```
CS 214 / 2021-04-21
===================

Questions for networking
    - how do we identify the hosts/processes communicating
    - how do messages get from their source to their destination
    - how do different machines send messages without interfering

Each level answers this question differently

Link level
    - typical examples: Ethernet, Wi-Fi (802.11)
        - less typical examples: ATM, Token ring, AppleTalk, Novell
        - Bluetooth can also be considered link-level

    Ethernet
        - addresses: MAC (must be unique within a network)
        - packet based
            - communication based on packets
            - each packet identifies the machine it is intended for
        - "best effort" delivery
             - packets can get lost
        - does not have "quality of service" guarantees
            - no way to reserve time on the network

    Hub and spoke model
        Each host on a link connects to a hub
            -> one hub, many hosts
        The hub forwards every message to every host
        Hosts ignore messages not intended for them
        -> not secure at all; Ethernet assumes all hosts on the network are trusted

    We can connect hubs to other hubs to increase the size of our network
    -> this does not scale well to large networks

    Switch
        optimized hub
            -> only sends messages to some hosts
            -> more expensive/complicated than a hub
        divides link into sub-links
            -> learns which MAC addresses are present in which sub-link
        messages are only sent to the sub-link that contains the destination


Routers - network-level connections
    -> a host on a link that connects to an outside network
    -> a connection point between two links
        -> can even connect different link types
            -> wi-fi to ethernet
            -> ethernet to DSL/cable modem/etc.
            -> etc.
    -> behaves a lot like a switch

Hub vs switch - same level, but have different complexity
Switch vs router - different levels (link vs network)

Internet level
    - identifies hosts using IP addresses
        IPv4 address is 4 bytes (32 bits)
            usually written in "dotted quad" form: each byte written in decimal
                separated by periods (eg., 127.0.0.1)
        IPv6 address is 16 bytes (128 bits)
    - most IP addresses refer to specific hosts
        - some hosts may have multiple IP addresses (rare)
        - some IP addresses have special meaning

    - IP addresses are obtained in blocks with a common prefix
        - these blocks are assigned to organizations by various international groups
        - ultimately, ICANN is responsible for distributing IP addresses

    IP addresses with a common prefix controlled by a single organization are
        called a "subnet"
        We can identify a subnet by writing a slash and the number of bits after
        the IP address

        a.b.c.d/p
            the first p bits identify the subnet
            the last 32-p bits identify a specific host within the subnet

        33.54.120.240/16
            33.54.x.x  <- subnet identifier / subnet mask

    Routers use subnet masks to simplify routing tables
        -> all IP addresses in the same subnet will be routed the same way
        "classless internet routing"

    IPv4 is still the dominant Internet standard
        2^32 possible addresses (roughly 10^9)
            -> less than the population of the earth
            -> not enough in a world where everyone has â‰¥1 Internet device

    IPv6 is the intended solution
        -> increases number of addresses to 2^128 (approx. 10^36)
        -> has not been widely adopted

    What we actually did was introduce network address translation (NAT)
        share a single IP address between multiple devices
            devices "behind" the NAT use a range of "local"/"private" IP addresses
                -> addresses that are not globally unique / cannot be routed
        NAT device translates addresses in messages intended for local hosts

        NAT effectively live at the Transport layer
            may have to keep track of multiple on-going connections and forward
                packets to the appropriate local device

        -> it is hard to run a server behind a NAT
            the NAT must know which local device will handle a connection request
        -> non-connection-based protocols (UDP) require special knowlege on the NAT

        As far as the larger Internet can tell, all devices behind the NAT are
            the same host

    -> NAT is similar to a firewall, but is not the same thing
    -> a "firewall" is something that blocks certain communications
        -> typically exist at the router level
        -> block attempts to connect to specific services

Transport layer - process to process
    - TCP and UDP: processes are identified by host address and port number

            44.55.66.77:80

                indicates port 80 on host 44.55.66.77

    - specific services are associated with port numbers
        - certain port numbers are standard (e.g., HTTP is 80)
        - ports above 5000 are usually free for personal use

        -> port numbers are mapped to processes within a host
        -> each port is used by a single process
            -> cannot have two processes on a single port
            -> attempting to "bind" a port that is already in use will fail

        -> nothing in TCP or UDP requires any specific port to be used for any
            specific purpose
        -> the "standard" ports are just suggestions, but going along with
            conventions leads to fewer surprises

    Typical TCP scenario
        client connects to server at well-known port number
            client is given an arbitrary port number at its end

        Both ends are identified by host address + port number

How do we actually open a connection?
