Introduction

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

Distributed Systems
ECE419

Overview

- Course administration
- What is a distributed system?
- Examples of distributed systems
- Distributed systems infrastructure
- Why study distributed systems?
- Distributed systems goals, challenges and methods
- Example: distributed key-value cache

Course instructor

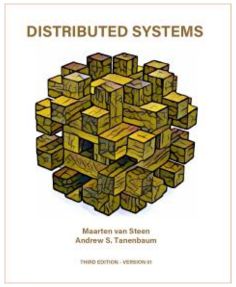
- Ashvin Goel
- Email: ashvin@eecg.toronto.edu
- Homepage: http://www.eecg.toronto.edu/~ashvin
- Office: Sandford Fleming 2001B
- Office hour: Thu 1-2 pm
- Research Interests: operating systems, cloud-scale systems

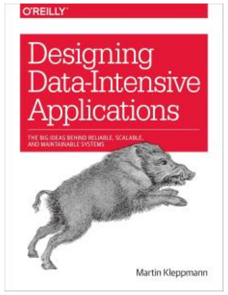
Course TAs

- Teaching assistants
    - Michail Bachras
    - Yunhao Mao
    - Shiquan Zhang
    - David Chu
    - Ding Guozhen
    - ChenXing Yang
    - Yuqin Yan
    - Victor Pineda Gonzalez
- Available during lab sessions or on Piazza

Recommended textbooks

- No required textbook
  - All relevant content will be covered in the lectures
  - We will post lecture slides and relevant resources
- Suggested resources and books
  - Distributed Systems Notes
    Tim Harris
  - Distributed Systems, 3rd Edition
    Maarten van Steen and Andrew S. Tanenbaum
  - Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems
    Martin Kleppmann




O'REILLY
Designing Data-Intensive Applications
THE BIG IDEAS BEHIND RELIABLE, SCALABLE, AND MAINTAINABLE SYSTEMS
Martin Kleppmann

Communication

- Quercus
  - For announcements, grades, course evaluations
- Piazza
  - Used for Q/A, discussion with peers, course staff
  - Please send message on Piazza before emailing course staff
- Course website
  - http://www.eecg.toronto.edu/~ashvin/teaching/ece419/current
  - Provides lecture slides, schedule, grading policy, lab info, etc.

Grading

- Exams (60%)
  - Mid-term (20% total), Mar 5, Wed 7-8:30pm (EX100)
  - Final (40%), date and time to be decided
- Lab assignments (40%)
  - 5 labs (varying % each - see course website)
- Grading policies available on course website

Labs

- All labs involve programming in Go language
- You need to work on all labs individually
  - Expect to spend significant time on lab assignments
- Lab sessions provide help
  - One or two TAs will be present
  - Attendance is not mandatory
- Lab submission
  - Electronic submission
  - Follow the submission procedure as specified in lab handouts
- Please don’t make lab code publicly available

Cheating

- Cheating is a serious offence, will be punished harshly
    - For first offense, 0 marks for assignment
- What is cheating?
    - Using someone else's solution to finish your assignment
    - Making your code available (even after course ends)
- What is NOT cheating?
    - Helping others use systems or tools
    - Helping others with high-level design issues
- We do use cheater-beaters
    - Automatically compares your solutions with others

How not to pass ECE419

- Do not come to lecture
  - It's nice outside, slides are online
  - Reality: It is much more efficient to learn through class discussion
- Do not ask questions during the lecture or piazza
  - It's scary, I don't want to embarrass myself
  - Reality: Asking questions is the best way to clarify lecture material
- Wait until the last couple of days to start a lab
  - Some of the lab assignments cannot be done in the last few days
- Copy other people's lab projects
  - That is cheating! How can you answer exam questions?

Term-work petitions

- Due to circumstances beyond your control, if you are unable to submit labs or do the mid-term exam
  - Please submit a term-work petition through the Engineering portal
  - We will provide accommodations
  - More details on course website

Before we start

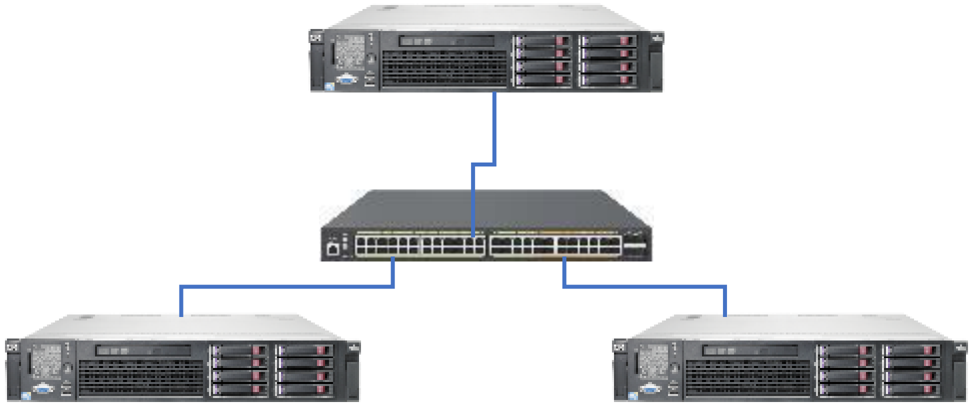
- Your background

- Any questions?

What is a Distributed System?

What is a distributed system?

- A set of computers (nodes) connected by a network
- Nodes do not share memory or clock
- Nodes work together as a single system
- Provide common set of services to users
- Enable scalable and reliable services



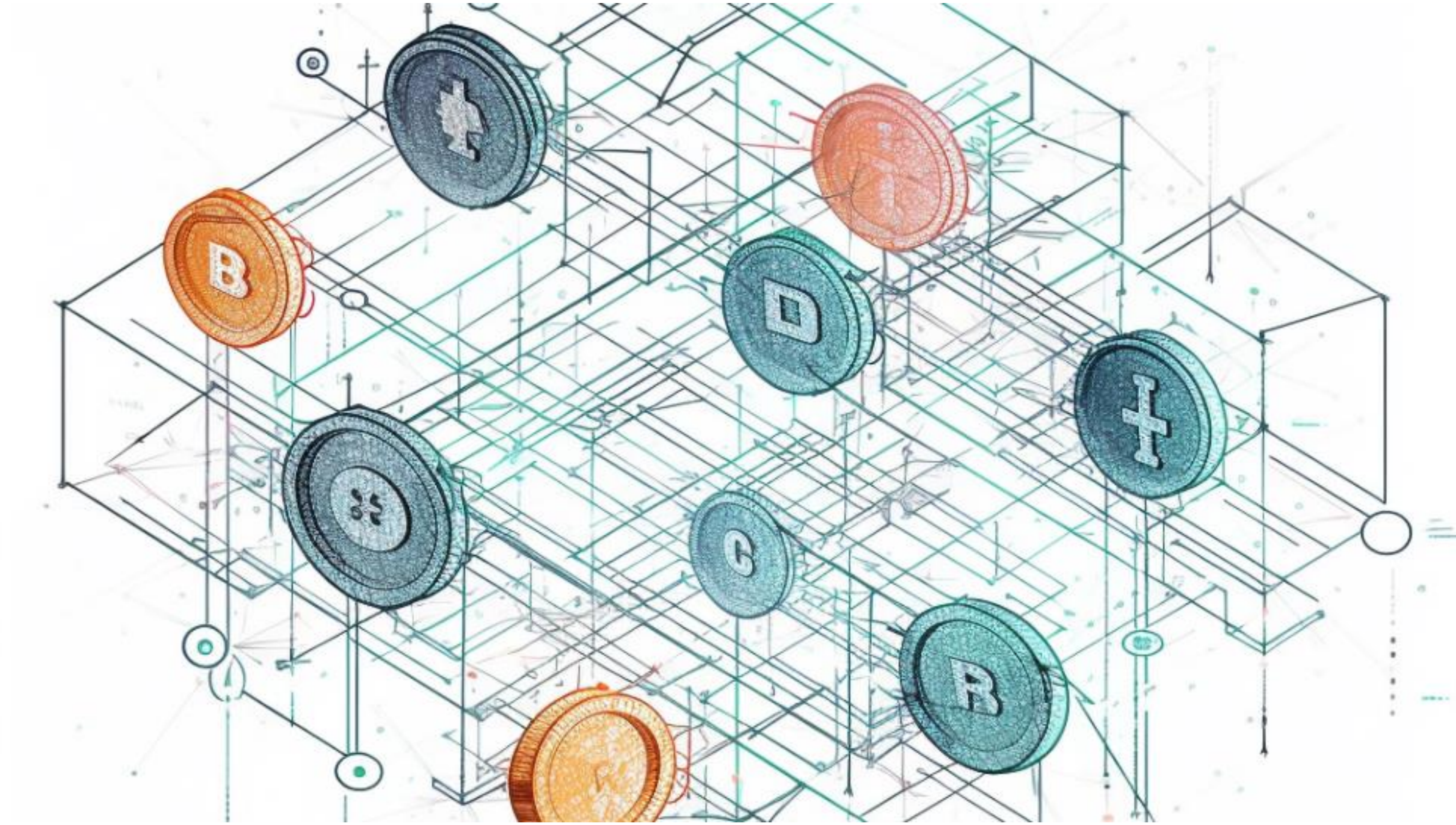
Why distributed systems?

