

UDP, TCP, and UNIX Sockets

George Porter

CSE 124

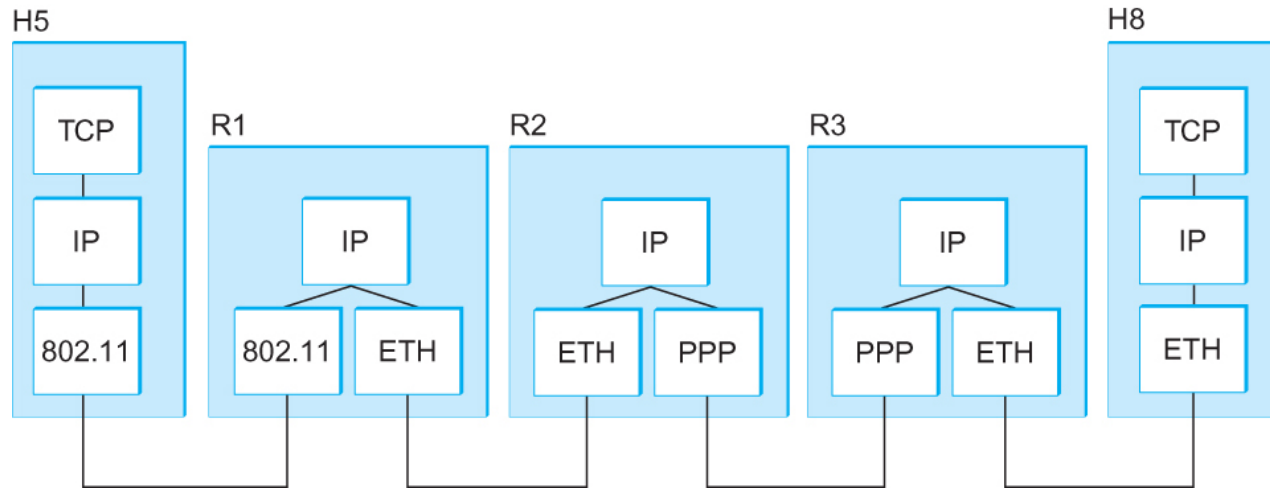
January 13, 2015

Annoucements

- First project available
 - HTTP Server: due Feb 3, 2015
- First homework available
 - due Jan 27, 2015
- For those of you who have not taken CSE 123:
 - Especially important to do the recommended reading in the Peterson & Davie book (see syllabus) and the “Practical TCP/IP” book

HTTP

Recall: layering



- Ethernet: (unreliable) packet delivery on a link
- IP: (unreliable) packet delivery across the Internet
- TCP: reliable delivery of a stream of bytes across the Internet
- HTTP: delivery of text, images, video, ...
 - Runs on top of TCP

Traditional Applications

- World Wide Web
 - The core idea of hypertext is that one document can link to another document, and the protocol (HTTP) and document language (HTML) were designed to meet that goal.
 - One helpful way to think of the Web is as a set of cooperating clients and servers, all of whom speak the same language: HTTP.
 - Most people are exposed to the Web through a graphical client program, or Web browser, like Safari, Chrome, Firefox or Internet Explorer.

