Concurrency & Performance

CS 475, Spring 2018
Concurrent & Distributed Systems

Locking in Java

- Most locks are *reentrant*: if you hold it, and ask for it again, you don't have to wait (because you already have it)
- Basic primitives:
  - synchronized{}
  - wait
  - notify
- Plus…
  - Lock API… lock.lock(), lock.unlock()
  - The *preferred* way

Locking Granularity

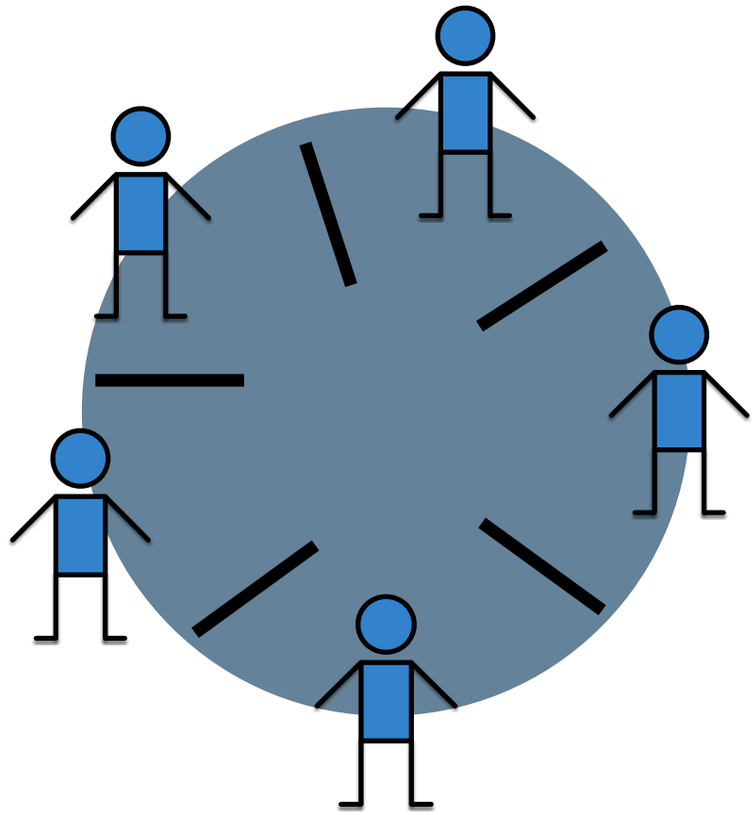
- BIG design question in writing concurrent programs: how many locks should you have?
- Example: Distributed filesystem
  - It would be *correct* to block all clients from reading *any* file, when one client writes a file
    - However, this would not be performant at all!
    - It would be much better to instead lock on *individual files*
- *More locks -> more complicated semantics and tricky to avoid deadlocks, races*

Designing Locking Strategies

- How we acquire and release locks can hugely impact performance
- Two solutions might be correct, but one may waste more time by:
  - Acquiring and releasing unnecessary locks
  - Waiting for locks

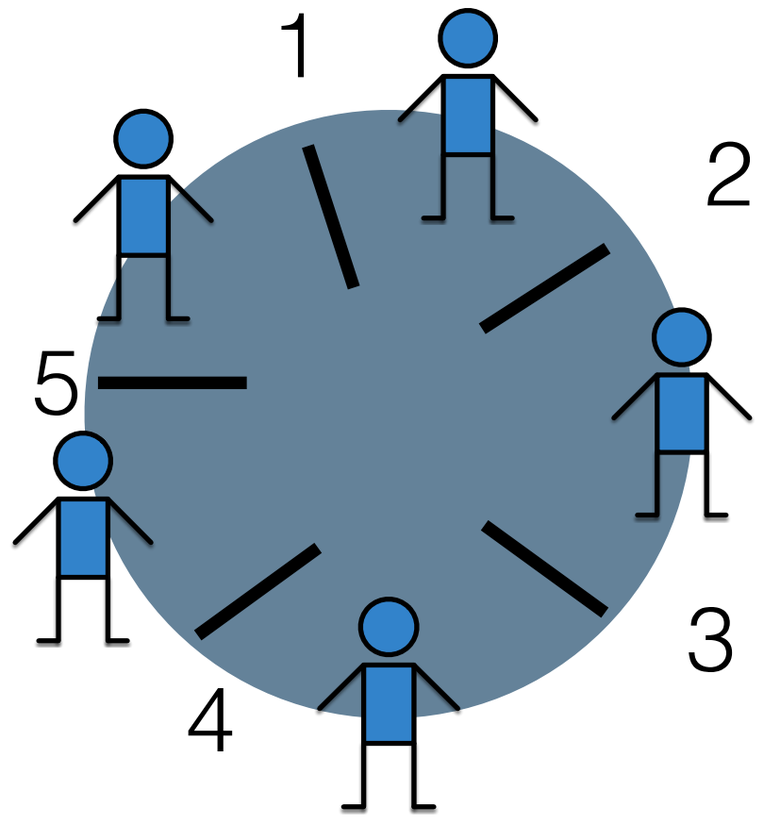
Dining Philosophers

- N philosophers seated around a circular table
  - One chopstick between each philosopher (N chopsticks)
  - A philosopher picks up both chopsticks next to him to eat
  - Philosophers may not pick up both chopsticks at the same time
- How do they all eat without deadlocking or starving?