- Or, why not one computer to rule them all?
- Limited computation, memory, storage, GPUs, …
- Single point of failures
- One physical location

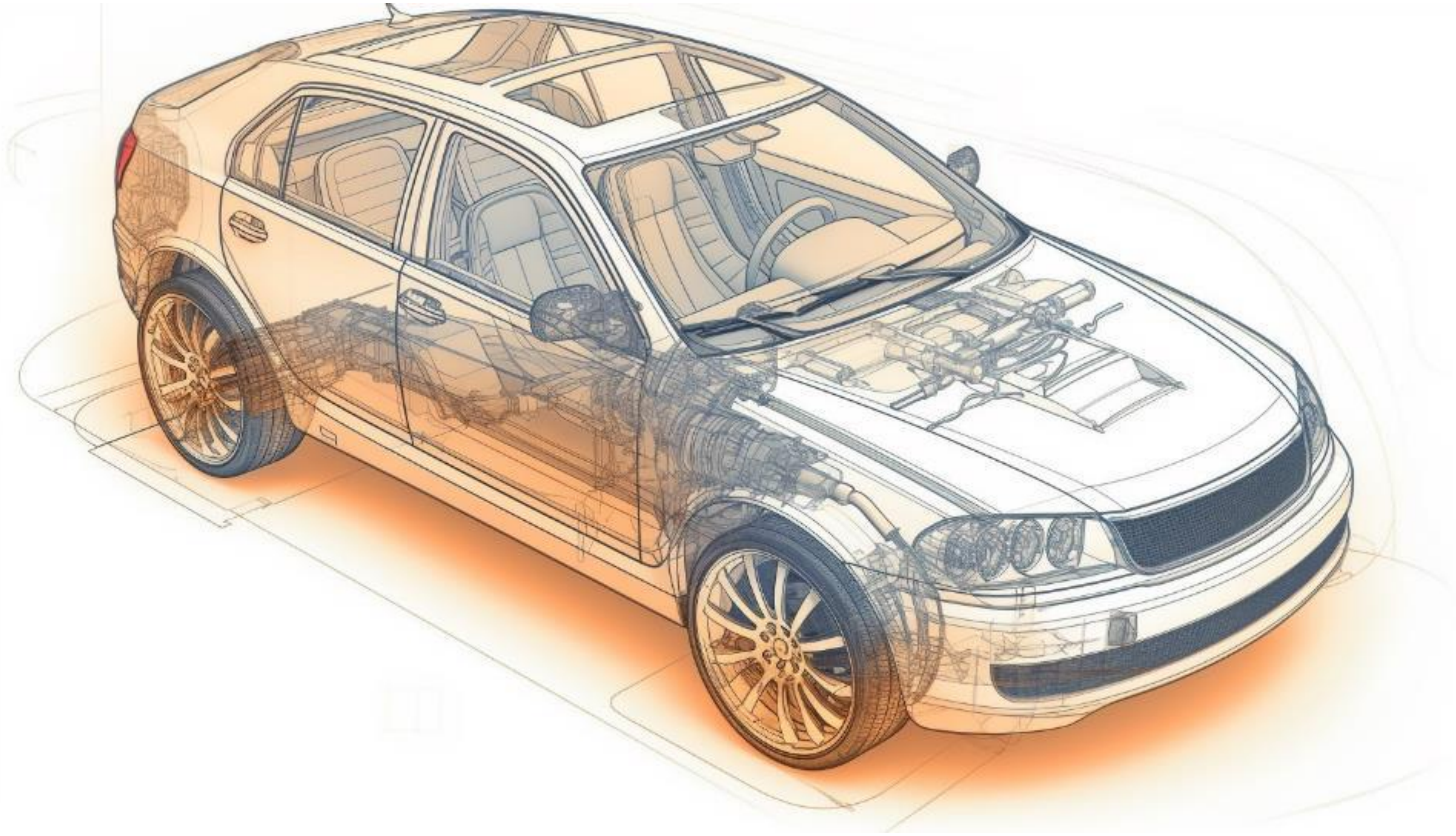
Examples of Distributed Systems

Email

Social Network

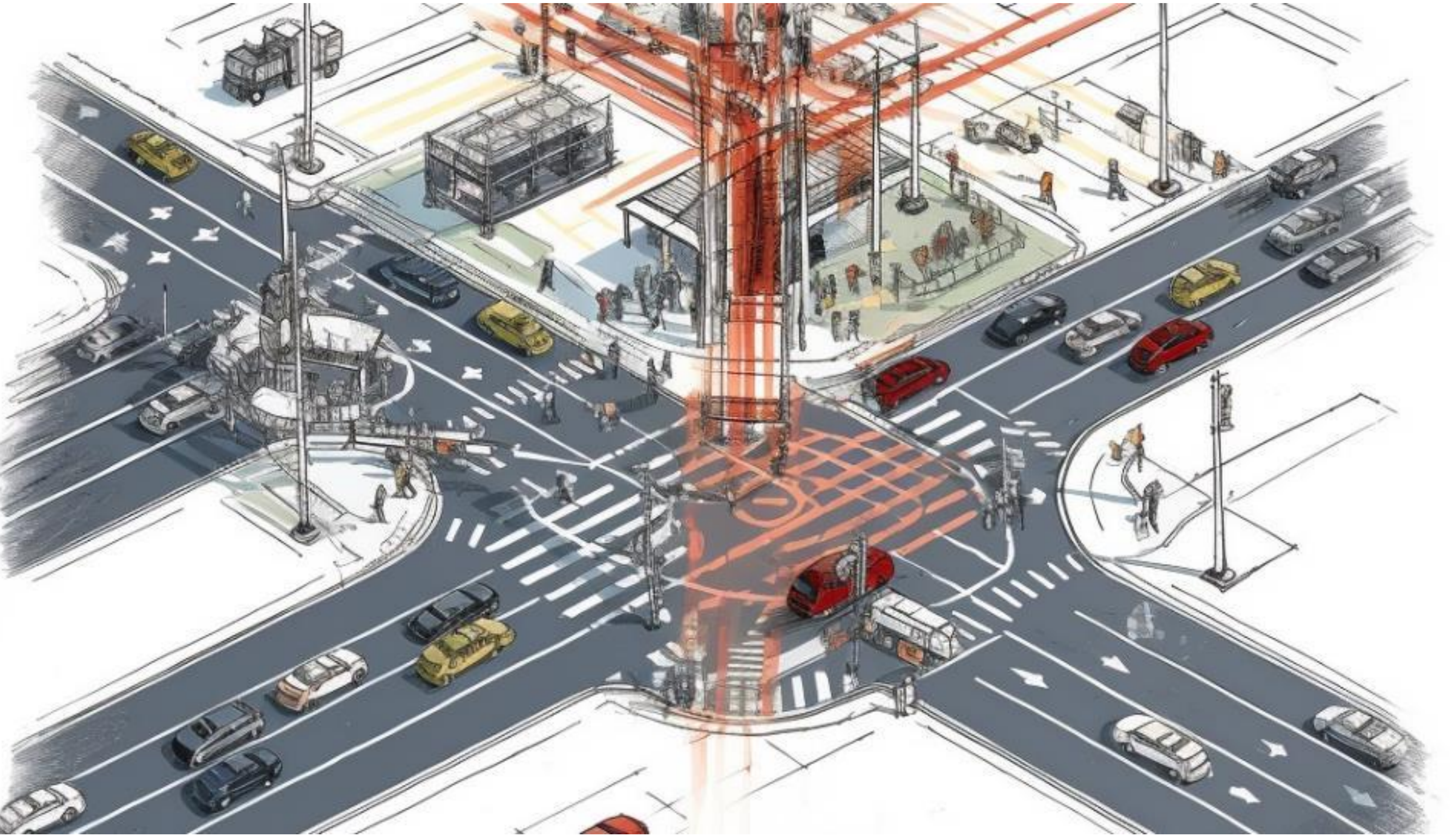


Blockchain

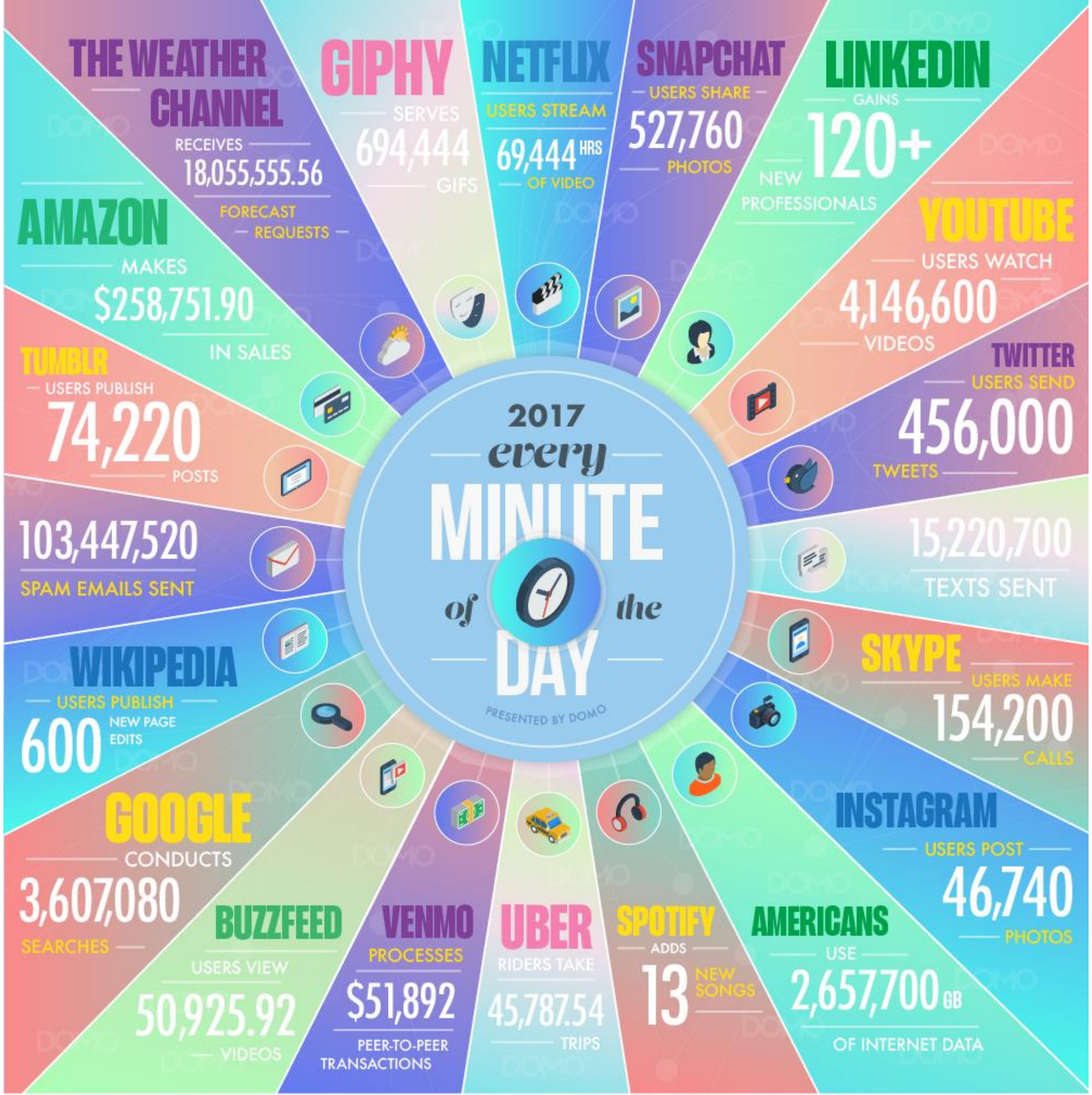


Cars

Airplanes and Air Traffic Control



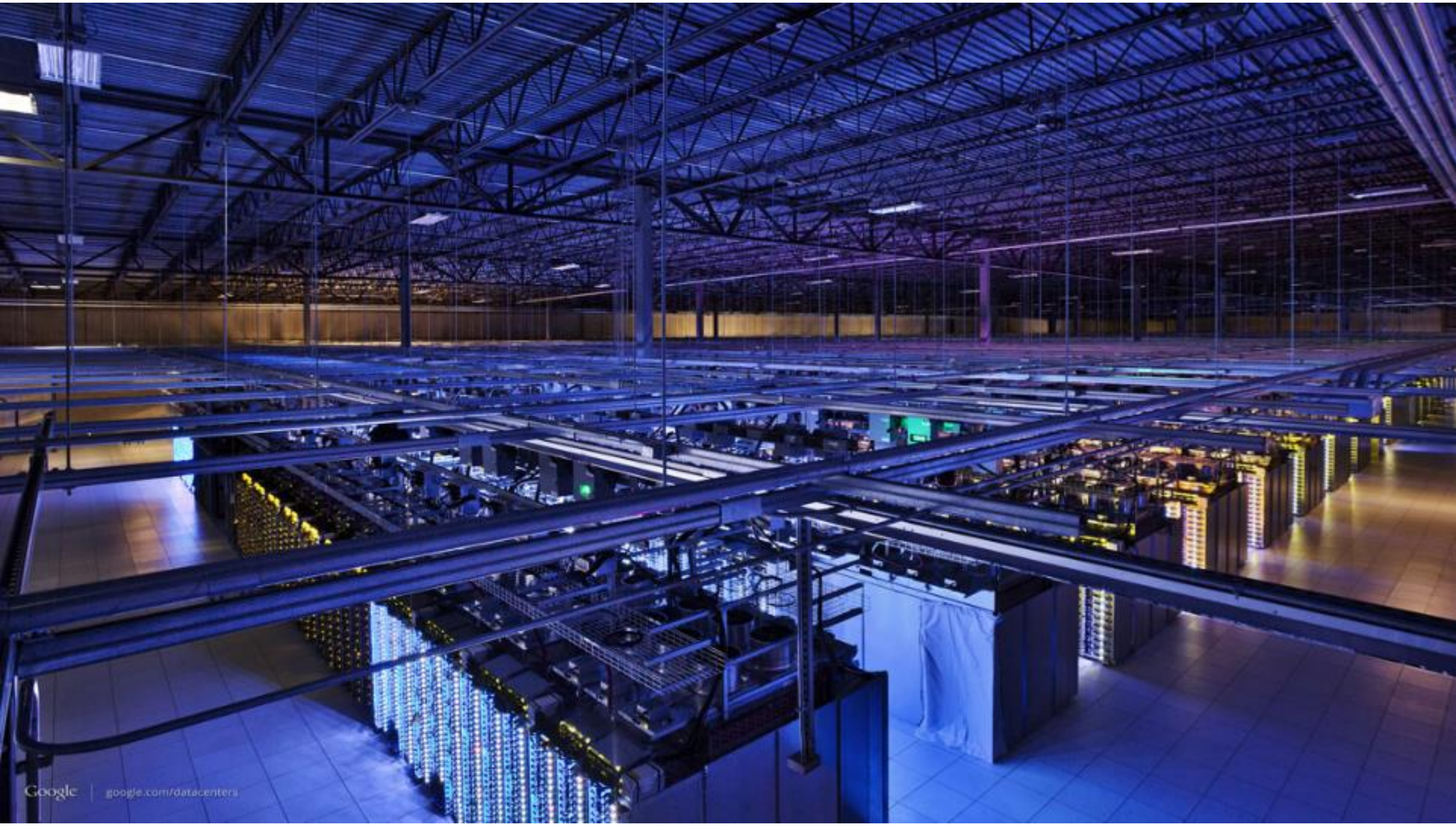
Traffic Light Controllers


2017 every MINUTE of the DAY
PRESENTED BY DOMO
THE WEATHER CHANNEL RECEIVES 18,055,555.56 FORECAST REQUESTS
GIPHY SERVES 694,444 GIFS
NETFLIX USERS STREAM 69,444 HRS OF VIDEO
SNAPCHAT USERS SHARE 527,760 PHOTOS
LINKEDIN GAINS 120+ NEW PROFESSIONALS
AMAZON MAKES $258,751.90 IN SALES
YOUTUBE USERS WATCH 4,146,600 VIDEOS
TUMBLR USERS PUBLISH 74,220 POSTS
TWITTER USERS SEND 456,000 TWEETS
103,447,520 SPAM EMAILS SENT
15,220,700 TEXTS SENT
WIKIPEDIA USERS PUBLISH 600 NEW PAGE EDITS
SKYPE USERS MAKE 154,200 CALLS
GOOGLE CONDUCTS 3,607,080 SEARCHES
INSTAGRAM USERS POST 46,740 PHOTOS
BUZZFEED USERS VIEW 50,925.92 VIDEOS
VENMO PROCESSES $51,892 PEER-TO-PEER TRANSACTIONS
UBER RIDERS TAKE 45,787.54 TRIPS
SPOTIFY ADDS 13 NEW SONGS
AMERICANS USE 2,657,700 GB OF INTERNET DATA