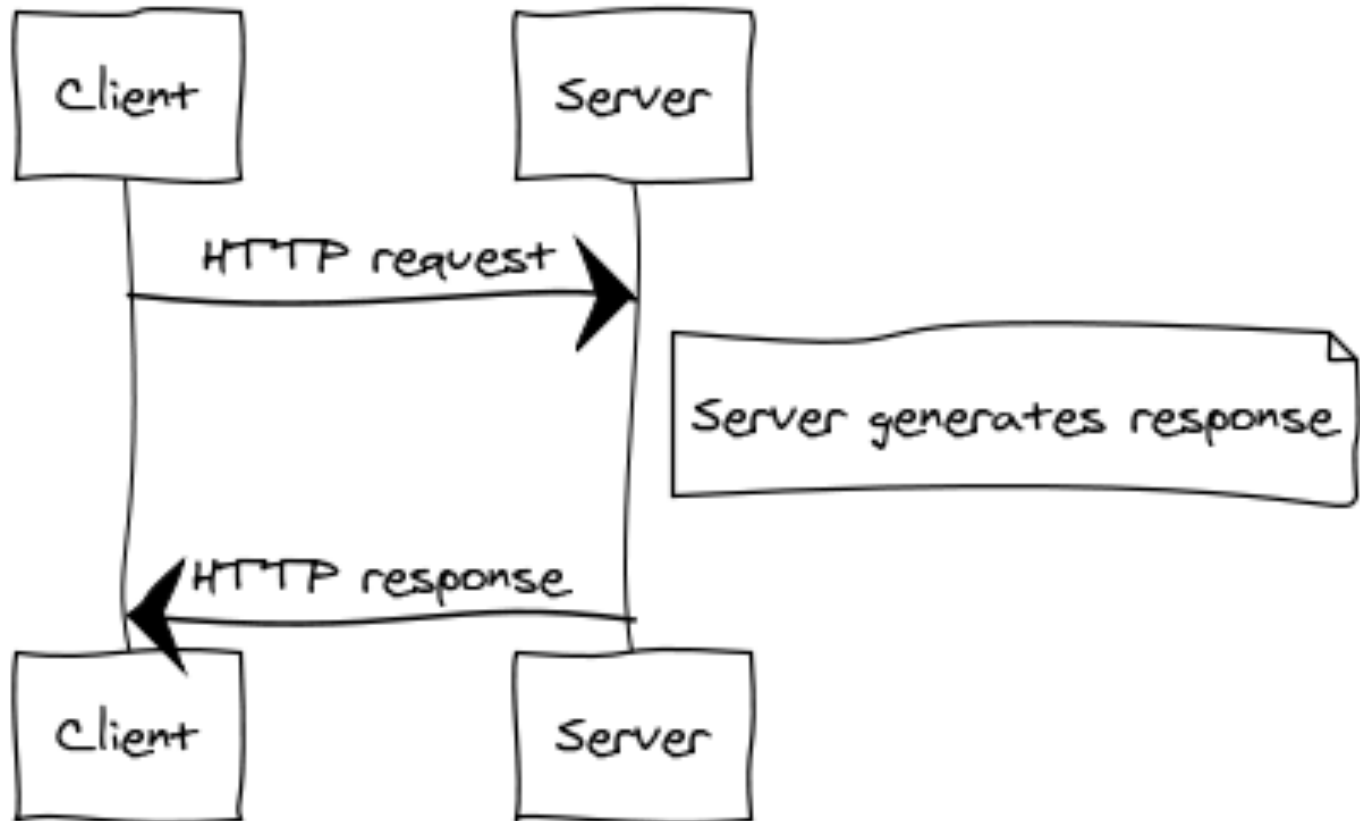
Traditional Applications

- World Wide Web

- When you ask your browser to view a page, your browser (the client) fetches the page from the server using HTTP running over TCP.
- HTTP is a text oriented protocol.
- At its core, HTTP is a request/response protocol, where every message has the general form

```
START_LINE <CRLF>
MESSAGE_HEADER <CRLF>
<CRLF>
MESSAGE_BODY <CRLF>
```
- where <CRLF> stands for carriage-return-line-feed.
- The first line (START LINE) indicates whether this is a request message or a response message.

HTTP 1.0 Example



HTTP requests

- Request Messages
 - The first line of an HTTP request message specifies three things: the operation (called *method*) to be performed, the Web page the operation should be performed on, and the version of HTTP being used.
 - We only focus on the 'GET' method in CSE 124
- Examples:
 - GET /index.html HTTP/1.0
 - GET /images/cating23.jpg HTTP/1.1
 - GET /contracts/contract3.txt HTTP/1.1

Request headers

- After the start line are zero or more *request headers*:
 - Text-based, key and value separated by a colon

- Example 1:

GET /index.html HTTP/1.0

User-Agent: Firefox 23.3.1

key  value

- Example 2:

GET /images/cat2.jpg HTTP/1.1

Host: www.cs.ucsd.edu

User-Agent: Chrome 12.1

HTTP Responses

- Response Messages
 - Like request messages, response messages begin with a single START LINE.
 - In this case, the line specifies the version of HTTP being used, a three-digit code indicating whether or not the request was successful, and a text string giving the reason for the response.
- Example:
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 291

```
borabora:~ gmporter$ telnet oec-vmweb09.ucsd.edu 80
Trying 132.239.8.67...
Connected to oec-vmweb09.ucsd.edu.
Escape character is '^]'.
```

Request

```
GET /index.html HTTP/1.0
```

```
HTTP/1.1 200 OK
Date: Mon, 12 Jan 2015 19:36:37 GMT
Server: Apache/2.2.22 (Ubuntu)
Last-Modified: Thu, 28 Feb 2013 17:35:36 GMT
ETag: "fc7b21-a-4d6cc51858aec"
Accept-Ranges: bytes
Content-Length: 10
Vary: Accept-Encoding
Connection: close
Content-Type: text/html
```

```
It works!
```

```
Connection closed by foreign host.
borabora:~ gmporter$
```

Response

HTTP Response Codes

Code	Type	Example Reasons
1xx	Informational	request received, continuing process
2xx	Success	action successfully received, understood, and accepted
3xx	Redirection	further action must be taken to complete the request
4xx	Client Error	request contains bad syntax or cannot be fulfilled
5xx	Server Error	server failed to fulfill an apparently valid request