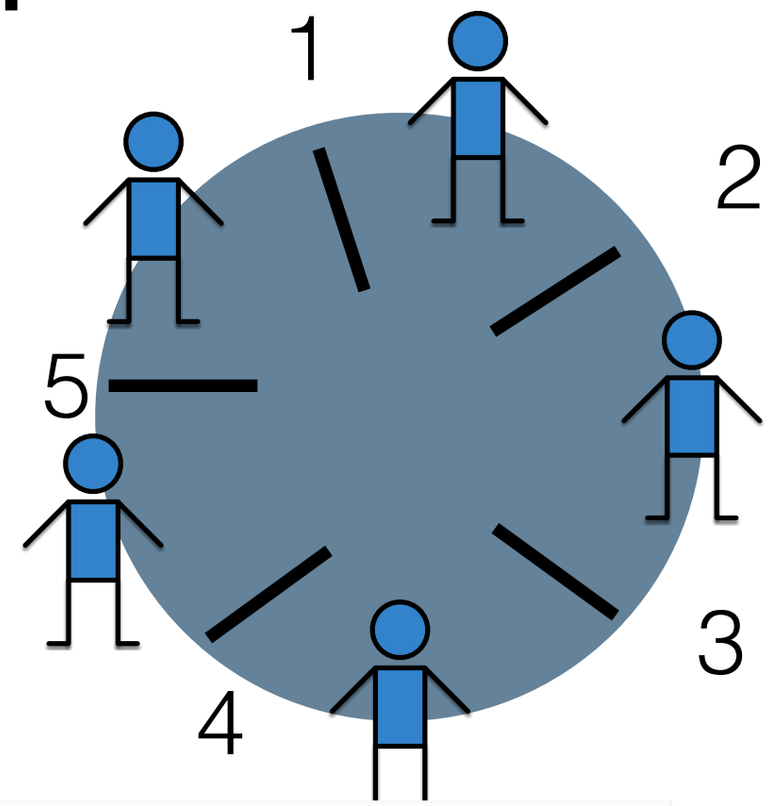
Dining Philosophers

- Give each chopstick a lock
- Is this enough?
- Could deadlock!
- Actual solutions:
  - Pick up one chopstick, wait for the other for N msec, otherwise put down what you have, wait, and try again
  - Only allow 4 philosophers to pick up chopsticks at once
  - Even # seats pick up right chopstick, odd # seats pick up left



Dining Philosophers

- Give each chopstick a lock
- Is this enough?
- Could deadlock!
- Actual solutions:
  - Pick up one chopstick, wait for the other for N msec, otherwise put down what you have, wait, and try again
  - Only allow 4 philosophers to pick up chopsticks at once
  - Even # seats pick up right chopstick, odd # seats pick up left



1,450 grains of rice/sec

5,431,616 grains of rice/sec

12,450,856 grains of rice/sec

Announcements

- Reminder: HW2 is out
  - http://www.jonbell.net/gmu-cs-475-spring-2018/homework-2/
- Today: Different concurrent programming models and their impacts on performance
  - Cool, vaguely relevant podcasts:
    - Radiolab - Million Dollar Microsecond
    - Planet Money - BOTUS
  - Java Streams tutorial - http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/
  - Fork/Join tutorial - https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html

Designing for Performance

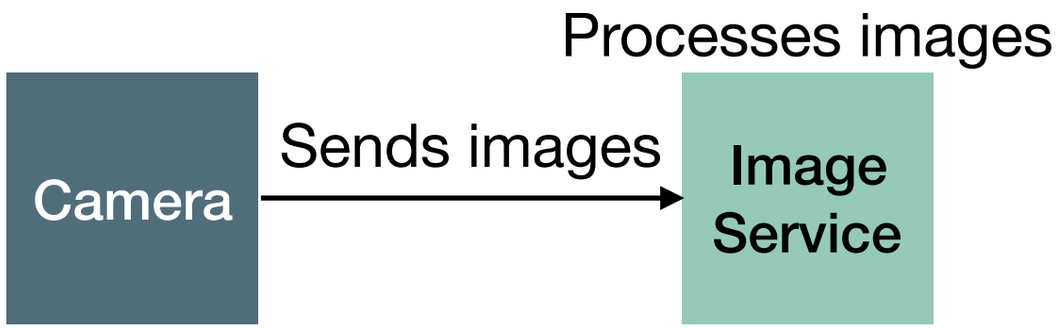
- What factors can impact performance?
  - Limits imposed by physics
  - Limits imposed by technology
  - Limits imposed by economics
- These limits can force us to make tradeoffs
  - Smaller chips are faster, but harder to dissipate heat
  - Need to serve X clients, can only spend Y on CPUs

Performance Metrics

- Capacity
  - Consistent measure of a service's size or amount of resources
- Utilization
  - Percentage of that resource used for a workload
- Overhead
  - Percentage of that utilization used for bookkeeping
- Useful Work
  - Percentage of that utilization used for what we actually need to do
- Latency
  - How long it takes an input to propagate through a system and generate an output
- Throughput
  - Work done per time

