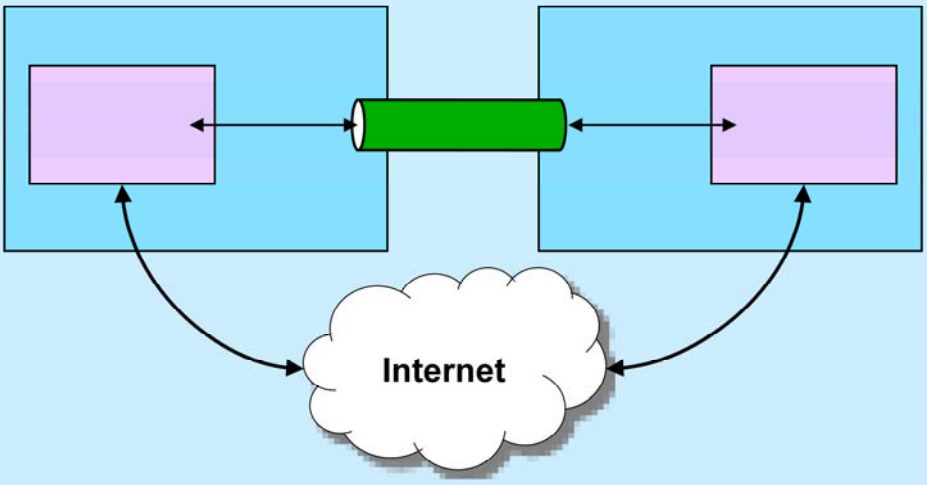
CS 33

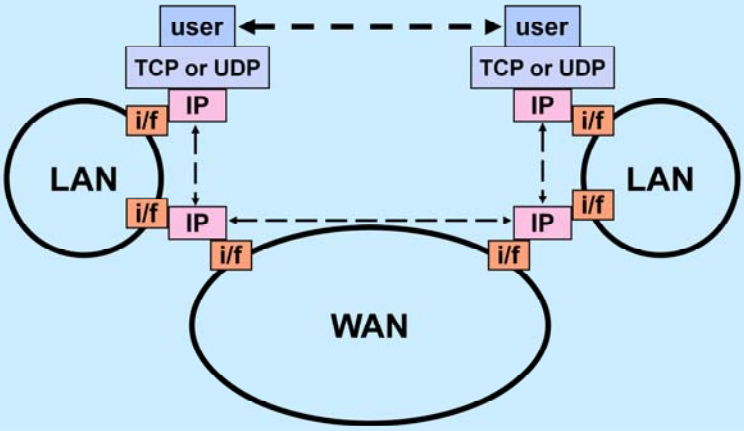
Network Programming

The source code used in this lecture is on the course web page.

Communicating Over the Internet



Internetworking



The Data Abstraction

- Byte stream
  - sequence of bytes
    - » as in pipes
  - any notion of a larger data aggregate is the responsibility of the programmer
- Record stream
  - sequence of variable-size "records"
  - boundaries between records maintained
  - receiver receives discrete records, as sent by sender

Reliability

- Two possibilities
  - don't worry about it
    - » just send it
      - if it arrives at its destination, that's good!
        - no verification
  - worry about it
    - » keep track of what's been successfully communicated
    - » retransmit until
      - data is received

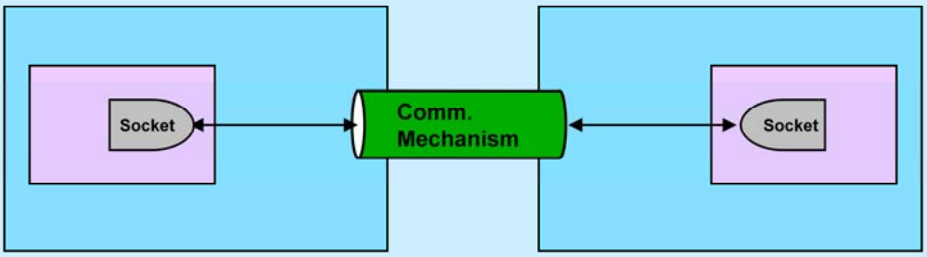
      or
      - it appears that "the network is down"

