Network Clients

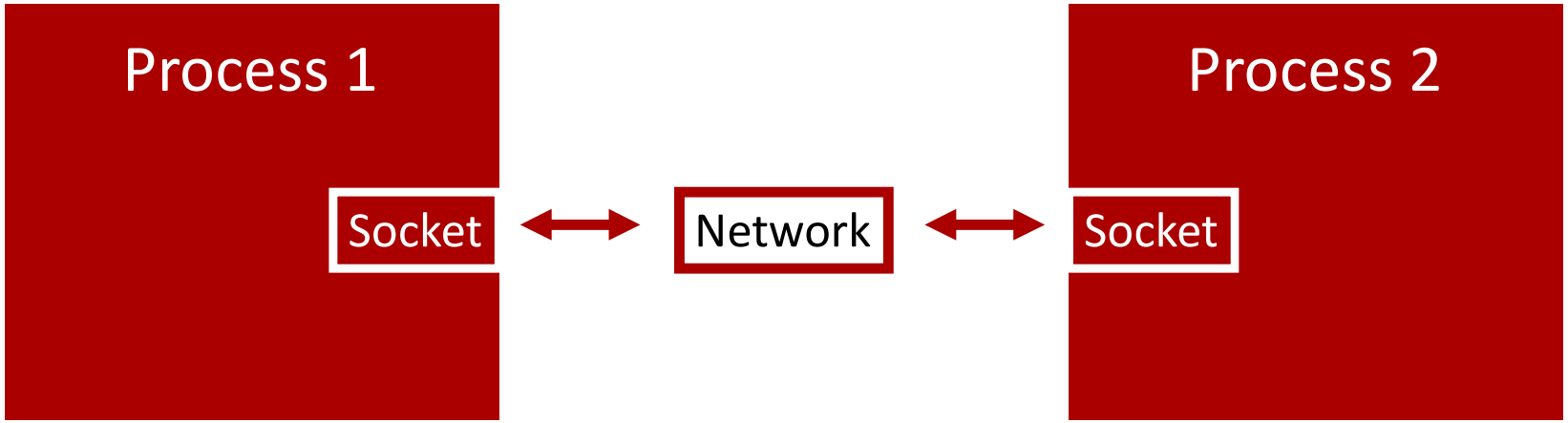
Benjamin Brewster

Except as noted, all images copyrighted with Creative Commons licenses,
with attributions given whenever available

Berkeley Sockets

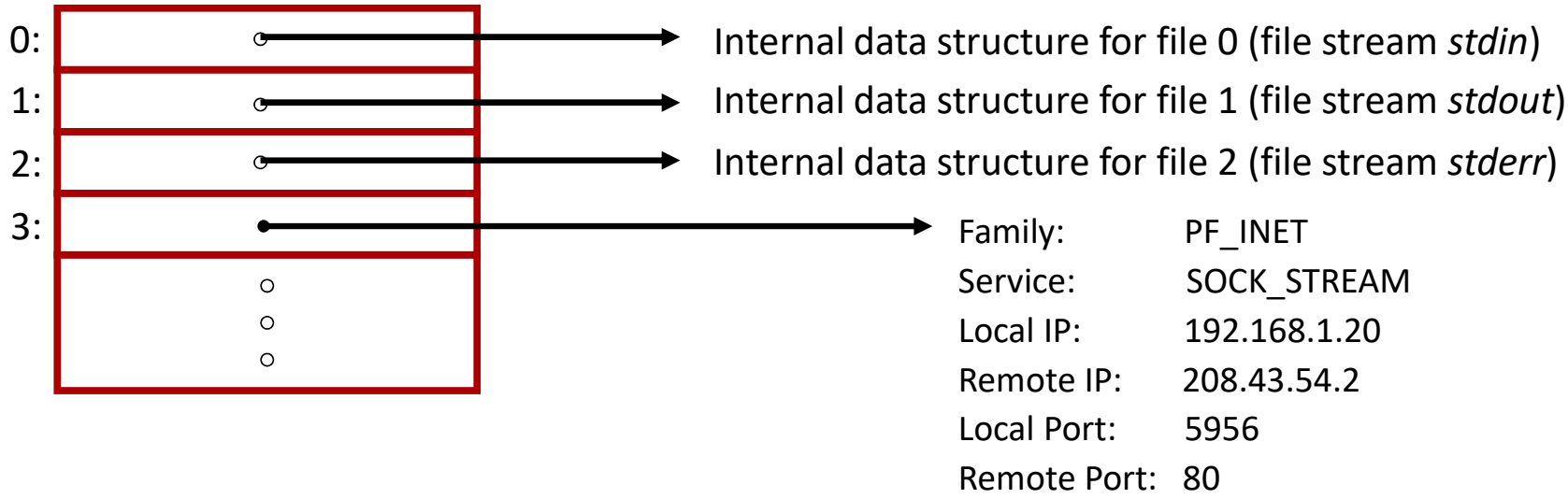
- Developed in the early 1980s for BSD Unix under a grant from DARPA
- The *de facto* network communication method across local area networks (LAN) and the internet.
  - Other transmission methods exist, but they require different transport protocols.
    - i.e., you'd have to write your own version of TCP for a different network protocol
      - Seriously, don't do this. Just use sockets
        - Decades of research! Thousands of scientists, academics, engineers, and hobbyists!
          - Think of the children!

