Sockets

Application Programming Interface

Recap

Past Topics

- Overview of networking and layered architecture
- Wireshark packet sniffer and Scapy packet manipulation
- Wired LAN, Wireless LANs, VLANs
- IPv4, IPv6 ARP, ICMP
- UDP
- DHCP

Today's Topics

- Sockets & Socket Programming

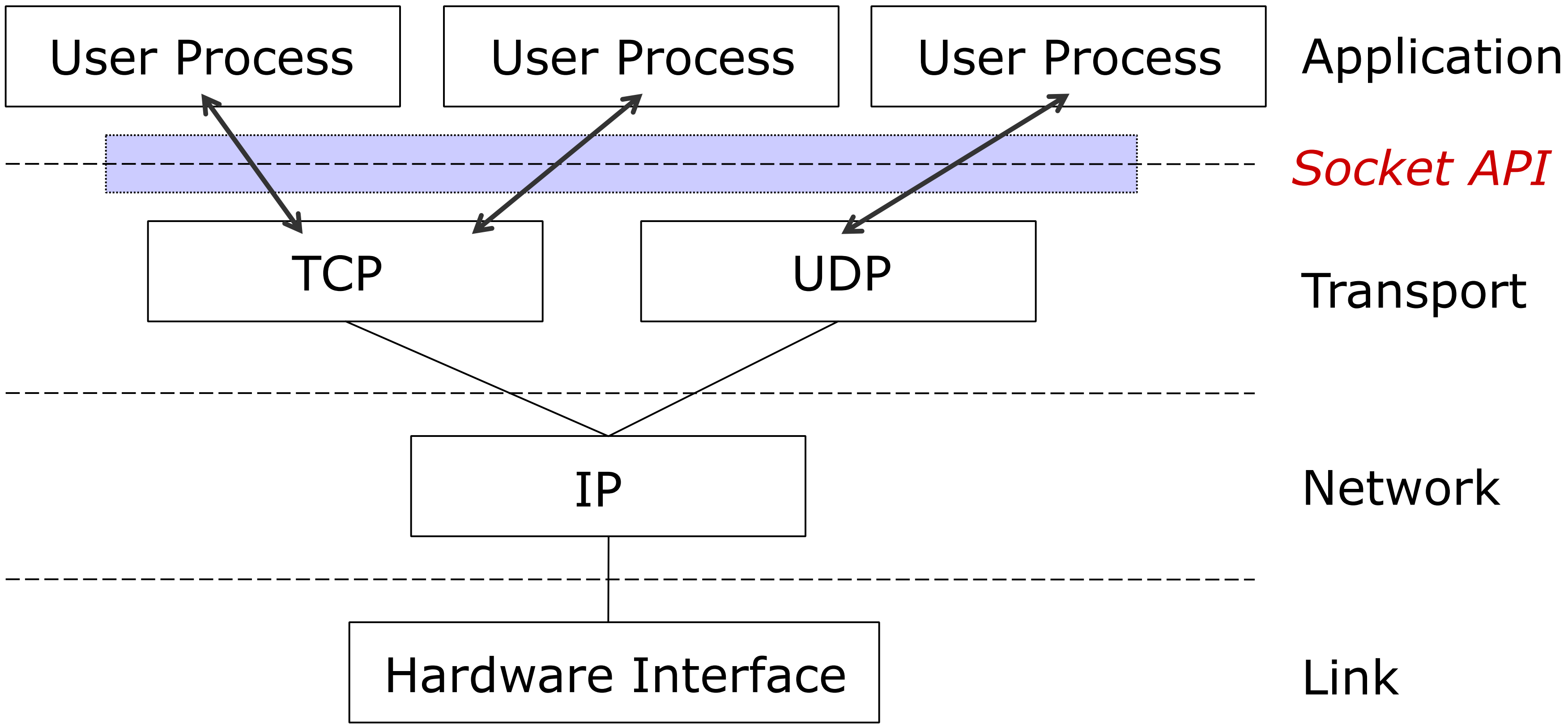
Why Do We Have Sockets?

- Challenge – **Inter-process communication**
- A **process** is an independent program running on a host
  - Separate memory space
- How do processes communicate with other processes
  - On the same host?
  - On different hosts?
- Send **messages** between each other

What is a Socket?

