CSCD433
Advanced Networks
Winter 2017
Lecture 14

Raw vs. Cooked Sockets
C Language

Introduction

- Raw Sockets in C
  - Review of Sockets in C
  - Look at how to write Raw Sockets in C
  - Contrast with Java

Sockets in C

- Normal socket operation in C is more detailed
- More steps to implement Client/Server connection
- Does not take care of details like address translation
- Programmers need to know more about network connections

Normal Socket Operations

- Look at TCP Client/Server example in C
- Then, look at a Raw Socket implementation in C
- For fun, see a Java Raw Socket example

Using Regular C Sockets

Steps to create a socket in C for Server vs. Client

- Server operations
  - Create a Socket
  - Fill in the Socket Address structure
    - Set the Port Number / IP Address
  - Name a Socket
  - Create a Socket Queue
  - Accepte Connections
    - Do something with the Connection – read or write data
  - Close a Socket

- Client operations
  - Requeste Connections
  - Use data from Server

Creating a Socket in C

- **`socket()`** system call

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

int socket (int domain, int type, int protocol);
```

- Socket system call returns a descriptor similar to low-level file descriptor
- Socket created is one end point of a communication channel

Fill in Address

- For AF_INET socket (Ipv4 socket)

```
struct sockaddr_in {
    short int sin_family;                    /* AF_INET */
    unsigned short int sin_port;  /* Port number */
    struct in_addr sin_addr;        /* IP address */
};
```

```
struct in_addr {
    unsigned long int s_addr;
};
```

Name a Socket

- AF_INET sockets are associated with an IP/ port number
- **bind()** system call assigns address specified in parameter, address, to unnamed socket and a port number

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

int bind(int socket, const struct sockaddr *address,
             size_t address_len);
```

Create a Socket Queue

- A server program must create a queue to store pending requests
- **`listen()`** system call.

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

int listen ( int socket, int backlog );
```

- Allows incoming connections to be pending while server is busy dealing with previous client
- Return values same as for bind

Accept Connections

- Wait for connections to socket by using **`accept()`** system call
- Creates a new socket to communicate with the client and returns its descriptor

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

int accept(int socket, struct sockaddr *address,
              size_t *address_len);
```

Sending or Writing to Socket

sendmsg (fd, msgstruct, flags);

fd is valid file descriptor from socket call
msgstruct which is used to pass information
to kernel like destination address and the buffer
the flags can be passed as 0

write (fd, data, length);

write is the "normal" write function; can be used with both files and sockets

Sending or Writing to Socket

send (fd, data, length, flags);
sendto (fd, data, length, flags, destaddress, addresslen);

send() is used for TCP SOCK_STREAM connected sockets, and

sendto() is used for UDP SOCK_DGRAM unconnected datagram sockets or RAW sockets

With unconnected sockets, you must specify destination of a packet each time you send one

Receiving or Reading From Socket

`recv` (int s, void *buf, size_t len, int flags);

`recvfrom` (int s, void *buf, size_t len, int
flags, struct sockaddr *from, socklen_t *fromlen);

recv() is for TCP SOCK_STREAM sockets

recvfrom() is for UDP SOCK_DGRAM or RAW sockets

- Both functions take socket descriptor s, a pointer to buffer buf, size (in bytes) of buffer, len, and a set of flags that control how the functions work
- recvfrom() takes a struct sockaddr*, from that will tell you where the data came from, and will fill in fromlen with the size of struct sockaddr

