1. DNS

- User clicks on http://www.nytimes.com/
- URL contains Internet name of machine (www.nytimes.com), but not Internet address
- Internet needs Internet address to send information to a machine
- Browser software uses Domain Name System (DNS) protocol to send query for Internet address
- DNS system responds with Internet address
TCP/IP Protocol Suite

- HTTP
- SMTP
- DNS
- RTP

TCP/IP Protocol Suite

- Reliable stream service
- User datagram service
- Distributed applications
- IP

UDP (User Datagram Protocol)

- UDP is a transport layer protocol
- Provides best-effort datagram service between two processes in two computers across the Internet
- Port numbers distinguish various processes in the same machine
- UDP is connectionless
- Datagram is sent immediately
- Quick, simple, but not reliable
TCP (Transmission Control Protocol)

- TCP is a transport layer protocol
- Provides **reliable byte stream service** between two processes in two computers across the Internet
- Sequence numbers keep track of the bytes that have been transmitted and received
- Error detection and retransmission used to recover from transmission errors and losses
- TCP is **connection-oriented**: the sender and receiver must first establish an association and set initial sequence numbers before data is transferred
- Connection ID is specified uniquely by

  \[(send \text{ port } #, \text{ send IP address}, receive \text{ port } #, \text{ receiver IP address})\]

Socket API

- **API (Application Programming Interface)**
  - Provides a standard set of functions that can be called by applications
- **Berkeley UNIX Sockets API**
  - Abstraction for applications to send & receive data
  - Applications create sockets that “plug into” network
  - Applications write/read to/from sockets
  - Implemented in the kernel
  - Facilitates development of network applications
  - Hides details of underlying protocols & mechanisms
- Also in Windows, Linux, and other OS’s
Communications through Socket Interface

Stream Mode of Service

Connection-oriented

- First, setup connection between two peer application processes
- Then, reliable bidirectional in-sequence transfer of byte stream (boundaries not preserved in transfer)
- Multiple write/read between peer processes
- Finally, connection release
- Uses TCP

Connectionless

- Immediate transfer of one block of information (boundaries preserved)
- No setup overhead & delay
- Destination address with each block
- Send/receive to/from multiple peer processes
- Best-effort service only
  - Possible out-of-order
  - Possible loss
- Uses UDP
Client & Server Differences

° Server
  • Specifies well-known port # when creating socket
  • May have multiple IP addresses (net interfaces)
  • Waits passively for client requests

° Client
  • Assigned ephemeral port #
  • Initiates communications with server
  • Needs to know server’s IP address & port #
    - DNS for URL & server well-known port #
  • Server learns client’s address & port #

Socket-based Network Programming

° Socket: provides an endpoint for communication between processes
  • Must be bound to a port # and an IP address at server side
  • Message destinations: (Internet address, local port)
  • Port: a message destination within a computer, an 16-bit integer
  • One port has one receiver (process), but can have many senders
  • One process can use multiple ports
UDP Datagram Communication

- **UDP**: The delivery of the message is not guaranteed
  - Message size: up to $2^{16}$ B (usual restriction 8 KB)
  - Non-blocking send and blocking receive
  - Timeout: to avoid infinite wait of blocking receive (*setSoTimeout*)
  - Receive from any
  - Ordering: messages can be delivered out of sender order
  - Omission failures: send-omission, receive-omission, channel-omission

Internet address = 138.37.94.248, Internet address = 138.37.88.249

Socket Calls for Connection-less Mode (UDP)

**Server**
- `socket()` creates socket of type UDP (datagram)
- `socket()` call returns: *descriptor*, or -1 if unsuccessful
- `bind` assigns local address & port # to socket with specified descriptor;

**Client**
- `socket()`
- `sendto()`
- `recvfrom()`
- `close()`
Socket Calls for Connection-less Mode (UDP)

- `recvfrom` copies bytes received in specified socket into a specified location
- `recvfrom` blocks until data arrives

