CE 528 Cloud Computing

Lecture 3: Introduction to Distributed Systems
Fall 2025

Prof. Yigong Hu

BOSTON UNIVERSITY

Slides courtesy of Chang Lou and Armando

Administrivia

We have sent an email with the project assignments.

- Please double-check your email if you did not see it.

Each student will receive **$150 in cloud credits**.

- $50 in Google Cloud credits
- $100 in AWS credits

Start your project description

- Template
- example

A Berkeley View of Cloud Computing

2/09 White paper by RAD Lab PI's/students

Goal: stimulate discussion on *what's new*

- Clarify terminology
- Quantify comparisons
- Identify challenges & opportunities

UC Berkeley perspective

- industry engagement but no axe to grind
- users of CC since late 2007

Why Now (not then)?

The Web "Space Race": Building-out of extremely large datacenters (10,000's of commodity PCs)

Driven by growth in demand (more users)

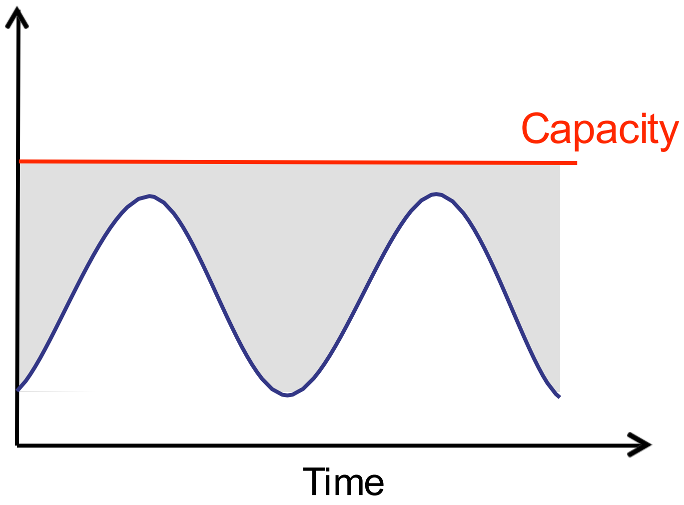
- Infrastructure software: e.g., Google File System
- Operational expertise
- Discovered economy of scale: 5-7x cheaper than provisioning a medium-sized (100's machines facility)

