CSE 333
Lecture 17 -- network programming intro

Hal Perkins

Department of Computer Science & Engineering

University of Washington

Administrivia

HW3 due Thursday night

HW4 out by Friday morning

  Due last Wednesday of the quarter (2 weeks from today)

  OK to use up to 2 late days if you still have them

Section tomorrow: networking (DNS, TCP client side, ...)

No more exercises due this week

  Next exercise out after section, due next Monday before class

Today

Network programming

    dive into the Berkeley / POSIX sockets API

Files and file descriptors

Remember open, read, write, and close?

POSIX system calls for interacting with files

open( ) returns a *file descriptor*

an integer that represents an open file

inside the OS, it's an index into a table that keeps track of any state associated with your interactions, such as the file position

you pass the file descriptor into read, write, and close

Networks and sockets

UNIX likes to make all I/O look like file I/O

the good news is that you can use read( ) and write( ) to interact with remote computers over a network!

just like with files....

your program can have multiple network channels open at once

you need to pass read( ) and write( ) a *file descriptor* to let the OS know which network channel you want to write to or read from

a file descriptor used for network communications is a **socket**

Pictorially

128.95.4.33

Web server

fd 5   fd 8   fd 9   fd 3

80   80

index.html

pic.png

client   client
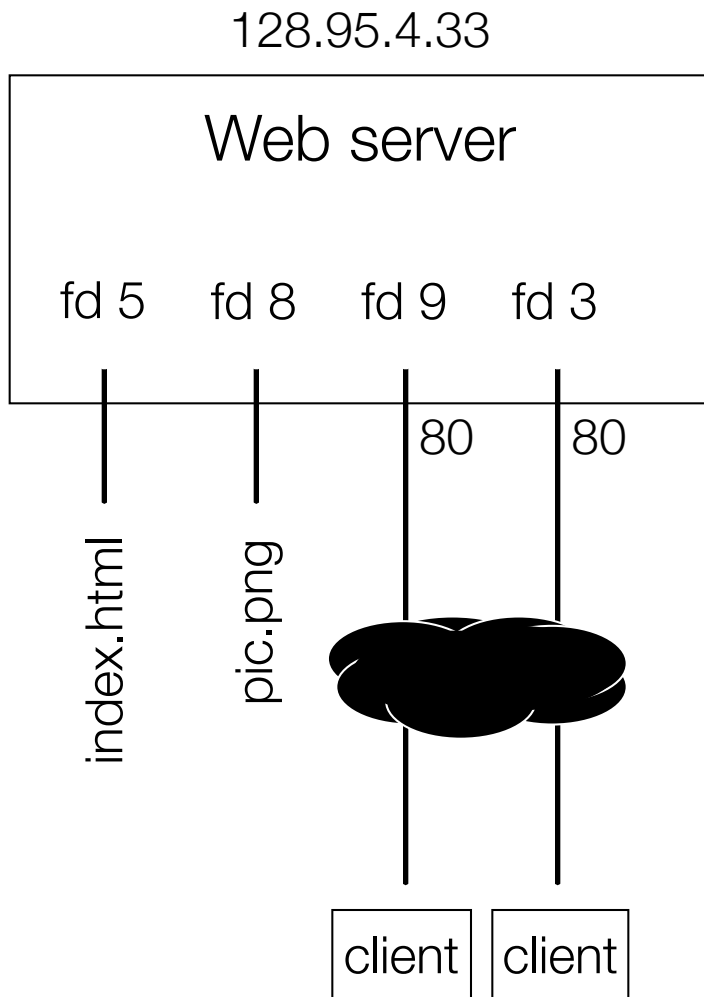
10.12.3.4 : 5544   **44.1.19.32** : 7113

OS's descriptor table

file descriptor	type	connected to?
0	pipe	stdin (console)
1	pipe	stdout (console)
2	pipe	stderr (console)
3	TCP socket	local: 128.95.4.33:80 remote: 44.1.19.32:7113
5	file	index.html
8	file	pic.png
9	TCP socket	local: 128.95.4.33:80 remote: 102.12.3.4:5544

