嵌入式与移动开发系列

NITE 国家信息技术紧缺人才培养工程
National Information Technology Education Project
国家信息技术紧缺人才培养工程系列丛书

众多专家、厂商联合推荐 • 业界权威培训机构的经验总结

嵌入式Linux应用程序开发标准教程（第2版）

华清远见嵌入式培训中心 编著

提供36小时嵌入式专家讲座视频和教学课件

Embeded Linux Application Development

人民邮电出版社
POSTS & TELECOM PRESS

第 10 章　嵌入式 Linux 网络编程

本章目标

本章将介绍嵌入式 Linux 网络编程的基础知识。由于网络在嵌入式中的应用非常广泛，基本上常见的应用都会与网络有关，因此，掌握这一部分的内容是非常重要的。经过本章的学习，读者将会掌握以下内容。

- 掌握 TCP/IP 协议的基础知识
- 掌握嵌入式 Linux 基础网络编程
- 掌握嵌入式 Linux 高级网络编程
- 分析理解 Ping 源代码
- 能够独立编写客户端、服务器端的通信程序
- 能够独立编写 NTP 协议实现程序

10.1 TCP/IP 协议概述

10.1.1 OSI 参考模型及 TCP/IP 参考模型

读者一定都听说过著名的 OSI 协议参考模型，它是基于国际标准化组织（ISO）的建议发展起来的，从上到下共分为 7 层：应用层、表示层、会话层、传输层、网络层、数据链路层及物理层。这个 7 层的协议模型虽然规定得非常细致和完善，但在实际中却得不到广泛的应用，其重要的原因之一就在于它过于复杂。但它仍是此后很多协议模型的基础，这种分层架构的思想在很多领域都得到了广泛的应用。

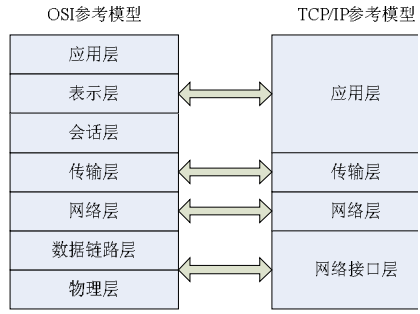

图 10.1 OSI 模型和 TCP/IP 参考模型对应

与此相区别的 TCP/IP 协议模型从一开始就遵循简单明确的设计思路，它将 TCP/IP 的 7 层协议模型简化为 4 层，从而更有利于实现和使用。TCP/IP 的协议参考模型和 OSI 协议参考模型的对应关系如图 10.1 所示。

下面分别对 TCP/IP 的 4 层模型进行简要介绍。

- 网络接口层：负责将二进制流转换为数据帧，并进行数据帧的发送和接收。要注意的是数据帧是独立的网络信息传输单元。
- 网络层：负责将数据帧封装成 IP 数据包，并运行必要的路由算法。
- 传输层：负责端对端之间的通信会话连接与建立。传输协议的选择根据数据传输方式而定。
- 应用层：负责应用程序的网络访问，这里通过端口号来识别各个不同的进程。

10.1.2 TCP/IP 协议族

虽然 TCP/IP 名称只包含了两个协议，但实际上，TCP/IP 是一个庞大的协议族，它包括了各个层次上的众多协议，图 10.2 列举了各层中一些重要的协议，并给出了各个协议在不同层次中所处的位置，如下所示。

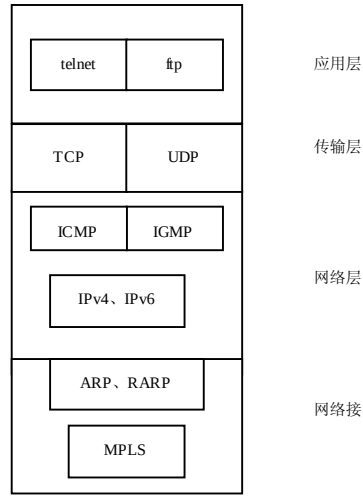

图 10.2 TCP/IP 协议族

- ARP：用于获得同一物理网络中的硬件主机地址。
- MPLS：多协议标签协议，是很有发展前景的下一代网络协议。
- IP：负责在主机和网络之间寻址和路由数据包。
- ICMP：用于发送有关数据包的传送错误的协议。
- IGMP：被 IP 主机用来向本地多路广播路由器报告主机组成员的协议。
- TCP：为应用程序提供可靠的通信连接。适合于一次传输大批数据的情况。并适用于要求得到响应的应用程序。
- UDP：提供了无连接通信，且不对传送包进行可靠性保证。适合于一次传输

少量数据，可靠性则由应用层来负责。

10.1.3 TCP 和 UDP

在此主要介绍在网络编程中涉及的传输层 TCP 和 UDP 协议。

1．TCP

（1）概述。

同其他任何协议栈一样，TCP 向相邻的高层提供服务。因为 TCP 的上一层就是应用层，因此，TCP 数据传输实现了从一个应用程序到另一个应用程序的数据传递。应用程序通过编程调用 TCP 并使用 TCP 服务，提供需要准备发送的数据，用来区分接收数据应用的目的地址和端口号。

通常应用程序通过打开一个 socket 来使用 TCP 服务，TCP 管理到其他 socket 的数据传递。可以说，通过 IP 的源/目的可以惟一地区分网络中两个设备的连接，通过 socket 的源/目的可以惟一地区分网络中两个应用程序的连接。

（2）三次握手协议。

TCP 对话通过三次握手来进行初始化。三次握手的目的是使数据段的发送和接收同步，告诉其他主机其一次可接收的数据量，并建立虚连接。

下面描述了这三次握手的简单过程。

- 初始化主机通过一个同步标志置位的数据段发出会话请求。
- 接收主机通过发回具有以下项目的数据段表示回复：同步标志置位、即将发送的数据段的起始字节的顺序号、应答并带有将收到的下一个数据段的字节顺序号。
- 请求主机再回送一个数据段，并带有确认顺序号和确认号。

图 10.3 就是这个流程的简单示意图。

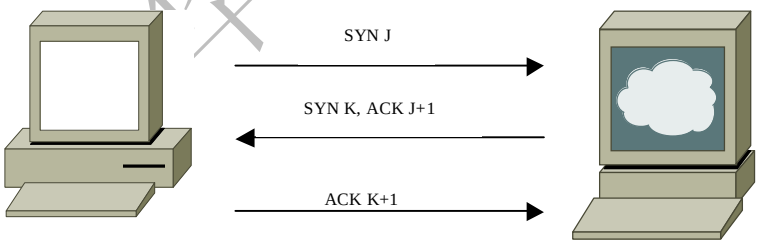

图 10.3　TCP 三次握手协议

TCP 实体所采用的基本协议是滑动窗口协议。当发送方传送一个数据报时，它将启动计时器。当该数据报到达目的地后，接收方的 TCP 实体往回发送一个数据报，其中包含有一个确认序号，它表示希望收到的下一个数据包的顺序号。如果发送方的定时器在确认信息到达之前超时，那么发送方会重发该数据包。