Receiving or Reading from Socket

recvmsg (int sockfd, struct msghdr *msg, int flags**);**

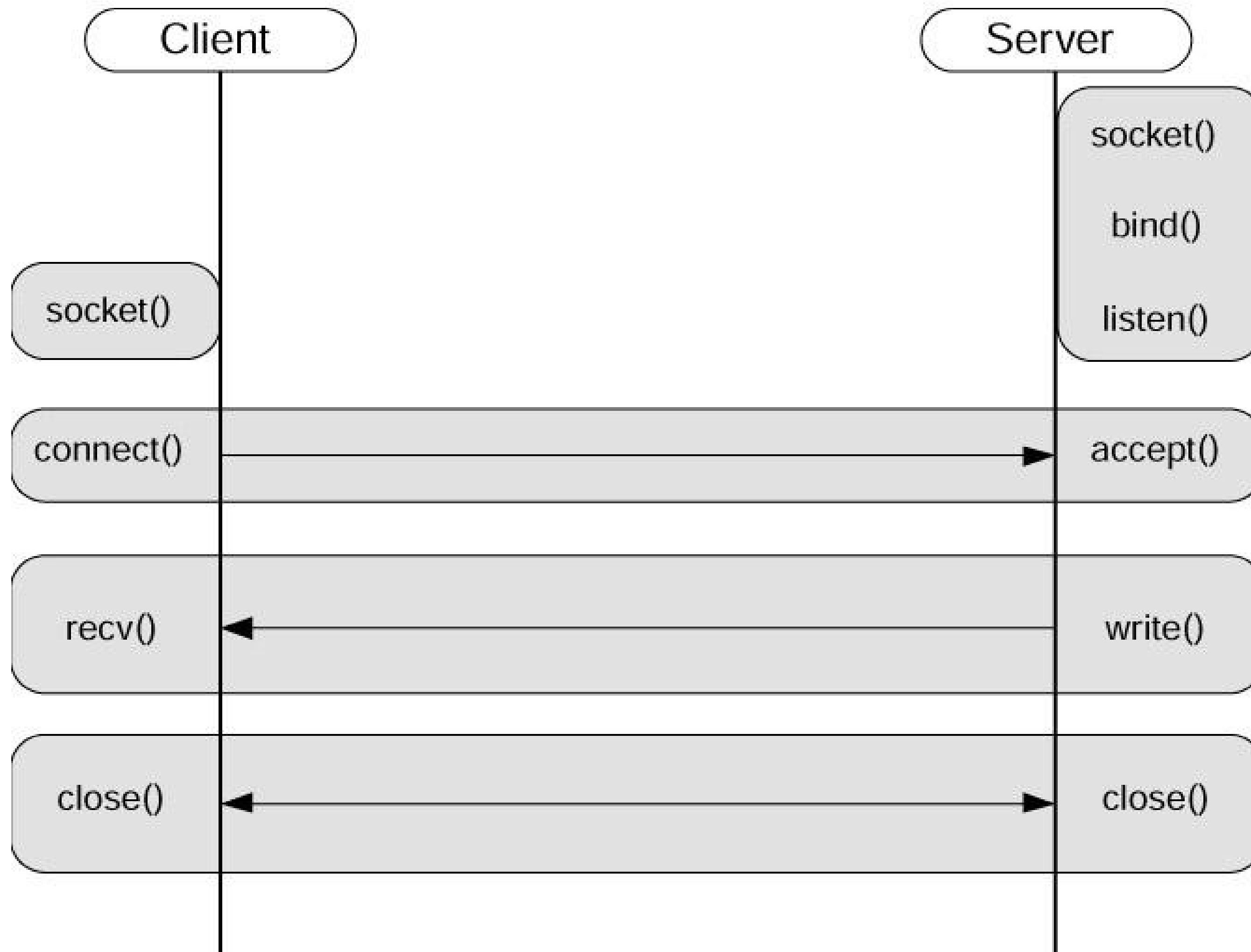
read (int fd, void *buf, size_t count);

recvmsg() offers some additional features that recv() does not ancillary data fields in the msghdr

read () attempts to read up to count bytes from file descriptor fd into the buffer starting at buf

File descriptor can also be a socket descriptor

Client-Server Request Flow



TCP Date and Time Server

TCP Date and Time Server

- Server starts
- Waits for a client to contact it
- Client connects – TCP C socket
- Server gets the Current Date and Tim
- Writes it to the Client
- Client displays it and closes connection