More pervasive broadband internet

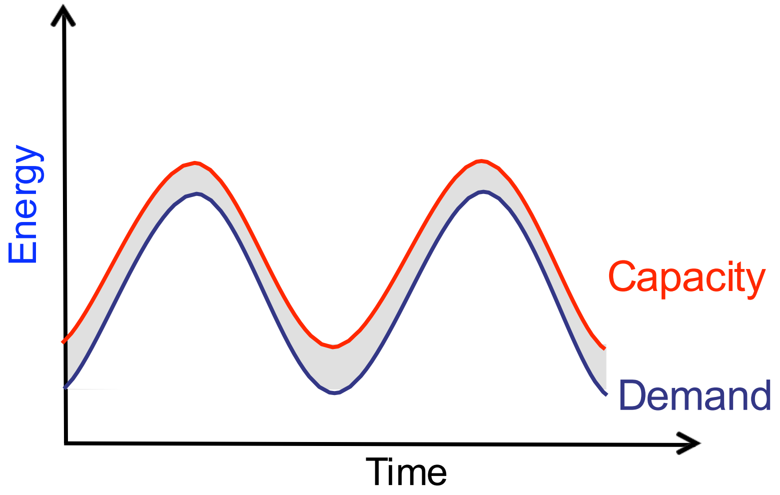
Free & open source software

Cloud Economics 101

Static provisioning for peak - wasteful, but necessary for SLA



“Statically provisioned” data center

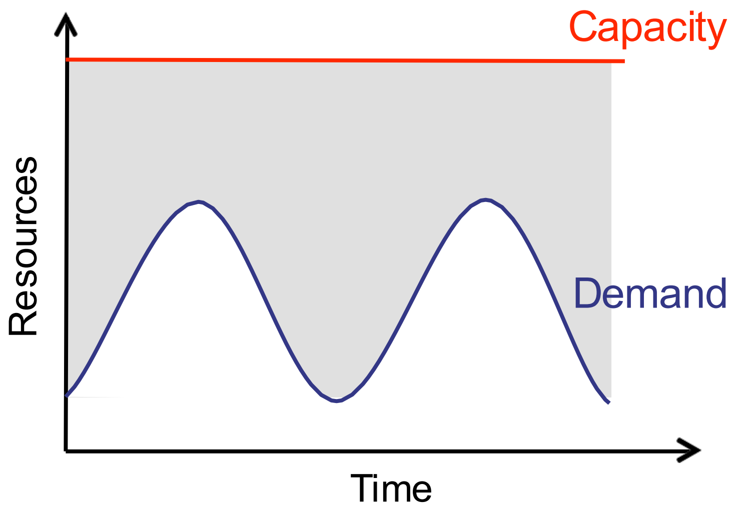


“Virtual” data center in the cloud

Unused resources

Risk of User Provisioning

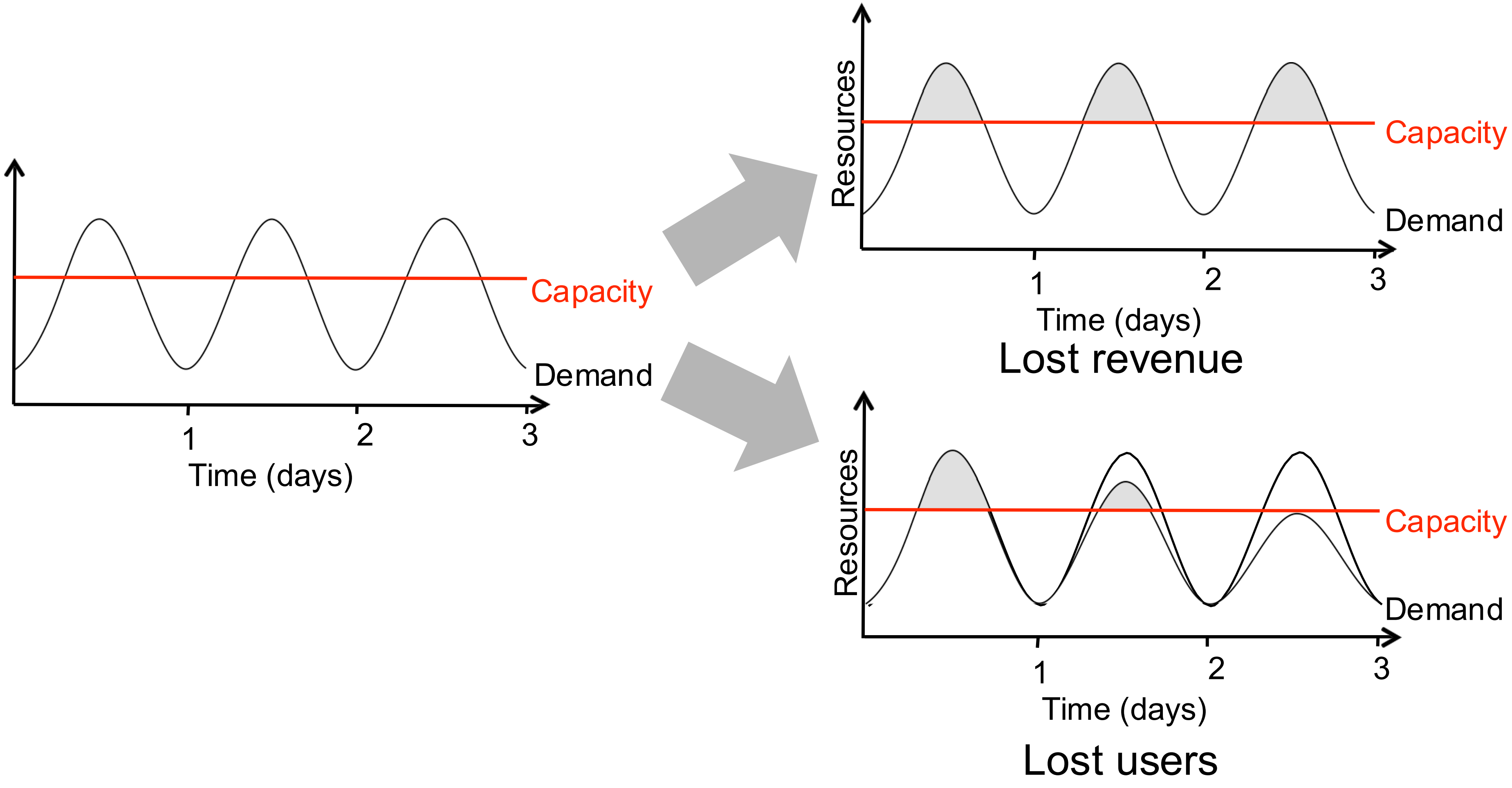
Underutilization results if “peak” predictions are too optimistic



Unused resources

Static data center

Risks of Under Provisioning



What can you do with this?

Risk Transfers

Cost Associativity:

- 1K CPUs x 1 hour == 1 GPUs x 1K hours

Enabler for SaaS startups

- *Animoto* Facebook plugin => traffic doubled every 12 hours for 3 days
- Scaled from 50 to >3500 servers
- And scaled back down

Challenge and Opportunity

Challenges to adoption, growth, & business/policy models

Both technical and nontechnical

Most translate to 1 or more

Complete list in paper

Challenge: Cloud Programming

Challenge: exposing parallelism

- MapReduce relies on “embarrassing parallelism”

Programmers must (re)write problems to expose this parallelism, if it’s there to be found

Tools still primitive, though progressing rapidly

Challenge: Big Data

Challenge: long-haul networking is most expensive cloud resource and improving most slowly

Copy 8TB to Amazon over ~20Mbps network
$\Rightarrow$~35 days, ~$800 in transfer fee (2010)

How about shipping 8TB drive to Amazon instead?
$\Rightarrow$1 days, ~$150

What Is a Distribute System?

A set of cooperating computers that are communicating with each other over network to get some coherent task done

- multiple cooperating computers
- storage for big web sites, MapReduce,
- peer-to-peer sharing

Why Do People Build Distributed Systems

High performance

- Achieve parallelism

Fault Tolerance

Physical reason

Security

Infrastructure for Cloud

Three components

- Storage
- Communication
- Computation

The big goals:

Abstractions that hide the complexity of distribution.

Topics in DS

Performance

- The goal: scalable throughput

Fault tolerance

- 1000s of servers, big network -> always something broken We'd like to hide these failures from the application.

Consistency

- General-purpose infrastructure needs well-defined behavior.

Imagine you are operating a Starbucks

Case Study: Starbucks

Case Study: Starbucks

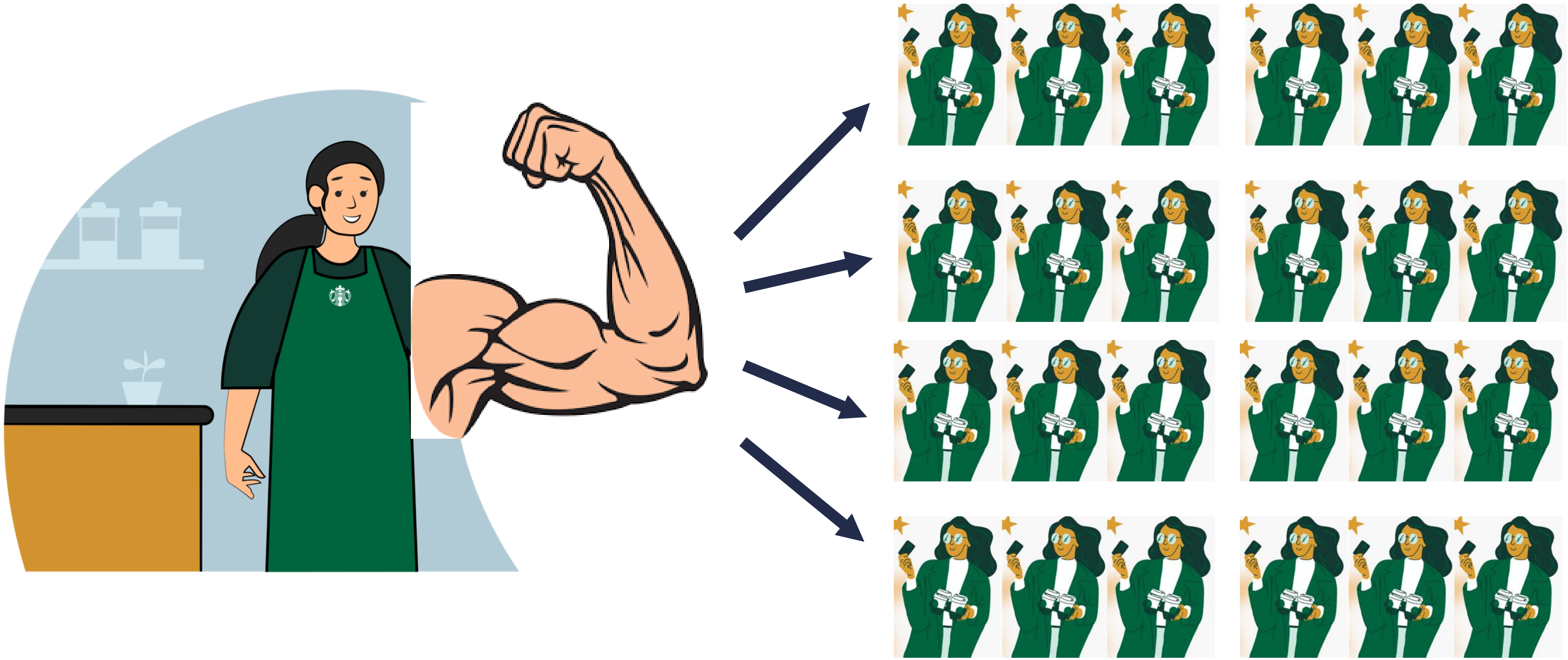
Now imagine 1000x customers?