（3）TCP 数据包头。

图 10.4 给出了 TCP 数据包头的格式。

TCP 数据包头的含义如下所示。

- 源端口、目的端口：16 位长。标识出远端和本地的端口号。

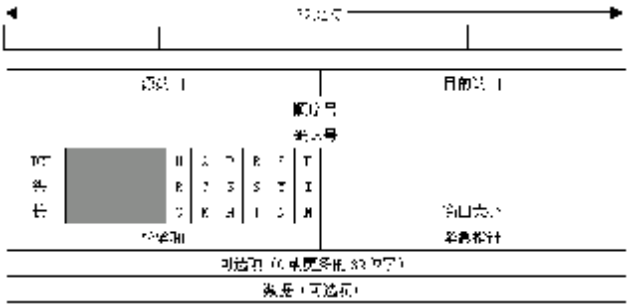

图 10.4　TCP 数据包头的格式

- 序号：32 位长。标识发送的数据报的顺序。
- 确认号：32 位长。希望收到的下一个数据包的序列号。
- TCP 头长：4 位长。表明 TCP 头中包含多少个 32 位字。
- 6 位未用。
- ACK：ACK 位置 1 表明确认号是合法的。如果 ACK 为 0，那么数据报不包含确认信息，确认字段被省略。
- PSH：表示是带有 PUSH 标志的数据。接收方因此请求数据包一到便将其送往应用程序而不必等到缓冲区装满时才传送。
- RST：用于复位由于主机崩溃或其他原因而出现的错误连接。还可以用于拒绝非法的数据包或拒绝连接请求。
- SYN：用于建立连接。
- FIN：用于释放连接。
- 窗口大小：16 位长。窗口大小字段表示在确认了字节之后还可以发送多少个字节。
- 校验和：16 位长。是为了确保高可靠性而设置的。它校验头部、数据和伪 TCP 头部之和。
- 可选项：0 个或多个 32 位字。包括最大 TCP 载荷，滑动窗口比例以及选择重发数据包等选项。

2．UDP

（1）概述。

UDP 即用户数据报协议，它是一种无连接协议，因此不需要像 TCP 那样通过三次握手来建立一个连接。同时，一个 UDP 应用可同时作为应用的客户或服务器方。由于 UDP 协议并不需要建立一个明确的连接，因此建立 UDP 应用要比建立 TCP 应用简单得多。

UDP 协议从问世至今已经被使用了很多年，虽然其最初的光彩已经被一些类似协议所掩盖，但是在网络质量越来越高的今天，UDP 的应用得到了大大的增强。它比 TCP 协议更为高效，也能更好地解决实时性的问题。如今，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使用 UDP 协议。

（2）UDP 数据报头。

UDP 数据报头如下图 10.5 所示。

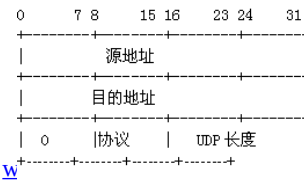

图 10.5　UDP 数据报头

- 源地址、目的地址：16 位长。标识出远端和本地的端口号。
- 数据报的长度是指包括报头和数据部分在内的总的字节数。因为报头的长度是固定的，所以该域主要用来计算可变长度的数据部分（又称为数据负载）。

3．协议的选择

协议的选择应该考虑到以下 3 个方面。

（1）对数据可靠性的要求。

对数据要求高可靠性的应用需选择 TCP 协议，如验证、密码字段的传送都是不允许出错的，而对数据的可靠性要求不那么高的应用可选择 UDP 传送。

（2）应用的实时性。

TCP 协议在传送过程中要使用三次握手、重传确认等手段来保证数据传输的可靠性。使用 TCP 协议会有较大的时延，因此不适合对实时性要求较高的应用，如 VOIP、视频监控等。相反，UDP 协议则在这些应用中能发挥很好的作用。

（3）网络的可靠性。

由于 TCP 协议的提出主要是解决网络的可靠性问题，它通过各种机制来减少错误发生的概率。因此，在网络状况不是很好的情况下需选用 TCP 协议（如在广域网等情况），但是若在网络状况很好的情况下（如局域网等）就不需要再采用 TCP 协议，而建议选择 UDP 协议来减少网络负荷。

10.2 网络基础编程

10.2.1 socket 概述

1．socket 定义

在 Linux 中的网络编程是通过 socket 接口来进行的。人们常说的 socket 是一种特殊的 I/O 接口，它也是一种文件描述符。socket 是一种常用的进程之间通信机制，通过它不仅能实现本地机器上的进程之间的通信，而且通过网络能够在不同机器上的进程之间进行通信。

每一个 socket 都用一个半相关描述{协议、本地地址、本地端口}来表示；一个完整的套接字则用一个相关描述{协议、本地地址、本地端口、远程地址、远程端口}来表示。socket 也有一个类似于打开文件的函数调用，该函数返回一个整型的 socket 描述符，随后的连接建立、数据传输等操作都是通过 socket 来实现的。

2．socket 类型

常见的 socket 有 3 种类型如下。

（1）流式 socket（SOCK_STREAM）。

流式套接字提供可靠的、面向连接的通信流；它使用 TCP 协议，从而保证了数据传输的正确性和顺序性。

（2）数据报 socket（SOCK_DGRAM）。

数据报套接字定义了一种无连接的服务，数据通过相互独立的报文进行传输，是无序的，并且不保证是可靠、无差错的。它使用数据报协议 UDP。

（3）原始 socket。

原始套接字允许对底层协议如 IP 或 ICMP 进行直接访问，它功能强大但使用较为不便，主要用于一些协议的开发。

10.2.2 地址及顺序处理

1．地址结构相关处理

（1）数据结构介绍。

下面首先介绍两个重要的数据类型：sockaddr 和 sockaddr_in，这两个结构类型都是用来保存 socket 信息的，如下所示：

```
struct sockaddr
{
      unsigned short sa_family; /*地址族*/
      char sa_data[14]; /*14 字节的协议地址，包含该 socket 的 IP 地址和端口号。*/
};
struct sockaddr_in
{
      short int sa_family; /*地址族*/
      unsigned short int sin_port; /*端口号*/
      struct in_addr sin_addr; /*IP 地址*/
      unsigned char sin_zero[8]; /*填充 0 以保持与 struct sockaddr 同样大小*/
};
```

这两个数据类型是等效的，可以相互转化，通常 sockaddr_in 数据类型使用更为方便。在建立 socketadd 或 sockaddr_in 后，就可以对该 socket 进行适当的操作了。

（2）结构字段。

表 10.1 列出了该结构 sa_family 字段可选的常见值。

表 10.1

结构定义头文件	#include <netinet/in.h>
sa_family	AF_INET：IPv4 协议
	AF_INET6：IPv6 协议
	AF_LOCAL：UNIX 域协议
	AF_LINK：链路地址协议
	AF_KEY：密钥套接字（socket）

sockaddr_in 其他字段的含义非常清楚，具体的设置涉及其他函数，在后面会有详细的讲解。

2．数据存储优先顺序

（1）函数说明。

计算机数据存储有两种字节优先顺序：高位字节优先（称为大端模式）和低位字节优先（称为小端模式，PC 机通常采用小端模式）。Internet 上数据以高位字节优先顺序在网络上传输，因此在有些情况下，需要对这两个字节存储优先顺序进行相互转化。这里用到了 4 个函数：htons()、ntohs()、htonl()和 ntohl()。这 4 个地址分别实现网络字节序和主机字节序的转化，这里的 h 代表 host，n 代表 network，s 代表 short，l 代表 long。通常 16 位的 IP 端口号用 s 代表，而 IP 地址用 l 来代表。