- An interface between process (application) and network
  - The application creates a socket
  - The socket *type* dictates the style of communication
    - Reliable vs. best effort
    - Connection-oriented vs. connectionless
- Once configured the application can
  - Pass data to the socket for network transmission
  - Receive data from the socket (transmitted through the network by some other host)

Sockets and the TCP/IP Suite



The Socket API

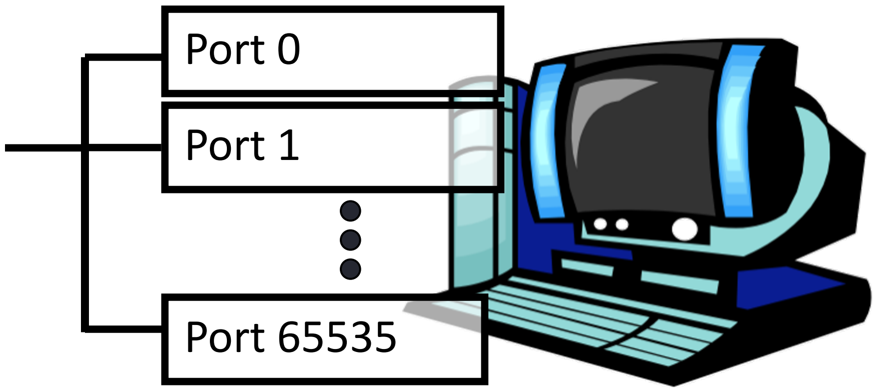
- A collection of system calls to write a networking program at user-level
- **Originally created in C**
  - Introduced in BSD4.1 UNIX, 1983
- **Python Socket API closely follows behavior**
- API is similar to Unix file I/O in many respects: open, close, read, write.
  - Data written into socket on one host can be read out of socket on other host
  - Difference: networking has notion of client and server

Addressing Processes

- To receive messages, each process on a host must have an **identifier**
  - IP addresses are unique
  - **Is this sufficient?**
- No, there can be thousands of processes running on a single machine (with 1 IP address)
- Identifier must include
  - IP address
  - **and** port number (example: 80 for web)

Ports

- Each host has 65,536 ports
- Some ports are *reserved for specific apps*
  - FTP (20, 21), Telnet (23), HTTP (80), etc…



- Outgoing ports (on clients) can be dynamically assigned by OS in upper region (above 49,152) – called **ephemeral ports**
- See http://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers

Socket Parameters

- A socket connection has 5 general parameters:
  - The protocol
    - Example: TCP, UDP etc.
  - The local and remote IP address
    - Example: 171.64.64.64
  - The local and remote port number
    - Need to determine to which process packets are delivered
    - Some ports are reserved (e.g. 80 for HTTP)
    - Root access required to listen on port numbers below 1024

Internet Transport Protocols

TCP Service

- Connection-oriented
  - Setup required between client and server processes
- Reliable transport between sending and receiving process
- Flow control
  - Sender won't overwhelm receiver
- Congestion control
  - Throttle sender when network overloaded
- Does not provide
  - Timing, minimum throughput guarantees, security

UDP Service

- Unreliable data transfer between sending and receiving process
- Does not provide
  - Connection setup
  - Reliability
  - Flow control
  - Congestion control
  - Timing
  - Throughput guarantee
  - Security

Why bother with UDP then?

Application-Layer Protocol

- Sockets just allow us to send raw messages between processes on different hosts
  - Transport service takes care of moving the data
- **What** exactly is sent is up to the application
  - An application-layer protocol
  - HTTP, IMAP, Skype, etc…

Application-Layer Protocol

- Both the client and server speaking the protocol must agree on
  - Types of messages exchanged
    - e.g., request, response
  - Message syntax
    - What fields are in messages
    - How fields are delineated
  - Message semantics
    - Meaning of information in fields
  - Rules for when and how processes send and respond to messages

Client-Server Architecture

- Using Socket API, you can program network applications with a client-server architecture
- Client-server architecture:
  - Server-side program
    - Always online
    - Passive: listening to be contacted
  - Client-side program
    - Can be offline, at times
    - Active: initiates the connection to the server
- The connection is always between a client and a server
  - Two server-side programs cannot establish a connection
  - Two client-side programs cannot establish a connection