Sockets

Communication abstraction

endpoint of communication path

Sockets



Socket Parameters

- Styles of communication:
  - stream: reliable, two-way byte streams
  - datagram: unreliable, two-way record-oriented
  - sequenced packet: reliable, two-way record-oriented
  - etc.
- Communication domains
  - UNIX
    - » endpoints (sockets) named with file-system pathnames
    - » supports stream and datagram
  - Internet
    - » endpoints named with IP addresses
    - » supports stream and datagram
  - others
- Protocols
  - the means for communicating data
  - e.g., TCP/IP, UDP/IP

Datagrams in the UNIX Domain (1)

- **Steps**
  1) **create socket**

```
int socket(int domain, int type,
    int protocol);

fd = socket(PF_UNIX, SOCK_DGRAM, 0);
```


The first step is to create a socket. We request a datagram socket in the Unix domain. The third argument specifies the protocol, but since in this and pretty much all examples the protocol is determined by the first two arguments, a zero may be used.

Datagrams in the UNIX Domain (2)

2) set up name

```
struct  sockaddr_un {
  short sun_family;      /* PF_UNIX */
  char sun_path[108];    /* path name */
} name;

name.sun_family = PF_UNIX;
memcpy(name.sun_path, path, strlen(path));
```


The next step is to put the name for the socket into the unix-domain-specific version of the *sockaddr* structure, called the *sockaddr_un* structure.

Datagrams in the UNIX Domain (3)

3) bind name to socket

```
name_len = sizeof(name.sun_family) +
    strlen(name.sun_path);
bind(fd, (struct sockaddr *)&name,
    name_len);
```


The bind system call is used to pass this name to the kernel and use it to name the socket. Bind takes a generic *struct sockaddr* argument and is given the combined length of the first part of the structure and that portion of the second part that is used.

Datagrams in the UNIX Domain (4)

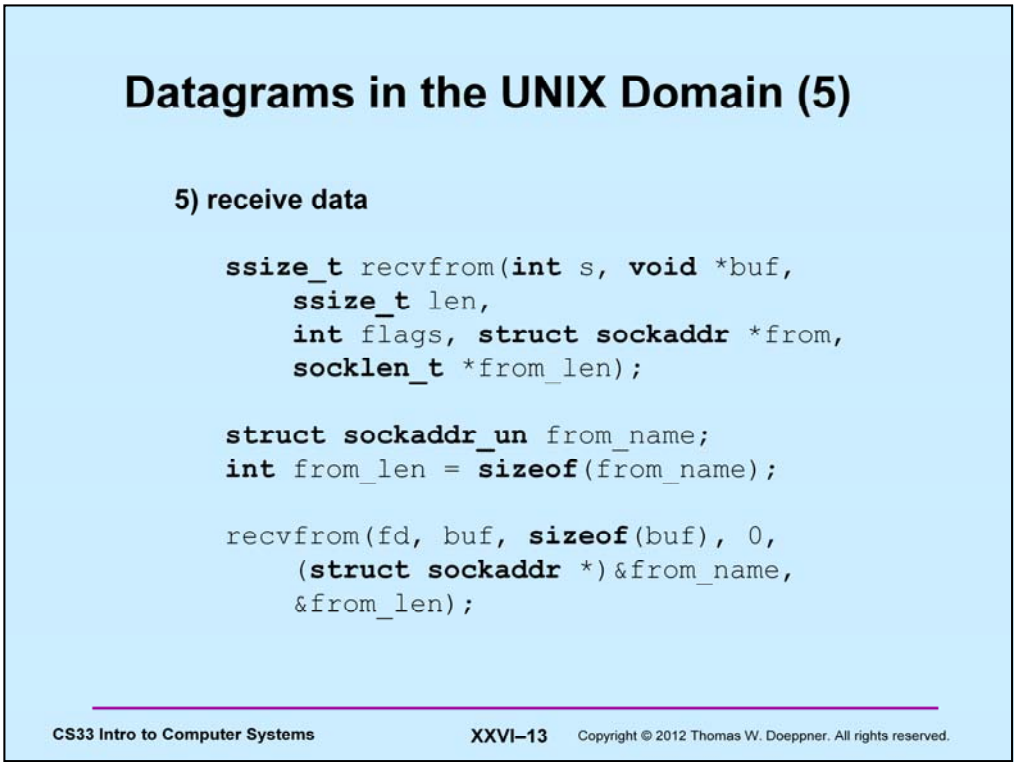
4) send data

```
ssize_t sendto(int fd, const void *buf,
    ssize_t len, int flags,
    const struct sockaddr *to,
    socklen_t to_len);

struct sockaddr_un to_name;
socklen_t to_len = sizeof(to_name);

sendto(fd, buf, sizeof(buf), 0,
    (struct sockaddr *)&to_name,
    to_len);
```


