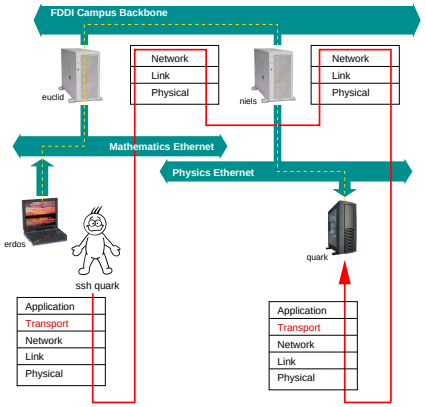
## NETWORKS

- Networks are all about transmitting messages between individuals
- As old as mankind
  - Consider Stone Age: $A$ wants to invite $B$ to his place, uses a drum
  - But $B$ is too far away to hear; then $A$ can
    1. get a bigger drum
    2. walk to $B$'s place
    3. ask $C$ (who lives halfway) to forward the invitation $\rightarrow$ networking!
  - Of course, we now use computers, fiber optics, satellites, etc. and we send each other emails or tweets



## NETWORKS (CONT'D)




| | |
|---|---|
| Application | C/C++ |
| Transport | TCP |
| Network | IP |
| Link | Ethernet |
| Physical | Twisted pair |

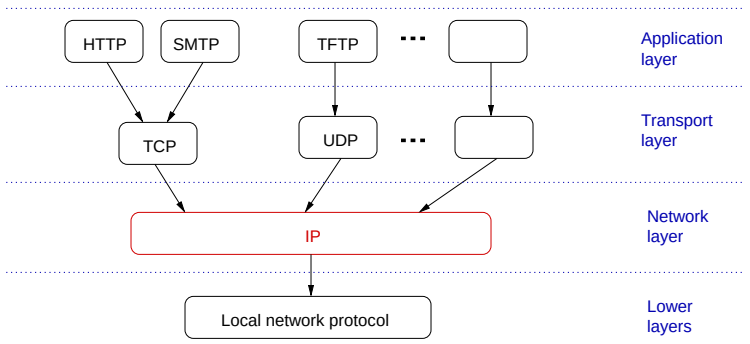


## THE INTERNET PROTOCOL (IP)

- A (connectionless) network layer protocol
- Designed for use in interconnected systems of packet-switched computer communication networks (store-and-forward paradigm)
- Provides for transmitting blocks of data called datagrams from sources to destinations
  - The datagram may possibly go through intermediate hosts
  - Sources and destinations are hosts identified by fixed length addresses
- Also provides for fragmentation and reassembly of long datagrams, if necessary, for transmission through "small packet" networks
- The workhorse of data exchange
- Both TCP and UDP use it to carry packets from one host to another
- Much like UDP (which is thus a thin layer on top of IP) in behaviour



## RELATION TO OTHER PROTOCOLS



## INTERFACES

- IP is called on by host-to-host protocols in an internet environment
- In turn, IP calls on local network protocols to carry the internet datagram to the next gateway or destination host
- a participating endpoint host needs to know its IP address (192.168.0.1), netmask (255.255.255.0), and its gateway address (192.168.0.254)
  - a host can infer its broadcast address (192.168.0.255) whose use implies the sending of the datagram to all the hosts within the netmask.
    - with these coordinates, the host sits in the 192.168.0.0 network
  - anything addressed to an IP address within the netmask is passed directly to the lower network layer (MAC)
  - other datagrams are sent to the gateway by calling once more the lower layer
- The gateway is a host that connects to two (or more) networks via two (or more) local network interfaces. It is also called a router



## IP ADDRESSES

- IP addresses have a fixed length of four bytes (32 bits)
- an address begins with a network number, followed by local address (called the "rest" field).
  - For instance 192.168.0.15 is formed from the 192.168.0.0 (192.168.0.0/24) network number followed by the local address 15.
- Three classes of IP addresses (historical importance only):

  **Class A** high order bit is 0, next 7 bits are the network, the last 24 bits are the rest

  **Class B** high order bits are 10, next 14 bits are the network, last 16 bits are the rest

  **Class C** high order bits are 110, next 21 bits are the network, last 8 bits are the rest
- Nowadays the network and the rest are given exclusively by the netmask



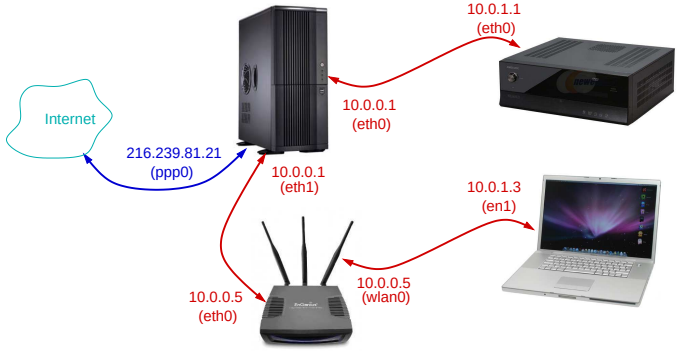
## PRIVATE NETWORKS

- A private network uses private IP address spaces (RFC 1918, RFC 4193)
- Private addresses are not globally delegated
  - They are not allocated to any specific organization; IP packets addressed by them cannot be transmitted onto the public Internet.
  - If a private network needs to connect to the Internet, it must use either a network address translator (NAT), or a proxy server.
- Private addresses can in fact coexist with "real" addresses
- Private IP ranges:

