Distributed Abstractions

SWE 622, Spring 2017
Distributed Software Engineering

GEORGE MASON UNIVERSITY

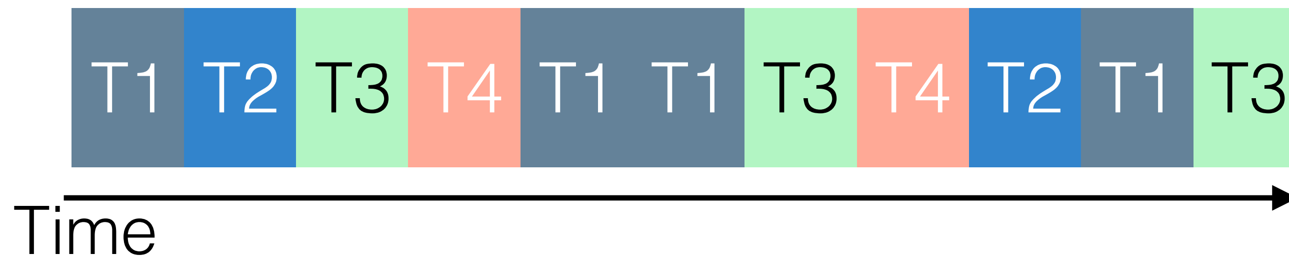
Today

- Abstractions + Models and how they can simplify design challenges
- Some “real” abstractions and models
  - Modeling time and events
  - Modeling caches
- Lab activity - Redis
- HW2 description

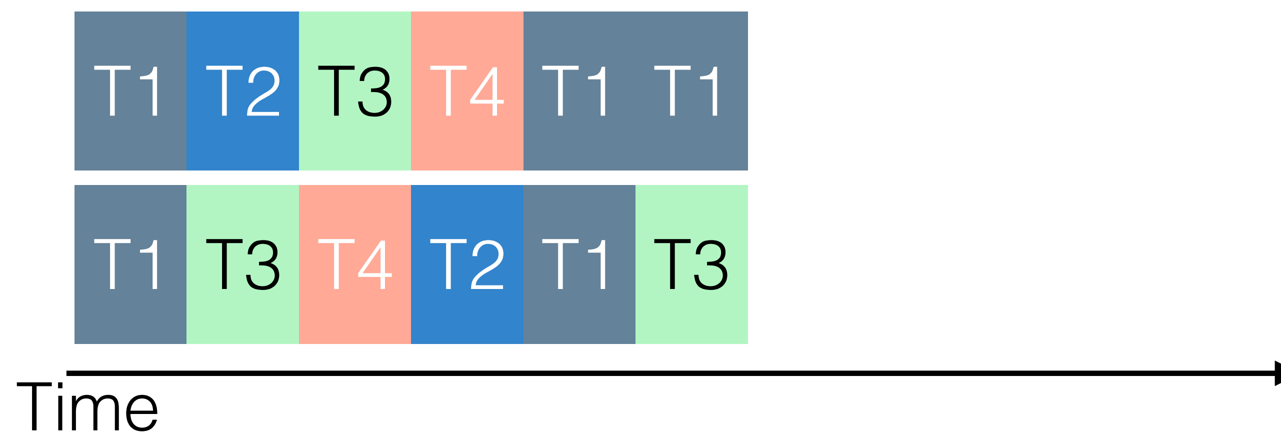
Review: Concurrency vs Parallelism

4 different threads: T1 T2 T3 T4

Concurrency:
(1 processor)

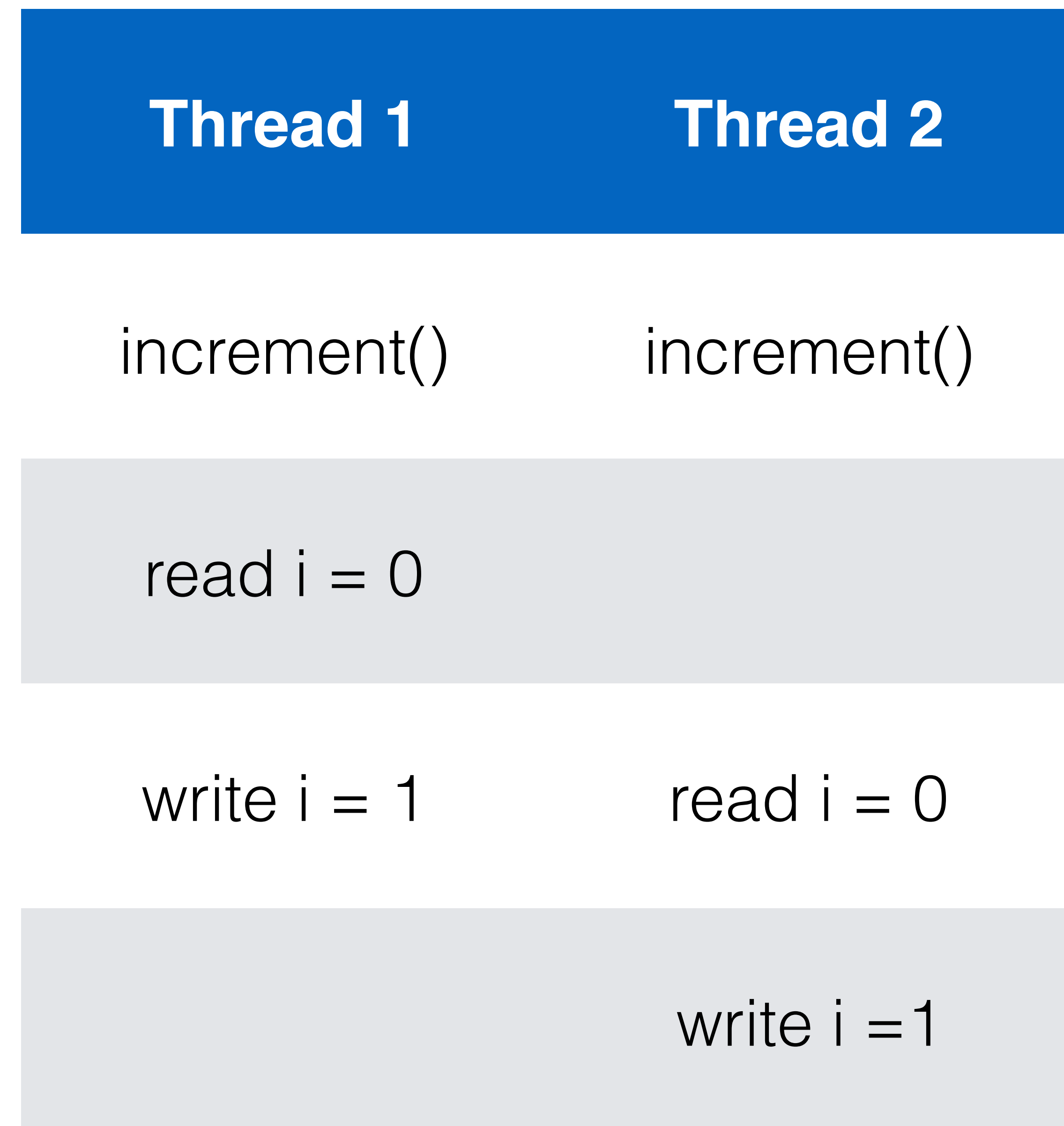


Parallelism:
(2 processors)



Review: Race Conditions

```java
static int i = 0;
public static void increment()
{
    i = i + 1;
}
```

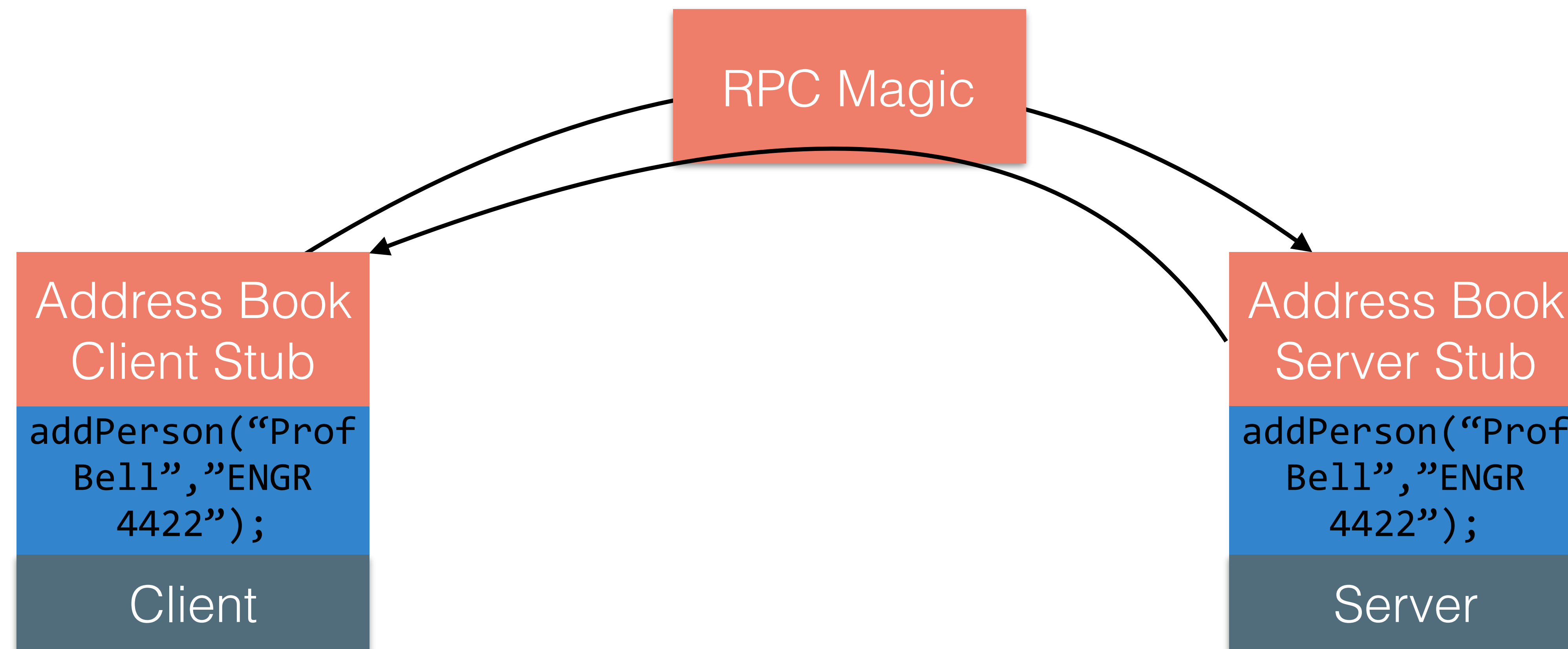


Thread 1	Thread 2
increment()	increment()
read i = 0	
write i = 1	read i = 0
	write i =1

Review: Java Lock API

```
static ReadWriteLock lock = new ReentrantReadWriteLock();
static int i;
public static void increment()
{
    lock.writeLock().lock();
    try{
        i = i + 1;
    } finally{
        lock.writeLock().unlock();
    }
}
public static int getI()
{
    lock.readLock().lock();
    try{
        return i;
    } finally{
        lock.readLock().unlock();
    }
}
```

Review: Remote Procedure Calls



Review: HW1

- Post-mortem feedback: http://b.socrative.com/ click on student login, then SWE622 as room name
- Building a cache in front of a filesystem
- In Java
- Tricky part:
    - Consistency (multiple processes can access filesystem at once)

Distributed Systems Abstractions

- Goal: find some way of making our distributed system look like a single system
- Never achievable in practice
- BUT if we can come up with some model of how the world might behave, we can come up with some generic solutions that work pretty well
  - And hopefully we can understand how they can go wrong

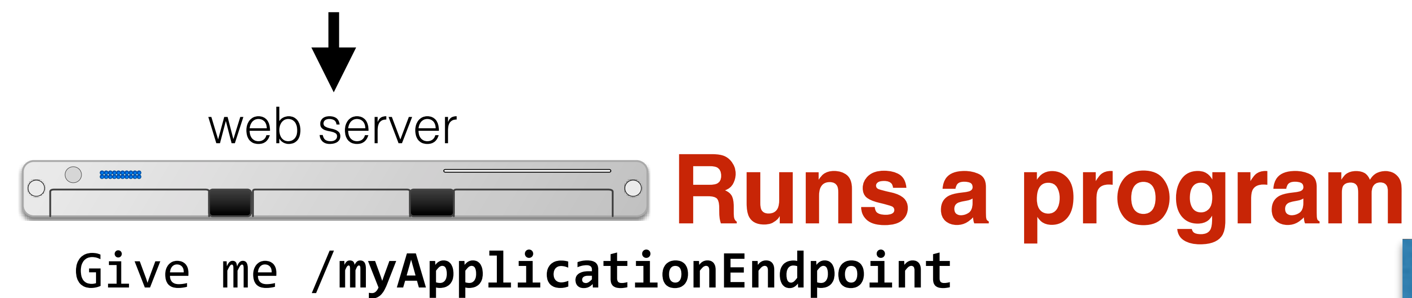
Abstractions & Architectures

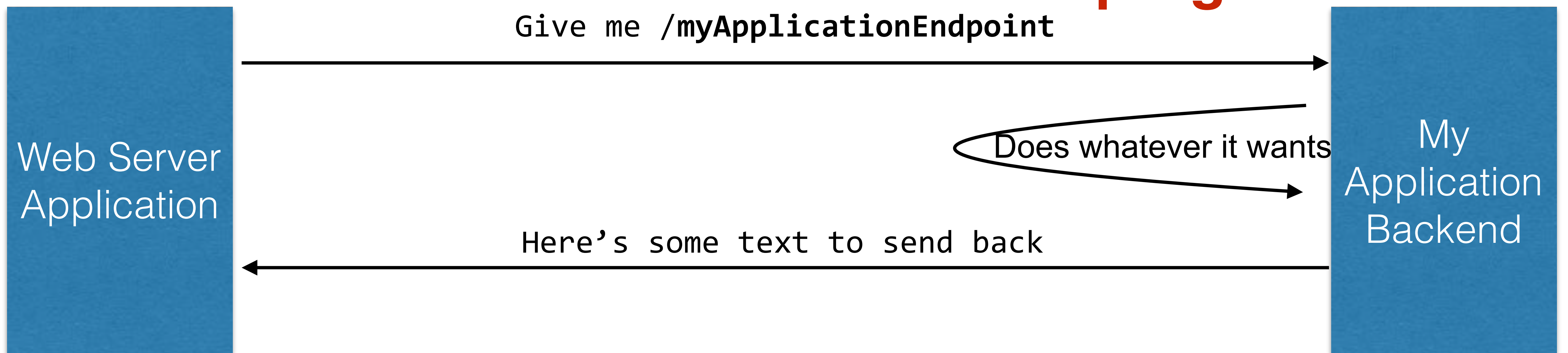
- We can design *architectures* that embody some systems model, providing some framework code to make it easier to get some task done
- Case study example: web architectures
- Assumptions:
    - "one" server, many clients
    - Synchronous communication
    - Client is unlikely to be partitioned from a subset of servers; likely some subset of servers are partitioned from other servers
    - Client is mostly stateless