Server Side: Date and Time Server

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

int main  (int argc, char *argv[])
{
   int listenfd = 0,     connfd = 0;
   struct sockaddr_in serv_addr;               // Will hold Server address and port number

   char sendBuff[1025];
   time_t ticks;

   listenfd = socket (AF_INET, SOCK_STREAM, 0);  // Create a TCP socket
   memset (&serv_addr, '0', sizeof(serv_addr));
   memset (sendBuff, '0', sizeof(sendBuff));

   serv_addr.sin_family = AF_INET;                          // Set the Socket type
   serv_addr.sin_addr.s_addr = htonl (INADDR_ANY);          // Set server address
   serv_addr.sin_port = htons(5000);                        // Set server port
```

Server Side: Date and Time Server

```
bind(listenfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)); // Bind & Listen on port
listen(listenfd, 10); //

while(1)
{
    connfd = accept(listenfd, (struct sockaddr*)      //  Accept connection from client
          NULL, NULL);

    ticks = time(NULL);                                                 // Read the time and date
    snprintf(sendBuff, sizeof(sendBuff), "%.24s\r\n", ctime(&ticks));  //Put in buffer
    write(connfd, sendBuff, strlen(sendBuff));                          // Write to socket

    close(connfd);  // Close the connection
    sleep(1);
 }
}
```

Client: Date and Time

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

int main(int argc, char *argv[])
{
   int sockfd = 0, n = 0;
   char recvBuff[1024];
   struct sockaddr_in serv_addr;

   if  (argc != 2)   // Give it IP address in command line
   {
      printf("\n Usage: %s <ip of server> \n",argv[0]);
      return 1;
   }

   memset(recvBuff, '0',sizeof(recvBuff));                                // Set up Receive buffer
   if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)  // Open TCP Socket
   {
      printf("\n Error : Could not create socket \n");
      return 1;
   }

   memset(&serv_addr, '0', sizeof(serv_addr));

   serv_addr.sin_family = AF_INET;   // Set socket family type
   serv_addr.sin_port = htons(5000); // Set port number
```

Client: Date and Time

```
if (inet_pton(AF_INET, argv[1], &serv_addr.sin_addr)<=0)
 {    // Convert into network address
    printf("\n inet_pton error occured\n");
    return 1;
 }
 if( connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)  //Connect to socket at server addr
 {
   printf("\n Error : Connect Failed \n");
   return 1;
 }
 while ( (n = read(sockfd, recvBuff, sizeof(recvBuff)-1)) > 0)  // Read from the socket into buffer
 {
    recvBuff[n] = 0;
    if(fputs(recvBuff, stdout) == EOF)  // Write the buffer to the screen
    {
       printf("\n Error : Fputs error\n");
    }
 }

 if(n < 0)
 {
    printf("\n Read error \n");
 }
 return 0;
}
```

Demo

Defining a Raw Socket in C

Just like normal sockets, we create raw sockets with the socket(2) system call:

```
int socket(int domain, int type, int protocol)
```

However, type and protocol parameters are set to **SOCK_RAW** and protocol name accordingly, this example is for **ICMP**:

```
if ((sd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP)) < 0) {
        ..
}
```

Also: **IPPROTO_TCP, IPPROTO_UDP and IPPROTO_IP** and others

Raw Sockets

Two ways to define a raw socket

1. Let kernel fill in the IP header and we just create transport protocol header or

2. We are responsible for the whole packet and create both the IP header and underlying protocol headers

Creating Raw Sockets

```
int sockfd;
sockfd = socket(AF_INET, SOCK_RAW, protocol);
```