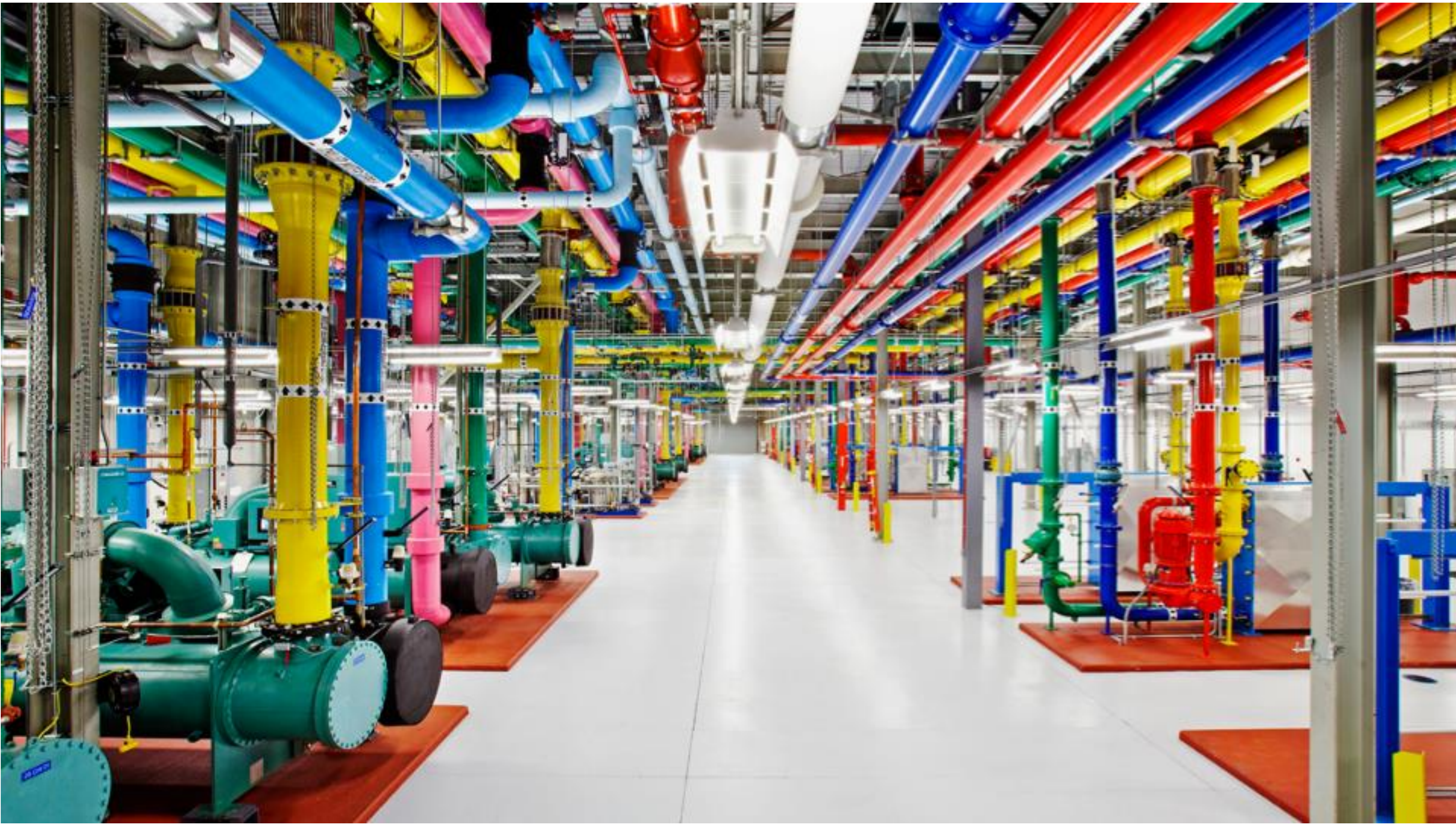
Distributed Systems Infrastructure

Google, 1997

Google, 2012



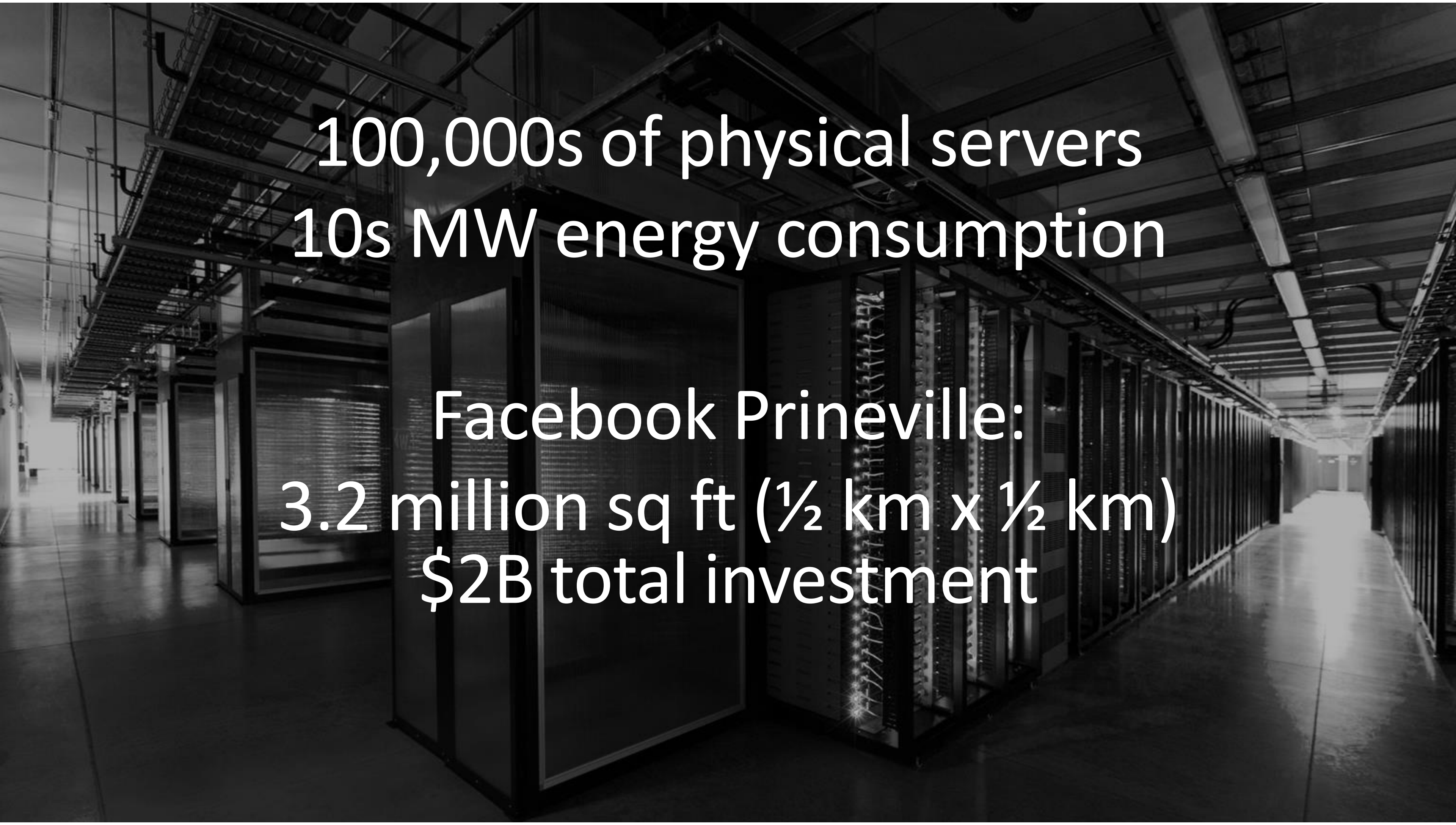
Google



Microsoft

Facebook

Facebook



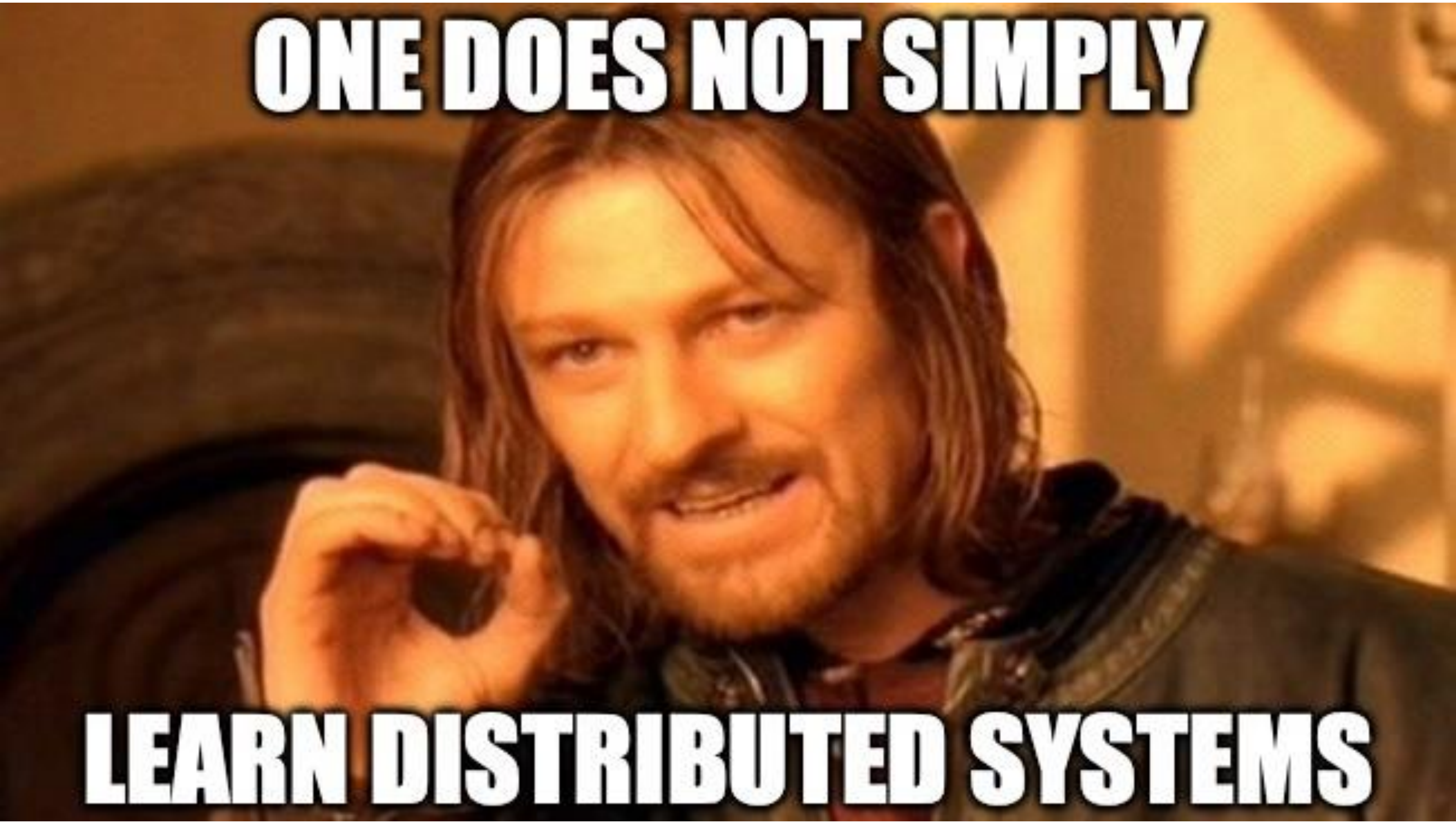
100,000s of physical servers
10s MW energy consumption

Facebook Prineville:
3.2 million sq ft (½ km x ½ km)
$2B total investment

Why Study Distributed Systems?

Distributed systems are challenging (and fun)

- Many “moving parts”
  - Execute software on many nodes concurrently
  - Unreliable communication over the network
  - No single global view of the system
  - Sometimes, no single entity is in charge of the system
- New kinds of problems
  - Partial failures
  - No global time
  - Correctness and consistency


ONE DOES NOT SIMPLY
LEARN DISTRIBUTED SYSTEMS

How to learn distributed systems?

- Distributed systems have strong theoretical underpinnings
- But you really learn by doing
  - Building systems
  - Debugging, modifying, dissecting existing systems
  - Testing for correctness
  - Evaluating performance, fault tolerance

What will you learn in this course?

- Foundational material in distributed systems
    - Principles, algorithms, architectures …
    - Focus on infrastructure for distributed applications
    - Essential for understanding modern computing
- Difficult and interesting problems
    - Active research area
- Several case studies of heavily used, real-world systems
    - Describes experience with practical deployments
- Lab experience with building distributed systems
    - Helps develop deep technical skills in building large-scale systems

Learning objectives

- Understand the design of distributed systems
  - Reason about concurrency, timing
  - Reason about failures
  - Reason about consistency
  - Reason about scalability
- Understand performance trade-offs
- Develop skills for building distributed systems

Course Topics

- Introduction
- Programming distributed systems
- Distributed storage systems
- Replicated storage systems
- Scalable storage systems
- Transactional storage systems
- Byzantine failures

Distributed Systems:
Goals, Challenges, Methods

Goals of a distributed system

- Abstraction: provide high-level interfaces for services
  - E.g., distributed file systems, key-value stores, databases
  - E.g., distributed computing frameworks
- Scalability: higher performance+capacity with more nodes
  - Users are not aware of the number of users of the system
- Fault tolerance: provide reliable service despite failures
  - Users are not aware when some nodes fail
- Consistency: behave like a single node, centralized system
  - Users are not aware of the difference
- Security: behave like a single, trusted system

Goals of a distributed system

- Abstraction: provide high-level interfaces for services
  - E.g., distributed file systems, key-value stores, databases
  - E.g., distributed compute engines
- Scalability: more performance+capacity with more nodes
  - Users are not aware of the number of users of the system
- Fault tolerance: provide reliable service despite failures
  - Users are not aware when some nodes fail
- Consistency: behave like a single node, centralized system
  - Users are not aware of the difference
- Security: behave like a single, trusted system

Applications focus on program logic,
System does the heavy lifting

Scalability

- Computation, data storage needs grow with more users
- Goal: higher performance and capacity with more nodes
    - Ideally, double the nodes ⇒ performance, storage capacity doubles
        - Weak scaling: double the job size, job takes same time to complete
        - Strong scaling: same job takes half the time to complete
- Challenges
    - How should data be partitioned across nodes?
        - Partition equally based on data size? data accesses?
    - How should a job be split and run in parallel across nodes?
        - When job is run in parallel, slowest node determines performance
        - Certain jobs cannot be easily parallelized, recall Amdahl's law
    - How to reduce network communication?