The good old days of web apps

HTTP Request

```
GET /myApplicationEndpoint HTTP/1.1
Host: cs.gmu.edu
Accept: text/html
```





HTTP Response

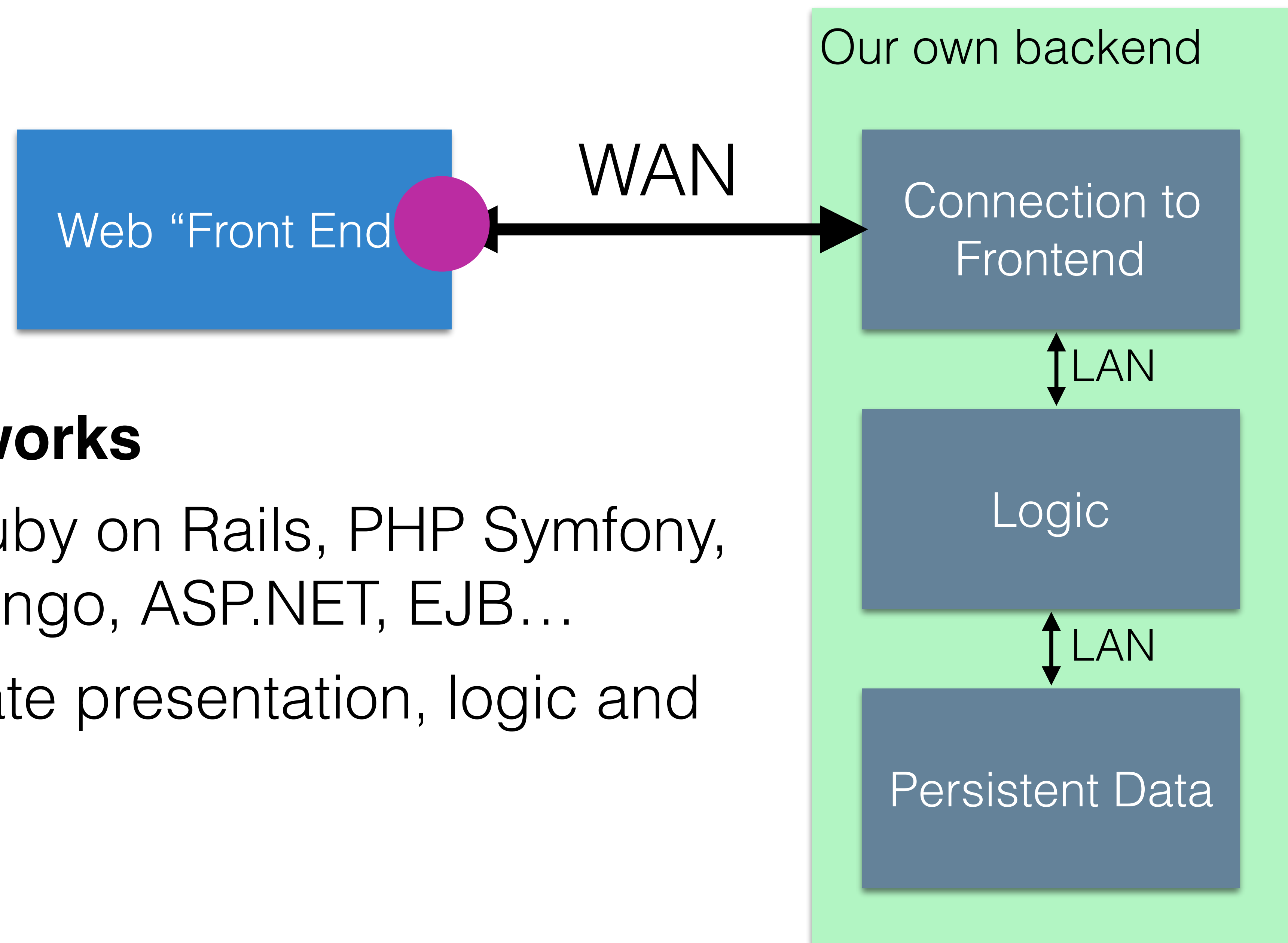
```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8

<html><head>...
```

Brief history of Backend Development

- In the beginning, you wrote whatever you wanted using whatever language you wanted and whatever framework you wanted
- Then… PHP and ASP
  - Languages “designed” for writing backends
  - Encouraged spaghetti code
  - A lot of the web was built on this
- A whole lot of other languages were also springing up in the 90’s…
  - Ruby, Python, JSP

Backend Frameworks



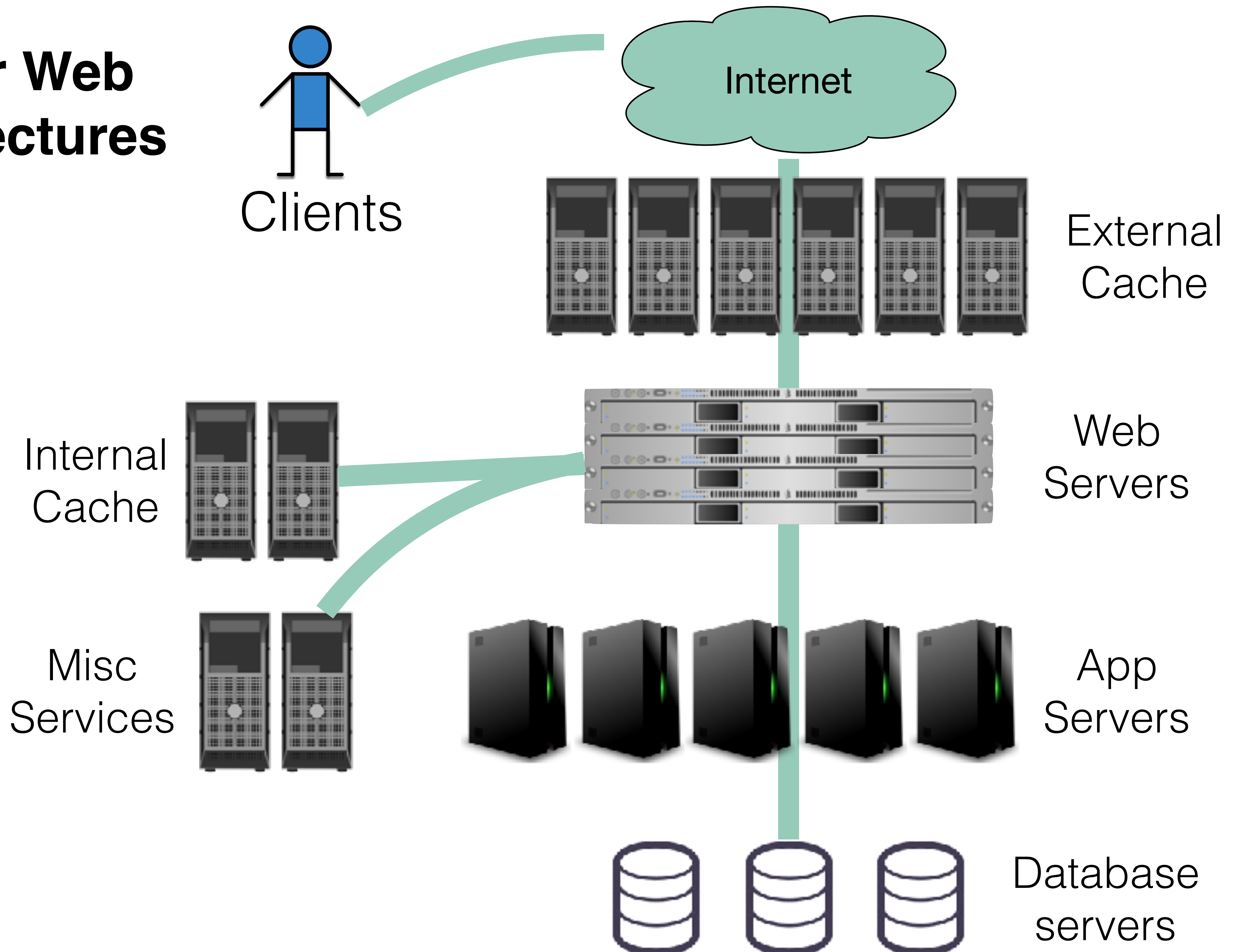
- Then: **frameworks**
  - SailsJS, Ruby on Rails, PHP Symfony, Python Django, ASP.NET, EJB…
- MVC - separate presentation, logic and persistence

Scaling web architectures up

- What happens when we have to use this approach to run, say… Facebook?
- Tons of dynamic content that needs to be updated, petabytes of static content (like pictures), users physically located all over, lots of stuff to keep track of, where do we start?

