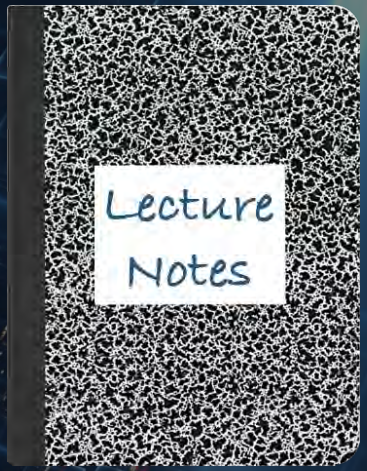
CS 417 – DISTRIBUTED SYSTEMS

Week 1: Part 1
Introduction to Distributed Systems



Paul Krzyzanowski

What is a Distributed System?

A collection of independent computers connected through a communication network that work together to accomplish some goal

No shared operating system

No shared memory

No shared clock

Almost as old as computers

SAGE defense
24 computers (1958)

ARPANET
(1969)

Distributed
File Systems
AFS (1983)
NFS (1984)

www
(1991)

Akamai
CDN
(1998)

Facebook
(2004)

Kubernetes
(2014)

1950 1960 1970 1980 1990 2000 2010 2020 2030

First computer:
Colossus (1944)

(1964)

Ethernet
(1973)

ARPANET
transitions to
TCP/IP (1983)

Facebook
(2004)

The Mother of All Demos

- Presented by Douglas Engelbart
  from the Stanford Research Institute's Augmentation Research Center
- Introduced the NLS (the oN-Line System)
- First demo of:

Windows	Mouse
Hypertext	Word processing
Bitmapped graphics	Revision control
Video conferencing	Real-time editing

All this in 1968!

Why do we need distributed systems?

1. Scale
2. Collaboration
3. Reduced latency
4. Mobility
5. High availability & Fault tolerance
6. Incremental cost
7. Delegated infrastructure & operations