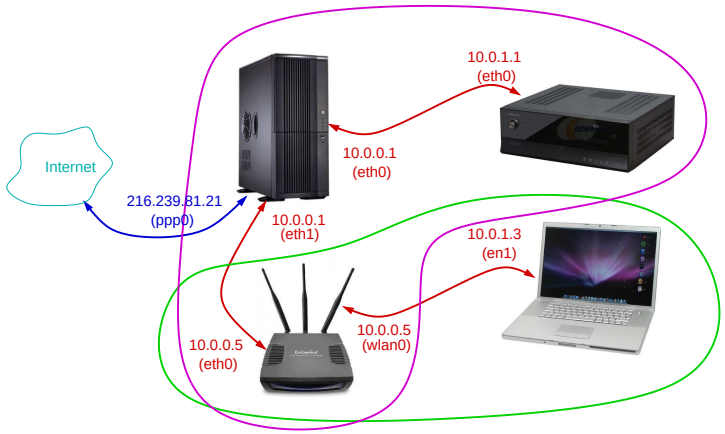
| Class | Address range | No. addresses | Mask | Rest size |
|---|---|---|---|---|
| Single class A | 10.0.0.0–10.255.255.255 | 16,777,216 | 255.0.0.0 | 24 bits |
| 16 class B | 172.16.0.0–172.31.255.255 | 1,048,576 | 255.240.0.0 | 20 bits |
| 256 class C | 192.168.0.0–192.168.255.255 | 65,536 | 255.255.0.0 | 16 bits |



## INTERFACES (CONT'D)

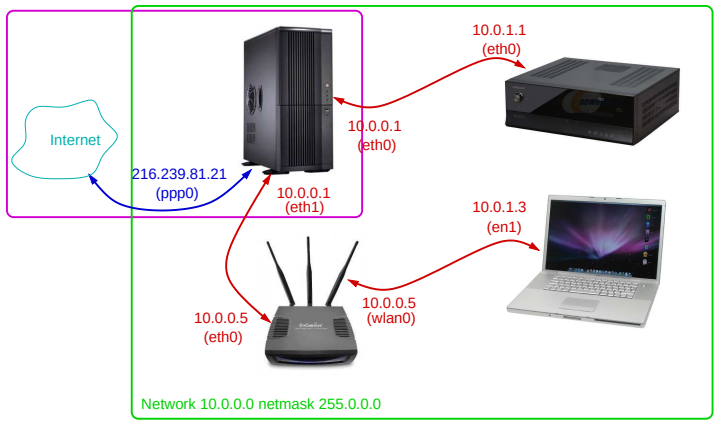


## INTERFACES (CONT'D)






## INTERFACES (CONT'D)






## EXAMPLE

- 10.0.1.3 sends a TCP packet to 216.109.118.67
- TCP calls on the IP to take a TCP packet (including the TCP header and user data) as the data portion of a datagram
  - TCP provides the addresses and other parameters
- IP assembles the datagram, notices that 216.109.118.67 is not a local address, and thus sends the packet to the gateway (10.0.0.1) through eth0.
- The gateway receives the packet and repeats the same algorithm
  - the destination address is not in the 10.0.0.0 network, so the gateway sends the packet through its ppp0 interface
  - NAT also takes place here (whenever applicable)



## SETTING UP A ROUTER: SCENARIO AND LEVEL 2 SETUP

- One ADSL card for the outside connection (`eth4`/`ppp0`)
  - Use PPPoE via `pppd`; once PPPoE is up we forget about `eth4` and use `ppp0`
- Two Ethernet cards for the local network (`eth0`, `eth1`)
- One wireless interface (`wlan0`)
  - Install and configure `hostapd`; `wlan0` becomes a normal network interface
- IP is not designed to work with multiple interfaces toward a single network
- So we bridge multiple interfaces (so that they appear as one at the IP level)

```
< post:/etc/conf.d > brctl addbr br0
< post:/etc/conf.d > brctl addif br0 eth0
< post:/etc/conf.d > brctl addif br0 eth1
< post:/etc/conf.d > brctl addif br0 wlan0
< post:/etc/conf.d > brctl show
bridge name     bridge id               STP enabled     interfaces
br0             8000.00104b9f2417       no              eth0
                                                        eth1
                                                        wlan0
```

## Setting up a Router: IP Addresses and Routes

- Assign an IP address and netmask to every interface (`ppp0` IP assigned via PPPoE)

```
ifconfig br0 10.0.0.1 broadcast 10.255.255.255 netmask 255.0.0.0
```

- Set the kernel to forward packets `echo 1 > /proc/sys/net/ipv4/ip_forward`
- Now the machine knows how to send packets

```
< post:/etc/conf.d > route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags   Use   Iface
0.0.0.0         216.239.80.253  0.0.0.0         UG        0   ppp0
10.0.0.0        0.0.0.0         255.0.0.0       U         0   br0
127.0.0.0       127.0.0.1       255.0.0.0       UG        0   lo
192.168.1.0     0.0.0.0         255.255.255.0   U         0   eth4
216.239.80.253  0.0.0.0         255.255.255.255 UH        0   ppp0
```

- Every machine in the `10.0.0.0` network specifies the router as "default gateway"

```
route add default gw 10.0.0.1
```

  - The gateway must already be reachable!



