ELEC / COMP 177 – Fall 2011

Computer Networking → Sockets API

Some slides from Kurose and Ross, *Computer Networking*, 5th Edition

Homework #3

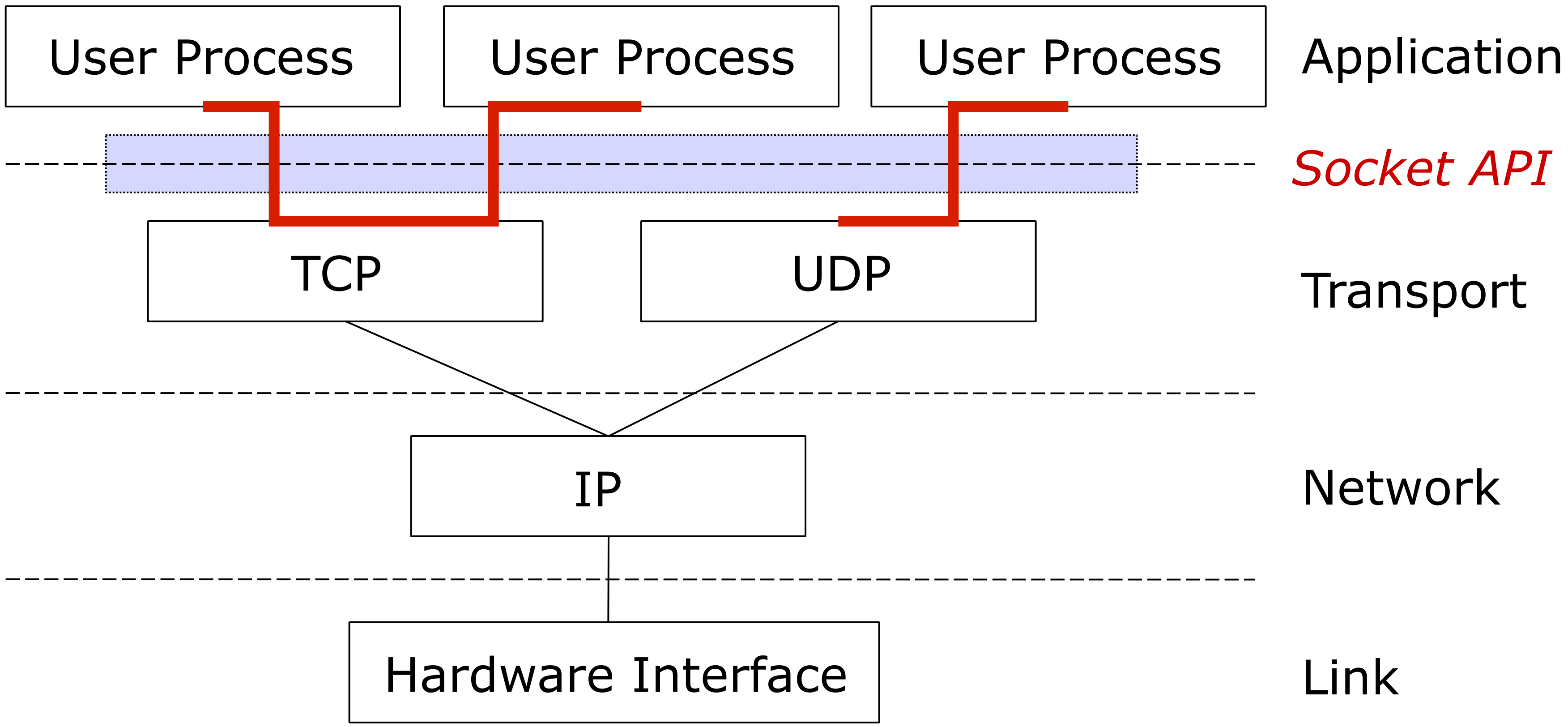
- How did it go?
- What was easy?
- What was hard?

- Are VPN accounts created yet?