Socket Programming Basics

- Server must be running before client can send anything to it
- Server must have a socket through which it receives and sends messages
- Similarly client needs a socket
- Socket is locally identified with a port number
- Client needs to know server IP address and socket port number
  - **How do we find this?**

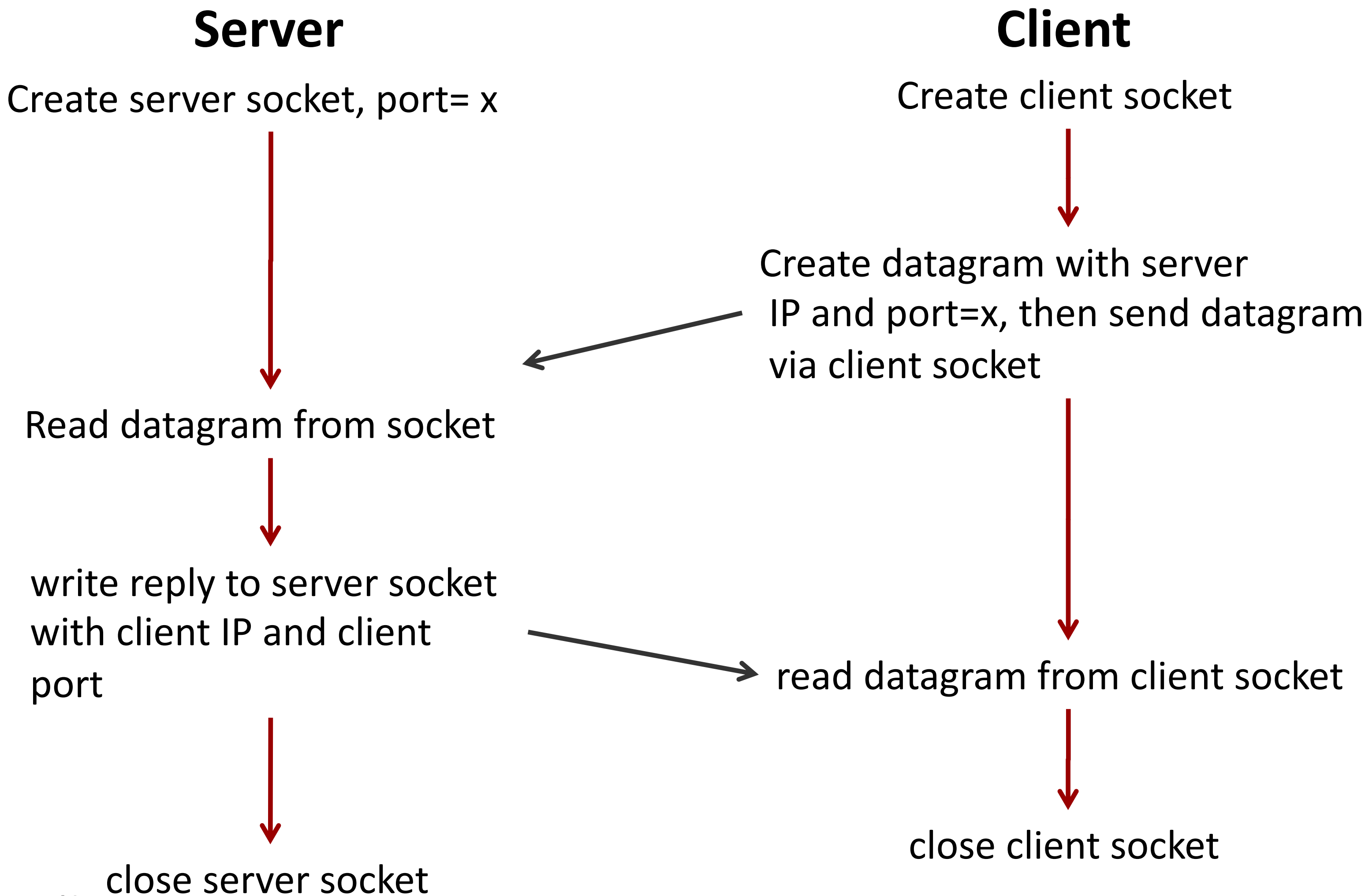
Socket Programming with UDP

- UDP: no "connection" between client and server
  - No handshaking
  - Sender explicitly attaches IP address and port of destination to each message
  - OS attaches IP address and port of sending socket to each segment
  - Server can extract IP address, port of sender from received segment

application viewpoint

UDP provides unreliable transfer of groups of bytes ("datagrams") between client and server

