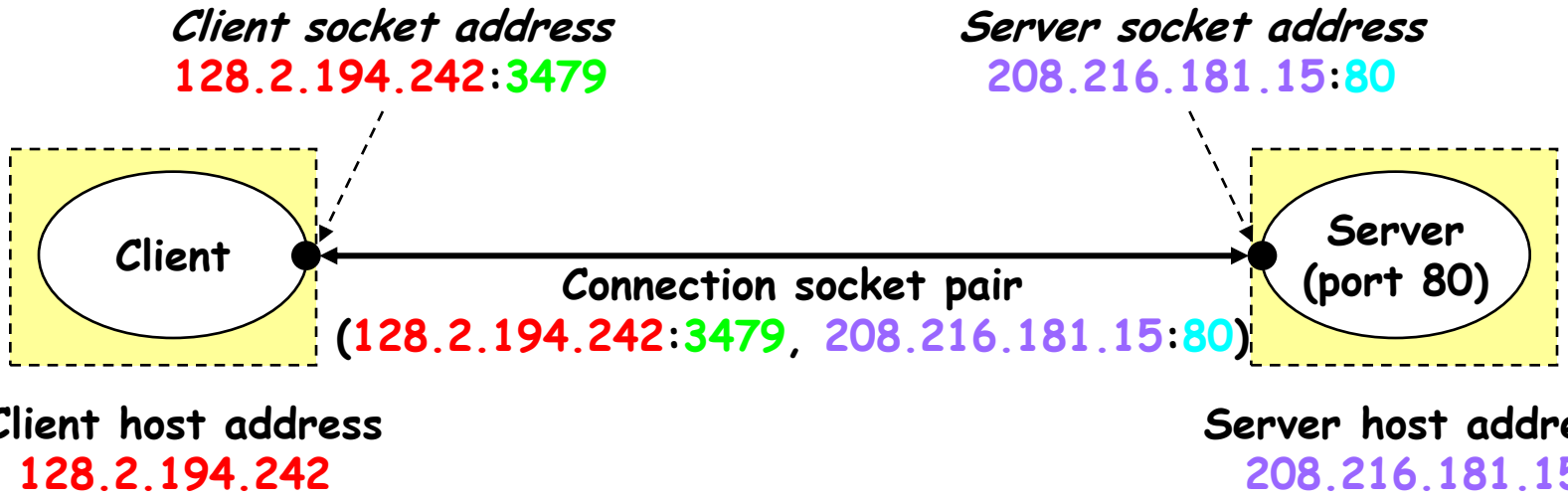
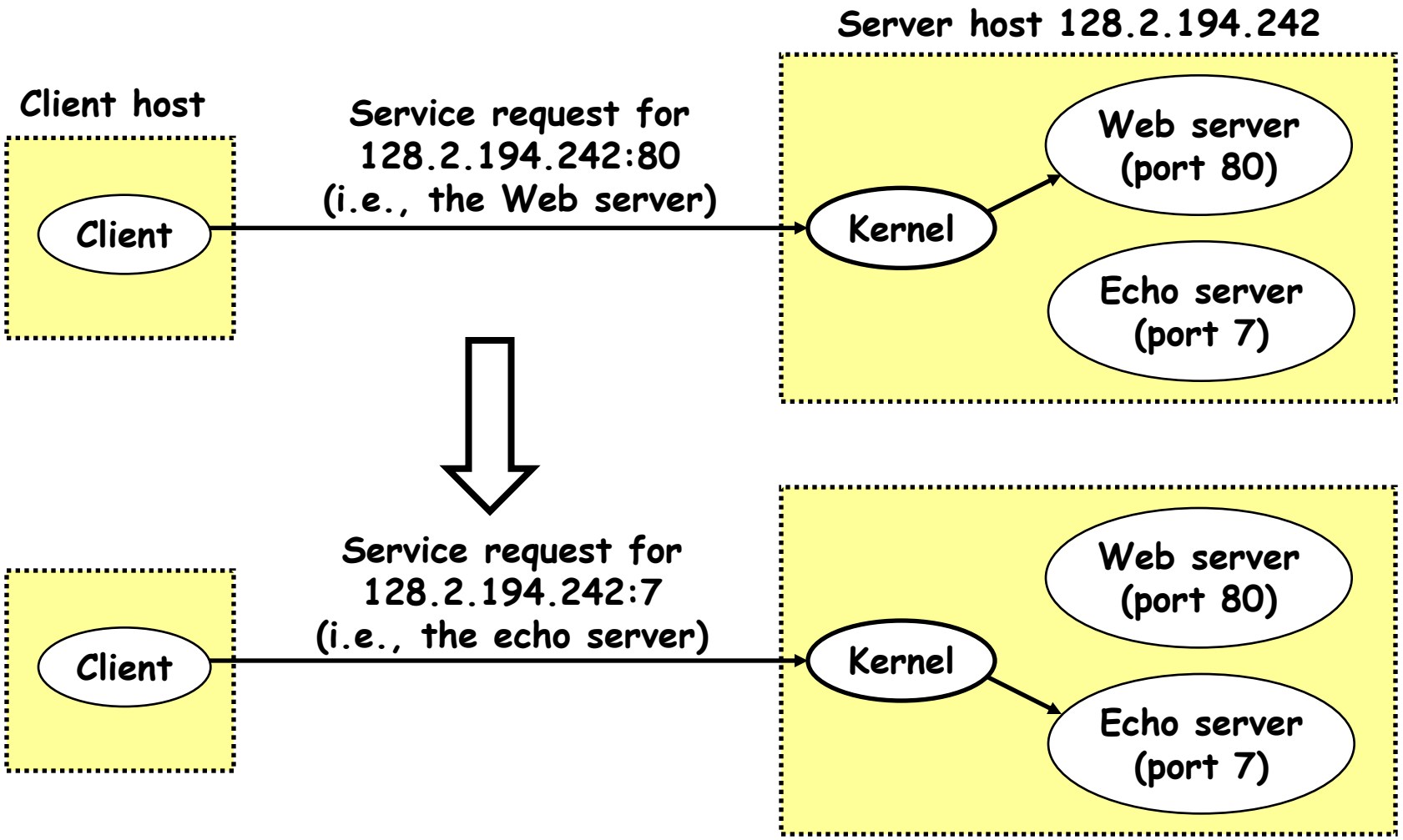
SOCKET PROGRAMMING