Use the *sendto* system call to send data to a particular destination socket.



Use the *recvfrom* system call not only to receive data, but also to obtain the sender's address.

UNIX Datagram Example (1)

- **Server side**

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#define NAME  "/home/twd/server"

main( ) {
  struct sockaddr_un sock_name;
  int fd, len;
  /* Step 1: create socket in UNIX domain for datagram
     communication.  The third argument specifies the
     protocol, but since there's only one such protocol in
     this domain, it's set to zero */
  if ((fd = socket(PF_UNIX, SOCK_DGRAM, 0)) < 0) {
    perror("socket");
    exit(1);
  }
```


UNIX Datagram Example (2)

```
/* Step 2: set up a sockaddr structure to contain the
   name we want to assign to the socket */
sock_name.sun_family = PF_UNIX;
strcpy(sock_name.sun_path, NAME);
len = strlen(NAME) + sizeof(sock_name.sun_family);


/* Step 3: bind the name to the socket */
if (bind(fd, (struct sockaddr *)&sock_name, len) < 0) {
  perror("bind");
  exit(1);
}
```


UNIX Datagram Example (3)

```
while (1) {
  char buf[1024];
  struct sockaddr_un from_addr;
  int from_len = sizeof(from_addr);
  int msg_size;

  /* Step 4: receive message from client */
  if ((msg_size = recvfrom(fd, buf, 1024, 0,
      (struct sockaddr *)&from_addr, &from_len)) < 0) {
    perror("recvfrom");
    exit(1);
  }
  buf[msg_size] = 0;
  printf("message from %s:\n%s\n", from_addr.sun_path,
      buf);
```

UNIX Datagram Example (4)

```
    /* Step 5: respond to client */
    if (sendto(fd, "thank you", 9, 0,
        (const struct sockaddr *)&from_addr,
        from_len) < 0) {
      perror("sendto");
      exit(1);
    }
  }
}
```

UNIX Datagram Example (5)

- **Client Side**

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

#define SNAME "/home/twd/server"
#define CNAME "/home/twd/client"

int main( ) {
  struct sockaddr_un server_name;
  struct sockaddr_un client_name;
  int fd, server_len, client_len;
```


UNIX Datagram Example (6)

```
/* Step 1: create socket in UNIX domain for datagram
   communication.  The third argument specifies the
   protocol, but since there's only one such protocol
   in this domain, it's set to zero */
if ((fd = socket(PF_UNIX, SOCK_DGRAM, 0)) < 0) {
  perror("socket");
  exit(1);
}

/* Step 2: set up a sockaddr structure to contain the
   name we want to assign to the socket */
client_name.sun_family = PF_UNIX;
strcpy(client_name.sun_path, CNAME);
client_len = strlen(CNAME) +
    sizeof(client_name.sun_family);
```


UNIX Datagram Example (7)

```
/* Step 3: bind the name to the socket */
if (bind(fd, (struct sockaddr *)&client_name,
    client_len) < 0) {
  perror("bind");
  exit(1);
}
```


It's not necessary to bind a name to the client socket — doing so is important only if required by the server (who might, for example, deal with requests only from certain clients). It's needed here so the server will know whom to send a response to.

