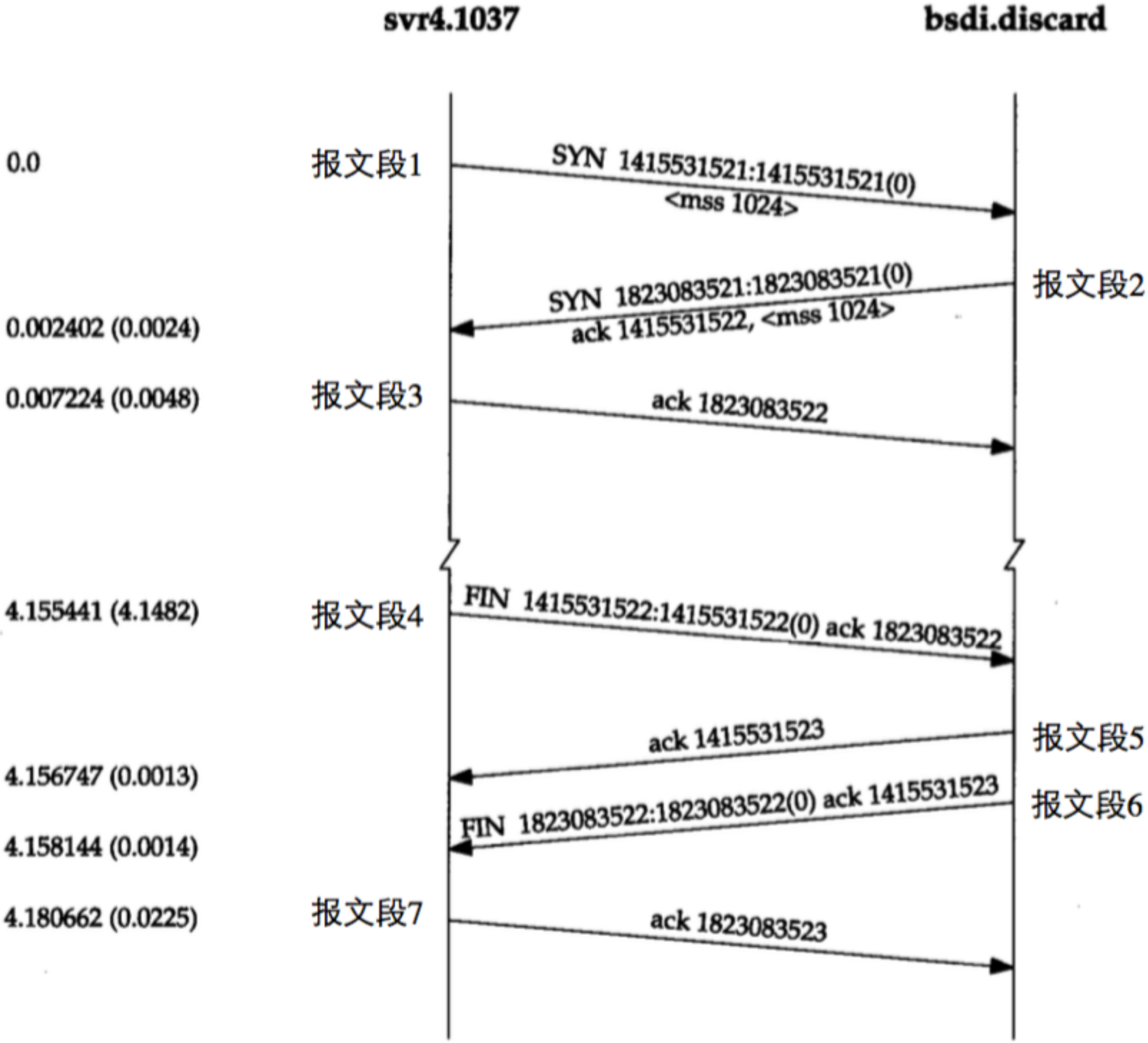
建立一条TCP连接

1) 请求端(通常称为客户)发送一个SYN段指明客户打算连接的服务器的端口,以及初始序号(ISN,在这个例子中为1415531521)。这个SYN段为报文段1。

2) 服务器发回包含服务器的初始序号的SYN报文段(报文段2)作为应答。同时,将确认序号设置为客户的ISN加1以对客户的SYN报文段进行确认。一个SYN将占用一个序号。

