Sockets

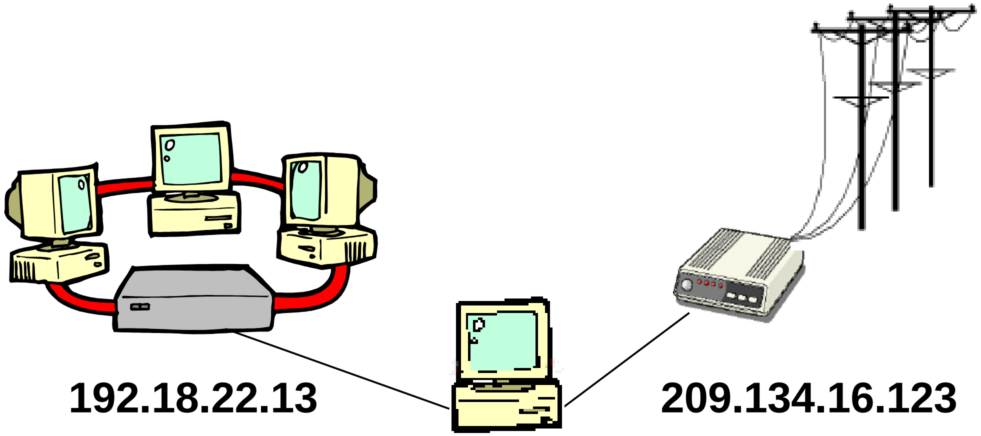
- TCP/IP Sockets in C: Practical Guide for Programmers (we are covering all chapters of Part 1; Part 2 is a good sockets API reference)

Internet Protocol (IP)

- Datagram (packet) protocol
- Best-effort service
  - Loss
  - Reordering
  - Duplication
  - Delay
- Host-to-host delivery
  (not application-to-application)

IP Address

- 32-bit identifier
- Dotted-quad: 192.118.56.25
- www.mkp.com –> 167.208.101.28
- Identifies a host interface (not a host)



Transport Protocols

Best-effort not sufficient!

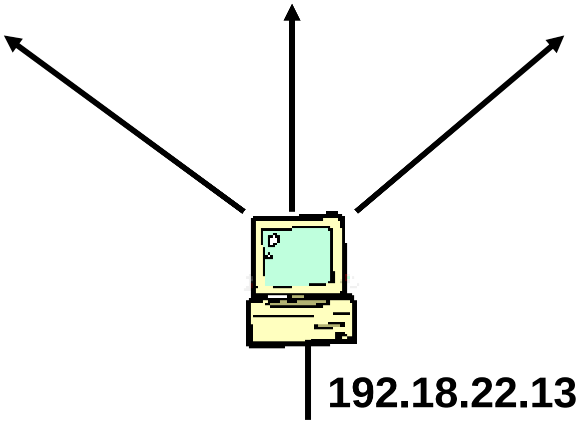
- Add services on top of IP
- User Datagram Protocol (UDP)
  - Data checksum
  - Best-effort
- Transmission Control Protocol (TCP)
  - Data checksum
  - Reliable byte-stream delivery
  - Flow and congestion control

Ports

Identifying the ultimate destination

- IP addresses identify hosts
- Host has many applications
- Ports (16-bit identifier)

Application	WWW	E-mail	Telnet
Port	80	25	23



192.18.22.13

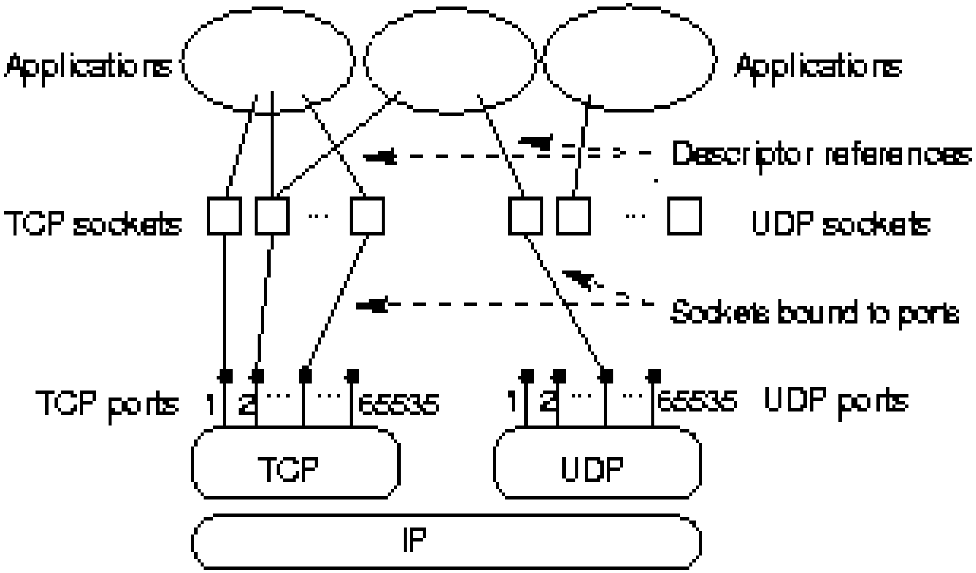
Socket

How does one speak TCP/IP?

