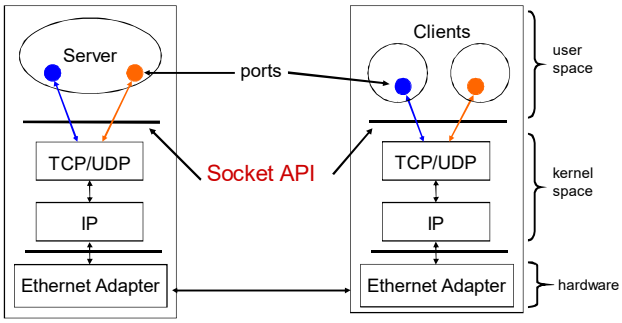
Socket programming in C

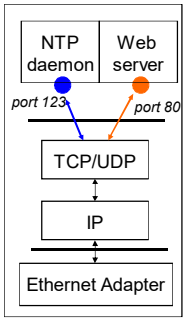
Server and Client

Server and Client exchange messages over the network through a common Socket API



Concept of Port Numbers

- Port numbers are used to identify "entities" on a host (16 bit)
- Port numbers can be
  - Well-known (port 0-1023)
  - Registered (port 1024 – 49151)
  - Dynamic or private (port 49152-65535)
- Servers/daemons usually use well-known ports
  - Any client can identify the server/service
  - HTTP = 80, FTP = 21, Telnet = 23, ...
  - /etc/services defines well-known ports
- Clients usually use dynamic ports
  - Assigned by the kernel at run time



Names and Addresses

- Each attachment point on Internet is given unique address
  - Based on location within network – like phone numbers
- Humans prefer to deal with names not addresses
  - DNS provides mapping of name to address
  - Name based on administrative ownership of host

Internet Addressing Data Structure

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

/* Internet address structure */
struct in_addr {
        u_long s_addr;     /* 32-bit IPv4 address */
};              /* network byte ordered */

/* Socket address, Internet style. */
struct sockaddr_in {
u_char  sin_family; /* Address Family */
u_short sin_port;       /* UDP or TCP Port# */
            /* network byte ordered */
struct in_addr sin_addr;    /* Internet Address */
char    sin_zero[8];    /* unused */
};
```

- sin_family = AF_INET selects Internet address family

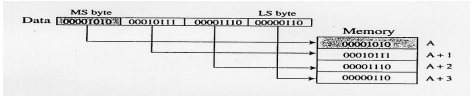
Byte Ordering

```
union {
   u_int32_t addr;  /* unsigned long int, 4
         bytes address */
   unsigned char c[4];
} un;
/* 128.2.194.95 */
un.addr = 0x8002c25f;
/* un.c[0] = ? */
```

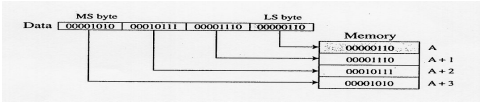
	c[0]	c[1]	c[2]	c[3]
• Big Endian →	128	2	194	95
– Sun Solaris, PowerPC, ...				
• Little Endian →	95	194	2	128
– i386, alpha, ...				

• Network byte order = Big Endian

- Byte Order
  - Different computers may have different internal representation of 16 / 32-bit integer (called **host byte order**).
  - Examples
    - Big-Endian byte order (e.g., used by Motorola 68000):



    - Little-Endian byte order (e.g., used by Intel 80x86):



Byte Ordering Functions

- Converts between **host byte order** and **network byte order**
  - 'h' = host byte order
  - 'n' = network byte order
  - 'l' = long (4 bytes), converts IP addresses
  - 's' = short (2 bytes), converts port numbers

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

unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

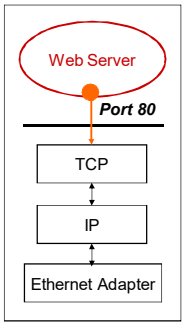
What is a Socket?

- A socket is a file descriptor that lets an application read/write data from/to the network

