Google Cluster Computing Faculty Training Workshop

Module II: Student Background Knowledge

Background Topics

- Programming Languages
- Systems:
  - Operating Systems
  - File Systems
  - Networking
- Databases

Programming Languages

- MapReduce is based on functional programming *map* and *fold*
- FP is taught in one quarter, but not reinforced
  - “Crash course” necessary
  - Worksheets to pose short problems in terms of map and fold
  - Immutable data a key concept

Multithreaded programming

- Taught in OS course at Washington
  - Not a prerequisite!
- Students need to understand multiple copies of same method running in parallel

File Systems

- Necessary to understand GFS
- Comparison to NFS, other distributed file systems relevant

Networking

- TCP/IP
- Concepts of “connection,” network splits, other failure modes
- Bandwidth issues

Other Systems Topics

- Process Scheduling
- Synchronization
- Memory coherency

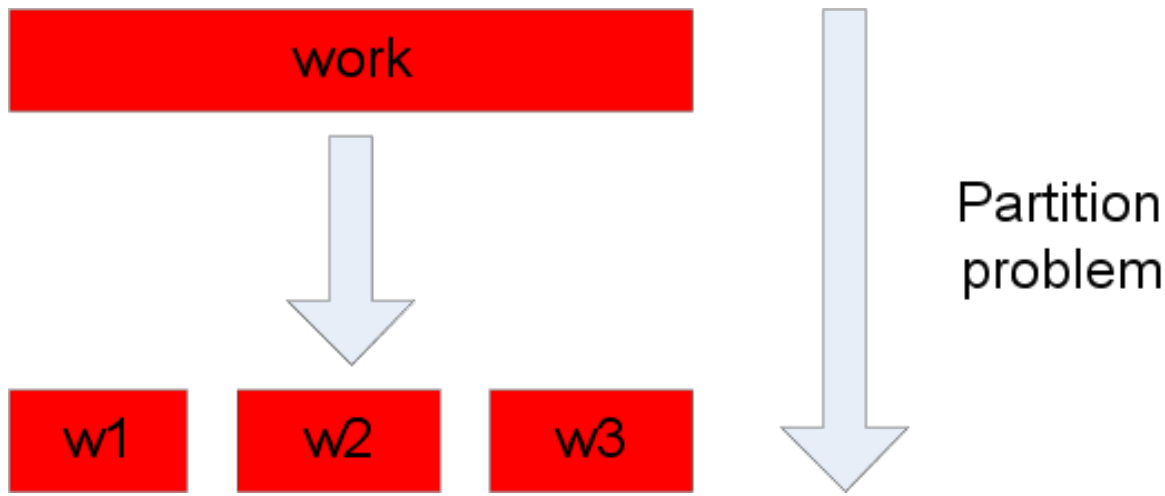
Databases

- Concept of shared consistency model
- Consensus
- ACID characteristics
  - Journaling
  - Multi-phase commit processes

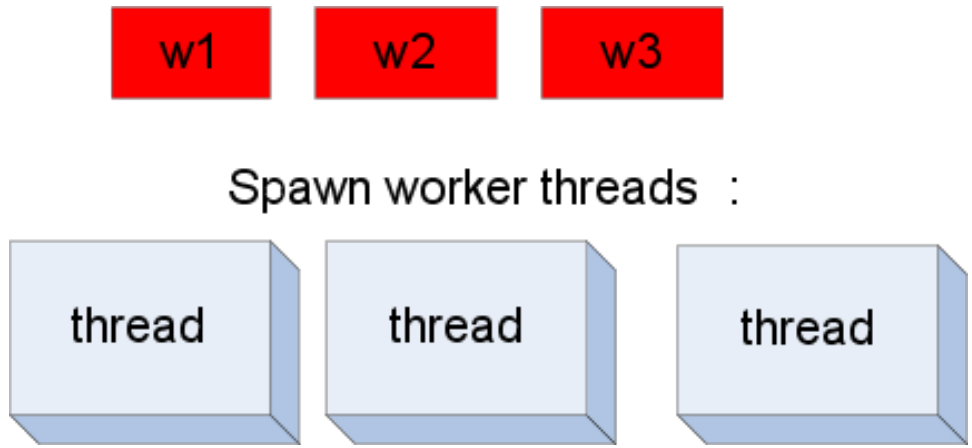
Parallelization & Synchronization

Parallelization Idea

- Parallelization is “easy” if processing can be cleanly split into n units:

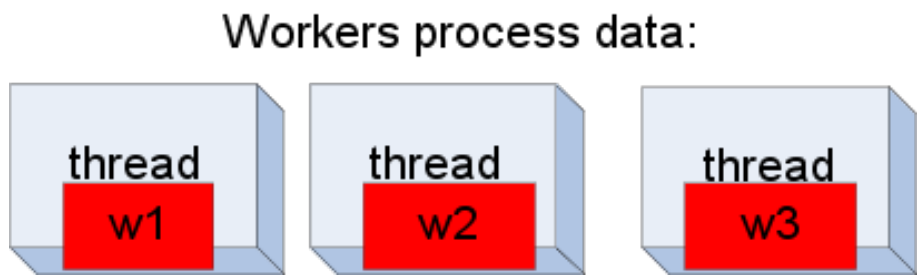


Parallelization Idea (2)

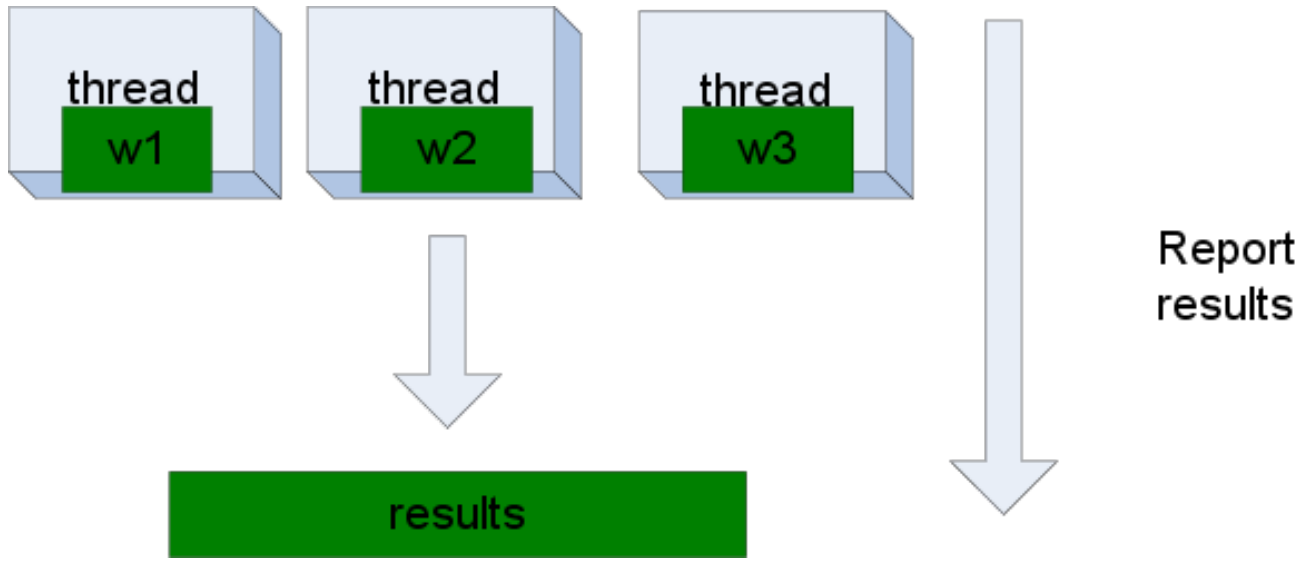


In a parallel computation, we would like to have as many threads as we have processors. e.g., a four-processor computer would be able to run four threads at the same time.

Parallelization Idea (3)



Parallelization Idea (4)

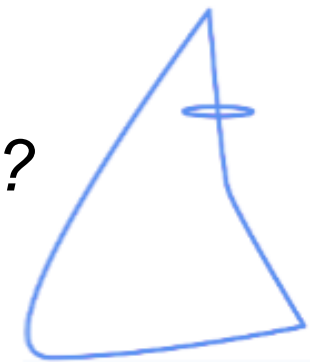


Parallelization Pitfalls

But this model is too simple!

- How do we assign work units to worker threads?
- What if we have more work units than threads?
- How do we aggregate the results at the end?
- How do we know all the workers have finished?
- What if the work cannot be divided into completely separate tasks?