- Sockets provides interface to TCP/IP
- Generic interface for many protocols

Sockets

- Identified by protocol and local/remote address/port
- Applications may refer to many sockets
- Sockets accessed by many applications



What is a Socket ?

- Socket **abstraction**: abstracts a mechanism for communication using different protocol families.
- Socket **API** : the interface between an Application and the TCP/IP set of protocols.

- Int socket(int protocolFamily, int type, int protocol)
  - protocolFamily: PF_INET
  - Type: SOCK_STREAM, SOCK_DGRAM, SOCK_RAW
  - Protocol: socket protocol: IPPROTO_TCP or IPPROTO_UDP

- Example: sockfd = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);

The History ….

- BSD vs Linux vs WINSOCK: different implementations but same abstraction.
- We focus on BSD. Sockets first appears in the 4.2 release. It has not changed very much although the TCP/IP stack has evolved significantly.

4.2 BSD (1982)

4.3 BSD (1986)

4.3 BSD Tahoe (1988)

4.3 BSD Reno (1990)

4.4 BSD (1993)

4.4 BSD Lite (1994)
(BSD/OS, FreeBSD
NetBSD, OpenBSD)

Abstract Address Structure

Generic Address structure defined in <sys/socket.h>

```
struct sockaddr {
            u_char   sa_len;             /* total length */
            u_char   sa_family;          /* address family */
            char     sa_data[14];        /* actually longer; address value */
};
```

Address Structure IPv4

For sa_family = AF_INET use the following (defined in <netinet/in.h>):

```
struct in_addr {
        u_int32_t s_addr;
};
struct sockaddr_in {
        u_char    sin_len;
        u_char    sin_family;    //This is the address family- not the same as PF
        u_short   sin_port;
        struct    in_addr sin_addr;
        char      sin_zero[8];
};
```

//Note: the ‘in’ implies ‘Internet’

Address Structure IPv6

```
struct sockaddr_in6 {
        sa_family_t sin6_family; // Internet protocol (AF_INET6)
        in_port_t sin6_port; // Address port (16 bits)
        uint32_t sin6_flowinfo; // Flow information
        struct in6_addr sin6_addr; // IPv6 address (128 bits)
        uint32_t sin6_scope_id; // Scope identifier
}


struct in6_addr {
        unsigned char  s6_addr[16];
};
```

TCP/IP Sockets

- mySock = socket(family, type, protocol);
- TCP/IP-specific sockets

	Family	Type	Protocol
TCP	PF_INET	SOCK_STREAM	IPPROTO_TCP
UDP		SOCK_DGRAM	IPPROTO_UDP

- Socket reference
  - – File (socket) descriptor in UNIX
  - – Socket handle in WinSock

Generic

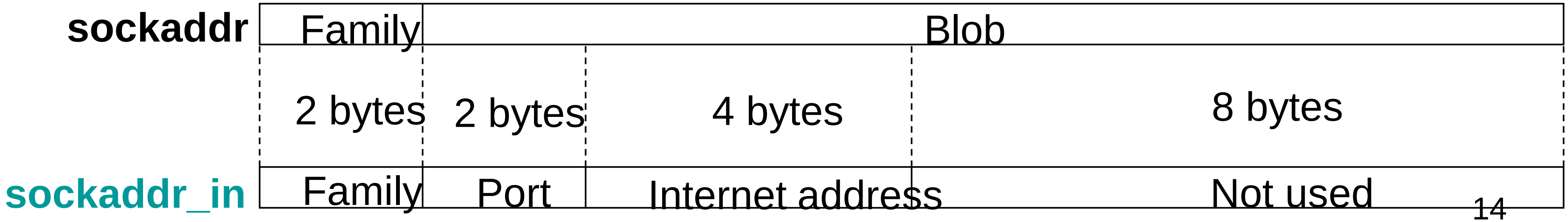
- struct sockaddr

```
{
        unsigned short sa_family;    /* Address family (e.g., AF_INET) */
        char sa_data[14];            /* Protocol-specific address information */
};
```

IP Specific

- struct sockaddr_in

```
{
        unsigned short sin_family;   /* Internet protocol (AF_INET) */
        unsigned short sin_port;     /* Port (16-bits) */
        struct in_addr sin_addr;     /* Internet address (32-bits) */
        char sin_zero[8];            /* Not used */
};
struct in_addr
{
         unsigned long s_addr;       /* Internet address (32-bits) */
};
```