（2）函数格式说明。

表 10.2 列出了这 4 个函数的语法格式。

表 10.2　　htons 等函数语法要点

所需头文件	#include <netinet/in.h>
函数原型	uint16_t htons(unit16_t host16bit) uint32_t htonl(unit32_t host32bit) uint16_t ntohs(unit16_t net16bit) uint32_t ntohs(unit32_t net32bit)
函数传入值	host16bit：主机字节序的 16 位数据
	host32bit：主机字节序的 32 位数据
	net16bit：网络字节序的 16 位数据
	net32bit：网络字节序的 32 位数据
函数返回值	成功：返回要转换的字节序
	出错：–1

注意　调用该函数只是使其得到相应的字节序，用户不需清楚该系统的主机字节序和网络字节序是否真正相等。如果是相同不需要转换的话，该系统的这些函数会定成空宏。

3．地址格式转化

（1）函数说明。

通常用户在表达地址时采用的是点分十进制表示的数值（或者是以冒号分开的十进制 IPv6 地址），而在通常使用的 socket 编程中所使用的则是二进制值，这就需要将这两个数值进行转换。这里在 IPv4 中用到的函数有 inet_aton()、inet_addr()和 inet_ntoa()，而 IPv4 和 IPv6 兼容的函数有 inet_pton()和 inet_ntop()。由于 IPv6 是下一代互联网的标准协议，因此，本书讲解的函数都能够同时兼容 IPv4 和 IPv6，但在具体举例时仍以 IPv4 为例。

这里 inet_pton()函数是将点分十进制地址映射为二进制地址，而 inet_ntop()是将二进制地址映射为点分十进制地址。

（2）函数格式。

表 10.3 列出了 inet_pton 函数的语法要点。

表 10.3　　inet_pton 函数语法要点

所需头文件	#include <arpa/inet.h>	
函数原型	int inet_pton(int family, const char *strptr, void *addrptr)	
函数传入值	family	AF_INET：IPv4 协议
		AF_INET6：IPv6 协议
	strptr：要转化的值	
	addrptr：转化后的地址	
函数返回值	成功：0	
	出错：−1	

表 10.4 列出了 inet_ntop 函数的语法要点。

表 10.4　　inet_ntop 函数语法要点

所需头文件	#include <arpa/inet.h>	
函数原型	int inet_ntop(int family, void *addrptr, char *strptr, size_t len)	
函数传入值	family	AF_INET：IPv4 协议
		AF_INET6：IPv6 协议
函数传入值	addrptr：转化后的地址	
	strptr：要转化的值	
	len：转化后值的大小	
函数返回值	成功：0	
	出错：−1	

4．名字地址转化

（1）函数说明。

通常，人们在使用过程中都不愿意记忆冗长的 IP 地址，尤其到 IPv6 时，地址长度多达 128 位，那时就更加不可能一次次记忆那么长的 IP 地址了。因此，使用主机名将会是很好的选择。在 Linux 中，同样有一些函数可以实现主机名和地址的转化，最为常见的有 gethostbyname()、gethostbyaddr()和 getaddrinfo()等，它们都可以实现 IPv4 和 IPv6 的地址和主机名之间的转化。其中 gethostbyname()是将主机名转化为 IP 地址，gethostbyaddr()则是逆操作，是将 IP 地址转化为主机名，另外 getaddrinfo()还能实现自动识别 IPv4 地址和 IPv6 地址。

gethostbyname()和 gethostbyaddr()都涉及一个 hostent 的结构体，如下所示：

```
struct hostent
{
        char *h_name;/*正式主机名*/
        char **h_aliases;/*主机别名*/
        int h_addrtype;/*地址类型*/
```

```
        int h_length;/*地址字节长度*/
        char **h_addr_list;/*指向 IPv4 或 IPv6 的地址指针数组*/
    }
```

调用 gethostbyname()函数或 gethostbyaddr()函数后就能返回 hostent 结构体的相关信息。

getaddrinfo()函数涉及一个 addrinfo 的结构体，如下所示：

```
    struct addrinfo
    {
        int ai_flags;/*AI_PASSIVE, AI_CANONNAME;*/
        int ai_family;/*地址族*/
        int ai_socktype;/*socket 类型*/
        int ai_protocol;/*协议类型*/
        size_t ai_addrlen;/*地址字节长度*/
        char *ai_canonname;/*主机名*/
        struct sockaddr *ai_addr;/*socket 结构体*/
        struct addrinfo *ai_next;/*下一个指针链表*/
    }
```

hostent 结构体而言，addrinfo 结构体包含更多的信息。

（2）函数格式。

表 10.5 列出了 gethostbyname()函数的语法要点。

表 10.5　　gethostbyname 函数语法要点

所需头文件	#include <netdb.h>
函数原型	struct hostent *gethostbyname(const char *hostname)
函数传入值	hostname：主机名
函数返回值	成功：hostent 类型指针
	出错：−1

调用该函数时可以首先对 hostent 结构体中的 h_addrtype 和 h_length 进行设置，若为 IPv4 可设置为 AF_INET 和 4;若为 IPv6 可设置为 AF_INET6 和 16;若不设置则默认为 IPv4 地址类型。

表 10.6 列出了 getaddrinfo()函数的语法要点。

表 10.6　　getaddrinfo()函数语法要点

所需头文件	#include <netdb.h>
函数原型	int getaddrinfo(const char *node, const char *service, const struct addrinfo *hi struct addrinfo **result)
函数传入值	node：网络地址或者网络主机名
	service：服务名或十进制的端口号字符串

	hints：服务线索
	result：返回结果
函数返回值	成功：0
	出错：−1

在调用之前，首先要对 hints 服务线索进行设置。它是一个 addrinfo 结构体，表 10.7 列举了该结构体常见的选项值。

表 10.7　　addrinfo 结构体常见选项值

结构体头文件	#include <netdb.h>		
ai_flags	AI_PASSIVE：该套接口是用作被动地打开		
	AI_CANONNAME：通知 getaddrinfo 函数返回主机的名字		
ai_family	AF_INET：IPv4 协议		
	AF_INET6：IPv6 协议		
	AF_UNSPEC：IPv4 或 IPv6 均可		
ai_socktype	SOCK_STREAM：字节流套接字 socket（TCP）		
	SOCK_DGRAM：数据报套接字 socket（UDP）		
ai_protocol	IPPROTO_IP：IP 协议		
	IPPROTO_IPV4：IPv4 协议	4	IPv4
	IPPROTO_IPV6：IPv6 协议		
	IPPROTO_UDP：UDP		
	IPPROTO_TCP：TCP		

