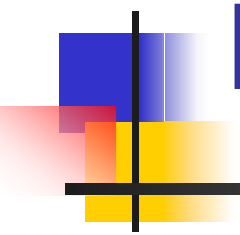
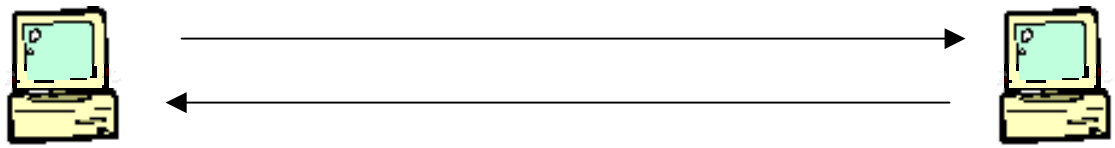
TCP/IP Sockets in C: Practical Guide for Programmers



Michael J. Donahoo
Kenneth L. Calvert

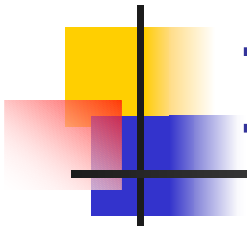
Computer Chat

- How do we make computers talk?



- How are they interconnected?

Internet Protocol (IP)

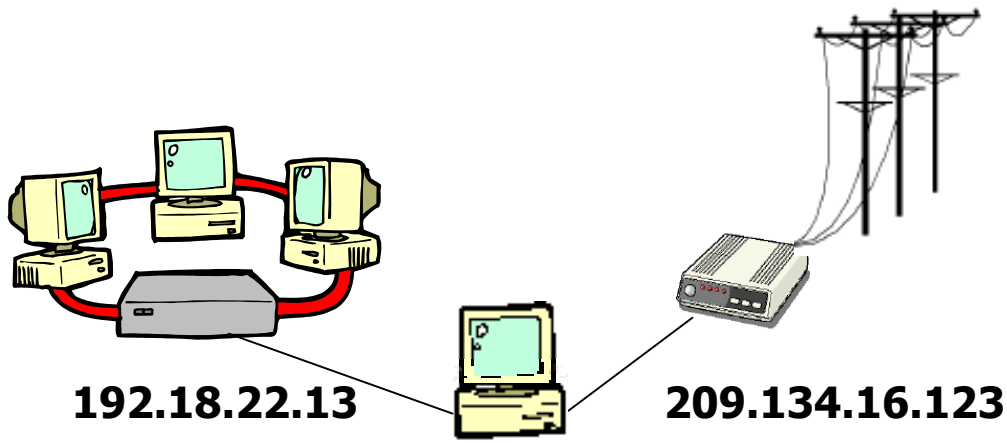


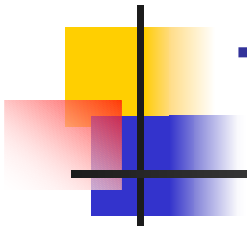
Internet Protocol (IP)

- 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

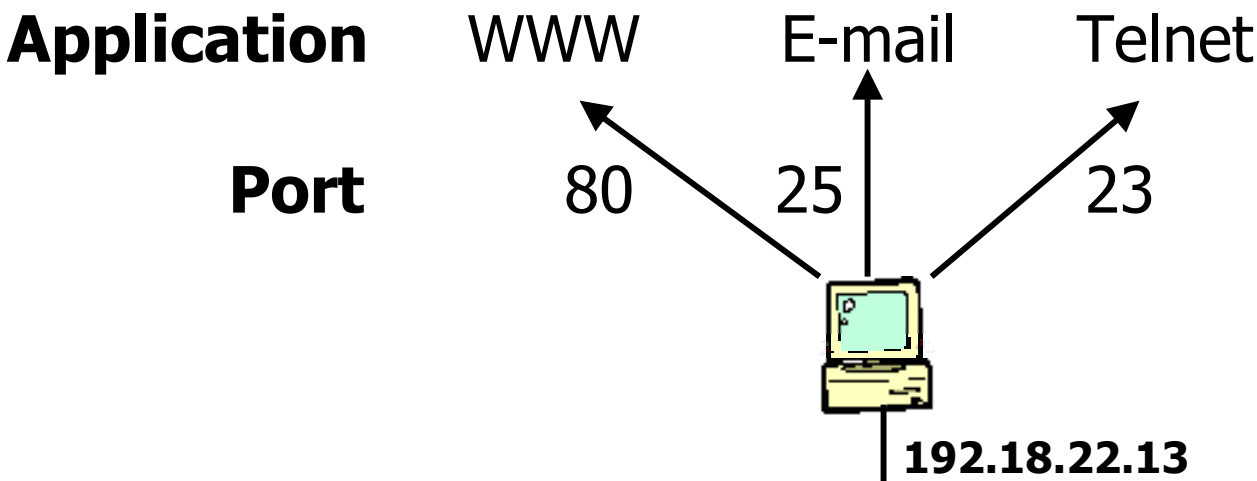
Best-effort not sufficient!