Adjusted by buying more resources

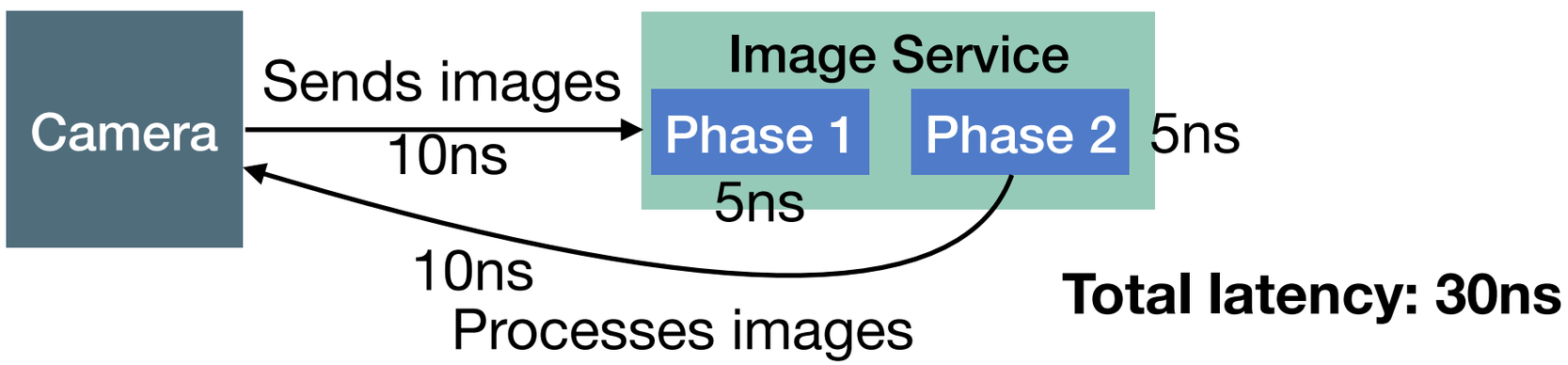
Resource Metrics - Example



- Say, capacity is measured in terms of processor cycles
- Workload might be processing a single image
- Utilization could 10% -> 90% of processor is unused
- Overhead could go to the OS, perhaps 5% of the CPU going to OS bookkeeping
- Useful work would then be 5%

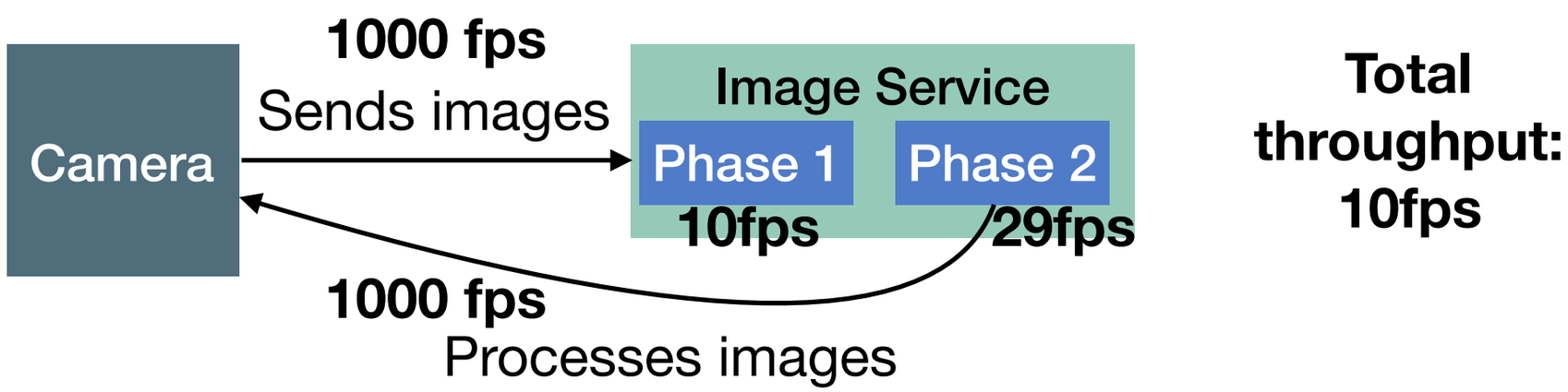
Latency

- In client/server model, latency is simply: time between client sending request and receiving response
- What contributes to latency?
  - Latency sending the message
  - Latency processing the message
  - Latency sending the response
- Adding pipelined components -> latency is cumulative



Throughput

- Measure of the rate of useful work done for a given workload
- Example:
  - Throughput is camera frames processed/second
- When adding multiple pipelined components -> throughput is the minimum value



Designing for Performance

- Measure system to find which aspect of performance is lacking (throughput or latency)
- Measure each component to identify bottleneck
- Identify if fixing that bottleneck will realistically improve system performance
- Measure improvement
- Repeat

Reducing Latency

- Often more challenging than increasing throughput
  - Examples:
    - Physical - Speed of light (network transmissions over long distances)
    - Algorithmic - Looking up an item in a hash table is limited by hash function
    - Economic - Adding more RAM gets expensive

Latency & Stock Trading

- Buy low/sell high
- Most of skill is in knowing what a stock will do **before** your competitors

Latency & Stock Trading

- Algorithmic trading -> computer programs look at various factors, place trades automatically
- Example:
  - President Trump tweets positively about a company -> price goes up
  - Write a script to check twitter for company mentions, immediately buy/sell stock
  - Get in and out before it hits CNN!
  - https://www.npr.org/sections/money/2017/04/07/522897876/meet-botus-planet-money-s-stock-trading-twitter-bot

Latency & Stock Trading

- This only works if you can make your trades **before** other people find out
- What if you set up this bot in Chicago, and I set one up in NYC?
  - I would beet you to it, every time.

Latency & Stock Trading

- What is the speed of light?
  - ~300,000 km/sec
- How fast does your CPU execute an instruction?
  - 0.33 nanoseconds (say, 3Ghz CPU)
- How far does light travel in 1 CPU cycle?
  - 10 cm
- How many instructions does your CPU execute in the time it takes light to travel from Chicago to NYC and back?
  - ~700 miles -> 7.4msec -> 22 million instructions
- Being in NYC would let me execute 22 million instructions in the time it took you to send your stock order to NYC and get a response!