## Setting up a Router: IPTables and NAT

- `iptables` processes packets going in, out, and through
- useful between others for firewalling and NAT
- Functionality based on rules included in chains, grouped into tables according to functionality
  - Interesting tables: `filter` (default) and `nat`
  - Interesting chains: `INPUT`, `OUTPUT`, `FORWARD`, `PREROUTING`, `POSTROUTING`
  - Each chain has a default policy (`ACCEPT`, `REJECT`, `DROP`)
- Roles are added one by one to chains
  - Drop incoming UDP 225: `iptables -I INPUT -p udp --dport 225 -j DROP`
  - Allow outgoing TCP 722: `iptables -A OUTPUT -p tcp --dport 722 -j ACCEPT`
  - "Get lost" to auth requests: `iptables -A INPUT -p tcp --dport auth -j REJECT`
  - Limis SSH connection attempts to 3 per minute:

```
iptables -I INPUT -p tcp --dport 22 -i $EXTIF -m state --state NEW \
   -m recent --set
iptables -I INPUT -p tcp --dport 22 -i $EXTIF -m state --state NEW \
   -m recent --update --seconds 60 --hitcount 4 -j DROP
```

  - `iptables -t nat -A POSTROUTING -o ppp0 -s 10.0.0.1/8 -j MASQUERADE`
  - `iptables -t nat -A PREROUTING -i ppp0 -p udp --dport 5060 -j DNAT \ --to 10.0.0.2`



## Setting up a Router: Daemons & Goodies

- DNS (domain name resolution) accessed through the `gethostbyname` system call
  - Has its own protocol (servers and clients)
  - Every machine that wants to use DNS has to know at least one DNS server by IP (`/etc/resolv.conf`)
  - DNS server solutions include `bind` (Berkeley Internet Name Domain) and `dnsmasq` (both DNS and DHCP)
- DHCP allows for a machine that joins the network to ask for and obtain an IP address automatically
  - Routing information (default gateway) and DNS servers are also obtained
  - IP address almost random, but MAC reservations can be set up
- Wireless authentication and accounting needed for the WiFi part
  - Shared key solution (WPA2 "personal"; WPA and WEP not recommended) implemented directly by `hostapd`
  - For WPA2 "enterprise" and accounting an external authentication and accounting server needed (most popular: `radius`)



## IP Operation

- Two basic functions: addressing and fragmentation
  - IP modules use the addresses carried in the internet header to transmit internet datagrams toward their destinations, hop by hop
  - This (distributed) selection of a transmission path is called routing
  - In the process the packets may be fragmented
    - the fragmenting and reassembling is the exclusive duty of IP
  - The model of operation is that an IP module resides in each host engaged in internet communication and in each gateway that interconnects networks
    - These modules share common rules for interpreting address fields and for fragmenting and assembling internet datagrams
    - In addition, these modules (especially in gateways) have procedures for making routing decisions and other functions (routing algorithms)
    - IP treats each internet datagram as an independent entity unrelated to any other internet datagram.
      - There are no connections or logical circuits

## KEY MECHANISMS

**Type of Service** indicate the quality of the service desired

- abstract or generalized set of parameters which characterize the service choices provided in the networks that make up the internet
- used by gateways to select the actual transmission parameters, the network to be used for the next hop, or the next gateway

**Time to Live** an upper bound on the lifetime of an internet datagram

- set by the sender and reduced at the points along the route where it is processed
- if the time to live reaches zero before the datagram reaches its destination, the datagram is destroyed



## KEY MECHANISMS (CONT'D)

**Options** provide for control functions needed or useful in some situations but unnecessary for the most common communications

- include provisions for timestamps, security, and special routing

**Header Checksum** provides a verification that the information used in processing internet datagram has been transmitted correctly

- if the header checksum fails, the datagram is discarded at once by the entity which detects the error.

- The internet protocol does not provide a reliable communication facility
  - no acknowledgments (either end-to-end or hop-by-hop)
  - no error control for data (only a header checksum)
  - no retransmissions
  - no flow control



## ROUTING

- IP provides a store-and-forward, packet switching internet
  - datagrams are stored into queues in various routers and forwarded between routers until they reach their destination
- An IP datagram has the capability to provide a route to be followed
  - a route is a sequence of IP addresses
  - split into two parts
    - recorded route or the route travelled so far, and
    - source route or the route yet to be followed
  - Then the routing algorithm is very simple
  - However, is the source route becomes empty at some point, the routing algorithm forwards the datagram solely according to the destination address
    - the recorded route continues to be filled in
    - we then enter the realm of real routing algorithms