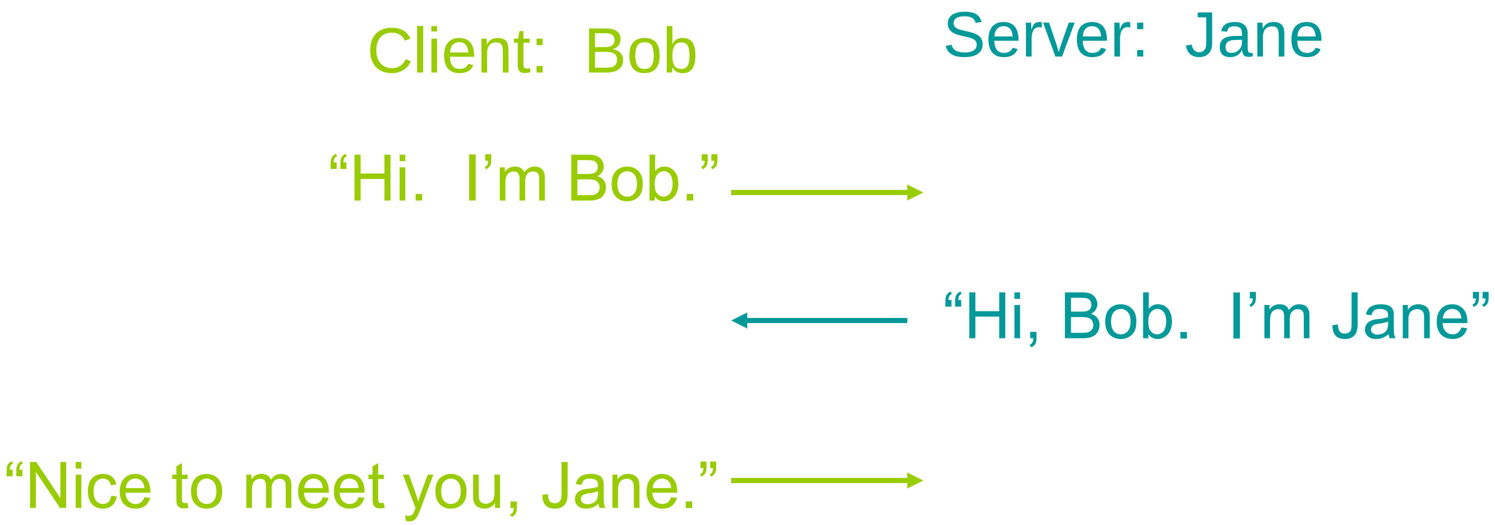
sockaddr: Family | Blob

2 bytes | 2 bytes | 4 bytes | 8 bytes

sockaddr_in: Family | Port | Internet address | Not used

Clients and Servers

- Client: Initiates the connection



Client: Bob

Server: Jane

“Hi. I’m Bob.”

“Hi, Bob. I’m Jane”

“Nice to meet you, Jane.”

- Server: Passively waits to respond

TCP Client/Server Interaction

Server starts by getting ready to receive client connections…

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
    1. Accept new connection
    2. Communicate
    3. Close the connection

TCP Client/Server Interaction

```
/* Create socket for incoming connections */
if ((servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
    DieWithError("socket() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
echoServAddr.sin_family = AF_INET;                    /* Internet address family */
echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY);/* Any incoming interface */
echoServAddr.sin_port = htons(echoServPort);          /* Local port */

if (bind(servSock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
    DieWithError("bind() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
    a. Accept new connection
    b. Communicate
    c. Close the connection

TCP Client/Server Interaction

```
/* Mark the socket so it will listen for incoming connections */
if (listen(servSock, MAXPENDING) < 0)
    DieWithError("listen() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
    a. Accept new connection
    b. Communicate
    c. Close the connection

TCP Client/Server Interaction

```
for (;;) /* Run forever */
{
    clntLen = sizeof(echoClntAddr);

    if ((clntSock=accept(servSock,(struct sockaddr *)&echoClntAddr,&clntLen)) < 0)
        DieWithError("accept() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

Server is now blocked waiting for connection from a client

Later, a client decides to talk to the server…

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
/* Create a reliable, stream socket using TCP */
if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
    DieWithError("socket() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
    a. Accept new connection
    b. Communicate
    c. Close the connection

TCP Client/Server Interaction

```
echoServAddr.sin_family      = AF_INET;              /* Internet address family */
echoServAddr.sin_addr.s_addr = inet_addr(servIP); /* Server IP address */
echoServAddr.sin_port        = htons(echoServPort); /* Server port */

if (connect(sock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
    DieWithError("connect() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
if ((clntSock=accept(servSock,(struct sockaddr *)&echoClntAddr,&clntLen)) < 0)
        DieWithError("accept() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
    a. Accept new connection
    b. Communicate
    c. Close the connection