Internet Connections (TCP/IP)

- Address the machine on the network
  - By IP address
- Address the process
  - By the "port"-number
- The pair of *IP-address + port* – makes up a "*socket-address*"



Using Ports to Identify Services



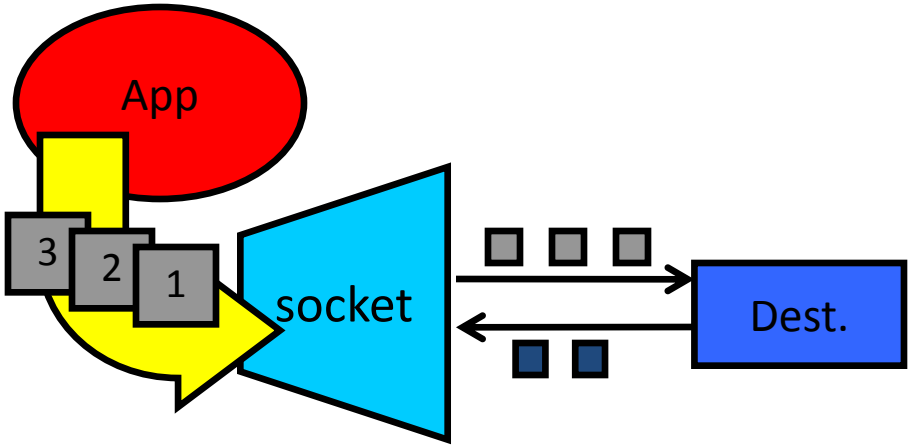
SOCKET

- An interface between application and network.
- To the kernel, a socket is an endpoint of communication.
- To an application, a socket is a file descriptor that lets the application read/write from/to the network.
- Sockets are protocol independent and language independent.
- In Socket terms a connections between two processes in called an association.