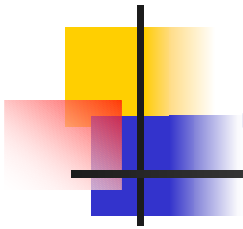
- Add services on top of IP
- User Datagram Protocol (UDP)
  - Data checksum
  - Best-effort
- Transmission Control Protocol (TCP)
  - Data checksum
  - Reliable byte-stream delivery
  - Flow and congestion control

Ports

Identifying the ultimate destination

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





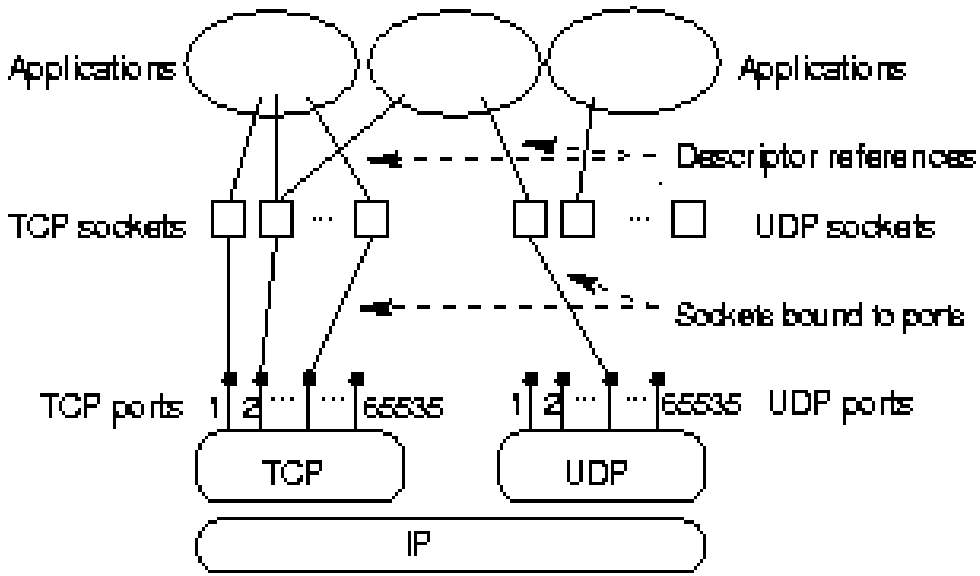
Socket

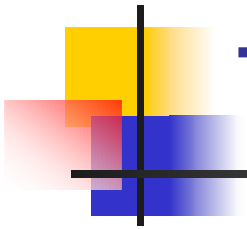
How does one speak TCP/IP?