Reducing Latency with $$$$

- People actually care a LOT about the latency between NYC and Chicago, because commodities are traded in Chicago and stocks are traded in NYC
  - Changes to commodities prices (e.g. **ethanol**) can dramatically impact price of some stocks

Reducing Latency with $$$$

- It's not quite as simple as 700 miles -> 7.4msec
- There are streams, mountains, etc… more like 1,000 miles
- Light is refracted in a fiber optic cable is ~31% slower
- What do we do if money is no object?

Reducing Latency with Billions of Dollars

ORIGINAL CABLE

Technology
Buried fiber-optic cable

Completion
Mid-1980s

Path length
~ 1,000 miles

Round-trip time for data
14.5
milliseconds and up

Approach
Multiple routes followed the easiest rights-of-way—along rail lines. But that means time sucking jogs and detours.

SPREAD NETWORKS

Technology
Buried fiber-optic cable

Completion
August 2010

Path length
825 miles

Round-trip time for data
13.1
milliseconds

Approach
Spread bought its own rights-of-way, avoiding a Philadelphia-ward dip in favor of a shorter path northwest through central Pennsylvania.

MCKAY BROTHERS

Technology
Microwave beams through air

Completion
July 4, 2012

Path length
744 miles

Round-trip time for data
9
milliseconds

Approach
Microwaves generally move faster than photons in optical fiber, and McKay's network uses just 20 towers on a nearly perfect great circle.

TRADEWORX

Technology
Microwave beams through air

Completion
Winter 2012

Path length
~731 miles

Round-trip time for data
8.5
milliseconds (est.)

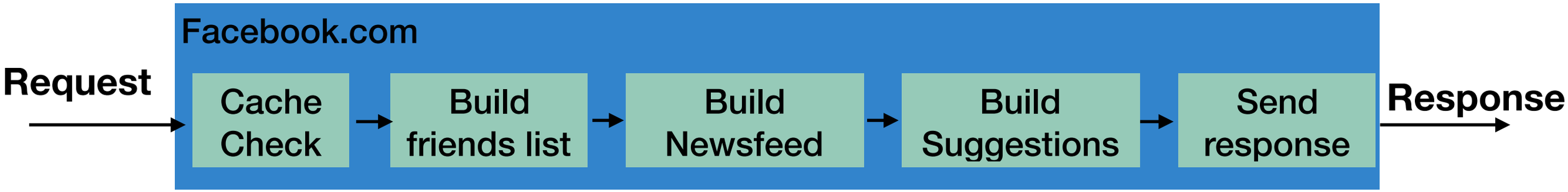
Approach
Tradeworx is highly secretive, but the company is open about the price of a subscription: $250,000 a year.

https://www.zerohedge.com/news/chicago-new-york-and-back-85-milliseconds

Reducing Latency for Mortals

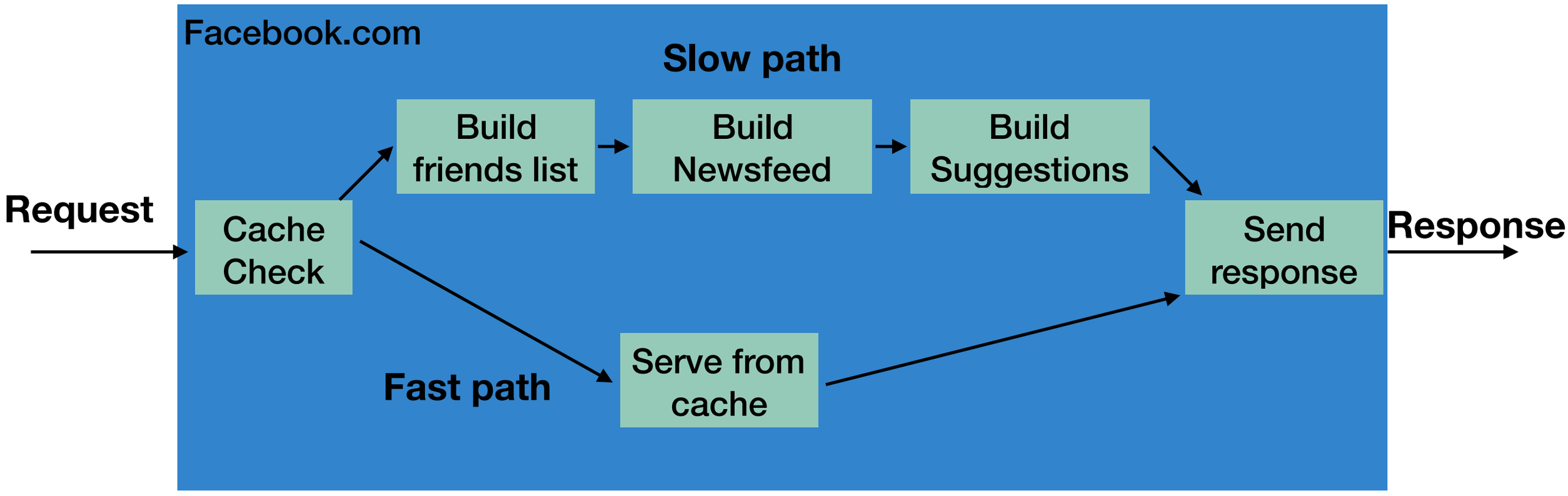
- Instead of fighting nature, look closer at the structure of the problem
- Normal trick is to exploit workload properties to reduce latency, instead