Client/Server Socket Interaction with UDP



UDP Question

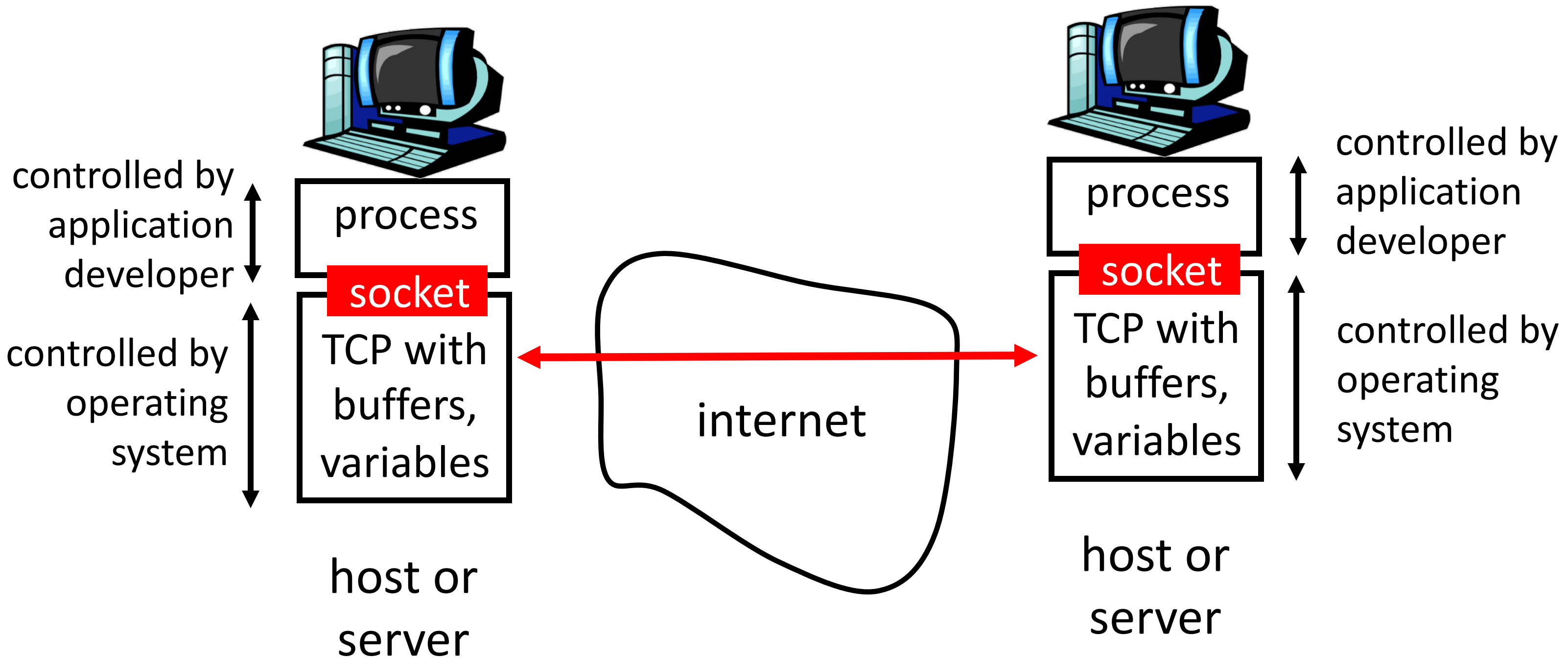
- **Can the client send a segment to server without knowing the server's IP address and port number?**
- Could use **broadcast IP address** of the subnet to get around lack of IP address knowledge…
- No way to avoid knowing port number…

UDP Observation

- Each UDP message is self-contained and complete
- Each time you read from a UDP socket, you get a complete message as sent by the sender
  - *That is, assuming it wasn't lost in transit!*
- Think of UDP sockets as putting a stamp on a letter and sticking it in the mail