```
int fd;         /* socket descriptor */
if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
perror("socket");
exit(1);
}
```

- ***socket*** returns an integer (socket descriptor)
  - fd < 0 indicates that an error occurred
  - socket descriptors are similar to file descriptors
- AF_INET: associates a socket with the Internet protocol family
- SOCK_STREAM: selects the TCP protocol
- SOCK_DGRAM: selects the UDP protocol

TCP Server



- For example: web server
- **What does a *web server* need to do so that a *web client* can connect to it?**

Socket I/O: socket()

- Since web traffic uses TCP, the web server must create a socket of type SOCK_STREAM

```
int fd;          /* socket descriptor */

if((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
perror("socket");
exit(1);
}
```

- ***socket*** returns an integer (**socket descriptor**)
  - ***fd*** < 0 indicates that an error occurred
- **AF_INET** associates a socket with the Internet protocol family
- **SOCK_STREAM selects the TCP protocol**

Socket I/O: bind()

- A ***socket*** can be bound to a ***port***

```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;          /* used by bind() */

/* create the socket */

srv.sin_family = AF_INET; /* use the Internet addr family */

srv.sin_port = htons(80); /* bind socket 'fd' to port 80*/

/* bind: a client may connect to any of server addresses */
srv.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
perror("bind"); exit(1);
}
```

- **Still not quite ready to communicate with a client...**

Socket I/O: listen()

- ***listen*** indicates that the server will accept a connection

```
int fd;                     /* socket descriptor */
struct sockaddr_in srv;     /* used by bind() */

/* 1) create the socket */
/* 2) bind the socket to a port */

if(listen(fd, 5) < 0) {
perror("listen");
exit(1);
}
```

- **Still not quite ready to communicate with a client...**

Socket I/O: accept()

- ***accept*** blocks waiting for a connection

```
int fd;                     /* socket descriptor */
struct sockaddr_in srv;     /* used by bind() */
struct sockaddr_in cli;     /* client's add used by accept() */
int newfd;                  /* returned by accept() */
int cli_len = sizeof(cli);  /* used by accept() */

/* 1) create the socket */
/* 2) bind the socket to a port */
/* 3) listen on the socket */

newfd = accept(fd, (struct sockaddr*) &cli, &cli_len);
if(newfd < 0) {
perror("accept");   exit(1);
}
```

- ***accept*** returns a new socket (***newfd***) with the same properties as the original socket (***fd***)
  - ***newfd*** < 0 indicates that an error occurred

Socket I/O: accept() continued...

```
struct sockaddr_in cli;     /* used by accept() */
int newfd;                  /* returned by accept() */
int cli_len = sizeof(cli);  /* used by accept() */

newfd = accept(fd, (struct sockaddr*) &cli, &cli_len);
if(newfd < 0) {
perror("accept");
exit(1);
}
```

- How does the server know which client it is?
  - **cli.sin_addr.s_addr** contains the client's ***IP address***
  - **cli.sin_port** contains the client's ***port number***
- Now the server can exchange data with the client by using ***read*** and ***write*** on the descriptor ***newfd***.
- Why does ***accept*** need to return a new descriptor?

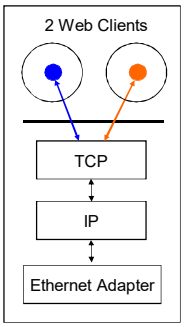
Socket I/O: read()

- ***read*** can be used with a socket
- ***read*** **blocks waiting for data from the client but does not guarantee that sizeof(buf) is read**

```
int fd;                     /* socket descriptor */
char buf[512];              /* used by read() */
int nbytes;                 /* used by read() */