Scalability metrics and methods

- Metrics
  - Throughput: jobs per second, higher is better
  - Latency: time to complete a job, lower is better
    - Tail latency also matters, i.e., how long does it take in the worst case
  - Cost: resource usage, lower is better
- Methods
  - Place data accessed together on same node/rack, better locality
  - Place job's tasks on nodes that store their data, better locality
  - Co-locate tasks that communicate heavily, better locality
  - Spread data and tasks across nodes, better load balance

Fault tolerance

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable

Leslie Lamport
Turing Award Winner, 2003

Fault tolerance

- Thousands of nodes, so some failed nodes are the norm
- Goal: provide reliable service despite failures
  - Durability: service doesn’t lose data
  - Availability: service continues to make progress
- Challenges
  - Nodes may fail temporarily or forever at any time
  - Network links may delay messages for a long time or drop them
  - Hard to know whether node has failed, or network is down
  - Nodes may lose data or return corrupt data after a crash
  - Networks may corrupt messages

Fault tolerance metrics and methods

- Metrics
  - MTBF: mean time between failures, higher is more reliable
  - Availability: fraction of time service functions correctly
- Methods
  - Retry operations on failure
  - Store data on persistent storage for durability
  - Replicate data for high availability
  - Run redundant computations for high availability

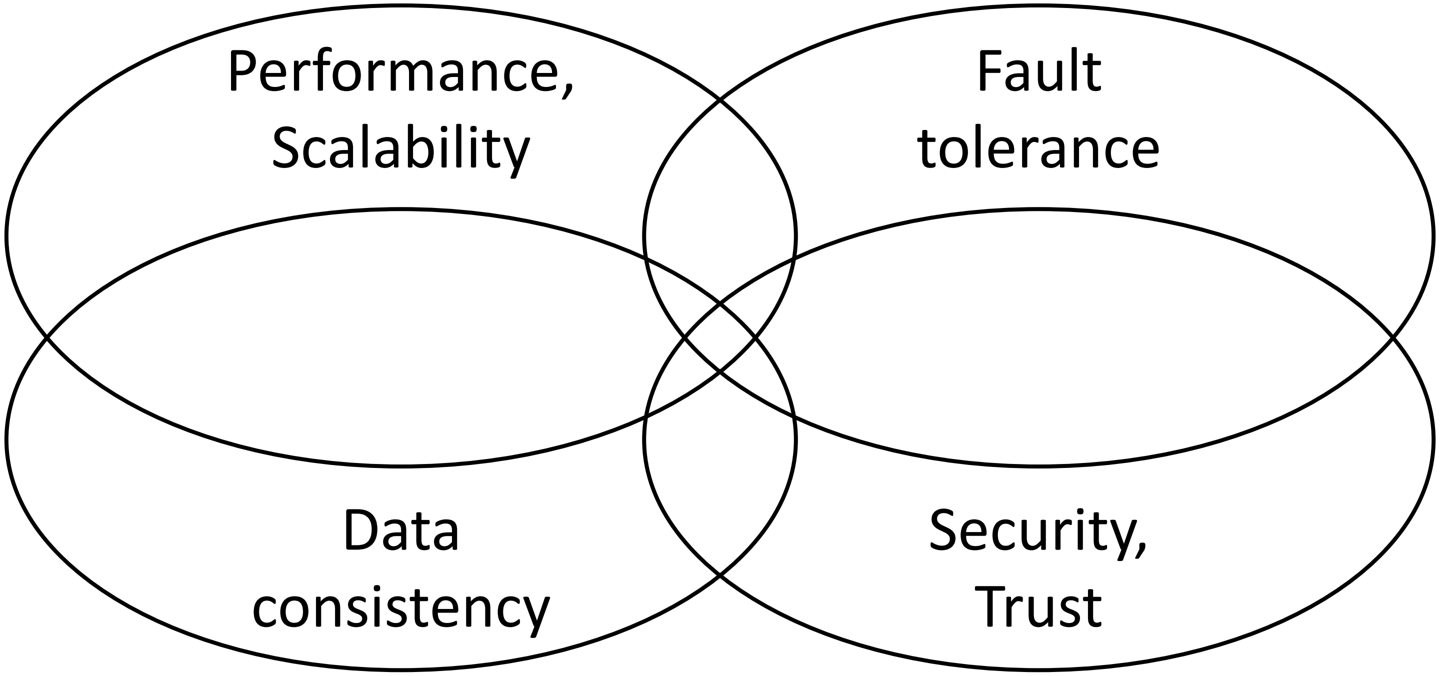
Data consistency

- Data is replicated (or cached) and accessed concurrently
    - Replicas may see reads and updates in different orders
- Goal: behave like a single node, centralized system
    - Reads return the value from the most recent write
- Challenges
    - How to order updates across replicas?
    - How to ensure updates occur exactly once?
    - How to ensure latest data is read?
    - How to ensure correctness under concurrent accesses?
    - How to ensure correctness under failures?

Security and trust

- Multiple nodes and network, so larger attack surface
- Goal: behave like a single, trusted system
- Challenges
  - Nodes may be compromised, behave maliciously
  - Nodes may alter, insert, drop messages
  - Nodes may send different messages to different nodes
  - …

Distributed systems make trade-offs



Typical trade-offs

- Performance over fault tolerance
  - Avoid replicating data, replicas increase overhead
- Performance over consistency
  - Avoid ordering updates
  - Read potentially stale data
- Fault tolerance over consistency
  - Update one replica synchronously, others asynchronously
  - Read from any replica
- Security over performance
  - Encrypt and authenticate data
  - Communicate with many replicas to build trust

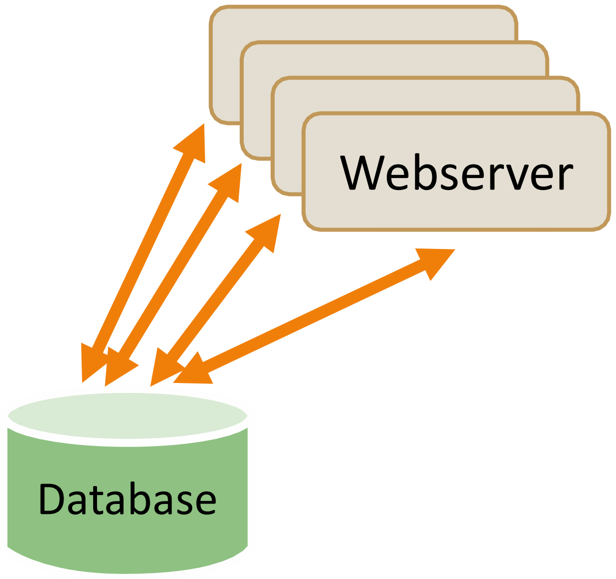
Case studies

- Scalability
  - Scalable coordination with Memcache
  - Optimistic replication with Dynamo
- Fault tolerance
  - State machine replication with RAFT
  - Coordination with ZooKeeper
- Security and fault tolerance
  - Bitcoin blockchain protocol
- We will discuss their different consistency guarantees

Example: Distributed Key-Value Cache

Why caching?

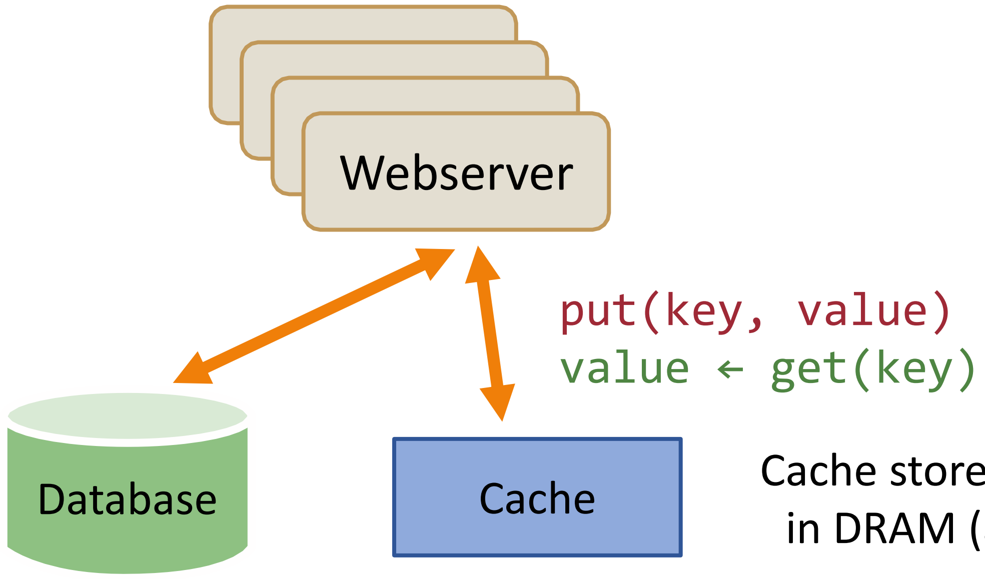
- Consider a web service with growing number of users
- Website adds more webservers for scaling service
- Database becomes bottleneck



Add a key-value cache service

- Let’s use a cache node to reduce load on the database
- Basic key-value cache abstraction is same as hash table

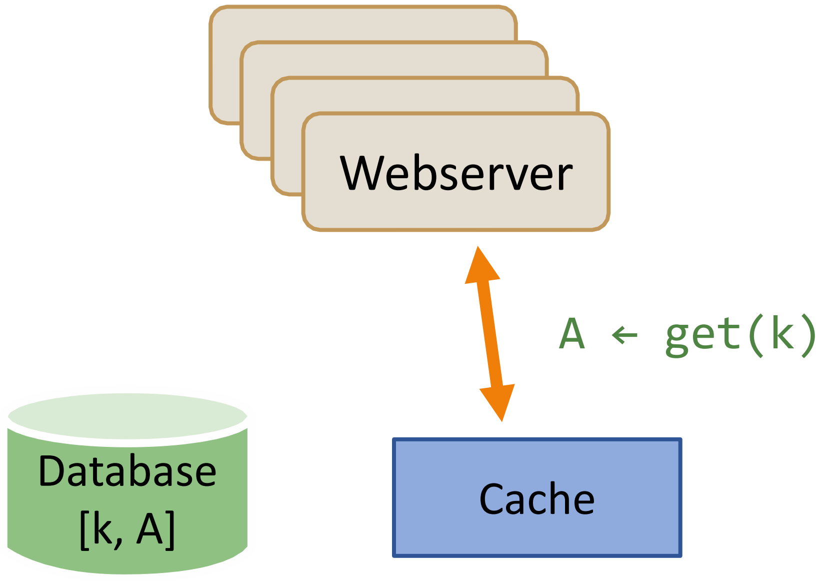
```
put(key, value)  // write (key, value)
value ← get(key) // read value associated with key
```