1. Scale

Scale: Increased Performance

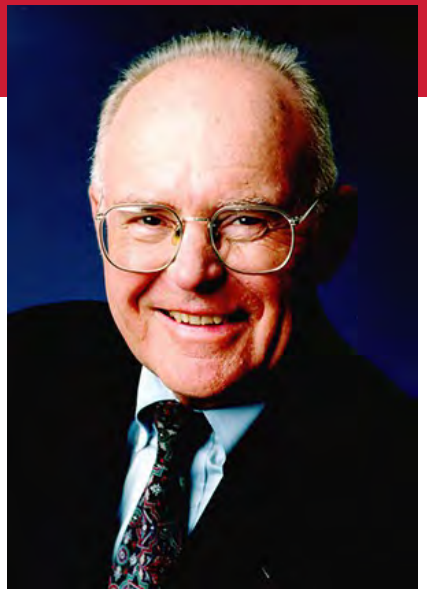
Computers are getting faster

Moore's Law

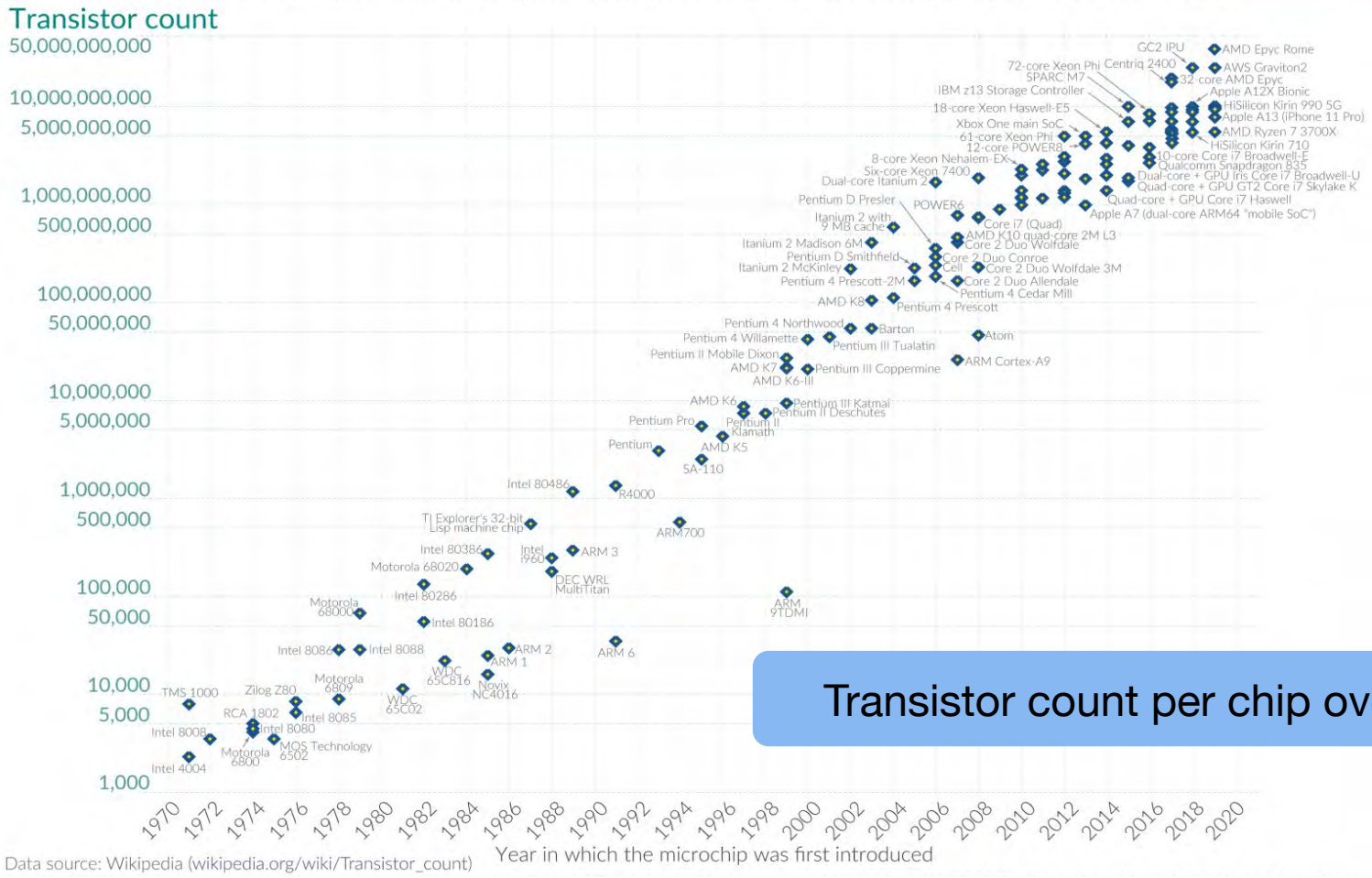
Prediction: the number of transistors on a chip doubles approximately every two years

This implied that performance doubles too

Not a real law – just an observation from 1965



Gordon Moore
Co-founder, Intel



Our World in Data

Transistor count per chip over time

Source: https://en.wikipedia.org/wiki/Moore%27s_law#/media/File:Moore's_Law_Transistor_Count_1970-2020.png

Scaling a single system has limits

It's getting harder for technology to keep up with Moore's law

- More cores per chip → requires multithreaded programming
- Technology limits the die size and # of transistors per chip
  - **Intel Xeon 8490H**: 60 cores, 120 threads (~17,000+/chip!)
    - Tens of billions transistors, 350 watts @ 1.9-3.5 GHz
  - **Apple M3 Ultra**: 32 cores, 80 graphics cores, 32-core neural engine
    - 184 billion transistors, 270 watts (max)
  - **AMD EPYC 9965**: 192 cores, 384 threads ($10,000+/chip)
    - 10s of billions transistors, 450-500 watts @ 2.25-3.7 GHz
  - **NVIDIA Blackwell B200 GPU**: 18,944 CUDA cores & 592 Tensor cores per GPU
    - 2 Blackwell GPUs + NVIDIA Grace CPU
    - 208 billion transistors

