CSCI-351
Data communication and Networks

Lecture 4: Crash Course in C Sockets (Prepare yourself for Project 1)

The slide is built with the help of Prof. Alan Mislove, Christo Wilson, and David Choffnes's class

Socket Programming

- Goal: familiarize yourself with socket programming
  - Why am I presenting C sockets?

Socket Programming

- Goal: familiarize yourself with socket programming
  - Why am I presenting C sockets?
  - Because C sockets are the de-facto standard for networking APIs

C Sockets

- Socket API since 1983
  - Berkeley Sockets
  - BSD Sockets (debuted with BSD 4.2)
  - Unix Sockets (originally included with AT&T Unix)
  - Posix Sockets (slight modifications)
- Original interface of TCP/IP
  - All other socket APIs based on C sockets

Outline

- High-level Design
- Server API
- Client API + Name resolution
- Other Considerations

Clients and Servers

- A fundamental problem: rendezvous
  - One or more parties want to provide a service
  - One or more parties want to use the service
  - How do you get them together?

Clients and Servers

- A fundamental problem: rendezvous
  - One or more parties want to provide a service
  - One or more parties want to use the service
  - How do you get them together?
- Solution: client-server architecture
  - Client: initiator of communication
  - Server: responder
  - At least one side has to wait for the other
    - Service provider (server) sits and waits
    - Clients locates servers, initiates contact
    - Use well-known semantic names for location (DNS)

Key Differences

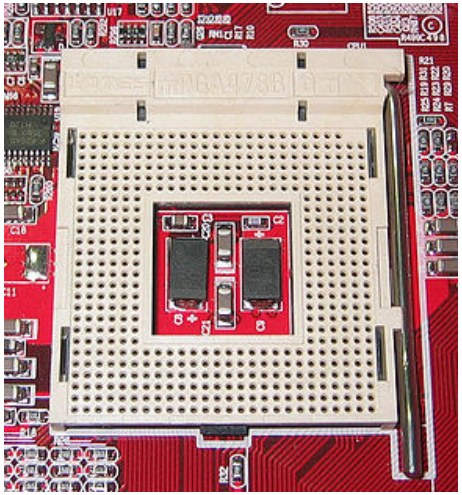
Clients	Servers
Execute on-demand	Always-on
Unprivileged	Privileged
Simple	Complex
(Usually) sequential	(Massively) concurrent
Not performance sensitive	High performance
	Scalable

Similarities

- Share common protocols
  - Application layer
  - Transport layer
  - Network layer
- Both rely on APIs for network access

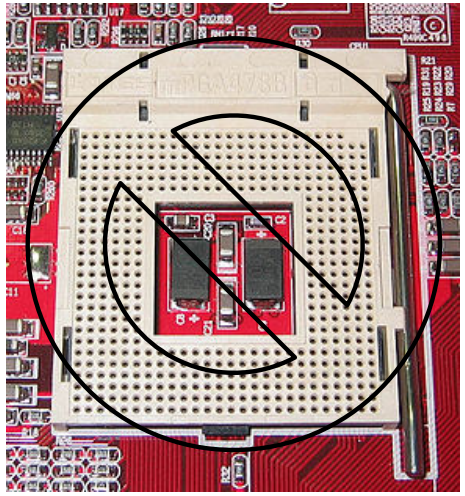
Sockets

- Basic network abstraction: the socket



Sockets

- Basic network abstraction: the socket



- Socket: an object that allows reading/writing from a network interface
- In Unix, sockets are just file descriptors
  - *read()* and *write()* both work on sockets
  - Caution: socket calls are blocking

C Socket API Overview

Clients

1. *gethostbyname()*
2. *socket()*
3. *connect()*
4. *write() / send()*
5. *read() / recv()*
6. *close()*

Servers

1. *socket()*
2. *bind()*
3. *listen()*
4. *while (whatever) {*
5. *accept()*
6. *read() / recv()*
7. *write() / send()*
8. *close()*
9. *}*
10. *close()*

C Socket API Overview

Clients

1. *gethostbyname()*
2. *socket()*
3. *connect()*
4. *write() / send()*
5. *read() / recv()*
6. *close()*

Servers

1. *socket()*
2. *bind()*
3. *listen()*
4. *while (whatever) {*
5. *accept()*
6. *read() / recv()*
7. *write() / send()*
8. *close()*
9. *}*
10. *close()*

int socket(int, int, int)

- Most basic call, used by clients and servers
- Get a new socket
- Parameters
  - int *domain*: a constant, usually *PF_INET*
  - int *type*: a constant, usually *SOCK_STREAM* or *SOCK_DGRAM*
    - *SOCK_STREAM* means TCP
    - *SOCK_DGRAM* means UDP
  - int *protocol*: usually 0 (zero)
- Return: new file descriptor, -1 on error
- Many other constants are available
  - Why so many options?

int socket(int, int, int)

- Most basic call, used by clients and servers
- Get a new socket
- Parameters
  - int *domain*: a constant, usually *PF_INET*
  - int *type*: a constant, usually *SOCK_STREAM* or *SOCK_DGRAM*
    - *SOCK_STREAM* means TCP
    - *SOCK_DGRAM* means UDP
  - int *protocol*: usually 0 (zero)
- Return: new file descriptor, -1 on error
- Many other constants are available
  - Why so many options?