Cache stores [key, value] tuples in a hash table in DRAM (and optionally on storage as well)

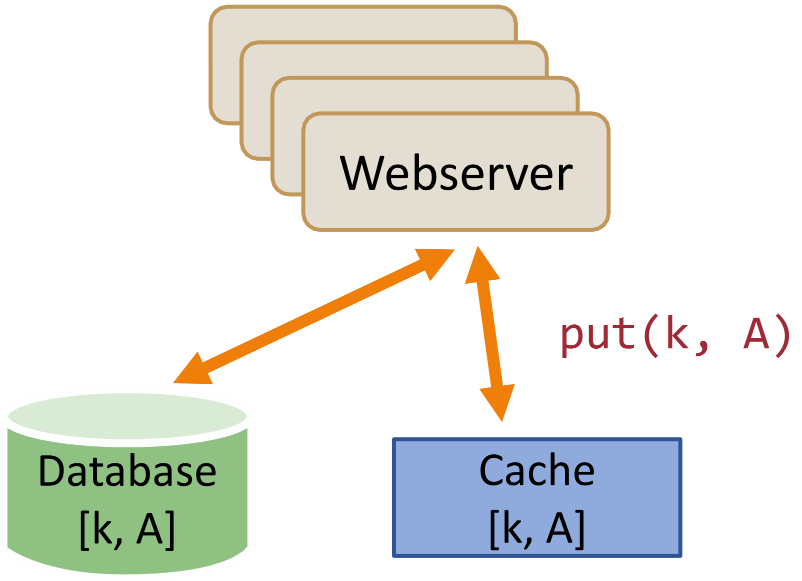
Cache operation

- Reads use get(k) to read data from cache



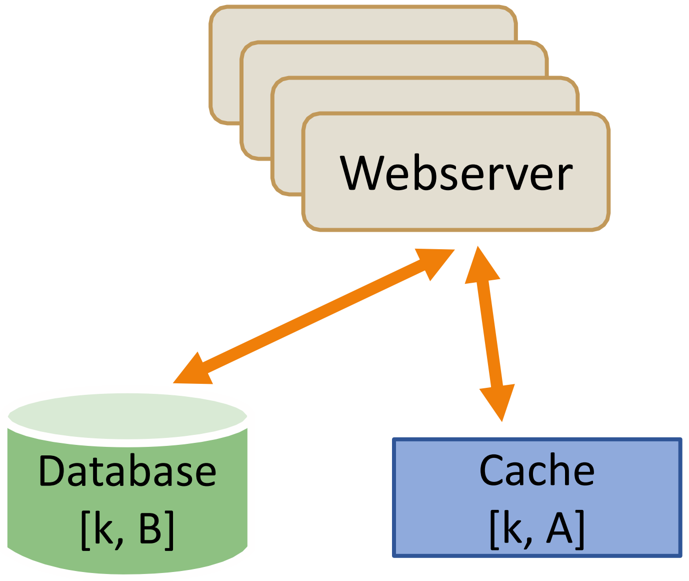
Cache operation

- Reads use get(k) to read data from cache
- If data is not cached, read from database, use put(k, value) to cache data
- Reduces load on database on future reads



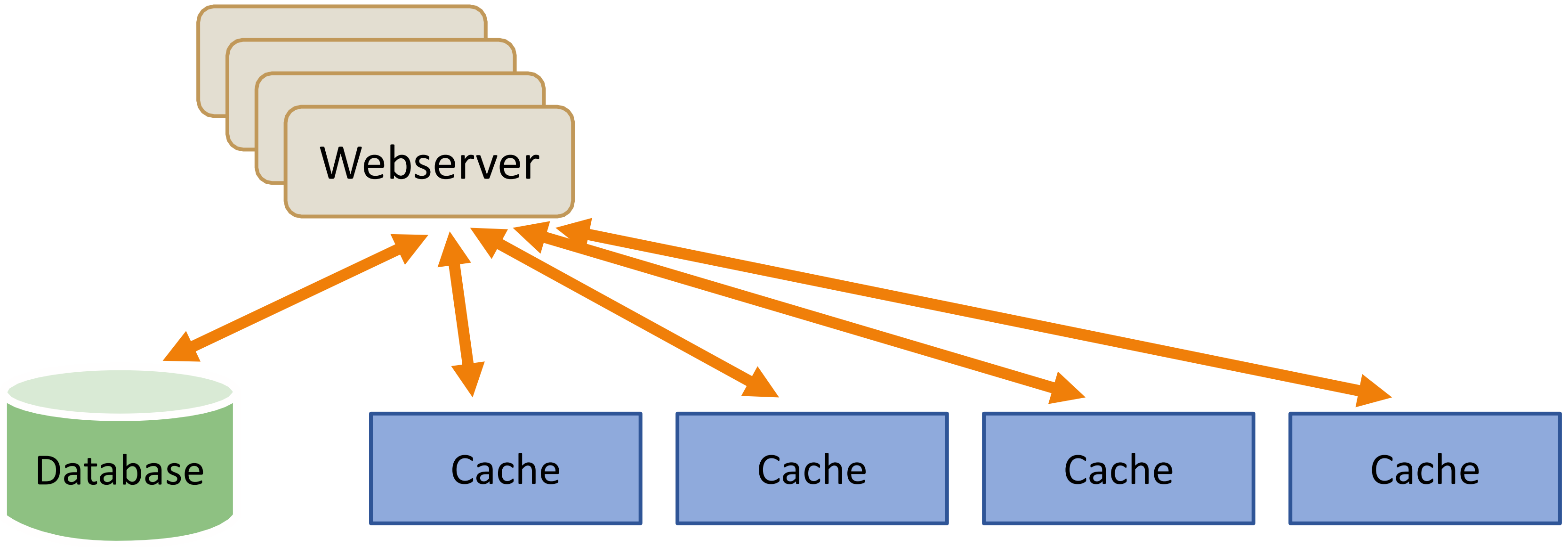
What about writes?

- Now we have two copies of data
  - Either invalidate cache or ensure both copies are consistent
  - Concurrency and failures cause complications
- We will look at this consistency problem later



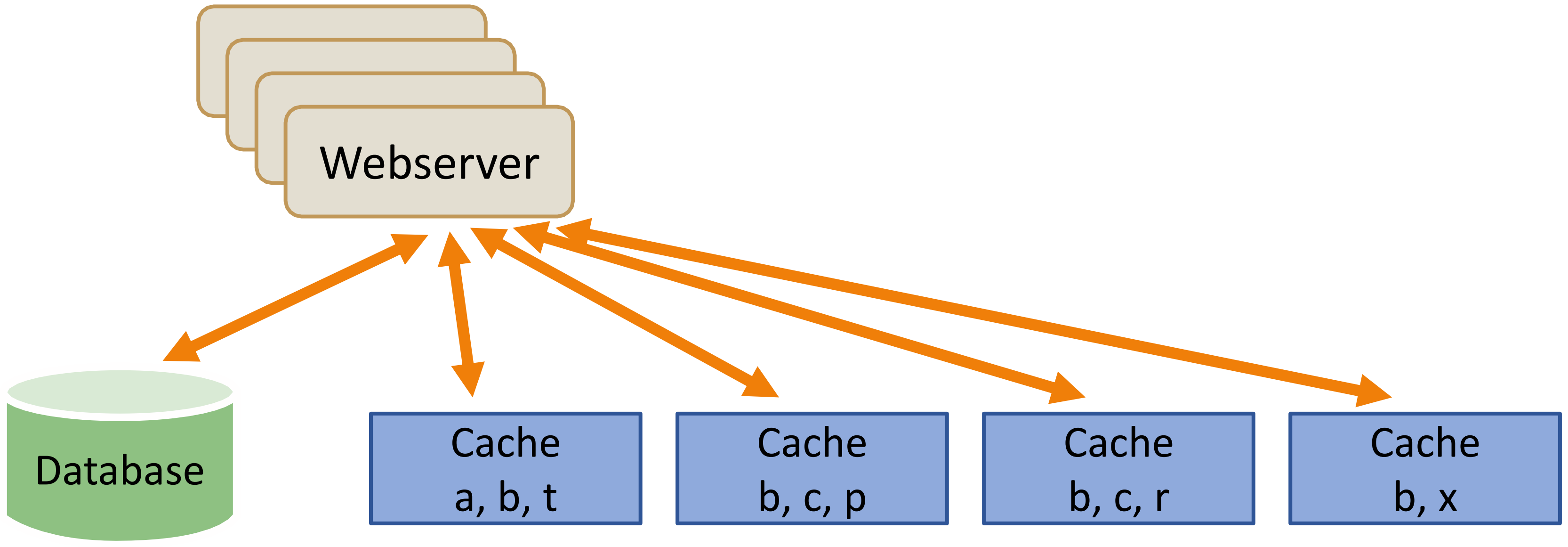
What if cache becomes the bottleneck?

- Add more cache nodes for scalability
- What data to store in each cache node?