The Socket API for C

- A collection of system calls to write a networking program at user-level
- API is similar to Unix file I/O in many respects: open, close, read, write.
  - Data written into socket on one host can be read out of socket on other host
  - Difference: networking has notion of client and server
- We're using the C language, but most of these terms and concepts translate directly into other languages too

Sockets and the TCP/IP Suite



Socket Parameters

- A socket connection has 5 general parameters:
  - The protocol
    - Example: TCP, UDP etc.
  - The local and remote IP address
    - Example: 171.64.64.64
  - The local and remote port number
    - Need to determine to which process packets are delivered
    - Some ports are reserved (e.g. 80 for HTTP)
    - Root access required to listen on port numbers below 1024

Reference Links

- Read this! Read this! Read this! Read this!
- Beej's Guide to Network Programming
  - http://beej.us/guide/bgnet/
  - In-depth explanations of all of the functions
  - **Complete example client and server with code**
    - Your assignment shares **much in common** with the simple stream client and server programs presented on this website.
- Read this! Read this! Read this! Read this!

Commentary

- Why are these functions and structures so ugly looking / confusing?
- Designed to be **flexible**
  - Support different forms of IP (IPv4, IPv6)
  - Support different types of application requirements (TCP, UDP, or RAW)
- Evolved over several decades (backwards compatibility)

Socket Descriptor

- Socket Descriptor: `int sockfd;`
  - Your process might have many sockets
    - Imagine a web server handling many clients simultaneously
  - Need a way to identify each of them

Socket Configuration

- How do we configure a socket?
- How do we learn about how an existing socket is configured, or who it is connected to?
- Answer: there are several key **structures** that contain all the meta-data associated with each socket

Structures in C

- **What is a structure?**
  - Grouping of separate variables

```
// Define structure
struct account {
   int account_number;
   char *first_name;
   char *last_name;
   float balance;
};

// Declare instance of the structure
struct account s;

// Access members
s.acount_number = ...
s.balance = ...
```

Struct addrinfo

- Used to prepare (configure) a socket, *plus*
- Used to store the results of a host name lookups (DNS)

```
struct addrinfo {
    int              ai_flags;     // AI_PASSIVE, AI_CANONNAME, etc.
    int              ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
    int              ai_protocol;  // use 0 for "any"
    size_t           ai_addrlen;   // size of ai_addr in bytes
    struct sockaddr *ai_addr;      // struct sockaddr_in or _in6
    char            *ai_canonname; // full canonical hostname
    struct addrinfo *ai_next;      // linked list, next node
};
```

Struct addrinfo

- `ai_flags` – Configuration Options
  - `AI_PASSIVE` – A passive socket is one that "listens" only (exactly what a server does)
    - Technically, one end of the socket will be my local IP address, and the other end will be *any* IP
  - … and many others …

```
struct addrinfo {
    int              ai_flags;     // AI_PASSIVE, AI_CANONNAME, etc.
    int              ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
    int              ai_protocol;  // use 0 for "any"
    size_t           ai_addrlen;   // size of ai_addr in bytes
    struct sockaddr *ai_addr;      // struct sockaddr_in or _in6
    char            *ai_canonname; // full canonical hostname
    struct addrinfo *ai_next;      // linked list, next node
};
```

Struct addrinfo

- `ai_family` – What version of IP to use?
  - IPv4 – `AF_INET`
  - IPv6 – `AF_INET6`
  - Don't care – `AI_UNSPEC`
    - The results of DNS will produce a IPv4 and/or IPv6 address

```
struct addrinfo {
    int              ai_flags;     // AI_PASSIVE, AI_CANONNAME, etc.
    int              ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
    int              ai_protocol;  // use 0 for "any"
    size_t           ai_addrlen;   // size of ai_addr in bytes
    struct sockaddr *ai_addr;      // struct sockaddr_in or _in6
    char            *ai_canonname; // full canonical hostname
    struct addrinfo *ai_next;      // linked list, next node
};
```