How Moore's Law Kept Up Over Time

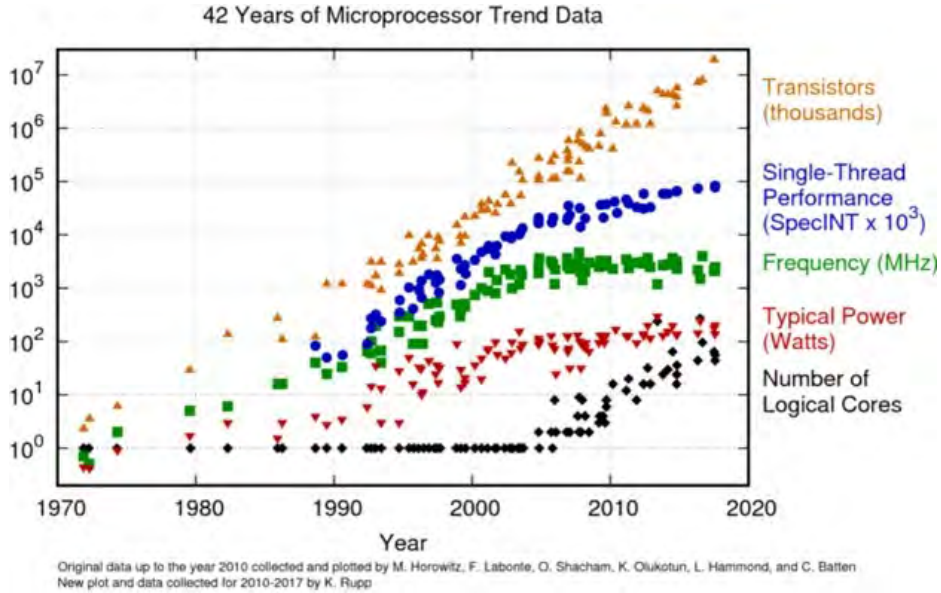
Transistor size & operating frequency couldn't keep up with Moore's Law

- Increases in processor performance have not been keeping up with Moore's Law since around 2005 (blue dots in the chart on the next page)
  - ***Dennard Scaling*** predicted that performance per watt will increase exponentially. This growth also tapered off in the early 2000s
- Adding more processor cores helped improve performance
  - ***Amdahl's law*** shows there's a diminishing return on parallelization … applications don't benefit from an unlimited number of cores
- **Heterogeneous computing** helped too
  - Adding specialized processing cores: graphics processors (GPU), image signal processors, cryptographic processors, …

Processor performance gains & limitations are due to multiple factors

- [Orange ▲]: overall performance, obtained by multiplying individual colors below
- [Blue ●] *Moore's law*: performance based on number of transistors in a processor
- [Red ▼] *Dennard Scaling*: performance per watt
- [Black ◆] Number of cores per chip (black): allowed performance to further increase

– *Amdahl's law*: decreasing returns on parallelization



2000s: a move to *heterogeneous computing* — multiple specialized processors on a chip: GPUs, neural engines, image signal processors, crypto processors

2023: System Technology Co-Optimization (STCO) – holistic view of heterogeneous computing – optimizes transistor & interconnect design for each functional component on a chip

More performance

Horizontal scaling
(distributed load across more systems)
vs.
Vertical scaling
(use more powerful systems)

What if we need more performance than we can get from a single CPU?

Combine them ⇒ multiprocessors
But these have scaling limits and cost $

Distributed systems allow us to achieve massive performance via horizontal scaling

Our computing needs exceed CPU advances (& bandwidth)

Pixar movie rendering
- Toy Story (1995): 294 CPU cores – 30 hours/frame
- Toy Story 4 (2019): 60-160 hours/frame
- Elemental (2023): 151,000 CPU cores
  - Some frames took 1,000 hours/frame

Meta
- ~100M requests/second with 4B+ users
- Async platform processes “trillions of function calls per day” on “more than 100,000 servers”
- Social graph: TAO “serves more than one quadrillion queries a day”

Google search
- About 100k searches per second
- A single query uses approximately 1,000 computers and produces a result in ~0.2 seconds
- Search index is over 100 million GB
- ~130 large-scale data centers