Reducing Latency



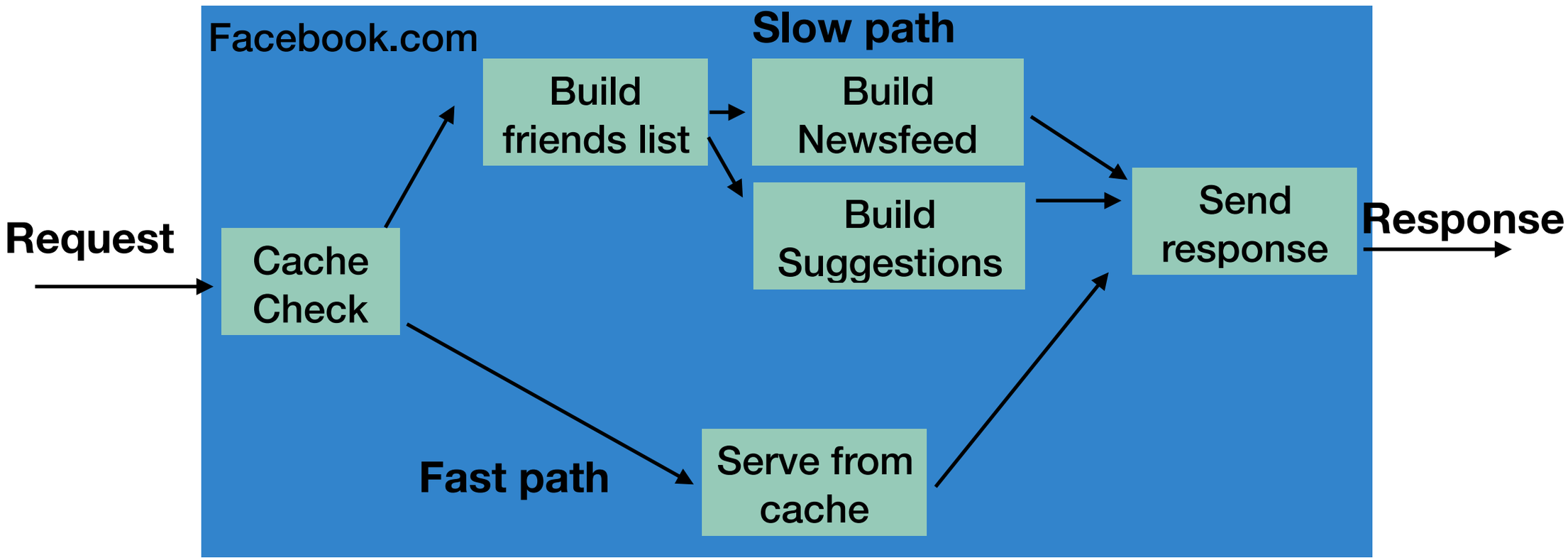
Reducing Latency

- Approach: **Optimize for the common case** (aka fast path and slow path)



Reducing Latency

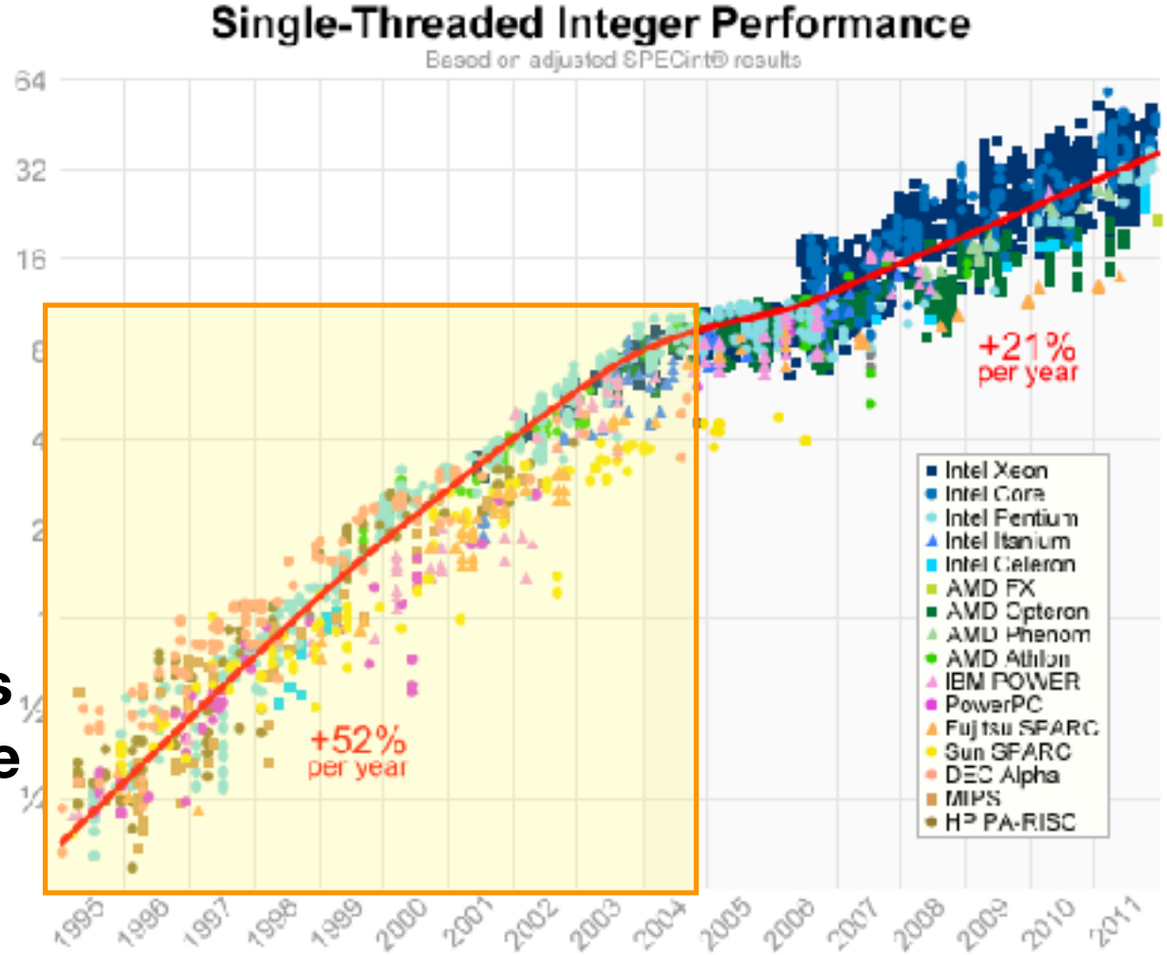
- Approach: use **concurrency**
- Limited by serial section



