CSE 333
Section 8
Client-Side Networking
Logistics

Friday, Feb 26 (tomorrow):
HW3 @ 12pm
Computer Networks: layers of abstraction

- How to connect computers by having hosts/processes communicates?
- A high-level requirement with lots of specifics involved
  - How to physically send data (in bits/bytes)?
  - structure and semantics of data
  - identification of hosts
  - etc.
- Application programmers don’t want to deal with all these.
- The common way to build system is through layers of abstractions
  - Each layer implements a part of the problem and provides an interface for the higher layers
  - Decomposes the problem and brings modularity (e.g. many types of services at same layer depending on requirement.)

(adapted from https://book.systemsapproach.org/foundation/architecture.html)
Computer Networks: layers of abstraction

- In networking context, abstract objects making up the layers are called protocols.
- A protocol specifies:
  - a service interface to high-level protocol/object (i.e. set of operations they can use)
  - a peer interface for syntax and semantics of messages between peers of the same protocol
- What would be a good set of layers offering useful service while being efficient?

(adapted from https://book.systemsapproach.org/foundation/architecture.html)
We introduce the 7-Layer OSI architecture, skipping presentation and session layers.
Modern Internet is based on the Internet Architecture, but layers map well to the OSI model.

(adapted from https://book.systemsapproach.org/foundation/architecture.html)
Computer Networks: A 7-ish Layer Cake
Computer Networks: A 7-ish Layer Cake

- Transmit signal through physical medium
- Bits from high/low voltage, frequency, etc.

bit encoding at signal level
Computer Networks: A 7-ish Layer Cake

- Specifies communication with other nodes on a link
- “Packetized” stream of bits into frames

implemented by network adaptors and device drivers

multiple computers on a local network

bit encoding at signal level
Computer Networks: A 7-ish Layer Cake

- Interconnect different network types
- Routers implements up to this layer
- IP packets within payload of data link’s packet (frame)

Routing of packets across networks

Multiple computers on a local network

Bit encoding at signal level
Computer Networks: A 7-ish Layer Cake

- Runs only on end hosts
- Provides process-to-process abstraction
- Again, packets nested inside payload of IP packets

<table>
<thead>
<tr>
<th></th>
<th>UDP</th>
<th>TCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>process-to-process abstraction!</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Sending data end-to-end
- Routing of packets across networks
- Multiple computers on a local network
- Bit encoding at signal level
Computer Networks: A 7-ish Layer Cake

- Format/meaning of high-level messages
- Sending data end-to-end
- Routing of packets across networks
- Multiple computers on a local network
- Bit encoding at signal level

HTTP
DNS

- Application
- Presentation
- Session
- Transport
- Network
- Data link
- Physical
Packet encapsulation

HTTP header | HTTP payload (e.g. chunk of HTML page)
---|---
TCP header | TCP payload
IP header | IP payload
destination address | source address
data
ethernet header | ethernet payload
Data flow
Exercise 1
Exercise 1

- format/meaning of messages
- sending data end-to-end
- routing of packets across networks
- multiple computers on a local network
- bit encoding at signal level
Exercise 1

- **DNS:** Translating between IP addresses and host names. (Application Layer)
- **IP:** Routing packets across the Internet. (Network Layer)
- **TCP:** Reliable, stream-based networking on top of IP. (Transport Layer)
- **UDP:** Unreliable, packet-based networking on top of IP. (Transport Layer)
- **HTTP:** Sending websites and data over the Internet. (Application Layer)
TCP versus UDP

Transmission Control Protocol (TCP)
- Connection oriented Service
- Reliable and Ordered
- Flow control

User Datagram Protocol (UDP)
- Connectionless service
- Unreliable packet delivery
- Faster
- No feedback
Sockets

- Just a file descriptor for network communication
  - processes communicate with the outside through I/O operations
  - sockets API enables access to the TCP/UDP transport protocol
  - transport protocol provides abstraction of processes over network

- Types of Sockets
  - Stream sockets (TCP)
  - Datagram sockets (UDP)

- Each socket is associated with a **port number** and an **IP address**
  - Both port and address are stored in network byte order (big endian)

```c
struct sockaddr_in:

<table>
<thead>
<tr>
<th>family</th>
<th>port</th>
<th>addr</th>
<th>zero</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

struct sockaddr_in6:

<table>
<thead>
<tr>
<th>fam</th>
<th>port</th>
<th>flow</th>
<th>addr</th>
<th>scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>24</td>
</tr>
</tbody>
</table>
```

18
# File Descriptor Table

![Diagram of file descriptors](image)