The C socket API is extensible.

The Internet isn't the only network domain

TCP/UDP aren't the only transport protocols

In theory, transport protocols may have different dialects

int bind(int, struct sockaddr *, int)

- Used by servers to associate a socket to a network interface and a port
  - Why is this necessary?
- Parameters:
  - int *sockfd*: an unbound socket
  - struct sockaddr * *my_addr*: the desired IP address and port
  - int *addrlen*: *sizeof(struct sockaddr)*
- Return: 0 on success, -1 on failure
  - Why might *bind()* fail?

int bind(int, struct sockaddr *, int)

- Used by servers to associate a socket to a network interface and a port
  - Why is this necessary?
- Parameters:
  - int *sockfd*: an unbound socket
  - struct sockaddr * *my_addr*: the desired IP address and port
  - int *addrlen*: *sizeof(struct sockaddr)*
- Return: 0 on success, -1 on failure
  - Why might *bind()* fail?

Each machine may have ***multiple*** network interfaces

Example: Wifi and Ethernet in your laptop

Example: Cellular and Bluetooth in your phone

Each network interface has its own IP address

We'll talk about ports next…

int bind(int, struct sockaddr *, int)

- Used by servers to associate a socket to a network interface and a port
  - Why is this necessary?
- Parameters:
  - int *sockfd*: an unbound socket
  - struct sockaddr * *my_addr*: the desired IP address and port
  - int *addrlen*: *sizeof(struct sockaddr)*
- Return: 0 on success, -1 on failure
  - Why might *bind()* fail?

Port Numbers

Port Numbers

- Basic mechanism for multiplexing applications per host
  - 65,535 ports available
  - Why?

Port Numbers

- Basic mechanism for multiplexing applications per host
  - 65,535 ports available
  - Why?

TCP/UDP port field is 16-bits wide

Port Numbers

- Basic mechanism for multiplexing applications per host
  - 65,535 ports available
  - Why?
- Ports <1024 are reserved
  - Only privileged processes (e.g. superuser) may access
  - Why?
  - Does this cause security issues?

Port Numbers

- Basic mechanism for multiplexing applications per host
  - 65,535 ports available
  - Why?
- Ports <1024 are reserved
  - Only privileged processes (e.g. superuser) may access
  - Why?
  - Does this cause security issues?

In olden times, all important apps used low port numbers
Examples: IMAP, POP, HTTP, SSH, FTP
This rule is no longer useful

Port Numbers

- Basic mechanism for multiplexing applications per host
  - 65,535 ports available
  - Why?
- Ports <1024 are reserved
  - Only privileged processes (e.g. superuser) may access
  - Why?
  - Does this cause security issues?
- “I tried to open a port and got an error”
  - Port collision: only one app per port per host
  - Dangling sockets…

Dangling Sockets

- Common error: bind fails with “already in use” error
- OS kernel keeps sockets alive in memory after *close()*
  - Usually a one minute timeout
  - Why?

Dangling Sockets

- Common error: bind fails with “already in use” error
- OS kernel keeps sockets alive in memory after *close()*
  - Usually a one minute timeout
  - Why?

Closing a TCP socket is a multi-step process
Involves contacting the remote machine
“Hey, this connection is closing”
Remote machine must acknowledge the closing
All this book keeping takes time

Dangling Sockets

- Common error: bind fails with “already in use” error
- OS kernel keeps sockets alive in memory after *close()*
  - Usually a one minute timeout
  - Why?

Dangling Sockets

- Common error: bind fails with “already in use” error
- OS kernel keeps sockets alive in memory after *close()*
  - Usually a one minute timeout
  - Why?
- Allowing socket reuse

```
int yes=1;
if (setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int))
== -1) { perror("setsockopt"); exit(1); }
```

struct sockaddr

- Structure for storing naming information
  - But, different networks have different naming conventions
  - Example: IPv4 (32-bit addresses) vs. IPv6 (64-bit addresses)

struct sockaddr

- Structure for storing naming information
  - But, different networks have different naming conventions
  - Example: IPv4 (32-bit addresses) vs. IPv6 (64-bit addresses)
- In practice, use more specific structure implementation

```
1. struct sockaddr_in my_addr;
2. memset(&my_addr, 0, sizeof(sockaddr_in));
3. my_addr.sin_family = htons(AF_INET);
4. my_addr.sin_port = htons(MyAwesomePort);
5. my_addr.sin_addr.s_addr = inet_addr("10.12.110.57");
```

htons(), htonl(), ntohs(), ntohl()

- Little Endian vs. Big Endian
  - Not a big deal as long as data stays local
  - What about when hosts communicate over networks?

htons(), htonl(), ntohs(), ntohl()