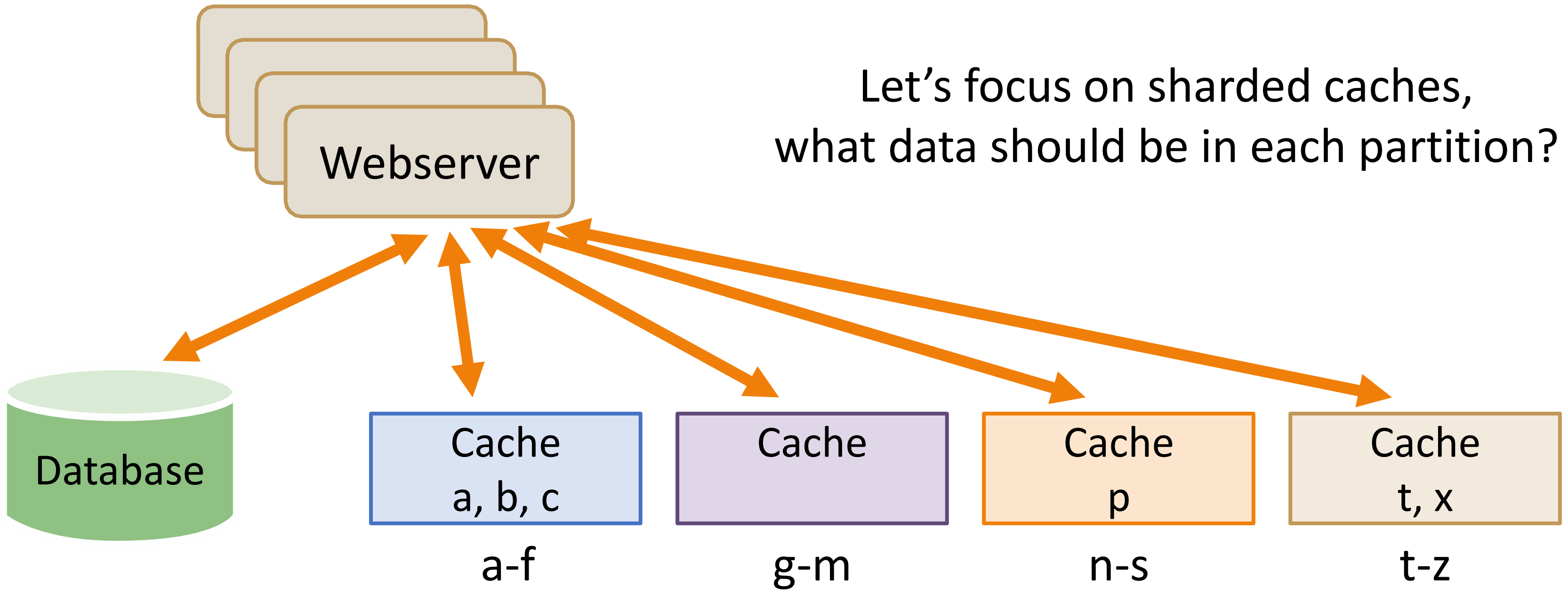
Option 1: replicated caches

- Each cache node can cache any data
  - Each webserver can access any cache node
  - Pros: hot data cached on multiple nodes, balances load on caches
  - Cons: data cached multiple times, caches are not used efficiently



Option 2: partitioned (sharded) caches

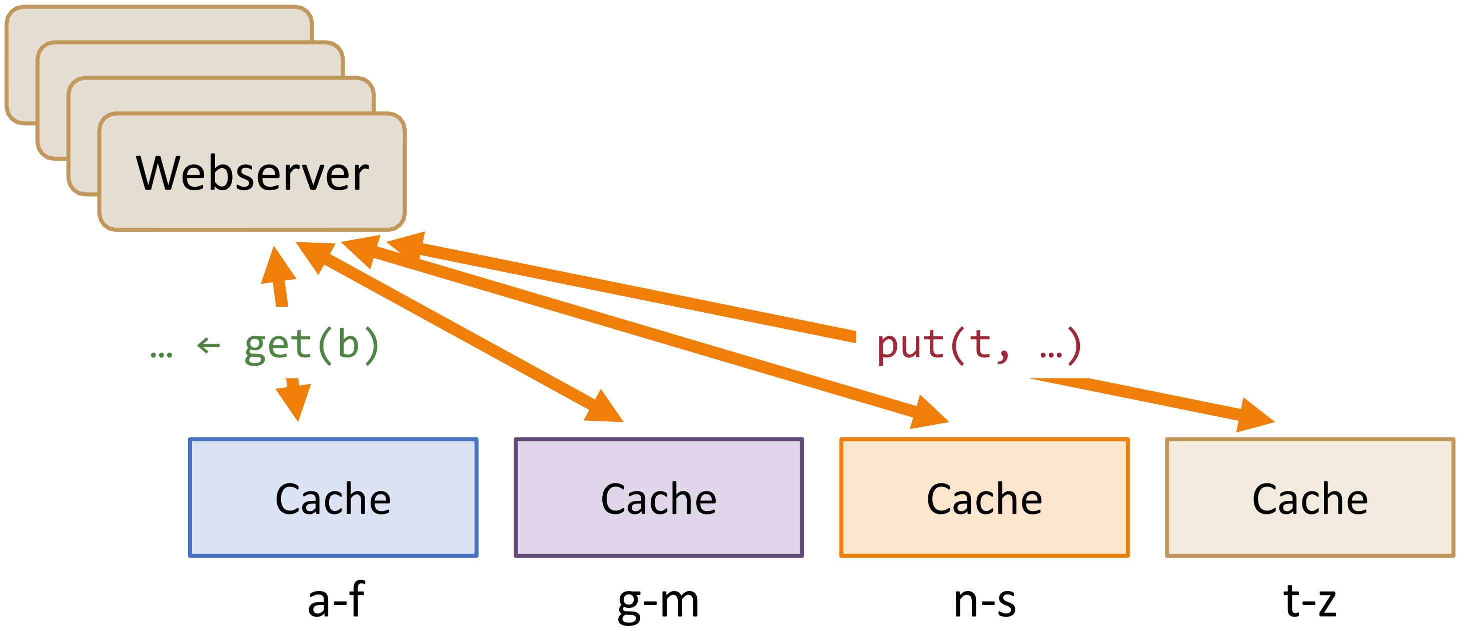
- Each cache node stores a partition of the data
  - Each webserver accesses cache node storing the partition
  - Pros: data cached once, caches are used efficiently
  - Cons: hot data cached on one node, node can become bottleneck



Let's focus on sharded caches,
what data should be in each partition?

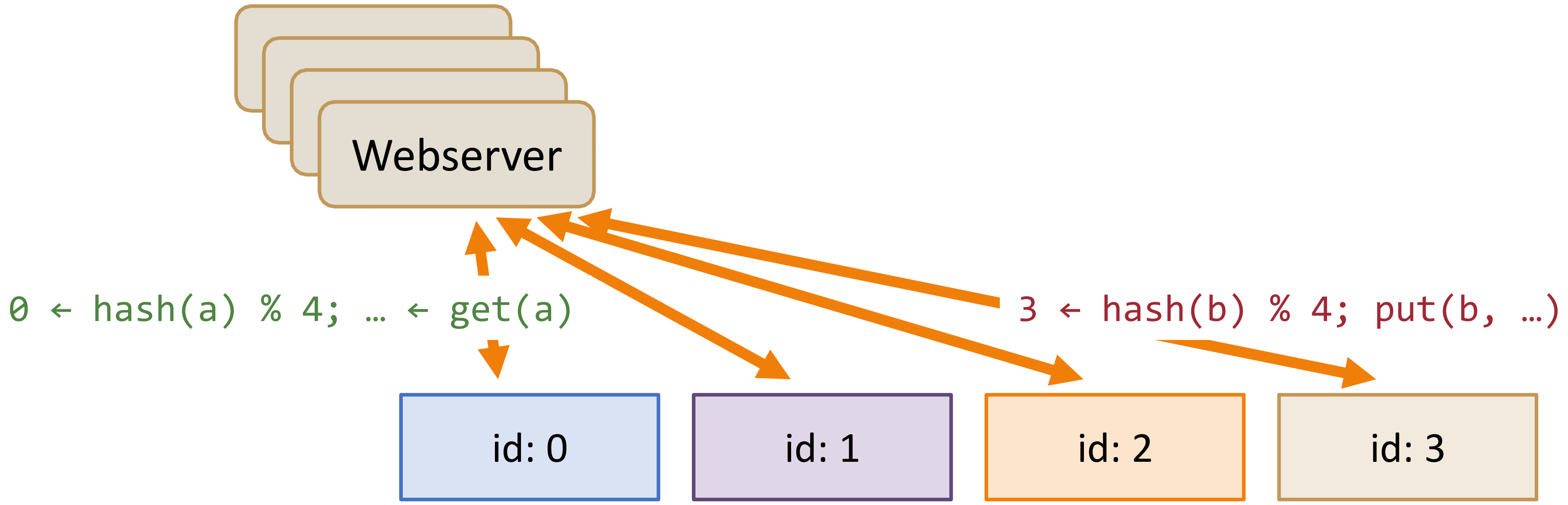
Range partitioning

- Each cache stores a contiguous range of keys
- How to decide partition range for each node?
  - Balance partition based on data size, number of accesses
  - Place data accessed together in same cache for locality



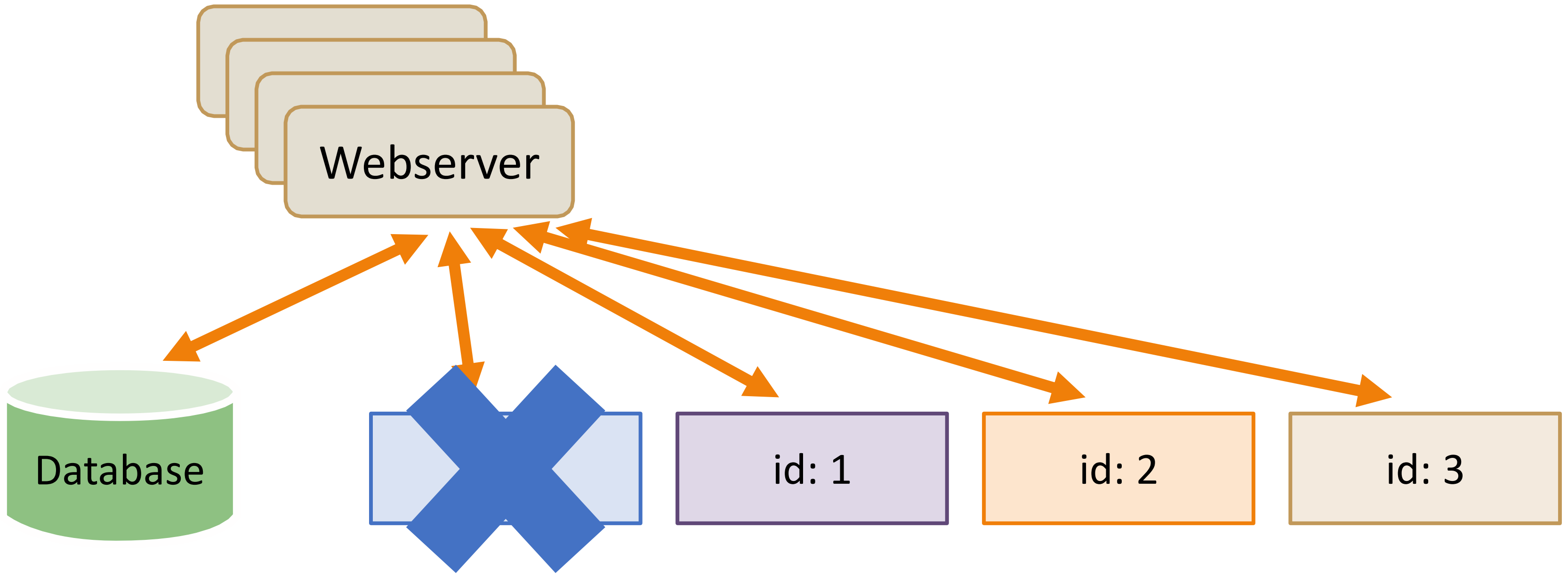
Hash partitioning

- Assume N cache nodes, key K stored on node hash(K) % N
- Simpler to implement than range partitioning
  - But no control over placement
  - Adding or removing cache nodes is expensive