3) 客户必须将确认序号设置为服务器的ISN加1以对服务器的SYN报文段进行确认(报文段3)。

这三个报文段完成连接的建立。这个过程也称为三次握手(three-way handshake)。


svr4.1037
bsdi.discard
0.0
报文段1
SYN 1415531521:1415531521(0)
<mss 1024>
SYN 1823083521:1823083521(0)
ack 1415531522, <mss 1024>
报文段2
0.002402 (0.0024)
0.007224 (0.0048)
报文段3
ack 1823083522
4.155441 (4.1482)
报文段4
FIN 1415531522:1415531522(0) ack 1823083522
ack 1415531523
报文段5
4.156747 (0.0013)
FIN 1823083522:1823083522(0) ack 1415531523
报文段6
4.158144 (0.0014)
4.180662 (0.0225)
报文段7
ack 1823083523

发送第一个SYN的一端将执行主动打开(activeopen)。接收这个SYN并发回下一个SYN的另一端执行被动打开(passiveopen)

当一端为建立连接而发送它的SYN时,它为连接选择一个初始序号。ISN随时间而变化,因此每个连接都将具有不同的ISN。RFC793[Postel1981c]指出ISN可看作是一个32比特的计数器,每4ms加1。这样选择序号的目的在于防止在网络中被延迟的分组在以后又被传送,而导致某个连接的一方对它作错误的解释。

如何进行序号选择?在4.4BSD(和多数的伯克利的实现版)中,系统初始化时初始的发送序号被初始化为1。这种方法违背了Host Requirements RFC(在这个代码中的一个注释确认这是一个错误)。这个变量每0.5秒增加64000,并每隔9.5小时又回到0
(对应这个计数器每8 ms加1,而不是每4 ms加1)。另外,每次建立一个连接后,这个 变量将增加64000。

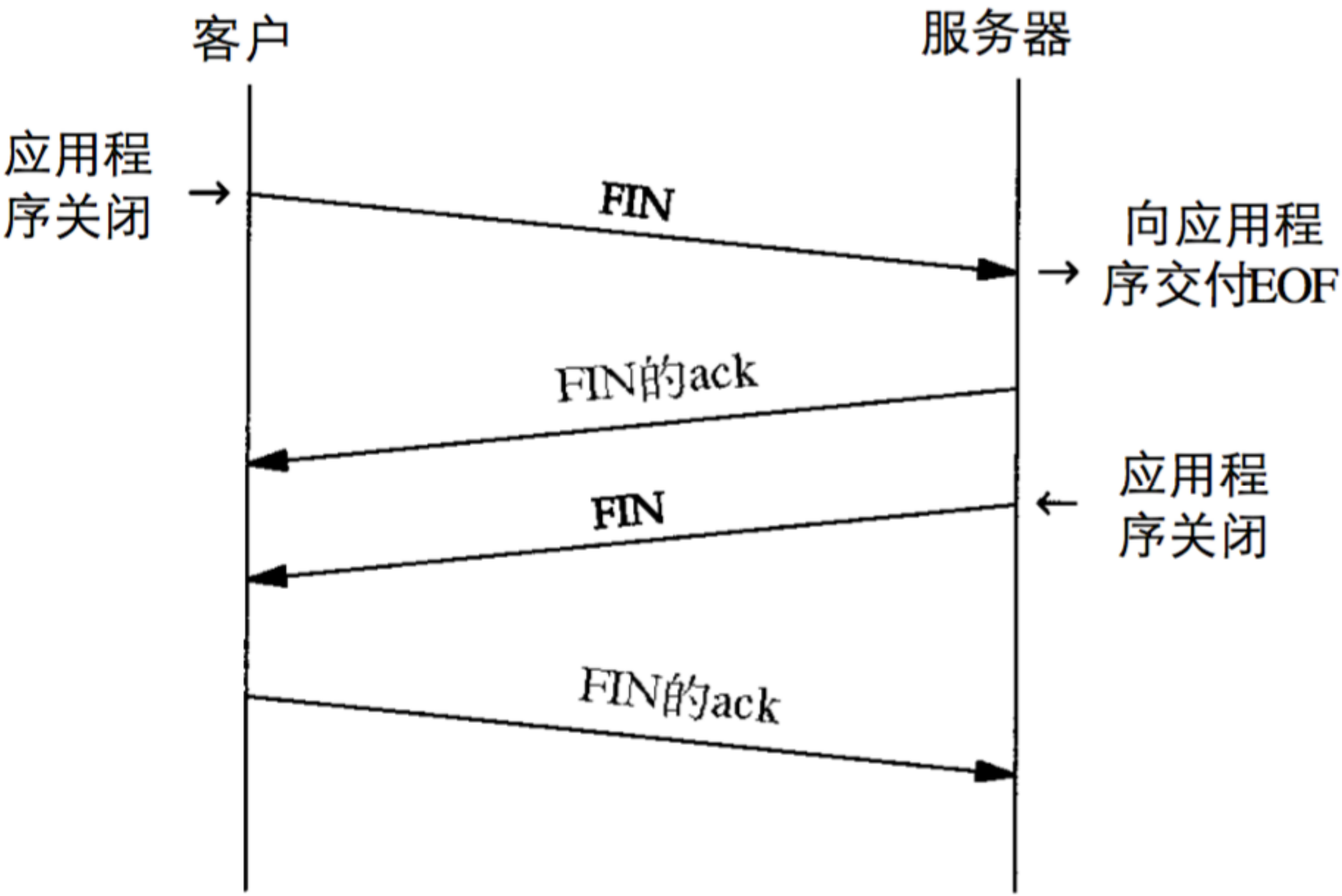
报文段3与报文段4之间4.1秒的时间间隔是建立TCP连接到向telnet键入quit命令来中止该连接的时间。