### Server
- `socket()`
- `bind()`
- `recvfrom()` (blocks until server receives data from client)
- `sendto()`
- `close()`

### Client
- `socket()`
- `sendto()`
- `recvfrom()`
- `close()`

---

Socket Calls for Connection-less Mode (UDP)

- Client started
- `socket` creates socket of type UDP (datagram)
- `socket` call returns: `descriptor`; or -1 if unsuccessful

### Server
- `socket()`
- `bind()`
- `recvfrom()` (blocks until server receives data from client)
- `sendto()`
- `close()`

### Client
- `socket()`
- `sendto()`
- `recvfrom()`
- `close()`
Socket Calls for Connection-less Mode (UDP)

Client started
- **sendto** transfer bytes in buffer to specified socket
- **recvfrom** specifies: socket descriptor; pointer to a buffer; amount of data; flags to control transmission behavior; destination address & port #; length of destination address structure
- **sendto** returns: # bytes sent; or -1 if unsuccessful

Client
- **socket()**
- **sendto()**
- **recvfrom()**
- **close()**

Server
- **socket()**
- **bind()**
- **recvfrom()**
- **sendto()**
- **close()**

Blocks until server receives data from client

Note: receivefrom returns data from at most one send, i.e. from one datagram

Client started
- **sendto** transfer bytes in buffer to specified socket
- **recvfrom** specifies: socket descriptor; pointer to a buffer; amount of data; flags to control transmission behavior; destination address & port #; length of destination address structure
- **sendto** returns: # bytes sent; or -1 if unsuccessful

Client
- **socket()**
- **sendto()**
- **recvfrom()**
- **close()**

Server
- **socket()**
- **bind()**
- **recvfrom()**
- **sendto()**
- **close()**
Socket Calls for Connection-less Mode (UDP)

**Socket Close**
- Client or server call `close` when socket is no longer needed
- `close` specifies the socket descriptor
- `close` call returns: 0 (success); or -1 (failure)

```
Example: UDP Echo Server
/* Echo server using UDP */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define SERVER_UDP_PORT 5000
#define MAXLEN 4096
int main(int argc, char **argv)
{
    int sd, client_len, port, n;
    char buf[MAXLEN];
    struct sockaddr_in server, client;
    switch(argc) {
        case 1:
            port = SERVER_UDP_PORT;
            break;
        case 2:
            port = atoi(argv[1]);
            break;
        default:
            fprintf(stderr, "Usage: %s [port]
            exit(1);
    }
    /* Create a socket */
    if ((sd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        fprintf(stderr, "Can't create a socket
        exit(1);
    }
    /* Bind an address to the socket */
    bzero((char *)&server, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_port = htons(port);
    server.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(sd, (struct sockaddr *)&server,
            sizeof(server)) == -1) {
        fprintf(stderr, "Can't bind name to
        exit(1);
    }
    while (1) {
        client_len = sizeof(client);
        if ((n = recvfrom(sd, buf, MAXLEN, 0,
                           (struct sockaddr *)&client, &client_len))
            < 0) {
            fprintf(stderr, "Can't receive
            exit(1);
        }
        if (sendto(sd, buf, n, 0,
                   (struct sockaddr *)&client, client_len)
            != n) {
            fprintf(stderr, "Can't send
            exit(1);
        }
        close(sd);
        return(0);
    }
```