OpenAI
- GPT-4 trained on ~25,000 NVIDIA A100 GPUs in 90-100 days
- Model size: ~1.7 trillion parameters across 120 layers

Example: Google

- In 1999, it took Google one month to crawl and build an index of about 50 million pages
- In 2012, the same task was accomplished in less than one minute … with more pages.
- 16% to 20% of queries that get asked every day have never been asked before
- Every query has to travel on average 1,500 miles to a data center and back to return the answer to the user
- A single Google query uses 1,000 computers in 0.2 seconds to retrieve an answer

Source: http://www.internetlivestats.com/google-search-statistics/

2. Collaboration & Network Effects

Collaboration & Content

- Collaborative work & play
- Social connectivity
- Commerce
- News & media

Network Effects = Metcalfe's Law

The value of a telecommunications network is proportional to the square of the number of connected users of the system.

The Network Effect ⇒ This makes networking interesting to us … and to investors!

3. Reduced latency

Reduced Latency

- Place data & computation close to the user
  - Regional data centers
  - Content delivery networks
  - Edge computing
- Caching vs. replication
  - **Replication**: multiple copies of data
    Goal: **increase fault tolerance**
  - **Caching**: temporary copies of frequently accessed data closer to where it's needed
    Goal: **reduce latency**
- Some caching services:
  Akamai, Cloudflare, Amazon Cloudfront, Apache Ignite

4. Mobility & Ubiquitous Computing

Mobility & Ubiquitous Computing

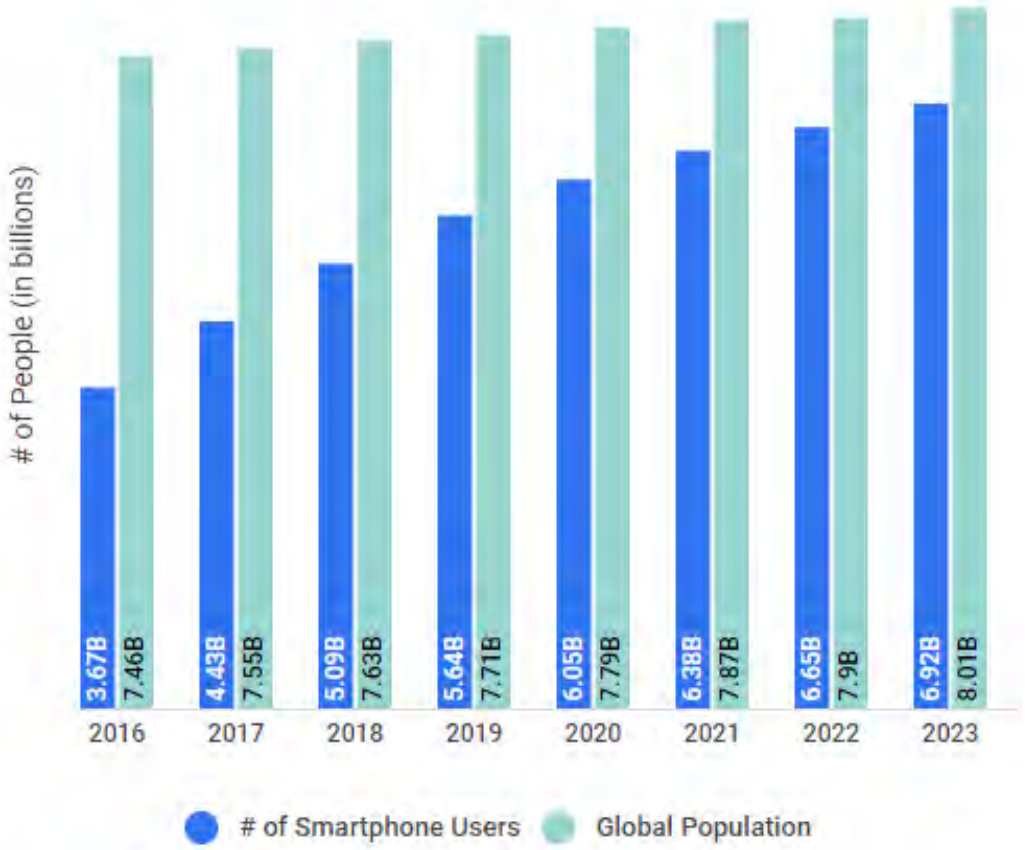
>6 billion smartphone users

Remote sensors

- Cars
- Traffic cameras
- Toll collection
- Shipping containers
- Vending machines