Struct addrinfo

- `ai_socktype` – TCP or UDP?
  - `SOCK_STREAM` – TCP sockets (**streaming**)
  - `SOCK_DGRAM` – UDP sockets (**datagrams**)
  - `SOCK_RAW` – No transport layer (controls exactly what the NIC sends). Neat but hard!

```
struct addrinfo {
    int              ai_flags;     // AI_PASSIVE, AI_CANONNAME, etc.
    int              ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
    int              ai_protocol;  // use 0 for "any"
    size_t           ai_addrlen;   // size of ai_addr in bytes
    struct sockaddr *ai_addr;      // struct sockaddr_in or _in6
    char            *ai_canonname; // full canonical hostname
    struct addrinfo *ai_next;      // linked list, next node
};
```

Struct addrinfo

- `ai_protocol` – Limits incoming sockets to a specific protocol
  - `0` – Any protocol (still limited by `SOCK_STREAM`)
  - `IPPROTO_TCP`, `IPPROTO_UDP` – Not often used

```
struct addrinfo {
    int              ai_flags;     // AI_PASSIVE, AI_CANONNAME, etc.
    int              ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
    int              ai_protocol;  // use 0 for "any"
    size_t           ai_addrlen;   // size of ai_addr in bytes
    struct sockaddr *ai_addr;      // struct sockaddr_in or _in6
    char            *ai_canonname; // full canonical hostname
    struct addrinfo *ai_next;      // linked list, next node
};
```

Struct addrinfo

- `ai_addrlen` – Size of `ai_addr` in bytes
  - `ai_addr` described next…

```
struct addrinfo {
    int              ai_flags;     // AI_PASSIVE, AI_CANONNAME, etc.
    int              ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
    int              ai_protocol;  // use 0 for "any"
    size_t           ai_addrlen;   // size of ai_addr in bytes
    struct sockaddr *ai_addr;      // struct sockaddr_in or _in6
    char            *ai_canonname; // full canonical hostname
    struct addrinfo *ai_next;      // linked list, next node
};
```

Struct addrinfo

- `ai_addr` – A pointer to a structure that contains *more* socket address info
  - Specifically details like port number, IP address (v4 or v6), etc…

```
struct addrinfo {
    int              ai_flags;     // AI_PASSIVE, AI_CANONNAME, etc.
    int              ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
    int              ai_protocol;  // use 0 for "any"
    size_t           ai_addrlen;   // size of ai_addr in bytes
    struct sockaddr *ai_addr;      // struct sockaddr_in or _in6
    char            *ai_canonname; // full canonical hostname
    struct addrinfo *ai_next;      // linked list, next node
};
```

Struct addrinfo

- `ai_cannonname`
  - The "true" DNS name ("canonical name")
  - DNS can have alias to other DNS entries
  - Usually NULL unless requested

```
struct addrinfo {
    int              ai_flags;     // AI_PASSIVE, AI_CANONNAME, etc.
    int              ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
    int              ai_protocol;  // use 0 for "any"
    size_t           ai_addrlen;   // size of ai_addr in bytes
    struct sockaddr *ai_addr;      // struct sockaddr_in or _in6
    char            *ai_canonname; // full canonical hostname
    struct addrinfo *ai_next;      // linked list, next node
};
```

Struct addrinfo

- `ai_next` – Pointer to the next struct `addrinfo`
  - We can have a linked-list of many instances of this structure, all in a row…
  - Useful in case DNS returns both IPv4 and IPv6

```
struct addrinfo {
    int              ai_flags;     // AI_PASSIVE, AI_CANONNAME, etc.
    int              ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
    int              ai_protocol;  // use 0 for "any”
    size_t           ai_addrlen;   // size of ai_addr in bytes
    struct sockaddr *ai_addr;      // struct sockaddr_in or _in6
    char            *ai_canonname; // full canonical hostname
    struct addrinfo *ai_next;      // linked list, next node
};
```

Struct sockaddr

- Description of the socket
  - IP address?
  - Port?
  - Other details
- Generic type: sockaddr

```
struct sockaddr {
   unsigned short    sa_family;    // address family, AF_xxx
   char              sa_data[14];  // 14 bytes of protocol address
};
```

Struct sockaddr

- sockaddr is generic
  - Handles IP and other obscure protocols
  - You can cast it to an IPv4-specific structure to easily access the underlying fields
  - **What is casting?**

```
struct sockaddr_in {
    short int          sin_family;  // Address family, AF_INET
    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; // that's a 32-bit int (4 bytes)
};
```