/* 1) create the socket */
/* 2) bind the socket to a port */
/* 3) listen on the socket */
/* 4) accept the incoming connection */
/* 5) Create a child process to attend the connection */
/*  Child will close fd, and continue communication with client */
/*  parent will close newfd, and continue with accepting connection*/

if((nbytes = read(newfd, buf, sizeof(buf))) < 0) {
perror("read"); exit(1);
}
```

TCP Client

- For example: web client
- **How does a *web client* connect to a *web server*?**



Dealing with IP Addresses

- IP Addresses are commonly written as strings ("128.2.35.50"), but programs deal with IP addresses as integers.

Converting strings to numerical address:

```
struct sockaddr_in srv;

srv.sin_addr.s_addr = inet_addr("128.2.35.50");
if(srv.sin_addr.s_addr == (in_addr_t) -1) {
fprintf(stderr, "inet_addr failed!\n"); exit(1);
}
```

Converting a numerical address to a string:

```
struct sockaddr_in srv;
char *t = inet_ntoa(srv.sin_addr);
if(t == 0) {
fprintf(stderr, "inet_ntoa failed!\n"); exit(1);
}
```

Translating Names to Addresses

- Gethostbyname provides interface to DNS
- Additional useful calls
  - Gethostbyaddr – returns `hostent` given sockaddr_in
  - Getservbyname
    - Used to get service description (typically port number)
    - Returns `servent` based on name

```
#include <netdb.h>

struct hostent *hp; /*ptr to host info for remote*/
struct sockaddr_in peeraddr;
char *name = "www.tezu.ernet.in";

peeraddr.sin_family = AF_INET;
hp = gethostbyname(name)
peeraddr.sin_addr.s_addr = ((struct in_addr*)(hp->h_addr))->s_addr;
```

Socket I/O: connect()

- ***connect*** allows a client to connect to a server...

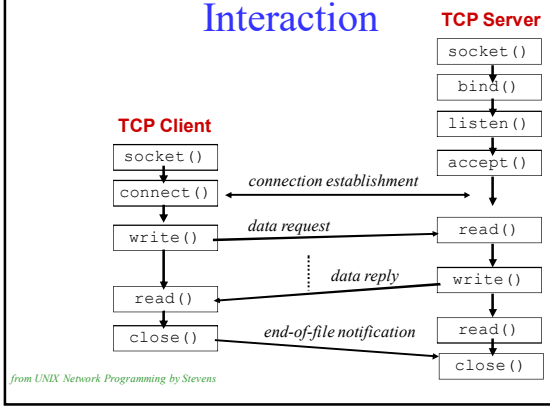
```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;          /* used by connect() */

/* create the socket */

/* connect: use the Internet address family */
srv.sin_family = AF_INET;

/* connect: socket 'fd' to port 80 */
srv.sin_port = htons(80);

/* connect: connect to IP Address "128.2.35.50" */
srv.sin_addr.s_addr = inet_addr("128.2.35.50");

if(connect(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
perror("connect"); exit(1);
}
```

Socket I/O: write()

- ***write*** can be used with a socket

```
int fd;                         /* socket descriptor */
struct sockaddr_in srv;         /* used by connect() */
char buf[512];                  /* used by write() */
int nbytes;                     /* used by write() */

/* 1) create the socket */
/* 2) connect() to the server */

/* Example: A client could "write" a request to a server
*/
if((nbytes = write(fd, buf, sizeof(buf))) < 0) {
perror("write");
exit(1);
}
```

Review: TCP Client-Server Interaction



from UNIX Network Programming by Stevens

Socket programming *with TCP*

Client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

Client contacts server by:

- creating client-local TCP socket
- specifying IP address, port number of server process
- When client creates socket: client TCP establishes connection to server TCP

- When contacted by client, server TCP creates new socket for server process to communicate with client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients.

application viewpoint

TCP provides reliable, in-order transfer of bytes ("pipe") between client and server

Example: C client (TCP)

