Lecture 10: UDP Sockets

(12/2/2020)

Lecture Outline

1. Timeline of function calls in a UDP session
2. The sendto and recvfrom functions
3. UDP echo client-server example
4. Discussions
5. Using connect function with UDP clients

1. Time line and functions invovled in a UDP communication session: Fig. 8.1, p.240.

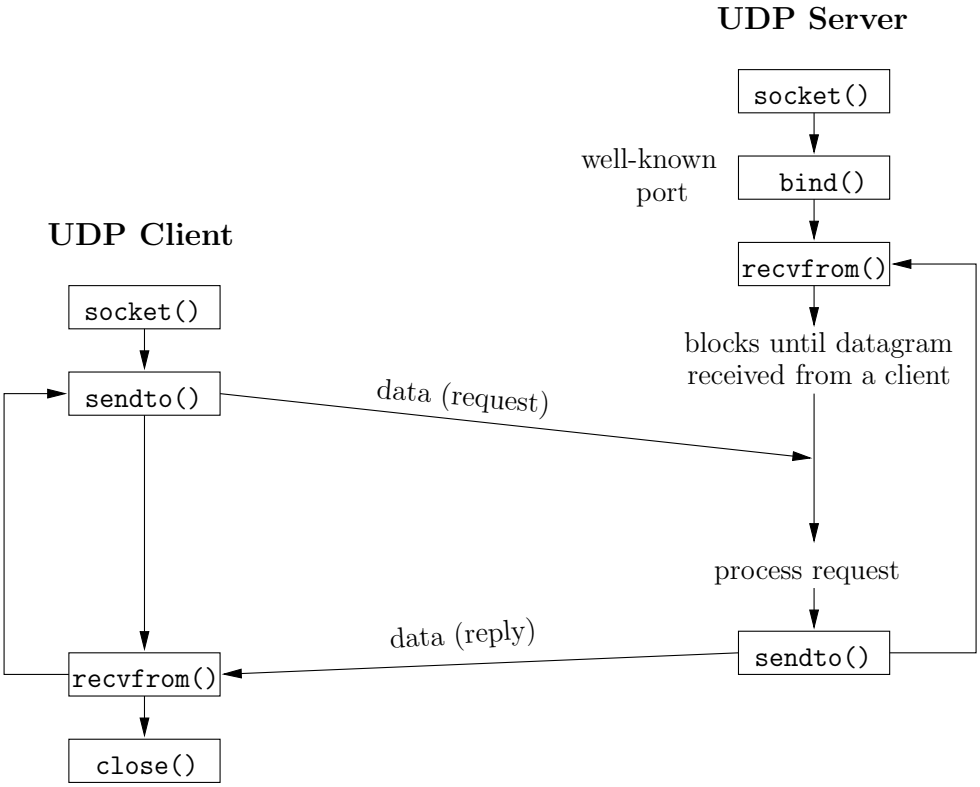

Figure 110: Socket functions for UDP client-server. (Fig.8.1,p.240)

2. *sendto* and *recvfrom* functions

网络编程send/sendto、recv/recvfrom的区别
一般情况下：
send(),recv()用于TCP，sendto()及recvfrom（）用于UDP

(1) Purpose: send or receive an UDP datagram.

(2) Syntax:

```
#include <sys/socket.h>
```

ssize_t recvfrom(int *sockfd*, void **buff*, size_t *nbytes*, int *flags*,
struct sockaddr **from*, socklen_t **addrlen*);
ssize_t sendto(int *sockfd*, const void **buff*, size_t *nbytes*, int *flags*,
const struct sockaddr **to*, socklen_t *addrlen*);

(3) semantics:

a. The *recvfrom* function reads *nbytes* from the socket specified by the 5th argument into the location pointed by the 2nd argument;

b. The 5th argument *from* and the last argument *addrlen* of the *recvfrom* function are value-result arguments;

c. The *sendto* function sends *nbytes* from the location pointed by the 2nd argument to the socket specified by the 1st and 5th arguments;

d. Both functions will return an integer as return value. If successful, a positive integer specifying the number of bytes actually received or sent is returned. On failure a -1 is returned.

3. UDP echo client-server example

(1) The model: Fig.8.2, p.241.