## THE OPTIMALITY PRINCIPLE

- All the routing algorithms are based on the optimality principle:

  If a router $J$ is on the optimal path from router $I$ to router $K$ then the optimal path from $J$ to $K$ also falls along the same route

## The Optimality Principle

- All the routing algorithms are based on the optimality principle:

  If a router $J$ is on the optimal path from router $I$ to router $K$ then the optimal path from $J$ to $K$ also falls along the same route

- As a consequence, the set of all the optimal routes from all the sources to a given destination form a tree rooted at the destination (the sink tree)
  - no loops, so each packet will be delivered after a finite number of hops if following the optimal route
  - in practice life is not that easy
    - links and routers go down and come back up
    - the network image of a router is not necessarily the same as the image of other routers



## Routing Algorithms

- Various algorithms have been used, including:

  **Flooding** every incoming datagram is sent to every outgoing line
  - is there any possibility that the number of duplicate datagrams increase without bounds?
  - flooding is very inefficient, but has its uses (e.g., in military applications)

  **Shortest path routing** when forwarding a packet, a router computes the shortest path to the destination and sends the datagram to the next hop along this path
  - the metrics used for paths are varied, including the number of hops, the geographic distance, and delivery delay (including queuing or not)
  - The shortest path is computed using a greedy algorithm such as Dijkstra's.

  **Distance vector routing** (ARPANET until 1979) each router maintains a table giving the best known distance to each destination and which network interface to use to get there
  - tables are constructed by exchanging information between routers



## Link State Routing

- Most widely used algorithm nowadays
- Each router performs an algorithm consisting in the following steps:
  1. Discover the neighbours and learn their network addresses
  2. Measure the delay or cost to each neighbour
  3. Construct a packet telling all it just learned
  4. Send this packet to all the other routers
  5. Compute the shortest path to all the other routers
- In effect, the topology of the network and the delays are experimentally measured
- Dijkstra's algorithm can then be used to find the shortest path

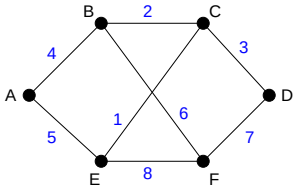


## Learn About Neighbours, Measure Delays

- Once a router is booted, it sends a "HELLO" packet to each interface
  - inter-router links are conceptually viewed as point-to-point
  - the router on the other end is supposed to send back a reply telling who it is
  - the names must be globally unique (e.g., the MAC address)
- The router sends then an "ECHO" packet to its neighbours, which is bounced back immediately
  - reasonable estimate of the delay
  - may include actual network traffic (by including queueing time) or not

## Constructing the Link State Packets

- the packet contains the identity of the sender, a sequence number, and age, and a list of neighbours
  - for each neighbour the delay to that neighbour is given




| A | B | C | D | E | F |
|---|---|---|---|---|---|
| Seq. | Seq. | Seq. | Seq. | Seq. | Seq. |
| Age | Age | Age | Age | Age | Age |
| B / 4 | A / 4 | B / 2 | C / 3 | A / 5 | B / 6 |
| E / 5 | C / 2 | D / 3 | F / 7 | C / 1 | D / 7 |
| | F / 6 | E / 1 | | F / 8 | E / 8 |

  - problem: when to build link state packets?
    - periodically, or
    - whenever a significant event occurs (neighbour goes down, neighbour comes back up, neighbour communication changes properties dramatically)



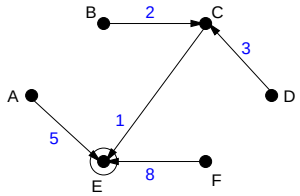
## Distribute Link State Packets

- We use flooding
- Routers keep track of all the source–sequence pairs they see to contain flooding
  - when a new packet comes in, it is checked against the corresponding pair
  - if it is new, it is forwarded on all the lines
  - if the stored sequence number is larger (or is a duplicate), the packet is discarded
  - the age is decremented each second and the packet is discarded when age reaches zero; this guards against corrupted or wrapped sequence numbers



## Compute Routes

- Once a router has accumulated all the packets, it can reconstruct the network graph
  - each edge in the graph is actually represented twice, once for each direction
- Now we run Dijkstra's algorithm at each router to compute the minimum-cost spanning tree from the router to all the other destination




- The result is installed in the router as a routing table and normal operation begins



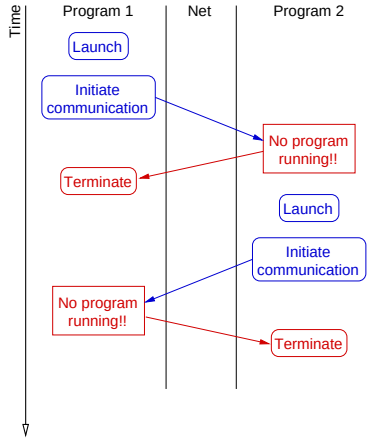
## Characteristics

- For an internet with $n$ routers of degree $k$ the memory required to store the routing table is $O(k \times n)$
  - For large internets this can be a problem
- Link state routing is sensitive to hardware failure (but what algorithm isn't?)
- In practical settings link state routing works well, so (slightly improved) variants are in wide use today
  - The Internet is a huge place, but internets are not very large since they are separated by "border" routers with routing tables that look like this:

```
Destination     Gateway         Genmask          Iface
216.239.80.245  0.0.0.0         255.255.255.255  ppp0
10.0.0.0        0.0.0.0         255.0.0.0        br0
127.0.0.0       0.0.0.0         255.0.0.0        lo
0.0.0.0         216.239.80.245  0.0.0.0          ppp0
```