Network Sockets



- Berkeley Socket API
  - A "socket" is the endpoint of a communication link between two processes
  - The socket API treats network connections like *files* as much as possible

File Descriptor Table - Sockets Show Up as Files



- The file descriptor number returned by `open()` is an index into an array of pointers to internal OS data structures
- Sockets are added to this table of descriptors in the same way

Multiple Process Communication

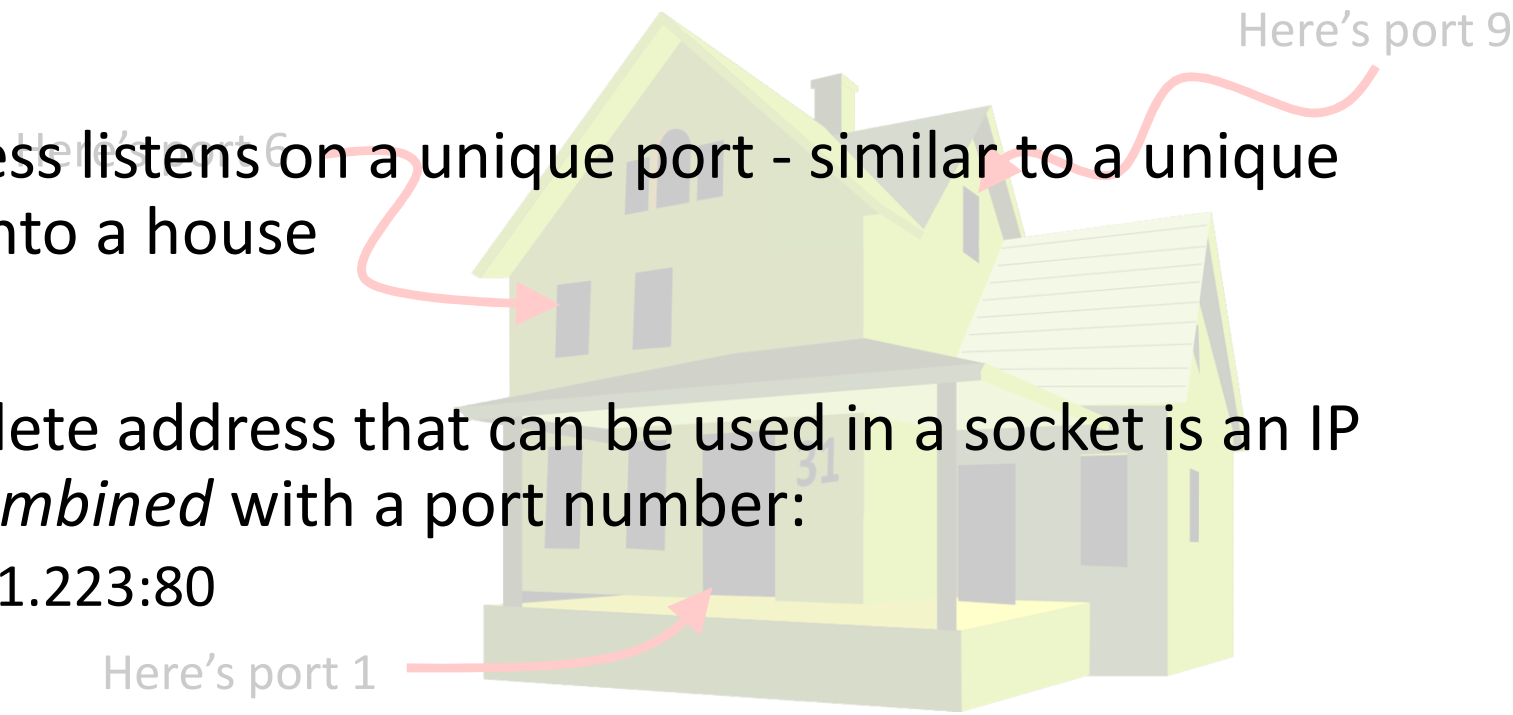
- Many different processes can be running on one computer
- However, an IP address only identifies the interface on the computer, not the process
- How do we know which process is communicating at that particular interface's IP address?