IoT = Internet of Things

- Since 2017: more IoT devices than humans



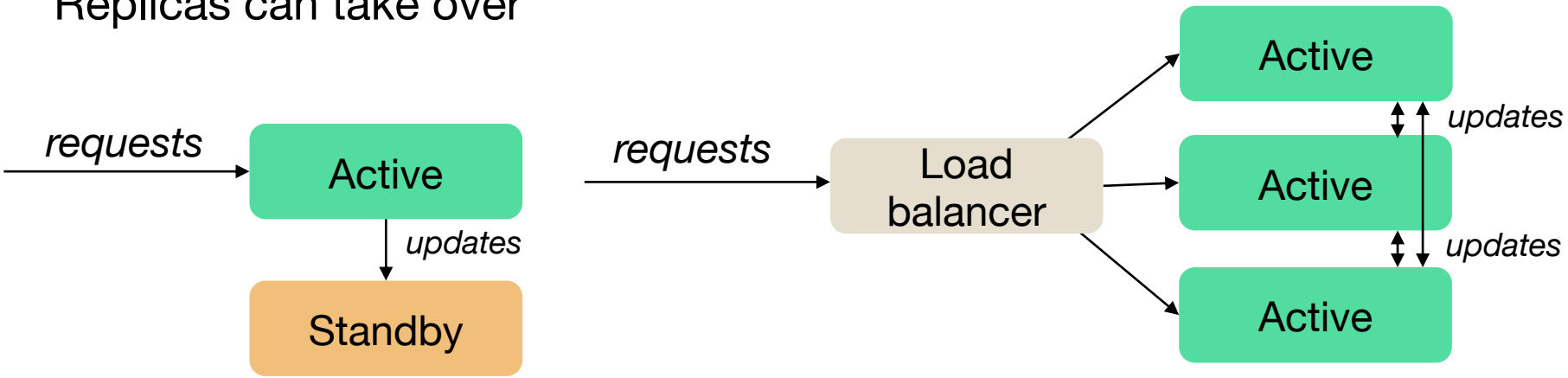
https://www.zippia.com/advice/smartphone-usage-statistics

5. High availability & Fault tolerance

High availability through replication

Components fail: computers, processes, disks, data centers, …

Replicas can take over



Availability requires fault tolerance

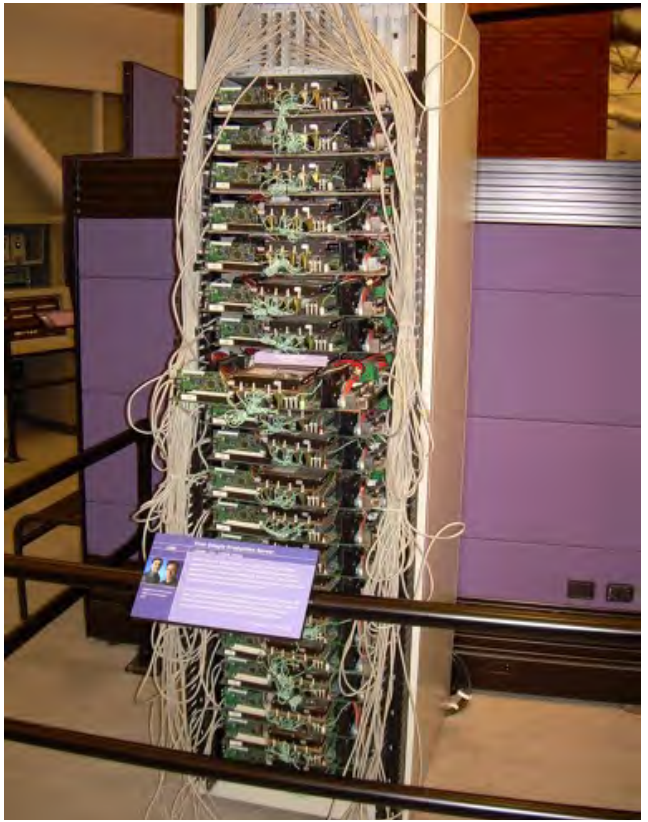
- Fault tolerance
  - Identify & recover from component failures
- Recoverability
  - Software can restart and function
  - May involve restoring state