注意

（1）通常服务器端在调用 getaddrinfo()之前，ai_flags 设置 AI_PASSIVE，用于 bind()函数（用于端口和地址的绑定，后面会讲到），主机名 nodename 通常会设置为 NULL。

（2）客户端调用 getaddrinfo()时，ai_flags 一般不设置 AI_PASSIVE，但是主机名 nodename 和服务名 servname（端口）则应该不为空。

（3）即使不设置 ai_flags 为 AI_PASSIVE，取出的地址也可以被绑定，很多程序中 ai_flags 直接设置为 0，即 3 个标志位都不设置，这种情况下只要 hostname 和 servname 设置的没有问题就可以正确绑定。

（3）使用实例。

下面的实例给出了 getaddrinfo 函数用法的示例，在后面小节中会给出 gethostbyname 函数用法的例子。

```
/* getaddrinfo.c */
#include <stdio.h>
#include <stdlib.h>
```

```
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

int main()
{
    struct addrinfo hints, *res = NULL;
    int rc;

    memset(&hints, 0, sizeof(hints));
    /*设置 addrinfo 结构体中各参数 */
    hints.ai_flags = AI_CANONNAME;
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_DGRAM;
    hints.ai_protocol = IPPROTO_UDP;
    /*调用 getaddinfo 函数*/
    rc = getaddrinfo("localhost", NULL, &hints, &res);
    if (rc != 0)
    {
        perror("getaddrinfo");
        exit(1);
    }
    else
    {
        printf("Host name is %s\n", res->ai_canonname);
    }
    exit(0);
}
```

10.2.3 socket 基础编程

（1）函数说明。

socket 编程的基本函数有 socket()、bind()、listen()、accept()、send()、sendto()、recv()以及 recvfrom()等，其中根据客户端还是服务端，或者根据使用 TCP 协议还是 UDP 协议，这些函数的调用流程都有所区别，这里先对每个函数进行说明，再给出各种情况下使用的流程图。

- socket()：该函数用于建立一个 socket 连接，可指定 socket 类型等信息。在建

立了 socket 连接之后，可对 sockaddr 或 sockaddr_in 结构进行初始化，以保存所建立的 socket 地址信息。

- bind()：该函数是用于将本地 IP 地址绑定到端口号，若绑定其他 IP 地址则不能成功。另外，它主要用于 TCP 的连接，而在 UDP 的连接中则无必要。
- listen()：在服务端程序成功建立套接字和与地址进行绑定之后，还需要准备在该套接字上接收新的连接请求。此时调用 listen()函数来创建一个等待队列，在其中存放未处理的客户端连接请求。
- accept()：服务端程序调用 listen()函数创建等待队列之后，调用 accept()函数等待并接收客户端的连接请求。它通常从由 bind()所创建的等待队列中取出第一个未处理的连接请求。
- connect()：该函数在 TCP 中是用于 bind()的之后的 client 端，用于与服务器端建立连接，而在 UDP 中由于没有了 bind()函数，因此用 connect()有点类似 bind()函数的作用。
- send()和 recv()：这两个函数分别用于发送和接收数据，可以用在 TCP 中，也可以用在 UDP 中。当用在 UDP 时，可以在 connect()函数建立连接之后再用。
- sendto()和 recvfrom()：这两个函数的作用与 send()和 recv()函数类似，也可以用在 TCP 和 UDP 中。当用在 TCP 时，后面的几个与地址有关参数不起作用，函数作用等同于 send()和 recv()；当用在 UDP 时，可以用在之前没有使用 connect()的情况下，这两个函数可以自动寻找指定地址并进行连接。

服务器端和客户端使用 TCP 协议的流程如图 10.6 所示。

服务器端和客户端使用 UDP 协议的流程如图 10.7 所示。

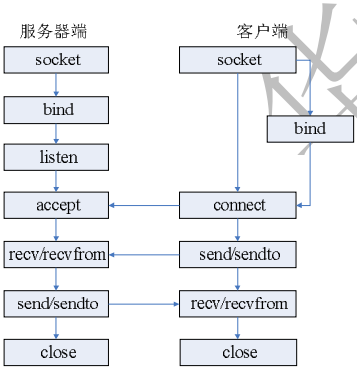

图 10.6　使用 TCP 协议 socket 编程流程图

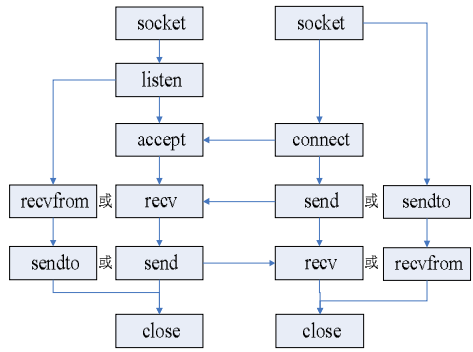

图 10.7　使用 UDP 协议 socket 编程流程图

（2）函数格式。

表 10.8 列出了 socket()函数的语法要点。

表 10.8　　socket()函数语法要点

所需头文件	#include <sys/socket.h>	
函数原型	int socket(int family, int type, int protocol)	
函数传入值	family：	AF_INET：IPv4 协议

	协议族	AF_INET6：IPv6 协议
		AF_LOCAL：UNIX 域协议
		AF_ROUTE：路由套接字（socket）
		AF_KEY：密钥套接字（socket）
	type： 套接字类型	SOCK_STREAM：字节流套接字 socket
		SOCK_DGRAM：数据报套接字 socket
		SOCK_RAW：原始套接字 socket
	protoco：0（原始套接字除外）	
函数返回值	成功：非负套接字描述符	
	出错：−1	

表 10.9 列出了 bind()函数的语法要点。

表 10.9　　bind()函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int bind(int sockfd, struct sockaddr *my_addr, int addrlen)
函数传入值	socktd：套接字描述符
	my_addr：本地地址
	addrlen：地址长度
函数返回值	成功：0
	出错：−1

端口号和地址在 my_addr 中给出了，若不指定地址，则内核随意分配一个临时端口给该应用程序。

表 10.10 列出了 listen()函数的语法要点。

表 10.10　　listen()函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int listen(int sockfd, int backlog)
函数传入值	socktd：套接字描述符
	backlog：请求队列中允许的最大请求数，大多数系统缺省值为 5
函数返回值	成功：0
	出错：−1

表 10.11 列出了 accept()函数的语法要点。

表 10.11　　accept()函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)
函数传入值	socktd：套接字描述符
	addr：客户端地址

	addrlen：地址长度
函数返回值	成功：0
	出错：–1

表 10.12 列出了 connect()函数的语法要点。

表 10.12　　connect()函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int connect(int sockfd, struct sockaddr *serv_addr, int addrlen)
函数传入值	socktd：套接字描述符
	serv_addr：服务器端地址
	addrlen：地址长度
函数返回值	成功：0
	出错：–1