Struct sockaddr

- `sockaddr_storage` is generic for both IPv4 and IPv6
  - It's big enough to hold either
  - It uses a consistent format
  - You can cast it to either the IPv4 or v6 type to easily access the underlying fields (and just ignore extra padding at the end that makes it generic)

Struct sockaddr

```
struct sockaddr_in6 {
    u_int16_t       sin6_family;   // address family, AF_INET6
    u_int16_t       sin6_port;     // port number, Network Byte Order
    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
};
```

Program Operation

- Ok, so we have (boring) structures
  - **How do we use them to create a socket?**
- Basic setup (for client or server)
  - Populate the structure with your socket settings
  - Call a function with the IP/port you want to connect to or listen on
  - That function produces a new output structure with all the right fields filled in
  - Then make a bunch of function calls

Sockets for Servers

Server Program Operation

- Let's take a simple **connection-oriented (TCP)** **<u>server</u>** first

1. `socket()` create the socket descriptor
2. `bind()` associate the local address
3. `listen()` wait for incoming connections from clients
4. `accept()` accept incoming connection
5. `send(),recv()` communicate with client
6. `close()` close the socket descriptor

Program Operation

- Helper variables
  - `int status;` // Test this to detect errors!
- Declare variable for input structure
  - `struct addrinfo hints;`
- Declare pointer to resulting structure
  - `struct addrinfo *res;`
- Make sure the structure is empty
  - `memset(&hints, 0, sizeof hints);`

Server Program Operation

- Populate fields with **server** settings
  - Don't care if IPv4 or IPv6
    - `hints.ai_family = AF_UNSPEC;`
  - TCP streaming sockets
    - `hints.ai_socktype = SOCK_STREAM;`
  - Fill in my IP (to listen on)
    - `hints.ai_flags = AI_PASSIVE;`

Server – getaddrinfo()

- Multi-purpose function
- Inputs:
  - A hostname (i.e. www.google.com)
  - A service name (i.e. HTTP) or port number
  - Your "hints" structure of desired configuration
- Output:
  - IP address (via DNS)
  - Port number (by looking up in a local config file)
  - A pointer to a fully-populated addrinfo structure

Server – getaddrinfo()

- Function prototype
  - ```
    int getaddrinfo(
      const char *node, // e.g. "www.example.com" or IP
      const char *service,  // e.g. "http" or port number
      const struct addrinfo *hints,
      struct addrinfo **res);
    ```
- Function call for server
  - ```
    status = getaddrinfo(
             NULL, "3490”, &hints, &res);
    ```
    - NULL: Fill in my IP (via AI_PASSIVE)
    - “3490” or any other valid port number

Server – getaddrinfo()

- A non-zero result indicates an error
  - You should notify the user!

- ```
  if (status != 0) {
      fprintf(stderr,
       "getaddrinfo error: %s\n",
       gai_strerror(status));

      exit(1);
  }
  ```
  ```

Server – getaddrinfo()

- What have we created with this function call?
  - A fully populated addrinfo structure with all the socket configuration info (with IP addresses and port numbers, not human-friendly host names or service names)
- We have **not** actually created a socket, or listened on a port, or sent/received any data

Server – socket()

- Let’s create the **server** socket now!
- Function prototype
  - `int socket(int domain, int type, int protocol);`
- Function call (using values from the structure we just created)
  - ```
    int sockfd;   // Store socket descriptor here!
    sockfd = socket(res->ai_family,
             res->ai_socktype,
             res->ai_protocol);
    ```
- socket() returns -1 for error and updates `errno`
    ```

Server – socket()

- What good is this socket descriptor?
- By itself, it does nothing
- But we can use it as the basis for future system calls

Sever – bind()

- `bind()` associates the server socket with a specific port on the local machine
  - The port specified in `addrinfo` structure
- Function prototype
  - ```
    int bind(int sockfd,
        struct sockaddr *my_addr,
        int addrlen);
    ```
- Function call
  - ```
    status = bind(sockfd,
        res->ai_addr,
        res->ai_addrlen);
    ```
- `bind()` returns -1 for error and updates `errno`