Reducing Latency

- Approach: **Use better technology**
- However, processors are limited by heat dissipation
- Waiting for processors to get faster is not a great approach these days
- Instead: move computation from resource constrained devices to cloud

Wait for better CPUs worked better before 2005

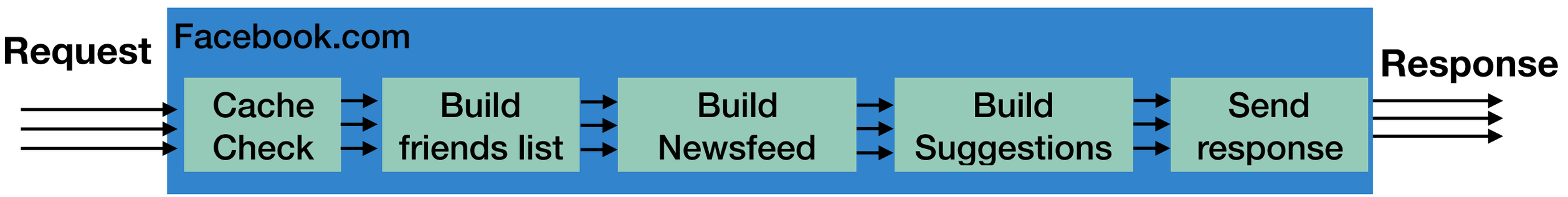


Improving Throughput

- Might be able to *hide* latency by overlapping many requests (increase throughput)
- Example: Want to make client/server application appear faster
- But, client on east coast communicates with west coast (20msec)
- In 20msec, a processor can execute millions of instructions
- Instead: batch lots of requests into a single message, have more processing on client side

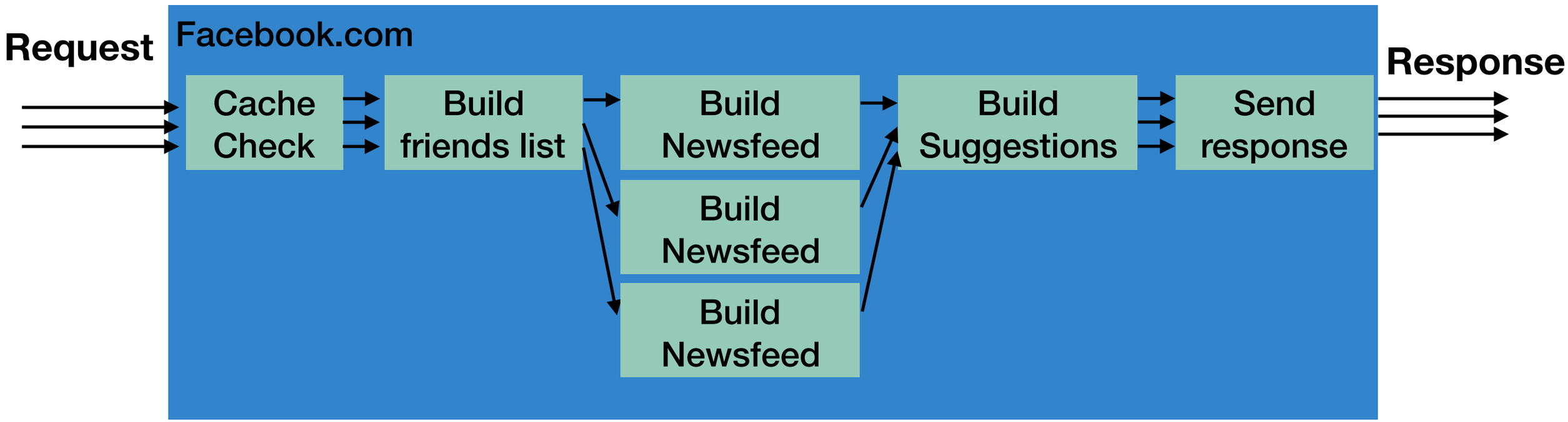
Improving Throughput

- Introduce concurrency into our pipeline
- Each stage runs in its own thread (or many threads, perhaps)
- If a stage completes its task, it can start processing the next request right away
  - E.g. our system will process multiple requests at the same time