UNIX Datagram Example (8)

```
/* Step 4: set up server's name */
server_name.sun_family = PF_UNIX;
strcpy(server_name.sun_path, SNAME);
server_len = strlen(SNAME) +
    sizeof(server_name.sun_family);

while (1) {
  char buf[1024];
  int msg_size;

  if (fgets(buf, 1024, stdin) == 0)
    break;
```

UNIX Datagram Example (9)

```
/* Step 5: send data to server */
if (sendto(fd, buf, strlen(buf), 0,
    (const struct sockaddr *)&server_name,
    server_len) < 0) {
  perror("sendto");
  exit(1);
}
```

UNIX Datagram Example (10)

```
    /* Step 6: receive response from server */
    if ((msg_size = recvfrom(fd, buf, 1024, 0, 0,
        0)) < 0) {
      perror("recvfrom");
      exit(1);
    }
    buf[msg_size] = 0;
    printf("Server says: %s\n", buf);
  }
  return(0);
}
```

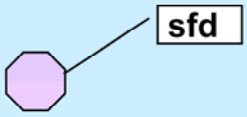
Reliable Communication

- The promise …
  - what is sent is received
  - order is preserved
- Set-up is required
  - two parties agree to communicate
    - » each side keeps track of what is sent, what is received
    - » received data is acknowledged
    - » unack'd data is re-sent
- The standard scenario
  - server receives connection requests
  - client makes connection requests

Streams in the Inet Domain (1)

- **Server steps**
  - **1) create socket**

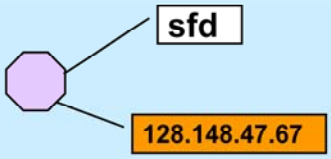
```
sfd = socket(PF_INET, SOCK_STREAM, 0);
```



Streams in the Inet Domain (2)

- **Server steps**
  2) bind name to socket

```
bind(sfd,
  (struct sockaddr *)&my_addr, sizeof(my_addr));
```



Internet Addresses

- IP (internet protocol) address
  - one per network interface
  - 32 bits (IPv4)
    - » 5 per square foot of RI
  - 128 bits (IPv6)
    - » one billion per cubic mile of a sphere whose radius is the mean distance from the Sun to the (former) planet Pluto
- Port number
  - one per application instance per machine
  - 16 bits
    - » port numbers less than 1024 are reserved for privileged applications

Some Details …

- **Server may have multiple interfaces; we want to be able to receive on all of them**
- **Must convert from *host byte order* to *network byte order***

```
struct sockaddr_in {
  short sin_family;
  u_short sin_port;
  struct in_addr sin_addr;
  char sin_zero[8];        /* padding */
} my_addr;

my_addr.sin_family = PF_INET;
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
my_addr.sin_port = htons(port);
```

Setting the network address is surprisingly complicated. One issue is that a machine might have multiple addresses — this is often the case for servers. Rather than having to specify all of them, one simply gives the "wildcard" address, meaning all the addresses on the machine. This is useful even on a machine with just one network interface, since the wildcard address in that case refers to just the one interface.

The other issue has to do with the byte order of integers on the local computer. Since different computers might have different byte orders, this could be a problem. One byte order is chosen as the standard; for network use, all must convert to that order if necessary. The macros "htonl()" ("host to network long") and "htons()" ("host to network short") do the conversions if necessary.