表 10.13 列出了 send()函数的语法要点。

表 10.13　　send()函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int send(int sockfd, const void *msg, int len, int flags)
函数传入值	socktd：套接字描述符
	msg：指向要发送数据的指针
	len：数据长度
	flags：一般为 0
函数返回值	成功：发送的字节数
	出错：–1

表 10.14 列出了 recv()函数的语法要点。

表 10.14　　recv()函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int recv(int sockfd, void *buf,int len, unsigned int flags)
函数传入值	socktd：套接字描述符
	buf：存放接收数据的缓冲区
	len：数据长度
	flags：一般为 0
函数返回值	成功：接收的字节数
	出错：–1

表 10.15 列出了 sendto()函数的语法要点。

表 10.15　　sendto()函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int sendto(int sockfd, const void *msg,int len, unsigned int flags, const struct socka

	*to, int tolen)
函数传入值	socktd：套接字描述符
	msg：指向要发送数据的指针
	len：数据长度
	flags：一般为 0
	to：目地机的 IP 地址和端口号信息
	tolen：地址长度
函数返回值	成功：发送的字节数
	出错：–1

表 10.16 列出了 recvfrom()函数的语法要点。

表 10.16　　recvfrom()函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int recvfrom(int sockfd,void *buf, int len, unsigned int flags, struct sockaddr *fr int *fromlen)
函数传入值	socktd：套接字描述符
	buf：存放接收数据的缓冲区
	len：数据长度
	flags：一般为 0
	from：源主机的 IP 地址和端口号信息
	tolen：地址长度
函数返回值	成功：接收的字节数
	出错：–1

（3）使用实例。

该实例分为客户端和服务器端两部分，其中服务器端首先建立起 socket，然后与本地端口进行绑定，接着就开始接收从客户端的连接请求并建立与它的连接，接下来，接收客户端发送的消息。客户端则在建立 socket 之后调用 connect()函数来建立连接。

服务端的代码如下所示：

```
/*server.c*/
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
```

```
#define PORT          4321
#define BUFFER_SIZE       1024
#define MAX_QUE_CONN_NM   5

int main()
{
    struct sockaddr_in server_sockaddr,client_sockaddr;
    int sin_size,recvbytes;
    int sockfd, client_fd;
    char buf[BUFFER_SIZE];

    /*建立 socket 连接*/
    if ((sockfd = socket(AF_INET,SOCK_STREAM,0))== -1)
    {
        perror("socket");
        exit(1);
    }
    printf("Socket id = %d\n",sockfd);

    /*设置 sockaddr_in 结构体中相关参数*/
    server_sockaddr.sin_family = AF_INET;
    server_sockaddr.sin_port = htons(PORT);
    server_sockaddr.sin_addr.s_addr = INADDR_ANY;
    bzero(&(server_sockaddr.sin_zero), 8);

    int i = 1;/* 允许重复使用本地地址与套接字进行绑定 */
    setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &i, sizeof(i));

    /*绑定函数 bind()*/
    if (bind(sockfd, (struct sockaddr *)&server_sockaddr,
                sizeof(struct sockaddr)) == -1)
    {
        perror("bind");
        exit(1);
    }
    printf("Bind success!\n");

    /*调用 listen()函数，创建未处理请求的队列*/
    if (listen(sockfd, MAX_QUE_CONN_NM) == -1)
```

```
    {
        perror("listen");
        exit(1);
    }
    printf("Listening....\n");

    /*调用 accept()函数，等待客户端的连接*/
    if ((client_fd = accept(sockfd,
            (struct sockaddr *)&client_sockaddr, &sin_size)) == -1)
    {
        perror("accept");
        exit(1);
    }

    /*调用 recv()函数接收客户端的请求*/
    memset(buf , 0, sizeof(buf));
    if ((recvbytes = recv(client_fd, buf, BUFFER_SIZE, 0)) == -1)
    {
        perror("recv");
        exit(1);
    }
    printf("Received a message: %s\n", buf);
    close(sockfd);
    exit(0);
}
```

客户端的代码如下所示：

```
/*client.c*/
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define PORT   4321
#define BUFFER_SIZE 1024
```

```
    int main(int argc, char *argv[])
    {
        int sockfd,sendbytes;
        char buf[BUFFER_SIZE];
        struct hostent *host;
        struct sockaddr_in serv_addr;

        if(argc < 3)
        {
            fprintf(stderr,"USAGE:  ./client  Hostname(or  ip  address)
Text\n");
            exit(1);
        }

        /*地址解析函数*/
        if ((host = gethostbyname(argv[1])) == NULL)
        {
            perror("gethostbyname");
            exit(1);
        }

        memset(buf, 0, sizeof(buf));
        sprintf(buf, "%s", argv[2]);

        /*创建 socket*/
        if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        {
            perror("socket");
            exit(1);
        }

        /*设置 sockaddr_in 结构体中相关参数*/
        serv_addr.sin_family = AF_INET;
        serv_addr.sin_port = htons(PORT);
        serv_addr.sin_addr = *((struct in_addr *)host->h_addr);
        bzero(&(serv_addr.sin_zero), 8);

        /*调用 connect 函数主动发起对服务器端的连接*/
        if(connect(sockfd,(struct sockaddr *)&serv_addr,
                                    sizeof(struct sockaddr))== -1)
```

```
        {
            perror("connect");
            exit(1);
        }

        /*发送消息给服务器端*/
        if ((sendbytes = send(sockfd, buf, strlen(buf), 0)) == -1)
        {
            perror("send");
            exit(1);
        }
        close(sockfd);
        exit(0);
    }
```

在运行时需要先启动服务器端，再启动客户端。这里可以把服务器端下载到开发板上，客户端在宿主机上运行，然后配置双方的 IP 地址，在确保双方可以通信（如使用 ping 命令验证）的情况下运行该程序即可。

```
$ ./server
Socket id = 3
Bind success!
Listening....
Received a message: Hello,Server!
$ ./client localhost(或者输入 IP 地址) Hello,Server!
```

10.3 网络高级编程

在实际情况中，人们往往遇到多个客户端连接服务器端的情况。由于之前介绍的如 connet()、recv()和 send()等都是阻塞性函数，如果资源没有准备好，则调用该函数的进程将进入睡眠状态，这样就无法处理 I/O 多路复用的情况了。本节给出了两种解决 I/O 多路复用的解决方法，这两个函数都是之前学过的 fcntl()和 select()（请读者先复习第 6 章中的相关内容）。可以看到，由于在 Linux 中把 socket 也作为一种特殊文件描述符，这给用户的处理带来了很大的方便。

1．fcntl()

函数 fcntl()针对 socket 编程提供了如下的编程特性。

- 非阻塞 I/O：可将 cmd 设置为 F_SETFL，将 lock 设置为 O_NONBLOCK。
- 异步 I/O：可将 cmd 设置为 F_SETFL，将 lock 设置为 O_ASYNC。