- 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





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

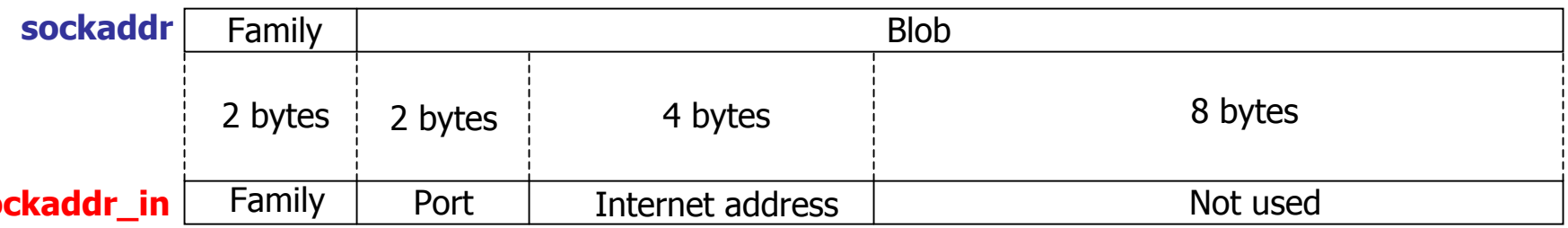
- **struct sockaddr**

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

IP Specific

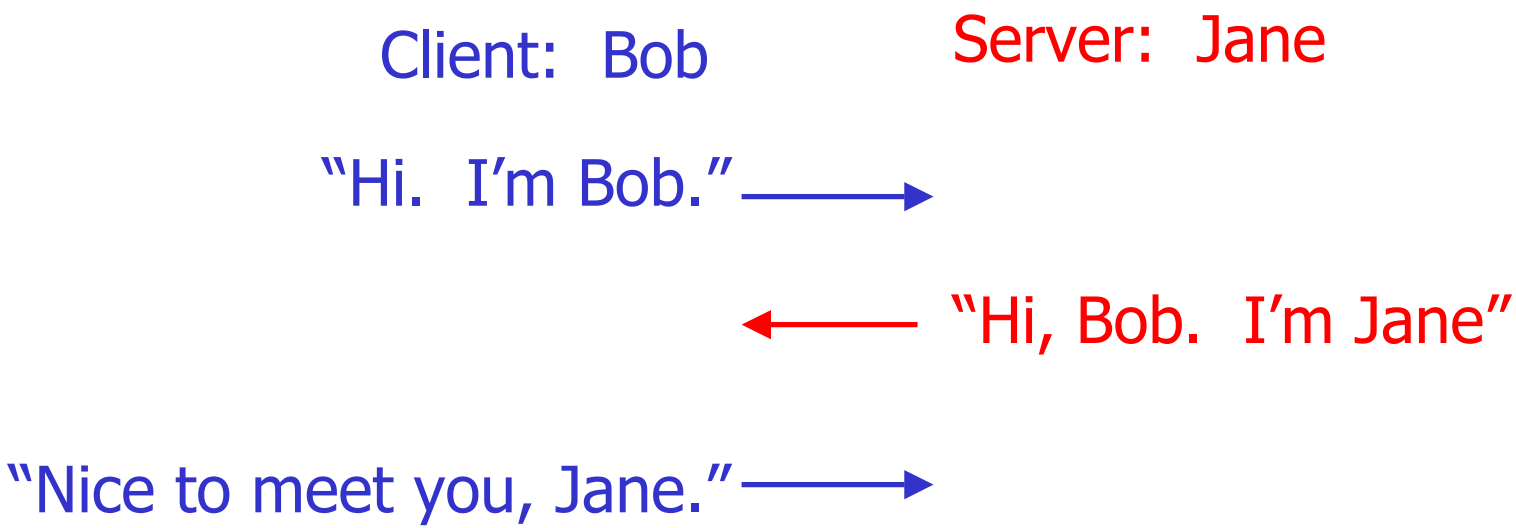
- **struct sockaddr_in**

```
{
        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) */
};
```

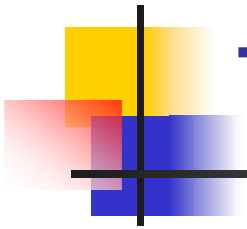


Clients and Servers

- Client: Initiates the connection



- 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:
    a. Accept new connection
    b. Communicate
    c. 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

