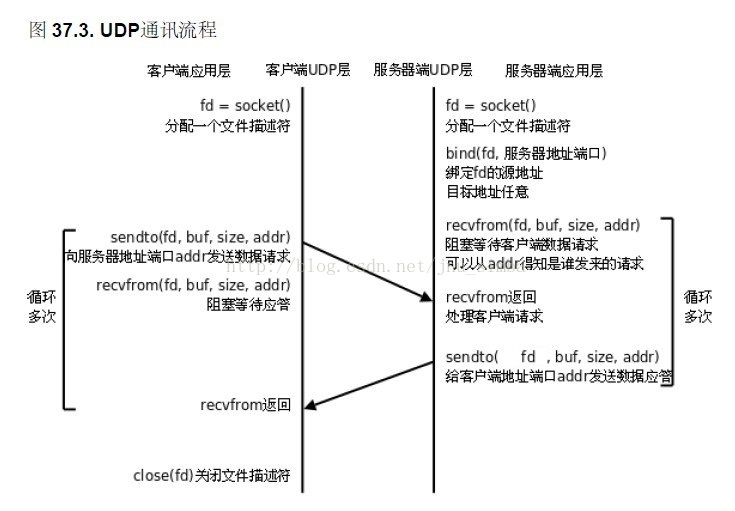
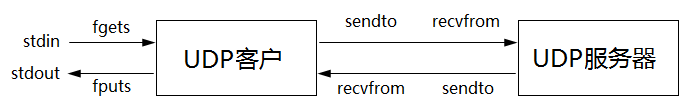
**第十四讲：基于UDP协议的网络程序**

**一、下图是典型的UDP客户端/服务器通讯过程**



下面依照通信流程，我们来实现一个UDP回射客户/服务器



#include <sys/types.h>

#include <sys/socket.h>

ssize\_t send(int sockfd, const void \*buf, size\_t len, int flags);

ssize\_t sendto(int sockfd, const void \*buf, size\_t len, int flags, const struct sockaddr \*dest\_addr, socklen\_t addrlen);

当套接字处于“已连接”的状态时，才可以使用send，当flags = 0 时 send 与 write 一致。

且 send(sockfd, buf, len, flags); 即 sendto(sockfd, buf, len, flags, NULL, 0);

ssize\_t recv(int sockfd, void \*buf, size\_t len, int flags);

ssize\_t recvfrom(int sockfd, void \*buf, size\_t len, int flags, struct sockaddr \*src\_addr, socklen\_t \*addrlen);

recv 与 recvfrom 的关系与 send 与 sendto 的关系一致。

echoser\_udp.c：

c++ code

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<errno.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<string.h>

#define ERR\_EXIT(m) \

do { \

perror(m); \

exit(EXIT\_FAILURE); \

} while (0)void echo\_ser(int sock)

{

char recvbuf[1024] = {0};

struct sockaddr\_in peeraddr;

socklen\_t peerlen;

int n;

while (1)

{

peerlen = sizeof(peeraddr);

memset(recvbuf, 0, sizeof(recvbuf));

n = recvfrom(sock, recvbuf, sizeof(recvbuf), 0,

(struct sockaddr \*)&peeraddr, &peerlen);

if (n == -1)

{

if (errno == EINTR)

continue;

ERR\_EXIT("recvfrom error");

}

else if(n > 0)

{

fputs(recvbuf, stdout);

sendto(sock, recvbuf, n, 0,

(struct sockaddr \*)&peeraddr, peerlen);

}

}

close(sock);

}int main(void)

{

int sock;

if ((sock = socket(PF\_INET, SOCK\_DGRAM, 0)) < 0)

ERR\_EXIT("socket error");

struct sockaddr\_in servaddr;

memset(&servaddr, 0, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(5188);

servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

if (bind(sock, (struct sockaddr \*)&servaddr, sizeof(servaddr)) < 0)

ERR\_EXIT("bind error");

echo\_ser(sock);

return 0;

}

echocli\_udp.c：

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)void echo\_cli(int sock)

{

struct sockaddr\_in servaddr;

memset(&servaddr, 0, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(5188);

servaddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

int ret;

char sendbuf[1024] = {0};

char recvbuf[1024] = {0};

while (fgets(sendbuf, sizeof(sendbuf), stdin) != NULL)

{

sendto(sock, sendbuf, strlen(sendbuf), 0, (struct sockaddr \*)&servaddr, sizeof(servaddr));

ret = recvfrom(sock, recvbuf, sizeof(recvbuf), 0, NULL, NULL);

if (ret == -1)

{

if (errno == EINTR)

continue;

ERR\_EXIT("recvfrom");

}

fputs(recvbuf, stdout);

memset(sendbuf, 0, sizeof(sendbuf));

memset(recvbuf, 0, sizeof(recvbuf));

}

close(sock);

}int main(void)

{

int sock;

if ((sock = socket(PF\_INET, SOCK\_DGRAM, 0)) < 0)

ERR\_EXIT("socket");

echo\_cli(sock);

return 0;

}

编译运行server，在两个终端里各开一个client与server交互，可以看到server具有并发服务的能力。用Ctrl+C关闭server，然后再运行server，此时client还能和server联系上。和前面TCP程序的运行结果相比较，我们可以体会无连接的含义。udp 协议来说，server与client 的界限更模糊了，只要知道对等方地址(ip和port) 都可以主动发数据。