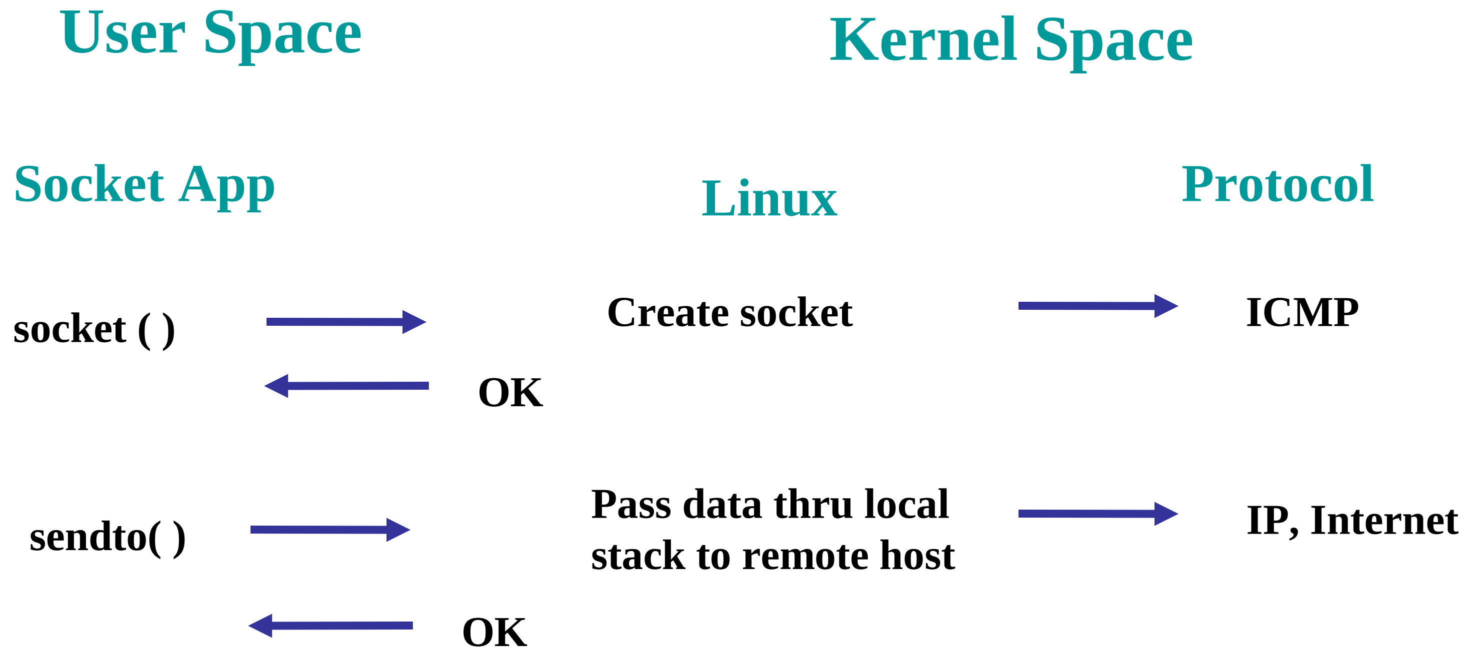
1. This creates a raw socket and lets the kernel fill in the ip header, we will fill in the protocol header for ICMP or IGMP or TCP or UDP

- -

```
const int on = 1;
setsockopt (sockfd, IPPROTO_IP, IP_HDRINCL,  &on, sizeof(on));
```

2. Setting option, IP_HDRINCL allows us to create our own IP header and not let the kernel do it for us

Raw Sockets Operation (ICMP)



Raw Socket Output

- Normal output performed using **sendto or sendmsg**
  - **Write or send** can be used if the socket has been connected
- If IP_HDRINCL not set, starting addr of data (buf) specifies first byte following IP header that kernel builds
- If IP_HDRINCL is set, starting addr of data identifies first byte of IP header, that we build

Raw Socket Input

- Most ICMP packets are passed to raw socket
  - Some exceptions for Berkeley-derived implementations
- All IGMP packets are passed to a raw socket
- All IP datagrams with protocol field that kernel does not understand (process) are passed to a raw socket
  - If packet has been fragmented, packet is reassembled before being passed to raw socket