连接终止协议

建立一个连接需要三次握手,而终止一个连接要经过4次握手。这由TCP的半关闭(half-close)造成的。既然一个TCP连接是全双工(即数据在两个方向上能同时传递),因此每个方向必须单独地进行关闭。这原则就是当一方完成它的数据发送任务后就能发送一个FIN来终止这个方向连接。当一端收到一个FIN,它必须通知应用层另一端几经终止了那个方向的数据传送。发送FIN通常是应用层进行关闭的结果。

收到一个FIN只意味着在这一方向上没有数据流动。一个TCP连接在收到一个FIN后仍能发送数据。而这对利用半关闭的应用来说是可能的,尽管在实际应用中只有很少的TCP应用程序这样做。

首先进行关闭的一方(即发送第一个FIN)将执行主动关闭,而另一方(收到这个FIN)执行被动关闭。


客户
服务器
应用程序关闭
FIN
向应用程序交付EOF
FIN的ack
FIN
应用程序关闭
FIN的ack

当服务器收到这个FIN,它发回一个ACK,确认序号为收到的序号加1(报文段5)。和SYN一样,一个FIN将占用一个序号。同时TCP服务器还向应用程序(即丢弃服务器)传送一个文件结束符。接着这个服务器程序就关闭它的连接,导致它的TCP端发送一个FIN(报文段6),客户必须发回一个确认,并将确认序号设置为收到序号加1(报文段7)。

图显示了终止一个连接的典型握手顺序。我们省略了序号。在这个图中,发送FIN将导致应用程序关闭它们的连接,这些FIN的ACK是由TCP软件自动产生的。

连接通常是由客户端发起的,这样第一个
这个连接(即首先发送FIN)。然而,一般由客户端决定何时终止连接,因为客户进程通常由用户交互控制,用户会键入诸如“quit”一样的命令来终止进程。在图中,我们能改变上边的标识,将左方定为服务器,右方定为客户,一切仍将像显示的一样工作。