下面是用 fcntl()将套接字设置为非阻塞 I/O 的实例代码：

```
/* net_fcntl.c */
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/un.h>
#include <sys/time.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <netinet/in.h>
#include <fcntl.h>

#define PORT                1234
#define MAX_QUE_CONN_NM       5
#define BUFFER_SIZE           1024

int main()
{
    struct sockaddr_in server_sockaddr, client_sockaddr;
    int sin_size, recvbytes, flags;
    int sockfd, client_fd;
    char buf[BUFFER_SIZE];

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        perror("socket");
        exit(1);
    }
    server_sockaddr.sin_family = AF_INET;
    server_sockaddr.sin_port = htons(PORT);
    server_sockaddr.sin_addr.s_addr = INADDR_ANY;
    bzero(&(server_sockaddr.sin_zero), 8);
    int i = 1;/* 允许重复使用本地地址与套接字进行绑定 */
    setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &i, sizeof(i));
    if (bind(sockfd, (struct sockaddr *)&server_sockaddr,
```

```
    sizeof(struct sockaddr)) == -1)
        {
            perror("bind");
            exit(1);
        }
        if(listen(sockfd,MAX_QUE_CONN_NM) == -1)
        {
            perror("listen");
            exit(1);
        }
        printf("Listening....\n");
        /* 调用 fcntl()函数给套接字设置非阻塞属性 */
        flags = fcntl(sockfd, F_GETFL);
        if (flags < 0 || fcntl(sockfd, F_SETFL, flags|O_NONBLOCK) < 0)
        {
            perror("fcntl");
            exit(1);
        }

        while(1)
        {
            sin_size = sizeof(struct sockaddr_in);
            if ((client_fd = accept(sockfd,
                        (struct sockaddr*)&client_sockaddr, &sin_size)) <
0)
            {
                perror("accept");
                exit(1);
            }

                if ((recvbytes = recv(client_fd, buf, BUFFER_SIZE, 0)) <
0)
                {
                perror("recv");
                exit(1);
            }
            printf("Received a message: %s\n", buf);
      } /*while*/

      close(client_fd);
```

```
    exit(1);
}
```

运行该程序，结果如下所示：

```
$ ./net_fcntl
Listening....
accept: Resource temporarily unavailable
```

可以看到，当 accept()的资源不可用（没有任何未处理的等待连接的请求）时，程序就会自动返回。

2．select()

使用 fcntl()函数虽然可以实现非阻塞 I/O 或信号驱动 I/O，但在实际使用时往往会对资源是否准备完毕进行循环测试，这样就大大增加了不必要的 CPU 资源的占用。在这里可以使用 select()函数来解决这个问题，同时，使用 select()函数还可以设置等待的时间，可以说功能更加强大。下面是使用 select()函数的服务器端源代码。客户端程序基本上与 10.2.3 小节中的例子相同，仅加入一行 sleep()函数，使得客户端进程等待几秒钟才结束。

```
/* net_select.c */
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <netinet/in.h>
#define PORT              4321
#define MAX_QUE_CONN_NM     5
#define MAX_SOCK_FD         FD_SETSIZE
#define BUFFER_SIZE         1024

int main()
{
    struct sockaddr_in server_sockaddr, client_sockaddr;
    int sin_size, count;
    fd_set inset, tmp_inset;
    int sockfd, client_fd, fd;
    char buf[BUFFER_SIZE];
```

```
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
    perror("socket");
    exit(1);
}
server_sockaddr.sin_family = AF_INET;
server_sockaddr.sin_port = htons(PORT);
server_sockaddr.sin_addr.s_addr = INADDR_ANY;
bzero(&(server_sockaddr.sin_zero), 8);
int i = 1;/* 允许重复使用本地地址与套接字进行绑定 */
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &i, sizeof(i));
if (bind(sockfd, (struct sockaddr *)&server_sockaddr,
            sizeof(struct sockaddr)) == -1)
{
    perror("bind");
    exit(1);
}

if(listen(sockfd, MAX_QUE_CONN_NM) == -1)
{
    perror("listen");
    exit(1);
}
printf("listening....\n");
/*将调用 socket()函数的描述符作为文件描述符*/
FD_ZERO(&inset);
FD_SET(sockfd, &inset);
while(1)
{
    tmp_inset = inset;
    sin_size=sizeof(struct sockaddr_in);
    memset(buf, 0, sizeof(buf));
    /*调用 select()函数*/
    if (!(select(MAX_SOCK_FD, &tmp_inset, NULL, NULL, NULL) > 0))
    {
        perror("select");
    }
    for (fd = 0; fd < MAX_SOCK_FD; fd++)
    {
```

```
                if (FD_ISSET(fd, &tmp_inset) > 0)
                {
                    if (fd == sockfd)
                    { /* 服务端接收客户端的连接请求 */
                       if ((client_fd = accept(sockfd,
                      (struct sockaddr *)&client_sockaddr, &sin_size))==
-1)
                          {
                            perror("accept");
                            exit(1);
                          }
                          FD_SET(client_fd, &inset);
                          printf("New   connection   from   %d(socket)\n",
client_fd);
                    }
                    else /* 处理从客户端发来的消息 */
                    {
                        if ((count = recv(client_fd, buf, BUFFER_SIZE, 0)) >
0)
                        {
                            printf("Received a message from %d: %s\n",
                                                    client_fd,
buf);
                        }
                        else
                        {
                           close(fd);
                           FD_CLR(fd, &inset);
                           printf("Client %d(socket) has left\n", fd);
                        }
                    }
                } /* end of if FD_ISSET*/
            } /* end of for fd*/
        } /* end if while while*/
        close(sockfd);
        exit(0);
    }
```

运行该程序时，可以先启动服务器端，再反复运行客户端程序（这里启动两个客户端进程）即可，服务器端运行结果如下所示：

```
    $ ./server
    listening....
    New connection from 4(socket)                /* 接受第一个客户端的连接请求
*/
    Received a message from 4: Hello,First!     /* 接收第一个客户端发送的数据
*/
    New connection from 5(socket)              /* 接受第二个客户端的连接请求*/
    Received a message from 5: Hello,Second!    /* 接收第二个客户端发送的数据
*/
    Client 4(socket) has left                  /* 检测到第一个客户端离线了*/
    Client 5(socket) has left                  /* 检测到第二个客户端离线了*/
    $ ./client localhost Hello,First! & ./client localhost Hello,Second
```

10.4 实验内容——NTP 协议实现

1. 实验目的

通过实现 NTP 协议的练习，进一步掌握 Linux 网络编程，并且提高协议的分析与实现能力，为参与完成综合性项目打下良好的基础。

2. 实验内容

Network Time Protocol（NTP）协议是用来使计算机时间同步化的一种协议，它可以使计算机对其服务器或时钟源（如石英钟，GPS 等）做同步化，它可以提供高精确度的时间校正（LAN 上与标准时间差小于 1 毫秒，WAN 上几十毫秒），且可用加密确认的方式来防止恶毒的协议攻击。

NTP 提供准确时间，首先要有准确的时间来源，这一时间应该是国际标准时间 UTC。 NTP 获得 UTC 的时间来源可以是原子钟、天文台、卫星，也可以从 Internet 上获取。这样就有了准确而可靠的时间源。时间是按 NTP 服务器的等级传播。按照距离外部 UTC 源的远近将所有服务器归入不同的 Stratun（层）中。Stratum-1 在顶层，有外部 UTC 接入，而 Stratum-2 则从 Stratum-1 获取时间，Stratum-3 从 Stratum-2 获取时间，以此类推，但 Stratum 层的总数限制在 15 以内。所有这些服务器在逻辑上形成阶梯式的架构并相互连接，而 Stratum-1 的时间服务器是整个系统的基础。