TCP Client/Server Interaction

```
echoStringLen = strlen(echoString);          /* Determine input length */

/* Send the string to the server */
   if (send(sock, echoString, echoStringLen, 0) != echoStringLen)
       DieWithError("send() sent a different number of bytes than expected");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
/* Receive message from client */
if ((recvMsgSize = recv(clntSocket, echoBuffer, RCVBUFSIZE, 0)) < 0)
    DieWithError("recv() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
    a. Accept new connection
    b. Communicate
    c. Close the connection

TCP Client/Server Interaction

close(sock);

close(clntSocket)

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Tidbits

- Client must know the server’s address and port
- Server only needs to know its own port
- No correlation between `send()` and `recv()`

Client	Server
send(“Hello Bob”)	
	recv() -> “Hello ”
	recv() -> “Bob”
	send(“Hi ”)
	send(“Jane”)
recv() -> “Hi Jane”	

Closing a Connection

- close() used to delimit communication
- Analogous to EOF

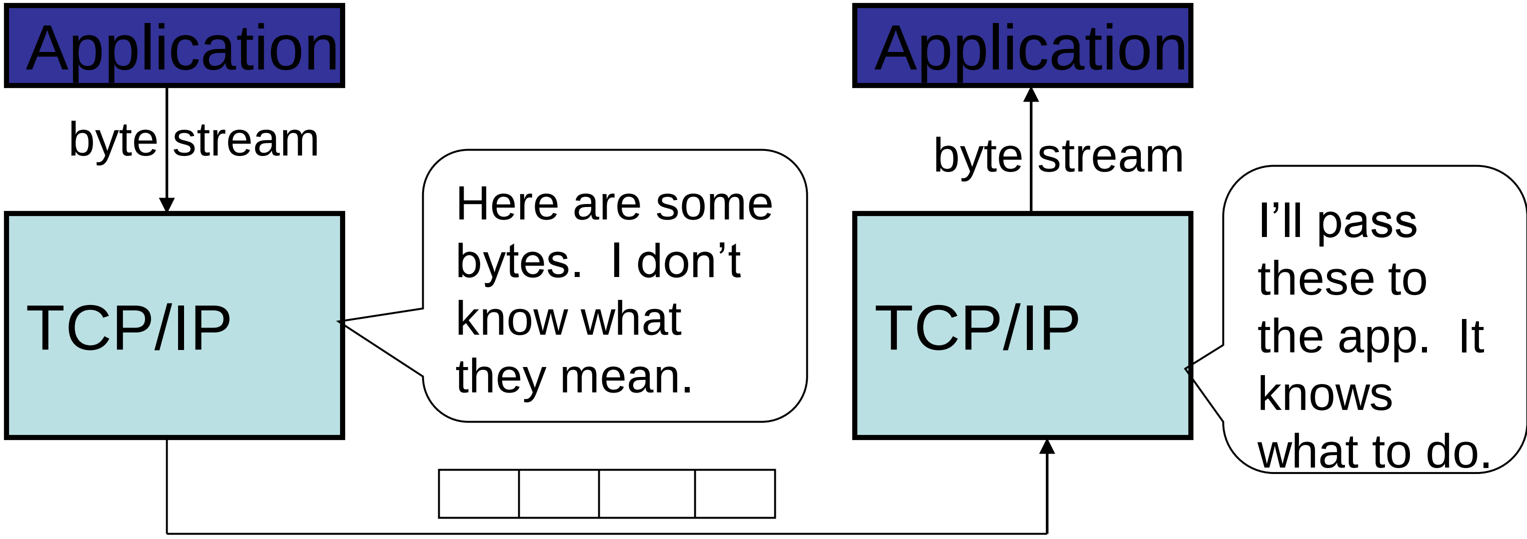
```
               Echo Client                 Echo Server
send(string)
                                           recv(buffer)
while (not received entire string)         while(client has not closed connection)
    recv(buffer)                               send(buffer)
    print(buffer)                              recv(buffer)