TCP其他

滑动窗口算法

超时重传机制

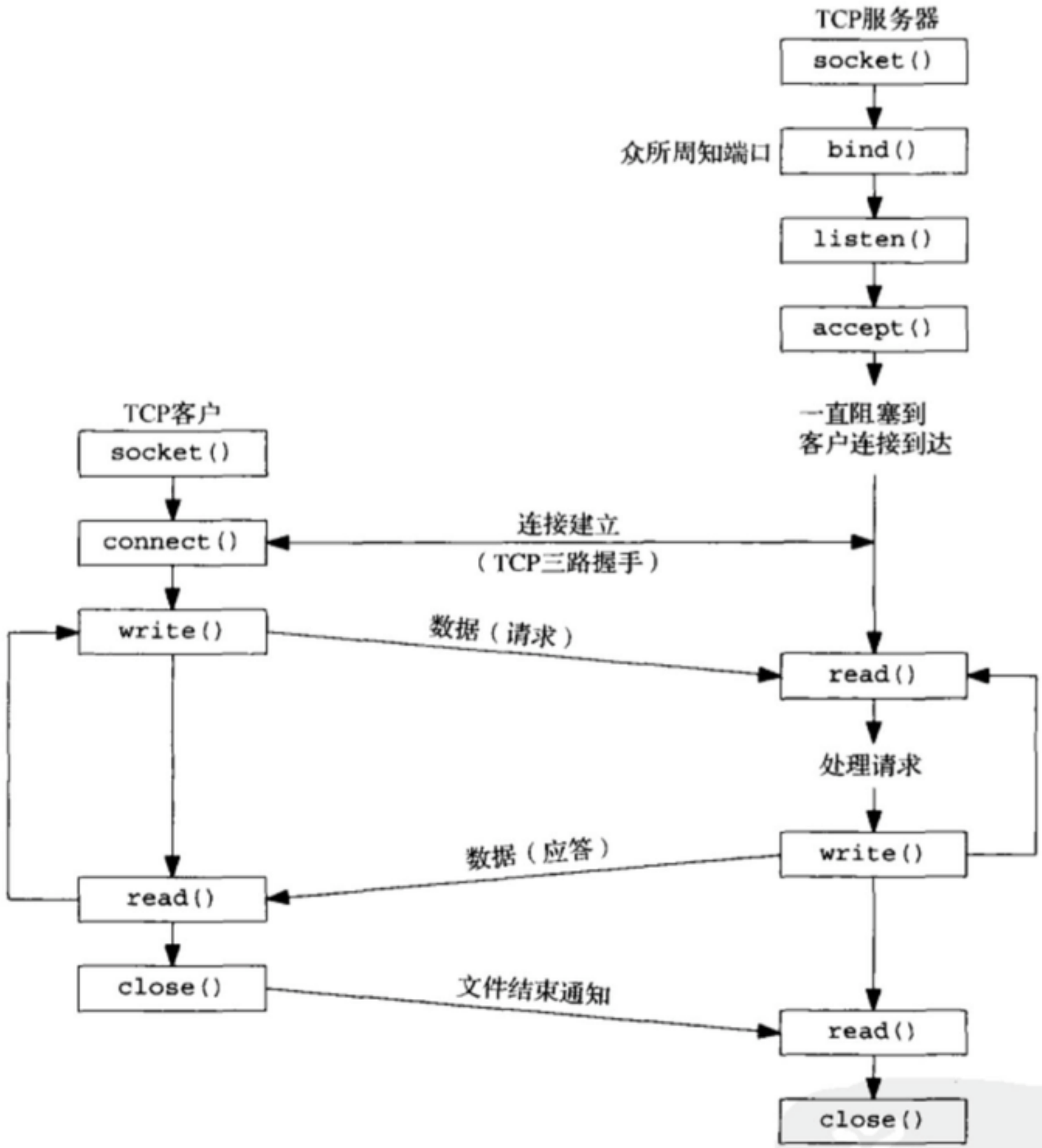
保活定时器

书籍推荐

- TCP/IP协议详解 I II II
- Unix系统高级编程
- C/C++ 编程
- Qt FFmpeg OpenGL iOS android.