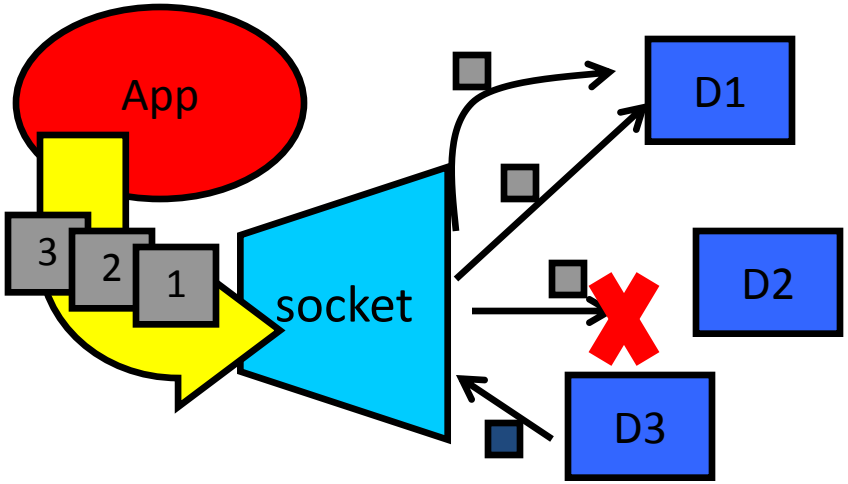
- A process in a local host can be identified by it’s protocol, IP address and port.
- A connection adds the IP address & port of the remote host.
- In UNIX the first 1024 ports for both protocols are called “well known ports” and are defined in the file /etc/services.

Two essential types of sockets

- SOCK_STREAM
  - a.k.a. TCP
  - reliable delivery
  - in-order guaranteed
  - connection-oriented
  - bidirectional



- SOCK_DGRAM
  - a.k.a. UDP
  - unreliable delivery
  - no order guarantees
  - no notion of “connection” – app indicates dest. for each packet
  - can send or receive

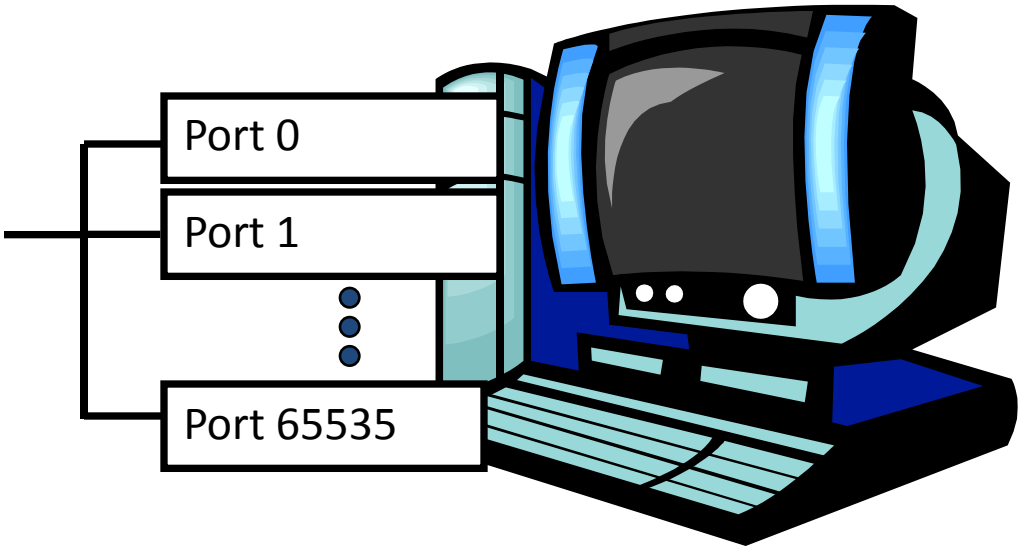


Socket Creation in C: socket

- int sock = socket(domain, type, protocol);
  - sock: socket descriptor, an integer (like a file-handle)
  - domain: integer, communication domain
    - e.g., PF_INET (IPv4 protocol) or AF_INET
  - type: communication type
    - SOCK_STREAM: reliable, 2-way, connection-based service
    - SOCK_DGRAM: unreliable, connectionless,
  - protocol: specifies protocol (see file /etc/protocols for a list of options) - usually set to 0
  - Returns -1 on error
- NOTE: socket call does not specify where data will be coming from, nor where it will be going to – it just creates the interface!