Host Names

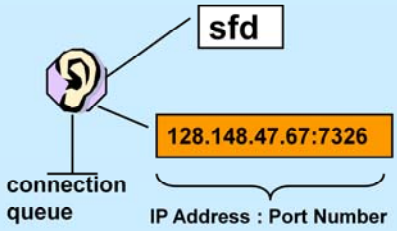
- Hosts are referred to by "DNS names"
  - e.g. maytag.cs.brown.edu
- DNS (Domain Name Service) is a distributed database
  - translates names to addresses
  - maytag.cs.brown.edu
    - » 128.148.31.201
    - » 128.148.38.201

    } dot notation: dots separate bytes
- The library routine *gethostbyname* performs DNS lookups

Streams in the Inet Domain (3)

- **Server steps**
  - **3) put socket in "listening mode"**

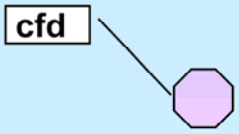
```
int listen(int sfd, int MaxQueueLength);
```



Streams in the Inet Domain (4)

- Cient steps
  1) create socket

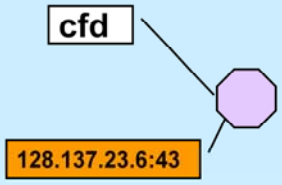
```
cfd = socket(PF_INET, SOCK_STREAM, 0);
```



Streams in the Inet Domain (5)

- **Client steps**
  - **2) bind name to socket**

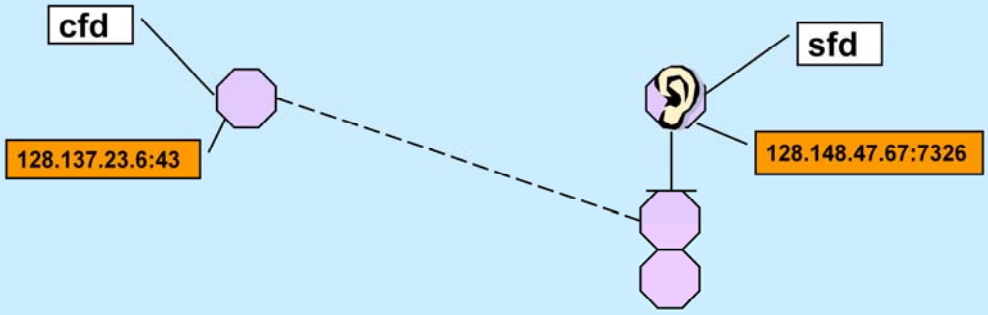
```
bind(cfd,
  (struct sockaddr *)&my_addr, sizeof(my_addr));
```



Streams in the Inet Domain (6)

- **Client steps**
  - **3) connect to server**

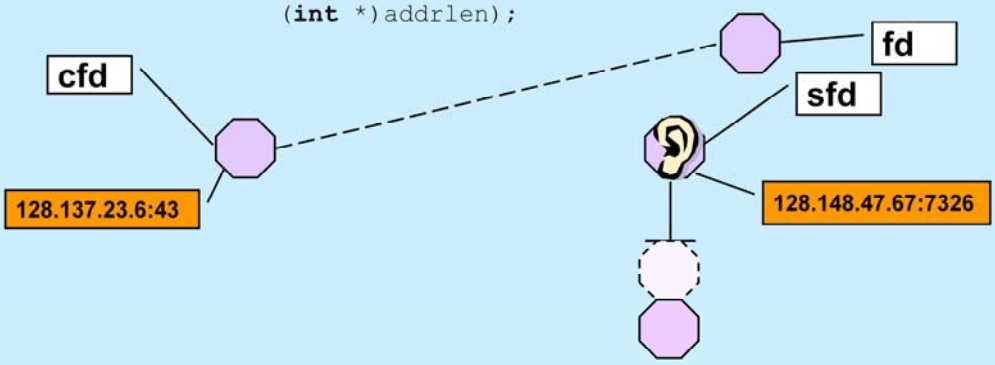
```
connect(cfd, (struct sockaddr *)&server_addr,
    sizeof(server_addr));
```



Streams in the Inet Domain (6)

- **Server steps**
  - **4) accept connection**

```
fd = accept((int)sfd, (struct sockaddr *)addr,
    (int *)addrlen);
```