Socket编程头文件

- sys/types.h：数据类型定义
- sys/socket.h：提供socket函数及数据结构
- netinet/in.h：定义数据结构sockaddr_in
- arpa/inet.h：提供IP地址转换函数
- netdb.h：　提供设置及获取域名的函数
- sys/ioctl.h：提供对I/O控制的函数
- sys/poll.h：提供socket等待测试机制的函数


TCP服务器
socket()
众所周知端口
bind()
listen()
accept()
一直阻塞到
客户连接到达
TCP客户
socket()
connect()
连接建立
（TCP三路握手）
write()
数据（请求）
read()
处理请求
write()
数据（应答）
read()
close()
文件结束通知
read()
close()

socket函数

为了执行网络I/0, 一个进程必须做的第一件事情就是调用socket函数,指定期望的通信协议类型(使用1Pv4的TCP、使用1Pv6的UDP、Unix域字节流协议等)。

```
#include <sys/socket.h>
int socket (int family, int type, int protocol);
```

返回:若成功则为非负描述符,若出错 返回 - 1

family	说　　明
AF_INET	IPv4协议
AF_INET6	IPv6协议
AF_LOCAL	Unix域协议（见第15章）
AF_ROUTE	路由套接字（见第18章）
AF_KEY	密钥套接字（见第19章）

参数type

type	说　　明
SOCK_STREAM	字节流套接字
SOCK_DGRAM	数据报套接字
SOCK_SEQPACKET	有序分组套接字
SOCK_RAW	原始套接字

参数 protocol

protocol	说　　明
IPPROTO_CP	TCP传输协议
IPPROTO_UDP	UDP传输协议
IPPROTO_SCTP	SCTP传输协议

bind函数

bind函数把一个本地协议地址赋予一个套接字。对于网际网协议,协议地址是32位的IPv4 地址或128位的1Pv6地址与16位的TCP或UDP端口号的组合。

#include <sys/socket. h>

int bind(int sockfd,canst struct sockaddr *myaddr,socklen_t addrlen);

返回:若成功则为0.若出错则为-1

第二个参数是一个指向特定于协议的地址结构的指针,第三个参数是该地址结构的长度。 对于TCP, 调用bind函数可以指定一个端口号,或指定一个IP地址,也可以两者都指定,还可以都不指定。

对于1Pv4来说,通配地址由常值INADDR_ANY 来指定,其值一般为0。它告知内核去选择IP 地址。

struct sockaddr_in servaddr;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);

如此赋值对1Pv4是可行的,因为其IP地址是一个32位的值,可以用一个简单的数字常值表示,对于1Pv6, 我们就不能这么做了,因为128位的1Pv6地址是存放在一个结构中 的。(在C语言中,赋值语句的右边无法表示常值结构。)为了解决这个问题,我们改写为:

```
struct sockaddr_in6 serv;
serv. sin6_addr = in6addr_any;
```

系统预先分配in6addr_any变量并将其初始化为常值IN6ADDR_ANY_INIT。头文件 <netinet/in.h>中含有in6addr_any的extern声明。

listen函数

listen函数仅由TCP服务器调用,它做两件事情。

(l)当socket函数创建一个套接字时,它被假设为一个主动套接字,也就是说,它是一个 将调用connect发起连接的客户套接字。listen函数把一个未连接的套接字转换成一个被动套 接字,指示内核应接受指向该套接字的连接请求。调用listen 导致套接字从CLOSED状态转换到LISTEN状态。

(2)本函数的第二个参数规定了内核应该为相应套接字排队的最大连接个数。

#include <sys/socket.h>

int listen(int sockfd, int backlog);