Server – listen()

- `listen()` listens for incoming messages on the socket
- Function prototype
  - `int listen(int sockfd, int backlog);`
  - backlog is number of incoming connections on queue (probably limited by OS to ~20)
- Function call
  - `status = listen(sockfd, 10);`
- listen() returns -1 for error and updates `errno`

Server – accept()

- `accept()` acknowledges an incoming connection
- Function prototype
  - ```
    int accept(int sockfd,
        struct sockaddr *addr,
        socklen_t *addrlen);
    ```
- Function call
  - ```
    int sockfd_client;
    struct sockaddr_storage their_addr;
    socklen_t addr_size;
    addr_size = sizeof their_addr;
    sockfd_client = accept(sockfd,
        (struct sockaddr *)&their_addr,
        &addr_size);
    ```
- `accept()` returns -1 for error and updates `errno`

Server – accept()

- Wait, what is happening here?
- I give `accept()`:
  - The socket descriptor for the server
  - A pointer to an empty `sockaddr_storage` structure
    - Generic for either IPv4 or IPv6
  - The size of that empty structure
- `accept()` runs and gives me
  - A **new** socket descriptor that connects to the client
  - A populated `sockaddr_storage` structure with details on the incoming socket (the IP and port of host that is connecting to me)
  - The size of the populated structure

Server Operation

- The socket returned by `accept()` is not the same socket that the server was listening on!
- A new socket, bound to a random port, is created to handle the connection
- New socket should be closed when done with communication
- Initial socket remains open, can still accept more connections
  - The initial socket never does any application-level communication. It just serves to generate new sockets

Server Recap Thus Far

- Someone from far far away will try to `connect()` to your machine on a port that you are `listen()`ing on.
- Their connection will be queued up waiting to be `accept()`ed
- You call `accept()` and you tell it to get the pending connection
- `accept()` will return to you a brand new socket file descriptor to use for this single connection!
- You now have two socket file descriptors for the price of one!
  - The original one is still listening for more new connections
  - The newly created one is finally ready to `send()` and `recv()`

send() and recv()

- Send and receive data on connected, streaming sockets (i.e. TCP)
  - We have different functions for unconnected / UDP sockets: `sendto()` and `recvfrom()`
- Function prototypes
  - ```
    int send(int sockfd,
        const void *msg, int len, int flags);
    ```
    - msg is the data you want to send
    - len is the length of the data
  - ```
    int recv(int sockfd, void *buf,
                 int len, int flags);
    ```
    - buf is where you want the data to be copied to
    - len is the maximum length of the buffer data

send() and recv()

- Send example:
  - ```
    char *msg = "Test Message";
    int len, bytes_sent;
    len = strlen(msg);
    bytes_sent =
     send(sockfd_client, msg, len, 0);
    ```
- Receive example:
  - ```
    char buf[500];
    int bytes_recvd;
    bytes_recvd =
       recv(sockfd_client, buf, 500-1, 0);
    ```

Pitsfalls

- send() and recv() are stream-oriented
  - Your messages are not independent, they're part of the first-in, first-out stream
- send() and recv() may only **partially succeed**
  - send() might only send 256 out of 512 bytes you requested
  - recv() might only fill your 4kB buffer with 1kB of data
- **You** (the poor, overworked programmer) are responsible for repeatedly calling send() and recv() until **all** your data is transferred
  - *For a partial solution, see Beej's guide section 7.3*

close()

- We're finished
- Function prototype:
  - `close(sockfd);`
- Don't forget to also deallocate the linked list generated by getaddrinfo() in the first step
  - `freeaddrinfo(res);`

Recap of Day 1

Server Functions – Recap

- **What does `socket()` do?**
  - Create the socket descriptor
- **What does `bind()` do?**
  - Assigns a local address/port to the socket
- **What does `listen()` do?**
  - Configures socket to accept incoming connections
- **What does `accept()` do?**
  - Accepts incoming connection (will block until connection)
- **What do `send()/recv()` do?**
  - Communicate with client
- **What does `close()` do?**
  - Close the socket descriptor