```
/* client.c  */
void main(int argc, char *argv[])
{
struct sockaddr_in sad; /* structure to hold an IP address */
int clientSocket;  /* socket descriptor */
struct hostent *ptrh;  /* pointer to a host table entry */

char Sentence[128];
char modifiedSentence[128];

host = argv[1]; port = atoi(argv[2]);

clientSocket = socket(PF_INET, SOCK_STREAM, 0);
memset((char *)&sad,0,sizeof(sad)); /* clear sockaddr structure */
sad.sin_family = AF_INET; /* set family to Internet */
sad.sin_port = htons((u_short)port);
ptrh = gethostbyname(host); /* Convert host name to IP address */
memcpy(&sad.sin_addr, ptrh->h_addr, ptrh->h_length); connect(clientSocket,
(struct sockaddr *)&sad, sizeof(sad));
```

Create client socket, connect to server

Example: C client (TCP), cont.

Get input stream from user

```
gets(Sentence);
```

Send line to server

```
n=write(clientSocket, Sentence, strlen(Sentence)+1);
```

Read line from server

```
n=read(clientSocket, modifiedSentence, sizeof(modifiedSentence));

printf("FROM SERVER: %s\n",modifiedSentence);
```

Close connection

```
close(clientSocket);
}
```

Example: C server (TCP)

```
/* server.c */
void main(int argc, char *argv[])
{
struct sockaddr_in sad; /* structure to hold an IP address */
struct sockaddr_in cad;
int welcomeSocket, connectionSocket; /* socket descriptor */
struct hostent *ptrh; /* pointer to a host table entry */

char clientSentence[128];
char capitalizedSentence[128];

port = atoi(argv[1]);

welcomeSocket = socket(PF_INET, SOCK_STREAM, 0);
memset((char *)&sad,0,sizeof(sad)); /* clear sockaddr structure */
sad.sin_family = AF_INET; /* set family to Internet */
sad.sin_addr.s_addr = INADDR_ANY; /* set the local IP address */
sad.sin_port = htons((u_short)port);/* set the port number */
bind(welcomeSocket, (struct sockaddr *)&sad, sizeof(sad));
```

Create welcoming socket at port & Bind a local address

Example: C server (TCP), cont

```
/* Specify the maximum number of clients that can be queued */
listen(welcomeSocket, 10)

while(1) {

   connectionSocket=accept(welcomeSocket, (struct sockaddr *)&cad, &alen);

   n=read(connectionSocket, clientSentence, sizeof(clientSentence));

   /* capitalize Sentence and store the result in capitalizedSentence*/


   n=write(connectionSocket, capitalizedSentence, strlen(capitalizedSentence)+1);

   close(connectionSocket);
   }
}
```

Wait, on welcoming socket for contact by a client

Write out the result to socket

End of while loop, loop back and wait for another client connection

Socket programming *with UDP*

UDP: no "connection" between client and server

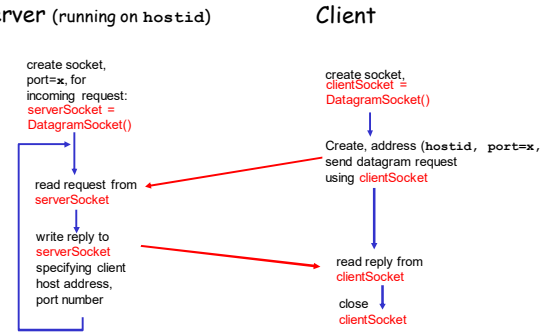
- no handshaking
- sender explicitly attaches IP address and port of destination to each packet
- server must extract IP address, port of sender from received packet

UDP: transmitted data may be received out of order, or lost

application viewpoint

UDP provides unreliable transfer of groups of bytes ("datagrams") between client and server

Client/server socket interaction: UDP



Example: C client (UDP)

```
/* client.c  */
void main(int argc, char *argv[])
{
struct sockaddr_in sad; /* structure to hold an IP address */
int clientSocket; /* socket descriptor */
struct hostent *ptrh; /* pointer to a host table entry */

char Sentence[128];
char modifiedSentence[128];

host = argv[1]; port = atoi(argv[2]);

clientSocket = socket(PF_INET, SOCK_DGRAM, 0);