Ports

- This house has address 31

Ports

- *Ports* are used to reach a specific process on a machine
- Each process listens on a unique port - similar to a unique entrance into a house
- So a complete address that can be used in a socket is an IP address *combined* with a port number:
  - 43.144.31.223:80



Socket Documentation

- Most socket related man pages are in the "3n" section
  - **`man -s 3n socket`**
  - **`man -k socket`**

- All the info you need to use the network library is scattered across different man pages

- It’s definitely best to work from a known good network program starting point! Stay tuned!

Creating and Connecting a Socket on the Client

- Process:
  1) Create the socket endpoint with `socket()`
  2) Connect the socket to the server with `connect()`
  3) Use `read()` and `write()`, or `send()` and `recv()`, to transfer data to and from the socket (which is sent automatically to and from the socket on the server)
- Sockets act like files, in that you can `read()` and `write()` to them
- `send()` and `recv()` are specialized, and can take special flags

Creating the Socket

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

Returns file descriptor or -1

For general-purpose sockets that can connect across a network, use AF_INET

For sockets that are used ONLY for same-machine IPC, use AF_UNIX

For TCP, use SOCK_STREAM
For UDP, use SOCK_DGRAM

Use 0 for normal behavior

```
int socketFD = socket(AF_INET, SOCK_STREAM, 0);
if (socketFD == -1) {
      perror("Hull breach: socket()"); exit(1);
}
```

Connecting the Socket to an Address

```
int connect(int sockfd, struct sockaddr* address, size_t address_size);
```

Returns 0 on success, -1 on failure

Socket you want to connect

A struct that holds the *address* of where you're connecting, plus other settings;

More on this coming up

The size of the address struct

```
if (connect(socketFD, (struct sockaddr*)&serverAddress, sizeof(serverAddress)))
{
      perror("Hull breach: connect()"); exit(1);
}
```

Filling the Address Struct: IP Address

- Getting the actual address into a form `connect()` can use it is tricky:

```
struct sockaddr_in serverAddress;

serverAddress.sin_family = AF_INET;
serverAddress.sin_port = htons(7000);
serverAddress.sin_addr.s_addr = inet_addr("192.168.1.1");
```

`htons()`: **h**ost-**to**-**n**etwork-**s**hort

Converts from *host/PC* byte order (LSB) to *network* byte order (MSB)

PCs store bytes with smallest digit first, but networks expect largest digit first

`inet_addr()` converts a standard dotted IP address string into an *integer* format that sockaddr_in requires

Filling the Address Struct: Domain Name

- Client connecting to server:

Do a DNS lookup and return address information

```
struct sockaddr_in serverAddress;
struct hostent* serverHostInfo;

serverHostInfo = gethostbyname("www.oregonstate.edu");
if (serverHostInfo == NULL) {
        fprintf(stderr, "could not resolve server host name\n");
        exit(1);
}
```

This will be used to connect to a server on port 80

```
serverAddress.sin_family = AF_INET;
serverAddress.sin_port = htons(80);
```

Destination, copying to

Preserve the special arrangement of the bytes in these variables by copying the bytes in the given order, regardless of format, structure, or meaning

```
memcpy((char*)&serverAddress.sin_addr.s_addr,
        (char*)serverHostInfo->h_addr, serverHostInfo->h_length);
```

Source, copying from

Sending Data

Socket file descriptor

Pointer to data that should be sent

```
ssize_t send(int sockfd, void *message, size_t message_size, int flags);
```

