University of Calgary – CPSC 441

Introduction to Socket Programming with C

What is a Socket?

- A socket is an interface between the application and the network (the lower levels of the protocol stack)
  - The application creates a socket
  - The socket type dictates the style of communication
    - reliable vs. best effort
    - connection-oriented vs. connectionless
- Once a socket is setup the application can:
  - pass data to the socket for network transmission
  - receive data from the socket (transmitted through the network, sent by some other host)

From: Dan Rubenstein's slides (http://www.cs.columbia.edu/~danr/courses/6761/Summer03/intro/6761-1b-sockets.ppt)

Most Popular Types of Sockets

TCP Socket

- Type: **`SOCK_STREAM`**
- reliable delivery
- in-order guaranteed
- connection-oriented
- bidirectional

UDP Socket

- Type: **`SOCK_DGRAM`**
- unreliable delivery
- no order guarantees
- no notion of "connection" – app indicates destination for each packet
- can send or receive

We focus on TCP

Socket Creation in C

```
int s = socket(domain, type, protocol);
```

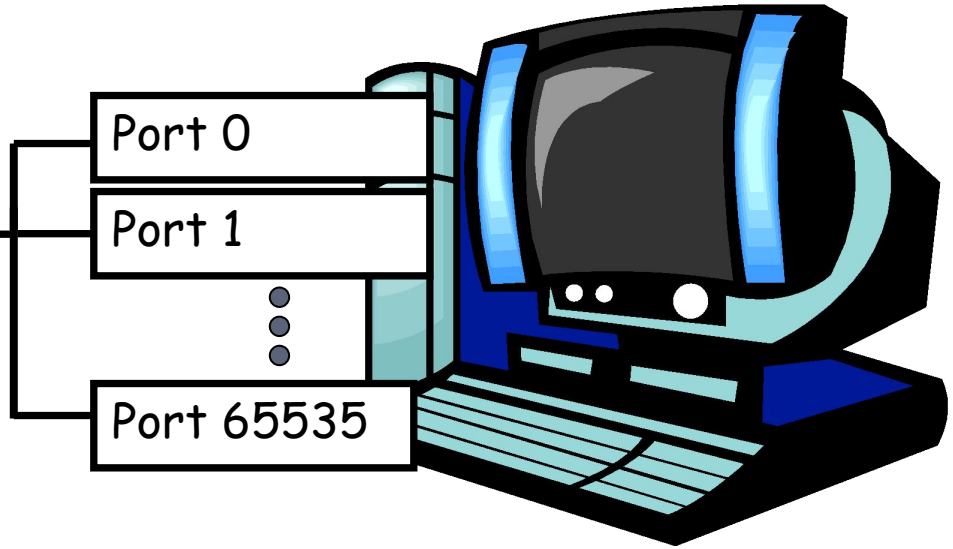
- s: socket descriptor, an integer (like a file-handle)
- domain: integer, communication domain
  - e.g., **`PF_INET`** (IPv4 protocol) – typically used
- type: communication type
  - **`SOCK_STREAM`**: reliable, 2-way, connection-based service
  - **`SOCK_DGRAM`**: unreliable, connectionless,
  - other values: need root permission, rarely used, or obsolete
- protocol: specifies protocol (see file /etc/protocols for a list of options) - usually set to 0

NOTE: **`socket`** call does not specify where data will be coming from, nor where it will be going to; it just creates the interface.

From: Dan Rubenstein's slides (http://www.cs.columbia.edu/~danr/courses/6761/Summer03/intro/6761-1b-sockets.ppt)

Ports

- Each host machine has an IP address (or more!)
- Each host has 65,536 ports (2^2)
- 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

From: Dan Rubenstein's slides (http://www.cs.columbia.edu/~danr/courses/6761/Summer03/intro/6761-1b-sockets.ppt)

Addresses, Ports and Sockets

- Like apartments and mailboxes
  - You are the application
  - Your apartment building address is the address
  - Your mailbox is the port
  - The post-office is the network
  - The socket is the key that gives you access to the right mailbox (one difference: assume outgoing mail is placed by you in your mailbox)
- Q: How do you choose which port a socket connects to?

From: Dan Rubenstein's slides (http://www.cs.columbia.edu/~danr/courses/6761/Summer03/intro/6761-1b-sockets.ppt)

The `Bind` Function

- 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`**, the (IP) address and port of the machine (address usually set to **`INADDR_ANY`** – chooses a local address)
  - size: the size (in bytes) of the addrport structure
- Q: **`bind`** can be skipped for both types of sockets. When and why?