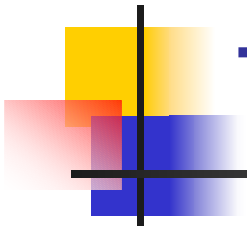
```
for (;;) /* Run forever */
{
    clntLen = sizeof(echoClntAddr);

    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

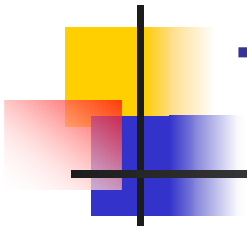
Later, a client decides to talk to the server…

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);

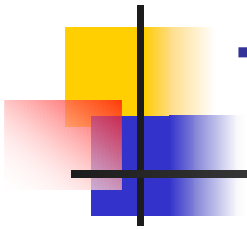
close(clntSocket)

Client

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

Server

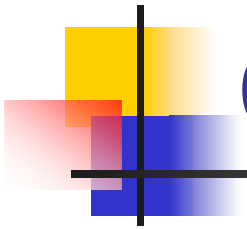
1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection



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()`

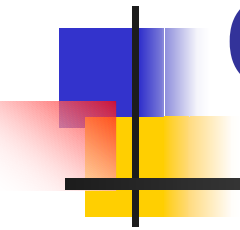
Client	Server
send("Hello Bob")	
	recv() -> "Hello "
	recv() -> "Bob"
	send("Hi ")
	send("Jane")
recv() -> "Hi Jane"	



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*)