## Client-Server Applications

- TCP/IP provides peer-to-peer communication.
- We launch two programs and want them to communicate with each other.
  - Chances are, we will not be able to convince them to meet.
- So we split responsibilities:
  - One party (the server) must start execution and wait indefinitely for incoming requests.
  - So the other party (the client) will simply connect, knowing that somebody at the other end will listen.
- This way, we also simplify the TCP/IP mechanisms, which do not need to create programs or something equally hairy.





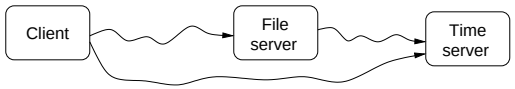

## Client Issues

- When connecting to a server, a client has to know the address of the machine and a port number
  - Port numbers identify the actual server to connect to
- Standard versus nonstandard
  - In any case, the client must speak the server's language
- Parameterization.
  - Some clients do one thing only, e.g., manage file transfers
  - Some (parameterized) clients can access many services
    - `telnet` is a fully parameterized client



## Server Issues

- Connection or connectionless
  - Connection-oriented servers assume that all the data packets arrive correctly and in order (TCP)
  - A connectionless server does no assume any delivery guarantee (UDP; there might be lost packets, duplicates, and out of order packets)
    - The application (both client and server) should contain code that deals with losses, duplication, etc
  - Major design issue. TCP introduces some overhead, but is in general preferred because it simplifies design
- Servers and clients (for other servers), e.g.,






## State Information

- To keep or not to keep state information, that is the question
- A stateless server does not remember what the client did, a stateful one does
  - Stateless or stateful?
    - File serverallowing clients to access a given piece of data from a given file
    - POP server, that allows clients to retrieve their email messages which have not been previously received
    - HTTP server for an e-commerce site
- Statelessness is a protocol issue
- A stateful server
  - may be more efficient
  - is difficult to maintain in case of loss of communication and/or computer crash
  - problems with identifying clients
- A stateless server
  - operations must be idempotent
  - but copes well with loss of communication/computer crash

## A TCP Client

1. Get the IP address and port number of the peer
2. Allocate a socket
3. Choose a local IP address
4. Allow TCP to choose an arbitrary, unused port number
5. Connect the socket to the server
6. Communicate with the server
   - i.e., send requests and await replies
   - we use here the application-level protocol
7. Close connection



## Peer Identification

- Depending on the actual application, the IP address of the peer (i.e., server) can be specified in more than one ways, including:
  - hardcoded (rarely)
  - as command-line argument (read from hard disk, etc.) – use `gethostbyname` to get the actual address (i.e., number)
  - use a separate protocol (broadcast or multicast) to find a server
- Ports can also be specified in many ways, including:
  - well-known port – use `getservbyname` to obtain the actual port number
  - hardcoded – e.g., when doing custom client-server applications
  - as command-line argument (read from hard disk, etc.)
    - especially good for fully parameterized clients

```
telnet cs-linux.ubishops.ca 22
```



## Allocate a Socket

- We have to specify:
  - the protocol family
  - the socket type (TCP here)

```
#include <sys/types.h>
#include <sys/socket.h>

int sd = socket(PF_INET, SOCK_STREAM, 0);
```

- We end up with a socket descriptor
  - Entry in the descriptor table and so usable as any such an entry



## Choosing a Local IP Address

- Why do we need the local IP address?

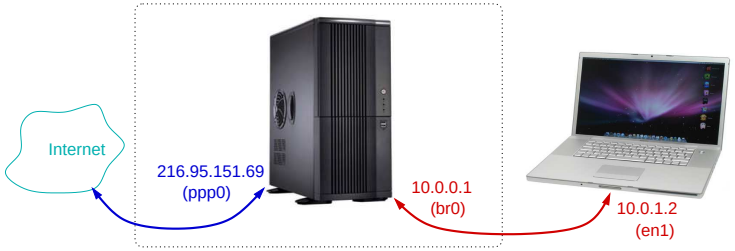
## Choosing a Local IP Address

- Why do we need the local IP address?
  - Because a connection is specified by two endpoints
- Why is it a problem to choose a local IP address?




## Choosing a Local IP Address

- Why do we need the local IP address?
  - Because a connection is specified by two endpoints
- Why is it a problem to choose a local IP address?