Types of sockets

Stream sockets

    for connection-oriented, point-to-point, reliable bytestreams

        uses TCP, SCTP, or other stream transports

Datagram sockets

    for connection-less, one-to-many, unreliable packets

        uses UDP or other packet transports

Raw sockets

    for layer-3 communication (raw IP packet manipulation)

Stream sockets

Typically used for client / server communications

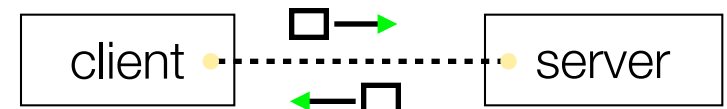> but also for other architectures, like peer-to-peer

Client

> an application that establishes a connection to a server

Server

> an application that receives connections from clients

client ┄┄┄┄┄▶ server

1. establish connection

client ┄┄┄┄┄ server

2. communicate

client ┄┄ ┄┄ server

3. close connection

Datagram sockets

1. create socket

host

host

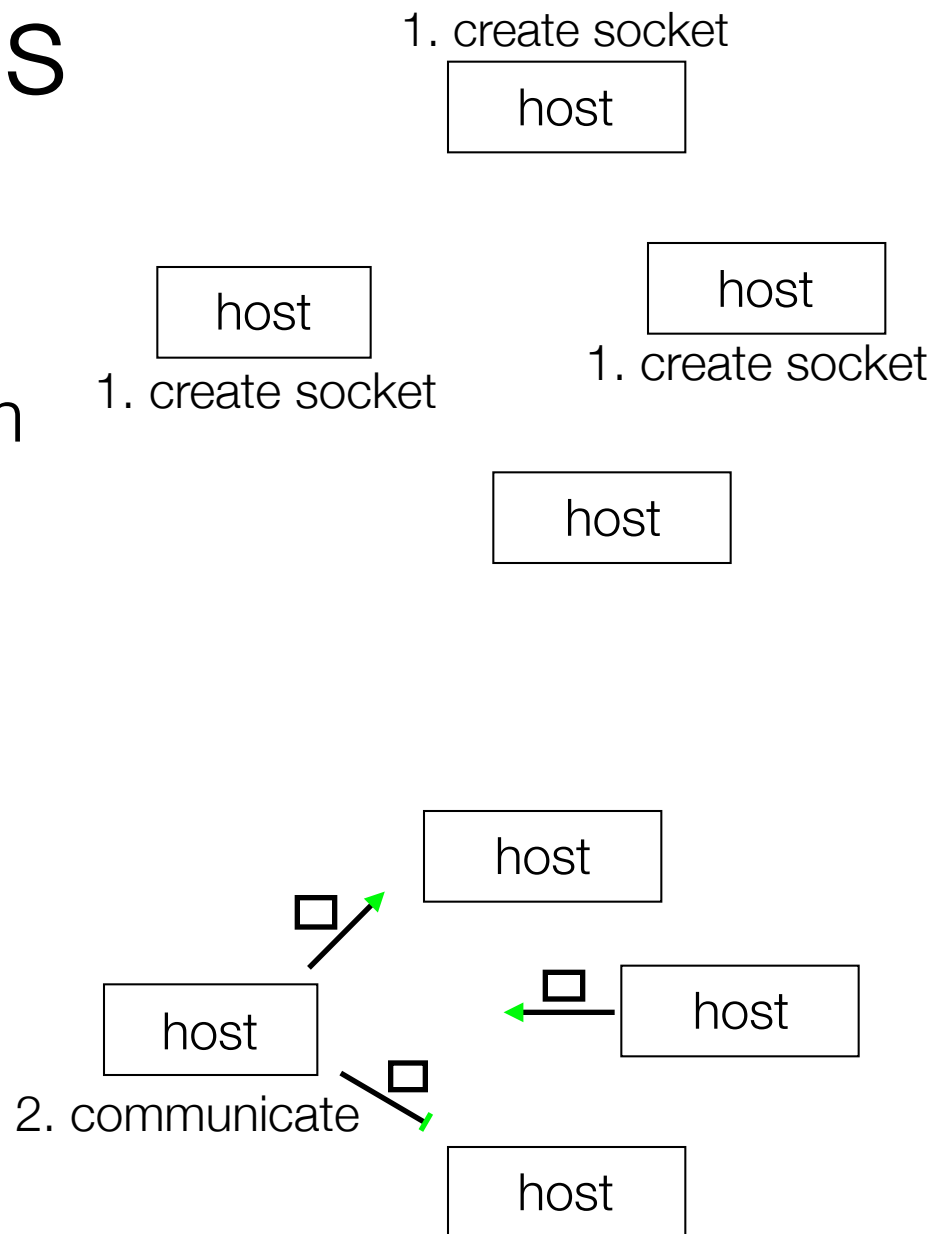1. create socket

host

1. create socket

host

Used less frequently than stream sockets

they provide no flow control, ordering, or reliability

Often used as a building block

streaming media applications

sometimes, DNS lookups

host

host

host

2. communicate

host

The sockets API

Berkeley sockets originated in 4.2 BSD Unix circa 1983

    it is the standard API for network programming

        available on most OSs

POSIX socket API

    a slight updating of the Berkeley sockets API

        a few functions were deprecated or replaced

        better support for multi-threading was added

Let's dive into it!

We'll start by looking at the API from the point of view of a client connecting to a server over TCP

there are five steps:

figure out the IP address and port to which to connect

create a socket

connect the socket to the remote server

read( ) and write( ) data using the socket