/* determine the server's address */
memset((char *)&sad,0,sizeof(sad));   /* clear sockaddr structure */
sad.sin_family = AF_INET; /* set family to Internet */
sad.sin_port = htons((u_short)port);
ptrh = gethostbyname(host); /* Convert host name to IP address */
memcpy(&sad.sin_addr, ptrh->h_addr, ptrh->h_length);
```

Create client socket, NO connection to server

Example: C client (UDP), cont.

```
gets(Sentence);

addr_len =sizeof(struct sockaddr);
n=sendto(clientSocket, Sentence, strlen(Sentence)+1,
         (struct sockaddr *) &sad, addr_len);

n=recvfrom(clientSocket, modifiedSentence, sizeof(modifiedSentence),
           (struct sockaddr *) &sad, &addr_len);

printf("FROM SERVER: %s\n",modifiedSentence);

close(clientSocket);
}
```

Get input stream from user

Send line to server

Read line from server

Close connection

Example: C server (UDP)

```
/* server.c  */
void main(int argc, char *argv[])
{
struct sockaddr_in sad; /* structure to hold an IP address */
struct sockaddr_in cad;
int serverSocket; /* socket descriptor */
struct hostent *ptrh; /* pointer to a host table entry */

char clientSentence[128];
char capitalizedSentence[128];

port = atoi(argv[1]);

serverSocket = socket(PF_INET, SOCK_DGRAM, 0);
memset((char *)&sad,0,sizeof(sad)); /* clear sockaddr structure */
sad.sin_family = AF_INET;           /* set family to Internet */
sad.sin_addr.s_addr = INADDR_ANY; /* set the local IP address */
sad.sin_port = htons((u_short)port);/* set the port number */
bind(serverSocket, (struct sockaddr *)&sad, sizeof(sad));
```

Create welcoming socket at port & Bind a local address

Example: C server (UDP), cont

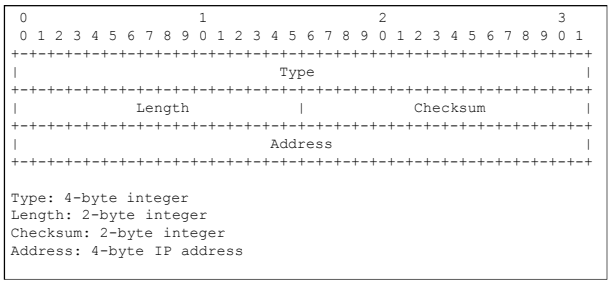
```
while(1) {

   n=recvfrom(serverSocket, clientSentence, sizeof(clientSentence), 0
              (struct sockaddr *) &cad, &addr_len );

   /* capitalize Sentence and store the result in capitalizedSentence*/

   n=sendto(connectionSocket, capitalizedSentence, strlen(capitalizedSentence)+1,0
            (struct sockaddr *) &cad, &addr_len);

   close(connectionSocket);
   }
}
```

Receive messages from clients

Write out the result to socket

End of while loop, loop back and wait for another client connection

A Few Programming Notes: Representing Packets



Type: 4-byte integer
Length: 2-byte integer
Checksum: 2-byte integer
Address: 4-byte IP address

A Few Programming Notes: Building a Packet in a Buffer

```
struct packet {
u_int32_t type;
u_int16_t length;
u_int16_t checksum;
u_int32_t address;
};

/* ================================================ */
char buf[1024];
struct packet *pkt;

pkt = (struct packet*) buf;
pkt->type = htonl(1);
pkt->length = htons(2);
pkt->checksum = htons(3);
pkt->address = htonl(4);
```

Socket Programming References

- Man page
  - usage: man <function name>
- Textbook
  - Stevens, Unix Network Programming, PHI