What is the common theme of all of these problems?



Parallelization Pitfalls (2)

- Each of these problems represents a point at which multiple threads must communicate with one another, or access a shared resource.

- Golden rule: Any memory that can be used by multiple threads must have an associated *synchronization system*!

What is Wrong With This?

Thread 1:

```
void foo() {
x++;
y = x;
}
```

Thread 2:

```
void bar() {
y++;
x++;
}
```

If the initial state is y = 0, x = 6, what happens after these threads finish running?

Multithreaded = Unpredictability

- Many things that look like “one step” operations actually take several steps under the hood:

Thread 1:

```
void foo() {
eax = mem[x];
inc eax;
mem[x] = eax;
ebx = mem[x];
mem[y] = ebx;
}
```

Thread 2:

```
void bar() {
eax = mem[y];
inc eax;
mem[y] = eax;
eax = mem[x];
inc eax;
mem[x] = eax;
}
```

- When we run a multithreaded program, we don’t know what order threads run in, nor do we know when they will interrupt one another.

Multithreaded = Unpredictability

This applies to more than just integers:

- Pulling work units from a queue
- Reporting work back to master unit
- Telling another thread that it can begin the “next phase” of processing

… All require synchronization!

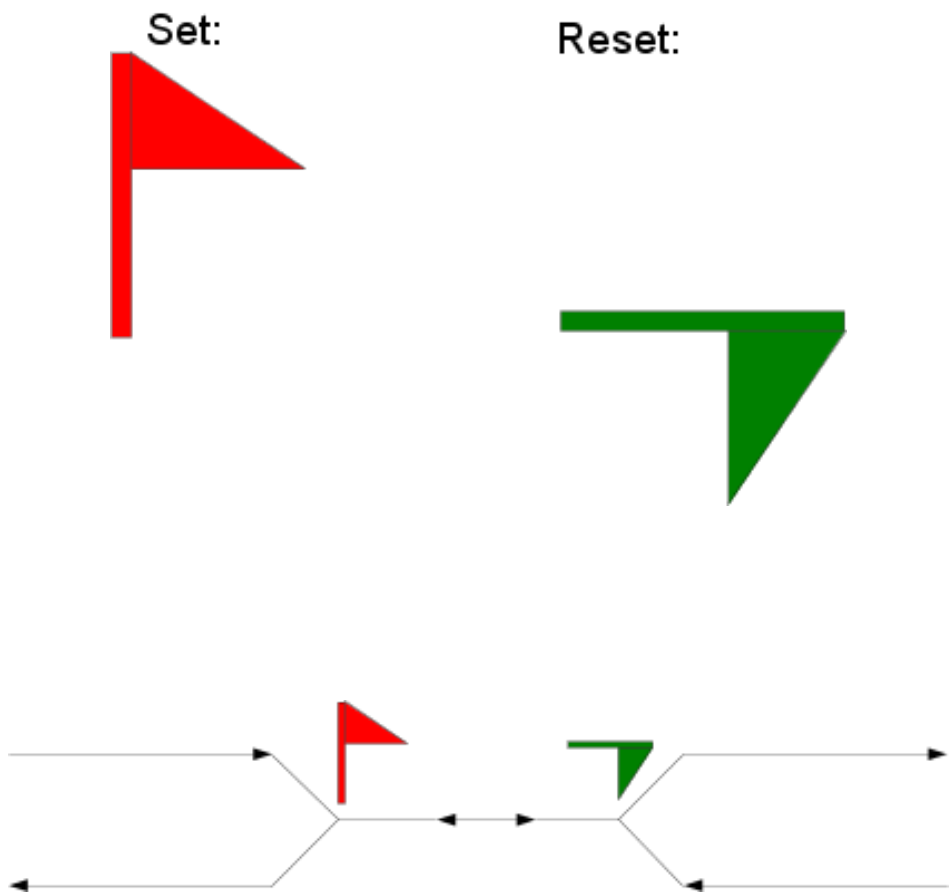
Synchronization Primitives

- A *synchronization primitive* is a special shared variable that guarantees that it can only be accessed **atomically**.

- Hardware support guarantees that operations on synchronization primitives only ever take one step

Semaphores

- A semaphore is a flag that can be raised or lowered in one step
- Semaphores were flags that railroad engineers would use when entering a shared track



Only one side of the semaphore can ever be red! (Can both be green?)