Byte Order Issues

- Packets containing integers traveling in network are all big-endian, meaning most significant bit of the octet is transferred first
- Includes port numbers and ip addresses
- If host system uses different byte-ordering scheme
  - e.g. i386 architecture is little-endian
  - Includes data in address fields
- Data must be converted to network byte order or vice versa
- On receive path, and if host byte ordering scheme is different from network byte order,
- Data must be converted from big-endian to little-endian
- On send path, reverse operation
  - Little-endian to big-endian

Byte Ordering Functions

htons(3) and htonl(3) convert 16 bit and 32 bit quantities from host byte order into the network byte order respectively

Similarly, ntohs(3) and ntohl(3) macros convert 16-bit and 32-bit quantities from network byte order into the host byte order.

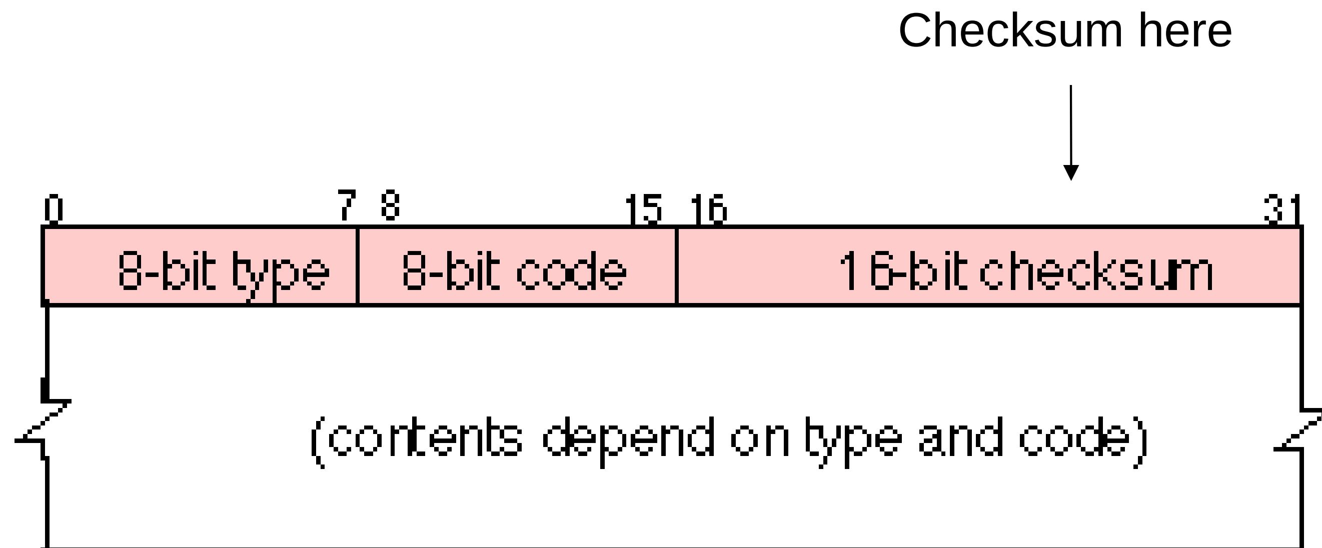
On machines which have a byte order same as the network order, routines are defined as null macros

Compute Internet Checksum

- You will also have to compute the packet checksum if you choose to construct headers yourself including ICMP
- Ready made routines to cut and paste into your code
- Checksums are normally computed for IP, ICMP, UDP and TCP packets

ICMP Header Fields

ICMP packet structure shows checksum



Raw ICMP Packet Example

- We will look at a simpler version of an ICMP program using Raw Sockets
- the Ping program !!!