Socket Programming with TCP

TCP service: reliable transfer of **bytes** from one process to another



Socket Programming with TCP

Client must contact server

- Server process must first be running
- Server must have created socket (door) that welcomes client's contact

Client contacts server by:

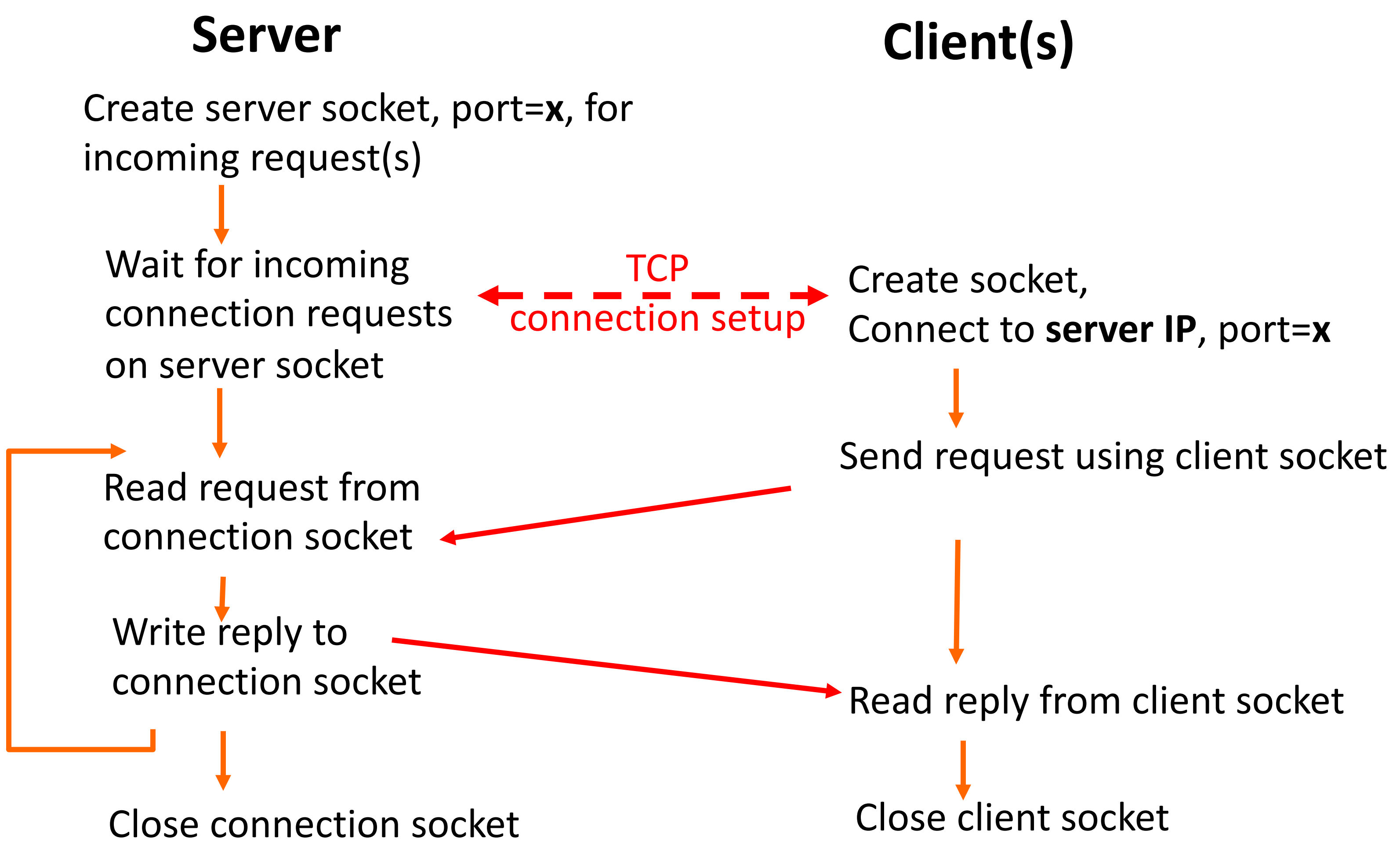
- Creating client-local TCP socket
- Specifying IP address, port number of server process
- When client creates socket: client TCP establishes connection to server TCP

- When contacted by client, server TCP creates new socket for server process to communicate with client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients

application viewpoint

TCP provides reliable, in-order transfer of bytes ("pipe") between client and server

Client/Server Socket Interaction with TCP



What is a Stream?