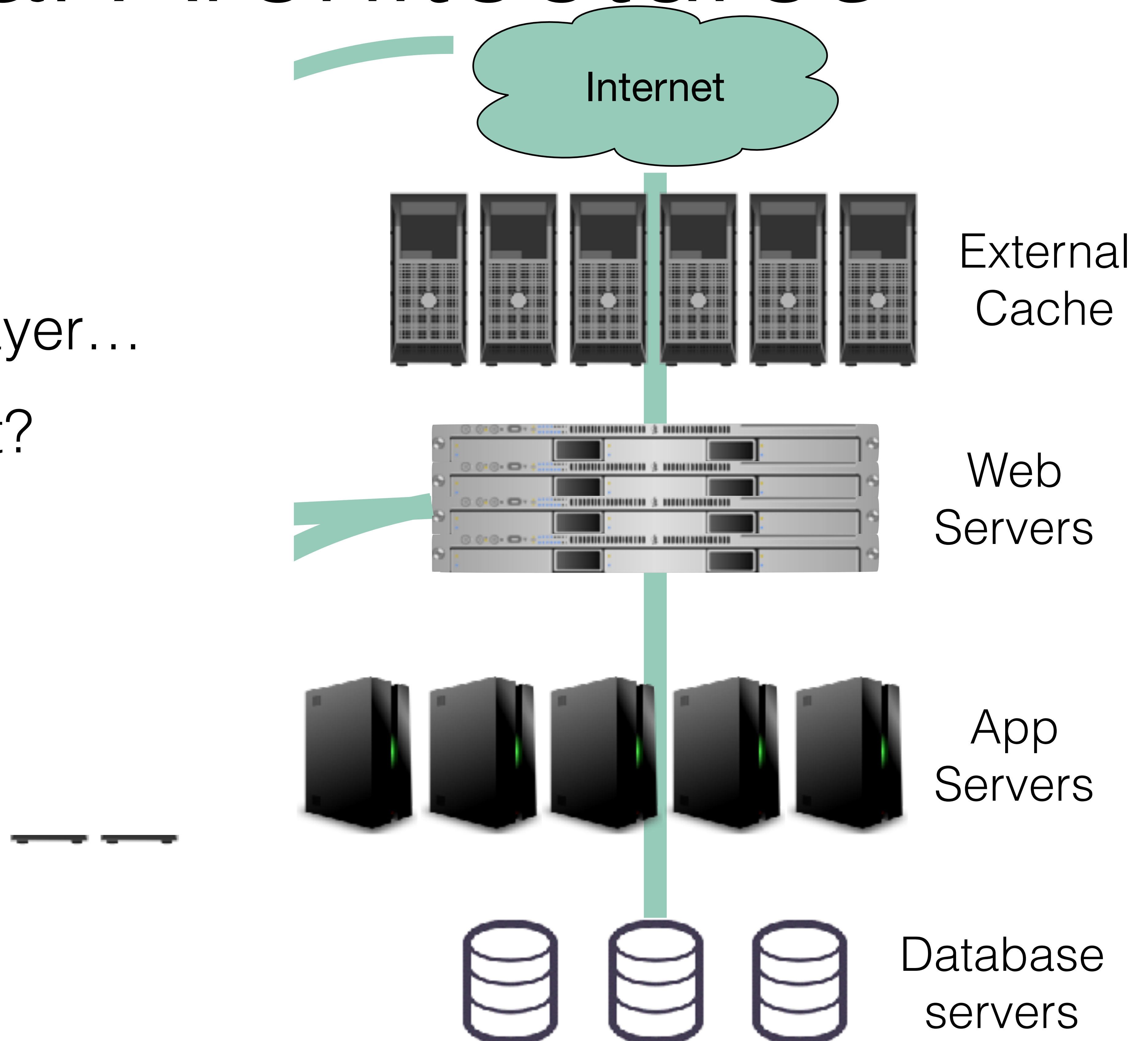
Real Architectures

N-Tier Web Architectures



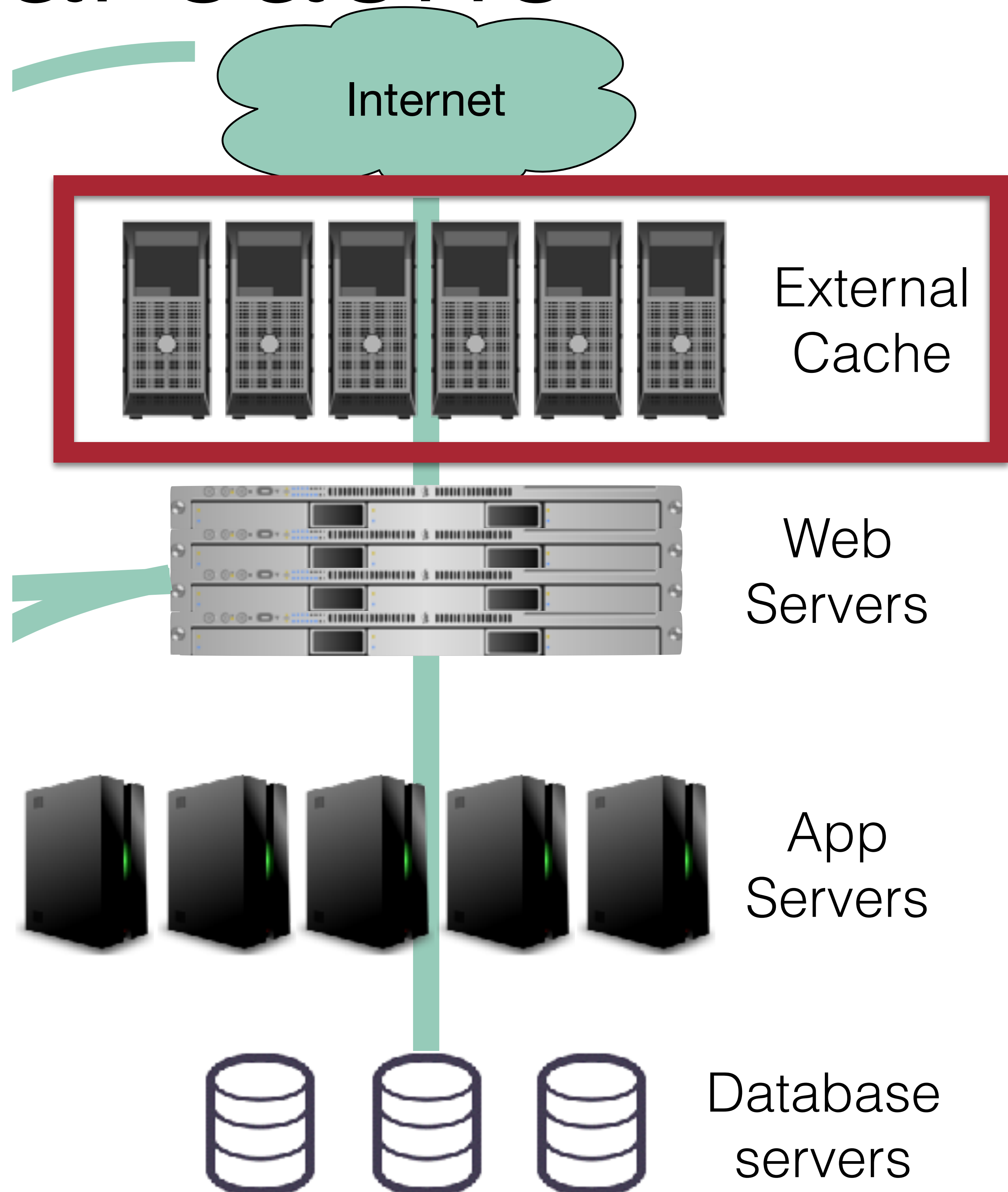
Real Architectures

- For each layer…
  - What is it?
  - Why?



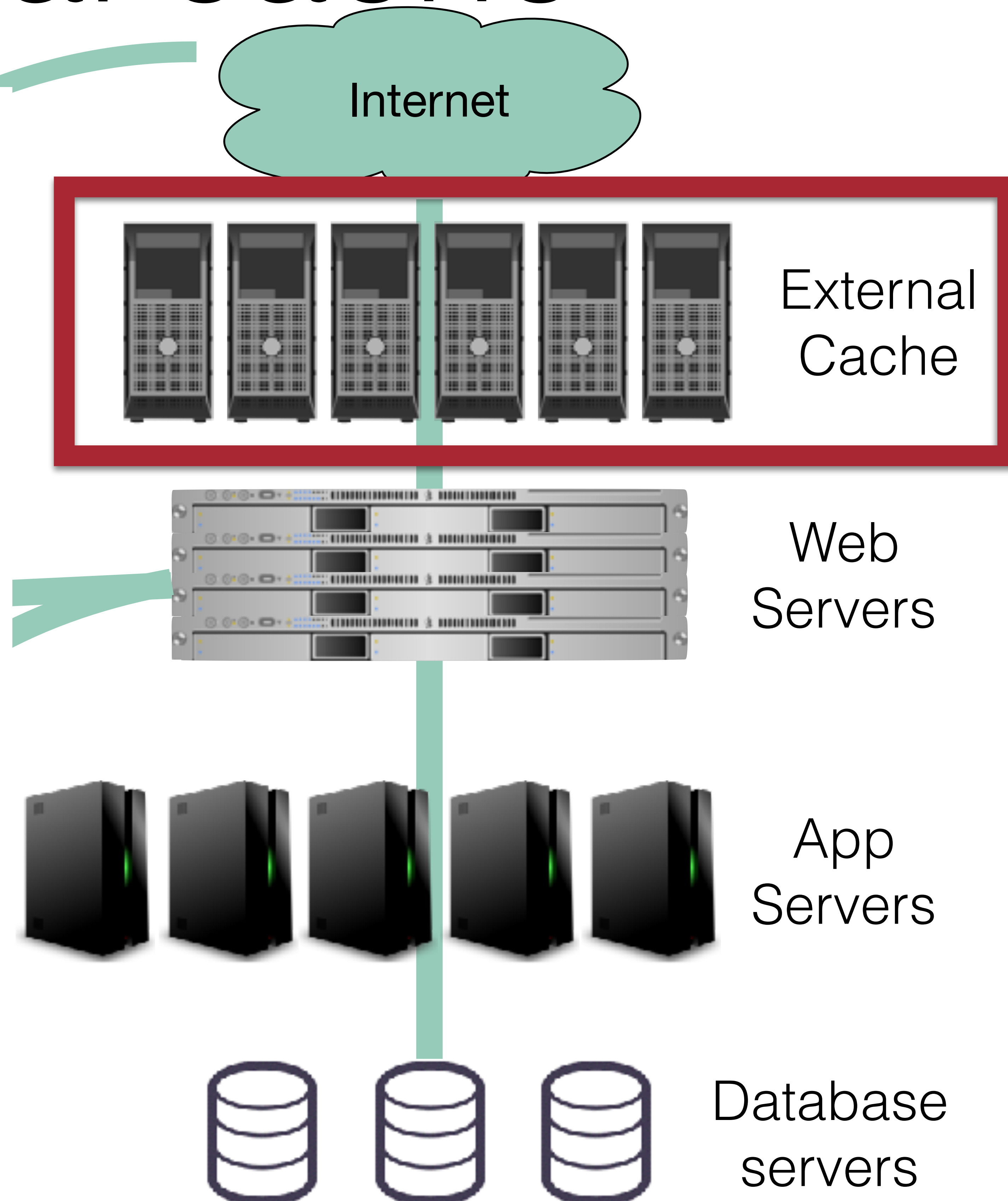
External cache

- What is it?
  - A proxy (e.g. squid, apache mod_proxy)
  - A content delivery network (CDN) e.g. Akamai, CloudFlare



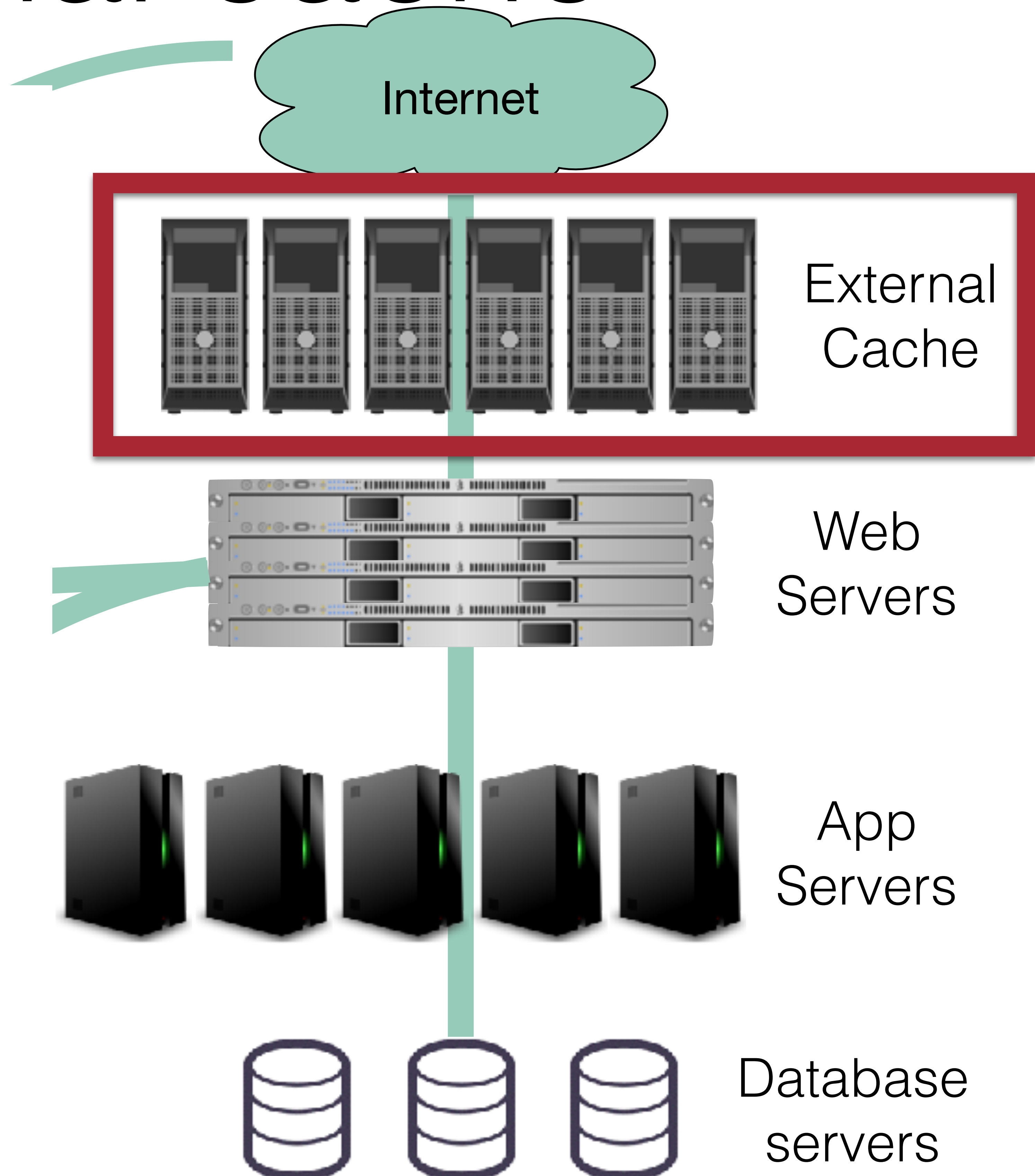
External cache

- What is it for?
  - Caches outbound data
    - Images, CSS, XML, HTML, pictures, videos, anything static (some stuff dynamic maybe)
  - DoS defense
  - Decrease latency - might be close to the user

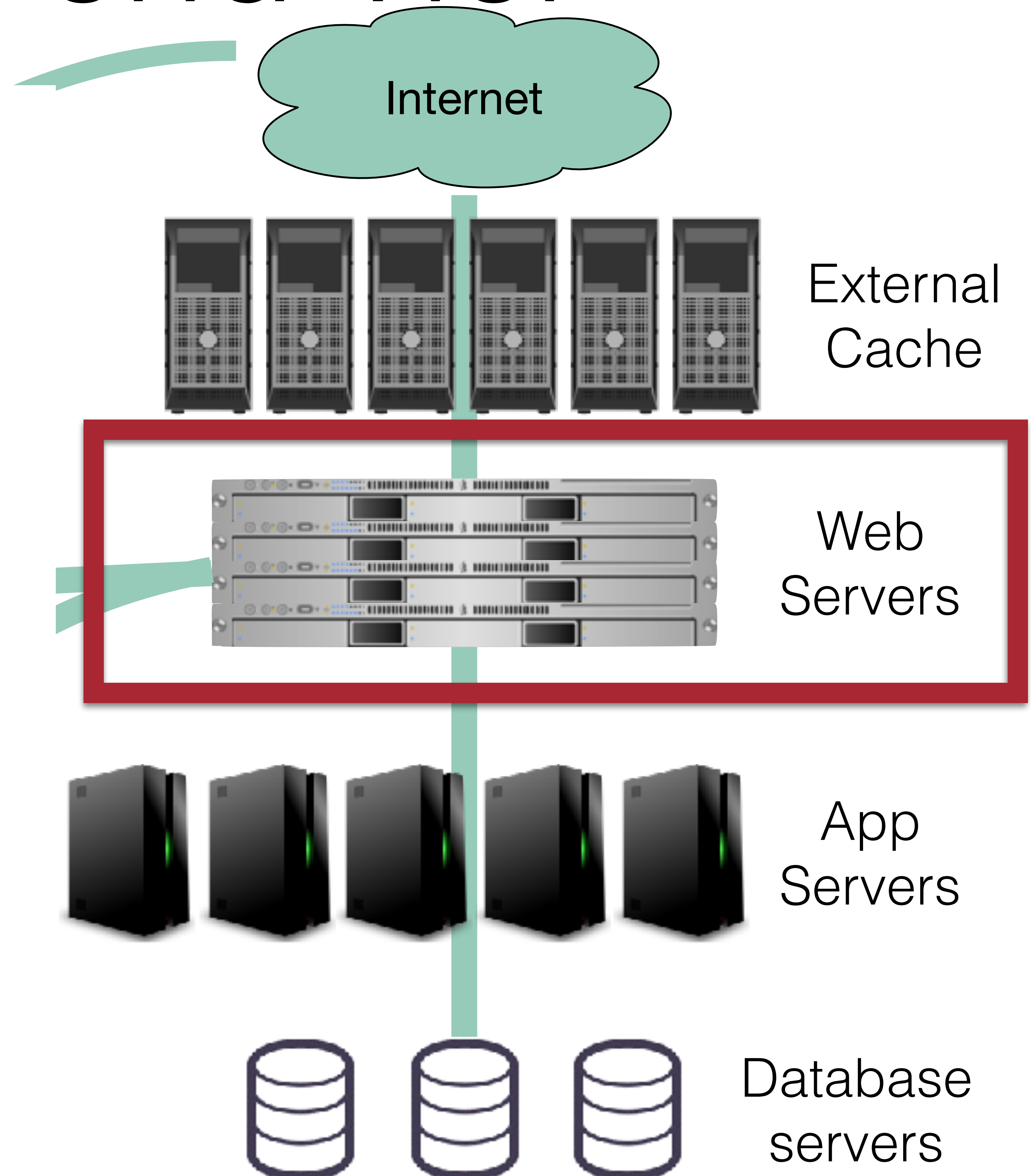


External cache

- What is it made of?
- Tons of RAM, fast network, physically located all over
- No need for much CPU