  - IP must be able to route packets in the right direction
  - Choosing the IP address is done after a dialogue with IP
  - The system call `connect` does it for us



## Choose a Port

- We have to specify a local port number for the same reasons we have to specify a local address
- The choice of port number does not matter as long as:
  - it does not conflict with the port assigned to a well-know service
  - it is not in use by another process
- We could try at random until we get a free port...
  - ...However, the system keeps track of port usage anyway, so this would be overkill
  - So the port number choice is again taken care of by the call to `connect`



## Connect to the Server

- We obtain the local coordinates (IP address, port) and we connect in one step:

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

- Something like this:

```
#include <errno.h>
extern int errno;
struct sockaddr_in sin;
int rc = connect(sd, (struct sockaddr *)&sin, sizeof(sin));
if (rc < 0) {
    perror("connect");
    exit(1);
}
```

## Filling in the Server Address

```
int connectbyportint(const char* host, const unsigned short port) {
    struct hostent *hinfo;              struct sockaddr_in sin;
    const int type = SOCK_STREAM;       int sd;

    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    hinfo = gethostbyname(host);
    if (hinfo == NULL) return err_host;
    sin.sin_addr=(unsigned int)hinfo->h_addr;

    sin.sin_port = port;

    sd = socket(PF_INET, type, 0);
    if ( sd < 0 ) return err_sock;

    int rc = connect(sd, (struct sockaddr *)&sin, sizeof(sin));
    if (rc < 0) {
        close(sd);
        return err_connect;
    }
    return sd;
}
```



## Filling in the Server Address

```
int connectbyportint(const char* host, const unsigned short port) {
    struct hostent *hinfo;              struct sockaddr_in sin;
    const int type = SOCK_STREAM;       int sd;

    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    hinfo = gethostbyname(host);
    if (hinfo == NULL) return err_host;
    sin.sin_addr=(unsigned int)hinfo->h_addr;

    sin.sin_port = port;

    sd = socket(PF_INET, type, 0);
    if ( sd < 0 ) return err_sock;

    int rc = connect(sd, (struct sockaddr *)&sin, sizeof(sin));
    if (rc < 0) {
        close(sd);
        return err_connect;
    }
    return sd;
}
```



## Filling in the Server Address

```
int connectbyportint(const char* host, const unsigned short port) {
    struct hostent *hinfo;              struct sockaddr_in sin;
    const int type = SOCK_STREAM;       int sd;

    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    hinfo = gethostbyname(host);
    if (hinfo == NULL) return err_host;
    sin.sin_addr=(unsigned int)htonl(hinfo->h_addr);

    sin.sin_port = (unsigned short)htons(port);

    sd = socket(PF_INET, type, 0);
    if ( sd < 0 ) return err_sock;

    int rc = connect(sd, (struct sockaddr *)&sin, sizeof(sin));
    if (rc < 0) {
        close(sd);
        return err_connect;
    }
    return sd;
}
```



## Filling in the Server Address

```
int connectbyportint(const char* host, const unsigned short port) {
    struct hostent *hinfo;              struct sockaddr_in sin;
    const int type = SOCK_STREAM;       int sd;

    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    hinfo = gethostbyname(host);
    if (hinfo == NULL) return err_host;
    memcpy(&sin.sin_addr, hinfo->h_addr, hinfo->h_length);

    sin.sin_port = (unsigned short)htons(port);

    sd = socket(PF_INET, type, 0);
    if ( sd < 0 ) return err_sock;

    int rc = connect(sd, (struct sockaddr *)&sin, sizeof(sin));
    if (rc < 0) {
        close(sd);
        return err_connect;
    }
    return sd;
}
```

## Communicate with the Server

- We send data using `send` or `write`
- We receive responses using `recv` or `read`
  - Note that the response could come in pieces, even if the server answers back in large chunks
  - You should be prepared to accept data a few bytes at a time

```
const int ALEN = 128;
char* req = "some sort of request";
char ans[ALEN];
char* ans_ptr = ans;
int ans_to_go = ALEN, n = 0;

send(sd,req,strlen(req),0);

while ( ( n = recv(sd,ans_ptr,ans_to_go,0) ) > 0 ) {
  ans_ptr += n;
  ans_to_go -= n;
}
```



## Communicate with the Server (cont'd)

- What if we do not know how large the response is?
- This all depends on the application-level protocol; for instance, the response may be:
  - One line of text, terminated by `'\n'`
    - We could use `readline` to read the answer
  - One line of text determines what comes after it
    - Again, we use `readline` to read one line at a time, and decide what to do next
  - As much as the server cares to send, no special end marker.
    - We read until there is no more data
    - But how?



## Communicate with the Server (cont'd)

- Communication is not instantaneous, so we have to give some time for the data to arrive.

```
const int recv_nodata = -2;

int recv_nonblock (int sd, char* buf, size_t max, int timeout) {
  struct pollfd pollrec;
  pollrec.fd = sd;
  pollrec.events = POLLIN;

  int polled = poll(&pollrec,1,timeout);
  if (polled == 0) return recv_nodata;
  if (polled == -1) return -1;
  return recv(sd,buf,max,0);
}
```

- Outcomes:
  - -2: no more data available within the given `timeout`
  - 0: end of file (when the server closes connection on us)
  - $n > 0$: $n$ characters have been read