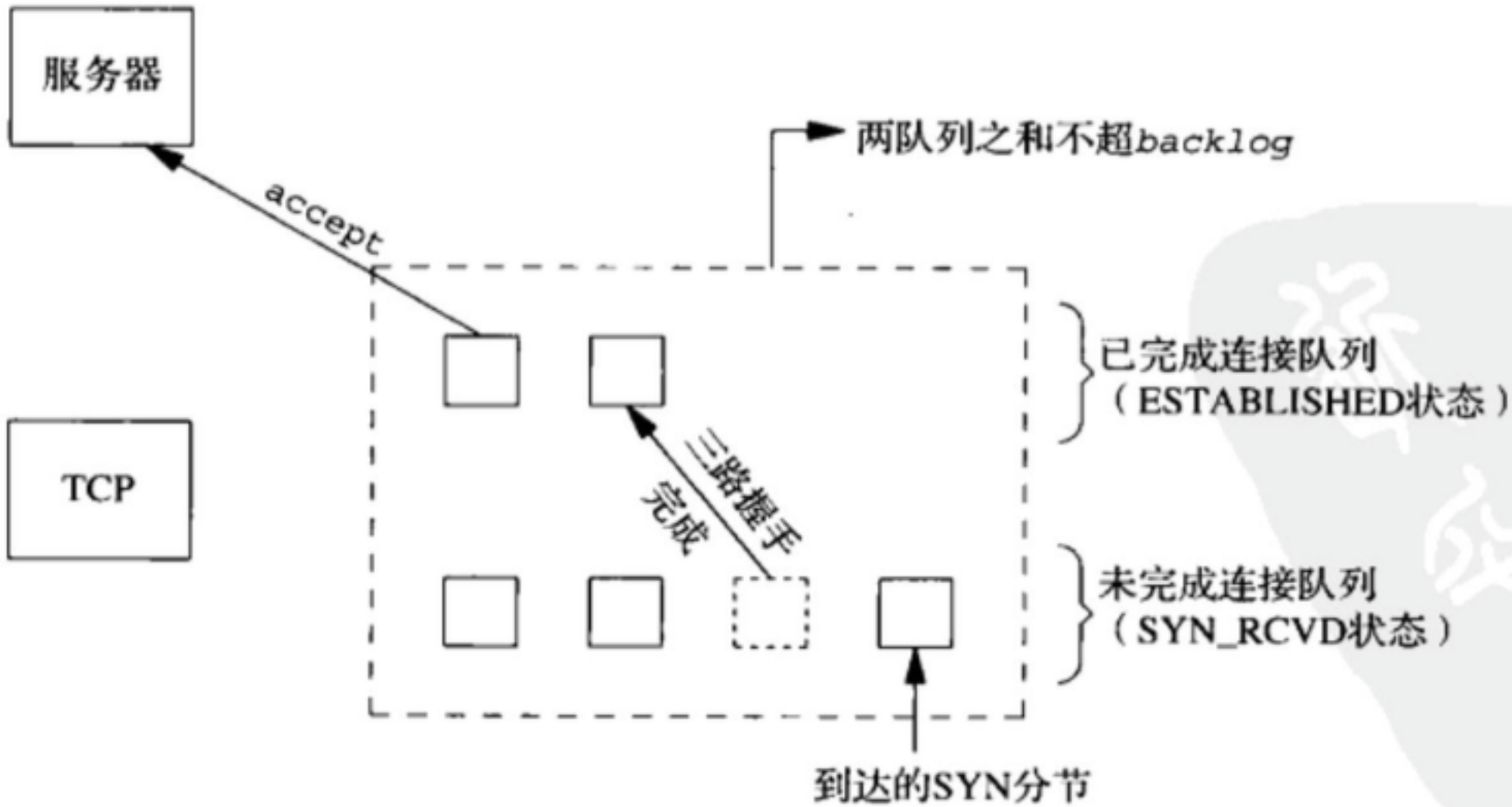
返回:若成功则为0. 若出错则为-1

本函数通常应该在调用socket和bind这两个函数之后,并在调用accept函数之前调用。

为了理解其中的backlog参数,我们必须认识到内核为任何一个给定的监听套接字维护两个队列:

(l)未完成连接队列(incomplete connection queue), 每个这样的SYN分节对应其中一项: 已由某个客户发出并到达服务器,而服务器正在等待完成相应的TCP三路握手过程。这些套接字处于SYN_RCVD状态。

(2)已完成连接队列(completed connectionqueue), 每个已完成TCP三路握手过程的客户对 应其中一项。这些套接字处于ESTABLISHED状态。


服务器
TCP
accept
两队列之和不超backlog
三路握手完成
已完成连接队列
（ESTABLISHED状态）
未完成连接队列
（SYN_RCVD状态）
到达的SYN分节

accept函数

accpet 函数用于从已完成连接队列队头返回下一个已完成连接，如果已完成连接队列为空，那么进程被投入睡眠(假定套接字为默认的阻塞方式)。

#include <sys/socket.h>

int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);