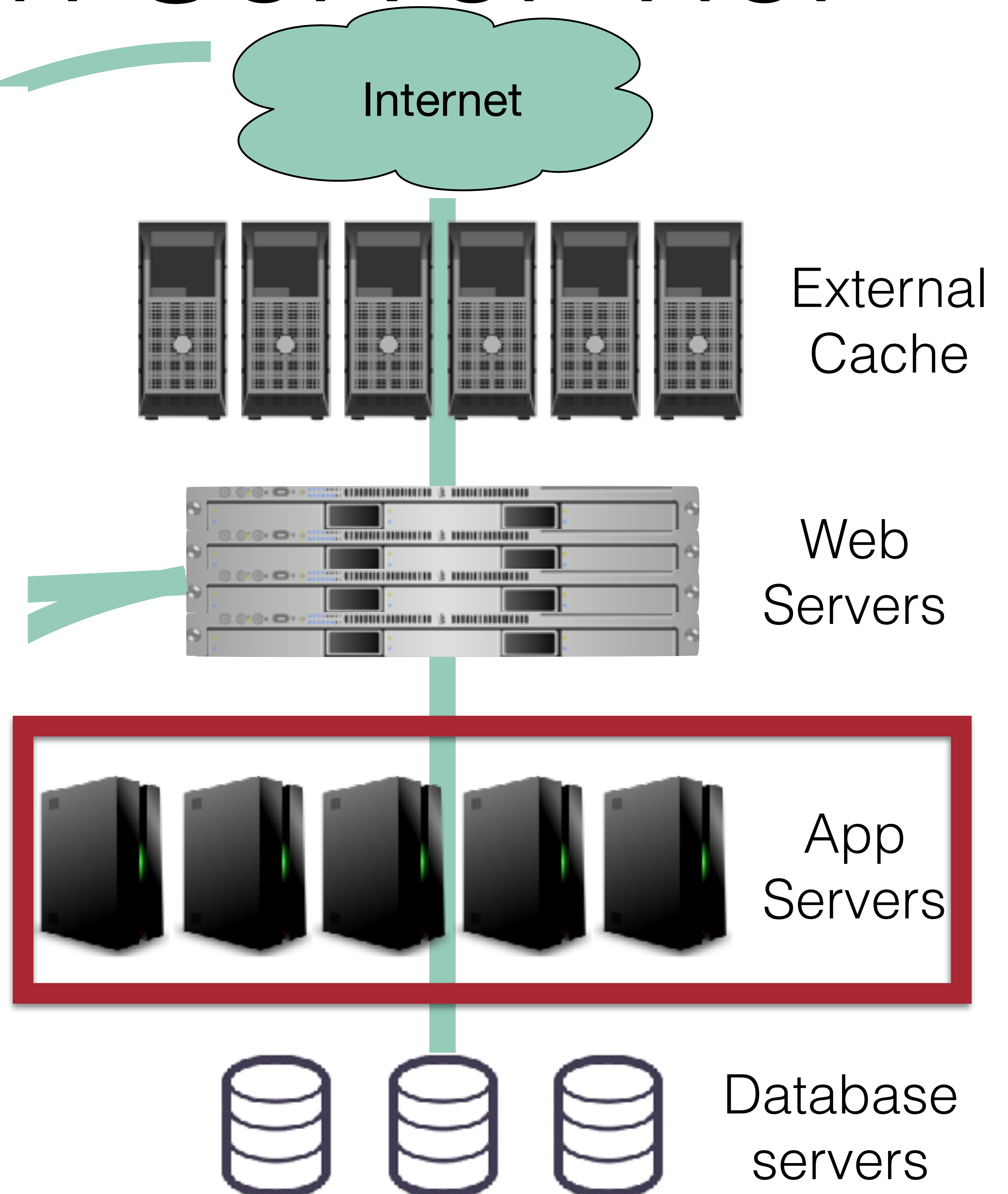
Front-end Tier



- Serves static content from disk, generates dynamic content by dispatching requests to app tier
- Speaks HTTP, HTTPS

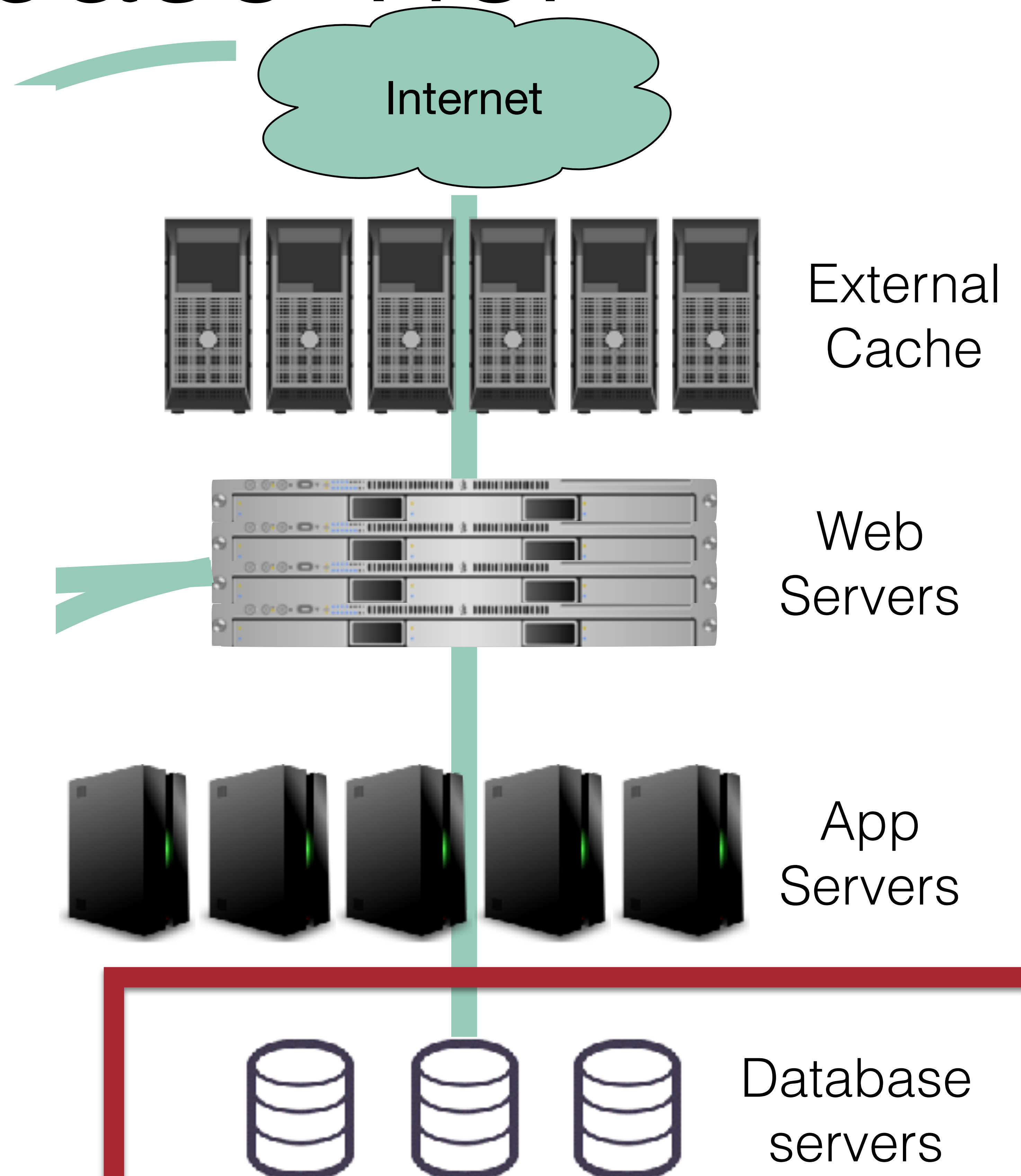
Application Server Tier

- Serves dynamic pages
- Provides internal services
  - E.g. search, shopping cart, account management
- Talks to web tier over..
  - RPC, REST, CORBA, RMI, SOAP, XMLRPC… whatever
- More CPU-bound than any other tier

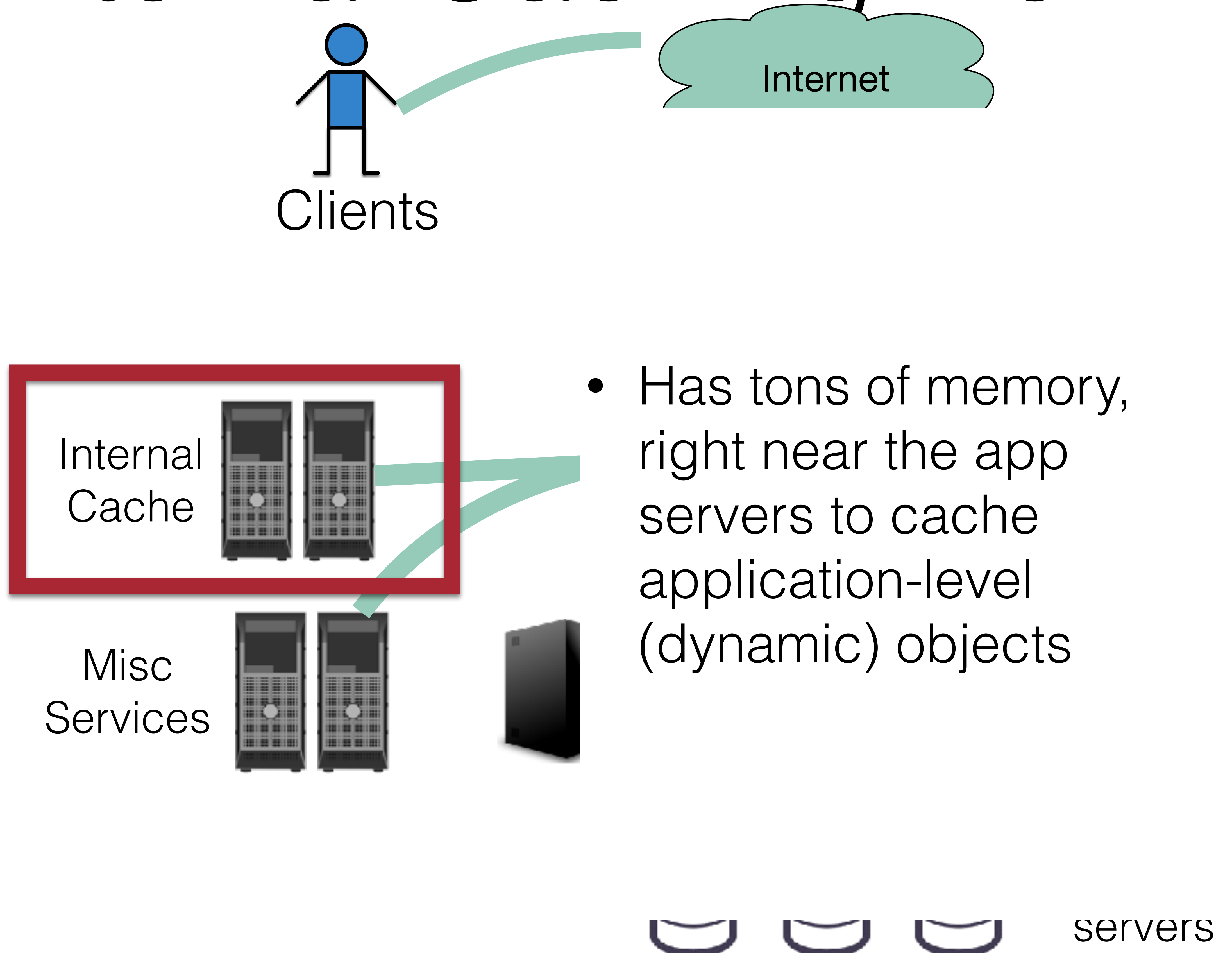


Database Tier

- Relational or non-relational DB
  - PostgreSQL, MySQL, Mongo, Cassandra, etc
- Most storage

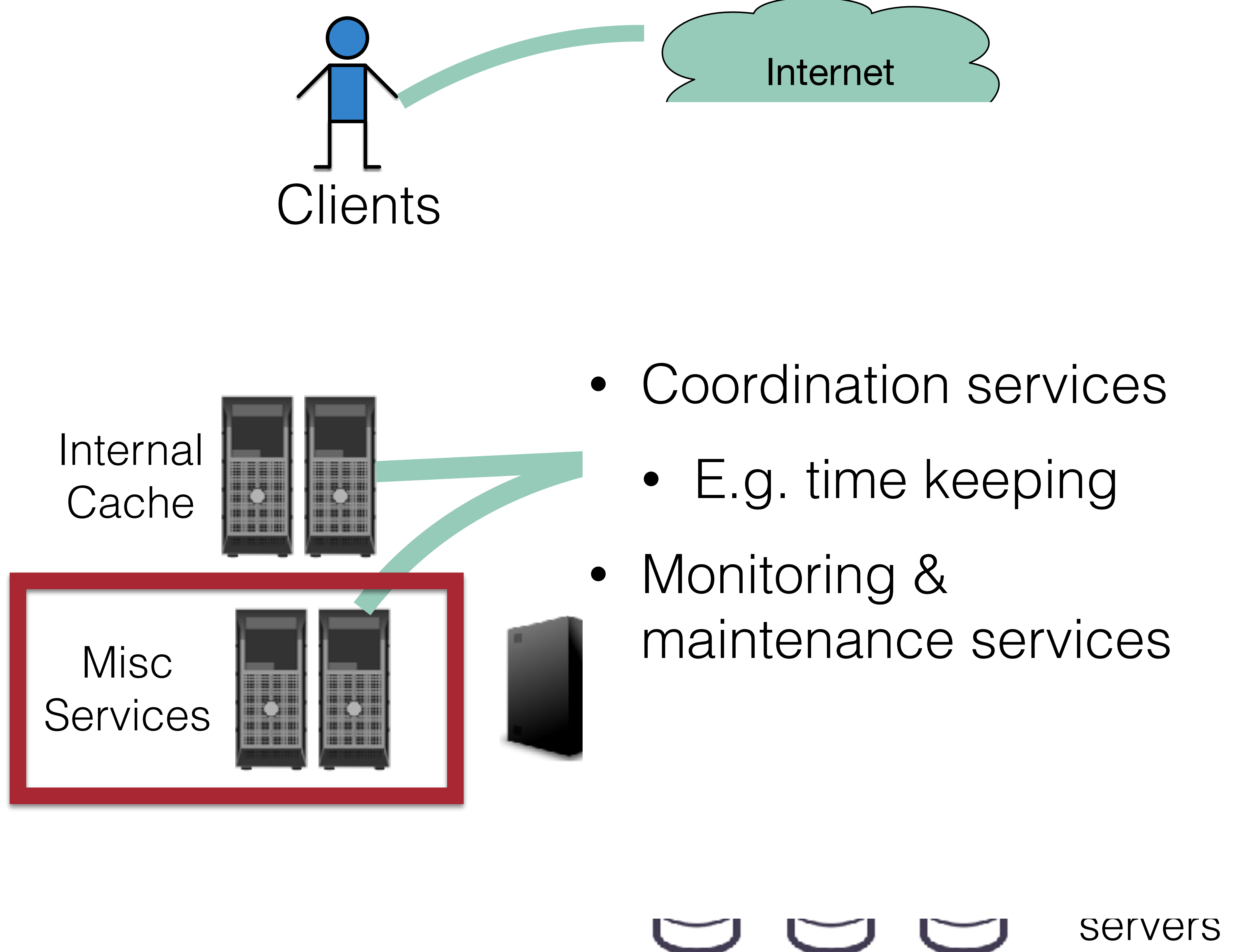


Internal Caching Tier



- Has tons of memory, right near the app servers to cache application-level (dynamic) objects