**二、UDP编程注意点**

1、UDP报文可能会丢失、重复

2、UDP报文可能会乱序

3、UDP缺乏流量控制

4、UDP协议数据报文截断

5、recvfrom返回0，不代表连接关闭，因为udp是无连接的。

6、ICMP异步错误

7、UDP connect

8、UDP外出接口的确定

9、太大的UDP包可能出现的问题

由于UDP不需要维护连接，程序逻辑简单了很多，但是UDP协议是不可靠的，实际上有很多保证通讯可靠性的机制需要在应用层实现，即123点所提到的。比如 如果发送端速度较快，而接收端较慢，很可能会产生 ICMP Source Quench Error，丢弃一些数据包。

对于第4点，可以写个小程序测试一下：

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)int main(void)

{

int sock;

if ((sock = socket(PF\_INET, SOCK\_DGRAM, 0)) < 0)

ERR\_EXIT("socket");

struct sockaddr\_in servaddr;

memset(&servaddr, 0, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(5188);

servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

if (bind(sock, (struct sockaddr \*)&servaddr, sizeof(servaddr)) < 0)

ERR\_EXIT("bind");

sendto(sock, "ABCD", 4, 0, (struct sockaddr \*)&servaddr, sizeof(servaddr));

char recvbuf[1];

int n;

int i;

for (i = 0; i < 4; i++)

{

/\* udp是报式协议，即若一次性接收的空间小于发来的数据，有可能造成报文截断，

\* 但一定没有tcp的粘包问题 \*/

n = recvfrom(sock, recvbuf, sizeof(recvbuf), 0, NULL, NULL);

if (n == -1)

{

if (errno == EINTR)

continue;

ERR\_EXIT("recvfrom");

}

else if(n > 0)

printf("n=%d %c\n", n, recvbuf[0]);

}

return 0;

}

上述程序是自己发送数据给自己，发送了4个字节，但我们只提供1个字节的缓冲区recvbuf，第一次recvfrom 读取一个字节，但接下去循环却读不到剩下的数据了，因为udp 是报式协议，如果一次性接收的缓冲区小于发来的数据，有可能造成报文截断，反观tcp流式协议，可以一次读取一个数据包的一部分，也可以一次性读取多个数据包，但这也正是其会造成粘包问题的来源，所以也说udp 协议不会有粘包问题，因为一次就接收一个消息。输出如下：

simba@ubuntu:~/Documents/code/linux\_programming/UNP/socket$ ./trunc

n=1 A

............

接收了一个字符之后，再次recvfrom 就阻塞了。

对于第5点，如果我们使用sendto 发送的数据大小为0，则发送给对方的是只含有各层协议头部的数据帧，recvfrom 会返回0，但并不代表对方关闭连接，因为udp 本身没有连接的概念。

第678点合起来一起讲，可以看到我们的客户端程序现在没有调用connect，不运行服务器程序，直接运行客户端程序，查看现象：

simba@ubuntu:~/Documents/code/linux\_programming/UNP/socket$ ./echocli\_udp

dfsaf

................

当我们在键盘敲入几个字符，sendto只是把Buf的数据拷贝到sock对应的缓冲区中，此时服务器未开启，协议栈返回一个ICMP异步错误，但因为前面没有调用connect“建立”一个连接，则recvfrom时不能收到这个错误而一直阻塞。现在我们在while 循环的外面添加一句：connect(sock, (struct sockaddr\*)&servaddr, sizeof(servaddr)); 再次测试一下：

simba@ubuntu:~/Documents/code/linux\_programming/UNP/socket$ ./echocli\_udp

dfsaf

recvfrom: Connection refused

此时recvfrom 就能接收到这个错误而返回了，并打印错误提示。

其实connect 并没有真正建立一个连接，即没有3次握手过程，只是维护了一种状态，绑定了远程地址，因为如此在调用sendto 时也可以不指定远程地址了，如 sendto(sock, sendbuf, strlen(sendbuf), 0, NULL, 0); 甚至也可以使用send 函数

send(sock, sendbuf, strlen(sendbuf), 0);

假设现在客户端有多个ip地址，由connect 或 sendto 函数提供的远程地址的参数，系统会选择一个合适的出口，比如远程ip 是192.168.2.10, 而客户端现在的ip 有 192.168.1.32 和 192.168.2.75 那么会自动选择192.168.2.75 这个ip 出去。

关于第9点。假设现在我们发送一个8192B 的IP数据报，必须分片传输，如果此时目的地址arp 并没有缓存，那么每一片都会发起arp 请求，此时会造成 arp flooding（RFC 建议的最大发送速率是每秒一次）。此时每一片都在等待arp reply，系统实现只会将最后一片发送到目的地（ARP input queue was LIFO ），也就是说，其他片都被丢弃了。对等方的IP层当接收到第一个到来的片时（不一定是偏移为0的片）会启动定时器，如果在30～60s

内的超时时间内没有接收到所有的片，则会丢弃所有接收到的片。但需要注意的是不一定会产生 ICMP "time exceeded during reassembly" error （ICMP 超时错误类型为11，code为0表示是TTL为0超时，code为1表示对方重组分片超时），只有在已经接收到偏移为0的片，即包含udp头部的片时才会产生此种错误，因为这个时候ICMP报文的接收方通过头部（源端口号，如下ICMP超时报文的payload）才知道是哪个进程发送的这个IP报文被丢弃了。实际上有没有产生ICMP超时报文并不是那么重要，因为系统假设TCP层或者使用UDP的应用程序最终会timeout 导致重传。