返回:若成功则为非负描述符,若出错则为-1

参数cliaddr和addrlen用来返回已连接的对端进程(客户)的协议地址。addrlen是值-结果 参数:调用前,我们将由*addrlen所引用的整数值置为由cliaddr所指的套接字地址结构 的长度,返回时,该整数值即为由内核存放在该套接字地址结构内的确切字节数。

如果accept成功,那么其返回值是由内核自动生成的一个全新描述符,代表与所返回客户 的TCP连接。

close函数

通常的Unix close函数也用来关闭套接字,并终止TCP连接。

#include <unistd.h>

int close (int sockfd;)

返回:若成功则为o. 若出错则为-1

close一个TCP套接字的默认行为是把该套接字标记成已关闭,然后立即返回到调用进程。该套接字描述符不能再由调用进程使用,也就是说它不能再作为read 的 或 write 的第一个参数,然而TCP将尝试发送已排队等待发送到对端的任何数据,发送完毕后发生的是正常的TCP连接终止序列.

SO_REUSEADDR

一般来说，一个端口释放后会等待两分钟之后才能再被使用，SO_REUSEADDR是让端口释放后立即就可以被再次使用。

SO_REUSEADDR用于对TCP套接字处于TIME_WAIT状态下的socket，才可以重复绑定使用。server程序总是应该在调用bind()之前设置SO_REUSEADDR套接字选项。TCP，先调用close()的一方会进入TIME_WAIT状态.

内核一旦发现进程指定的一个或多个1/0条件就绪(也就是说输入已准备好被 读取,或者描述符已能承接更多的输出),它就通知进程。这个能力称为I/0 复用(I/0 multiplexing), 是由select和poll这两个函数支持的。

select函数

该函数允许进程指示内核等待多个事件中的任何一个发生,并只在有一个或多个事件发生或经历一段指定的时间后才唤醒它。

也就是说,我们调用select告知内核对哪些描述符(就读、写或异常条件)感兴趣以及等 待多长时间。

#include <sys/select.h> #include <sys/tirne.h>

int select(int maxfdpl, fd_set *readset, fd_set* writeset, fd_set* exceptset,const struct timeval* timeout);

返回:若有就绪描述符则为其描述符数目， 若超时则为0, 若出错则为-1

Unix 线程

在传统的UNIX模型中,当一个进程需要另一个实体来完成某事时,它就fork一个子进程去执行处理。Unix上的大多数网络服务器程序就是这么编写的,正如我们在早先讲 解的并发服务器程序例子中看到的那样:父进程accept一个连接,fork一个子进程,该子进程 处理与该连接对端的客户之间的通信。

尽管这种范式多少年来一直用得挺好,fork调用却存在一些问题。• fork是昂贵的。 fork要把父进程的内存映像复制到子进程,并在子进程中复制所有描述

符,如此等等。 当今的实现使用称为写时复制(copy-on-write)的技术,用以避免在子 进程切实需要自己的副本之前把父进程的数据空间复制到子进程。 然而即便有这样的优 化措施,fork仍然是昂贵的。

• fork返回之后父子进程之间信息的传递需要进程间通信(IPC)机制。 调用fork之前父 进程向尚未存在的子进程传递信息相当容易,因为子进程将从父进程数据空间及所有描 述符的一个副本开始运行。 然而从子进程往父进程返回信息却比较费力。

线程有助于解决这两个问题。 线程有时称为轻权进程(lightweight process), 因为线程比进程”权重轻些“。 也就是说,线程的创建可能比进程的创建快10-100倍。

同一进程内的所有线程共享相同的全局内存。这使得线程之间易千共享信息,然而伴随这种简易性而来的却是同步(synchronization)问题。

同一进程内的所有线程除了共享全局变量外还共享:
进程指令;
大多数数据: .

打开的文件(即描述符); .

信号处理函数和信号处置; .