Challenge 1: Ever-growing Load

Data is big. Users are many. Requests are even more.

Google get 8.5 billion searches per day.

Scale-up?

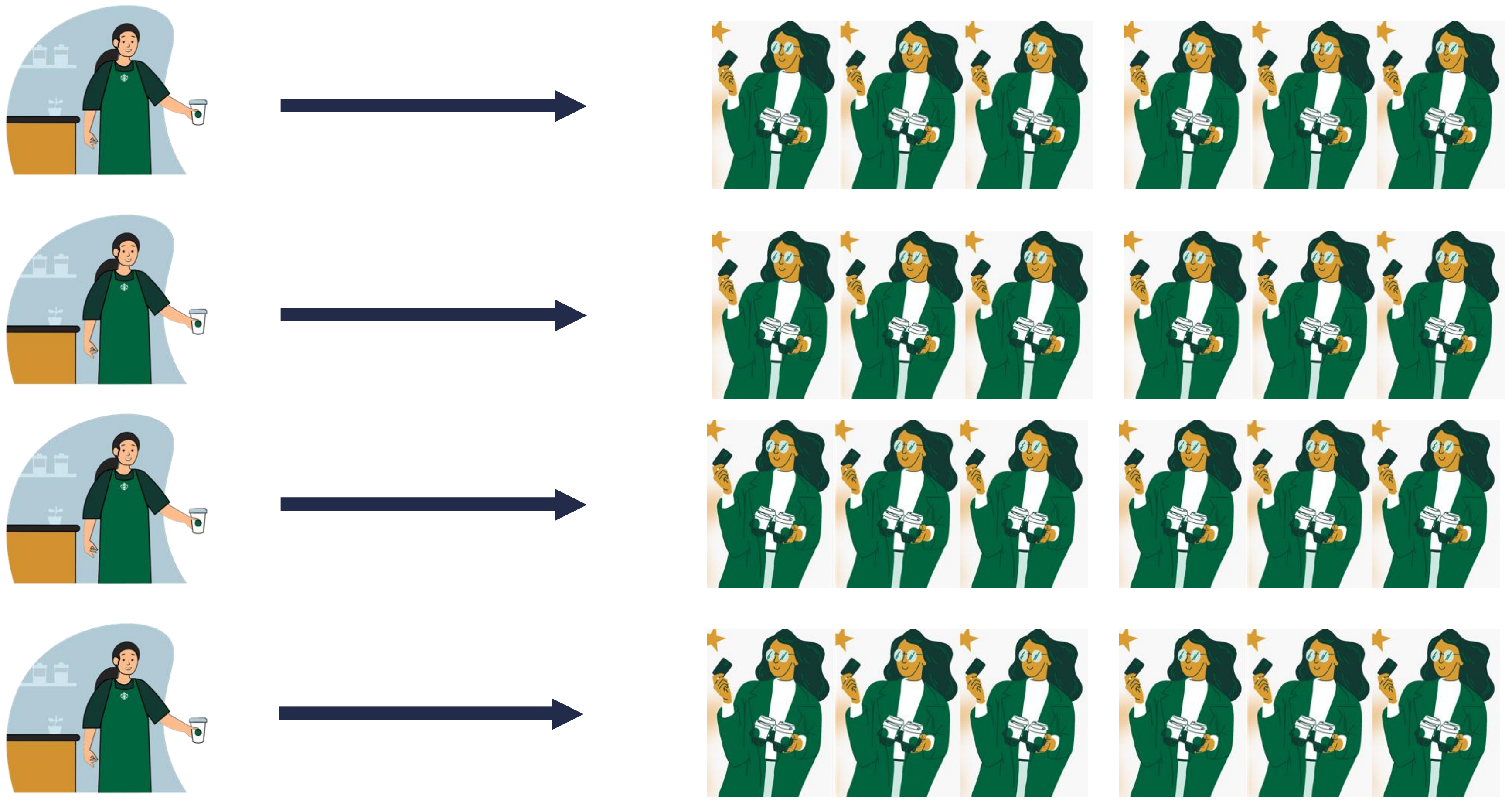


Scale-up?

- You can always add more compute resources—such as CPU, memory, and disk capacity.

- But no single machine can handle the ever-growing load

Approach 1: Scale-out (Sharing)!