进行网络协议实现时最重要的是了解协议数据格式。NTP 数据包有 48 个字节，其中 NTP 包头 16 字节，时间戳 32 个字节。其协议格式如图 10.9 所示。

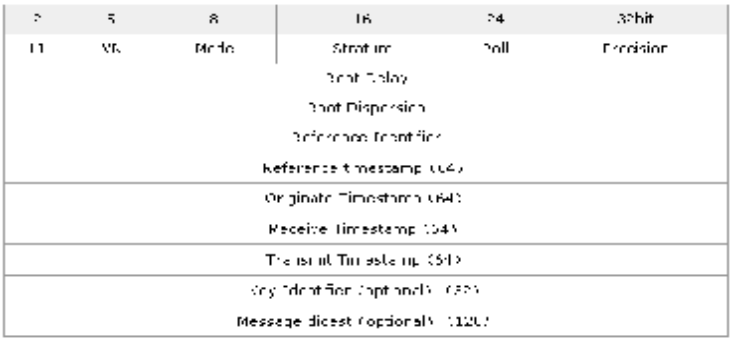

图 10.9　NTP 协议数据格式

其协议字段的含义如下所示。

- LI：跳跃指示器，警告在当月最后一天的最终时刻插入的迫近闰秒（闰秒）。
- VN：版本号。
- Mode：工作模式。该字段包括以下值：0－预留；1－对称行为；3－客户机；4－服务器；5－广播；6－NTP 控制信息。NTP 协议具有 3 种工作模式，分别为主/被动对称模式、客户/服务器模式、广播模式。 在主/被动对称模式中，有一对一的连接，双方均可同步对方或被对方同步，先发出申请建立连接的一方工作在主动模式下，另一方工作在被动模式下；客户/服务器模式与主/被动模式基本相同，惟一区别在于客户方可被服务器同步，但服务器不能被客户同步；在广播模式中，有一对多的连接，服务器不论客户工作在何种模式下，都会主动发出时间信息，客户根据此信息调整自己的时间。
- Stratum：对本地时钟级别的整体识别。
- Poll：有符号整数表示连续信息间的最大间隔。
- Precision：有符号整数表示本地时钟精确度。
- Root Delay：表示到达主参考源的一次往复的总延迟，它是有 15～16 位小数部分的符号定点小数。
- Root Dispersion：表示一次到达主参考源的标准误差，它是有 15～16 位小数部分的无符号定点小数。
- Reference Identifier：识别特殊参考源。
- Originate Timestamp：这是向服务器请求分离客户机的时间，采用 64 位时标格式。
- Receive Timestamp：这是向服务器请求到达客户机的时间，采用 64 位时标格式。
- Transmit Timestamp：这是向客户机答复分离服务器的时间，采用 64 位时标格式。
- Authenticator（Optional）：当实现了 NTP 认证模式时，主要标识符和信息数字域就包括已定义的信息认证代码（MAC）信息。

由于 NTP 协议中涉及比较多的时间相关的操作，为了简化实现过程，在本实验中，仅要求实现 NTP 协议客户端部分的网络通信模块，也就是构造 NTP 协议字段进行发送和接收，最后与时间相关的操作不需进行处理。NTP 协议是作为 OSI 参考模型的高层协议比较适合采用 UDP 传输协议进行数据传输，专用端口号为 123。在实验中，