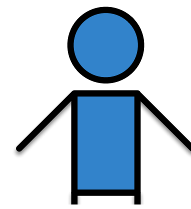
Internal Services Tier

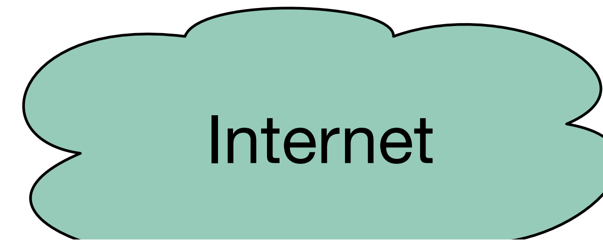


- Coordination services
  - E.g. time keeping
- Monitoring & maintenance services

Real Architectures

N-Tier Web Architectures





Separate out responsibilities with abstractions: each tier cares about a different aspect of getting the client their response

External Cache

Web Servers

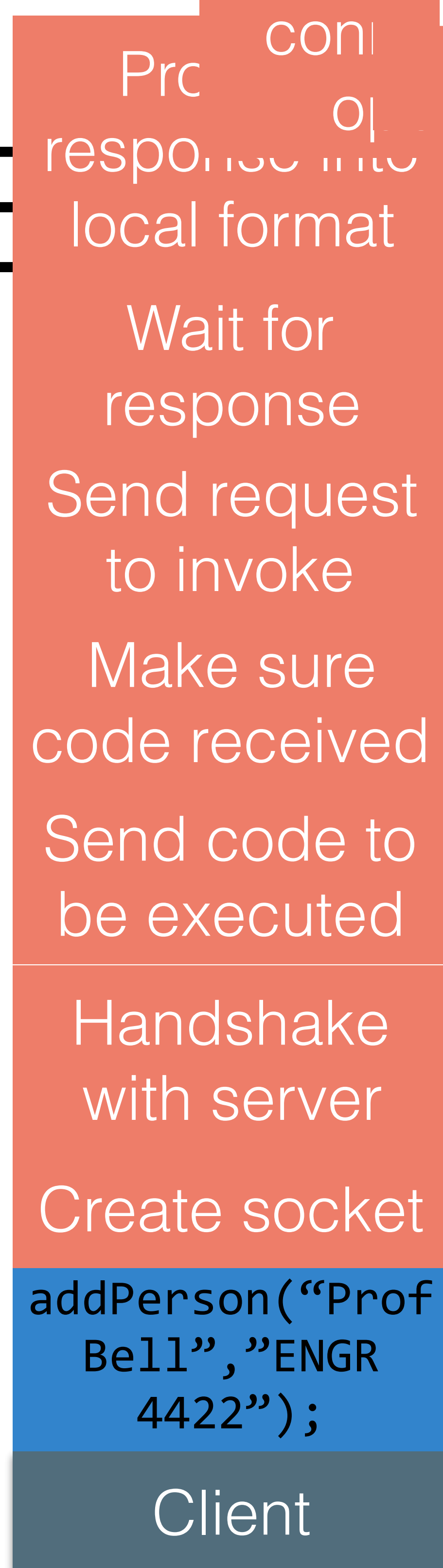
Internal Cache

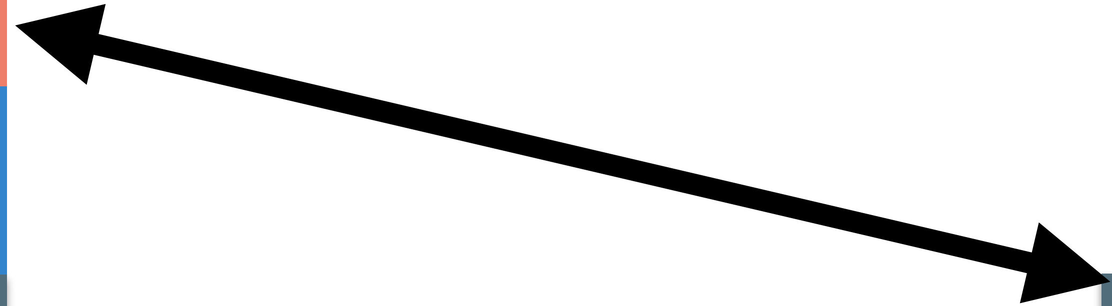
Misc Services

App Servers

Database servers

Example: invoking a method on server

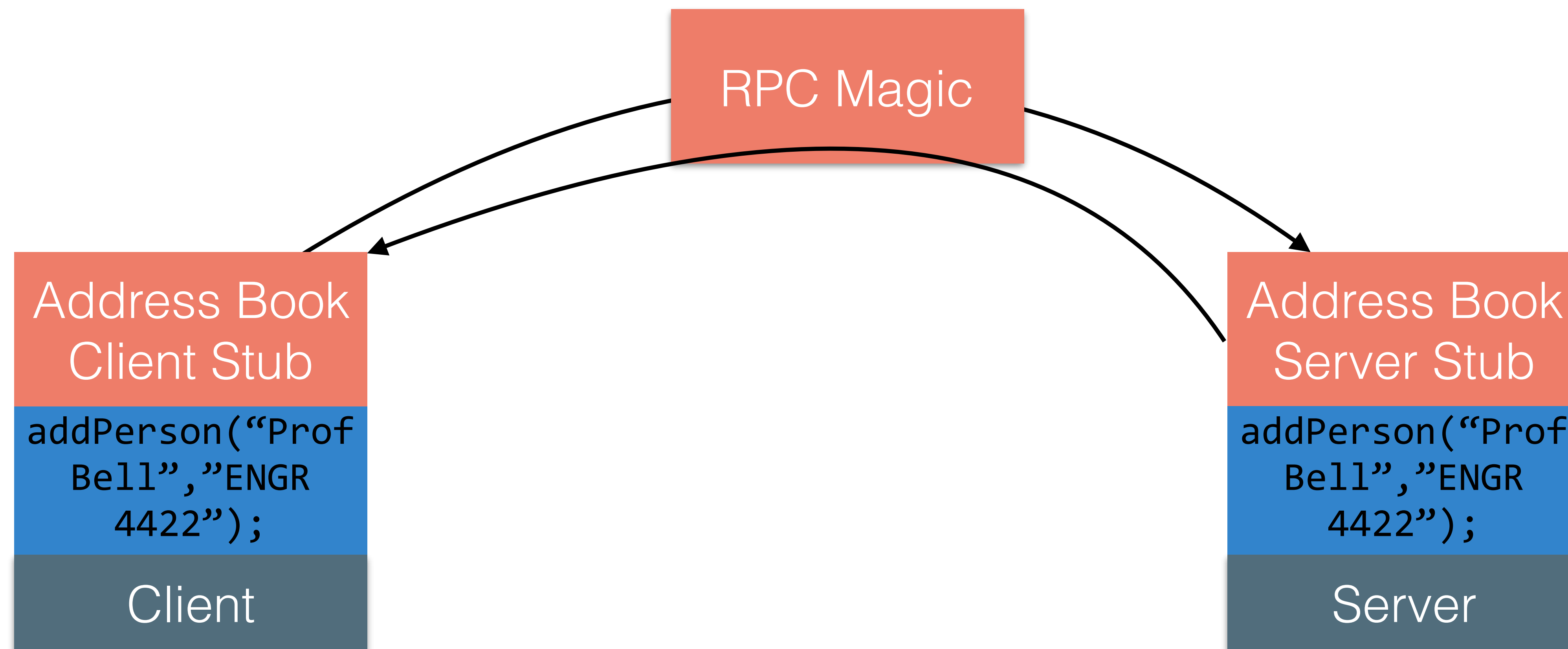




Server

Abstractions: Example

RMI (RPC) is a form of abstraction



Abstracting the tiers

- Take, for instance, this *internal cache*
- Can we build one really good internal cache, and use it for all of our problems?
- What is a reasonable model for the cache?
  - Partition: yes (get more RAM to use from other servers)
  - Replicate: NO (don't care about crash-failures)
  - Consistency: Problem shouldn't arise (aside from figuring out keys)

How much more can we abstract our system?

- At its most basic… what does a program in a distributed system look like?
  - It runs concurrently on multiple nodes
  - Those nodes are connected by some network (which surely isn't perfectly reliable)
  - There is no shared memory or clock
- So…
  - Knowledge can be localized to a node
  - Nodes can fail/recover independently
  - Messages can be delayed or lost
  - Clocks are not necessarily synchronized -> hard to identify global order

Back to reality

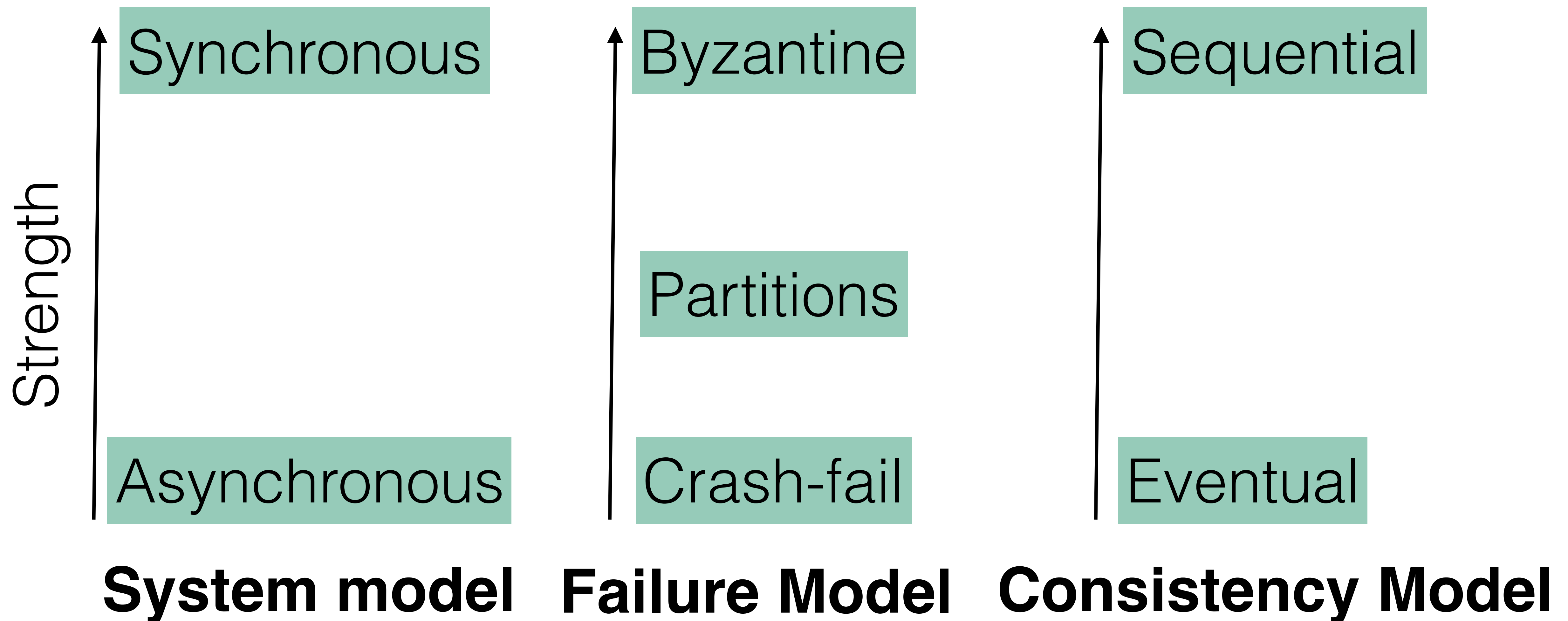
- That's a little TOO abstract - given that system, how can we define a good way to build one?
- In practice, we need to make assumptions about:
  - Node capabilities, and how they fail
  - Communication links, and how they fail
  - Properties of the overall system (e.g. assumptions about time and order)

Designing and Building Distributed Systems

To help design our algorithms and systems, we tend to leverage abstractions and models to make assumptions