Goal 1: Scalability

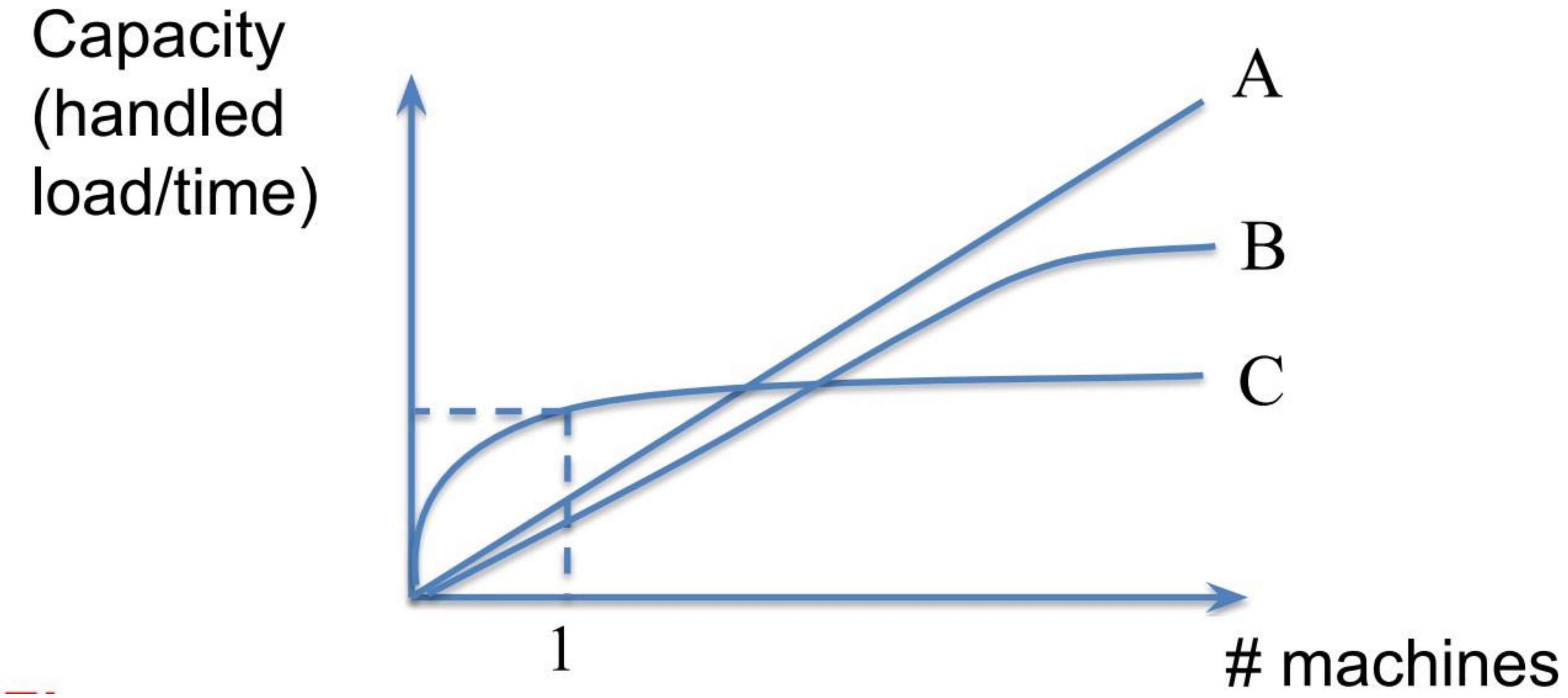



The more resource you add, the more requests you can serve.

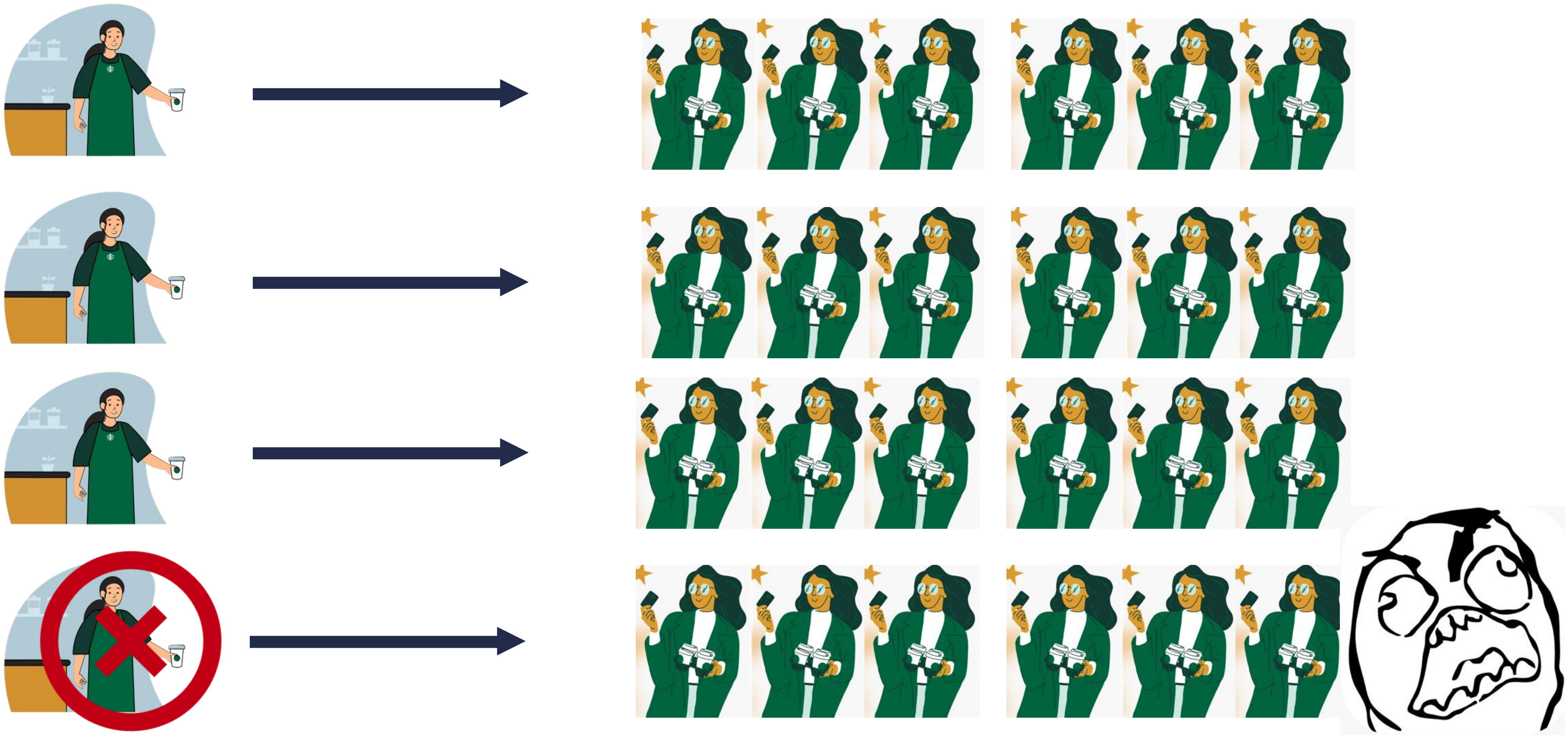
But never hope for perfect scalability

- add one machine, increase your capacity proportionally forever?