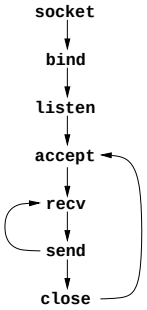
## Closing the Connection

- `close` closes the connection and destroys the socket.
- Sometimes we want to shut down communication in one direction only:
  - The server receive a request and responds to it
  - What does it do now with the connection?
    - If the client has in fact more requests, the connection should stay open
    - If this is the last request, the connection should be closed
- A client or server can partially close a connection to let its peer know that it is done

```
int err = shutdown(sd,SHUT_WR);
```

  - The other side will then receive and end of file
- The second argument of `shutdown` can be
  - `SHUT_RD` (0): further receives will be disallowed
  - `SHUT_WR` (1): further sends will be disallowed
  - `SHUT_RDWR` (2): neither receives, nor sends will be allowed

## SERVERS



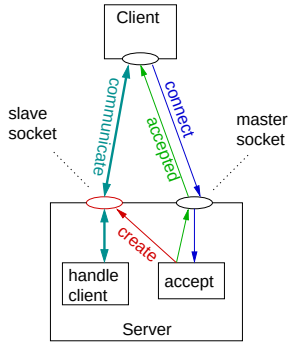

| | | |
|---|---|---|
| TCP | iterative connection−oriented | concurrent connection−oriented |
| UDP | iterative connectionless | concurrent connectionless |

- We consider TCP servers
  - point-to-point communication
  - reliable connection establishment and delivery
  - flow-controlled transfer
  - full duplex transfer
  - stream paradigm (no message boundaries)



## ITERATIVE SERVERS

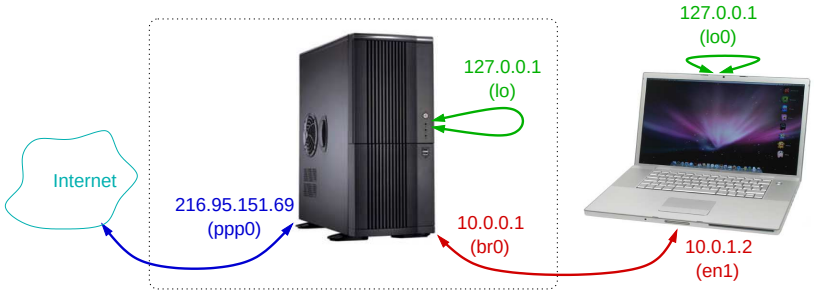
1. create a master socket
2. bind the socket to a known address (IP address + port number)
3. place the socket in passive mode
4. repeat forever:
   (a) accept the next connection request from the socket and create a new slave socket $s$ for the connection.
   (b) read a request from the client
   (c) serve the request and reply to the client
   (d) if finished with the client, close the socket $s$ (civilized servers shut down $s$ first!); otherwise, repeat from 4b






## BINDING THE SOCKET

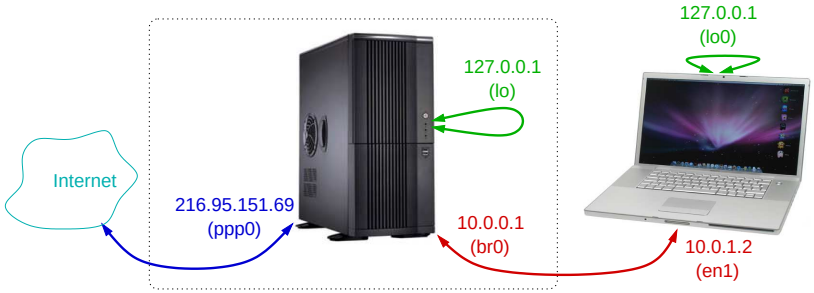
- We specify the IP address and the port number using the structure `sockaddr_in`
  - Yes, but what address do we provide?






## BINDING THE SOCKET

- We specify the IP address and the port number using the structure `sockaddr_in`
  - Yes, but what address do we provide?




  - We can use `INADDR_ANY`
  - This denotes a “wildcard” that matches all the IP addresses of the given host.

## The Problem with Iterative Servers (and a Solution)

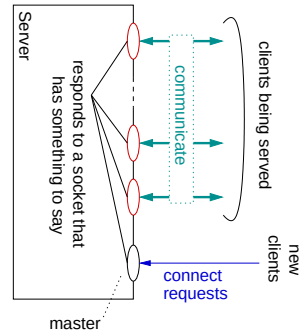
- If two clients connect quasi-simultaneously, one of them will have to wait till the other closes its connection
  - This could be a loong wait
  - We need some form of concurrency no matter what; let's fake it:



## The Problem with Iterative Servers (and a Solution)

- If two clients connect quasi-simultaneously, one of them will have to wait till the other closes its connection
  - This could be a loong wait
  - We need some form of concurrency no matter what; let's fake it:

1. create, bind and place in passive mode the master socket
2. repeat forever:
   (a) from all the open sockets, select a socket $s$ that has data available
   (b) if $s$ is the master socket, then
      i. accept the next connection request from the socket and create a new slave socket for the connection.
   (c) otherwise,
      i. read a request from $s$
      ii. serve the request and reply
      iii. if finished with the corresponding client, close $s$

