

# UDP Sockets Programming

- Creating UDP sockets.
  - Client
  - Server
- Sending data.
- Receiving data.
- Connected Mode.

# Creating a UDP socket

```
int socket(int family,int type,int proto);
```

```
int sock;
```

```
sock = socket( PF_INET,  
              SOCK_DGRAM,  
              0 );
```

```
if (sock<0) { /* ERROR */ }
```

# Binding to well known address (typically done by server only)

```
int mysock;  
struct sockaddr_in myaddr;  
  
mysock = socket(PF_INET, SOCK_DGRAM, 0);  
myaddr.sin_family = AF_INET;  
myaddr.sin_port = htons( 1234 );  
myaddr.sin_addr = htonl( INADDR_ANY );  
  
bind(mysock, &myaddr, sizeof(myaddr));
```

# Accept ( )

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

**sockfd** is the passive mode TCP socket.

**cliaddr** is a pointer to allocated space.

**addrlen** is a value-result argument, must be set to the size of cliaddr, will be set on return to be the number of bytes in cliaddr set by the call to accept.

# Sending UDP Datagrams

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

**sockfd** is a UDP socket

**buff** is the address of the data (**nbytes** long)

**to** is the address of a `sockaddr` containing the destination address.

Return value is the number of bytes sent, or -1 on error.

# sendto ( )

- You can send 0 bytes of data!
- Some possible errors :
  - EBADF , ENOTSOCK**: bad socket descriptor
  - EFAULT**: bad buffer address
  - EMSGSIZE**: message too large
  - ENOBUFS**: system buffers are full

## More `sendto()`

- The return value of `sendto()` indicates how much data was accepted by the O.S. for sending as a datagram - not how much data made it to the destination.
- There is no error condition that indicates that the destination did not get the data!!!

# Receiving UDP Datagrams

```
ssize_t recvfrom( int sockfd,  
                  void *buff,  
                  size_t nbytes,  
                  int flags,  
                  struct sockaddr* from,  
                  socklen_t *fromaddrlen);
```

**sockfd** is a UDP socket

**buff** is the address of a buffer (**nbytes** long)

**from** is the address of a `sockaddr`.

Return value is the number of bytes received and put into **buff**, or -1 on error.

# recvfrom( )

- If `buff` is not large enough, any extra data is lost forever...
- You can receive 0 bytes of data!
- The `sockaddr` at `from` is filled in with the address of the sender.
- You should set `fromaddrlen` before calling.
- If `from` and `fromaddrlen` are NULL we don't find out who sent the data.

## More `recvfrom()`

- Same errors as `sendto`, but also:
  - **EINTR**: System call interrupted by signal.
- Unless you do something special - `recvfrom` doesn't return until there is a datagram available.

# Typical UDP client code

- Create UDP socket.
- Create `sockaddr` with address of server.
- Call `sendto()`, sending request to the server. **No call to `bind()` is necessary!**
- Possibly call `recvfrom()` (if we need a reply).