Sample Scalability Curves



Challenge 2: Failure



Goal 2: Fault Tolerance

Goal is to hide failures as much as possible to provide a service that e.g., finishes the computation fast despite failures, stores some data reliably despite failures, …

Fault tolerance subsumes:

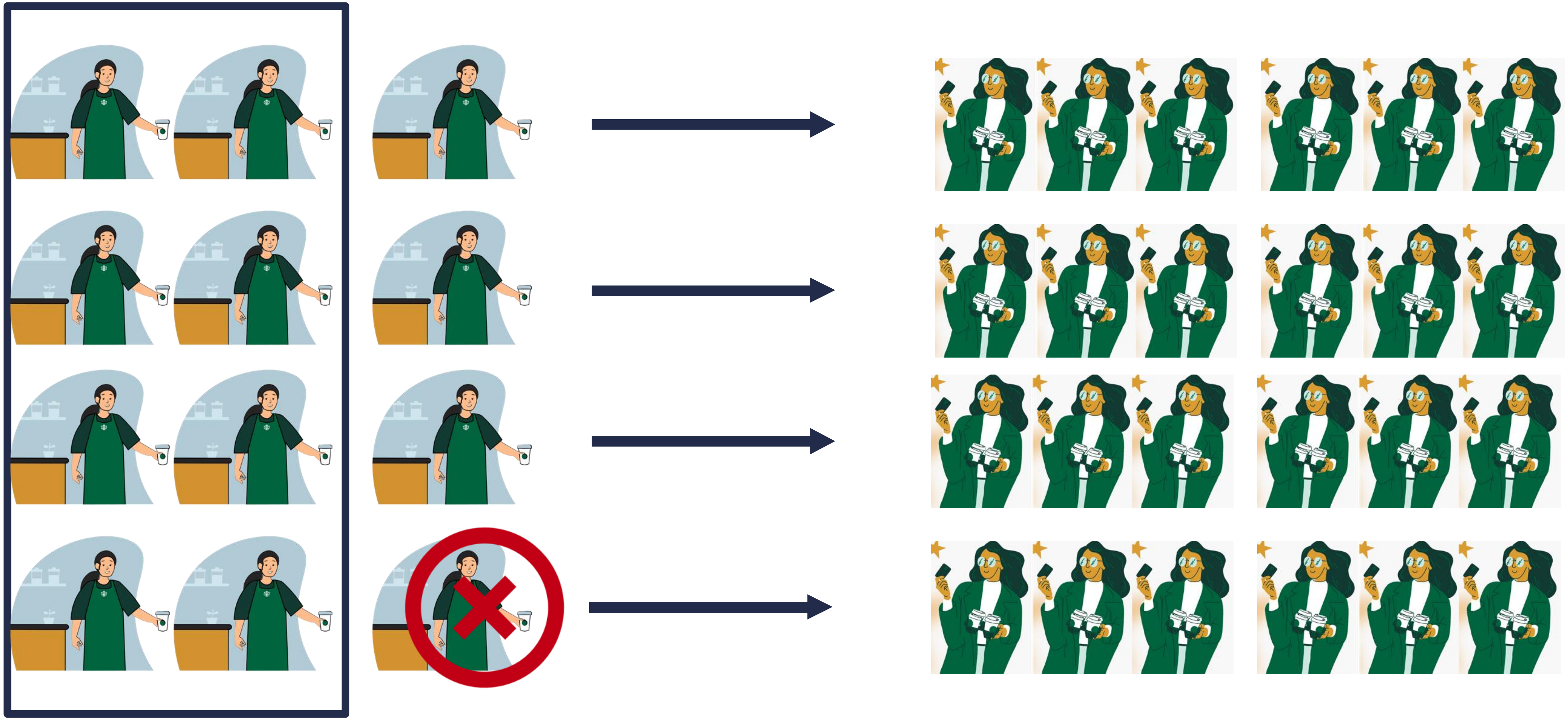
- Availability: the service/data continues to be operational despite failures.
- Durability: some data or updates that have been acknowledged by the system will persist despite failures and will eventually become available

Goal 2: Fault Tolerance

Availability: I expect someone will always take my order ..

Durability: staff remember my choice and wouldn't ask me again (even w/ failures)..

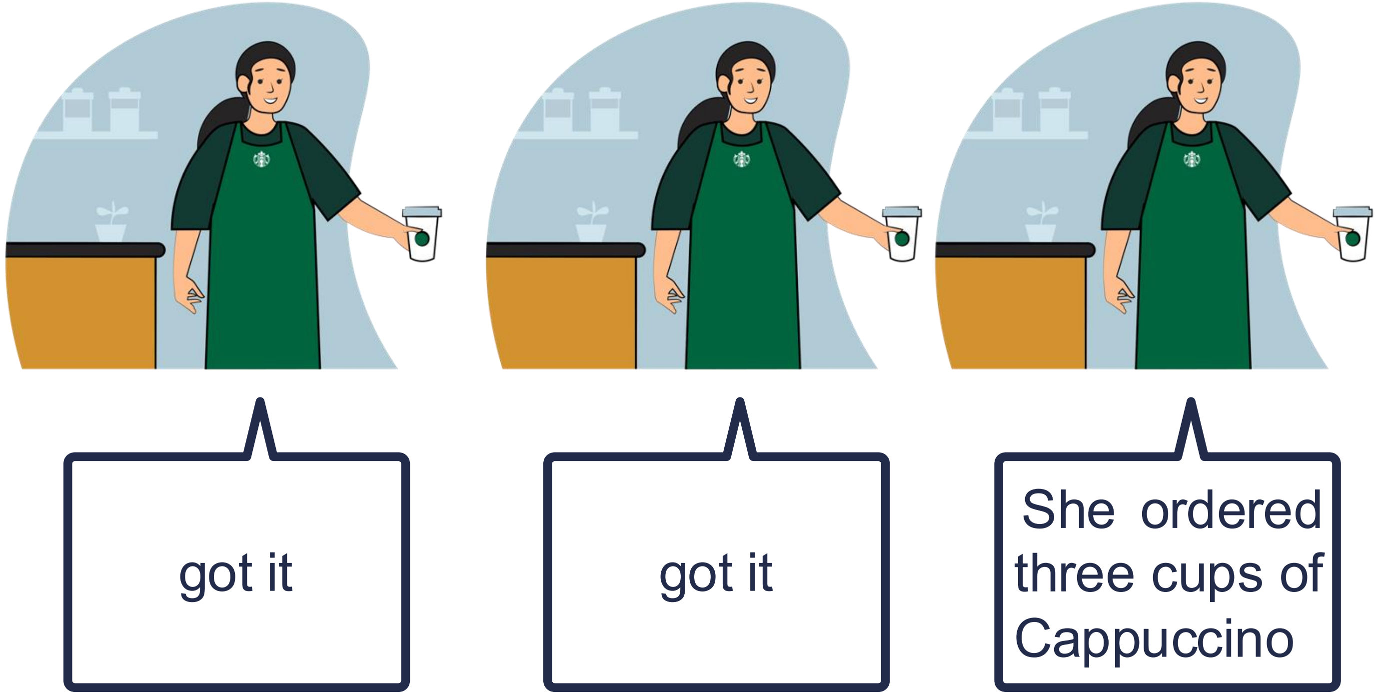
Approach 2: Replication



Challenge 3: Consistency

Three cups of Cappuccino please

Challenge 3: Consistency



Challenge 3: Consistency

(didn't hear)

got it

Change to Espresso

Wait, change to Espresso

Challenge 3: Consistency

Your Cappuccino is ready.