6. Incremental growth & cost

Incremental cost

Version 1.0 does not have to be the full system

- Add more servers & storage over time
- Scale also implies cost:
  you don't need millions of $ for v1.0



Google

7. Delegated infrastructure & operations

Delegated operations

- **Offload responsibility**
  - Let someone else manage systems
  - Use third-party services
- **Speed deployment**
  - Don’t buy & configure your own systems
  - Don’t build your own data center
- **Modularize services on different systems**
  - Dedicated systems for storage, email, etc.
- **Use cloud, network attached storage**
  - Let someone else figure out how to expand storage and do backups

Transparency

High level: hide distribution from users

Low level: hide distribution from software

- **Location transparency**
  Users don't care where resources are
- **Migration transparency**
  Resources move at will
- **Replication transparency**
  Users cannot tell whether there are copies of resources
- **Concurrency transparency**
  Users share resources transparently
- **Parallelism transparency**
  Operations take place in parallel without user's knowledge
- **Failure transparency**
  Lower-level software works around any failures – things just work

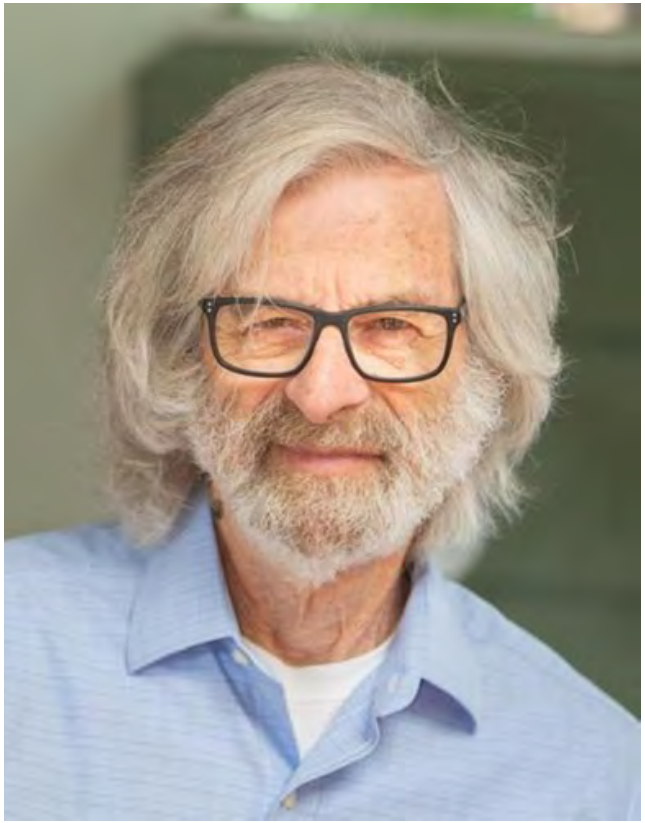
Single System Image

A collection of independent computers that appears to its users as a single coherent system

Core challenges in distributed systems design

1. Partial Failure
2. Lack of global state
3. Unreliable communication

Partial Failure



You know you have a distributed system when the crash of a computer you've never heard of stops you from getting any work done.

— *Leslie Lamport*

Handling failure

Failure is a fact of life in distributed systems!

In local systems, failure is usually **total** (*all-or-nothing*)

In distributed systems, we get **partial failure**

- A component can fail while others continue to work
- Failure of a network link is indistinguishable from a remote server failure
- Send a request but don't get a response ⇒ what happened?

No **global state**

- There is no global state that can be examined to determine errors
- There is no agent that can determine which components failed and inform everyone else

Handling failure

Handle **detection**, **recovery**, and **restart**

Availability = fraction of time system is usable

- Achieve with redundancy
- But keeping all copies consistent is an issue!