Semaphores

- set() and reset() can be thought of as lock() and unlock()
- Calls to lock() when the semaphore is already locked cause the thread to **block**.

- *Pitfalls: Must "bind" semaphores to particular objects; must remember to unlock correctly*

The “Corrected” Example

Thread 1:

```
void foo() {
sem.lock();
x++;
y = x;
sem.unlock();
}
```

Thread 2:

```
void bar() {
sem.lock();
y++;
x++;
sem.unlock();
}
```

Global var “Semaphore sem = new Semaphore();” guards access to x & y

Condition Variables

- A condition variable notifies threads that a particular condition has been met
- Inform another thread that a queue now contains elements to pull from (or that it's empty – request more elements!)
- *Pitfall: What if nobody's listening?*

The final example

Thread 1:

```
void foo() {
sem.lock();
x++;
y = x;
fooDone = true;
sem.unlock();
fooFinishedCV.notify();
}
```

Thread 2:

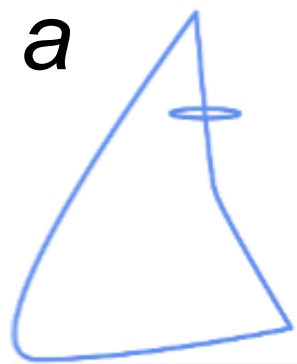
```
void bar() {
sem.lock();
while(!fooDone) fooFinishedCV.
wait(sem);
y++;
x++;
sem.unlock();
}
```

Global vars: Semaphore sem = new Semaphore(); ConditionVar fooFinishedCV = new ConditionVar(); boolean fooDone = false;

Barriers

- A barrier knows in advance how many threads it should wait for. Threads "register" with the barrier when they reach it, and fall asleep.
- Barrier wakes up all registered threads when total count is correct

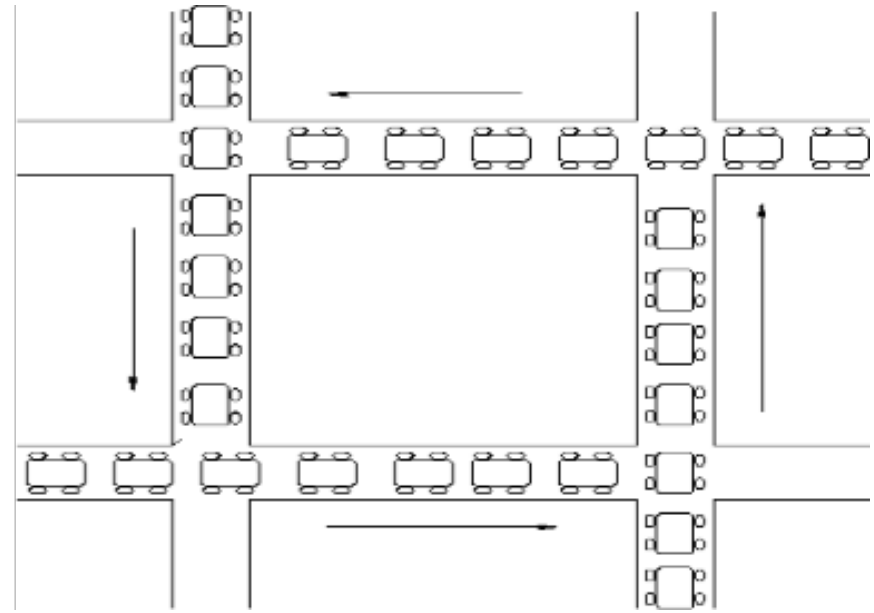
- *Pitfall: What happens if a thread takes a long time?*



Too Much Synchronization? Deadlock

Synchronization becomes even more complicated when multiple locks can be used

Can cause entire system to “get stuck”



Thread A:

```
semaphore1.lock();
semaphore2.lock();
/* use data guarded by
semaphores */
semaphore1.unlock();
semaphore2.unlock();
```

Thread B:

```
semaphore2.lock();
semaphore1.lock();
/* use data guarded by
semaphores */
semaphore1.unlock();
semaphore2.unlock();
```


(Image: RPI CSCI.4210 Operating Systems notes)

And if you thought I was joking…

The Moral: Be Careful!