Ports

- Each host has 65,536 ports
- Some ports are *reserved for specific apps*
  - 20,21: FTP
  - 23: Telnet
  - 80: HTTP
  - see RFC 1700 (about 2000 ports are reserved)



- A socket provides an interface to send data to/from the network through a port

The struct sockaddr

- Structure holds socket address info for many types of sockets
- The generic:

```
struct sockaddr {
        u_short sa_family;
        char sa_data[14];
};
```

  - sa_family
    - specifies which address family is being used
    - determines how the remaining 14 bytes are used
    - Typically use AF_INET

- The Internet-specific:

```
struct sockaddr_in {
        short sin_family;
        u_short sin_port;
        struct in_addr sin_addr;
        char sin_zero[8];
};
```

  - sin_family = AF_INET
  - sin_port: port # (1-65535)
  - sin_addr: IP-address
  - sin_zero: used for padding, memset to all 0s

Structs

- struct in_addr {

  unsigned long s_addr; // that's 32-bit long

  };
- Assume we have to store, say, "100.47.59.226", struct sockaddr_in s, use inet_addr()
  - s.sin_addr.s_addr = inet_addr("100.47.59.226");

  inet_addr() returns the address in Network Byte Order
- Use AF_INET in your struct sockaddr_in and PF_INET in your call to socket()

The bind function

- Associates and (can exclusively) reserves a port for use by the socket
- int status = bind(sockid, &addrport, size);
  - status: error status, = -1 if bind failed
  - sockid: integer, socket descriptor
  - addrport: struct sockaddr, is a pointer to a struct sockaddr that contains information about your address, namely, port and IP address
  - size: the size (in bytes) of the addrport structure

Connection Setup (SOCK_STREAM)

- Recall: no connection setup for SOCK_DGRAM
- A connection occurs between two kinds of participants
  - passive: waits for an active participant to request connection
  - active: initiates connection request to passive side

connect()..How to connect to a remote host

- int status = connect(sock, &server_addr, addrlen);
  - status: 0 if successful connect, -1 otherwise
  - sock: integer, socket to be used in connection
  - server_addr : struct sockaddr: address of server to connect to
  - addrlen: integer, sizeof (server_addr)

Connection setup: listen & accept

- Called by passive participant (server)
- int status = listen(sock, queuelen);
  - status: 0 if listening, -1 if error
  - sock: integer, socket file descriptor
  - queuelen: integer, # of active participants that can "wait" for a connection
  - listen is **non-blocking**: returns immediately
  - we need to call bind() before we call listen() or the kernel will have us listening on a random port

accept()

- int s = accept(sock, &addr, &addrlen);
  - s: integer, the new socket (used for data-transfer)
  - sock: integer, the orig. socket (being listened on)
  - addr: struct sockaddr, address of the active participant
  - addrlen: sizeof (struct sockaddr):
    - must be set appropriately before call
    - adjusted by OS upon return
  - accept is **blocking**: waits for connection before returning

accept()…

- Someone will try to connect() to your machine on a port that you are listen()ing on.
- connection will be queued up waiting to be accept()ed.
- You call accept() and you tell it to get the pending connection.
- It’ll return to you a brand new socket file descriptor to use for this single connection
- The original one is still listening on your port and the newly created one is finally ready to send() and recv()

Sending / Receiving Data

- With a connection (SOCK_STREAM):
  - int count = send(sock, &buf, len, flags);
    - count: # bytes transmitted (-1 if error)
    - Sock : socket descriptor you want to send data to
    - buf: char[], buffer to be transmitted
    - len: integer, length of buffer (in bytes) to transmit
    - flags: integer, special options, usually just 0
  - int count = recv(sock, &buf, len, flags);
    - count: # bytes received (-1 if error)
    - Sock : socket descriptor you want to read data from
    - buf: void[], stores received bytes
    - len: # bytes received
    - flags: integer, special options, usually just 0
  - Calls are **blocking** [returns only after data is sent (to socket buf) / received]

Sending / Receiving Data (cont'd)