---

**Server**
- `socket()`
- `bind()`
- `recvfrom()`

**Client**
- `socket()`
- `sendto()`
- `recvfrom()`
- `close()`

Data

Server
- `socket()`
- `bind()`
- `recvfrom()`

Client
- `socket()`
- `sendto()`
- `recvfrom()`
- `close()`
Example: UDP Echo Client

```c
#include <stdio.h>
#include <string.h>
#include <sys/time.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SERVER_UDP_PORT         5000
#define MAXLEN                  4096
#define DEFLEN                  64

long delay(struct timeval t1, struct timeval t2)
{
    long d;
    d = (t2.tv_sec - t1.tv_sec) * 1000;
    d += ((t2.tv_usec - t1.tv_usec + 500) / 1000);
    return(d);
}

int main(int argc, char **argv)
{
    int     data_size = DEFLEN, port = SERVER_UDP_PORT;
    int     i, j, sd, server_len;
    char    *pname, *host, rbuf[MAXLEN], sbuf[MAXLEN];
    struct  hostent         *hp;
    struct  sockaddr_in     server;
    struct  timeval         start, end;
    unsigned long address;

    pname = argv[0];
    argc--;
    argv++;
    if (argc > 0 && (strcmp(*argv, "-s") == 0)) {
        if (--argc > 0 && (data_size = atoi(*++argv))) {
            argc--;
            argv++;
        } else {
            fprintf(stderr, "Usage: %s [-s data_size] host [port]", pname);
            exit(1);
        }
    } else {
        host = *argv;
        if (--argc > 0)
            port = atoi(*++argv);
    }

    if ((sd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        fprintf(stderr, "Can't create a socket n");
        exit(1);
    }

    bzero((char *)&server, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_port = htons(port);
    if ((hp = gethostbyname(host)) == NULL) {
        fprintf(stderr, "Can't get server's IP address n");
        exit(1);
    }
    bcopy(hp->h_addr, (char *) &server.sin_addr, hp->h_length);

    if (data_size > MAXLEN) {
        fprintf(stderr, "Data is too big n");
        exit(1);
    }

    for (i = 0; i < data_size; i++) {
        j = (i < 26) ? i : i % 26;
        sbuf[i] = 'a' + j;
    }

    gettimeofday(&start, NULL); /* start delay measurement */
    server_len = sizeof(server);
    if (sendto(sd, sbuf, data_size, 0, (struct sockaddr *)&server, server_len) == -1) {
        fprintf(stderr, "sendto error n");
        exit(1);
    }

    if (recvfrom(sd, rbuf, MAXLEN, 0, (struct sockaddr *)&server, &server_len) < 0) {
        fprintf(stderr, "recvfrom error n");
        exit(1);
    }

    gettimeofday(&end, NULL); /* end delay measurement */
    if (strncmp(sbuf, rbuf, data_size) != 0)
        printf("Data is corrupted n");

    close(sd);
    return(0);
}
```

TCP Stream Communication

- TCP: The delivery of the message is “guaranteed”
  - Message size: unlimited, handled by underlying TCP protocol
  - Loss: an acknowledgement scheme; timeout and retransmission
  - Flow control: to match the speeds of the processes that read and write to a stream; if writing too fast, blocked
  - Duplication and ordering: message identifiers associated with each packet; reordering provided
  - Message destinations: explicit after connection is built (connect, accept)
TCP Stream Communication Issues

° Matching of data items: two ends agree on the contents (order) of the data; e.g., int + double
° Blocking: reading blocks until data becomes available; TCP flow control protocol will block a writing process if reading is too slow
° Threads: when accepting a connection, it generally creates a new thread with a socket to communicate to the new client for higher concurrency; no thread? select() in Unix

Socket Calls for Connection-Oriented Mode (TCP)

Server does Passive Open
- socket creates socket to listen for connection requests
- Server specifies type: TCP (stream)
- socket call returns: non-negative integer descriptor; or -1 if unsuccessful

Client
- socket
- connect()
- read()
- write()
- close()
Socket Calls for Connection-Oriented Mode (TCP)

Server does Passive Open
- **bind** assigns local address & port # to socket with specified descriptor
- Can wildcard IP address for multiple net interfaces
- **bind** call returns: 0 (success); or -1 (failure)
- Failure if port # already in use or if reuse option not set

Server
- `socket()`
- `bind()`
- `listen()`
- `accept()`

Blocks
- `read()`
- `write()`
- `close()`

Client
- `socket()`
- `connect()`
- `write()`
- `read()`
- `close()`

Server does Passive Open
- **listen** indicates to TCP readiness to receive connection requests for socket with given descriptor
- Parameter specifies max number of requests that may be queued while waiting for server to accept them
- **listen** call returns: 0 (success); or -1 (failure)
Socket Calls for Connection-Oriented Mode (TCP)