Reliability: How long the system can run without failing

- Includes ensuring data does not get lost
- Includes security

A system can be highly available but not reliable if it recovers quickly from failure, but may return stale results

A system can be reliable but not highly available if it does not return data rather than risk returning incorrect (stale) results.

Handling failure

Handle **detection**, **recovery**, and **restart**

Availability = fraction of time system is usable

- Achieve with redundancy
- But keeping all copies consistent is an issue!

Reliability: How long the system can run without failing

- Includes ensuring data does not get lost
- Includes security

*A system can be **highly available but not reliable** if it recovers quickly from failure, but may return stale results)*

*A system can be **reliable but not highly available** if it does not return data rather than risk returning incorrect (stale) results.*

Nines

Availability metrics are often expressed in nines

Nines	Availability	Downtime per year	Downtime per month	Downtime per week
Two nines	99%	3.65 days	7.3 h	101 min
Three nines	99.9%	8.76 h	43.8 min	10.1 min
Four nines	99.99%	52.6 min	4.39 min	1.01 min
Five nines	99.999%	5.26 min	2.19 min	6 s

Increasing availability through redundancy

Redundancy = replicated components = **parallel system**

The service can run even if some systems die

Reminder: $P(\text{A and B}) = P(\text{A}) \times P(\text{B})$

If P(any one system down) = 5%

P(two systems down at the same time) = 5% × 5% = 0.25%

Uptime = 1 – downtime = 1 – 0.0025 = 99.75%

We get 99.7% uptime instead of 95% because we need *both* replicated components to fail instead of just one.

~~High availability~~

No redundancy = dependence on all components = **series system**

We need ***all*** systems running to provide a service

Example – same assumption as before: P(any one system down) = 5% = 0.05

P(complete system down) = 1 - *P*(A is up AND B is up)

= 1 – (1 – 0.05) × (1 – 0.05) = 1 – 0.95 × 0.95 = **9.75%**

⇒ 39x greater than a single component failure with redundancy!

Uptime = 1 – downtime = 1 – 0.0975 = 90.25%

With a large # of systems, P(any system down) approaches 100% !

Requiring a lot of components to be up & running is a losing proposition.
With large enough systems, something is always breaking!

Availability: series & parallel systems

Series system: The system fails if ANY of its components fail

P(system failure) = 1 - P(system survival)

If P_i = *P*(component *i* fails) then for *n* components:

$$P(system\ failure) = 1 - \prod_{i}^{n}(1 - P_i)$$

Parallel system: The system fails only if ALL of its components fail

P(system failure) = P(component$_1$ fails) × P(component$_2$ fails) …

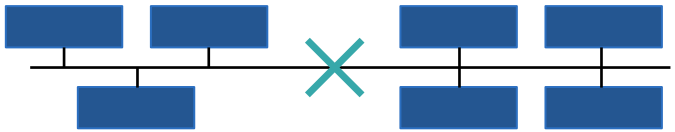
$$P(system\ failure) = \prod_{i}^{n} P_i$$

System Failure Types

- **Fail-silent**
  - Failed component stops functioning and stops responding
  - Detect failed components via **timeouts**
    - But you can't count on timeouts in asynchronous networks
      - And what if the network isn't reliable?
- **Fail-stop**
  - Like fail-silent but we assume we can reliably detect the failure
- **Fail-restart**
  - Component stops but then restarts
  - Danger:
    possible **stale state** — the system didn't get updates when it was dead

Network Failure Types

- **Omission**
  - Failure to send or receive messages
    - Due to queue overflow in router, corrupted data, receive buffer overflow
- **Timing**
  - Messages take longer than expected
    - We may assume a system is dead when it isn't
    - Unsynchronized clocks can alter process coordination
- **Partition**
  - Network breaks into two or more sub-networks that cannot communicate with each other



Network & System Failure Types

- **Silent failures (fail-silent)**
  - A failed component (process or hardware) does not produce any output

- **Byzantine failures**
  - Instead of stopping, a component produces faulty data
  - Due to bad hardware, software, network problems, or malicious interference

Design goal: avoid single points of failure

Lack of Global State (Knowledge)

What is state?

