Sockets and a high level intro to the Internet

- 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)
- The slides assume the reader is aware of the layered network model, and in particular, the 5 layers of TCP/IP
Internet Protocol (IP)

- Datagram (packet) protocol that represents the network layer
- Best-effort service - job is to deliver a datagram to the destination network that is indicated in the IP datagram header destination network address field.
  - datagrams or packets sent by one host to another host might suffer impairments such as loss, Reordering, Duplication or Delay
- Within a network, routers ONLY look at the network prefix in the destination address and forward based on that (using the contents of the routing information that it has in its local route table).
  - Conceptually, this is similar to a package arriving at a central post office which might just use the zip code (and not the full address) to determine the next central office to ship the package.
Host A has a message to send to Host B
The application is to use UDP....

Steps:
- Create the message
- Create a socket and issue a send
- The send call passes the data to UDP which produces a UDP datagram
- UDP passes this to IP which produces an IP datagram
- IP passes this to the link layer which produces a frame
- The Link layer passes the frame to the physical layer whose job is to transmit each data bit to a receiver
The life of a packet….

Host A
- Application
- Transport: UDP, TCP, ICMP
- IP – network layer
- Link
- Physical layer

Router 1
- IP
- Link
- Phy

Host B
- Application
- Transport: UDP, TCP, ICMP
- IP – network layer
- Link
- Physical layer

---

- Host A Physical layer transmits all bytes In the frame…… bit by bit over the link to the Router 1
- The time it takes to transmit the frame is \( T = \frac{N}{C} \) Time in seconds (Tx Time) is the number of bits divided by the channel speed (bits per second).
- In addition, once a bit is transmit, since it is now in a form of energy that is propagating over the physical medium – the laws of physics come into play to determine the actual amount of time it takes a bit to propagate over the wire…always less than the speed of light. But…..over a Gbps link, the prop delay is the HUGE portion of the time it takes over a link that travels hundreds or thousands of miles.
- Router 1 receives the frame and forwards to the IP layer.
- The IP layer looks at the dest address in the IP header to make a decision as to which link to forward
  - Imagine a router that is connected to 6 different networks, each by a separate Ethernet link. The IP ’s forwarding decision matches the network prefix in the dst IP address of the packet to an entry in the forwarding table.
  - The table is a list of all network prefixes known, with the correct Ethernet interface with each entry.
The life of a packet…

- The network can range from
  - Host A directly connected to Host B via a Ethernet switch or a single directly connected cable
  - Or the network might be over the Internet and consist of dozens of ‘hops’ -
    - Each Hop corresponds to a link to a router.
    - If there are 12 hops, the time it takes to transmit the frame is the sum of each Tx time over each of the 12 links.
    - Further, the total propagation delay is the sum of all prop delays over each link.
- Eventually, the network delivers the packet to the destination network that Host B directly connects with.
The life of a packet....

Application  UDPEcho (port 5000)

UDP- looks at the dst port and finds the application instance that has a socket bound to that port.....
IP layer- looks at the protocol id to determine to send to UDP, TCP, ICMP
Link
Physical

- Host B Phy and Link layers receive the frame, determine that Host B is the intended dst Host and passes the IP datagram to the IP layer.
- KEY: The network prefix of the routers IP and the Host B IP MUST be the same!!!!!
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
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.

<table>
<thead>
<tr>
<th>BSD Version</th>
<th>Release Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2 BSD</td>
<td>1982</td>
</tr>
<tr>
<td>4.3 BSD</td>
<td>1986</td>
</tr>
<tr>
<td>4.3 BSD Tahoe</td>
<td>1988</td>
</tr>
<tr>
<td>4.3 BSD Reno</td>
<td>1990</td>
</tr>
<tr>
<td>4.4 BSD</td>
<td>1993</td>
</tr>
<tr>
<td>4.4 BSD Lite</td>
<td>1994</td>
</tr>
</tbody>
</table>

(BSD/OS, FreeBSD NetBSD, OpenBSD)
TCP/IP Sockets

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

<table>
<thead>
<tr>
<th>Family</th>
<th>Type</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>PF_INET</td>
<td>SOCK_STREAM</td>
</tr>
<tr>
<td>UDP</td>
<td></td>
<td>SOCK_DGRAM</td>
</tr>
</tbody>
</table>

- Socket reference
  - File (socket) descriptor in UNIX
  - Socket handle in WinSock
- struct sockaddr
  {
    unsigned short sa_family; /* Address family (e.g., AF_INET) */
    char sa_data[14]; /* Protocol-specific address information */
  }

- 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) */
  }

<table>
<thead>
<tr>
<th>sockaddr</th>
<th>Family</th>
<th>Blob</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 bytes</td>
<td>2 bytes</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sockaddr_in</th>
<th>Family</th>
<th>Port</th>
<th>Internet address</th>
<th>Not used</th>
</tr>
</thead>
</table>
TCP Client/Server Interaction

Server starts by getting ready to receive client connections…

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Create a TCP socket</td>
<td>1. Create a TCP socket</td>
</tr>
<tr>
<td>2. Establish connection</td>
<td>2. Assign a port to socket</td>
</tr>
<tr>
<td>3. Communicate</td>
<td>3. Set socket to listen</td>
</tr>
<tr>
<td>4. Close the connection</td>
<td>4. Repeatedly:</td>
</tr>
<tr>
<td></td>
<td>a. Accept new connection</td>
</tr>
<tr>
<td></td>
<td>b. Communicate</td>
</tr>
<tr>
<td></td>
<td>c. Close the connection</td>
</tr>
</tbody>
</table>
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

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

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

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

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

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

```c
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");
```
TCP Client/Server Interaction

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

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

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

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

Client
```
send("Hello Bob")
```

Server
```
recv() -> "Hello 
recv() -> "Bob"
send("Hi ")
send("Jane")
recv() -> "Hi Jane"
```
Closing a Connection

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

Echo Client

```c
send(string)
while (not received entire string)
    recv(buffer)
    print(buffer)
close(socket)
```

Echo Server

```c
recv(buffer)
while (client has not closed connection)
    send(buffer)
    recv(buffer)
```

close(client socket)
## UDP Client / Server Interaction

<table>
<thead>
<tr>
<th>UDP Client</th>
<th>UDP Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>socke()</td>
<td>socke()</td>
</tr>
<tr>
<td>bind()</td>
<td>bind()</td>
</tr>
<tr>
<td>recvfrom()</td>
<td>recvfrom()</td>
</tr>
<tr>
<td>socke()</td>
<td>socke()</td>
</tr>
<tr>
<td>sendto()</td>
<td>sendto()</td>
</tr>
<tr>
<td>recvfrom()</td>
<td>close()</td>
</tr>
<tr>
<td>close()</td>
<td>close()</td>
</tr>
</tbody>
</table>
UDP Socket Calls

```c
#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

```c
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));
```
#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>]
        
        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");

    return 0;                /* Normal completion */
}
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

```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");
```
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");

    /* 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");
}

Example Iterative Server:
EchoUDPServer.c
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 server
  • 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”.
Topics presented in other slides

- Socket options
- Multiplexing sockets
- Concurrent server design
- Implementing well designed full duplex applications
- Data representation –
  - Primitive types vs structs
  - Streams versus messages
  - Data serialization – translating internal representation of data into efficient, serialized buffers that can be transmitted over a network
    - Basic methods
    - More advanced….standards