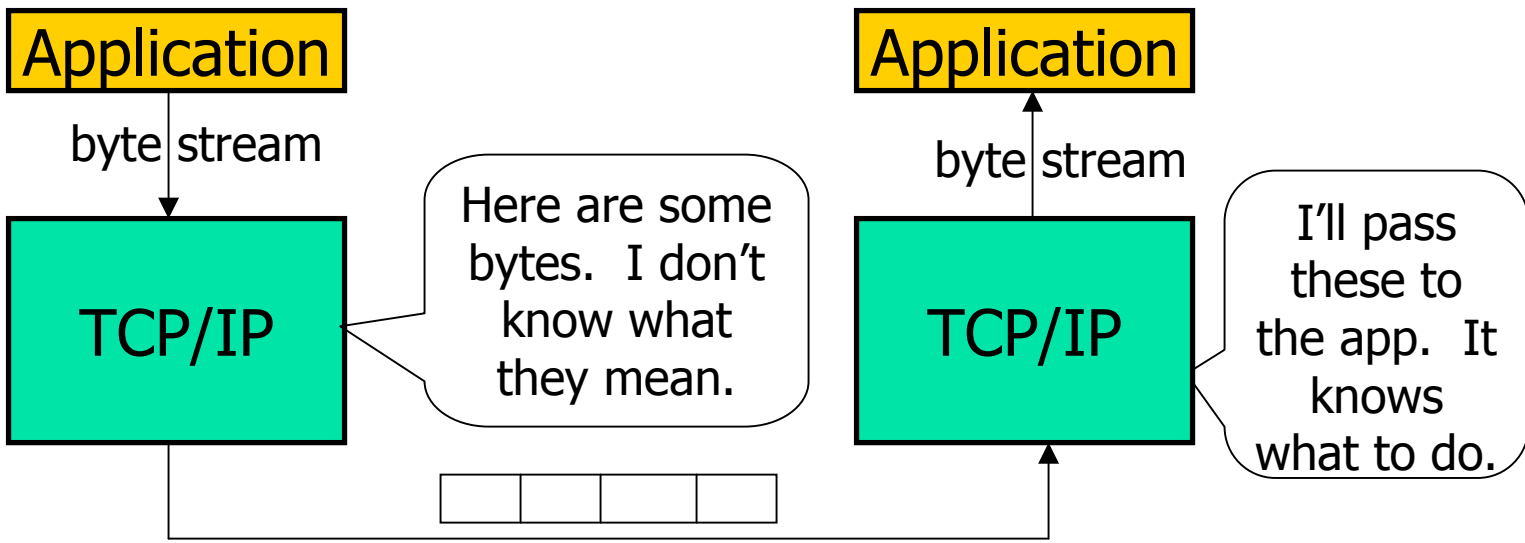


Constructing Messages

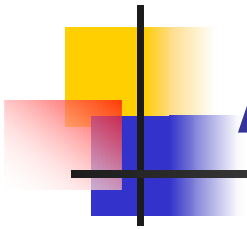
…beyond simple strings

TCP/IP Byte Transport

- TCP/IP protocols transports bytes



- Application protocol provides semantics

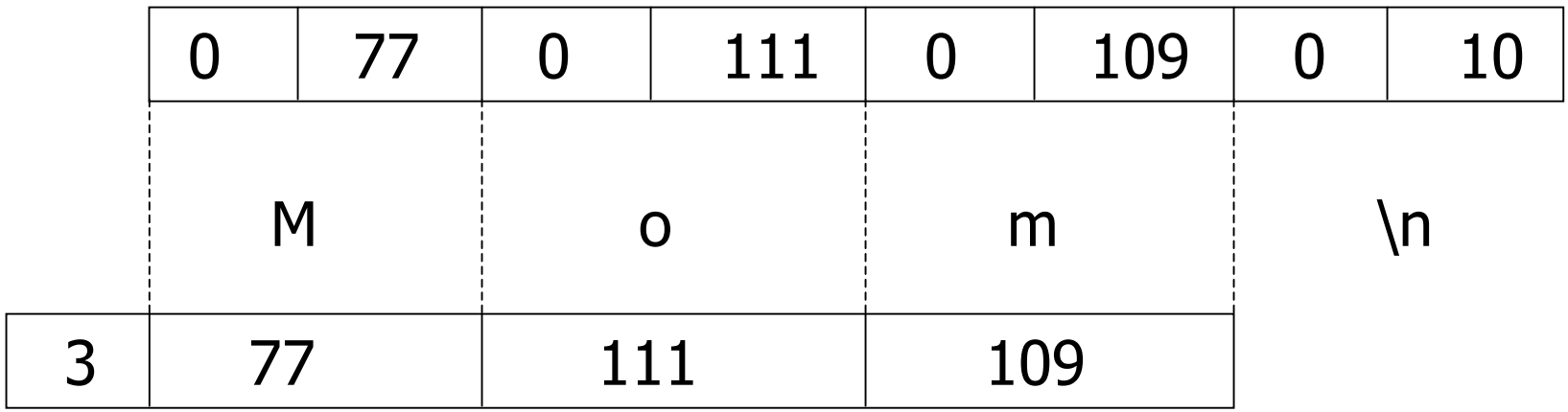


Application Protocol

- 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



Primitive Types

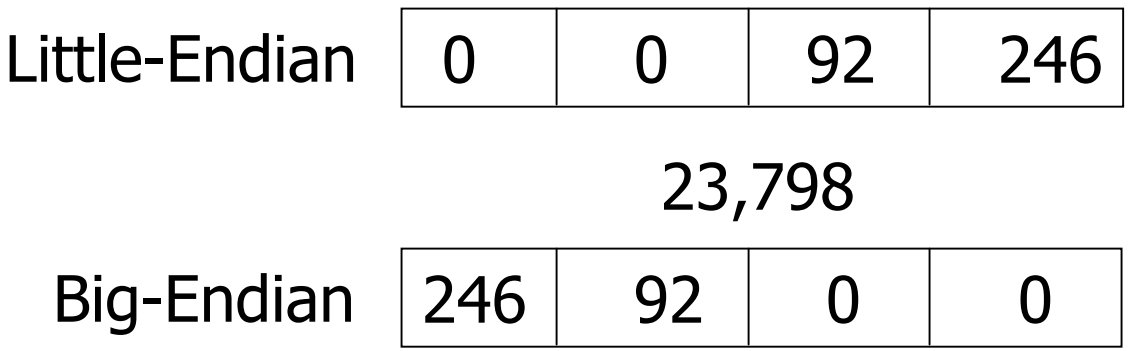
- Integer
  - Strings of character encoded decimal digits

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



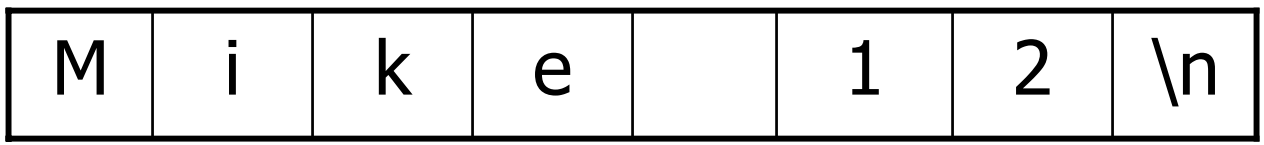
4-byte two's-complement integer

  - 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

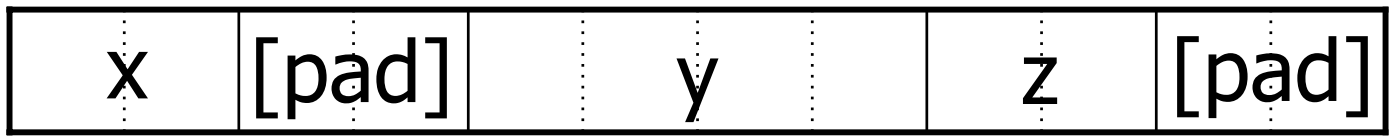
  - Variable-length fields



"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