Server does Passive Open
- Server calls `accept` to accept incoming requests
- `accept` blocks if queue is empty

Server
- `socket()`
- `bind()`
- `listen()`
- `accept()`
- `read()`
- `write()`
- `close()`

Client
- `socket()`
- `connect()`
- `write()`
- `read()`
- `close()`

Client does Active Open
- `socket` creates socket to connect to server
- Client specifies type: TCP (stream)
- `socket` call returns: non-negative integer `descriptor`; or -1 if unsuccessful

Server
- `socket()`
- `bind()`
- `listen()`
- `accept()`
- `read()`
- `write()`
- `close()`

Client
- `socket()`
- `connect()`
- `write()`
- `read()`
- `close()`
Client does Active Open
- **connect** establishes a connection on the local socket with the specified descriptor to the specified remote address and port #
- **connect** returns 0 if successful; -1 if unsuccessful

**Note:** **connect** initiates TCP three-way handshake

- **accept** wakes with incoming connection request
- **accept** fills client address & port # into address structure
- **accept** call returns: descriptor of new connection socket (success); or -1 (failure)
- Server uses new socket for data transfer from/to client
- Original socket continues to listen for new requests
Socket Calls for Connection-Oriented Mode (TCP)

Data Transfer
- Client or server call `read` to receive data from a connected socket.
- `read` specifies: socket descriptor; pointer to a buffer; amount of data (stream-oriented -> repeated reads).
- `read` call returns: # bytes read (success); or -1 (failure); blocks if no data arrives.

Note: `write` and `read` can be called multiple times to transfer byte streams in both directions.

Socket Calls for Connection-Oriented Mode (TCP)

Data Transfer
- Client or server call `write` to transmit data into a connected socket.
- `write` specifies: socket descriptor; pointer to a buffer; amount of data; flags to control transmission behavior.
- `write` call returns: # bytes transferred (success); or -1 (failure); blocks until all data transferred.
Socket Calls for Connection-Oriented Mode (TCP)

### Connection Termination
- Client or server call `close` when socket is no longer needed
- `close` specifies the socket descriptor
- `close` call returns: 0 (success); or -1 (failure)

---

TCP Server: Socket Establishment

```c
int establish (unsigned short portnum)
{
    char   myname[MAXHOSTNAME+1];
    int    s;
    struct sockaddr_in sa;
    struct hostent *hp;

    memset (&sa, 0, sizeof(struct sockaddr_in));  /* clear our address */
    gethostbyname (myname, MAXHOSTNAME);        /* who are we? */
    hp= gethostbyname (myname);                 /* get our address info */
    if (hp == NULL)                             /* we don't exist !! */
        return(-1);
    sa.sin_family= hp->h_addrtype;              /* this is our host address */
    sa.sin_port= htons (portnum);              /* this is our port number */
    if ((s= socket (AF_INET, SOCK_STREAM, 0)) < 0) /* create socket */
        return(-1);
    if (bind (s, (struct sockaddr *)&sa,sizeof (struct sockaddr_in)) < 0) 
        close(s);
    listen (s, 3);                            /* max # of queued connects */
    return (s);
}
```
**TCP Server: Connection Accept**

/* wait for a connection to occur on a socket created with establish() */
int get_connection (int s)
{
    int t; /* socket of connection */
    if ((t = accept (s,NULL,NULL)) < 0) /* accept connection if there is one */
        return(-1);
    return(t);
}