Ping.c Example

```
static int in_cksum(unsigned short *buf, int sz)
{ The checksum is calculated by forming the ones' complement of the ones' complement
   sum of the header's 16-bit words
}      ……

static void ping(const char *host)
{
  struct hostent *h;            // structure stores, host name, address, any aliases for host
  struct sockaddr_in pingaddr;  // create the socket address for ping target
  struct icmp *pkt;             // ICMP Header
  int pingsock, c;
  char packet[DEFDATALEN + MAXIPLEN + MAXICMPLEN]; // max length

  if ((pingsock = socket (AF_INET, SOCK_RAW, 1)) < 0) {    // Create socket, 1 == ICMP/
    perror("ping: creating a raw socket");
    exit(1);
  }
```

Ping.c

```
pingaddr.sin_family = AF_INET; // Set domain
if (!(h = gethostbyname(host))) {  // get IP address
  fprintf(stderr, "ping: unknown host %s\n", host);
  exit(1);
}
memcpy(&pingaddr.sin_addr, h->h_addr,
  sizeof(pingaddr.sin_addr));  //Set up host addr
hostname = h->h_name;
```

ping.c

- Build the icmp packet + checksum, send it

```
pkt = (struct icmp *) packet;
memset(pkt, 0, sizeof(packet));
pkt->icmp_type = ICMP_ECHO; //Set ICMP type
pkt->icmp_cksum = in_cksum((unsigned short *) pkt,
 sizeof(packet));
 c = sendto(pingsock, packet, sizeof(packet), 0,
 (struct sockaddr *) &pingaddr, sizeof(struct sockaddr_in));
```

ping.c

```
/* listen for replies */
 while (1) {
  struct sockaddr_in from;
  size_t fromlen = sizeof(from);

  if ((c = recvfrom(pingsock, packet, sizeof(packet), 0,
                  (struct sockaddr *) &from, &fromlen)) < 0) {
   if (errno == EINTR)
     continue;
   perror("ping: recvfrom");
   continue;
  }
```

ping.c

```
if (c >= 76) {                  /* ip + icmp */
        struct iphdr *iphdr = (struct iphdr *) packet;
        pkt = (struct icmp *) (packet + (iphdr->ihl << 2));
   /* skip ip hdr */
       if (pkt->icmp_type == ICMP_ECHOREPLY)
        break;
    }
  }
  printf("%s is alive!\n", hostname);
  return;
```

ping.c

```
int main ()
{
  ping ("www.yahoo.com");


}
```

Look at Some More Examples

This URL provides good examples for each type of common network traffic and how to spoof an IP address:

http://www.enderunix.org/docs/en/rawipspoof/

Contrast with Java

```
import jpcap.JpcapHandler;
import jpcap.Jpcap;
import jpcap.Packet;

public class JpcapTip implements JpcapHandler {
    public void handlePacket(Packet packet){
        System.out.println(packet);
    }
    public static void main(String[] args) throws java.io.IOException{
        String[] devices = Jpcap.getDeviceList();
        for (int i = 0; i < devices.length; i++) {
            System.out.println(devices[i]);
        }
        String deviceName = devices[0];

        Jpcap jpcap = Jpcap.openDevice(deviceName, 1028, false, 1);
        jpcap.loopPacket(-1, new JpcapTip());
   }
}
```

JPCAP Example

The output of executing the test class looks like this

It's shortened for space:

```
ARP REQUEST 00:06:5b:01:b2:4d(192.168.15.79)
 00:00:00:00:00:00(192.168.15.34)
ARP REQUEST 00:06:5b:01:b2:4d(192.168.15.79)
 00:00:00:00:00:00(192.168.15.34)
1052251329:525479 192.168.15.103->255.255.255.255 protocol(17)
    priority(0)  hop(offset(0) ident(59244) UDP 1211  1211
```

References

Raw Socket Sources

Opensource Guide to Raw Sockets – low level

http://opensourceforu.com/2015/03/a-guide-to-using-raw-sockets/

Micro HowTo Raw Sockets

http://www.microhowto.info/howto/send_an_arbitrary_ipv4_datagram_using_a_raw_socket_in_c.html

Spoofing Raw Sockets Interface

http://www.enderunix.org/docs/en/rawipspoof/

Regular Sockets

Beej;s Guide to Network Programming

http://beej.us/guide/bgnet/output/html/multipage/index.html

Summary

- Raw Sockets in C more direct in native code
  - You have to do more work in defining headers yourself for injecting packets
  - But, you have more control
  - Gives power to you, the programmer!!!