- Little Endian vs. Big Endian
  - Not a big deal as long as data stays local
  - What about when hosts communicate over networks?
- Network byte order
  - Standardized to Big Endian
  - Be careful: x86 is Little Endian
- Functions for converting host order to network order
  - h to n s – host to network short (16 bits)
  - h to n l – host to network long (32 bits)
  - n to h * – the opposite

Binding Shortcuts

- If you don't care about the port
  - my_addr.sin_port = htons(0);
  - Chooses a free port at random
  - This is rarely the behavior you want

Binding Shortcuts

- If you don't care about the port
  - my_addr.sin_port = htons(0);
  - Chooses a free port at random
  - This is rarely the behavior you want
- If you don't care about the IP address
  - my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  - INADDR_ANY == 0
  - Meaning: don't bind to a specific IP
  - Traffic on any interface will reach the server
    - Assuming its on the right port
  - This is usually the behavior you want

int listen(int, int)

- Put a socket into listen mode
  - Used on the server side
  - Wait around for a client to *connect()*
- Parameters
  - int *sockfd*: the socket
  - int *backlog*: length of the pending connection queue
    - New connections wait around until you *accept()* them
    - Just set this to a semi-large number, e.g. 1000
- Return: 0 on success, -1 on error

int accept(int, void *, int *)

- Accept an incoming connection on a socket
- Parameters
  - int *sockfd*: the *listen()*ing socket
  - void * *addr*: pointer to an empty *struct sockaddr*
    - Clients IP address and port number go here
    - In practice, use a *struct sockaddr_in*
  - int * *addrlen*: length of the data in *addr*
    - In practice, *addrlen* == sizeof(*struct sockaddr_in*)
- Return: a new socket for the client, or -1 on error
  - Why?

int accept(int, void *, int *)

- Accept an incoming connection on a socket
- Parameters
  - int *sockfd*: the *listen()*ing socket
  - void * *addr*: pointer to an empty *struct sockaddr*
    - Clients IP address and port number go here
    - In practice, use a *struct sockaddr_in*
  - int * *addrlen*: length of the data in *addr*
    - In practice, *addrlen* == sizeof(*struct sockaddr_in*)
- Return: a new socket for the client, or -1 on error
  - Why?

You don't want to consume your *listen()* socket
Otherwise, how would you serve more clients?
Closing a client connection shouldn't close the server

close(int sockfd)

- Close a socket
  - No more sending or receiving
- *shutdown(int sockfd, int how)*
  - Partially close a socket
    - how = 0; // no more receiving
    - how = 1; // no more sending
    - how = 2; // just like *close()*
  - Note: *shutdown()* does not free the file descriptor
  - Still need to *close()* to free the file descriptor

C Socket API Overview

Clients

1. *gethostbyname()*
2. *socket()*
3. *connect()*
4. *write() / send()*
5. *read() / recv()*
6. *close()*

Servers

1. *socket()*
2. *bind()*
3. *listen()*
4. *while (whatever) {*
5. *accept()*
6. *read() / recv()*
7. *write() / send()*
8. *close()*
9. *}*
10. *close()*

struct * gethostbyname(char *)

- Returns information about a given host
- Parameters
  - const char * *name*: the domain name or IP address of a host
  - Examples: “www.google.com”, “10.137.4.61”
- Return: pointer to a *hostent* structure, 0 on failure
  - Various fields, most of which aren’t important

```
1. struct hostent * h = gethostname(“www.google.com”);
2. struct sockaddr_in my_addr;
3. memcpy(&my_addr.sin_addr.s_addr, h->h_addr,
       h->h_length);
```

int connect(int, struct sockaddr *, int)

- Connect a client socket to a *listen()*ing server socket
- Parameters
  - int *sockfd*: the client socket
  - struct sockaddr * *serv_addr*: address and port of the server
  - int *addrlen*: length of the sockaddr structure
- Return: 0 on success, -1 on failure
- Notice that we don't *bind()* the client socket
  - Why?

write() and send()

- ssize_t write(int *fd*, const void **buf*, size_t *count*);
  - *fd*: file descriptor (ie. your socket)
  - *buf*: the buffer of data to send
  - *count*: number of bytes in *buf*
  - Return: number of bytes actually written
- int send(int sockfd, const void *msg, int len, int flags);
  - First three, same as above
  - *flags*: additional options, usually 0
  - Return: number of bytes actually written
- Do not assume that *count* / *len* == the return value!
  - Why might this happen?

read() and recv()

- ssize_t read(int fd, void *buf, size_t count);
  - Fairly obvious what this does
- int recv(int sockfd, void *buf, int len, unsigned int flags);
  - Seeing a pattern yet?
- Return values:
  - -1: there was an error reading from the socket
    - Usually unrecoverable. *close()* the socket and move on
  - >0: number of bytes received
    - May be less than *count* / *len*
  - 0: the sender has closed the socket

More Resources

- Beej's famous socket tutorial
  - http://beej.us/net2/html/syscalls.html