Return Values – Recap

- **Why do we have to check return values / error codes for every single socket function?**

Send/Recv Pitfalls - Recap

- Common pitfalls with TCP sockets
- **What is happening in these scenarios?**
  - "My client program sent 100 bytes, but the server program only got 50."
  - "My client program sent several small packets, but the server program received one large packet."
- Ans: TCP is a **stream protocol**
  - The sender or receiver (or both!) can segment and recombine the stream at arbitrary locations

From: http://tangentsoft.net/wskfaq/articles/effective-tcp.html (*Good article to read!*)

Send/Recv Pitfalls - Recap

- Common pitfalls with TCP sockets
- "How can I find out how many bytes are waiting on a given socket, so I can set up a receive buffer for the size of the packet?"
  - **You don't!** Declare a reasonable fixed size buffer when your program starts (say, 32kB) and always receive data *into* that buffer
  - Then, **copy data out of your buffer** into the rest of your program as needed

From: http://tangentsoft.net/wskfaq/articles/effective-tcp.html (*Good article to read!*)

Sockets for Clients

Client Program Operation

- Let's look at a simple **connection-oriented (TCP) client** now
  - We don't need bind(), listen(), or accept()!

1. `socket()` create the socket descriptor
2. `connect()` connect to the remote server.
3. `send()`,`recv()` communicate with the server
4. `close()` end communication by closing socket descriptor

Client Program Operation

- Helper variables
  - `int status;` // Test this to detect errors!
- Declare variable for input structure
  - `struct addrinfo hints;`
- Declare pointer to resulting structure
  - `struct addrinfo *res;`
- Make sure the structure is empty
  - `memset(&hints, 0, sizeof hints);`

Client Program Operation

- Populate fields with **client** settings
  - Don't care if IPv4 or IPv6
    - `hints.ai_family = AF_UNSPEC;`
  - TCP streaming sockets
    - `hints.ai_socktype = SOCK_STREAM;`

Client – socket()

- A client can use socket() just like a server does to create a new socket
- Slightly different setup structure, though
  - ```
    getaddrinfo("www.example.com",
        "3490", &hints, &res);
    ```
    - Specify the **hostname** *(or IP)* and **port** *(or service type)* of the **remote machine** to connect to
    ```

Client – connect()

- Now that we have a socket on the client, connect that socket to a remote system (where a server is listening…)
- Function prototype
  - ```
    int connect(int sockfd,
        struct sockaddr *serv_addr,
        int addrlen);
    ```
- Function call
  - ```
    connect(sockfd,
      res->ai_addr, res->ai_addrlen);
    ```
- connect() returns -1 for error and updates `errno`

Client – send()/recv()/close()

- After that, it’s all the same
  - **send()** data
  - **recv()** data
  - **close()** the socket when finished

Related Programming Topics

What is errno?

- Common variable used by many system calls
  - Thread-safe, "global" (ish)
- Holds a code representing what error has occurred
  - A list of all possible errors is in errno.h
- Pre-built functions will decode `errno`
  - ```
    printf("An error has occurred: %s\n",
              strerror(errno));
    ```
- Tip! Mixing these two decoders up will produce **wrong error messages** (which are worse then none at all!)
  - Use **gai_strerror** only when decoding status of **getaddrinfo**
  - Use **strerror** when decoding status of all other functions
    ```