- A **stream** is a sequence of characters that flow into or out of a process.
- An **input stream** is attached to some input source for the process, e.g., keyboard or socket.
- An **output stream** is attached to an output source, e.g., monitor or socket.

TCP Observations

- TCP sockets are **stream** based
  - At the receiver, each read on a TCP socket is **not** guaranteed to produce the same number of bytes as were sent by the transmitter
  - All you know is that you'll get the **next** set of bytes
    - Keep reading, and eventually you'll get them all
  - Your **application** must have some way to separate a stream of bytes into discrete messages
- Server has two types of sockets
  - One that listens for incoming connections
  - One on a per-client basis after a connection is opened

Sockets for Servers

Server Program Operation

- Let's take a simple **connection-oriented (TCP)** <u>**server**</u> first

1.	`socket()`	create the socket descriptor
2.	`bind()`	associate the local address
3.	`listen()`	wait for incoming connections from clients
4.	`accept()`	accept incoming connection
5.	`send(),recv()`	communicate with client
6.	`close()`	close the socket descriptor

Server – socket()

- Let’s create the server socket now!
- Function prototype
  - `descriptor = socket(family, type)`
  - Family: `AF_INET` (IPv4) or `AF_INET6` (IPv6)
  - Type: `SOCK_STREAM` (TCP) or `SOCK_DGRAM` (UDP)
- Returns a socket descriptor (class)
- Raises an exception (Socket.Error) if error occurs

Sever – bind()

- `bind()` associates the server socket with a specific port on the local machine
- Function prototype
  - `bind(address)`
- Address format
  - IPv4: `(host, port)`
  - IPv6: `(host, port, flowinfo, scopeid)`
- Raises an exception (Socket.Error) if error occurs

Server – listen()

- **`listen()`** listens for incoming messages on the socket
- Function prototype
  - `listen(backlog)`
  - backlog is number of incoming connections on queue (probably limited by OS to ~20)
- Raises an exception (Socket.Error) if error occurs

Server – accept()

- `accept()` acknowledges an incoming connection
- Function prototype
  - `(new_socket, address) = accept();`
- Raises an exception (Socket.Error) if error occurs

Server – accept()

- **Wait, what is happening here?**
- I give `accept()`:
  - The socket descriptor for the server
- `accept()` runs and gives me
  - A **new** socket descriptor that connects to the client
  - Details on the incoming socket (the IP and port of host that is connecting to me)

Server Operation

- The socket returned by `accept()` is not the same socket that the server was listening on!
- A new socket, bound to a random port, is created to handle the connection
- New socket should be closed when done with communication
- Initial socket remains open and can still accept more connections
  - The initial socket never does any application-level communication. It just serves to generate new sockets

Server Recap Thus Far

- Someone from far far away will try to **`connect()`** to your machine on a port that you are **`listen()`**ing on.
- Their connection will be queued up waiting to be **`accept()`**ed
- You call **`accept()`** and you tell it to get the pending connection
- **`accept()`** will return to you a brand new socket file descriptor to use for this single connection!
- You now have two socket file descriptors for the price of one!
  - The original one is still listening for more new connections
  - The newly created one is finally ready to **`send()`** and **`recv()`**

send() and recv()

- Send and receive data on connected, streaming sockets (i.e. TCP)
  - We have different functions for unconnected / UDP sockets: **sendto()** and **recvfrom()**
- Function prototypes
  - `bytes_sent = send(bytes, flags);`
    - bytes is the data you want to send
  - `buffer = recv(buf_size, flags);`
    - buffer is where you want the data to be copied to
    - buf_size is the size of the buffer