### OS’s File Descriptor Table for the Process

<table>
<thead>
<tr>
<th>File Descriptor</th>
<th>Type</th>
<th>Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>pipe</td>
<td>stdin (console)</td>
</tr>
<tr>
<td>1</td>
<td>pipe</td>
<td>stdout (console)</td>
</tr>
<tr>
<td>2</td>
<td>pipe</td>
<td>stderr (console)</td>
</tr>
<tr>
<td>3</td>
<td>TCP socket</td>
<td>local: 128.95.4.33:80 remote: 44.1.19.32:7113</td>
</tr>
<tr>
<td>5</td>
<td>file</td>
<td>index.html</td>
</tr>
<tr>
<td>8</td>
<td>file</td>
<td>pic.png</td>
</tr>
<tr>
<td>9</td>
<td>TCP socket</td>
<td>local: 128.95.4.33:80 remote: 102.12.3.4:5544</td>
</tr>
</tbody>
</table>
Sockets/Address

**struct sockaddr** (pointer to this struct is used as parameter type in system calls)

| fam | ????
|-----|--------

**struct sockaddr_in** (IPv4)

<table>
<thead>
<tr>
<th>fam</th>
<th>port</th>
<th>addr</th>
<th>zero</th>
</tr>
</thead>
</table>

| 16 |

**struct sockaddr_in6** (IPv6)

<table>
<thead>
<tr>
<th>fam</th>
<th>port</th>
<th>flow</th>
<th>addr</th>
<th>scope</th>
</tr>
</thead>
</table>

Big enough to hold either
Byte Ordering and Endianness

- **Network Byte Order (Big Endian)**
  - The most significant byte is stored in the highest address

- **Host byte order**
  - Might be big or little endian, depending on the hardware

- To convert between orderings, we can use
  - `uint16_t htons (uint16_t hostlong);`
  - `uint16_t ntohs (uint16_t hostlong);`
  - `uint32_t htonl (uint32_t hostlong);`
  - `uint32_t ntohl (uint32_t hostlong);`
Exercise 2
client-side example

Figure out what IP address and port to talk to. (getaddrinfo())

Create a socket. (socket())

Connect to the server. (connect())

Transfer data through the socket. (read() and write()) (i.e. I/O with the peer process to implement your application!)

Close the socket when done. (close())
1.

specify lookup hints

(\text{hostname, servname, \_\_, \_\_})
1. **getaddrinfo()**

- Performs a **DNS Lookup** for a hostname

```
int getaddrinfo(const char *hostname,
                const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);
```
1. **getaddrinfo()**

- Performs a **DNS Lookup** for a hostname
- Use “hints” to specify constraints (**struct addrinfo** *)
- Get back a linked list of **struct addrinfo** results

```c
int getaddrinfo(const char *hostname,
                const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);
```
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
DNS – Domain Name System/Service

- A hierarchical distributed naming system any resource connected to the Internet or a private network.

- Resolves queries for names into IP addresses.

- The sockets API lets you convert between the two.
  - Aside: getnameinfo() is the inverse of getaddrinfo()

- Is on the application layer on the Internet protocol suite.

- POSIX form of resolving DNS names is getaddrinfo()
  - dig +trace attu.cs.washington.edu shown later
1. `getaddrinfo()` - Interpreting Results

```c
struct addrinfo {
    int ai_flags; // additional flags
    int ai_family; // AF_INET, AF_INET6, AF_UNSPEC
    int ai_socktype; // SOCK_STREAM, SOCK_DGRAM, 0
    int ai_protocol; // IPPROTO_TCP, IPPROTO_UDP, 0
    size_t ai_addrlen; // length of socket addr in bytes
    struct sockaddr* ai_addr; // pointer to socket addr
    char* ai_canonname; // canonical name
    struct addrinfo* ai_next; // can form a linked list
};
```

- `ai_addr` points to a `struct sockaddr` describing the socket address
1. getaddrinfo() - Interpreting Results

With a `struct sockaddr*`:

- The field `sa_family` describes if it is IPv4 or IPv6
- Cast to `struct sockaddr_in*` (v4) or `struct sockaddr_in6*` (v6) to access/modify specific fields
- Store results in a `struct sockaddr_storage` to have a space big enough for either

```
    extract fields from result
    (IPv4 vs IPv6)
```

```
    struct addrinfo*
```
2. extract fields from result (IPv4 vs IPv6)

\[ \text{_______} \]

\[ \text{_______}(\text{____, type, protocol}) \]
2. **socket()**

- Creates a “raw” socket, ready to be bound
- Returns file descriptor (sockfd) on success, -1 on failure

```c
int socket(int domain,       // AF_INET, AF_INET6
           int type,         // SOCK_STREAM (TCP)
           int protocol);   // 0
```
3.

extract fields from result (IPv4 vs IPv6)

int (sa_family)

2

(____, type, protocol)

int (sockfd)

3

(____, ______, addrlen)
3. connect()

- Connects an available socket to a specified address
- Returns 0 on success, -1 on failure

```
int connect (int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
```
3. connect()

- Connects an available socket to a specified address
- Returns 0 on success, -1 on failure

```c
int connect (int sockfd,       // from 2
             const struct sockaddr *serv_addr, // from 1
             socklen_t addrlen);      // size of serv_addr
```
4. read/write and 5. close

- Thanks to the file descriptor abstraction, use as normal!
- read from and write to a buffer, the OS will take care of sending/receiving data across the network
- Make sure to close the fd afterward