From: Dan Rubenstein's slides (http://www.cs.columbia.edu/~danr/courses/6761/Summer03/intro/6761-1b-sockets.ppt)

On the Connecting End

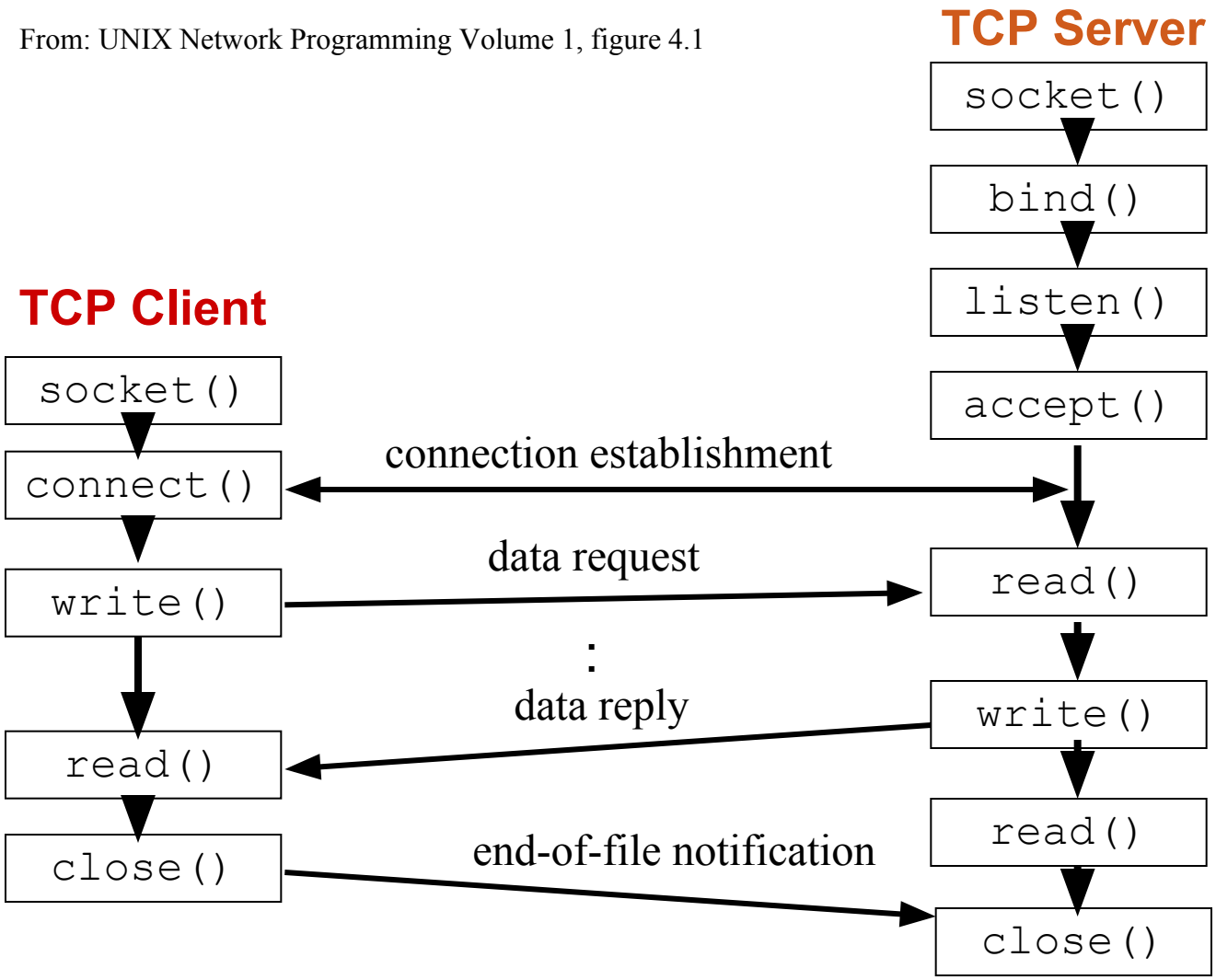
- When connecting to another host (i.e., connecting end is the client and the receiving end is the server), the OS automatically assigns a free port for the outgoing connection.
- During connection setup, receiving end is informed of port)
- You can however bind to a specific port if need be.