Pitsfalls

- `send()` and `recv()` are stream-oriented
  - Your messages are not independent, they're part of the first-in, first-out stream
- `send()` and `recv()` may only **partially succeed**
  - `send()` might only send 256 out of 512 bytes you requested
  - `recv()` might only fill your 4kB buffer with 1kB of data
- **You** (the poor, overworked programmer) are responsible for repeatedly calling `send()` and `recv()` until **all** your data is transferred
  - Look at `sendall()` to make sending easier…

close()

- We’re finished
- Function prototype:
  - `close()`

Server Functions – Recap

- **What does `socket()` do?**
  - Create the socket descriptor
- **What does `bind()` do?**
  - Assigns a local address/port to the socket
- **What does `listen()` do?**
  - Configures socket to accept incoming connections
- **What does `accept()` do?**
  - Accepts incoming connection (will block until connection)
- **What do `send()/recv()` do?**
  - Communicate with client
- **What does `close()` do?**
  - Close the socket descriptor

Send/Recv Pitfalls - Recap

- **What is happening in these TCP socket scenarios?**
  - "My client program sent 100 bytes, but the server program only got 50."
  - "My client program sent several small packets, but the server program received one large packet."
- Ans: TCP is a **stream protocol**
  - The sender or receiver (or both!) can segment and recombine the stream at arbitrary locations

http://tangentsoft.net/wskfaq/articles/effective-tcp.html
(Good article to read!)

Send/Recv Pitfalls - Recap

- **"How can I find out how many bytes are waiting on a given socket, so I can set up a receive buffer for the size of the packet?"**
  - **You don't!** Declare a reasonable fixed size buffer when your program starts (say, 32kB) and always receive data *into* that buffer
    - Return value of recv() is the number of bytes saved into the buffer
  - Then, **copy data out of your buffer** into the rest of your program as needed

http://tangentsoft.net/wskfaq/articles/effective-tcp.html
(*Good article to read!*)

Return Values – Recap

- **Why is it important to check for exceptions after every single socket function?**
  - Python will catch the exception and exit automatically
  - In C, however, there are no exceptions and the program will just blindly continue on!

http://tangentsoft.net/wskfaq/articles/effective-tcp.html
(*Good article to read!*)

Sockets for Clients

Client Program Operation

- Let's look at a simple **connection-oriented (TCP)** **<u>client</u>** now
  - We don't need bind(), listen(), or accept()!

1. `socket()` create the socket descriptor
2. `connect()` connect to the remote server
3. `send(),recv()` communicate with the server
4. `close()` end communication by closing socket descriptor

Client – socket()

- A client can use socket() just like a server does to create a new socket
- Function prototype
  - `descriptor = socket(family, type)`
  - Family: `AF_INET` (IPv4) or `AF_INET6` (IPv6)
  - Type: `SOCK_STREAM` (TCP) or `SOCK_DGRAM` (UDP)
- Returns a socket descriptor (class)
- Raises an exception (Socket.Error) if error occurs

Client – connect()

- Now that we have a socket on the client, connect that socket to a remote system (where a server is listening…)
- Function prototype
  - `connect(address)`
- Address format
  - IPv4: `(host, port)` where host could be "www.google.com" or IP address
  - IPv6: `(host, port, flowinfo, scopeid)`
- Raises an exception (Socket.Error) if error occurs

Client – send()/recv()/close()

- After that, it's all the same
  - `send()` data
  - `recv()` data
  - `close()` the socket when finished