- Synchronization is hard
  - Need to consider all possible shared state
  - Must keep locks organized and use them consistently and correctly
- Knowing there are bugs may be tricky; fixing them can be even worse!
- Keeping shared state to a minimum reduces total system complexity

Fundamentals of Networking

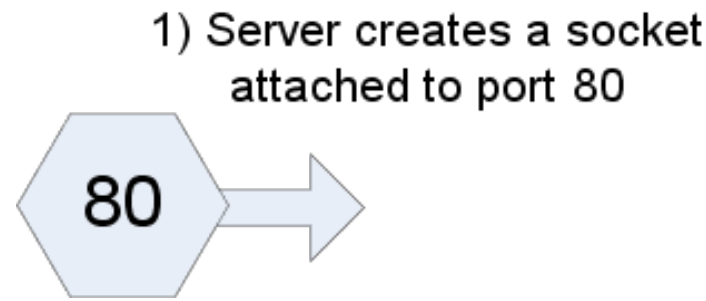
Sockets: The Internet = tubes?

- A socket is the basic network interface
- Provides a two-way “pipe” abstraction between two applications
- Client creates a socket, and connects to the server, who receives a socket representing the other side

Ports

- Within an IP address, a *port* is a sub-address identifying a listening program
- Allows multiple clients to connect to a server at once

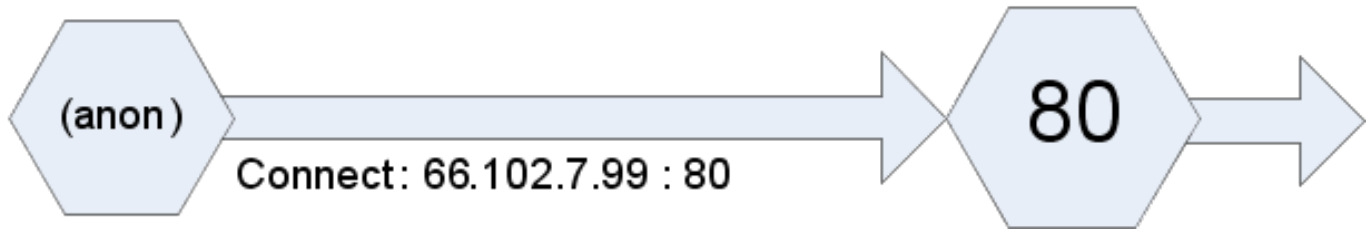
Example: Web Server (1/3)



The server creates a *listener* socket attached to a specific port.
80 is the agreed-upon port number for web traffic.

Example: Web Server (2/3)

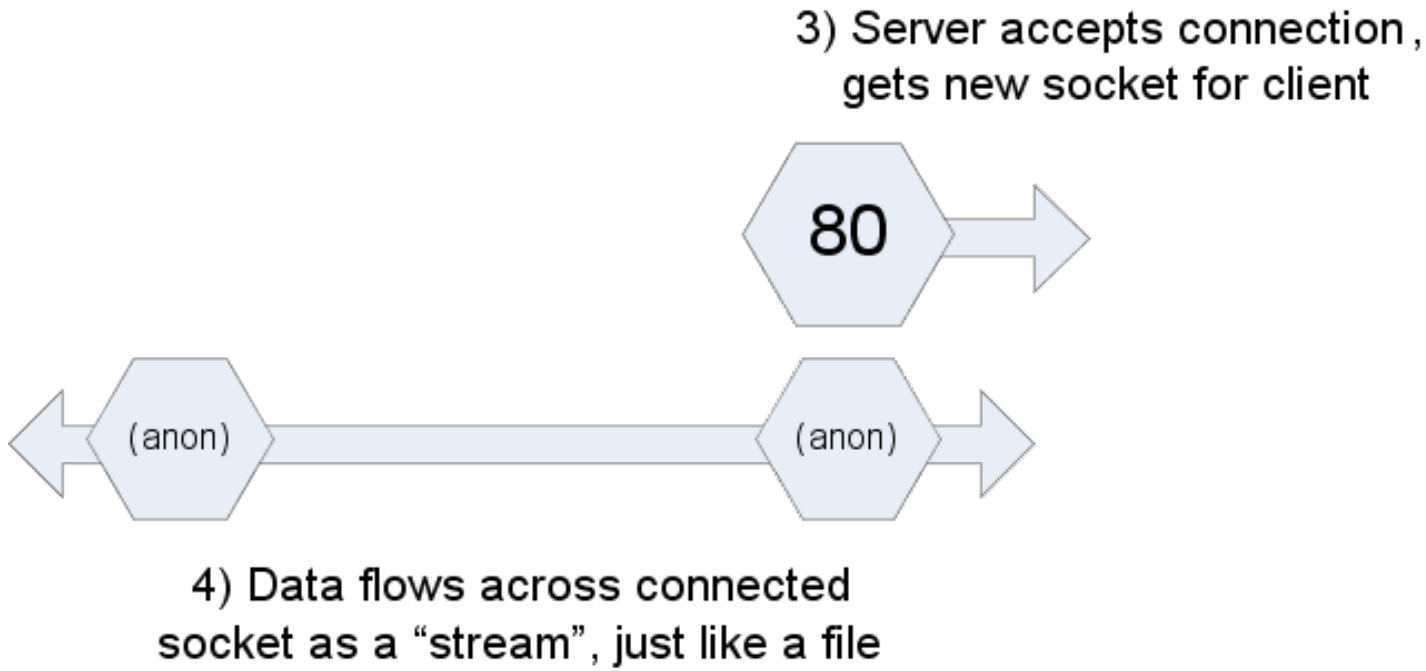
2) Client creates a socket and connects to host