close(socket)
                                           close(client socket)
```

Constructing Messages

…beyond simple strings

TCP/IP Byte Transport

- TCP/IP protocols transports bytes


Application
byte stream
TCP/IP
Here are some bytes. I don't know what they mean.
Application
byte stream
TCP/IP
I'll pass these to the app. It knows what to do.

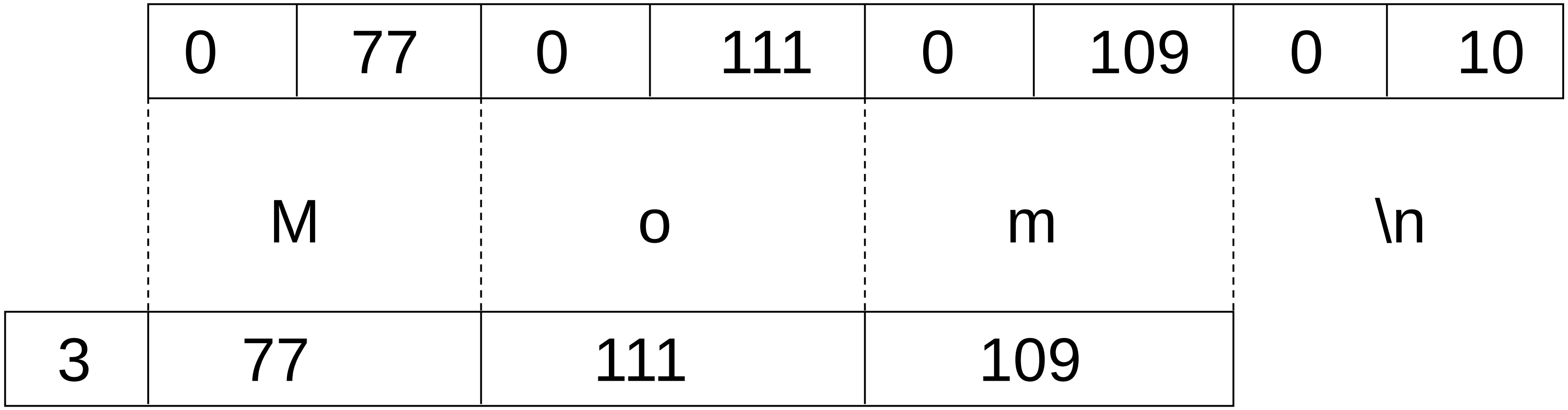
- Application protocol provides semantics

Application Protocol

- Encode information in bytes
- Sender and receiver must agree on semantics
- Data encoding
  - Primitive types: strings, integers, and etc.
  - Composed types: message with fields

Primitive Types

- String
  - Character encoding: ASCII, Unicode, UTF
  - Delimit: length vs. termination character



0	77	0	111	0	109	0	10

M	o	m	\n

3	77	111	109

Primitive Types

- Integer
  - – Strings of character encoded decimal digits

49	55	57	57	56	55	48	10
‘1’	‘7’	‘9’	‘9’	‘8’	‘7’	‘0’	\n

- Advantage: 1. Human readable
  2. Arbitrary size
- Disadvantage: 1. Inefficient
  2. Arithmetic manipulation

Primitive Types

- Integer
  - Native representation

Little-Endian	0	0	92	246

23,798

Big-Endian	246	92	0	0

4-byte
two’s-complement
integer

  - Network byte order (Big-Endian)
    - Use for multi-byte, binary data exchange
    - htonl(), htons(), ntohl(), ntohs()

Message Composition

- Message composed of fields
  - Fixed-length fields

integer	short	short

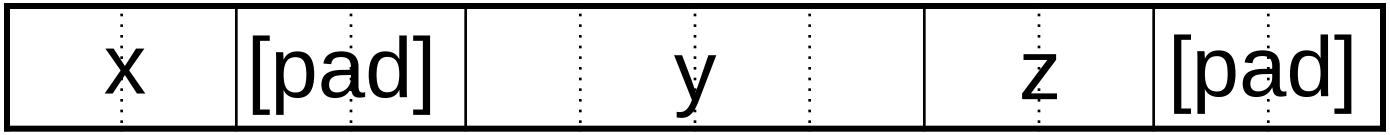
  - Variable-length fields

M	i	k	e		1	2	\n

“Beware the bytes of padding”
-- Julius Caesar, Shakespeare

- Architecture alignment restrictions
- Compiler pads structs to accommodate

```
struct tst {
  short x;
  int y;
  short z;
};
```



- Problem: Alignment restrictions vary
- Solution: 1) Rearrange struct members

  2) Serialize struct by-member

UDP Client / Server Interaction

UDP Client	UDP Server
	socket()
	bind()
	recvfrom()
socket()	
sendto()	
	sendto()
recvfrom()	
close()	close()

UDP Socket Calls

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

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

flags: nonblocking, only peek at data

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

flags: nonblocking

The BIND

- **Bind: Select a local address and port (or chose defaults)**
    - int bind(int socketfd, const sockaddr *localaddr, addrLen)

A server would typically set the sockaddr with a wildcard and valid port