close the socket

Connecting from a client to a server.

Step 1. Figure out the IP address and port to which to connect.

Network addresses

For IPv4, an IP address is a 4-byte tuple

    e.g., 128.95.4.1 (80:5f:04:01 in hex)

For IPv6, an IP address is a 16-byte tuple

    e.g., 2d01:0db8:f188:0000:0000:0000:0000:1f33

      2d01:0db8:f188::1f33 in shorthand

IPv4 address structures

```
// Port numbers and addresses are in *network order*.

// A mostly-protocol-independent address structure.
struct sockaddr {
    short int      sa_family;    // Address family; AF_INET, AF_INET6
    char           sa_data[14];  // 14 bytes of protocol address
};


// An IPv4 specific address structure.
struct sockaddr_in {
    short int          sin_family;  // Address family, AF_INET == IPv4
    unsigned short int sin_port;    // Port number
    struct in_addr     sin_addr;    // Internet address
    unsigned char      sin_zero[8]; // Same size as struct sockaddr
};


struct in_addr {
    uint32_t s_addr;  // IPv4 address
};
```

IPv6 address structures

```
// A structure big enough to hold either IPv4 or IPv6 structures.
struct sockaddr_storage {
    sa_family_t  ss_family;     // address family

    // a bunch of padding; safe to ignore it.
    char        __ss_pad1[_SS_PAD1SIZE];
    int64_t     __ss_align;
    char        __ss_pad2[_SS_PAD2SIZE];
};

// An IPv6 specific address structure.
struct sockaddr_in6 {
    u_int16_t        sin6_family;   // address family, AF_INET6
    u_int16_t        sin6_port;     // Port number
    u_int32_t        sin6_flowinfo; // IPv6 flow information
    struct in6_addr  sin6_addr;     // IPv6 address
    u_int32_t        sin6_scope_id; // Scope ID
};

struct in6_addr {
    unsigned char   s6_addr[16];   // IPv6 address
};
```