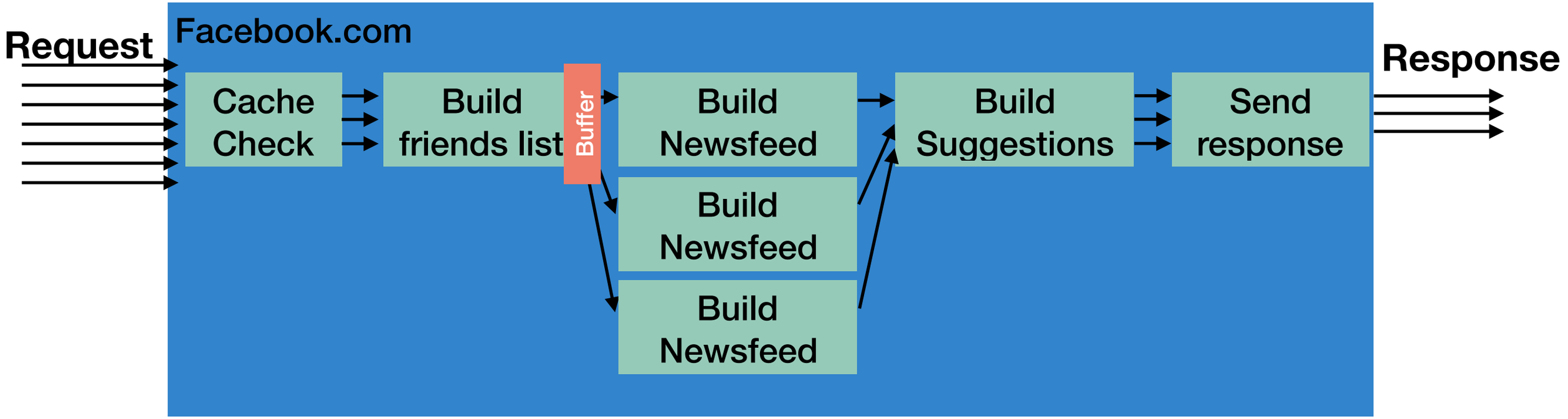
Improving Throughput

- Can also introduce concurrency to stages
- If one stage is a bottleneck, can we add more copies of it?



Queueing and Overload

- What happens when a slow component gets overloaded?
- Need to place a bounded buffer in between components!
  - When buffer becomes full, it prevents new requests from being accepted



Asynchronous Programming

- AKA event-driven programming
- A paradigm that lends itself well to scaling, especially in a multi-stage systems (like the example with Facebook)
- Allows us to think about **what** is done, **abstract** away **how** it is done
- We will discuss two asynchronous models: **streams,** and **Promises**, **neither of which** make you think about threads (or locks?)

Streams

- Java 8 introduced the concept of **Streams**
- A stream is a sequence of objects
- Streams have functions that you can perform on them, which are (mostly) **non-interfering** and **stateless**
  - Non-interfering: Does not modify the actual stream
  - Stateless: Each time the function is called on the same data, get same result
- Example:

```
IntStream.range(1, 1000000) //Generate a stream of all ints 1 - 1m
.filter(x -> isPrime(x)) //Retain only values that pass some expensive isPrime
function
.forEach(System.out::println); //For each value returned by filter, print it
```

Sidebar: Lambdas

- I don't know if you have seen this before

```
IntStream.range(1, 1000000)
.filter(x -> isPrime(x))
.forEach(System.out::println);
```

- This line is called a lambda expression
- We should have shown it to you before, because it's a core part of Java syntax since Java 8 was released in 2014
- Effectively, think of this as shorthand for:

```
IntStream.range(1, 1000000)
.filter(new IntPredicate() {
    @Override
    public boolean test(int x) {
        return isPrime(x);
    }
})
.forEach(System.out::println);
```

- In fact, javac generates exactly the long-hand code for that shorthand

Streams

```
IntStream.range(1, 1000000) //Generate a stream of all ints 1 - 1m
.filter(x -> isPrime(x)) //Retain only values that pass some expensive
isPrime function
.forEach(System.out::println); //For each value returned by filter, print
it
```

- Why use the stream interface instead of

```
for(int i = 1; i < 1000000; i++)
    if(isPrime(x))
        System.out.println(x);
```

- Who wants to write the parallel version of this?

```
IntStream.range(1, 1000000) //Generate a stream of all ints 1 - 1m
.filter(x -> isPrime(x)) //Retain only values that pass some expensive
isPrime function
.parallel() //Do the filtering in parallel
.forEach(System.out::println); //For each value returned by filter, print
it
```

- The magic works as long as isPrime is stateless!

Streams - what can't be parallelized

- Interference

```
List<String> list = new ArrayList<>(Arrays.asList("Luke", "Leia", "Han"));
list.stream()
.peek(name -> {
    if (name.equals("Han")) {
        list.add("Chewie"); // Adds to list that we are peeking into
    }
})
.forEach(i -> {});
```

- Stateful

```
boolean tooBusy = false;
public void isPrime(int x)
{
    if(tooBusy)
        return false;//don't bother running if another thread set tooBusy
    else
        //do a sieve of erasthenes
}
```

- Side effects

```
List<Integer> list = new ArrayList<>(
Arrays.asList(1,3,5,7,9,11,13,15,17,19));
List<Integer> result = new ArrayList<>();
list.parallelStream()
.filter(x -> isPrime(x))
.forEach(x -> result.add(x)); //Changing external state, which may not (is not) thread safe
```

Streams under the hood

- Just adding more parallel() doesn't always make it faster! (see: law of leaky abstractions)
- There is some overhead to how a parallel operation occurs
- Internally, Java keeps a pool of **worker threads** (rather than make new threads for each parallel task)
- Streams use a special kind of pool, called a **ForkJoinPool**

Fork/Join Programming

- Special kind of task - fork() defines how to create subtasks, join() defines how to combine the results
- Similar to map/reduce, but not distributed
- For streams:
  - **Fork** a task into subtasks for many threads to work on
  - **Join** the results together