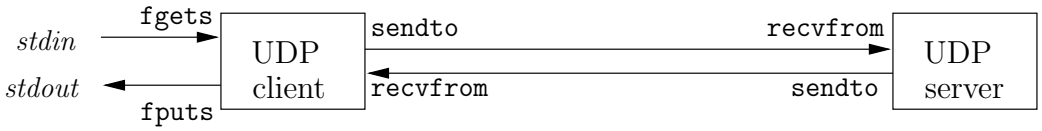

Figure 111: Simple echo client and server using UDP. (Fig.8.2,p.241)

(2) UDP server *main* function and the function *dg_echo*: Fig.8.3 and Fig.8.4, p.242

```
1  #include   "unp.h"

2  int
3  main(int argc, char **argv)
4  {
5     int               sockfd;
6     struct sockaddr_in   servaddr, cliaddr;

7     sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

8     bzero(&servaddr, sizeof(servaddr));
9     servaddr.sin_family      = AF_INET;
```

```
10      servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
11      servaddr.sin_port        = htons(SERV_PORT);

12      Bind(sockfd, (SA *) &servaddr, sizeof(servaddr));

13      dg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr));
14  }
```

Figure 8.3 UDP echo server.

```
 1  #include   "unp.h"

 2  void
 3  dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
 4  {
 5      int         n;
 6      socklen_t   len;
 7      char      mesg[MAXLINE];

 8      for ( ; ; ) {
 9          len = clilen;
10          n = Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

11          Sendto(sockfd, mesg, n, 0, pcliaddr, len);
12      }
13  }
```

Figure 8.4 `dg_echo` function: echo lines on a datagram socket.

(3) Comparison of the echo client-server example implemented using TCP and UDP: Fig.8.5 and Fig.8.6, p.243

   a. The TCP server (implemented in Chapter 5) forks child processes. We already saw that with *select* or *poll* functions it is possible for the server not to fork child processes;

   b. For TCP a client must first make a connection;

   c. A UDP server normally is an iterative server;

   d. No connection is necessary for UDP client.

(4) UDP client `main` function and the function `dg_cli`: Fig.8.7, p.244, and Fig.8.8, p.245

```
 1  #include   "unp.h"

 2  int
 3  main(int argc, char **argv)
```

```
 4  {
 5     int               sockfd;
 6     struct sockaddr_in   servaddr;

 7     if (argc != 2)
 8        err_quit("usage: udpcli <IPaddress>");

 9     bzero(&servaddr, sizeof(servaddr));
10     servaddr.sin_family = AF_INET;
11     servaddr.sin_port = htons(SERV_PORT);
12     Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

13     sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

14     dg_cli(stdin, sockfd, (SA *) &servaddr, sizeof(servaddr));

15     exit(0);
16  }
```

Figure 8.7 UDP echo client.

```
 1  #include   "unp.h"

 2  void
 3  dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 4  {
 5     int   n;
 6     char   sendline[MAXLINE], recvline[MAXLINE + 1];

 7     while (Fgets(sendline, MAXLINE, fp) != NULL) {

 8         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

 9         n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);

10         recvline[n] = 0;   /* null terminate */
11         Fputs(recvline, stdout);
12     }
13  }
```

Figure 8.8 `dg_cli` function: client processing loop.

4. Discussions of UDP client-server model

   (1) Lost datagrams and their effects

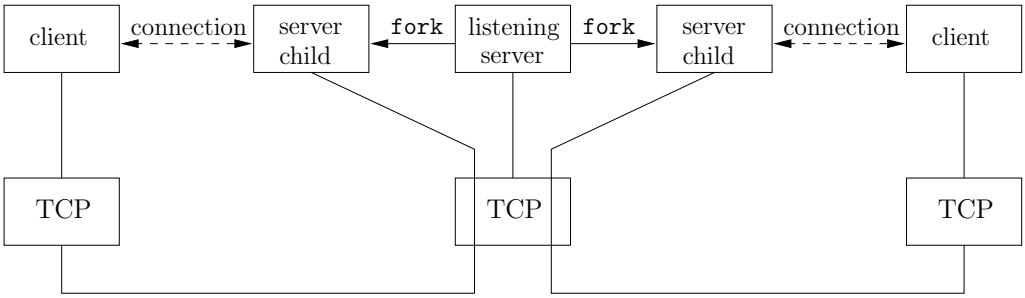

Figure 112: Summary of TCP client-sever with two clients. (Fig.8.5,p.243)