Returns number of bytes sent

Number of bytes to send, starting at address in message

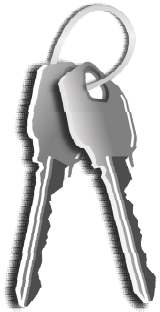
Configuration flags

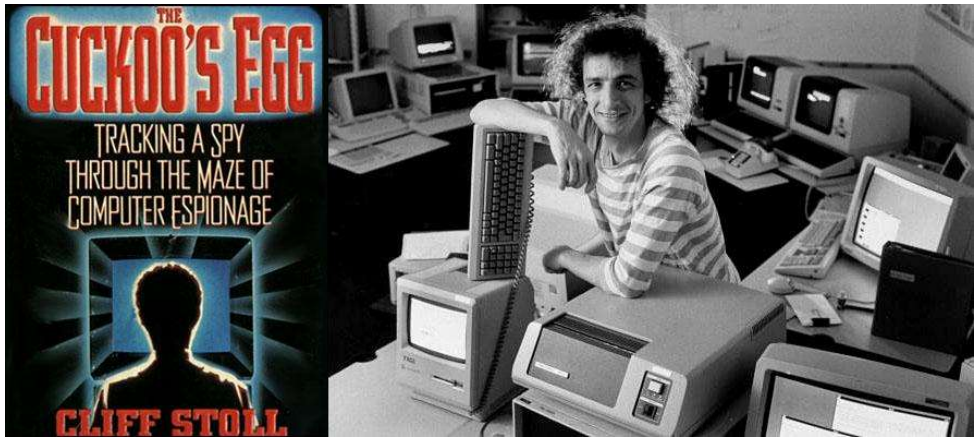
```
char msg[1024];
…
r = send(socketFD, msg, 1024, 0);
if (r < 1024)
     {} // handle possible error
```

If this happens, you'll have to call `send()` again to send what didn't get sent previously

Sending Data

- Send will block until all the data has been sent, the connection goes away, or a signal handler interrupts the `write()` system call
- Remember that internet connections fail all the time
  - Client intentionally disconnects (STOP button in a web browser)
  - Network failure
  - etc.





Receiving Data

Socket file descriptor

Pointer to where received data should be written

```
ssize_t recv(int sockfd, void *buffer, size_t buffer_size, int flags);
```

Returns number of bytes read

Maximum number of bytes to receive

Configuration flags

```
char buffer[1024];
memset(buffer, '\0', sizeof(buffer));
r = recv(socketFD, buffer, sizeof(buffer) - 1, 0);
if (r < sizeof(buffer) - 1)
        {} // handle possible error
```

- if r == -1, ERROR
- if 0 < r < sizeof(buffer) - 1, there may be more data
- if r == 0, sender shut down OR sent a 0-length packet OR 0 bytes were requested

Gray is really, really unlikely

Receiving Data

- Data may arrive in odd size bundles!
- `recv()` or `read()` will return exactly the amount of data that has already arrived
- `recv()` and `read()` will block if the connection is open but *no* data is available
  - So be careful to match what you send with what you receive, or use:

```
fcntl(socketFD, F_SETFL, O_NONBLOCK);
```

…to set the socket to not block if there's no data, but that means you're polling the socket, waiting for data; `select()` would be better (see next lecture!)

Receiving Data - Using Control Codes

- Similar to controlling data being sent through pipes, you can watch for the amount of data coming through `recv()` if you know how much there should be, or use codes:

```
…
char completeMessage[512], readBuffer[10];
memset(completeMessage, '\0', sizeof(completeMessage)); // Clear the buffer

while (strstr(completeMessage, "@@") == NULL) // As long as we haven't found the terminal...
{
          memset(readBuffer, '\0', sizeof(readBuffer)); // Clear the buffer
          r = recv(socketFD, readBuffer, sizeof(readBuffer) - 1, 0); // Get the next chunk
          strcat(completeMessage, readBuffer); // Add that chunk to what we have so far
          printf("PARENT: Message received from child: \"%s\", total: \"%s\"\n", readBuffer, completeMessage);
          if (r == -1) { printf("r == -1\n"); break; } // Check for errors
          if (r == 0) { printf("r == 0\n"); break; }
}

int terminalLocation = strstr(completeMessage, "@@") - completeMessage; // Where is the terminal
completeMessage[terminalLocation] = '\0'; // End the string early to wipe out the terminal
printf("PARENT: Complete string: \"%s\"\n", completeMessage);
…
```