Five types of HTTP result codes

200: OK

400: Client Error

403: Not allowed

404: Not found

Traditional Applications

- World Wide Web

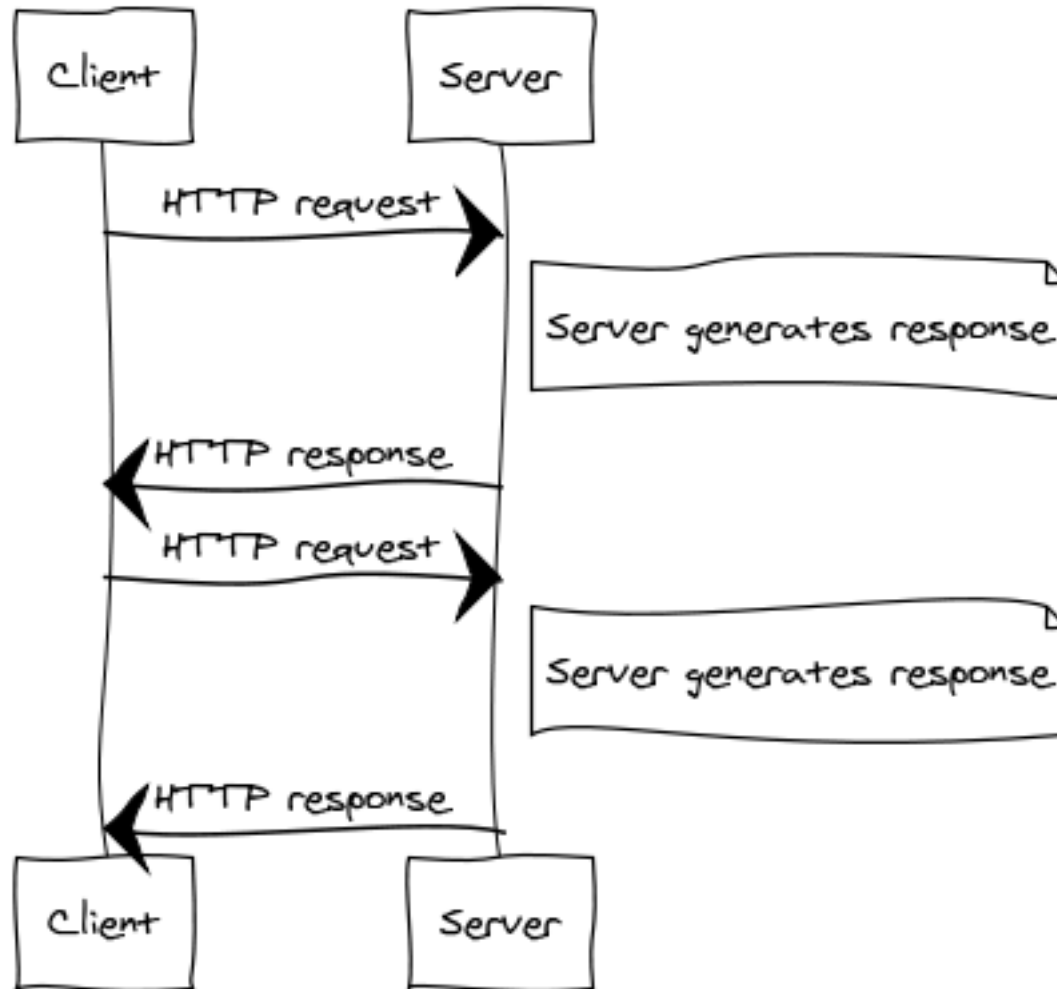
- TCP Connections

- The original version of HTTP (1.0) established a separate TCP connection for each data item retrieved from the server.
 - It's not too hard to see how this was a very inefficient mechanism: connection setup and teardown messages had to be exchanged between the client and server even if all the client wanted to do was verify that it had the most recent copy of a page.
 - Thus, retrieving a page that included some text and a dozen icons or other small graphics would result in 13 separate TCP connections being established and closed.

HTTP 1.1

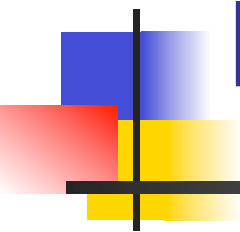
- TCP Connections
 - To overcome this situation, HTTP version 1.1 introduced *persistent connections*— the client and server can exchange multiple request/response messages over the same TCP connection.
 - Persistent connections have many advantages.
 - First, they obviously eliminate the connection setup overhead, thereby reducing the load on the server, the load on the network caused by the additional TCP packets, and the delay perceived by the user.
 - Second, because a client can send multiple request messages down a single TCP connection, TCP's congestion window mechanism is able to operate more efficiently.
 - » This is because it's not necessary to go through the slow start phase for each page.

HTTP 1.1 “Persistent Connections”



Sockets overview

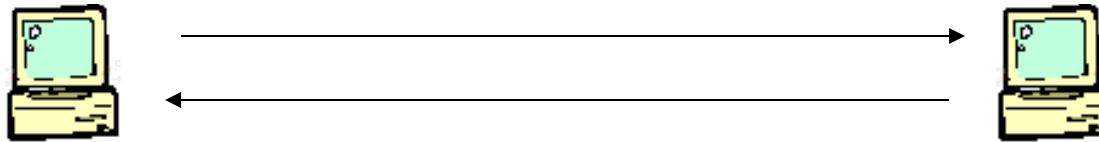
TCP/IP Sockets in C: Practical Guide for Programmers



Michael J. Donahoo
Kenneth L. Calvert

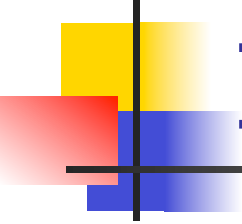
Computer Chat

- How do we make computers talk?



- How are they interconnected?

Internet Protocol (IP)

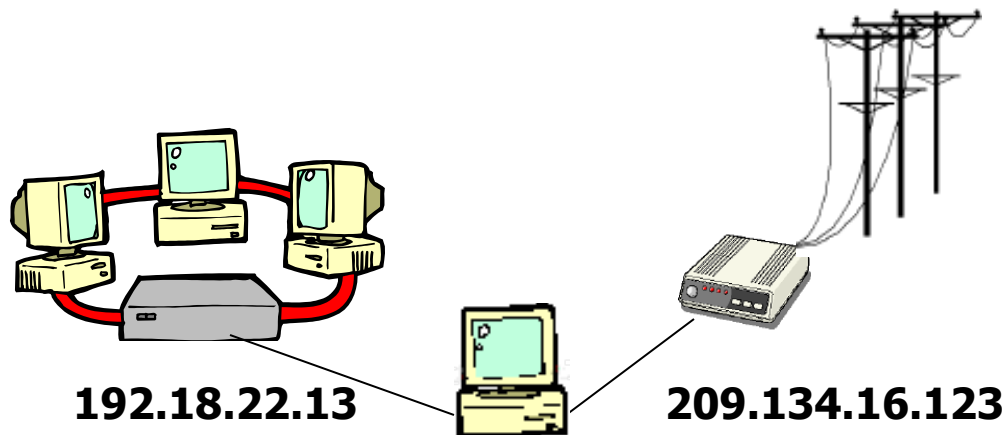


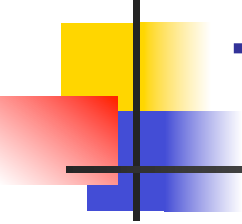
Internet Protocol (IP)