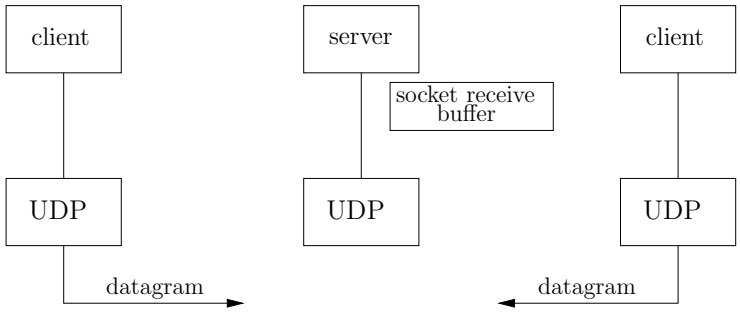

Figure 113: Summary of UDP client-sever with two clients. (Fig.8.6,p.243)

a. UDP datagrams can be lost. In our current implementation, when a request datagram or reply from the server is lost, the client will be blocked at the `recvfrom` function call in `dg_cli` trying to receive a reply;
b. One possible solution to use timeout on the client's call to `recvfrom` (Chapter 14.2);
c. However, a simple timeout will allow the client to know why an expected reply does not come back: lost request datagram or reply datagram (Sect. 22.5 covers more about this).

(2) Verifying received response

a. The UDP client as it is will receive any datagram addressed to its ephemeral port number (assigned by the client's kernel). This is a potential security hole.
b. Solution: use the 5th argument of the `recvfrom` function to verify the sender's IP and port number, and discards all faked replies that do not come from the expected IP and port.
c. Revised `dg_cli` function: Fig.8.9, p.247.

```
1  #include   "unp.h"
```

```
 2  void
 3  dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 4  {
 5     int            n;
 6     char         sendline[MAXLINE], recvline[MAXLINE + 1];
 7     socklen_t      len;
 8     struct sockaddr   *preply_addr;

 9     preply_addr = Malloc(servlen);

10     while (Fgets(sendline, MAXLINE, fp) != NULL) {

11        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

12        len = servlen;
13        n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
14        if (len != servlen || memcmp(pservaddr, preply_addr, len) != 0) {
15           printf("reply from %s (ignored)\n",
16                 Sock_ntop(preply_addr, len));
17           continue;
18        }

19        recvline[n] = 0;   /* null terminate */
20        Fputs(recvline, stdout);
21     }
22  }
```

Figure 8.9 Version of `dg_cli` function that verifies returned socket address.

d. Results of running the revised client: p.247,248

- The server has two IPs. The server kernel may use different IPs in different datagrams;
- The client uses one of the two server's IPs. It sends two requests. The first reply carries the server IP that the client used. The second IP carries a different IP of the server and is ignored by the client.

(3) Server not running

a. Server is not started. The client will send a request and be blocked at its `recvfrom` function call;

b. Understanding the underlying protocols: p.248, outcome of `tcpdump` command:

tcpdump 是一款强大的网络抓包工具，运行在 linux 平台上。熟悉 tcpdump 的使用能够帮助你分析、调试网络数据。

- The ARP protocol was used to find the Ethernet address of the server host;

地址解析协议（英語：Address Resolution Protocol，缩写：ARP）是一个通过解析网络层地址来找寻数据链路层地址的网络传输协议，

- The server host responded to the ARP request of the client host;

- The client host delivered an UDP datagram to the server host;
- The server host responded with an *udp port 9877 unreachable* error. This is an ICMP error. However it is never returned to the client process;
- The ICMP error is a result of the `sendto` function call. However, `sendto` returns ok because no connection is needed. The error occurs sometime later and hence is called an asynchronous error. It is not returned to the client process because the `sendto` function call that caused the error has finished with an OK return value;
- One solution to this problem is to use the `connect` function in the UDP client (to be discussed next).

(4) Summary of UDP example:

a. Client's perspective: Fig.8.11, p.250

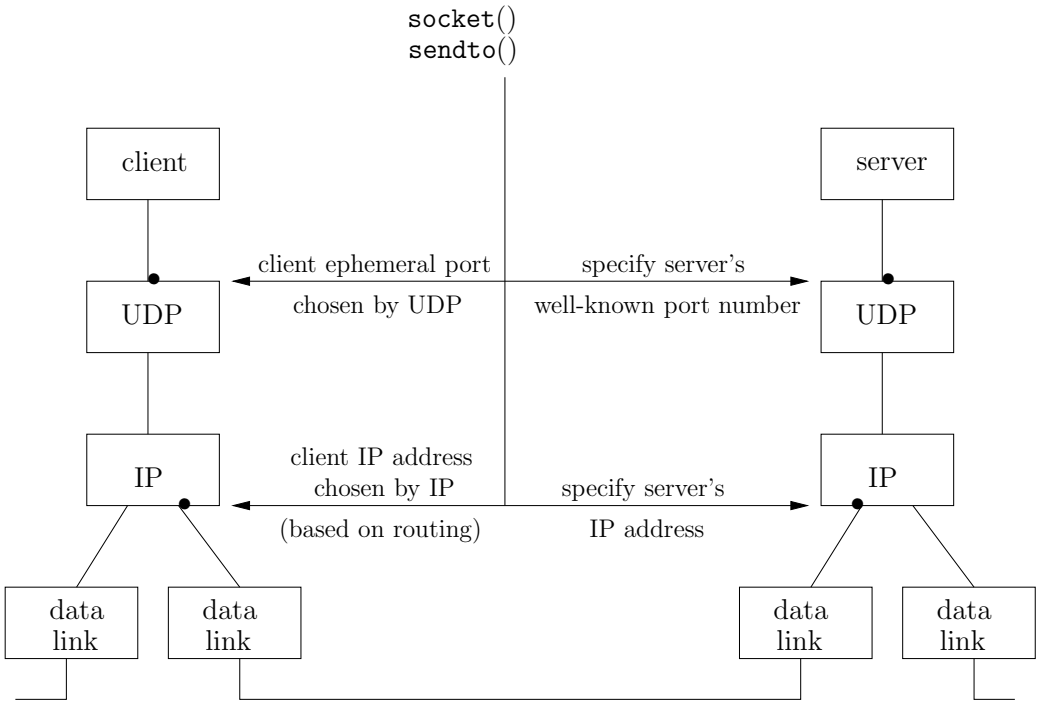

Figure 114: Summary of UDP client-server from client's perspective. (Fig.8.11,p.250)

b. Server's perspective: Fig.8.12, p.251

(5) Information available to server from an arriving IP datagram or packet: Fig.8.13, p.251

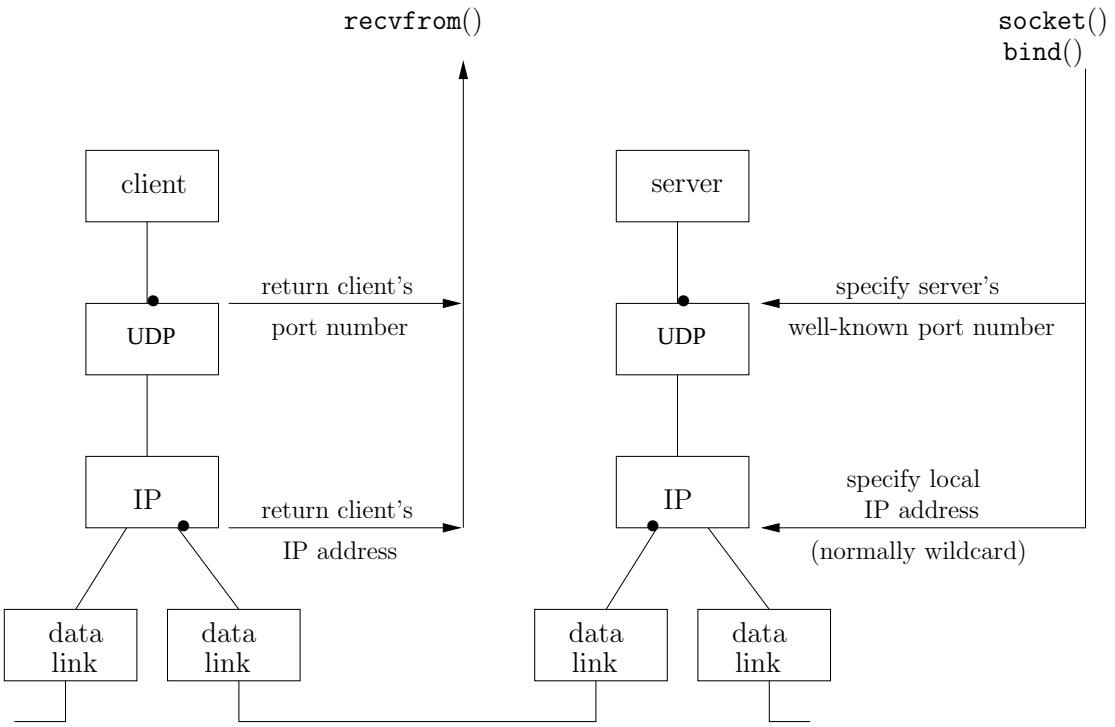

Figure 115: Summary of UDP client-server from server's perspective. (Fig.8.12,p.251)

From client's IP dg or pk	TCP server	UDP server
source IP address	`accept`	`recvfrom`
source port number	`accept`	`recvfrom`
destination IP address	`getsockname`	`recvmsg`
destination port number	`getsockname`	`getsockname`

Figure 8.13 Information available to server from arriving IP datagram.