Your Espresso is ready.

???

Goal 3: Consistency Guarantee

Distributed systems try to create an illusion that users are using one single powerful machine

- They guarantee that every replica has the same view of data

Approach 3: Protocols

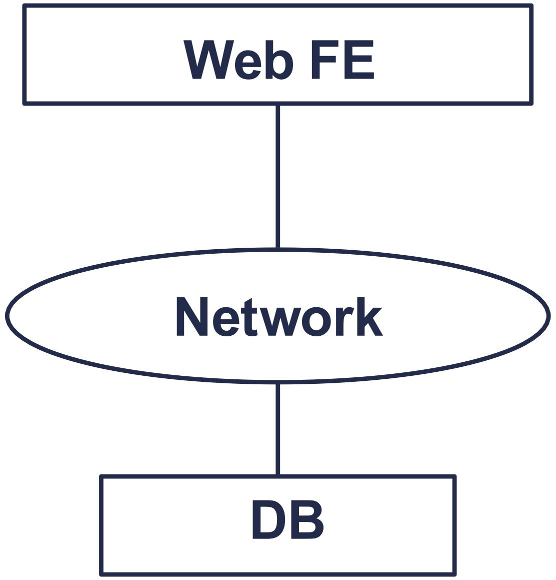
- The general approach is to develop rigorous protocols, which we will generically call agreement protocols, that allow workers and replicas to coordinate in a consistent way despite failures.

- Agreement protocols often rely on the notion of majorities: as long as a majority agrees on a value, the idea is that it can be safe to continue making that action.

- Different protocols exist for different consistency challenges, and often the protocols can be composed to address bigger, more realistic challenges

Example: Web Service Architecture

Basic Architecture

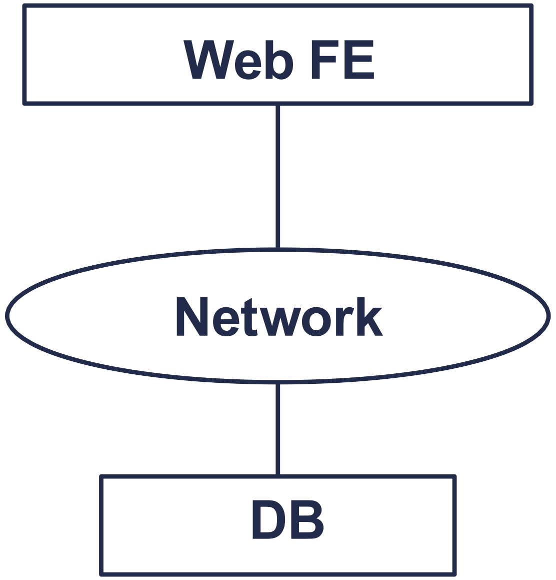


Web front end (FE), database server (DB), network. FE is stateless, all state in DB.

Properties

- Performance?
- Fault tolerance?
- Scalability?
- Semantics?

Basic Architecture



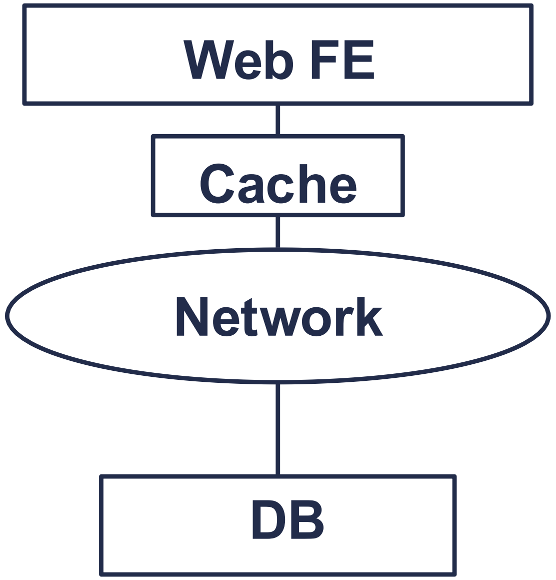
Web front end (FE), database server (DB), network. FE is stateless, all state in DB.

Properties

- Performance: poor
- Fault tolerance: poor
- Scalability: poor
- Semantics: great!

Let's improve performance first!

Goal: Reduce Latency



Performance

- Read?
- Write?

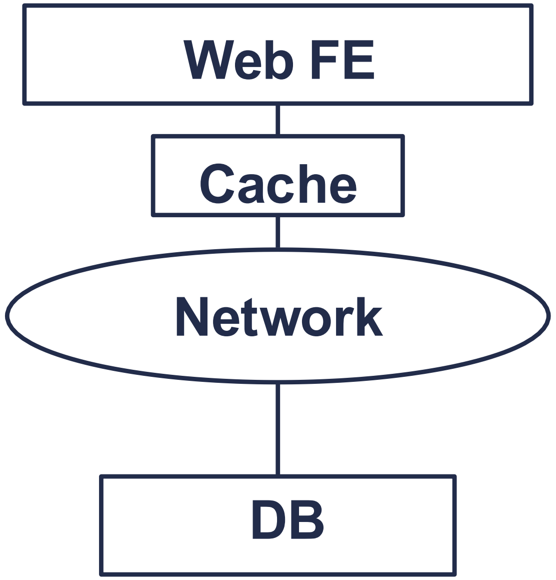
Fault tolerance

- Availability?
- Durability?

Scalability?

Semantics (consistency)?

Goal: Reduce Latency



Read latency: improved if working set fits in memory.