当前工作目录;

用户ID和组ID。 不过每个线程有各自的:

线程ID; .寄存器集合,包括程序计数器和栈指针:

栈(用于存放局部变量和返回地址);

• errno;

.信号掩码; 优先级。

基本线程函数:创建和终止

当一个程序由exec启动执行时,称为初始线程(initial thread)或主线程(main thread)的单个线程就创建了。其余线程则由pthread_create函数创建。

#include <pthread.h>

int. pthread_create (pthread_t *tid, const pthread_attr_t *attr, void* (*fanc) (void *), void *arg);

返回:若成功则为0若. 出错则为正的Exx值.

pthread_join函数

我们可以通过调用pthread_join等待一个给定线程终止。对比线程和UNIX进程,
pthread_create类似千fork, pthread,_join类似waitpid。

#include <pthread.h>

int pthread_join (pthread_t *tid, void **status)

返回:若成功则为o. 若出错则为正的Exxx值

pthread_detach函数

一个线程或者是可汇合的(joinable, 默认值),或者是脱离的(detached)。 当一个可汇合 的线程终止时,它的线程ID和退出状态将留存到另一个线程对它调用pthread_join. 脱离的线 程却像守护进程,当它们终止时,所有相关资源都被释放,我们不能等待它们终止。 如果一个 线程需要知道另一个线程什么时候终止,那就朵好保持第二个线程的可汇合状态。

pthread_detach函数把指定的线程转变为脱离状态。

```
#include <Pthread.h>
int pthread_dectach(pthread_t lid);
```

返回:若成功则为0. 若出错则为正的Exxx值

pthread_exit函数

让一个线程终止的方法之一是调用pthread_exit。

```
#include <pthead.h>
void pthread_exit(void *status);
```

errno

- EAGAIN：套接字已标记为非阻塞，而接收操作被阻塞或者接收超时
- EBADF：sock不是有效的描述词
- ECONNREFUSE：远程主机阻绝网络连接
- EFAULT：内存空间访问出错
- EINTR：操作被信号中断
- EINVAL：参数无效
- ENOMEM：内存不足
- ENOTCONN：与面向连接关联的套接字尚未被连接上
- ENOTSOCK：sock索引的不是套接字 当返回值是0时，为正常关闭连接；
- EWOULDBLOCK：用于非阻塞模式，不需要重新读或者写

UDP

- UDP协议全称是用户数据报协议 ，在网络中它与TCP协议一样用于处理数据包，是一种无连接的协议。在OSI模型中，在第四层——传输层，处于IP协议的上一层。UDP有不提供数据包分组、组装和不能对数据包进行排序的缺点，也就是说，当报文发送之后，是无法得知其是否安全完整到达的。UDP用来支持那些需要在计算机之间传输数据的网络应用。包括网络视频会议系统在内的众多的客户/服务器模式的网络应用都需要使用UDP协议。

- 与所熟知的TCP（传输控制协议）协议一样，UDP协议直接位于IP（网际协议）协议的顶层。根据OSI（开放系统互连）参考模型，UDP和TCP都属于传输层协议。UDP协议的主要作用是将网络数据流量压缩成数据包的形式。一个典型的数据包就是一个二进制数据的传输单位。每一个数据包的前8个字节用来包含报头信息，剩余字节则用来包含具体的传输数据。

UDT

- UDP为基础开发的一个库.
- http://udt.sourceforge.net/

勘误

strcpy() strncpy()

```
typedef struct  cc_userData
{
        char msgHeader[4];
        char userName[32];
        char password[32];
        char reserved;
 }CC_UserDataDefine;


CC_UserDataDefine  userInfo;

strcpy((char *)userInfo.msgHeader,"CCTD");//错误代码，内存崩溃

strncpy((char *)userInfo.msgHeader,"CCTD",sizeof(userInfo.msgHeader));
```

```
char strBuff[4]={0};
strBuff[0]='C';
strBuff[1]='C';
strBuff[2]='T';
strBuff[3]='D';

NSString* headerString=[NSString stringWithCString: (const char*)strBuff encodding:
                                                                    NSASCIIENCODING];

if([headerString compare: @"CCTD"]==NSOrderedSame)
{

}
```

```
char cmBuffer[5]={0};
memcpy(cmBuffer,strBuff, sizeof(strBuff));
memset(cmBuffer+4,0,1);


NSString* headerString1=[NSString stringWithCString: (const char*)cmBuffer encodding:
                                                                    NSASCIIENCODING];
if([headerString1 compare: @"CCTD"]==NSOrderedSame)
{

}
```