Connection Setup

- A connection occurs between two ends
  - Server: waits for an active participant to request connection
  - Client: initiates connection request to passive side
- Once connection is established, server and client ends are “similar”
  - both can send & receive data
  - either can terminate the connection

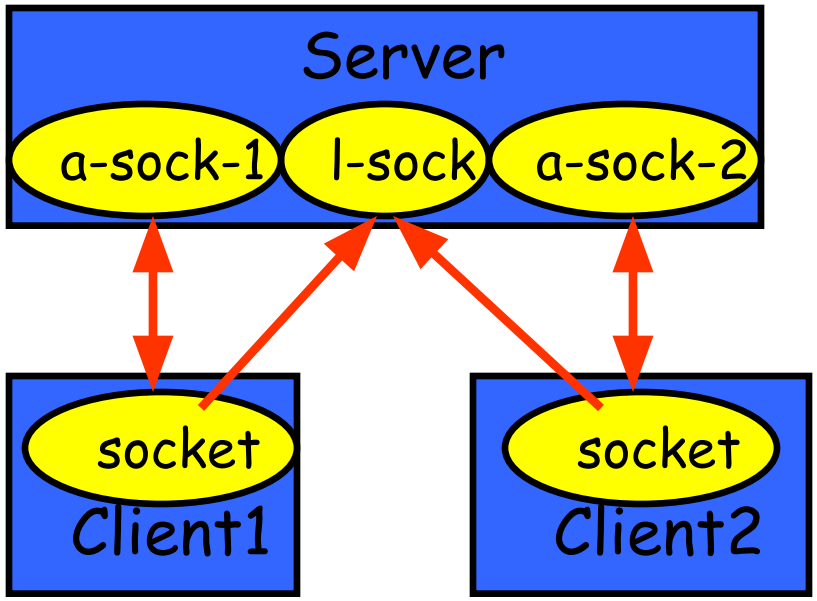
Server and Clients

From: UNIX Network Programming Volume 1, figure 4.1



Connection Setup Steps

- Client end:
  - step 2: request & establish connection
  - step 4: send/recv
- Server end:
  - step 1: listen (for incoming requests)
  - step 3: accept (a request)
  - step 4: send/receive



- The accepted connection is on a new socket
- The old socket continues to listen for other active participants

From: Dan Rubenstein's slides (http://www.cs.columbia.edu/~danr/courses/6761/Summer03/intro/6761-1b-sockets.ppt)

Server Socket: Listen & Accept

Called on server side:

- `int status = listen(sock, queuelen);`
  - status: 0 if listening, -1 if error
  - sock: integer, socket descriptor
  - queuelen: integer, # of active participants that can "wait" for a connection
  - listen is **non-blocking**: returns immediately
- `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(addr): value/result parameter
  - must be set appropriately before call
  - adjusted by OS upon return
  - accept is **blocking**: waits for connection before returning

From: Dan Rubenstein's slides (http://www.cs.columbia.edu/~danr/courses/6761/Summer03/intro/6761-1b-sockets.ppt)

Connect

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

From: Dan Rubenstein's slides (http://www.cs.columbia.edu/~danr/courses/6761/Summer03/intro/6761-1b-sockets.ppt)

Sending / Receiving Data

- `int count = send(sock, &buf, len, flags);`
  - count: # bytes transmitted (-1 if error)
  - buf: void*, 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)
  - buf: void*, stores received bytes
  - len: # bytes received
  - flags: integer, special options, usually just 0
- Calls are **blocking**

From: Dan Rubenstein's slides (http://www.cs.columbia.edu/~danr/courses/6761/Summer03/intro/6761-1b-sockets.ppt)

Close

- When finished using a socket, the socket should be closed.
- `status = close(s);`
  - status: 0 if successful, -1 if error
  - s: the file descriptor (socket being closed)
- Closing a socket
  - closes a connection
  - frees up the port used by the socket

From: Dan Rubenstein's slides (http://www.cs.columbia.edu/~danr/courses/6761/Summer03/intro/6761-1b-sockets.ppt)

The struct sockaddr

- The struct to store the Internet address of a host:

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

- sin_family
  - Specifies the address family
  - E.g. **AF_INET**
- sin_port
  - Specifies the port number (0-65535)
- sin_addr
  - Specifies the IP address
- sin_zero
  - unused!