New Assignment

Assignment 4, Packet sniffer

Using Regular C Sockets

Steps to create a socket in C for Server vs. Client

- Server operations
  - Create a Socket
  - Fill in the Socket Address structure
    - Set the Port Number / IP Address
  - Name a Socket
  - Create a Socket Queue
  - Accepte Connections
    - Do something with the Connection – read or write data
  - Close a Socket

- Client operations
  - Requeste Connections
  - Use data from Server

Creating a Socket in C

- **`socket()`** system call

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

int socket (int domain, int type, int protocol);
```

- Socket system call returns a descriptor similar to low-level file descriptor
- Socket created is one end point of a communication channel

Fill in Address

- For AF_INET socket (Ipv4 socket)

```
struct sockaddr_in {
   short int sin_family;             /* AF_INET */
   unsigned short int sin_port;  /* Port number */
   struct in_addr sin_addr;       /* IP address */
};
```

```
struct in_addr {
   unsigned long int s_addr;
};
```

Name a Socket

- AF_INET sockets are associated with an IP/ port number
- **bind()** system call assigns address specified in parameter, address, to unnamed socket and a port number

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

int bind(int socket, const struct sockaddr *address,
              size_t address_len);
```

Create a Socket Queue

- A server program must create a queue to store pending requests
- **`listen()`** system call.

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

int listen ( int socket, int backlog );
```

- Allows incoming connections to be pending while server is busy dealing with previous client
- Return values same as for bind

Accept Connections

- Wait for connections to socket by using **`accept()`** system call
- Creates a new socket to communicate with the client and returns its descriptor

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

int accept(int socket, struct sockaddr *address,
              size_t *address_len);
```

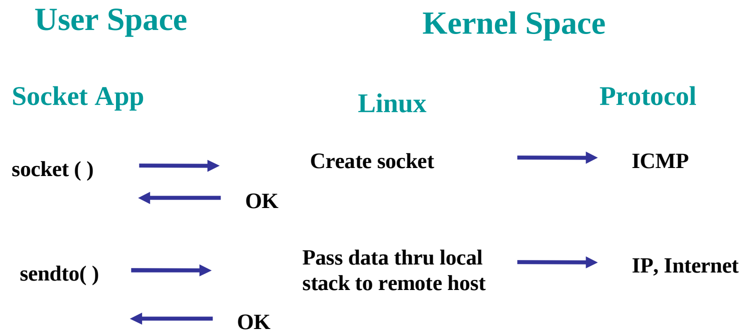
Sending or Writing to Socket

sendmsg (fd, msgstruct, flags);
fd is valid file descriptor from socket call
msgstruct which is used to pass information
to kernel like destination address and the buffer
the flags can be passed as 0

write (fd, data, length);
write is the “normal” write function; can be used with both files and sockets

Client-Server Request Flow

Raw Sockets Operation (ICMP)



Raw Socket Output

- Normal output performed using **sendto or sendmsg**
  - **Write or send** can be used if the socket has been connected
- If IP_HDRINCL not set, starting addr of data (buf) specifies first byte following IP header that kernel builds
- If IP_HDRINCL is set, starting addr of data identifies first byte of IP header, that we build

Raw Socket Input

- Most ICMP packets are passed to raw socket
  - Some exceptions for Berkeley-derived implementations
- All IGMP packets are passed to a raw socket
- All IP datagrams with protocol field that kernel does not understand (process) are passed to a raw socket
  - If packet has been fragmented, packet is reassembled before being passed to raw socket