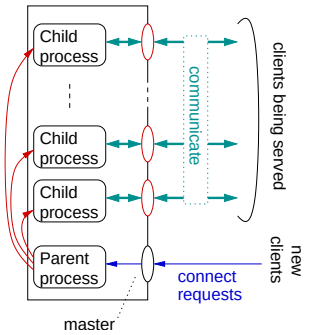



CS 409, Fall 2013 Networking/47

## (Really) Concurrent Servers

- Apparent concurrency is certainly possible, but hairy
- But then we do not need to fake concurrency since it is offered by the system anyway

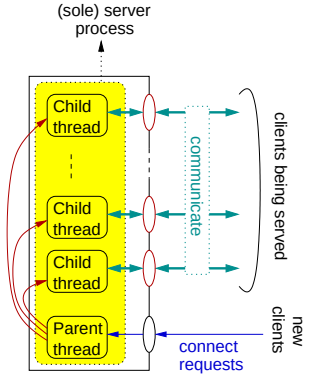
1. create, bind and place in passive mode the master socket
2. repeat forever:
   (a) accept the next connection request from the socket and create a new slave socket $s$ for the connection.
   (b) fork
   (c) if children process then
      i. close master socket
      ii. read a request from the client
      iii. serve the request and reply
      iv. if finished with the client, close $s$ and terminate; otherwise, repeat from 2(c)ii
   (d) otherwise (i.e., if parent process),
      i. close slave socket






## Multi-Threaded Servers

1. create, bind and place in passive mode the master socket
2. repeat forever:
   (a) accept the next connection request from the socket and create a new slave socket $s$ for the connection.
   (b) pthread_create; in the new thread:
      i. do not close master socket
      ii. read a request from the client
      iii. serve the request and reply
      iv. if finished with the client, close $s$ and terminate; otherwise, repeat from 2(b)ii
   (c) do not close slave socket



## Monitoring a Server

- Gathering statistics on server usage is easy in a multithreaded environment, because of the global variables that are accessible from all the threads:
  - We build a structure with statistical data of interest
  - We create a monitor thread that will from time to time process the statistical data and print it (or write it in a log file, etc.)
  - The other threads update this structure according to what they did
  - Since the structure is used by all the running threads, we have to put all the accesses to it in critical regions



## Managing Concurrency

- In the general case though, concurrency does yield better performance
- We use as concurrency measure the number $n$ of simultaneous threads that execute at a given time (be they in the same process or in different processes)
- Still, demand-driven concurrency is not necessarily the best choice
  - For one thing, $n$ can grow unboundedly. Anything that grows without bounds is bad
  - In particular, if we have tons of threads, we end up spending most of the time doing context switching
- Idea no. 1: limit the number of threads that can run simultaneously to a fixed limit $n_{\max}$. (how?)



## Managing Concurrency

- In the general case though, concurrency does yield better performance
- We use as concurrency measure the number $n$ of simultaneous threads that execute at a given time (be they in the same process or in different processes)
- Still, demand-driven concurrency is not necessarily the best choice
  - For one thing, $n$ can grow unboundedly. Anything that grows without bounds is bad
  - In particular, if we have tons of threads, we end up spending most of the time doing context switching
- Idea no. 1: limit the number of threads that can run simultaneously to a fixed limit $n_{\max}$. (how?)
  - When using threads, we can use a semaphore $h$
    - we initialize $h$ with $n_{\max}$
    - each time we create a thread we wait on $h$
    - each time we return from a thread we post $h$



## Managing Concurrency

- In the general case though, concurrency does yield better performance
- We use as concurrency measure the number $n$ of simultaneous threads that execute at a given time (be they in the same process or in different processes)
- Still, demand-driven concurrency is not necessarily the best choice
  - For one thing, $n$ can grow unboundedly. Anything that grows without bounds is bad
  - In particular, if we have tons of threads, we end up spending most of the time doing context switching
- Idea no. 1: limit the number of threads that can run simultaneously to a fixed limit $n_{\max}$. (how?)



## Preallocation

- Idea no. 2: Since we have a maximum number of threads anyway, we might just as well create all of them at the beginning
  - Threads are preallocated, put to sleep, and woken up as needed by sharing the master socket
  - In other words, we put the call to `accept` inside the child threads
  - A thread is idle when it blocks on the call to `accept`
  - Once a client comes, the quickest idle thread will accept the connection and this will wake it up
  - The other idle threads will continue to block on `accept`
  - The thread just woken up will then handle the client
  - Once the client finishes the interaction, the handling thread will go back to accepting new connections, and will block on `accept` in the case that no clients are craving for a connection
  - After creating the child threads, the master thread does not need to do anything else

## ADVANTAGES OF PREALLOCATION

- The main advantage: We reduce the system overhead, and thus we increase efficiency, response time, you name it
  - Process/thread creation does take some time, so we spend all of this time when the server starts (once a week in the middle of the night maybe) instead of spending bits of it each time a client connects
  - We practically never spend time to destroy processes or threads!
  - Idle threads will be in the Blocked queue, so they will not be dispatched to the CPU, and so they do not add overhead
- Besides reducing overhead, we also set a bound to the maximum number of threads of execution running concurrently (a good thing, remember)