5. Using `connect` function with UDP

   (1) An UDP client (server too. To be discussed at the end of this section) can also use `connect` function call, with different semantics. A `connect` call in an UDP client will only cause the client's kernel to store the `servaddr` specified, so that the system knows where to send future data that the process writes to the `sockfd` descriptor. Here are detailed consequences if a client makes a `connect` function call:

      a. The client cannot specify the destination address and port number in an output operation. Therefore the clinet has to use `write` or `send` (not covered yet), instead of `sendto`. Fig.8.14,p.253 summarizes this;

b. Similarly the client cannot specify the destination address and port number in an input operation. Therefore the clinet has to use `read` or `recv` (not covered yet), instead of `recvfrom`. A summary similar to Fig.8.14 can be easily constructed for the input case;

c. Only datagrams from the specified IP address and port number will be received by the socket. Therefore asynchonous errors will be returned.

Type of socket	`write` or `send`	`sendto` that does not specify a destination	`sendto` that specifies a destination
TCP socket	OK	OK	`EISCONN`
UDP socket,connected	OK	OK	`EISCONN`
UDP socket,unconnected	`EDESTADDRREQ`	`EDESTADDRREQ`	OK

Figure 8.14 TCP and UDP sockets: can a destination protocol address be specified?

(2) Illustration: a connected UDP socket: Fig.8.15, p.253.

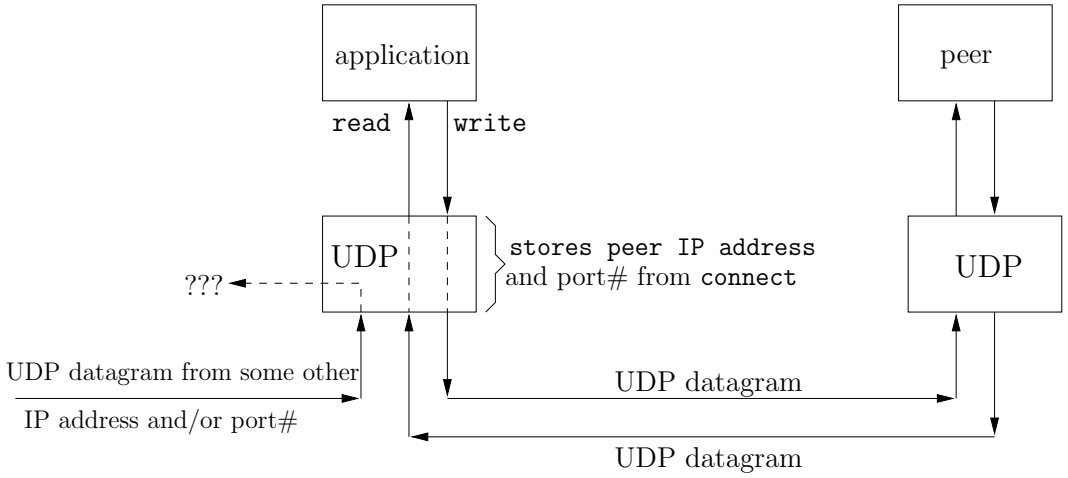

Figure 116: Connected UDP socket. (Fig.8.15,p.53)

(3) UDP Applications that use `connect` function:

a. DNS: a DNS client that is configured to use one DN server can use connect function (Fig.8.16, p.254). Notice that DNS clients configured to use two or more DNS servers are not allowed to use `connect` function, nor does a DNS server (because it has to handle any client's request).

b. Normally an UDP server will not use `connect` function. However, in rare cases where an UDP server expects to communicate with a client for an extended

period of time, the server may also use `connect` function. The TFTP is a good example.

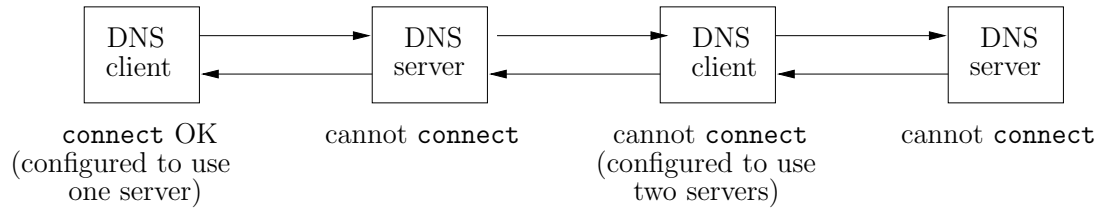

Figure 117: Example of DNS clients and servers and the `connect` function. (Fig.8.16,p.254)