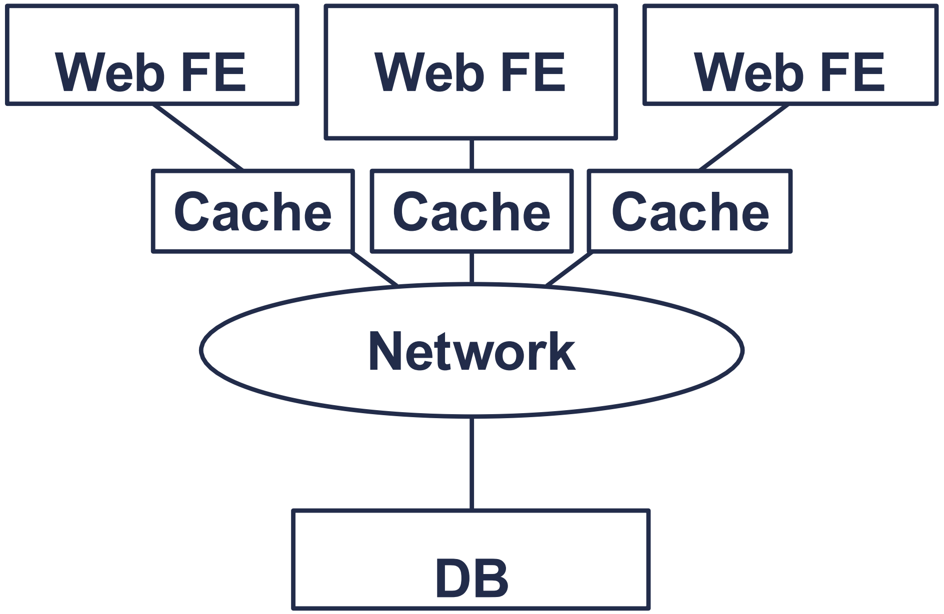
Durability: depends on cache: good for write-through $$, poor for write-back $$.

Write latency is opposite: good with writeback, poor with write-through.

Consistency: good: you have 1 FE accesses DB, going through 1 $$, so behavior is equivalent to single machine.

Let's deal with scalability, first on FE and later on DB.

Goal: Scale out the FE

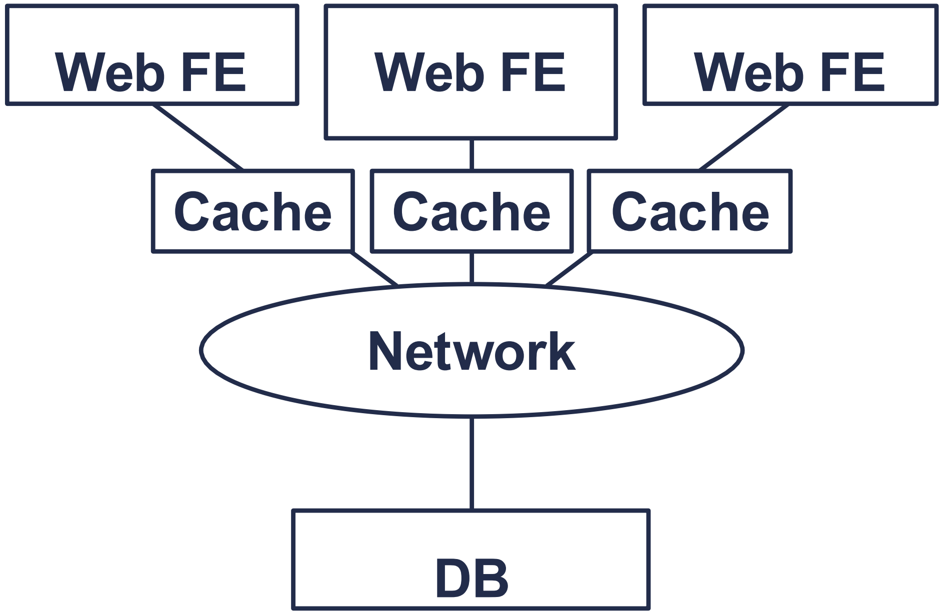


Launch multiple FEs. Each has its own local cache, which we'll assume is writethrough.

Properties:

- Performance?
- Fault tolerance?
- Scalability?
- Semantics?

Goal: Fault Tolerance For DB

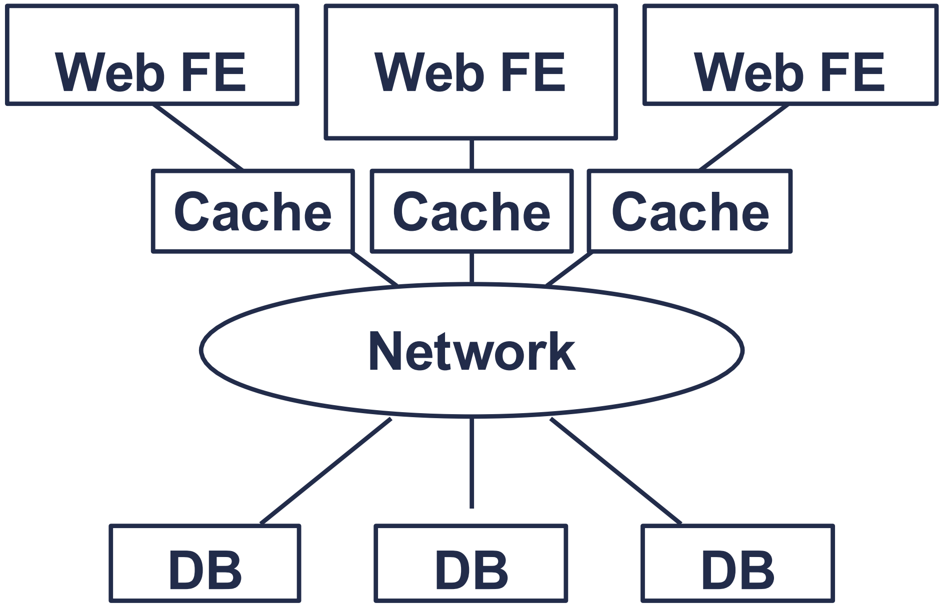


Launch **identical** replicas of the DB server, each with its disk. All replicas hold all data, writes go to all.