Endianness

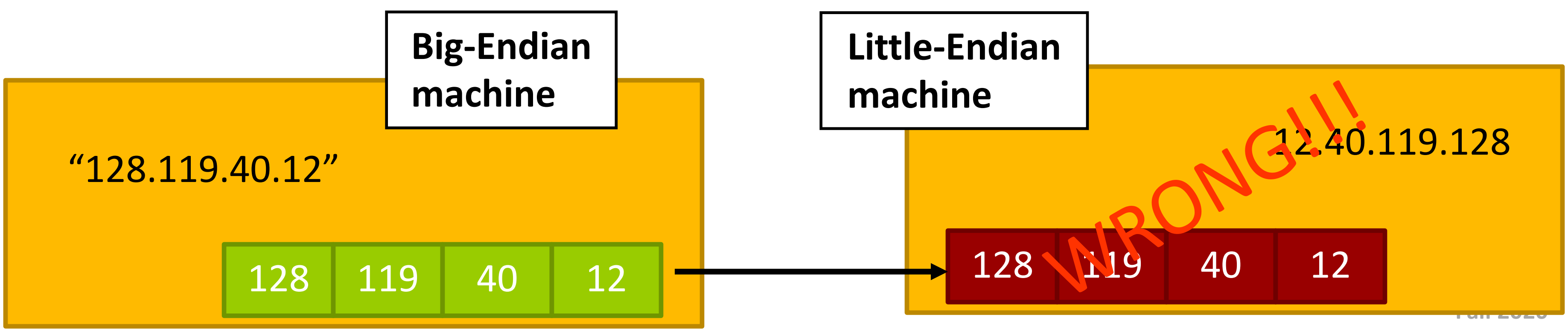
Endianness

- **What is a little endian computer system?**
  - Little-endian: lower bytes come first (stored in lower memory addresses)
- **What is a big endian computer system?**
  - Higher bytes come first

Gulliver's Travels

Address and Port byte-ordering

- Address and port are stored as integers in packet headers
  - Port: 16 bit integer
  - IPv4 address: 32 bit integer
  - IPv6 address: 128 bit integer
- Problem:
  - Different machines / OS's order bytes differently in a word!
  - These machines may communicate with one another over the network



Solution: Network Byte-Ordering

- 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)

Network Byte-Ordering

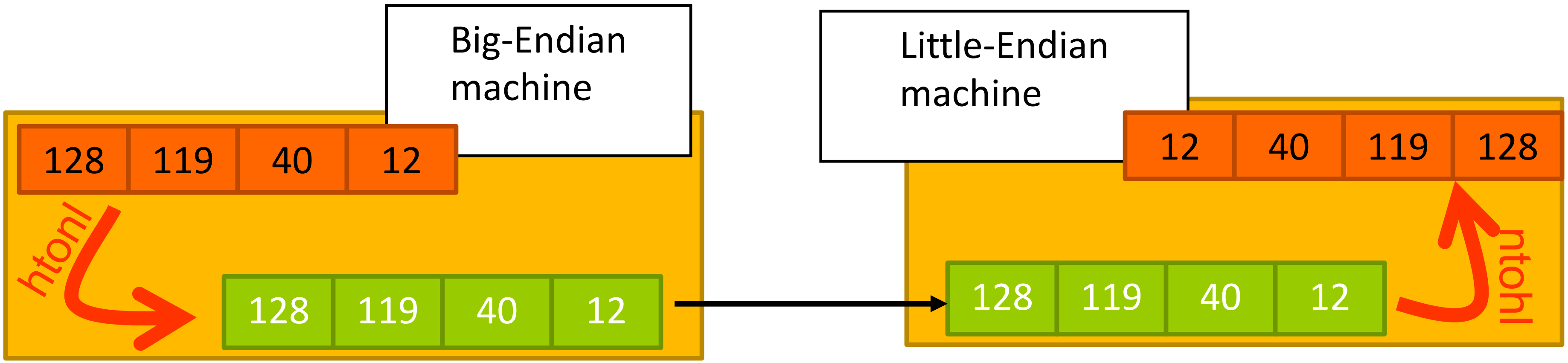
- **Should the socket perform the endianness conversion automatically?**
  - **No –** Not all data needs to be flipped
  - Imagine a stream of characters…
- **Given big-endian machines don't need conversion routines and little-endian machines do, how do we avoid writing two versions of code?**

Byte-ordering Functions

```
y = htonl(x); # 32 bits
x = ntohl(y);
```

```
y = htons(x);  # 16 bits
x = ntohs(y);
```

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



- Same code will work regardless of endian-ness of the two machines

Byte-ordering Functions

- **htonl**
  - **H**ost **to N**etwork Order – **L**ong (32 bits)
- **htons**
  - **H**ost to **N**etwork Order – **S**hort (16 bits)
- **ntohl**
  - **N**etwork **to H**ost Order – **L**ong (32 bits)
- **ntohs**
  - **N**etwork to **H**ost Order – **S**hort (16 bits)

Closing Thoughts

Recap

- Today we discussed
  - Transport layer services
  - Socket API in Python
  - Endianness

Next Class

- HTTP
  - *And then Project 3 (Web server)!*