Generally: Stronger assumptions -> worse performance

Weaker assumptions -> more complicated



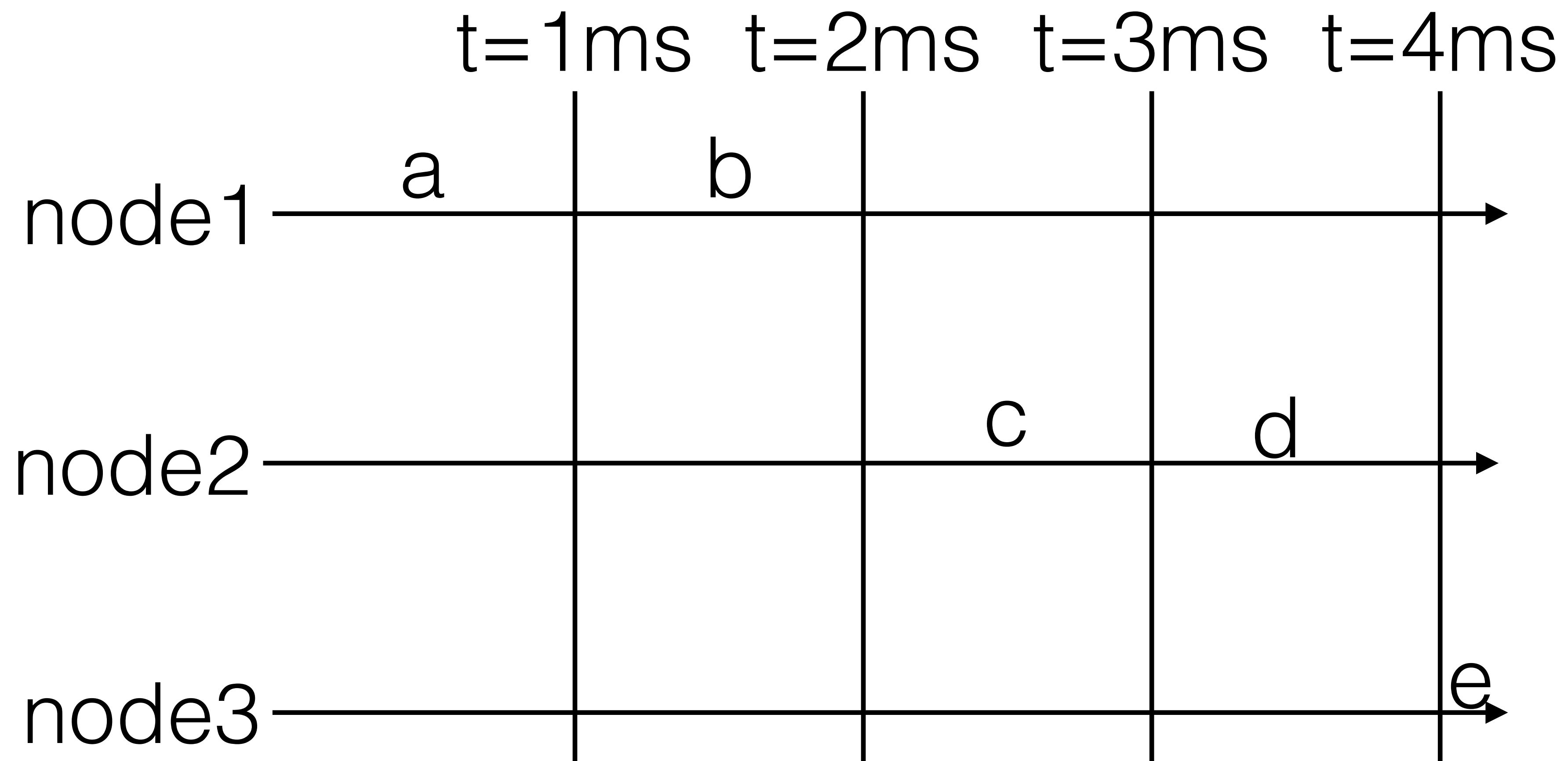
Timing & Ordering Assumptions

- No matter what, there will be *some* latency between nodes processing the same thing
- What model do we assume though?
- Synchronous
  - Processes execute in lock-step
  - We (the designers) have a known upper bound on message transmission delay
  - Each process (somehow) maintains an accurate clock

Modeling network transmissions

- Assuming how long it can take a message to be delivered helps us figure out what a failure is
- Assume (for instance), messages are always delivered (and never lost) within 1 sec of being sent
- Now, if no response received after 2 sec, we know remote host failed
- Typically NOT reasonable assumptions

Modeling Clocks: Global Ordering

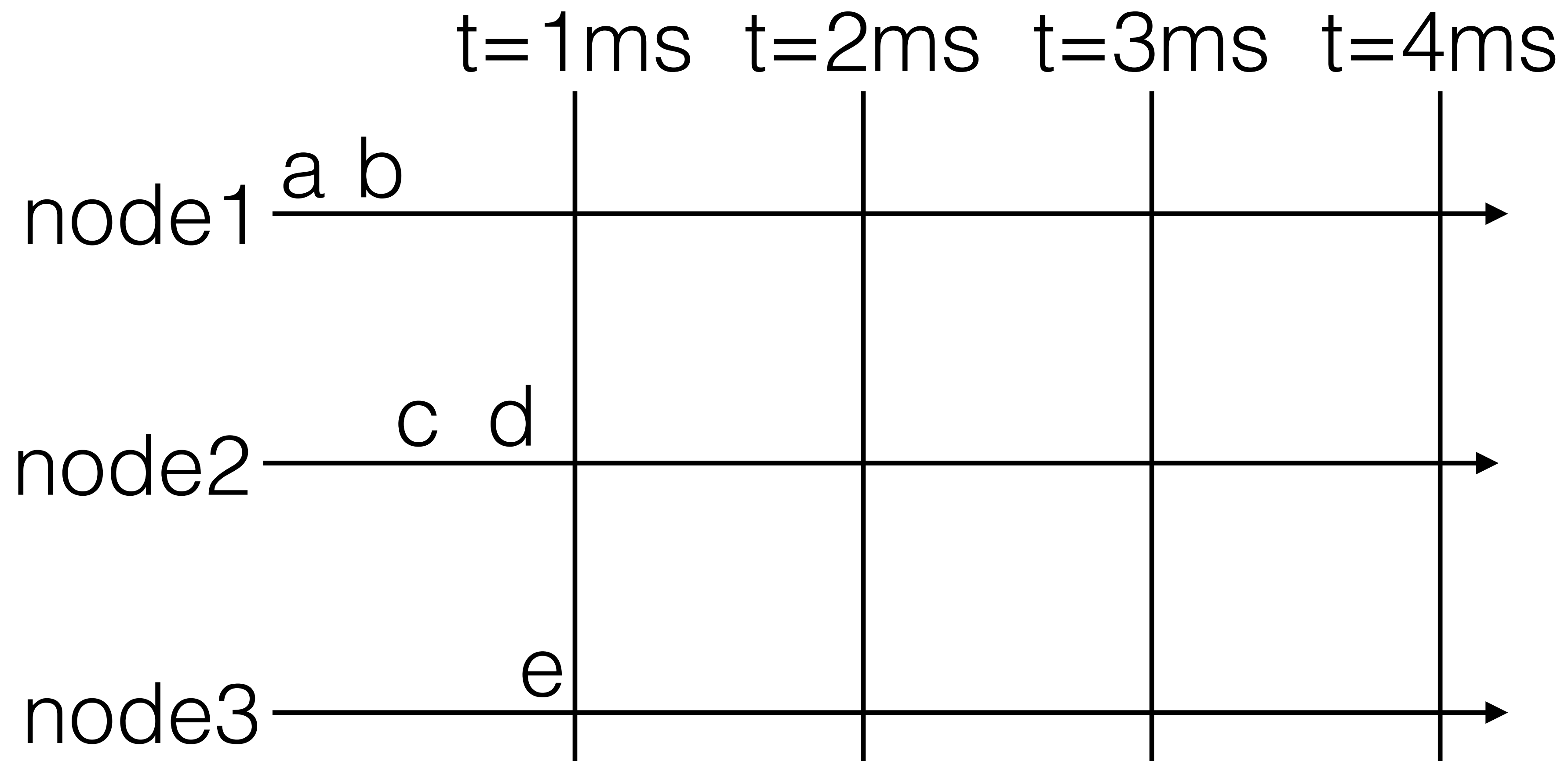


How do we know what the *actual* order of these events was?

Modeling clocks

- Our synchronous system (and any other really) needs some notion of time
- Are we doing things at the same time? Are they in order?
- One option… synchronize clocks (i.e. via NTP)
- NTP is a great way to automatically synchronize computer clocks, with accuracy of?
    - ~1ms between in local networks, ~10ms across internet

Modeling Clocks: Global Ordering

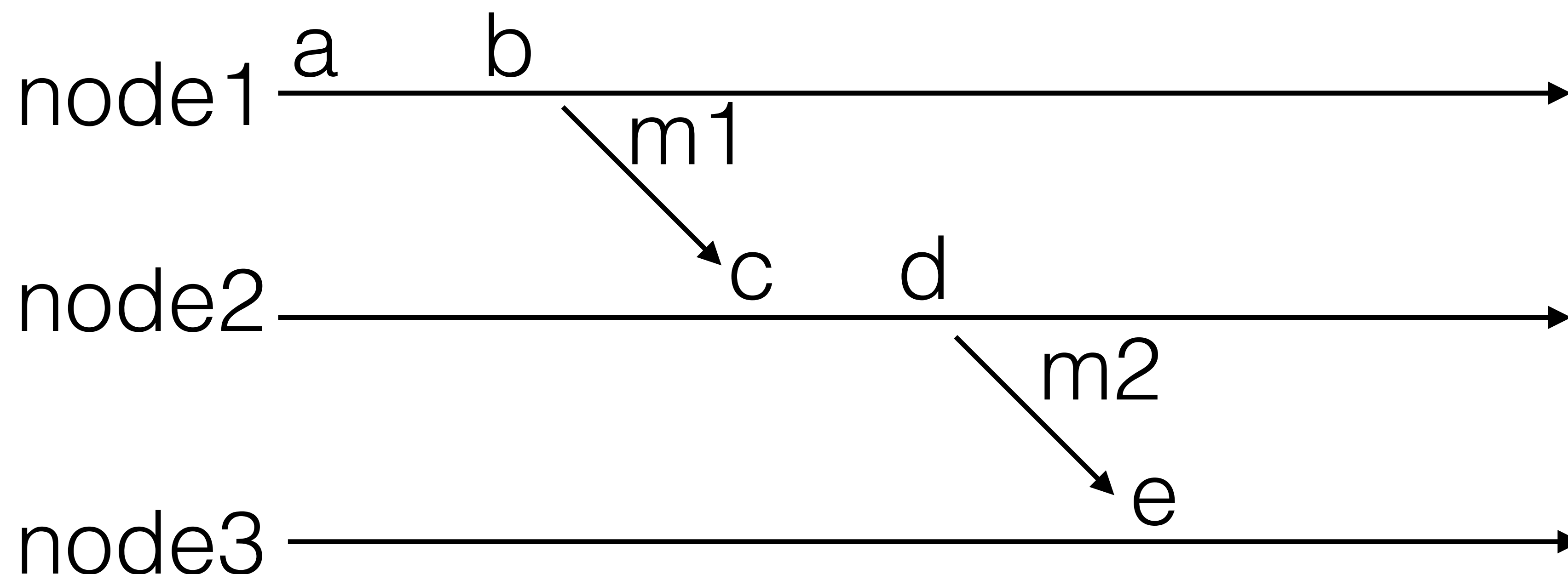


How do we know what the *actual* order of these events was?

Lamport Clocks

- Idea:
  - We just care about “happens-before”
  - NOT the infinitesimal granularity of time
- Don’t track time in an absolute sense
- Just track time relative to things that you care about
- Within a process:
  - You can keep track locally
- Between processes:
  - Send a message… and assume that when you receive it is after when you sent it

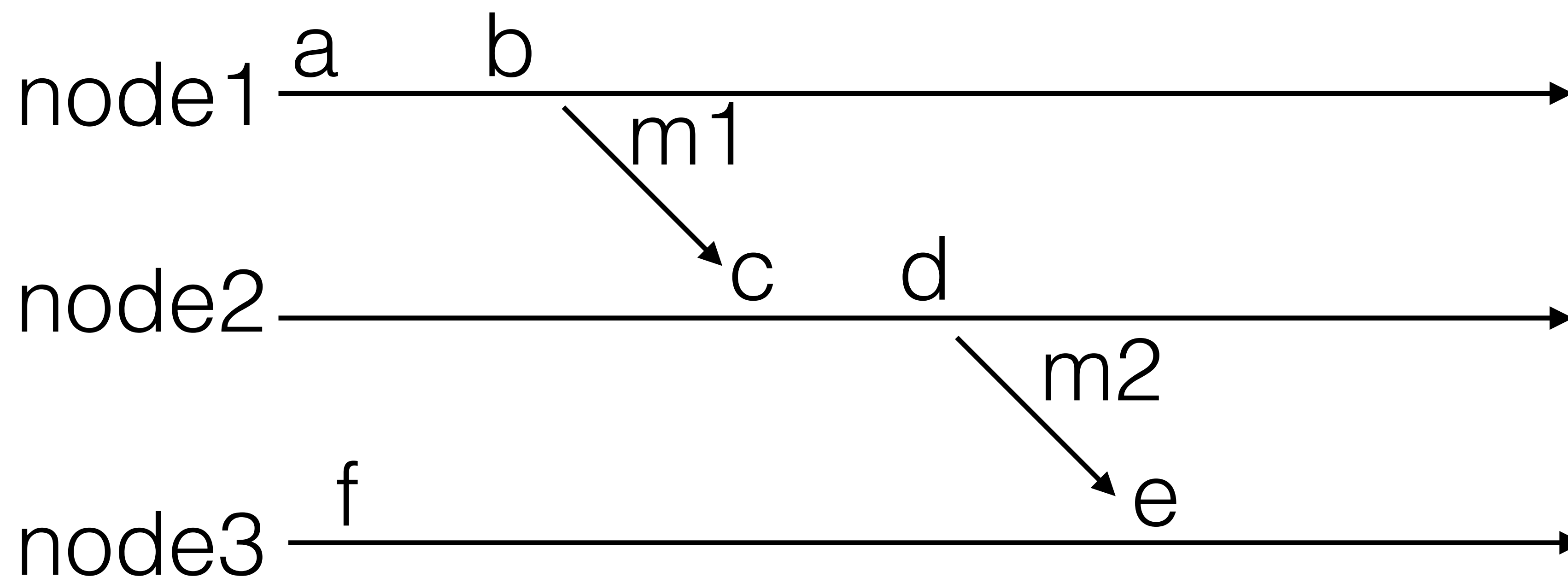
Modeling Clocks: Global Ordering



Node1 sends m1 *after b,* Node2 receives it *before c*, then sends m2 *after* c, which is received by node3 *after* d