Generating these structures

Often you have a string representation of an address

how do you generate one of the address structures?

```
#include <stdlib.h>                                          genaddr.cc
#include <arpa/inet.h>

int main(int argc, char **argv) {
  struct sockaddr_in sa; // IPv4
  struct sockaddr_in6 sa6; // IPv6

  // IPv4 string to sockaddr_in.
  inet_pton(AF_INET, "192.0.2.1", &(sa.sin_addr));

  // IPv6 string to sockaddr_in6.
  inet_pton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr));

  return EXIT_SUCCESS;
}
```

Generating these structures

How about going in reverse?

```
#include <stdlib.h>                              genstring.cc
#include <arpa/inet.h>
#include <iostream>

int main(int argc, char **argv) {
  struct sockaddr_in6 sa6;          // IPv6
  char astring[INET6_ADDRSTRLEN];   // IPv6

  // IPv6 string to sockaddr_in6.
  inet_pton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr));

  // sockaddr_in6 to IPv6 string.
  inet_ntop(AF_INET6, &(sa6.sin6_addr), astring, INET6_ADDRSTRLEN);
  std::cout << astring << std::endl;

  return EXIT_SUCCESS;
}
```

DNS

People tend to use DNS names, not IP addresses

    the sockets API lets you convert between the two

    it's a complicated process, though:

        a given DNS name can have many IP addresses

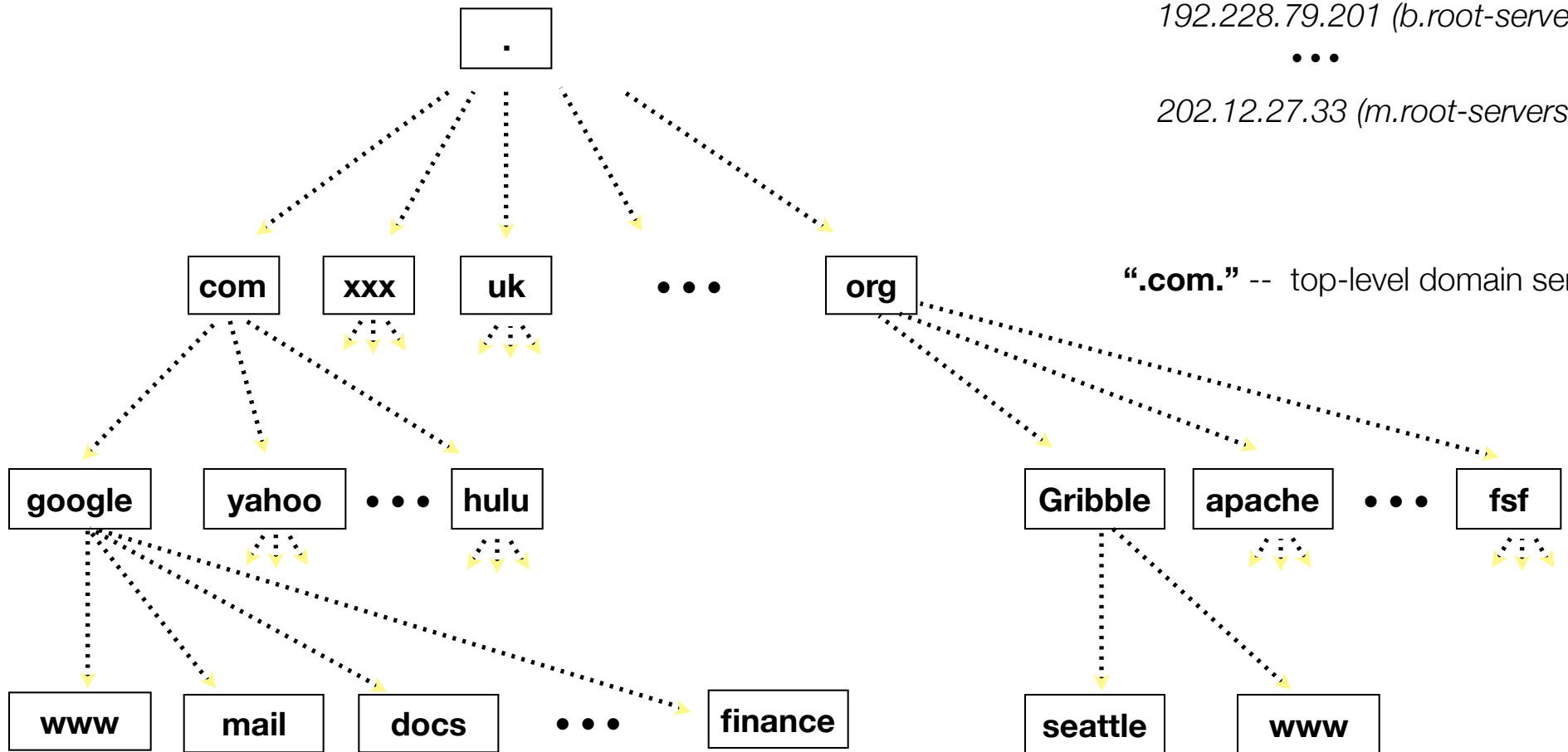   many different DNS names can map to the same IP address