Fork/Join Programming

- Obligatory array sum example

```
class Sum extends RecursiveTask<Long> {
    static final int SEQUENTIAL_THRESHOLD = 5000;

    int low;
    int high;
    int[] array;

    Sum(int[] arr, int lo, int hi) {
        array = arr;
        low   = lo;
        high  = hi;
    }

    protected Long compute() {
        if(high - low <= SEQUENTIAL_THRESHOLD) {
            long sum = 0;
            for(int i=low; i < high; ++i)
                sum += array[i];
            return sum;
        } else {
            int mid = low + (high - low) / 2;
            Sum left  = new Sum(array, low, mid);
            Sum right = new Sum(array, mid, high);
            left.fork();
            long rightAns = right.compute();
            long leftAns  = left.join();
            return leftAns + rightAns;
        }
    }

    static long sumArray(int[] array) {
        return ForkJoinPool.commonPool().invoke(new Sum(array,0,array.length));
    }
}
```

Promise

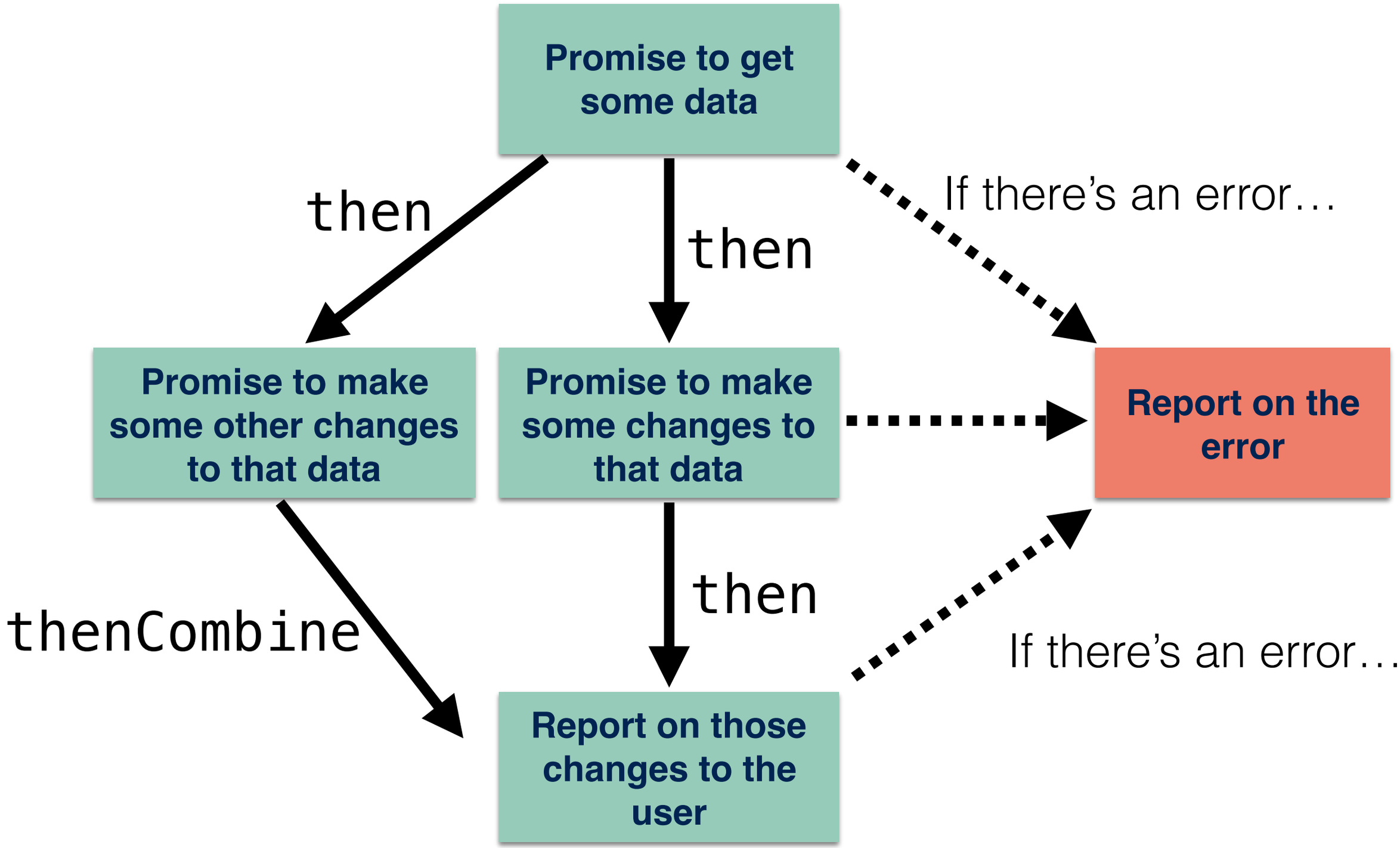
- What if we want to run some task, and do stuff while we are waiting for it to be done?
- You COULD do it with a complicated combination of **synchronized**, **wait**, and **notify**
- You can use the **Promise** abstraction instead
  - Called a **CompletableFuture** in Java 8

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    }
    return "Result of the asynchronous computation";
});
// Block and get the result of the Future
String result = future.get();
System.out.println(result);
```

Promise Use-Cases

- Any case where you need to have multiple things happen in the background, but care about the result, and care about them happening in some order
- Asynchronous I/O
  - Read data from a web service
  - Then process it
  - Then save it to a file

Chaining Promises



Promises

- Catch errors by providing a callback function for **exceptionally** (called when an exception occurs in any of those threads
- API: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html