Hence, the global order is:
a b c d e

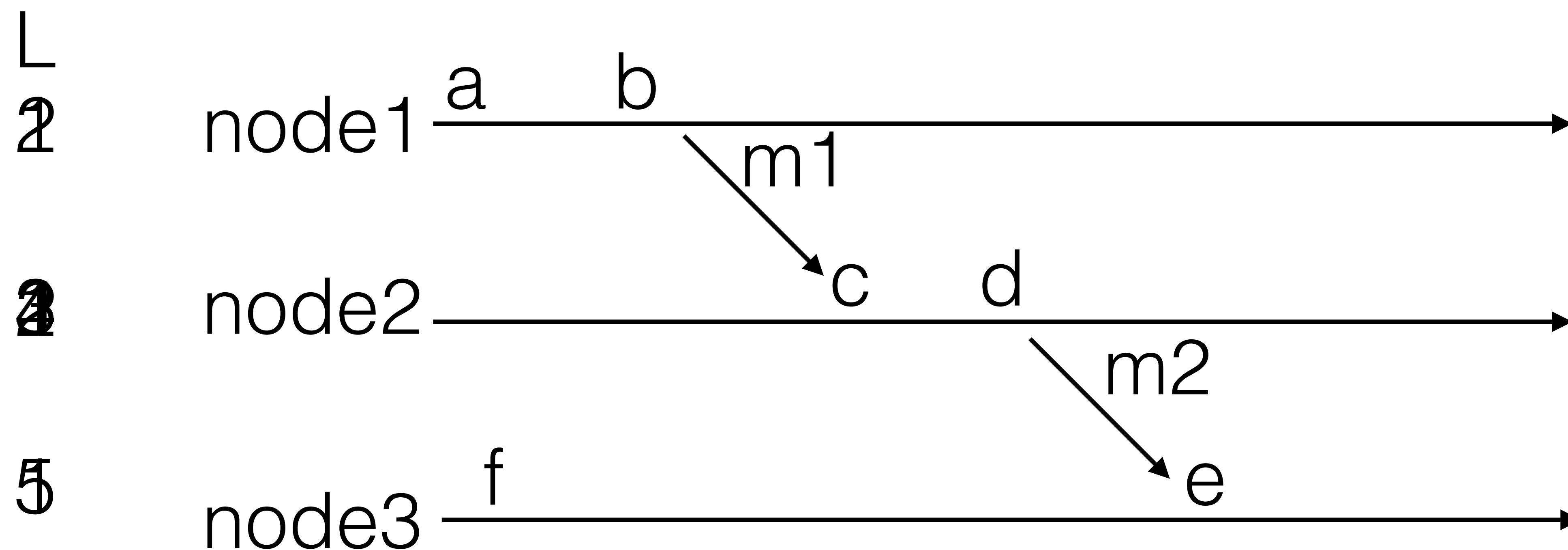
Modeling Clocks: Global Ordering



Not all events are related

a and *f* are *concurrent* - we don't know which happened first (maybe didn't care?)

Modeling Clocks: Global Ordering



Implementation: Each node maintains a logical clock L

- L is incremented by 1 before each event at each process
- When a process sends a message m, it sends its current time counter
- When a process receives a message, it updates its clock L from that message as the max of it and its time

Logical Clocks

- Lamport clocks are a *logical clock*
  - Encodes a causality relationship
  - They are relative: if e1 happened before e2, then L(e1) < L(e2)
- Vector clocks: similar to Lamport clocks, also provide exact causality
  - e1 happens before e2 means V(e1) < V(e2)
  - ALSO: if V(e1) < V(e2) that means e1 happens before e2

Back to synchronous models…

- So, it'll be possible to ensure that one event doesn't occur before another
- It won't be free
    - We need to keep these clocks in sync by sending messages back and forth!
- We now have a new way to fail…
    - If these clock messages start getting dropped

Asynchronous Systems

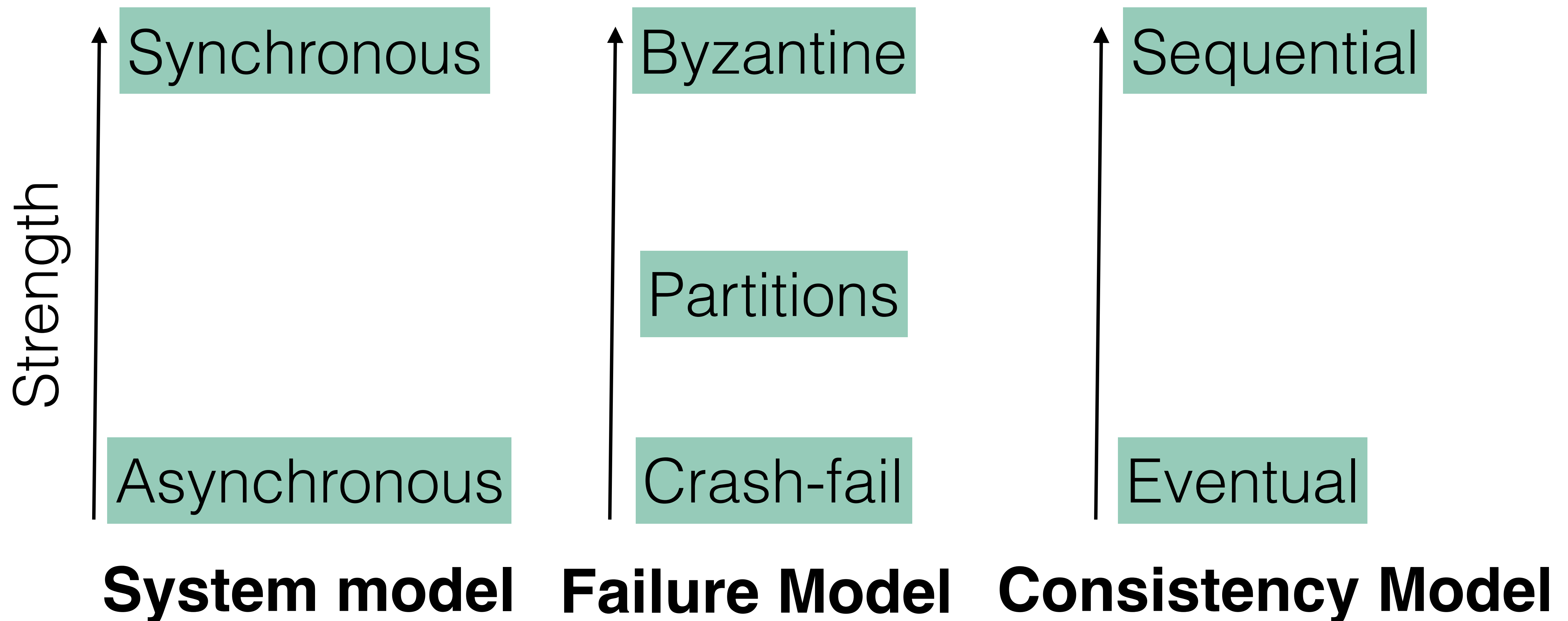
- The exact opposite
- Do not rely on *any* timing assumptions
- Processes execute at their own rates
- Messages can be delayed indefinitely
- No useful clocks

Designing and Building Distributed Systems

To help design our algorithms and systems, we tend to leverage abstractions and models to make assumptions

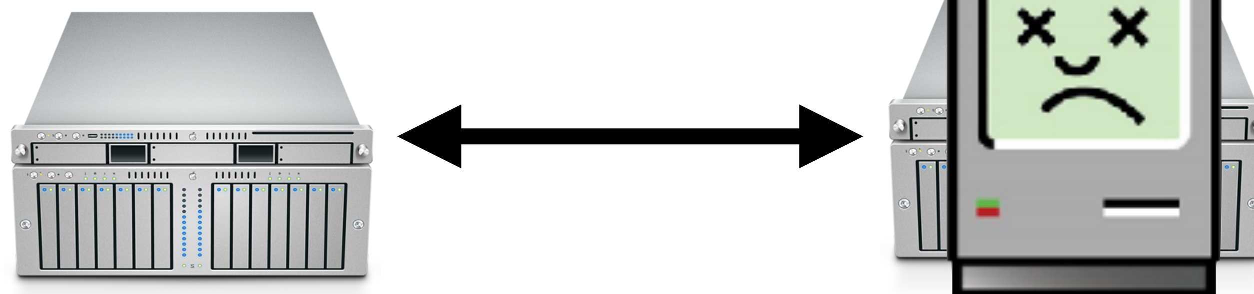
Generally: Stronger assumptions -> worse performance

Weaker assumptions -> more complicated



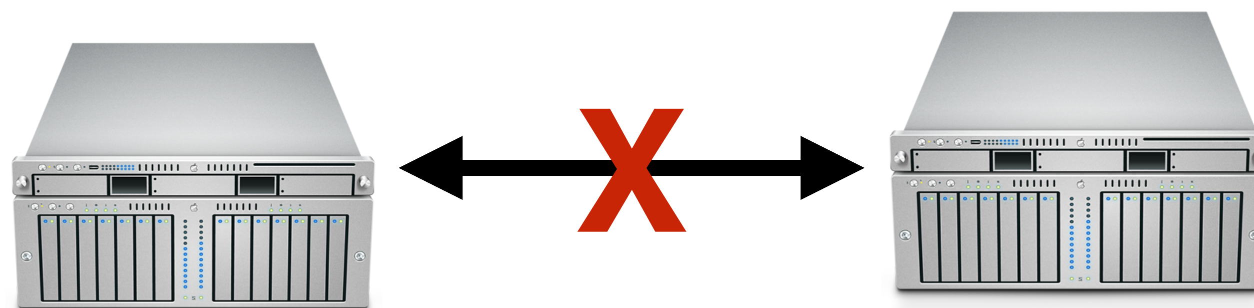
Nodes in our system

- Nodes…
  - Run programs
  - Store data in volatile memory (lost on crash) and stable storage (persists past some crashes)
  - Have some clock (may or may not be accurate)
- Key assumptions:
  - Nodes should typically behave *deterministically* given the same inputs/messages/system state
  - Nodes are well-behaved, and fail by crashing (which they might be able to self-recover from)

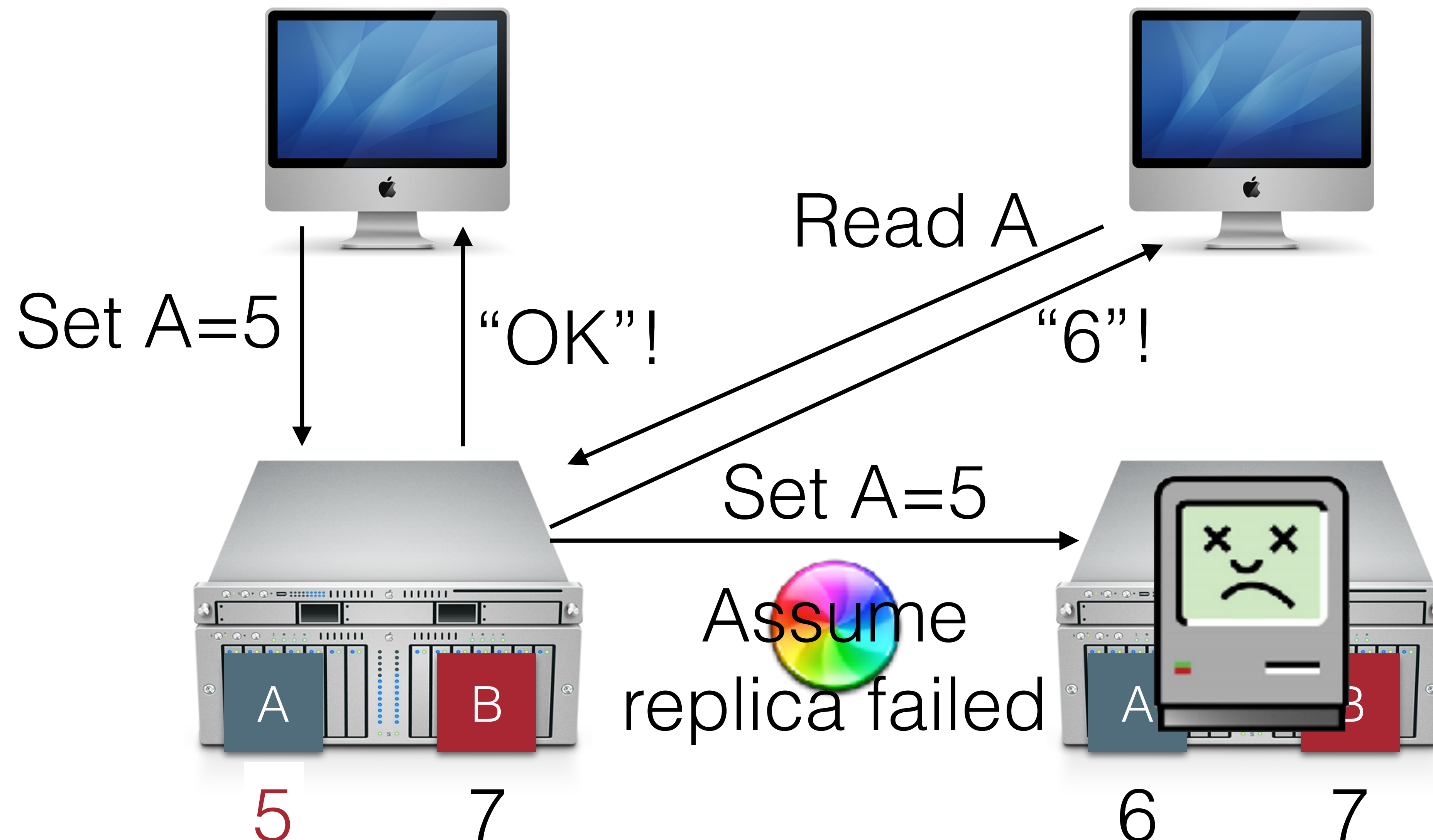


Communication Links in our System

- Does the network ensure ordering?
  - E.g. FIFO
- Is the network reliable?
  - E.g. will all messages eventually be delivered?
- Typically we make no other assumptions about the network
- Fail via *partition*