# Typical UDP Server code

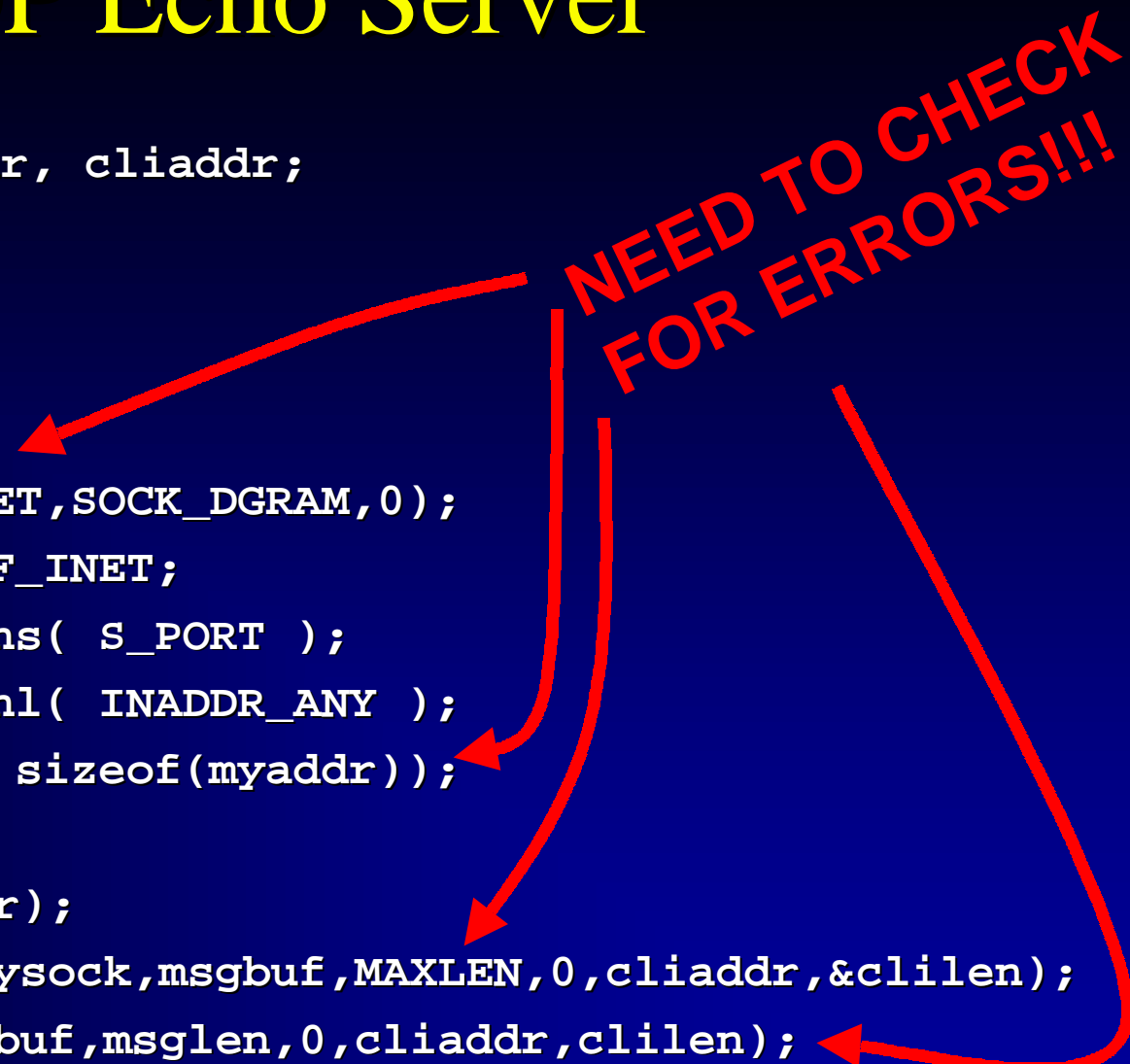
- Create UDP socket and bind to well known address.
- Call `recvfrom()` to get a request, noting the address of the client.
- Process request and send reply back with `sendto()`.

# UDP Echo Server

```
int mysock;
struct sockaddr_in myaddr, cliaddr;
char msgbuf[MAXLEN];
socklen_t clilen;
int msglen;

mysock = socket(PF_INET,SOCK_DGRAM,0);
myaddr.sin_family = AF_INET;
myaddr.sin_port = htons( S_PORT );
myaddr.sin_addr = htonl( INADDR_ANY );
bind(mysock, &myaddr, sizeof(myaddr));
while (1) {
    len=sizeof(cliaddr);
    msglen=recvfrom(mysock,msgbuf,MAXLEN,0,cliaddr,&clilen);
    sendto(mysock,msgbuf,msglen,0,cliaddr,clilen);
}
```

**NEED TO CHECK FOR ERRORS!!!**



# Debugging

- Debugging UDP can be difficult.
- Write routines to print out sockaddrs.
- Use trace, strace, ptrace, truss, etc.
- Include code that can handle unexpected situations.

# Timeout when calling `recvfrom()`

- It might be nice to have each call to `recvfrom()` return after a specified period of time even if there is no incoming datagram.
- We can do this by using `SIGALRM` and wrapping each call to `recvfrom()` with a call to `alarm()`

# recvfrom() and alarm()

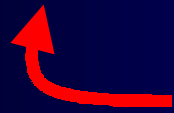
```
signal(SIGALRM, sig_alarm);
alarm(max_time_to_wait);
if (recvfrom(...)<0)
    if (errno==EINTR)
        /* timed out */
    else
        /* some other error */
else
    /* no error or time out
    - turn off alarm */
    alarm(0);
```

# Connected mode

- A UDP socket can be used in a call to **connect ( )**.
- This simply tells the O.S. the address of the peer.
- No handshake is made as with TCP to establish that the peer exists.
- No data of any kind is sent as a result of calling **connect ( )** on a UDP socket.

# Connected UDP

- Once a UDP socket is *connected*:
  - can use `sendto()` with a null dest. address
  - can use `write()` and `send()`
  - can use `read()` and `recv()`
    - only datagrams from the peer will be returned.
  - Asynchronous errors will be returned to the process.



**OS Specific, some won't do this!**

# Asynchronous Errors

- What happens if a client sends data to a server that is not running?
  - ICMP “port unreachable” error is generated by receiving host and send to sending host.
  - The ICMP error may reach the sending host after `sendto()` has already returned!
  - The next call dealing with the socket could return the error.

## Back to UDP connect()

- Connect() is typically used with UDP when communication is with a single peer only.
- Many UDP clients use connect().
- Some servers (TFTP).
- It is possible to disconnect and connect to another peer.