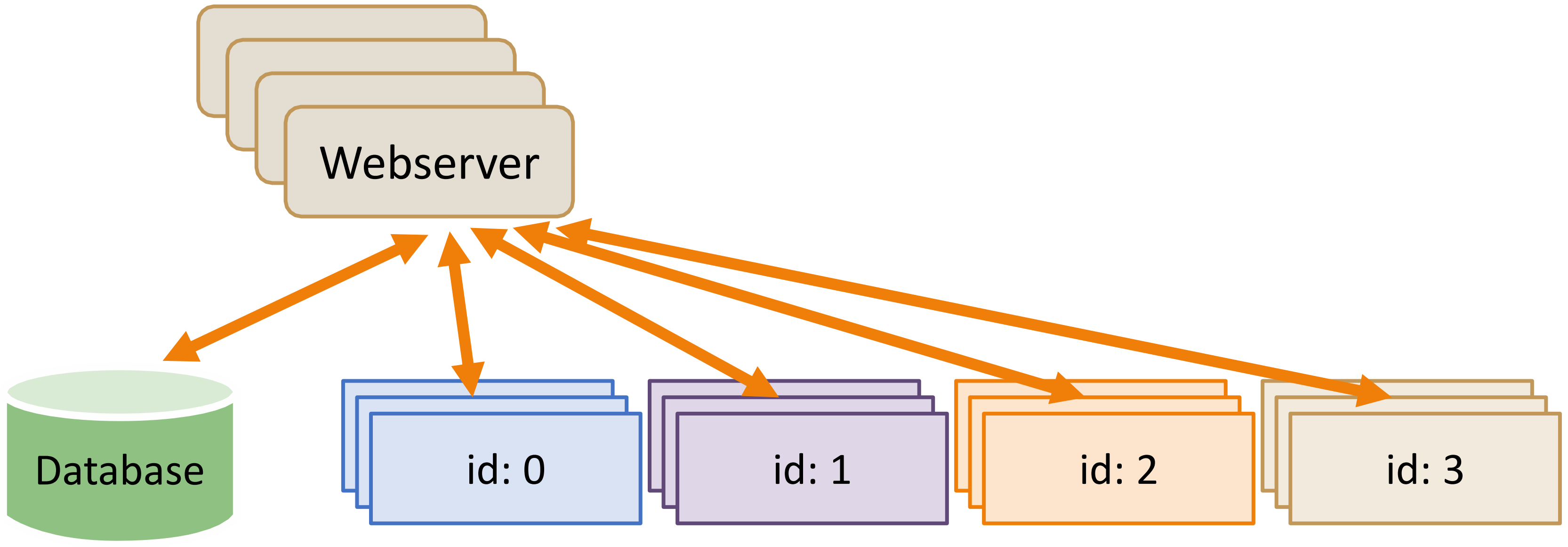
What about node failures?

- If a node fails, webservers can access database directly
- When node restarts, it can start caching data again
- But a cold cache puts too much load on the database



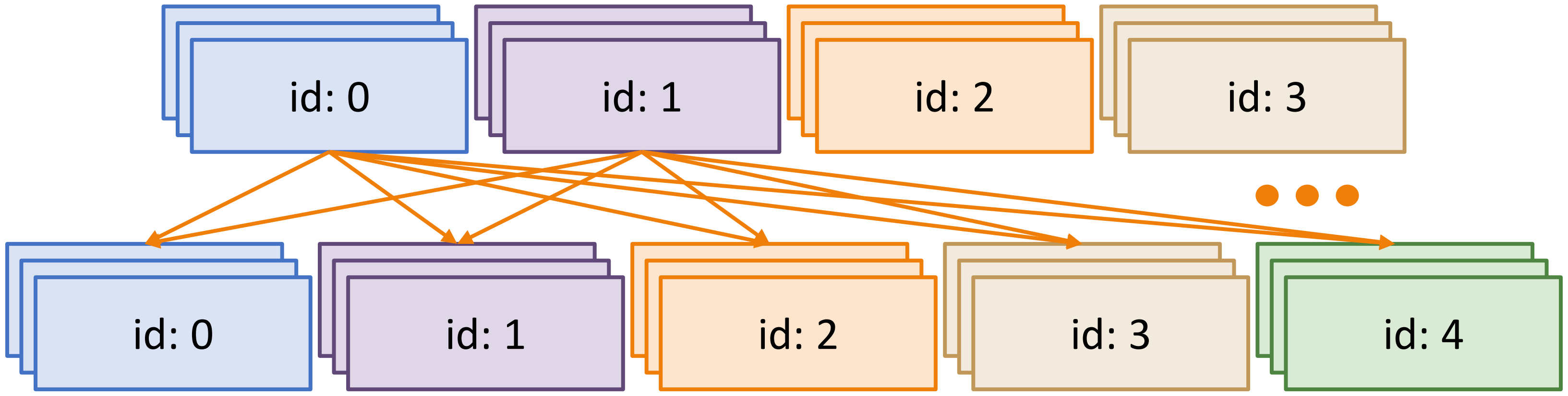
Replicate the sharded caches

- We can replicate each cached node for fault tolerance
  - If a node fails, replicas of the node can still serve data
- How to ensure consistency of all data copies?
- We will look at this replication problem later



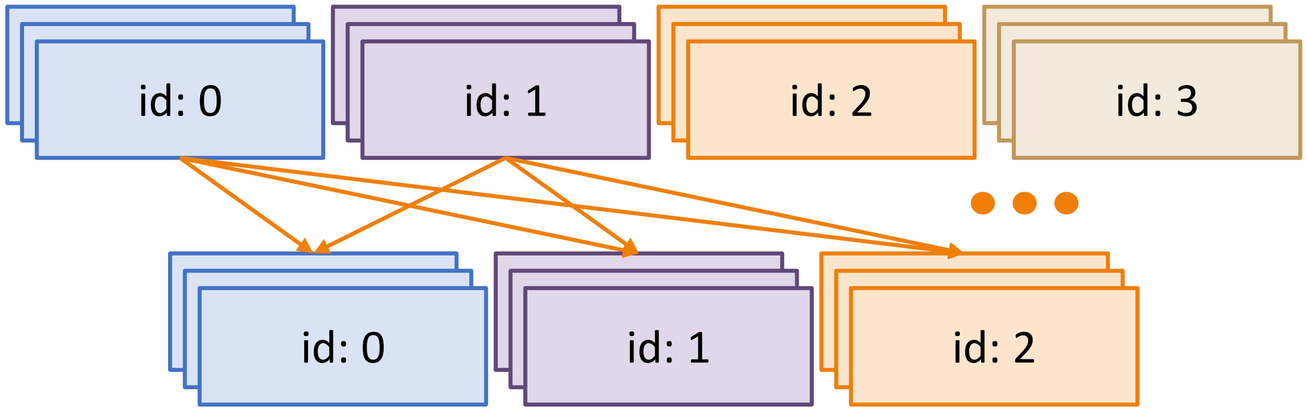
What if load increases?

- Add a cache node automatically for elastic scaling
  - Option 1: drop cache data
    - Drop cache data at all nodes, restart caching with N+1 nodes
    - Cold caches put too much load on database
  - Option 2: repartition cache data
    - Repartition (shuffle) the cache data from N nodes to N+1 nodes
    - Cache remains warm but repartitioning adds load on the system



What if load decreases?

- Remove a cache node automatically
  - Same drop and repartition cache options
  - For repartitioning from N to N-1 nodes, evict least recently used cache data
  - Removed cache node can be used for other purposes, reduces cost of caching



Where is the cache, cached data?

- Webservers (clients) needs to know
  - Membership: location of all the cache nodes, e.g., IP addresses
    - Membership changes as cache nodes are added or removed
  - Mapping: location of cache node that holds data for a given key
    - Mapping changes with repartitioning
- Typically, this information is kept at a coordination service
  - Webservers may periodically check for changed information, or
  - Service can inform webservers when information changes
  - Coordination service must itself by scalable and reliable
- We will look at this coordination problem later

What about data consistency?

- Many reasons for inconsistent data accesses
  - Cache and database are accessed concurrently
  - Cache node fails, restarts, recovers data from disk, has stale data
  - Some clients bypass cache and access database directly
  - Caches are replicated and inconsistent with each other
- Data consistency
  - Reads return the value from most recent write
  - Makes it easier to write applications
    - No stale reads, no conflicting updates

Do we always need data consistency?

- While sharding helps scaling the cache,
  data consistency can limit scalability and availability
  - Need to know whether cache is up-to-date
  - Need to wait for cache to be up-to-date
  - What if network between cache, database has temporarily failed?
- Often, applications can handle stale data
  - Users don’t care about the exact value of “likes” on a webpage
  - Compute based on stale data, incorporate new data as it arrives
  - Sell an item, if eventually not available, issue refund
- We will look at various consistency models later

Conclusions

- This course teaches you about distributed systems
- A distributed system consists of
  - Multiple nodes connected by a network
  - Nodes work together to provide services
- Goals
  - Provide scalable, fault tolerant, consistent, secure services
  - Applications use these services, focus on program logic

When you hear large-scale, web, cloud, data center services or apps …

Think distributed systems

They are everywhere!