Inet Stream Example (1)

- **Server side**

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

int main(int argc, char *argv[ ]) {
  struct sockaddr_in my_addr;
  int lsock;
  void serve(int);
  if (argc != 2) {
    fprintf(stderr, "Usage: server port\n");
    exit(1);
  }
```


Inet Stream Example (2)

```
// Step 1: establish a socket for TCP
if ((lsock = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
  perror("socket");
  exit(1);
}
```


Inet Stream Example (3)

```
/* Step 2: set up our address */
memset(&my_addr, 0, sizeof(my_addr));
my_addr.sin_family = PF_INET;
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
my_addr.sin_port = htons(atoi(argv[1]));

/* Step 3: bind the address to our socket */
if (bind(lsock, (struct sockaddr *)&my_addr,
    sizeof(my_addr)) < 0) {
  perror("bind");
  exit(1);
}
```


The *memset* command copies some number of instances of its second argument into what its first argument points to. The number of instances is given by its third argument. As used here it is setting *my_addr* to all zeroes.

Inet Stream Example (4)

```
/* Step 4: put socket into "listening mode" */
if (listen(lsock, 5) < 0) {
  perror("listen");
  exit(1);
}
while (1) {
  int csock;
  struct sockaddr_in client_addr;
  int client_len = sizeof(client_addr);

  /* Step 5: receive a connection */
  csock = accept(lsock,
      (struct sockaddr *)&client_addr, &client_len);
  printf("Received connection from %s\n",
      inet_ntoa(client_addr.sin_addr));
```


The library routine "inet_ntoa" converts a 32-bit network address into an ASCII string in "dot notation" (bytes are separated by dots).

Inet Stream Example (5)

```
    switch (fork( )) {
    case -1:
      perror("fork");
      exit(1);
    case 0:
      // Step 6: create a new process to handle connection
      serve(csock);
      exit(0);
    default:
      close(csock);
      break;
    }
  }
}
```

Inet Stream Example (6)

```
void serve(int fd) {
  char buf[1024];
  int count;

  // Step 7: read incoming data from connection
  while ((count = read(fd, buf, 1024)) > 0) {
    write(1, buf, count);
  }
  if (count == -1) {
    perror("read");
    exit(1);
  }
  printf("connection terminated\n");
}
```

Inet Stream Example (7)

- **Client side**

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

int main(int argc, char *argv[]) {
  struct sockaddr_in server_addr;
  int sock;
  struct hostent *hostinfo;

  char buf[1024];
  if (argc != 3) {
    fprintf(stderr, "Usage: client host port\n");
    exit(1);
  }
```


Inet Stream Example (8)

```
// set up socket for TCP
if ((sock = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
  perror("socket");
  exit(1);
}

// Step 2: find the internet address of the server
if ((hostinfo = gethostbyname(argv[1])) == 0) {
  fprintf(stderr, "Host %s does not exist\n", argv[1]);
  exit(1);
}
memset(&server_addr, 0, sizeof(server_addr));
server_addr.sin_family = PF_INET;
memcpy(&server_addr.sin_addr, hostinfo->h_addr_list[0],
    hostinfo->h_length);
server_addr.sin_port = htons(atoi(argv[2]));
```


We call gethostbyname() to translate from the DNS name of the host to its IP address. The result is placed in the storage pointed to by hostinfo, a *struct hostent*. In general this contains an array of IP addresses, one for each network interface of the host. We choose the first one.

Inet Stream Example (9)

```
    // Step 3: connect to the server
    if (connect(sock, (struct sockaddr *)&server_addr,
        sizeof(server_addr)) < 0) {
      perror("connect");
      exit(1);
    }

    // Step 4: send data to the server
    while(fgets(buf, 1024, stdin) != 0) {
      if (write(sock, buf, strlen(buf)) < 0) {
        perror("write");
        exit(1);
      }
    }
    return(0);
  }
```