Networking:
IPv4 - AF_INET

Adapted by Joseph Lunderville
from slides by Dr. Brian Fraser
and course material by Dr. Steve Ko

Topics

- How can we use sockets on a network (AF_INET)?
- How do different computer architectures affect data formats?

AF_INET
Data Structure

AF_INET and AF_INET6

- *IP Addresses*
  - IPv4: `AF_INET` uses 4 bytes for IP addresses
    - Written in “dotted-quad”, e.g. 192.168.7.2
  - IPv6: `AF_INET6` uses 16 bytes for IP addresses
    - Written a 8 hex groups separated by colons, e.g. 2F10:C203:A135:DC3F:35:6F2:1:F603
  - More info:
    `man 7 ip`
    `man 7 ipv6`
  - We’ll focus on `AF_INET`
- *AF_INET addresses*
  - `struct sockaddr_in` shown:
  - “_in” for INternet

```
struct in_addr {
    in_addr_t s_addr;
};

struct sockaddr_in {
    sa_family_t     sin_family;
    in_port_t       sin_port;
    struct in_addr  sin_addr;
    unsigned char   __pad[X];
}
```

sockaddr_in.sin_addr

- *Binary format*
  - Humans write IPv4 addresses as “192.168.7.1”
  - Computer represents as 4-byte value, either array of uint8_t, or single uint32_t
- *Convert address*
  - inet_pton() e.g. “192.168.0.1” → C0 A8 00 01 *(presentation to network)*

    Shows up as 0x0100A8C0 on a little-endian machine: now you see why we were drilling you on this earlier!
  - inet_ntop() e.g. C0 A8 00 01 → “192.168.0.1” (network to presentation)
  - These handle both IPv4 and IPv6
- *Presentation String Lengths*
  - Max string length in `<netinet/in.h>` (but `inet_*to*()` are in `<arpa/inet.h>`!?)
  - IPv4: `INET_ADDRSTRLEN`
    IPv6: `INET6_ADDRSTRLEN`

```
struct in_addr {
    in_addr_t s_addr;
};

struct sockaddr_in {
    sa_family_t     sin_family;
    in_port_t       sin_port;
    struct in_addr  sin_addr;
    unsigned char   __pad[X];
}
```

sin_addr - Special addresses

- loopback address: 127.0.0.1

```
sin_addr.s_addr = INADDR_LOOPBACK;
```

  - Local communication, similar to UNIX domain sockets
  - Packets stay within the OS: does not reach hardware

- Wildcard address

```
sin_addr.s_addr = INADDR_ANY;
```

  - A machine can have multiple interfaces
  - e.g., wireless and wired Ethernet
  - e.g., multihomed configuration...
  - Use with `bind()`
  - listens on every address on this machine

```
struct in_addr {
    in_addr_t s_addr;
};

struct sockaddr_in {
    sa_family_t     sin_family;
    in_port_t       sin_port;
    struct in_addr  sin_addr;
    unsigned char   __pad[X];
}
```

sockaddr_in.sin_port

- *`bind()` needs IP address and **port***
  - Port number identifies a specific socket on the machine
  - Some ports are well known, such as:
    - SSH: 22
    - HTTP: 80
  - Clients know to look at these ports
- *Ephemeral port*
  - If we don't `bind()` to a *specific* port, one will be chosen for us
  - Usually a random-ish, high-numbered port like 58032

```
struct in_addr {
    in_addr_t s_addr;
};

struct sockaddr_in {
    sa_family_t     sin_family;
    in_port_t       sin_port;
    struct in_addr  sin_addr;
    unsigned char   __pad[X];
}
```

Byte Order and Hosts

Byte Order

- *Recall: different computer architectures use different byte orders*
  - Consider the number 12345 = 0x3039
  - Little Endian: store the little part (least-significant byte=LSB) first (at lower address)
  - Big Endian: store the big part (MSB) first (at lower address)
- *Network Byte Order*
  - Different computers communicate, so network must have established byte order
  - “Network Byte Order” is *big-endian*
  - E.g., port number and the IP address are multi byte, so they are sent MSB first: `{ 0x30, 0x39 }`
  - For your own protocols, you may use whatever order you want, but if you are accessing data in packet headers, expect “network” order

Footnote - Endianness

- People have in the past been very militant about the “correctness” of big- or little-endian
- Big-endian makes for readable data dumps, and the proponents are very vocal, but the industry converged on little-endian when 68k disappeared
  - When doing long-hand math operations not natively supported by the CPU, you most likely want to work from least-significant to most-significant
  - It’s convention that pointers start at the lowest address and grow to highest, so ideally you want the iteration order to match the order of bits from least-significant to most
  - Try writing a program to add arbitrary-precision integers and you will understand what I mean
    - Bonus: if you are on a little-endian machine, you can change an arbitrary-precision addition algorithm from byte- to word-at-a-time without changing the data format!
  - This also shows up in packed data formats, usually it’s the least-significant bits sent always, and the most-significant optionally… it’s simpler when the mandatory parts are sent first
  - *Very* few programmers are staring at raw memory dumps regularly, nowadays
- In the context of networking, if you are looking at bits transmitted over the wire, you must also consider the *bit*-endianness
  - for many types of serial transmission, including (lower-speed) Ethernet, this is LSB-first, meaning at the hardware level network order is actually *middle*-endian: oops
- **If you really don’t like how your little-endian memory dumps look, you can reverse *all* the data: try formatting them right-to-left!***

  ** I don’t actually recommend this, but if you try it let me know how it works out for you…*

Network Byte Order

- *Byte-order translation functions*
  - `man byteorder`
  - “Host To Network Long”, etc.
- *Only for multi-byte values*
  - single byte data (`char`, `uint8_t`) just sent one at a time

```
#include <arpa/inet.h>
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

Activity

- *Create two programs, server and client (15m)*
  - Implement the socket sequence (TCP stream) using `AF_INET`
  - Send messages from the client and print them out from the server
  - Use port 8000 on the server.
- *Recall*
  - `AF_INET` uses `sockaddr_in`

```
struct in_addr {
    in_addr_t s_addr;
};

struct sockaddr_in {
    sa_family_t     sin_family;
    in_port_t       sin_port;
    struct in_addr  sin_addr;
    unsigned char   __pad[X];
}
```

Host Names

- *Can use a host name instead of an IP address*
  - “Host” is the computer, host name is the computer name
  - `getaddrinfo()`

    Converts host name (string) to set of all possible options (combinations of stream type etc., probably)
  - `getnameinfo()`

    Does the reverse: IP to host name

recv() and send()

- `ssize_t recv(int sockfd, void *buf, size_t len, int flags);`
  - Similar to read() but socket specific
  - Provides more control
    - `MSG_DONTWAIT`: Non-blocking
    - `MSG_PEEK`: read but don't remove
- `ssize_t send(int sockfd, const void *buf, size_t len, int flags);`
  - Similar to write() but socket specific
  - Provides more control
    - `MSG_DONTWAIT`: Non-blocking

Summary

- *Use AF_INET for IPv4*

```
struct in_addr {
   in_addr_t s_addr;
};

struct sockaddr_in {
   sa_family_t     sin_family;
   in_port_t       sin_port;
   struct in_addr  sin_addr;
   unsigned char   __pad[X];
}
```

- *Network Byte Order*
  - Big-Endian: most significant (biggest) byte is first