- Without a connection (SOCK_DGRAM):
  - int count = sendto(sock, &buf, len, flags, &addr, addrlen);
    - count, sock, buf, len, flags: same as send
    - addr: struct sockaddr, address of the destination
    - addrlen: sizeof(addr)
  - int count = recvfrom(sock, &buf, len, flags, &addr, &addrlen);
    - count, sock, buf, len, flags: same as recv
    - name: struct sockaddr, address of the source
    - namelen: sizeof(name): value/result parameter
- Calls are **blocking** [returns only after data is sent (to socket buf) / received]

close

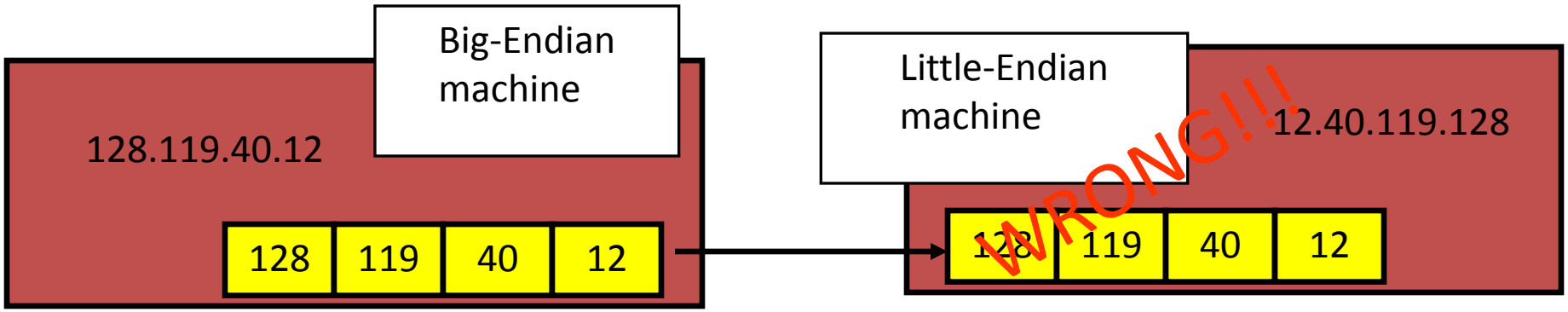
- When finished using a socket, the socket should be closed:
- status = close(sock);
  - status: 0 if successful, -1 if error
  - closes a connection
  - frees up the port used by the socket
  - This will prevent any more reads and writes to the socket

Address and port byte-ordering

- Address and port are stored as integers
  - u_short sin_port; (16 bit)
  - in_addr sin_addr; (32 bit)

```
struct in_addr {
  u_long s_addr;
};
```

- Problem:
  - different machines / OS's use different word orderings
    - little-endian: lower bytes first
    - big-endian: higher bytes first
  - these machines may communicate with one another over the network



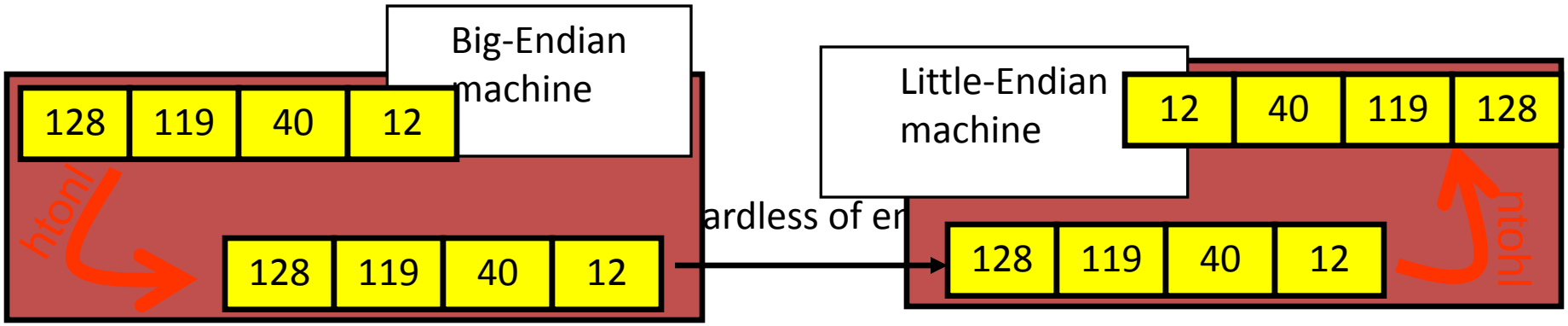
Solution: Network Byte-Ordering

- Defs:
  - 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)

UNIX's byte-ordering funcs

- u_long htonl(u_long x);
- u_short htons(u_short x);
- u_long ntohl(u_long x);
- u_short ntohs(u_short x);

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



Final Thoughts

- Make sure to #include the header files that define used functions
- Check out http://beej.us/guide/bgnet/output/print/bgnet_A4.pdf