```
struct sockaddr_in, servaddr;
int listenfd;

listenfd = socket(PF_INET, SOCK_STREAM, IPPROTO_UDP);
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family      = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port        = htons(TEST_PORT);

bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr));
```

Sockets

```
int getsockopt(int s, int level, int optname, void *optval, int *optlen)
int setsockopt(int s, int level, int optname, const void *optval, int optlen)
```

SO_DEBUG	**enables recording of debugging information**
SO_REUSEADDR	**enables local address reuse**
SO_REUSEPORT	**enables duplicate address and port bindings**
SO_KEEPALIVE	**enables keep connections alive**
SO_DONTROUTE	**enables routing bypass for outgoing messages**
SO_LINGER	**linger on close if data present**
SO_BROADCAST	**enables permission to transmit broadcast messages**
SO_OOBINLINE	**enables reception of out-of-band data in band**
SO_SNDBUF	**set buffer size for output**
O_RCVBUF	**set buffer size for input**
SO_SNDLOWAT	**set minimum count for output**
SO_RCVLOWAT	**set minimum count for input**
SO_SNDTIMEO	**set timeout value for output**
SO_RCVTIMEO	**set timeout value for input**
SO_TYPE	**get the type of the socket (get only)**
SO_ERROR	**get and clear error on the socket**

Helper Functions : to convert between IP addresses and dotted decimal

- inet_aton, inet_ntoa, inet_addr convert from an IPV4 dotted decimal string (“152.1.80.3”) to a 32 bit IP address.
- Inet_pton and inet_ntop:
  - Replacements for inet_aton and inet_ntoa.
  - Convert between “presentation” (ascii) and “numeric” (binary)
  - Supports IPV4 and IPV6

Helper Functions : to convert between IP addresses and names

gethostbyname, gethostbyaddr : convert between hostnames and IP addresses

```
struct hostent *  gethostbyname (const char *name)

struct hostent {
  char *h_name;   //official name of host
  char **h_aliases; //alias list
  int h_addrtype;   //host address type
  int h_length;     //length of address
  char **h_addr_list;  //list of addresses from name server
}
```

- getservbyname, getservbyport : converts between services and ports

UDPEchoClient1.c (Ch 4 of Donahoo)

```
#include "UDPEcho.h"
#include <signal.h>

void clientCNTCCode();

int main(int argc, char *argv[])
{
    int sock;                        /* Socket descriptor */
    struct sockaddr_in echoServAddr; /* Echo server address */
    struct sockaddr_in fromAddr;     /* Source address of echo */
    struct hostent *thehost;         /* Hostent from gethostbyname() */
    unsigned short echoServPort;     /* Echo server port */
    unsigned int fromSize;           /* In-out of address size for recvfrom() */
    char *servIP;                    /* IP address of server */
    char *echoString;                /* String to send to echo server */
    char echoBuffer[ECHOMAX+1];      /* Buffer for receiving echoed string */
    int echoStringLen;               /* Length of string to echo */
    int respStringLen;               /* Length of received response */

    if ((argc < 3) || (argc > 4))    /* Test for correct number of arguments */
    {
        fprintf(stderr,"Usage: %s <Server IP> <Echo Word> [<Echo Port>]\n", argv[0]);
        exit(1);
    }
    signal (SIGINT, clientCNTCCode);
    servIP = argv[1];           /* First arg: server IP address (dotted quad) */
    echoString = argv[2];       /* Second arg: string to echo */

    if ((echoStringLen = strlen(echoString)) > ECHOMAX)  /* Check input length */
        DieWithError("Echo word too long");
```

UDPEchoClient1.c (Ch 4 of Donahoo)

```
if (argc == 4)
        echoServPort = atoi(argv[3]);  /* Use given port, if any */
    else
        echoServPort = 7;  /* 7 is the well-known port for the echo service */

    /* Create a datagram/UDP socket */
    if ((sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
        DieWithError("socket() failed");

    /* Construct the server address structure */
    memset(&echoServAddr, 0, sizeof(echoServAddr));    /* Zero out structure */
    echoServAddr.sin_family = AF_INET;                 /* Internet addr family */
    echoServAddr.sin_addr.s_addr = inet_addr(servIP);  /* Server IP address */
    echoServAddr.sin_port   = htons(echoServPort);     /* Server port */

    /* If user gave a dotted decimal address, we need to resolve it  */
    if (echoServAddr.sin_addr.s_addr == -1) {
        thehost = gethostbyname(servIP);
            echoServAddr.sin_addr.s_addr = *((unsigned long *) thehost->h_addr_list[0]);
    }
```

UDPEchoClient1.c (Ch 4 of Donahoo)

```
/* Send the string to the server */
    printf("UDPEchoClient: Send the string: %s to the server: %s \n", echoString,servIP);
    if (sendto(sock, echoString, echoStringLen, 0, (struct sockaddr *)
               &echoServAddr, sizeof(echoServAddr)) != echoStringLen)
      DieWithError("sendto() sent a different number of bytes than expected");