Debugging the Contents of Buffers

- Often, when writing send and receive functions, you'll get garbage. Here's an easy way to actually check what's in a buffer:

```
int x = 0;
printf("CHAR INT\n");
for (x = 0; x < strlen(buffer); x++)
  printf(" %c   %d\n", buffer[x], buffer[x]);
```

Show all chars up to the first newline

- Or:

```
int x = 0;
printf("CHAR INT\n");
for (x = 0; x < sizeof(buffer); x++)
  printf(" %c   %d\n", buffer[x], buffer[x]);
```

Show all chars in the entire array

Debugging the Contents of Buffers: Results

```
CHAR INT
 o   111
 O   79
 0   48
     32      Space
 l   108
     10      New line
 L   76
 1   49
     0       Null terminator
```

- Look up these ints in a good ASCII table, like this one:
  http://www.asciitable.com

client.c 1 of 2

This is a basic client program that can send and receive.

It is intended to pair with server.c

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

void error(const char *msg) { perror(msg); exit(0); } // Error function used for reporting issues

int main(int argc, char *argv[])
{
    int socketFD, portNumber, charsWritten, charsRead;
    struct sockaddr_in serverAddress;
    struct hostent* serverHostInfo;
    char buffer[256];

    if (argc < 3) { fprintf(stderr,"USAGE: %s hostname port\n", argv[0]); exit(0); } // Check usage & args

    // Set up the server address struct
    memset((char*)&serverAddress, '\0', sizeof(serverAddress)); // Clear out the address struct
    portNumber = atoi(argv[2]); // Get the port number, convert to an integer from a string
    serverAddress.sin_family = AF_INET; // Create a network-capable socket
    serverAddress.sin_port = htons(portNumber); // Store the port number
    serverHostInfo = gethostbyname(argv[1]); // Convert the machine name into a special form of address
    if (serverHostInfo == NULL) { fprintf(stderr, "CLIENT: ERROR, no such host\n"); exit(0); }
    memcpy((char*)&serverAddress.sin_addr.s_addr, (char*)serverHostInfo->h_addr, serverHostInfo->h_length);
                                                                                    // Copy in the address
```

client.c 2 of 2

```
    // Set up the socket
    socketFD = socket(AF_INET, SOCK_STREAM, 0); // Create the socket
    if (socketFD < 0) error("CLIENT: ERROR opening socket");

    // Connect to server
    if (connect(socketFD, (struct sockaddr*)&serverAddress, sizeof(serverAddress)) < 0) // Connect socket to addy
        error("CLIENT: ERROR connecting");

    // Get input message from user
    printf("CLIENT: Enter text to send to the server, and then hit enter: ");
    memset(buffer, '\0', sizeof(buffer)); // Clear out the buffer array
    fgets(buffer, sizeof(buffer) - 1, stdin); // Get input from the user, trunc to buffer - 1 chars, leaving \0
    buffer[strcspn(buffer, "\n")] = '\0'; // Remove the trailing \n that fgets adds

    // Send message to server
    charsWritten = send(socketFD, buffer, strlen(buffer), 0); // Write to the server
    if (charsWritten < 0) error("CLIENT: ERROR writing to socket");
    if (charsWritten < strlen(buffer)) printf("CLIENT: WARNING: Not all data written to socket!\n");

    // Get return message from server
    memset(buffer, '\0', sizeof(buffer)); // Clear out the buffer again for reuse
    charsRead = recv(socketFD, buffer, sizeof(buffer) - 1, 0); // Read data from the socket, leaving \0 at end
    if (charsRead < 0) error("CLIENT: ERROR reading from socket");
    printf("CLIENT: I received this from the server: \"%s\"\n", buffer);

    close(socketFD); // Close the socket
    return 0;
}
```

Client/Server Results

```
$ gcc -o client client.c
$ gcc -o server server.c
$ ./server 51717 &
[1] 21094
$ ./client localhost 51717
CLIENT: Enter text to send to the server, and then hit enter: AWESOMESAUCE
SERVER: I received this from the client: "AWESOMESAUCE"
CLIENT: I received this from the server: "I am the server, and I got your message"
[1]+  Done                    ./server 51717
$
```