-> socket interface (originally for BSD Unix, now part of Posix)

#include <sys/socket.h>

    socket - creates an entry in the file table that we can use to connect over
        a network

        int   // returns a file descriptor, or -1 on failure (sets errno)
        socket(
            int domain,     // what kind of network/protocol family
                            // typical: AF_INET, AF_INET6
            int type,       // what are the semantics of the socket
                            // SOCK_STREAM  - this is a streaming socket (TCP)
                            // SOCK_DGRAM   - this is a datagram socket (UDP)
            int protocol    // identify the protocol, if more than one have the same
                            // semantics (almost always 0)
        );

    connect - establish a streaming connection to a remote host
        - only used with connection-oriented protocols (TCP)

        int    // 0 for success, -1 for failure
        connect(
            int socket,
            struct sockaddr *address,  // e.g., IP address + port
            socklen_t address_size     // size (in bytes) of *address
        );


        -> struct sockaddr is a lie; essentially acts a void pointer
            each domain has its own socket address struct (e.g., sockaddr_inet)
            the pointer we pass has to be to one of those
        -> we specify the size, because different domains have different-sized addresses
            IPv6 address larger than IPv4

How do we get the struct sockaddr for our specific domain?

We can construct it manually, but doing so is difficult
-> we need to be concerned about big- vs little-endian data
    -> IP addresses and port numbers must be given in "network order" (big-endian)
            hton() converts host-order integers to network order

-> we also need to look up the address of the remote host
    hosts are usually identified by domain name (candle.cs.rutgers.edu)
    domain name service (DNS) maps domain names to specific IP addresses

solution:
    getaddrinfo()
        -> we specify domain name or ip address + service name or port number
        -> it gives us the sockaddr with everything set up
        -> it is a little cumbersome, but it is the recommended way to do this

    I will post some sample code with comments explaining how to use getaddrinfo


    #include <netdb.h>

        struct addrinfo {
           int              ai_flags;
           int              ai_family;    // used with socket
           int              ai_socktype;  // used with socket
           int              ai_protocol;  // used with socket
           socklen_t        ai_addrlen;   // used with bind/connect
           struct sockaddr *ai_addr;      // used with bind/connect
           char            *ai_canonname;
           struct addrinfo *ai_next;
   };


    int    // 0 for success, non-zero for failure
    getaddrinfo(
        char *host,              // e.g., domain name or IP address (dotted quad)
        char *service,           // e.g., port number or service name
        struct addrinfo *hints,  // additional information (narrows down what we get)
        struct addrinfo **list   // output var: list will point to a linked list
                                                     // of struct addrinfo nodes
    );

        linked list is used in case there are multiple ways to connect to the remote
                host (e.g., if both IPv4 and IPv6 are available)

        typical use: iterate through list until we successfully connect or bind

        use freeaddrinfo() to deallocate the list once you are done with it


Basics (see sample code for a more complete version)

To connect to a remote host using TCP

        int sock, err;
        struct addrinfo hints, *info;

        // initialize hints
        memset(&hints, 0, sizeof(struct addrinfo));  // initialize all bytes to 0
        hints.ai_family   = AF_UNSPEC;    // allow multiple domains (AF_INET and AF_INET6)
        hints.ai_socktype = SOCK_STREAM;  // we want a streaming connection


        err = getaddrinfo(remote_host, service, &hints, &info);
                // getaddrinfo("www.rutgers.edu", "http", &hints, &info)
                // getaddrinfo("candle.cs.rutgers.edu", "22", &hints, &info)
        if (err != 0) ...

        // now info points to a linked list of addresses

        // create socket
        sock = socket(info->ai_family, info->ai_socktype, info->ai_protocol);
        if (sock < 0) ...

        // connect to remote host
        err = connect(sock, info->ai_addr, info->ai_addrlen);
        if (err) ...

        freeaddrinfo(info);

        write(sock, "Hello\n", 6);

To accept incoming connections

        int listener, connection, err;
        struct addrinfo hints, *info;
        struct sockaddr_storage remote_addr;  // as big as the largest supported address struct
        socklen_t remote_addrlen;

        // initialize hints
        memset(&hints, 0, sizeof(struct addrinfo));
        hints.ai_family   = AF_UNSPEC;
        hints.ai_socktype = SOCK_STREAM;
        hitns.ai_flags    = AI_PASSIVE;   // indicate we are going to use listen()

        err = getaddrinfo(NULL, service, &hints, &info);
                // NULL addresss means we want a port on our own host
        if (err != 0) ...

        listener = socket(info->ai_family, info->ai_socktype, info->ai_protocol);
        if (listener < 0) ...

        // associate this socket with a port
        err = bind(listener, info->ai_addr, info->ai_addrlen);
        if (err != 0) ...

        // set up queue of incoming connection requests
        err = listen(listener, queue_length);
        if (err != 0) ...

        freeaddrinfo(info);

        // wait for an incoming connection request
        connection = accept(listener, (struct sockaddr *) &remote_addr, &remote_addrlen);
                                // accept(listener, NULL, NULL)
        if (connection < 0) ...

                // use getnameinfo() to convert remote_addr back to human-readable strings

        read(connection, buf, buflen);
        write(connection, response, responselen);
```