Other Useful Functions

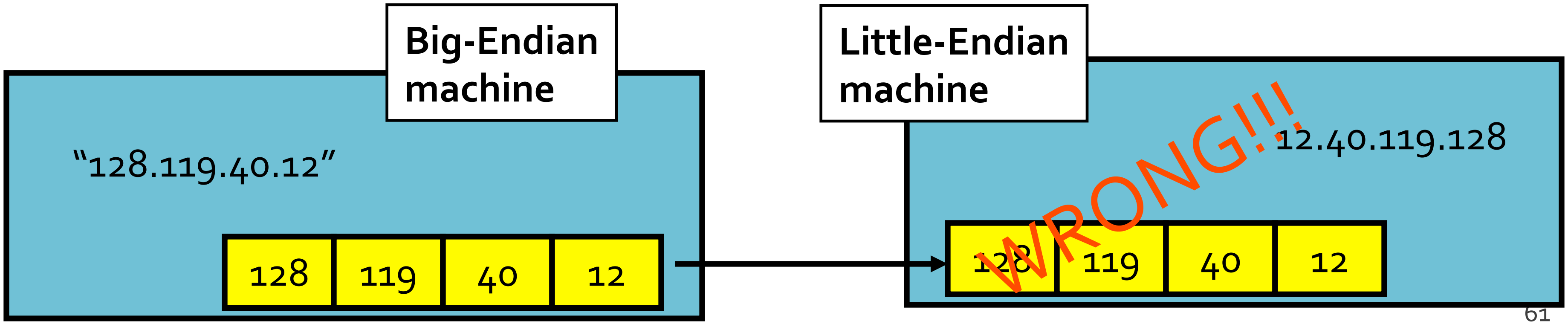
- Address conversion routines
  - Convert between system's representation of IP addresses and readable strings (e.g. "171.64.64.64")
  - Look up `inet_ntop()` or `inet_pton()`
    - p = "printable"
    - n = "network"
- `getpeername()` - Who are you?
- `gethostname()` - Who am I?

Libraries to Include

- I needed to include the following libraries when writing socket programs
  - `#include <stdio.h>`
  - `#include <stdlib.h>`
  - `#include <string.h>`
  - `#include <unistd.h>`
  - `#include <signal.h>`
  - `#include <errno.h>`
  - `#include <sys/types.h>`
  - `#include <sys/socket.h>`
  - `#include <netdb.h>`

Address and port byte-ordering

- Address and port are stored as integers
  - u_short sin_port; (16 bit)
  - in_addr sin_addr; (32 bit)
- Problem:
  - Different machines / OS's order bytes differently in a word!
    - Little-endian: lower bytes come first (stored in lower memory addresses)
    - Big-endian: higher bytes come first
  - These machines may communicate with one another over the network



Solution: Network Byte-Ordering

- Host **Byte-Ordering**
  - The byte ordering used by a host (big or little)
- Network **Byte-Ordering**
  - The byte ordering used by the network
  - Always **big-endian**
- Any words sent through the network should be converted to *network byte order* prior to transmission (and back to *host byte order* once received)

Network Byte-Ordering

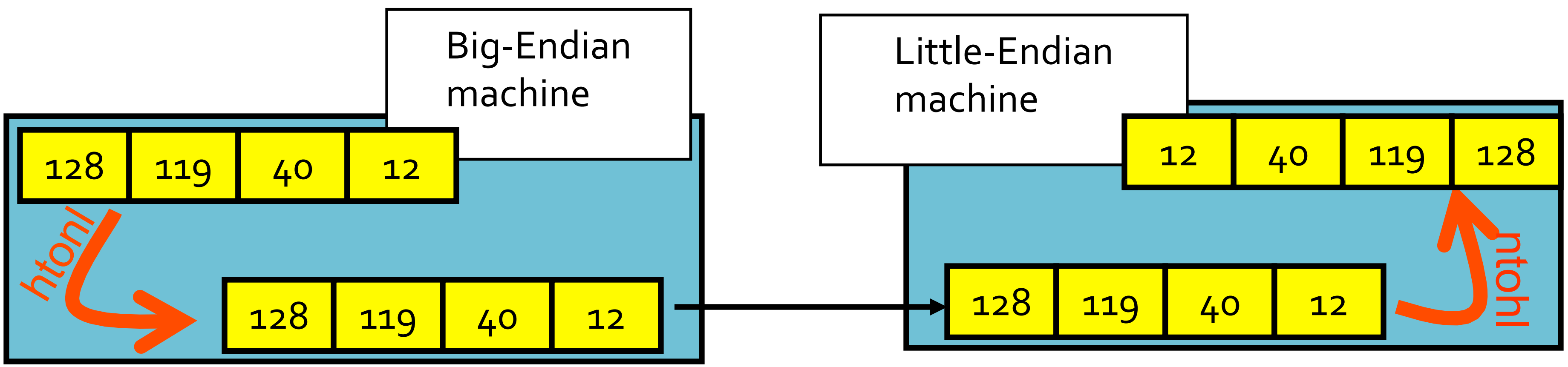
- **Should the socket perform the endianness conversion automatically?**
  - **No –** Not all data needs to be flipped
  - Imagine a stream of characters…
- **Given big-endian machines don't need conversion routines and little-endian machines do, how do we avoid writing two versions of code?**