From: Dan Rubenstein's slides (http://www.cs.columbia.edu/~danr/courses/6761/Summer03/intro/6761-1b-sockets.ppt)

Example

```
struct sockaddr_in server;                    // definition
memset(&server, 0, sizeof(server));           // init to 0
server.sin_family = AF_INET;                  // address family
server.sin_port = htons(MYPORTNUM);           // port
server.sin_addr.s_addr = htonl(INADDR_ANY); // address
```

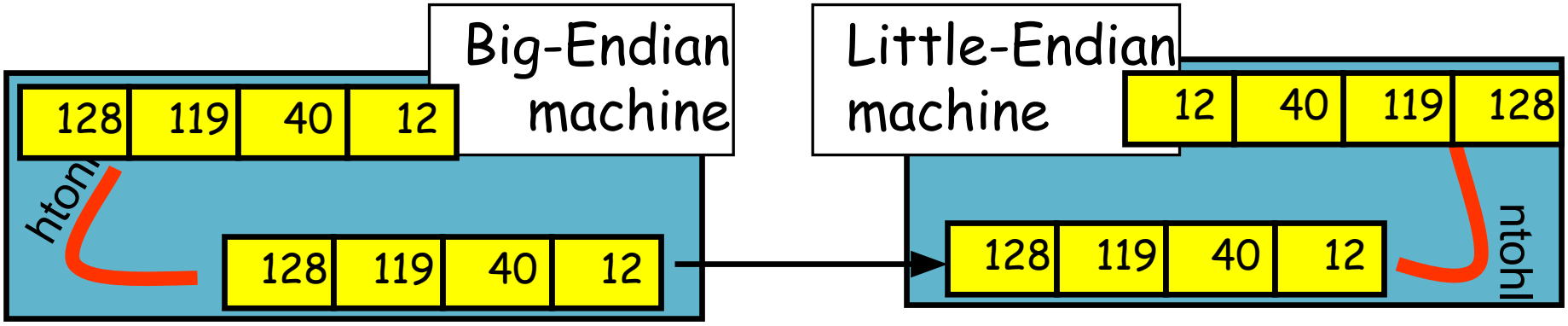
- Host Byte-Ordering: the byte ordering used by a host (big-endian or little-endian)
- Network Byte-Ordering: the byte ordering used by the network – always big-endian
- Any words sent through the network should be converted to Network Byte-Order prior to transmission (and back to Host Byte-Order once received)

From: Dan Rubenstein's slides (http://www.cs.columbia.edu/~danr/courses/6761/Summer03/intro/6761-1b-sockets.ppt)

Network Byte-Ordering

- `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



From: Dan Rubenstein's slides (http://www.cs.columbia.edu/~danr/courses/6761/Summer03/intro/6761-1b-sockets.ppt)

Tips (1/2)

- Sometimes, an ungraceful exit from a program (e.g., ctrl-c) does not properly free up a port
- Eventually (after a few minutes), the port will be freed
- You can kill the process, or to reduce the likelihood of this problem, include the following code:
  - In header include:

```
#include <signal.h>
void cleanExit(){exit(0);}
```

  - In socket code add:

```
signal(SIGTERM, cleanExit);
signal(SIGINT, cleanExit);
```

From: Dan Rubenstein's slides (http://www.cs.columbia.edu/~danr/courses/6761/Summer03/intro/6761-1b-sockets.ppt)

Tips (2/2)

Q: How to find the IP address of the machine my server program is running on?

- Use 127.0.0.1 or localhost for accessing a server running on your local machine.
- For a remote server running Linux use the bash shell command: **`ifconfig`**
- For Windows, use cmd to invoke: **`ipconfig`**

From: Dan Rubenstein's slides (http://www.cs.columbia.edu/~danr/courses/6761/Summer03/intro/6761-1b-sockets.ppt)

Let's Write Some Code

- Sample socket program:
  1. Echo server: echo's what it receives back to client
  2. Client/server example

References

These are good references for further study of Socket programming with C:

- Beej's Guide to Network Programming Using Internet Sockets
  http://beej.us/guide/bgnet/output/html/multipage/index.html
- Search the specification for the function you need to use for more info, or check the man pages.
- Dan Rubenstein's lecture on Socket "Programming":
  http://www.cs.columbia.edu/~danr/courses/6761/Summer03/intro/6761-1b-sockets.ppt

From: Dan Rubenstein's slides (http://www.cs.columbia.edu/~danr/courses/6761/Summer03/intro/6761-1b-sockets.ppt)

Tips for Assignment 1