以国家授时中心服务器（IP 地址为 202.72.145.44）作为 NTP（网络时间）服务器。

3．实验步骤

（1）画出流程图。

简易 NTP 客户端的实现流程如图 10.10 所示。

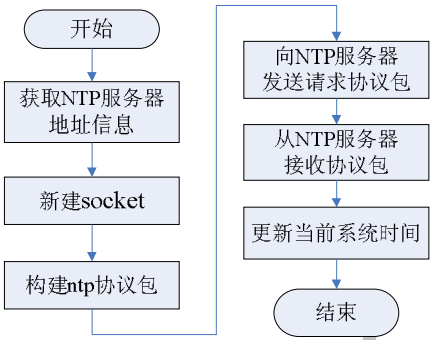

图 10.10　简易 NTP 客户端流程图

（2）编写程序。

具体代码如下：

```
/* ntp.c */
#include <sys/socket.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/un.h>
#include <sys/time.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <netinet/in.h>
#include <string.h>
#include <netdb.h>

#define NTP_PORT            123             /*NTP 专用端口号字符串*/
#define TIME_PORT           37             /* TIME/UDP 端口号 */
#define NTP_SERVER_IP      "210.72.145.44"  /*国家授时中心 IP*/
#define NTP_PORT_STR       "123"          /*NTP 专用端口号字符串*/
#define NTPV1              "NTP/V1"      /*协议及其版本号*/
#define NTPV2              "NTP/V2"
```

```
    #define NTPV3              "NTP/V3"
    #define NTPV4              "NTP/V4"
    #define TIME              "TIME/UDP"

    #define NTP_PCK_LEN 48
    #define LI 0
    #define VN 3
    #define MODE 3
    #define STRATUM 0
    #define POLL 4
    #define PREC -6

    #define JAN_1970  0x83aa7e80  /* 1900年~1970年之间的时间秒数 */
    #define NTPFRAC(x)    (4294 * (x) + ((1981 * (x)) >> 11))
    #define USEC(x)       (((x) >> 12) - 759 * ((((x) >> 10) + 32768) >>
16))

    typedef struct _ntp_time
    {
       unsigned int coarse;
       unsigned int fine;
    } ntp_time;

    struct ntp_packet
    {
        unsigned char leap_ver_mode;
        unsigned char startum;
        char poll;
        char precision;
        int  root_delay;
        int  root_dispersion;
        int reference_identifier;
        ntp_time reference_timestamp;
        ntp_time originage_timestamp;
        ntp_time receive_timestamp;
        ntp_time transmit_timestamp;
    };

    char protocol[32];
    /*构建NTP协议包*/
```

```
int construct_packet(char *packet)
{
    char version = 1;
    long tmp_wrd;
    int port;
    time_t timer;
    strcpy(protocol, NTPV3);
    /*判断协议版本*/
    if(!strcmp(protocol, NTPV1)||!strcmp(protocol, NTPV2)
        ||!strcmp(protocol, NTPV3)||!strcmp(protocol, NTPV4))
    {
        memset(packet, 0, NTP_PCK_LEN);
        port = NTP_PORT;
        /*设置16字节的包头*/
        version = protocol[6] - 0x30;
        tmp_wrd = htonl((LI << 30)|(version << 27)
            |(MODE << 24)|(STRATUM << 16)|(POLL << 8)|(PREC & 0xff));
        memcpy(packet, &tmp_wrd, sizeof(tmp_wrd));

        /*设置Root Delay、Root Dispersion和Reference Indentifier */
        tmp_wrd = htonl(1<<16);
        memcpy(&packet[4], &tmp_wrd, sizeof(tmp_wrd));
        memcpy(&packet[8], &tmp_wrd, sizeof(tmp_wrd));
        /*设置Timestamp部分*/
        time(&timer);
        /*设置Transmit Timestamp coarse*/
        tmp_wrd = htonl(JAN_1970 + (long)timer);
        memcpy(&packet[40], &tmp_wrd, sizeof(tmp_wrd));
        /*设置Transmit Timestamp fine*/
        tmp_wrd = htonl((long)NTPFRAC(timer));
        memcpy(&packet[44], &tmp_wrd, sizeof(tmp_wrd));
        return NTP_PCK_LEN;
    }
    else if (!strcmp(protocol, TIME))/* "TIME/UDP" */
    {
        port = TIME_PORT;
        memset(packet, 0, 4);
        return 4;
    }
    return 0;
```

```
    }

    /*获取 NTP 时间*/
    int get_ntp_time(int sk, struct addrinfo *addr, struct ntp_packet *ret_time)
    {
        fd_set pending_data;
        struct timeval block_time;
        char data[NTP_PCK_LEN * 8];
        int  packet_len, data_len = addr->ai_addrlen, count = 0, result, i,
re;

        if (!(packet_len = construct_packet(data)))
        {
            return 0;
        }
        /*客户端给服务器端发送 NTP 协议数据包*/
        if ((result = sendto(sk, data,
               packet_len, 0, addr->ai_addr, data_len)) < 0)
        {
            perror("sendto");
            return 0;
        }

        /*调用 select()函数，并设定超时时间为 1s*/
        FD_ZERO(&pending_data);
        FD_SET(sk, &pending_data);
        block_time.tv_sec=10;
        block_time.tv_usec=0;
        if (select(sk + 1, &pending_data, NULL, NULL, &block_time) > 0)
        {
            /*接收服务器端的信息*/
            if ((count = recvfrom(sk, data,
                        NTP_PCK_LEN * 8, 0, addr->ai_addr, &data_len)) <
0)
            {
                perror("recvfrom");
                return 0;
            }

            if (protocol == TIME)
```

```
            {
                memcpy(&ret_time->transmit_timestamp, data, 4);
                return 1;
            }
            else if (count < NTP_PCK_LEN)
            {
                return 0;
            }
            /* 设置接收 NTP 包的数据结构 */
            ret_time->leap_ver_mode = ntohl(data[0]);
            ret_time->startum = ntohl(data[1]);
            ret_time->poll = ntohl(data[2]);
            ret_time->precision = ntohl(data[3]);
            ret_time->root_delay = ntohl(*(int*)&(data[4]));
            ret_time->root_dispersion = ntohl(*(int*)&(data[8]));
            ret_time->reference_identifier = ntohl(*(int*)&(data[12]));
            ret_time->reference_timestamp.coarse          =          ntohl
*(int*)&(data[16]));
            ret_time->reference_timestamp.fine                          =
ntohl(*(int*)&(data[20]));
            ret_time->originage_timestamp.coarse                        =
ntohl(*(int*)&(data[24]));
            ret_time->originage_timestamp.fine                          =
ntohl(*(int*)&(data[28]));
            ret_time->receive_timestamp.coarse                          =
ntohl(*(int*)&(data[32]));
            ret_time->receive_timestamp.fine                            =
ntohl(*(int*)&(data[36]));
            ret_time->transmit_timestamp.coarse=ntohl(*(int*)&(data[40]));
            ret_time->transmit_timestamp.fine                           =
ntohl(*(int*)&(data[44]));
            return 1;
        } /* end of if select */
        return 0;
    }

    /* 修改本地时间 */
    int set_local_time(struct ntp_packet * pnew_time_packet)
    {
        struct timeval tv;
```

```
        tv.tv_sec    =    pnew_time_packet->transmit_timestamp.coarse    -
JAN_1970;
        tv.tv_usec = USEC(pnew_time_packet->transmit_timestamp.fine);
        return settimeofday(&tv, NULL);
    }

    int main()
    {
        int sockfd, rc;
        struct addrinfo hints, *res = NULL;
        struct ntp_packet new_time_packet;

        memset(&hints, 0, sizeof(hints));
        hints.ai_family = AF_UNSPEC;
        hints.ai_socktype = SOCK_DGRAM;
        hints.ai_protocol = IPPROTO_UDP;
        /*调用 getaddrinfo()函数，获取地址信息*/
        rc = getaddrinfo(NTP_SERVER_IP, NTP_PORT_STR, &hints, &res);
        if (rc != 0)
        {
            perror("getaddrinfo");
            return 1;
        }
        /* 创建套接字 */
        sockfd      =      socket(res->ai_family,      res->ai_socktype,
res->ai_protocol);
        if (sockfd <0 )
        {
            perror("socket");
            return 1;
        }
    /*调用取得 NTP 时间的函数*/
        if (get_ntp_time(sockfd, res, &new_time_packet))
        {
            /*调整本地时间*/
            if (!set_local_time(&new_time_packet))
            {
                printf("NTP client success!\n");
            }
        }
```

```
        close(sockfd);
        return 0;
    }
```

为了更好地观察程序的效果，先用 date 命令修改一下系统时间，再运行实例程序。运行完了之后再查看系统时间，可以发现已经恢复准确的系统时间了。具体运行结果如下所示。

```
$ date -s "2001-01-01 1:00:00"
2001年 01月 01日 星期一 01:00:00 EST
$ date
2001年 01月 01日 星期一 01:00:00 EST
$ ./ntp
NTP client success!
$ date
能够显示当前准确的日期和时间了!
```

10.5 本章小结

本章首先概括地讲解了 OSI 分层结构以及 TCP/IP 协议各层的主要功能，介绍了常见的 TCP/IP 协议族，并且重点讲解了网络编程中需要用到的 TCP 和 UDP 协议，为嵌入式 Linux 的网络编程打下良好的基础。

接着本章介绍了 socket 的定义及其类型，并逐个介绍常见的 socket 相关的基本函数，包括地址处理函数、数据存储转换函数等，这些函数都是最为常用的函数，要在理解概念的基础上熟练掌握。

接下来介绍的是网络编程中的基本函数，这也是最为常见的几个函数，这里要注意 TCP 和 UDP 在处理过程中的不同。同时，本章还介绍了较为高级的网络编程，包括调用 fcntl()和 select()函数，这两个函数在前面的章节中都已经讲解过，但在本章中有特殊的用途。

最后，本章以 ping 程序为例，讲解了常见协议的实现过程，读者可以看到一个成熟的协议是如何实现的。

本章的实验安排了实现一个比较简单但完整的 NTP 客户端程序，主要实现了其中数据收发的主要功能，以及时间同步调整的功能。

10.6 思考与练习

1．分别用多线程和多路复用实现网络聊天程序。

2．实现一个小型模拟的路由器，就是接收从某个 IP 地址的连接请求，再把该请

求转发到另一个 IP 地址的主机上去。