UNIX Byte-ordering Functions

```
uint32_t htonl(uint32_t x);
uint32_t ntohl(uint32_t x);
```

```
uint16_t htons(u_short x);
uint16_t ntohs(uint16_t x);
```

- On big-endian machines, these routines do nothing!
- On little-endian machines, they reverse the byte order



- Same code will work regardless of endian-ness of the two machines

UNIX Byte-ordering Functions

- **htonl**
  - **H**ost **to N**etwork **O**rder – **L**ong (32 bits)
- **htons**
  - **H**ost to **N**etwork **O**rder – **S**hort (16 bits)

Structures and Packets

- Goal: We want to build up structures in memory that match our packet (protocol) format

- Example *(for purposes of discussion)*: **Ethernet header**
  - 48-bit destination MAC
  - 48-bit source MAC
  - 16-bit type/length field

Structures and Packets

- You might create a *structure* to make it easy to access each individual field

```
struct ethernet_header
{
    uint8_t  ether_dhost[6];
    uint8_t  ether_shost[6];
    uint16_t ether_type;
}
```

- **How many bytes do you think this takes in memory?**
  - Who knows! *(The perils of the optimizing compiler!)*

Structures and Packets

- *Think back to ECPE 170…*
  - *Think about CPU cache*
  - *Think about accessing memory…*
- It is generally more efficient for the CPU to access data that is **aligned**
  - Perhaps on word **boundaries**…
  - Perhaps on cache line boundaries…
- 48 bits (6 bytes) is not a word boundary on a modern 64 bit (8 byte) CPU

Structures and Packets

- The compiler may "optimize" your network structure like this!

```
struct ethernet_header
{
    uint8_t  ether_dhost[6];
    uint8_t  unused_space[2];
    uint8_t  ether_shost[6];
    uint8_t  unused_space[2];
    uint16_t ether_type;
    uint8_t  unused_space[6];
}

// Next item in memory starts here...
```

Structures and Packets

- Solution? **Tell the compiler** to **stop** doing that!

```
struct ethernet_header
{
   uint8_t  ether_dhost[6];
   uint8_t  ether_shost[6];
   uint16_t ether_type;
} __attribute__ ((packed)) ;
```

- See: http://gcc.gnu.org/onlinedocs/gcc-3.2.3/gcc/Type-Attributes.html

Using Structures

- Casting is your friend here!
- Say somebody gives you a pointer to a buffer in memory containing a packet
  - You get `uint8_t *packet`
- Accessing that data byte-by-byte is tedious!
- Solution?
  - Make a new pointer to your fancy structure
    - `struct ethernet_header *eth;`
  - Assign the old pointers to the new pointer via a *cast*
    - `eth = (struct ethernet_header *) packet;`

Using Structures

- Now you can use the new *pointer to a structure* to easily access individual parts of the memory buffer
  - eth->ether_type
  - eth->ether_dhost[0]
  - eth->ether_dhost[1]
  - …

Pitfalls to Discuss Later

- Other issues beyond the scope of the *first* programming exercise
  - Handling of partial sends() / recvs()
  - Multi-threaded applications (say, one thread per socket/client)

Reference Links

- Read this! Read this! Read this! Read this!
- Beej's Guide to Network Programming
  - http://beej.us/guide/bgnet/
  - In-depth explanations of all of the functions
  - **Complete example client and server with code**
    - Your assignment shares **much in common** with the simple stream client and server programs presented on this website.
- Read this! Read this! Read this! Read this!

Project #1 Discussion

- Start work today!
- Due: Oct 20th (2.5 weeks)
- Requirements