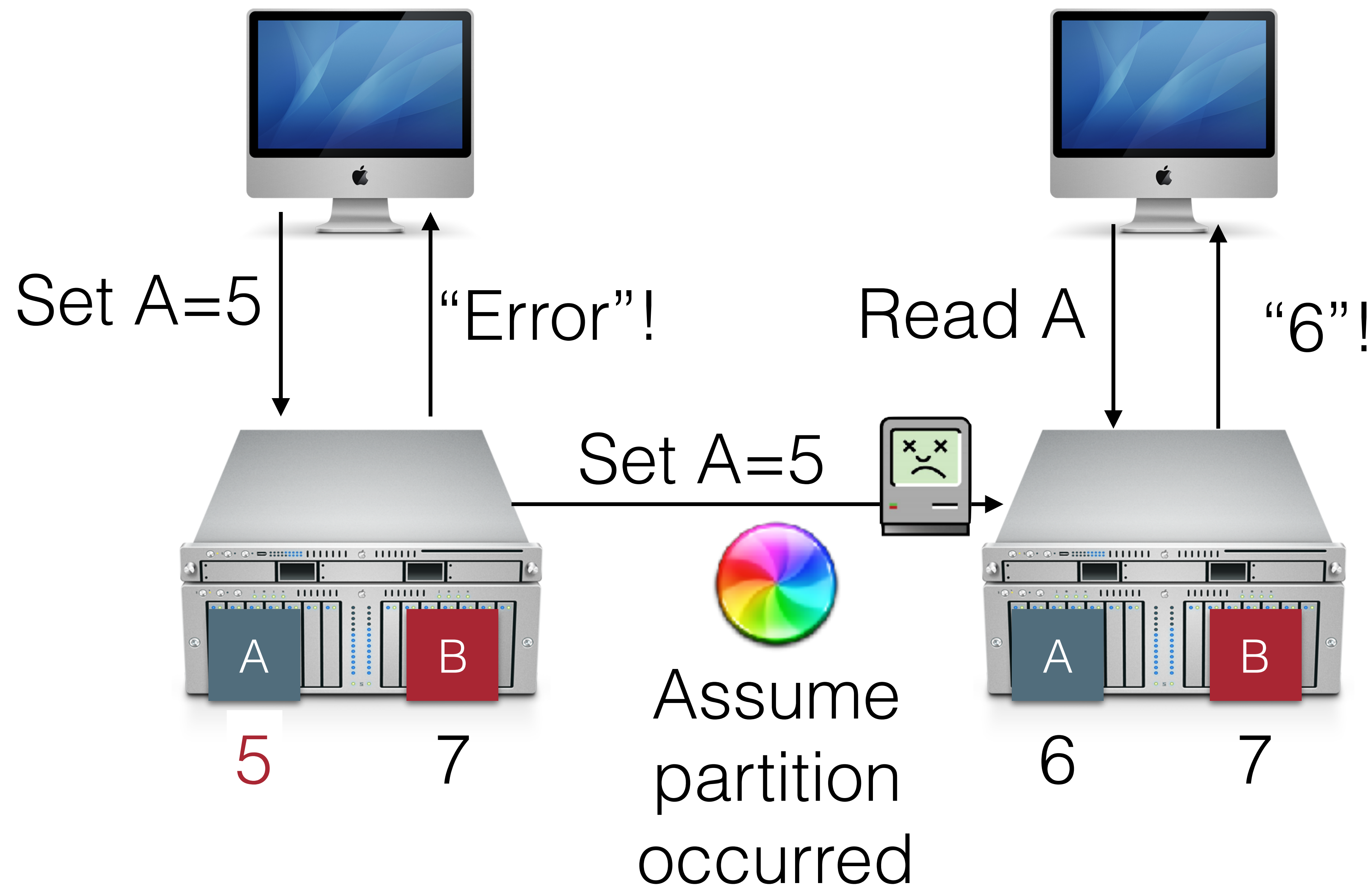


Failure model - review



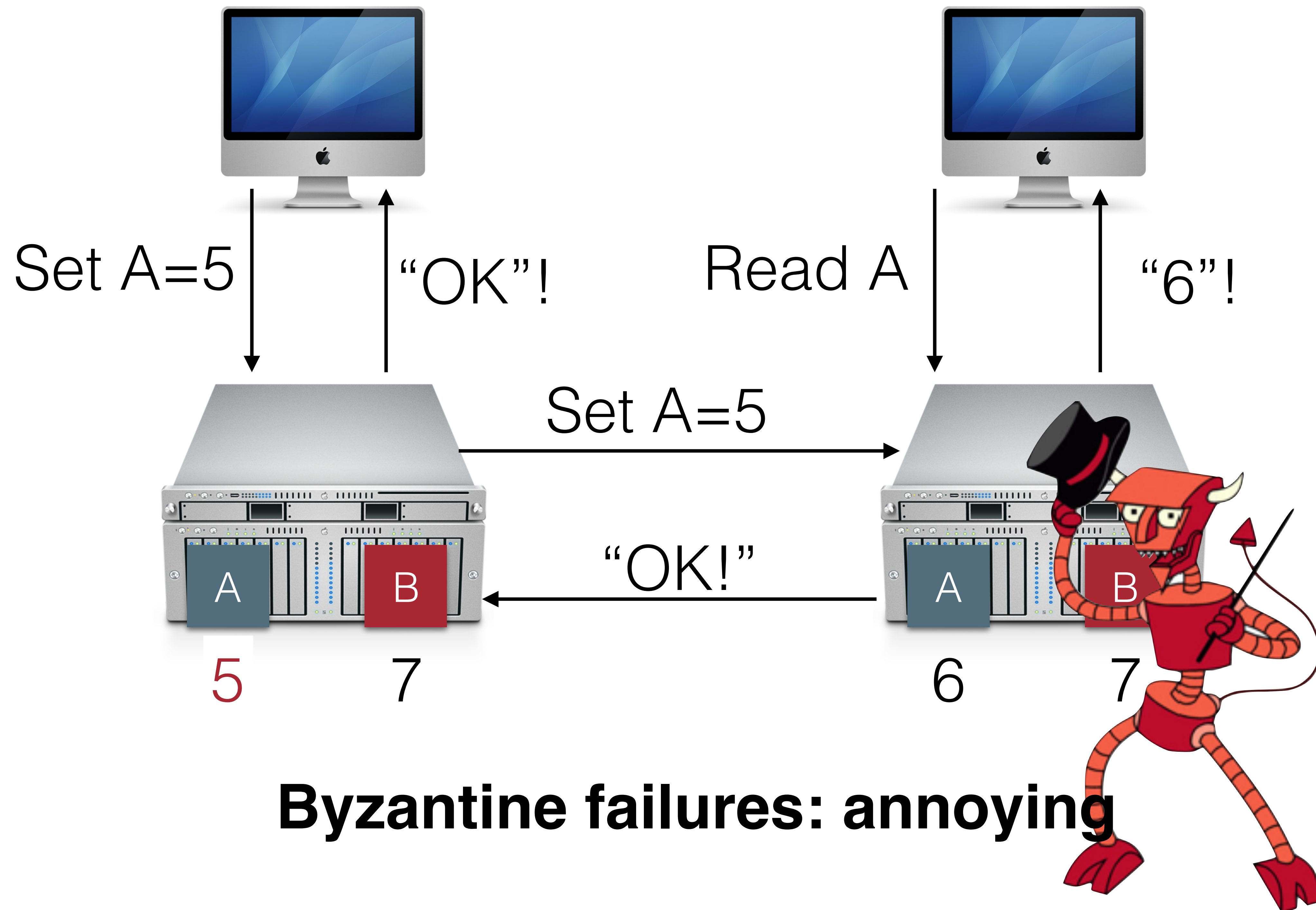
Tolerating crash-failures

Failure model - review



Tolerating partition-failures

Failure model - review

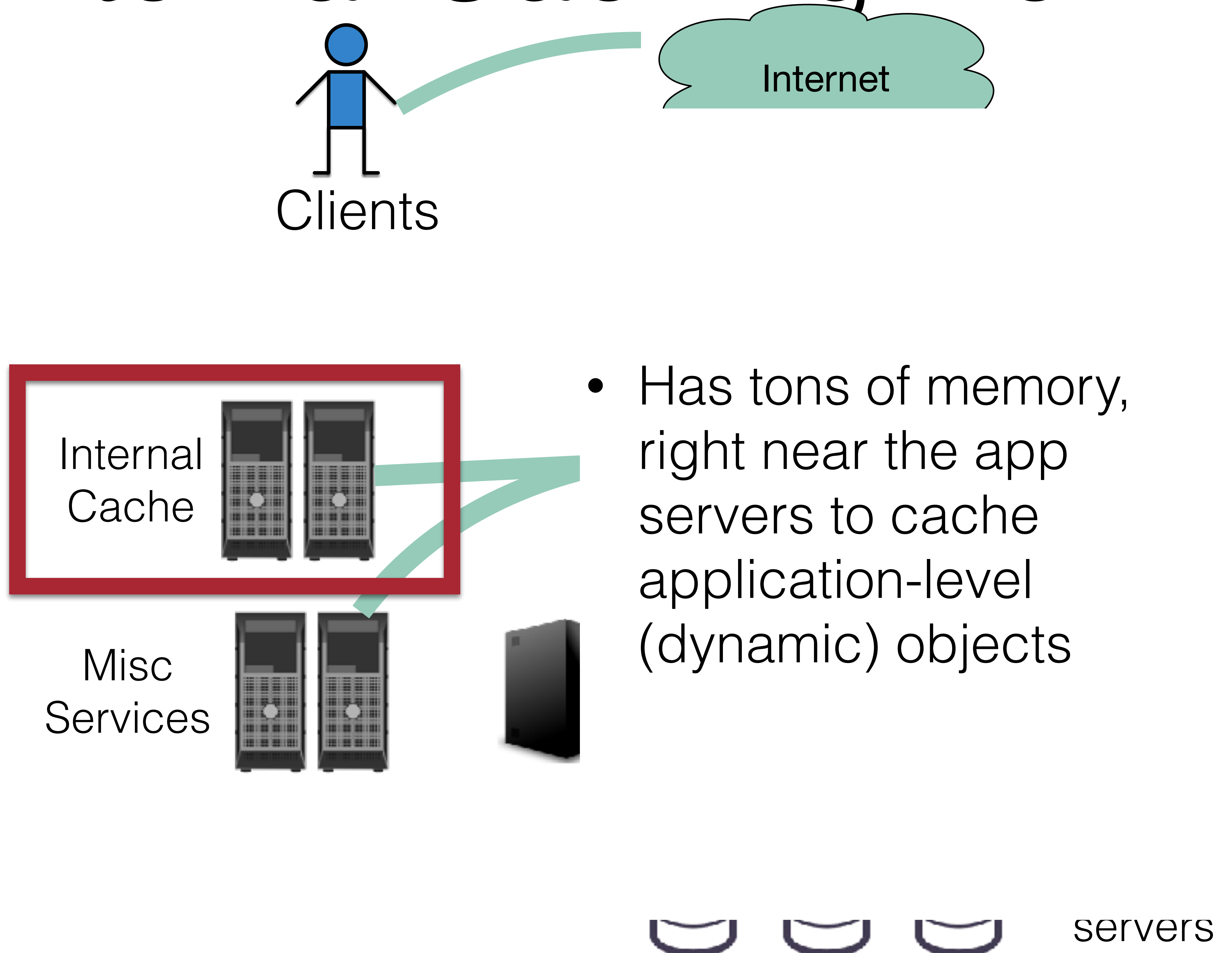


Byzantine failures: annoying

Consistency

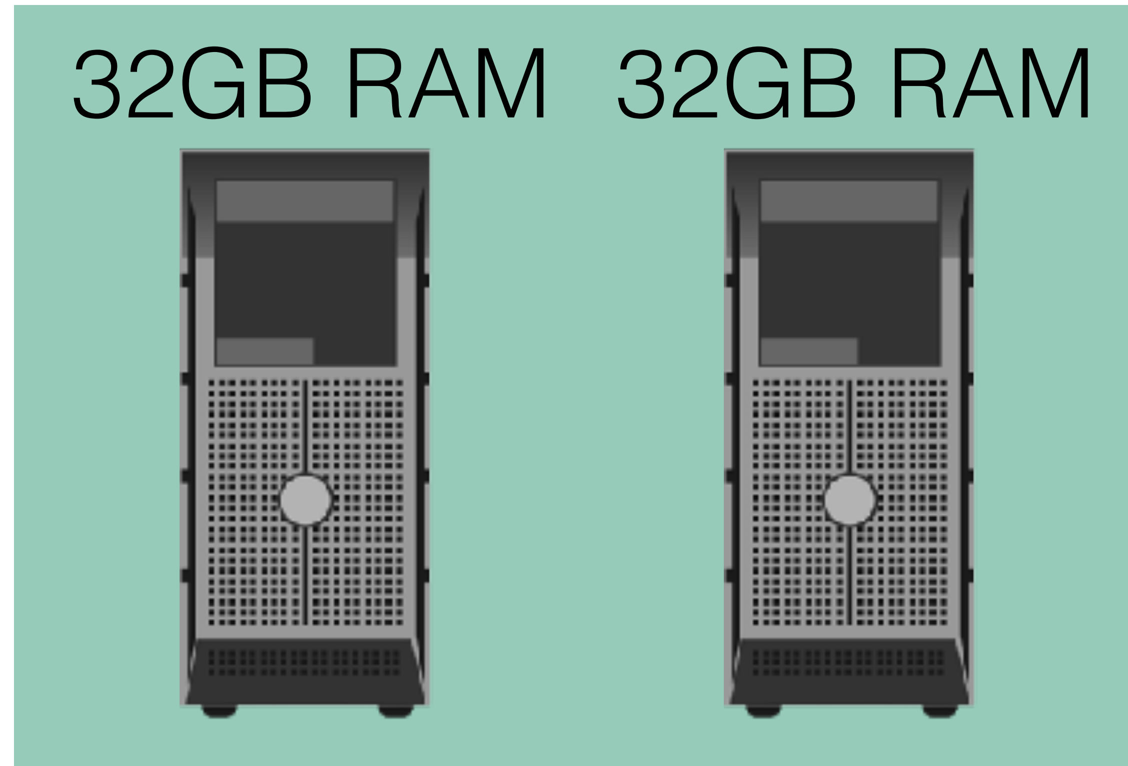
- We'll spend the next 2 lectures talking about what it means to be consistent, or partially consistent

Internal Caching Tier



- Has tons of memory, right near the app servers to cache application-level (dynamic) objects

A sharded cache



Total: 64 GB cache

Redis

- Redis is an in-memory caching server
- There are plenty of others (memcached is also popular and Redis was heavily influenced by it)
- Redis supports master-slave replication too for availability
- Asynchronously stores data to disk
- Operates directly on data structures (e.g. lists, sets, maps)

Redis

- Java API: Jedis
  - Uses connection pooling to be thread-safe
- (Documentation)
- Requests are processed asynchronously
- Has a mechanism for doing locking: DO NOT USE IT
  - When you try to replicate Redis, its locking is very not-safe (although it claims to be), and given that we're going to be replicating it next assignment, let's not get in the habit. https://aphyr.com/posts/283-jepsen-redis

Lab: Redis

- Address book
- Reddis stores the address book
  - Set stores list of names
  - Map stores info on individuals
- RMI handles locking
- This will be VERY similar to the challenges you see in HW2!!!