The client-side socket is still connected to a port, but the OS chooses a random unused port number

When the client requests a URL (e.g., “www.google.com”), its OS uses a system called *DNS* to find its IP address.

Example: Web Server (3/3)

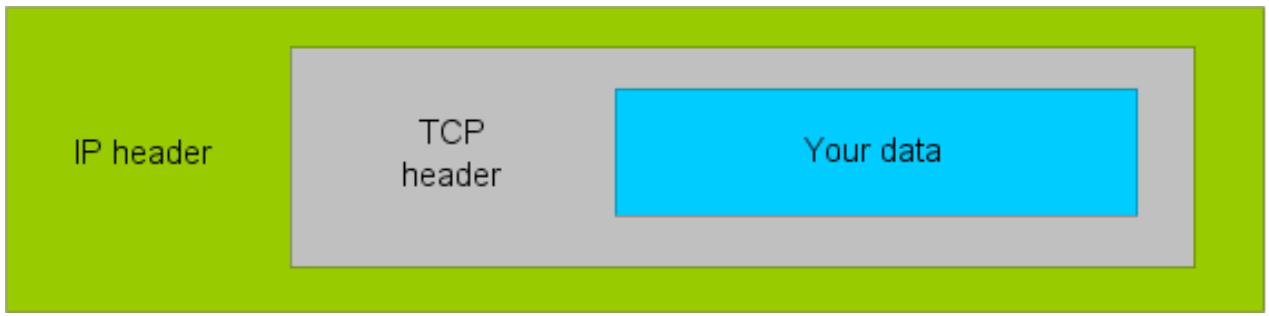


Server chooses a randomly-numbered port to handle this particular client

Listener is ready for more incoming connections, while we process the current connection in parallel

What makes this work?

- Underneath the socket layer are several more protocols
- Most important are TCP and IP (which are used hand-in-hand so often, they're often spoken of as one protocol: TCP/IP)



Even more low-level protocols handle how data is sent over Ethernet wires, or how bits are sent through the air using 802.11 wireless…