- 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)

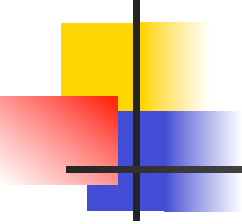




Transport Protocols

Best-effort not sufficient!

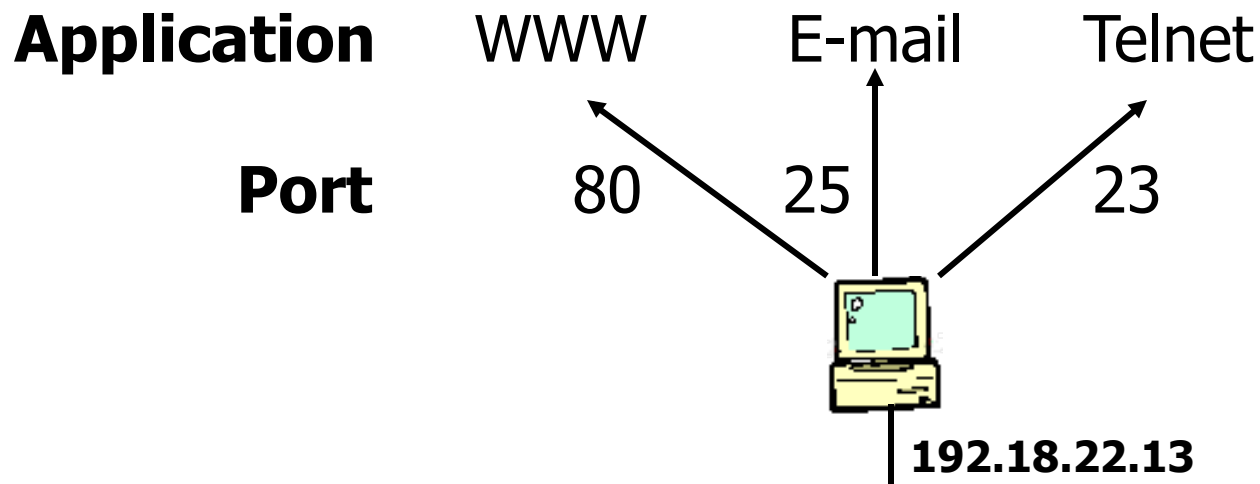
- Add services on top of IP
- User Datagram Protocol (UDP)
 - Data checksum
 - Best-effort
- Transmission Control Protocol (TCP)
 - Data checksum
 - Reliable byte-stream delivery
 - Flow and congestion control