    /* Recv a response */
    printf("UDPEchoClient: And now wait for a response... \n");
    fromSize = sizeof(fromAddr);
    if ((respStringLen = recvfrom(sock, echoBuffer, ECHOMAX, 0,
         (struct sockaddr *) &fromAddr, &fromSize)) != echoStringLen)
        DieWithError("recvfrom() failed");

    if (echoServAddr.sin_addr.s_addr != fromAddr.sin_addr.s_addr)
    {
        fprintf(stderr,"Error: received a packet from unknown source \n");
    }
    /* null-terminate the received data */
    echoBuffer[respStringLen] = '\0';
    printf("UDPEchoClient:  Received the following data: %s\n",echoBuffer);
    close(sock);
    exit(0);
}

void clientCNTCCode() {

  printf("UDPEchoClient:  CNT-C Interrupt,  exiting....\n");
}
```

Example Iterative Server: UDPEchoServer.c

```
int sock;                          /* Socket */
struct sockaddr_in echoServAddr; /* Local address */
struct sockaddr_in echoClntAddr; /* Client address */
unsigned int cliAddrLen;          /* Length of incoming message */
char echoBuffer[ECHOMAX];         /* Buffer for echo string */
unsigned short echoServPort;      /* Server port */
int recvMsgSize;                  /* Size of received message */
 if ((sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
        DieWithError("socket() failed");

    /* Construct local address structure */
    memset(&echoServAddr, 0, sizeof(echoServAddr));   /* Zero out structure */
    echoServAddr.sin_family = AF_INET;                /* Internet address family */
    echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY); /* Any incoming interface */
    echoServAddr.sin_port = htons(echoServPort);      /* Local port */

    /* Bind to the local address */
    printf("UDPEchoServer: About to bind to port %d\n", echoServPort);
    if (bind(sock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
        DieWithError("bind() failed");
```

Example Iterative Server: EchoUDPServer.c

```
for (;;) /* Run forever */
  {
     /* Set the size of the in-out parameter */
     cliAddrLen = sizeof(echoClntAddr);

     /* Block until receive message from a client */
     if ((recvMsgSize = recvfrom(sock, echoBuffer, ECHOMAX, 0,
         (struct sockaddr *) &echoClntAddr, &cliAddrLen)) < 0)
         DieWithError("recvfrom() failed");

     printf("Handling client %s\n", inet_ntoa(echoClntAddr.sin_addr));

     /* Send received datagram back to the client */
     if (sendto(sock, echoBuffer, recvMsgSize, 0,
          (struct sockaddr *) &echoClntAddr, sizeof(echoClntAddr)) != recvMsgSize)
         DieWithError("sendto() sent a different number of bytes than expected");
  }
```

Socket Select

- Programs such as Inetd need to manage multiple sockets
- Unix provides a mechanism to handle this: the select()
    - Allows a program to specify a list of socket descriptors
    - When one of the descriptors becomes ready to perform I/O it returns an indication of which descriptors are ready

```
int select (int maxDescPlus1, fd_set *readDescs, fd_set *writeDescs,
            fd_set *exceptionDescs, struct timeval *timeout)
```

maxDescPlus1: max number descriptor plus 1
e.g., if descriptors 0,3,5 are in the list, maxDescPlus1 is 6
readDescs: Descriptors in this vector are checked for input data
writeDescs: Descriptors in this vector are checked for the ability to write data
exceptionDescs: Descriptors in this vector are checked for pending exceptions
Struct timeval *timeout : controls how long the select is allowed to wait

Socket Select

MACROS

FD_ZERO(fd_set *descriptorVector)   //clears all descriptors
FD_CLEAR(int descriptor, fd_set *descriptorVector) //removes a desc
FD_SET(int descriptor, fd_set *descriptorVector)   //Adds a desc

Example (section 5.5 of Donahoo book): A TCP echo server running on multiple ports. We create a socket for each port and insert them in the readDescriptor list. Pass a NULL for the writeDescriptor and exceptDescriptor as we are not interested in these events. The select() blocks until data is ready to be read on one or more sockets.

The example is slightly strange in that the server program runs until any key from standard in is entered. The descriptor associated with this port is added to the readDescriptor list (source line # 51).

Socket Select

```
#include "TCPEchoServer.h"  /* TCP echo server includes */
#include <sys/time.h>       /* for struct timeval {} */
#include <fcntl.h>          /* for fcntl() */

int main(int argc, char *argv[])
{
    int *servSock;                   /* Socket descriptors for server */
    int maxDescriptor;                   /* Maximum socket descriptor value */
    fd_set sockSet;                  /* Set of socket descriptors for select() */
    long timeout;                    /* Timeout value given on command-line */
    struct timeval selTimeout;       /* Timeout for select() */
    int running = 1;                 /* 1 if server should be running; 0 otherwise */
    int noPorts;                     /* Number of port specified on command-line */
    int port;                        /* Looping variable for ports */
    unsigned short portNo;           /* Actual port number */

    if (argc < 3)     /* Test for correct number of arguments */
    {
        fprintf(stderr, "Usage:  %s <Timeout (secs.)> <Port 1> ...\n", argv[0]);
        exit(1);
    }

    timeout = atol(argv[1]);        /* First arg: Timeout */
    noPorts = argc - 2;             /* Number of ports is argument count minus 2 */
```

Socket Select