Properties:

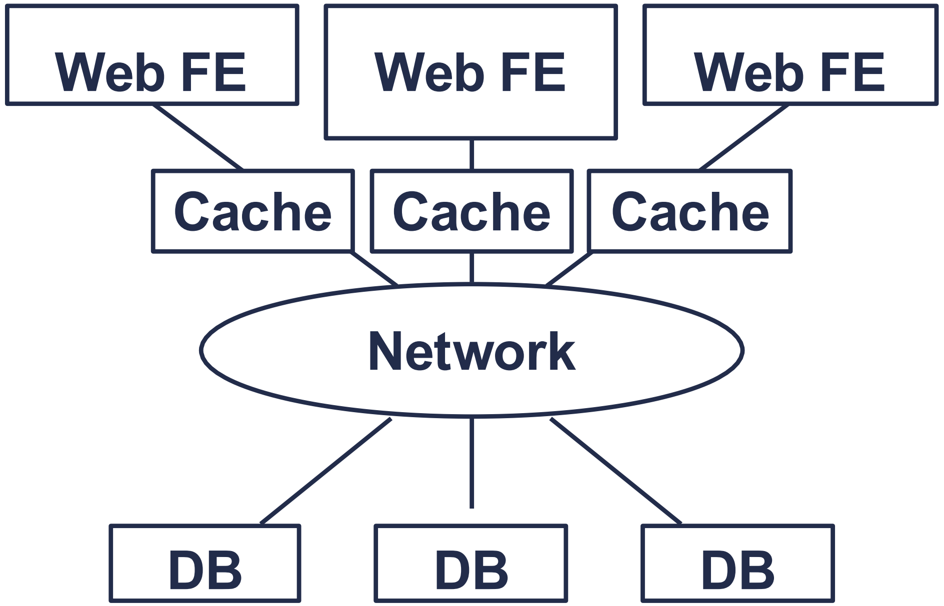
- Performance?
- Fault tolerance?
- Scalability?
- Semantics?

Goal: Fault Tolerance For DB



- Writes now need to propagate to all replicas. So they are much slower! Even if done in parallel, because FE now needs to wait for the slowest of DB replicas to commit (assuming write-through cache, which offers the best durability).
- All replicas must see all writes IN THE SAME ORDER! If order is exchanged, they can quickly go “out of sync”! So lots more consistency issues

Goal: Fault Tolerance For DB

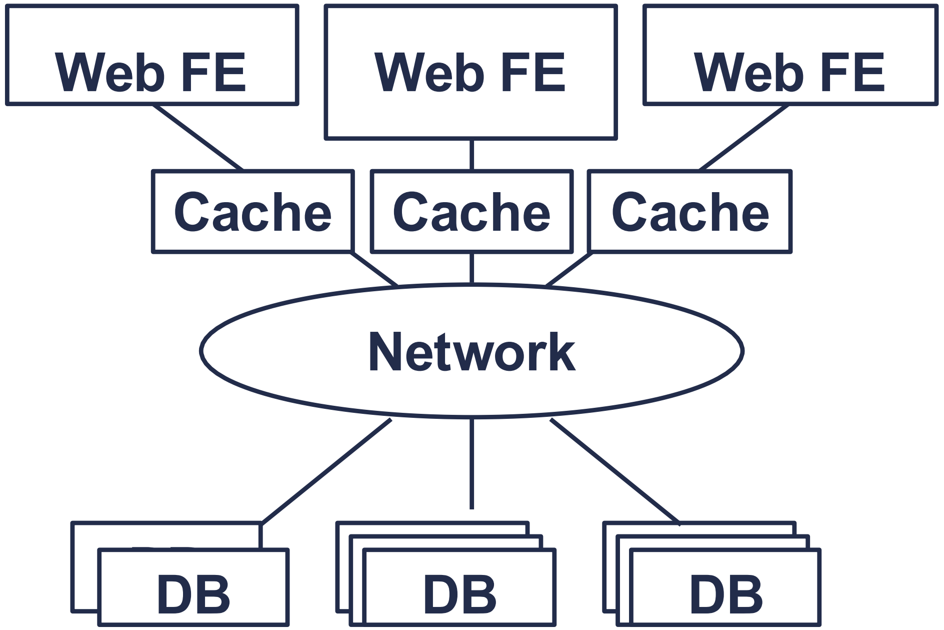


There are also availability issues. If you require all the replicas to be available when a write is satisfied (for durability), availability goes DOWN! Consensus protocols, which work on a majority of replicas, address this.

Another consistency challenge: how are reads handled? If you read from one replica, which one should you read given that updates to the item you're interested in might be in flight as you attempt to read it? We'll address these issues in future lectures by structuring the replica set

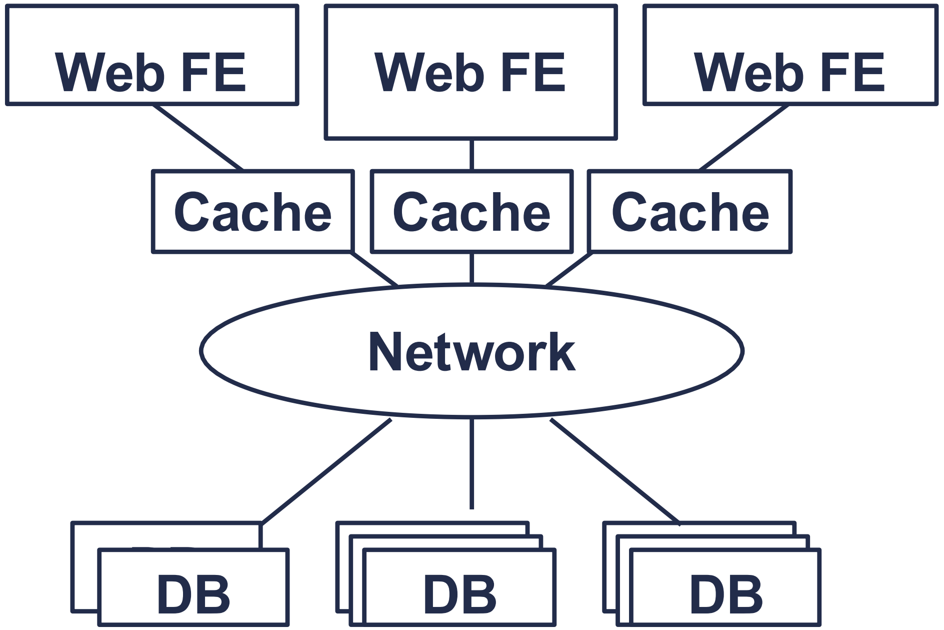
Let's deal with DB scalability.

Last Goal: Scale out The DB



Partition the database into multiple shards, replicate each multiple times for fault tolerance. Requests for different shards go to different replica groups.

Last Goal: Scale out The DB



New challenges that arise:

- How should data be sharded? Based on users, on some property of the data?

- How should different partitions get assigned to replica groups? How do clients know which servers serve/store which shards?

- If the FE wants to write/read multiple entries in the DB, how can it do that atomically if they span multiple shards? If different replica groups need to coordinate to implement atomic updates across shards, won't that hinder scalability?

Next Time

Read Google File System