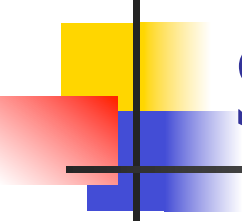


Ports

Identifying the ultimate destination

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





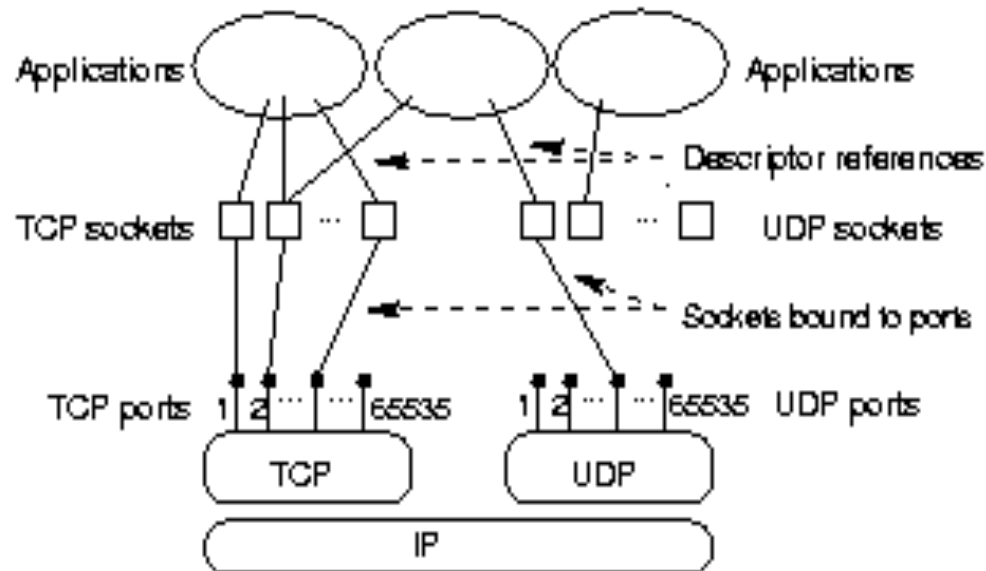
Socket

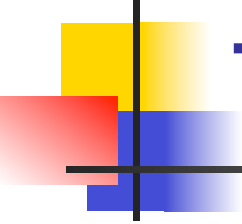
How does one speak TCP/IP?

- 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

Generic

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

IP Specific

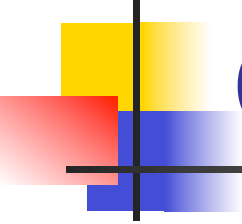
```
■ struct sockaddr_in
{
    unsigned short sin_family; /* Internet protocol (AF_INET) */
    unsigned short sin_port;   /* Port (16-bits) */
    struct in_addr sin_addr;   /* Internet address (32-bits) */
    char sin_zero[8];         /* Not used */
};

struct in_addr
{
    unsigned long s_addr;     /* Internet address (32-bits) */
};
```

sockaddr

Family	Blob		
2 bytes	2 bytes	4 bytes	8 bytes
Family	Port	Internet address	Not used

sockaddr_in



Clients and Servers

- Client: Initiates the connection

Client: Bob

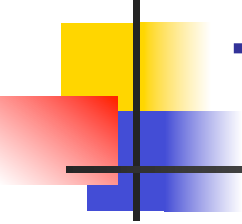
Server: Jane

“Hi. I’ m Bob.” →

← “Hi, Bob. I’ m Jane”

“Nice to meet you, Jane.” →

- Server: Passively waits to respond



TCP Client/Server Interaction

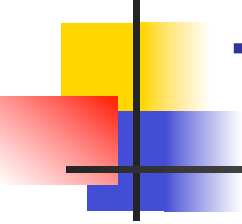
Server starts by getting ready to receive client connections...

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



TCP Client/Server Interaction

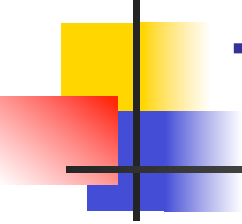
```
/* Create socket for incoming connections */  
if ((servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)  
    DieWithError("socket() failed");
```

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 */
```

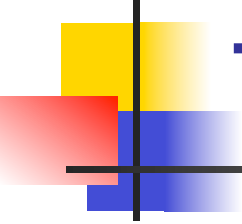
```
if (bind(servSock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)  
    DieWithError("bind() failed");
```

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

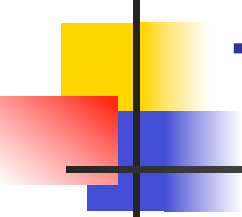
```
/* Mark the socket so it will listen for incoming connections */  
if (listen(servSock, MAXPENDING) < 0)  
    DieWithError("listen() failed");
```

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);
```

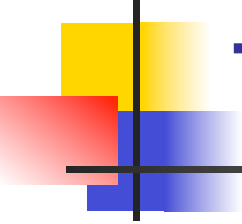
```
    if ((clntSock=accept(servSock,(struct sockaddr *)&echoClntAddr,&clntLen)) < 0)  
        DieWithError("accept() failed");
```

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

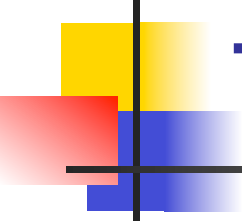
Later, a client decides to talk to the server...