```
/* Allocate list of sockets for incoming connections */
servSock = (int *) malloc(noPorts * sizeof(int));
/* Initialize maxDescriptor for use by select() */
maxDescriptor = -1;

/* Create list of ports and sockets to handle ports */
for (port = 0; port < noPorts; port++)
{
    /* Add port to port list */
    portNo = atoi(argv[port + 2]);  /* Skip first two arguments */

    /* Create port socket */
    servSock[port] = CreateTCPServerSocket(portNo);

    /* Determine if new descriptor is the largest */
    if (servSock[port] > maxDescriptor)
        maxDescriptor = servSock[port];
}
```

Socket Select

```
printf("Starting server:  Hit return to shutdown\n");
    while (running)
    {
        /* Zero socket descriptor vector and set for server sockets */
        /* This must be reset every time select() is called */
        FD_ZERO(&sockSet);
        /* Add keyboard to descriptor vector */
        FD_SET(STDIN_FILENO, &sockSet);
        for (port = 0; port < noPorts; port++)
            FD_SET(servSock[port], &sockSet);

        /* Timeout specification */
        /* This must be reset every time select() is called */
        selTimeout.tv_sec = timeout;       /* timeout (secs.) */
        selTimeout.tv_usec = 0;           /* 0 microseconds */
```

Socket Select

```
/* Suspend program until descriptor is ready or timeout */
    if (select(maxDescriptor + 1, &sockSet, NULL, NULL, &selTimeout) == 0)
      printf("No echo requests for %ld secs...Server still alive\n", timeout);
    else
    {
      if (FD_ISSET(0, &sockSet)) /* Check keyboard */
      {
        printf("Shutting down server\n");
        getchar();
        running = 0;
      }

      for (port = 0; port < noPorts; port++)
        if (FD_ISSET(servSock[port], &sockSet))
        {
          printf("Request on port %d:  ", port);
          HandleTCPClient(AcceptTCPConnection(servSock[port]));
        }
    }
  }
  /* Close sockets */
  for (port = 0; port < noPorts; port++)
    close(servSock[port]);

  /* Free list of sockets */
  free(servSock);

  exit(0);
}
```

Client/Server Paradigm

- Server (any program that offers a service that can be reached over a network):
  - Open a port
  - Wait for a client
  - Start slave
  - Continue
- Client (a program that sends a request to a server and waits for the response):
  - Open a port
  - Connect with a server
  - Interact with the sever
  - Close

Client/Server Paradigm

- Concepts to be aware of:
  - Standard vs nonStandard services (e.g., ftp vs UDPEchoServer)
  - Connectionless vs connection-oriented
  - Stateless vs stateful
  - Concurrent vs iterative (server)
    - Concurrent : if the server handles multiple requests concurrently
    - Iterative: “one request at a time”.

TCP Sockets :
Client / Server Interaction

TCP Client	TCP Server
	socket()
	bind()
	listen()
	accept()
socket()	
connect()	
write()	read()
read()	write()
close()	read()
	close()

TCP Socket Calls

- int connect(int sockfd, const struct sockaddr *dstaddr, int addrlen)
  - used by TCP to establish a connection
- ssize_t read(int sockfd, void *buf, size_t nbytes)
- ssize_t write(int sockfd, void *buf, size_t nbytes)

TCP Socket Calls

- Listen: called only by a TCP server.
    - Converts an active socket into a passive socket meaning the kernel should accept incoming connection requests.
    - Sets the maximum number of connections the kernel should queue for this socket.

- int listen (int sockfd, int backlog)
    - There are two queues:
        - incompleted cx queue
        - completed cx queue
    - the backlog (roughly) indicates the sum of the two queues

TCP Socket Calls

- Accept: Called only by a TCP server to return the next completed connection from the completed queue.
- int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addren)
- Returns a new socket descriptor. Thus a server will have a listenfd and a connectfd.

Send socket call

- int send(int socket, const void*msg, unsigned int msgLength, int flags)
    - socket: must be connected!!
    - msg : ptr to data
    - msgLength : # bytes to send
    - flags : control flags (0 unless you know what you are doing!!)

    - Returns number of bytes sent, -1 otherwise
- If the msgLength exceeds socket send buffer size, the process blocks until all data can be accepted by TCP.

Recv() socket call

- int recv(int socket, const void*rcvBuffer, unsigned int bufferLength, int flags)
    - socket: must be connected!!
    - msg : ptr to where data is to go
    - bufferLength : max number of bytes to rx
    - flags : control flags (0 unless you know what you are doing!!)

    - Returns number of bytes received, -1 on error, a 1 if the other side disconnects.

- Note: you might have to loop until all bufferLength bytes arrive