**TCP Client: Socket Establishment and Connection**

int call_socket (char *hostname, unsigned short portnum) // how to call a server socket
{
    struct sockaddr_in sa;
    struct hostent *hp;
    int a, s;
    if ((hp = gethostbyname (hostname)) == NULL) { /* do we know the host's */
        errno=ECONNREFUSED; /* address? */
        return(-1); /* no */
    }
    memset (&sa,0,sizeof(sa)); /* clear our address */
    memcpy ((char *)&sa.sin_addr, hp->h_addr, hp->h_length); /* set address */
    sa.sin_family= hp->h_addrtype;
    sa.sin_port= htons ((u_short) portnum); /* host byte order to network byte order */
    if ((s= socket(hp->h_addrtype,SOCK_STREAM,0)) < 0) /* get socket */
        return(-1);
    if (connect (s, (struct sockaddr *)&sa, sizeof sa) < 0) /* connect (dialing) */
    {
        close(s);
        return(-1);
    }
    return(s);
}
TCP Server: Multiple Connections

```c
#include <errno.h> // you may still accept calls while processing previous connections.
#include <stdio.h> // For this reason you usually fork off child jobs to handle each connection.
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <netinet/in.h>
#include <netdb.h>

PORTNUM 50000 /* random port number, we need something */
void fireman(void);
void do_something(int);

main()
{
    int s, t;
    if ((s = establish(PORTNUM)) < 0) { /* plug in the phone */
        perror("establish");
        exit(1);
    }

    /* as children die we should catch their returns or else we get zombies, A Bad Thing.
     fireman() catches falling children. */
    signal(SIGCHLD, fireman); /* this eliminates zombies */

    for (;;) { /* loop for phone calls */
        if ((t = get_connection(s)) < 0) { /* get a connection */
            if (errno == EINTR) /* EINTR might happen on accept(), */
                continue; /* try again */
            perror("accept"); /* bad */
            exit(1);
        }

        switch (fork()) { /* try to handle connection */
            case -1 : /* bad news. scream and die */
                perror("fork");
                close(s); close(t); exit(1);
            case 0 : /* we're the child, do something */
                close(s);
                do_something(t);
                exit(0);
            default : /* we're the parent so look for */
                close(t); /* another connection */
                continue;
        }
    }
}

/* main() ends */
void fireman(void) { while (waitpid(-1, NULL, WNOHANG) > 0) ; }
/* this is the function that plays with the socket. it will be called after getting a connection. */
void do_something(int s) { /* do your thing with the socket here */ }
```

TCP Server: Multiple Connections (cont.)

```c
for (;;) { /* loop for phone calls */
    if (((= get_connection(s)) < 0) { /* get a connection */
        if (errno == EINTR) /* EINTR might happen on accept(), */
            continue; /* try again */
        perror("accept"); /* bad */
        exit(1);
    }

    switch (fork()) { /* try to handle connection */
        case -1 : /* bad news. scream and die */
            perror("fork");
            close(s); close(t); exit(1);
        case 0 : /* we're the child, do something */
            close(s);
            do_something(t);
            exit(0);
        default : /* we're the parent so look for */
            close(t); /* another connection */
            continue;
    }
}
/* main() ends */
void fireman(void) { while (waitpid(-1, NULL, WNOHANG) > 0) ; }
/* this is the function that plays with the socket. it will be called after getting a connection. */
void do_something(int s) { /* do your thing with the socket here */ }
```
### How to Talk between Sockets

```c
int read_data (int s, /* connected socket */
    char *buf, /* pointer to the buffer */
    int n) /* number of characters (bytes) we want */
{
    int bcount; /* counts bytes read */
    int br; /* bytes read this pass */

    // you don't usually get back the same number of characters that you asked for,
    // so you must loop until you have read the number of characters that you want.
    bcount = 0;
    br = 0;
    while (bcount < n) /* loop until full buffer due to stream-oriented communication */
    {
        if ((br = read (s, buf, n - bcount)) > 0) {
            bcount += br; /* increment byte counter */
            buf += br; /* move buffer ptr for next read */
        } else if (br < 0) /* signal an error to the caller */
            return (-1);
    }
    return (bcount);
}
```