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

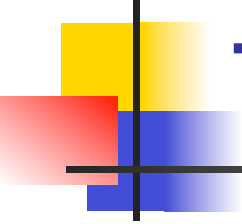
```
/* Create a reliable, stream socket using TCP */  
if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)  
    DieWithError("socket() failed");
```

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 */
```

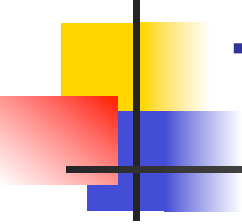
```
if (connect(sock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
    DieWithError("connect() failed");
```

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

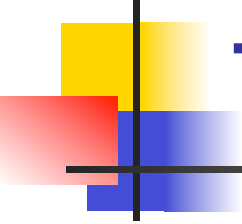
```
if ((clntSock=accept(servSock,(struct sockaddr *)&echoClntAddr,&clntLen)) < 0)
    DieWithError("accept() failed");
```

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

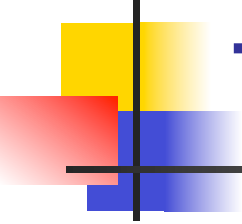
```
echoStringLen = strlen(echoString);           /* Determine input length */  
  
/* Send the string to the server */  
if (send(sock, echoString, echoStringLen, 0) != echoStringLen)  
    DieWithError("send() sent a different number of bytes than expected");
```

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

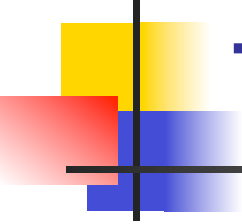
```
/* Receive message from client */  
if ((recvMsgSize = recv(clntSocket, echoBuffer, RCVBUFSIZE, 0)) < 0)  
    DieWithError("recv() failed");
```

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);`

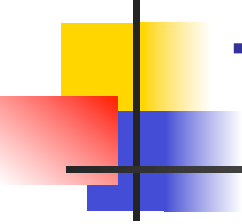
`close(clntSocket)`

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")`

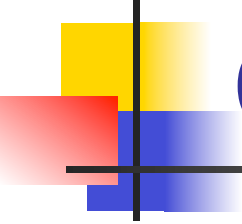
`recv()` -> "Hello "

`recv()` -> "Bob"

`send("Hi ")`

`send("Jane")`

`recv()` -> "Hi Jane"



Closing a Connection