            an IP address will reverse map into at most one DNS names, and maybe none

        a DNS lookup may require interacting with many DNS servers

You can use the "dig" Linux program to explore DNS

    "man dig"

DNS hierarchy

"." -- root name servers

198.41.0.4 (a.root-servers.net)
192.228.79.201 (b.root-servers.net)
• • •
202.12.27.33 (m.root-servers.net)

".com." -- top-level domain server

Resolving DNS names

The POSIX way is to use **getaddrinfo( )**

a pretty complicated system call; the basic idea…

set up a "hints" structure with constraints you want respected

e.g., IPv6, IPv4, or either

tell getaddrinfo( ) which host and port you want resolved

host: a string representation; DNS name or IP address

getaddrinfo( ) gives you a list of results packet in an "addrinfo" struct

free the addrinfo structure using freeaddrinfo( )

DNS lookup example

see dnsresolve.cc

Connecting from a client to a server.

Step 2. Create a socket.

Creating a socket

Use the **socket** system call

creating a socket doesn't yet bind it to a local address or port

```
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <iostream>

int main(int argc, char **argv) {
  int socket_fd = socket(PF_INET, SOCK_STREAM, 0);
  if (socket_fd == -1) {
    std::cerr << strerror(errno) << std::endl;
    return EXIT_FAILURE;
  }
  close(socket_fd);
  return EXIT_SUCCESS;
}
```

Connecting from a client to a server.

Step 3. Connect the socket to the remote server.

connect( )

The **connect( )** system call establishes a connection to a remote host

>> you pass the following arguments to connect( ):

>>> the socket file descriptor you created in step 2

>>> one of the address structures you created in step 1

>> connect may take some time to return

>>> it is a **blocking** call by default

>>> the network stack within the OS will communicate with the remote host to establish a TCP connection to it

>>> this involves ~2 *round trips* across the network

connect example

see connect.cc

Connecting from a client to a server.

Step 4. read( ) and write( ) data using the socket.

read( )

By default, a blocking call

if there is data that has already been received by the network stack, then read will return immediately with it

thus, read might return with less data than you asked for

if there is no data waiting for you, by default read( ) will block until some arrives

pop quiz: how might this cause **deadlock**?

write( )

By default, a blocking call

    but, in a more sneaky way

    when write( ) returns, the receiver (i.e., the other end of the connection) probably has not yet received the data

        in fact, the data might not have been sent on the network yet!

        write( ) enqueues your data in a send buffer in the OS, and then returns; the OS will transmit the data in the background

    if there is no more space left in the send buffer, by default write( ) will block

        how might this cause **deadlock**?

read/write example

see sendreceive.cc

Connecting from a client to a server.

Step 5. close( ) the socket.

See you on Friday!

Exercise 1

Write a client that:

    reads DNS names, one per line, from stdin

    translates each name to one or more IP addresses

    prints out each IP address to stdout, one per line