### Socket Programming Example:
#### Internet File Server (p.490-1)

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>

#define SERVER_PORT 12345 /* arbitrary, but client & server must agree */
#define BUFSIZE 4096 /* block transfer size */

int main(int argc, char **argv)
{
    int c, s, bcount = 0, readBytes;
    char buf[BUFSIZE]; /* buffer for incoming file */
    struct sockaddr_in sa; /* info about server */
    struct sockopt_in channel; /* holds IP address */
    FILE *f;

    if (argc < 3) fatal("Usage: client server-name file-name");
    h = gethostbyname(argv[1]); /* look up host's IP address */
    if (h == NULL) fatal("gethostbyname failed");
    c = connect(s, (struct sockaddr_in *) &channel, sizeof(channel));
    if (c < 0) fatal("connect failed");

    /* Connection is now established. Send file name including 0 byte at end. */
    switch (argv[2], strlen(argv[2]) + 1);
    { /* Go get the file and write it to standard output */
        while (1) {
            readBytes = read(s, buf, BUFSIZE); /* read from socket */
            // check for end of file
            if (readBytes == 0) exit(0);
            if (write(1, buf, bytes) == -1) fatal("write to standard output failed");
        }
    }
    printf("exit", string);
    exit(1);
}
```
Socket Programming Example: Internet File Server (2)

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#define SERVER_PORT 12345
#define BUF_SIZE 4096
#define QUEUE_SIZE 10
int main(int argc, char *argv[])
{
    int s, b, l, i, aw, bytes, cm = 1;
    char buf[BUF_SIZE];
    struct sockaddr_in channel;
    /* buffer for outgoing file */
    struct sockaddr_in channel;
    /* hold's IP address */
    /* Build address structure to bind to socket */
    memset(&channel, 0, sizeof(channel));
    /* zero channel */
    channel.sin_family = AF_INET;
    channel.sin_addr.s_addr = htonl(INADDR_ANY);
    channel.sin_port = htons(SERVER_PORT);
    /* Passive open. Wait for connection. */
    s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    /* create socket */
    if (s < 0) fatal("socket failed");
    setsockopt(SOL_SOCKET, SO_REUSEADDR,
                (char *) &cm, sizeof(cm));
    b = bind(s, (struct sockaddr *)&channel,
             sizeof(channel));
    if (b < 0) fatal("bind failed");
    /* listen, QUEUE_SIZE; */
    /* specify queue size */
    /* Socket is now set up and bound. Wait for connection and process it. */
    while (1) {
        /* accept() */
        if (s < 0) fatal("accept failed");
        read(s, buf, BUF_SIZE); /* read file name from socket */
        /* Get and return the file */
        fl = open(buf, O_RDONLY); /* open the file to be sent back */
        if (fl < 0) fatal("open failed");
        while (1) {
            bytes = read(fl, buf, BUF_SIZE);
            /* read from file */
            if (bytes <= 0) break;
            /* check for end of file */
            write(s, buf, bytes);
            /* write bytes to socket */
        }
        close(fl);
        /* close file */
        close(s);
        /* close connection */
    }
}
```

Server code using sockets.

Summary

- Very widely used primitives started with TCP on UNIX
  - Notion of “sockets” as transport endpoints
  - Like simple set plus SOCKET, BIND, and ACCEPT

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOCKET</td>
<td>Create a new communication end point</td>
</tr>
<tr>
<td>BIND</td>
<td>Associate a local address with a socket</td>
</tr>
<tr>
<td>LISTEN</td>
<td>Announce willingness to accept connections; give queue size</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>Passively establish an incoming connection</td>
</tr>
<tr>
<td>CONNECT</td>
<td>Actively attempt to establish a connection</td>
</tr>
<tr>
<td>SEND</td>
<td>Send some data over the connection</td>
</tr>
<tr>
<td>RECEIVE</td>
<td>Receive some data from the connection</td>
</tr>
<tr>
<td>CLOSE</td>
<td>Release the connection</td>
</tr>
</tbody>
</table>
Where to get more information

° BSD manual: http://www.freebsd.org/
° http://java.sun.com