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

Echo Client

`send(string)`

`while (not received entire string)`

`recv(buffer)`

`print(buffer)`

`close(socket)`

Echo Server

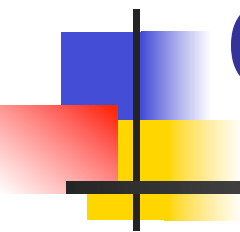
`recv(buffer)`

`while(client has not closed connection)`

`send(buffer)`

`recv(buffer)`

`close(client socket)`

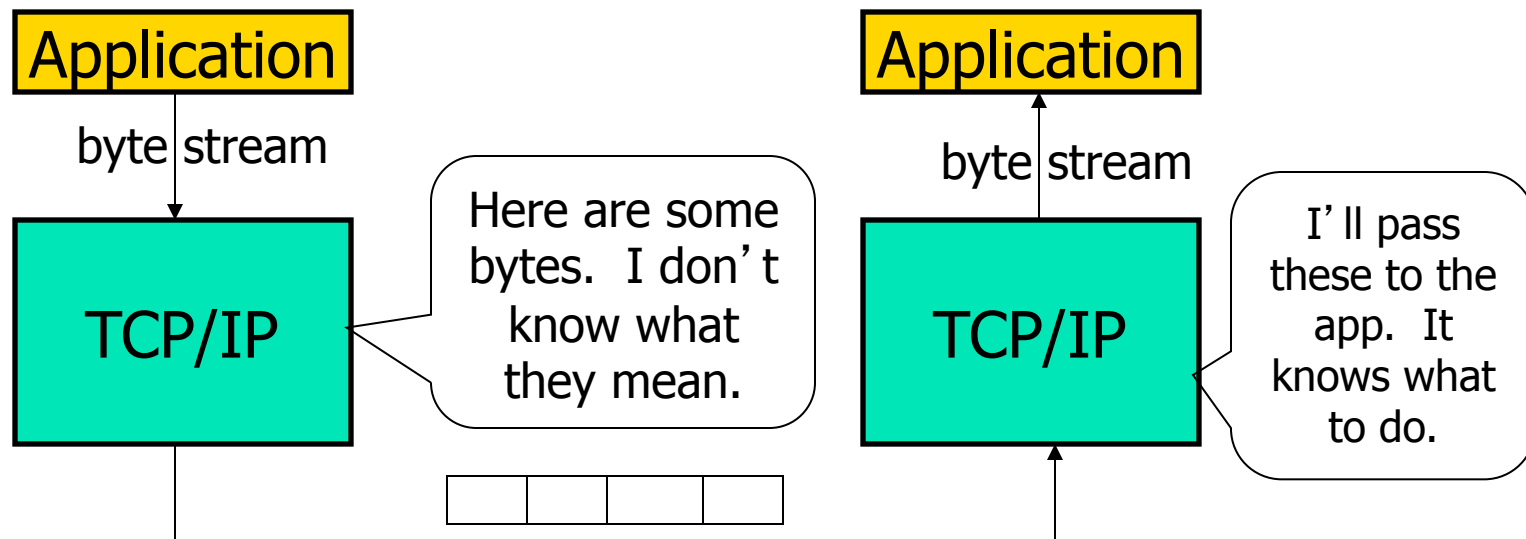


Constructing Messages

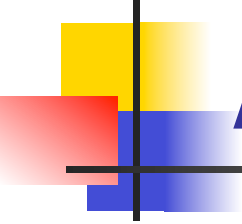
...beyond simple strings

TCP/IP Byte Transport

- TCP/IP protocols transports **bytes**

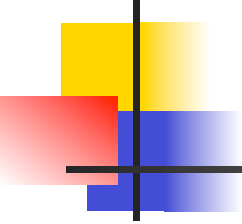


- Application protocol provides semantics



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

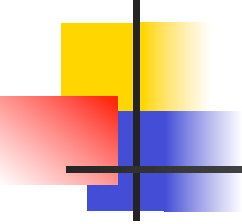


Primitive Types

- String

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

	0	77	0	111	0	109	0	10
	M		o		m		\n	
3	77		111		109			



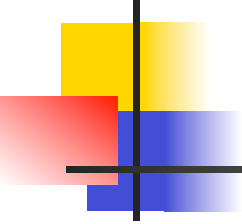
Primitive Types

- 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

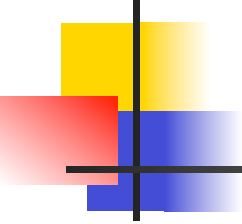
- Integer

- Native representation

Little-Endian	0	0	92	246	4-byte two's-complement integer
23,798					
Big-Endian	246	92	0	0	

- Network byte order (Big-Endian)

- Use for multi-byte, binary data exchange
 - htonl(), htons(), ntohl(), ntohs()



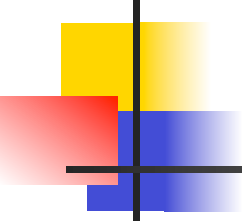
Message Composition

- Message composed of fields
 - Fixed-length fields

integer	short	short
---------	-------	-------

- Variable-length fields

M	i	k	e		1	2	\n
---	---	---	---	--	---	---	----

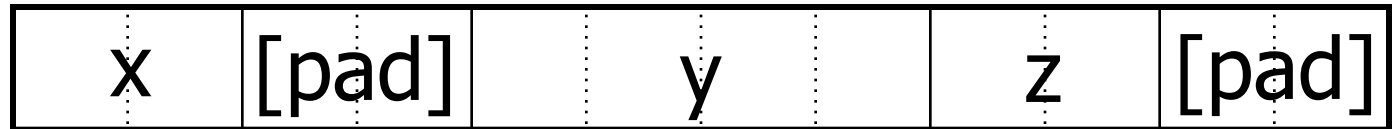


“Beware the bytes of padding”

-- 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