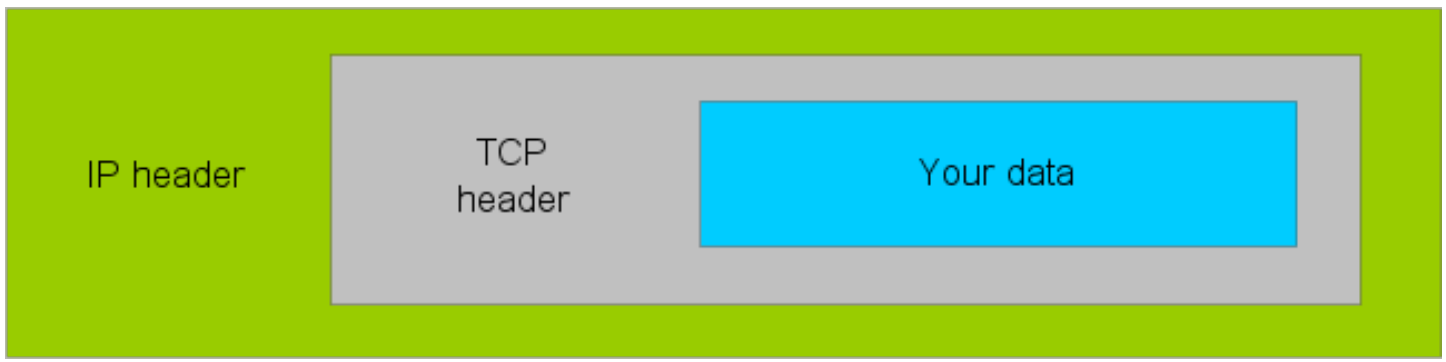
IP: The Internet Protocol

- Defines the addressing scheme for computers
- Encapsulates internal data in a “packet”

- Does not provide reliability
- Just includes enough information for the data to tell routers where to send it

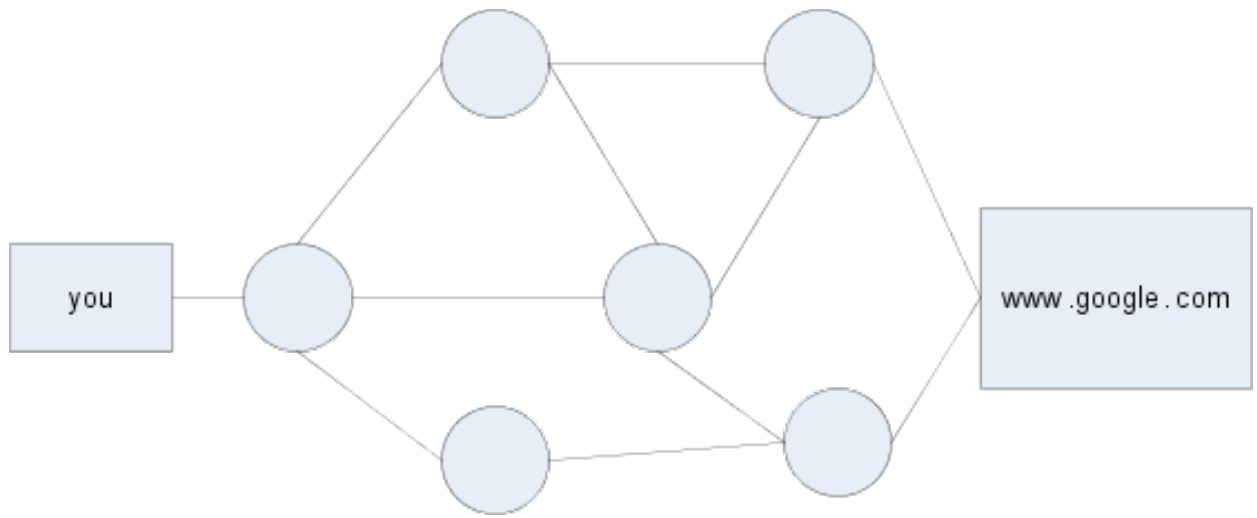
TCP: Transmission Control Protocol

- Built on top of IP
- Introduces concept of “connection”
- Provides reliability and ordering



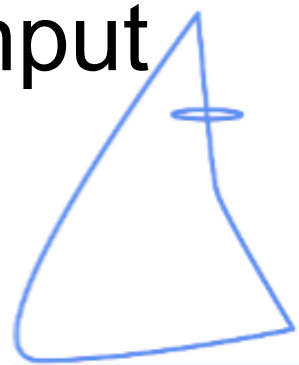
Why is This Necessary?

- Not actually tube-like “underneath the hood”
- Unlike phone system (circuit switched), the *packet switched* Internet uses many routes at once



Networking Issues

- If a party to a socket disconnects, how much data did they receive?
- … Did they crash? Or did a machine in the middle?
- Can someone in the middle intercept/modify our data?
- Traffic congestion makes switch/router topology important for efficient throughput



Final Thoughts

- Various background topics fit into this course
- Two examples highlighted
- Other background topics may benefit from expansion, worksheets, reinforcement