State = Information about some component at a point in time

- Network connection info, process memory, list of clients with open files, lists of which clients finished their tasks

No global knowledge

- Nobody has the true **global state** of a system
  - There is no global state that can be examined to determine errors
  - There is no agent that can determine which components failed and inform everyone else
  - No shared memory
- A process knows its current state
  - It may know the *last reported state* of other processes
  - It may periodically report its state to others

No global state means no foolproof way to detect failure in all cases

Unreliable Communications

Latency

Network messages may take a long time to arrive

- **Synchronous network model**
  - There is some upper bound, *T*, between when a node sends a message and another node receives it
  - Knowing *T* enables a node to distinguish between a node that has failed and a node that is taking a long time to respond
- **Partially synchronous network model**
  - There's an upper bound for message delivery, but the programmer doesn't know it – it has to be discovered
  - Protocols will operate correctly only if all messages are received within some time, *T*
    - We cannot make assumptions on the delay time distribution
- **Asynchronous network model**
  - Messages can take arbitrarily long to reach a peer node
  - **This is what we get from the Internet!**

Latency & asynchronous networks

Asynchronous networks can be challenging

Messages may take an unpredictable amount of time

- We may think a message is lost but it's really delayed
- May lead to retransmissions → duplicate messages
- We may assume a service is dead when it isn't
- May cause messages to arrive in a different order
  … or a different order on different systems
  → Our perception of event ordering can be different from reality

Latency

- Speed up data access via **caching** – temporary copies of data
- Keep data close to where it's processed to maximize efficiency
  - Memory vs. disk
  - Local disk vs. remote server
  - Remote memory vs. remote disk
  - **Cache coherence**: cached data can become **stale**
    - The main version may change → cached data will need to be invalidated or updated
      - Need mechanism for systems to detect that the cached copies are no longer valid
    - System using the cache may change the data → needs to propagate results
      - ***Write-through cache***
      - But updates take time ⇒
        meanwhile, access to data leads to **inconsistencies** (**incoherent views**)

Security

Security

- Traditionally managed by operating systems
  - Users authenticate themselves to the system
  - Each user has a unique user ID (UID)
  - Access permissions are enforced by the OS and based on the user ID
- Now applications must take responsibility for
  - Identification
  - Authentication
  - Access control
  - Encryption, tamper detection
  - Audit trail

Security

- The environment
  - Public networks, remotely-managed services, 3rd party services
  - Trust: do you trust how the 3rd party services are written & managed?
- Some issues:
  - Malicious interference, bad user input, impersonation of users & services
  - Protocol attacks, input validation attacks, time-based attacks, replay attacks

We will rely on cryptography (hashes, cryptography) for identity management, authentication, encryption, tamper detection

- *Users also want convenience*
  - Single sign-on, no repeated entering of login credentials
- Controlled access to services
  - You may allow a service may to access your photos but not your contacts)

Other design considerations

Other considerations

- Scaling up & scaling down
  - Need to be able to add and remove components
  - Impacts failure handling
    - If failed components are removed, the system should still work
    - If replacements are brought in, the system should integrate them
- Algorithms & environment
  - Distributable vs. centralized algorithms
  - Programming languages
  - APIs and frameworks

Main Themes:
How will we do all this?

Main themes in distributed systems

- **Availability & fault tolerance**
  - Fraction of time that the system is functioning
  - Dead systems, dead processes, dead communication links, lost messages
- **Scalability (Elasticity)**
  - Things may be easy on a small scale
  - But less so on a large scale
    - Geographic latency (multiple data centers), administering many thousands of systems
- **Latency & asynchronous processes**
  - Processes run asynchronously: concurrency
  - Some messages may take longer to arrive than others
- **Security**
  - Authentication, authorization, encryption

Key approaches in distributed systems

- **Divide & conquer**
  - Break up data sets (sharding) and have each system work on a small part
  - Merging results is usually the easy & efficient part

- **Replication**
  - For high availability, caching, and sharing data
  - Challenge: keep replicas consistent even if systems go down and come up

- **Quorum/consensus**
  - Enable a group to reach agreement

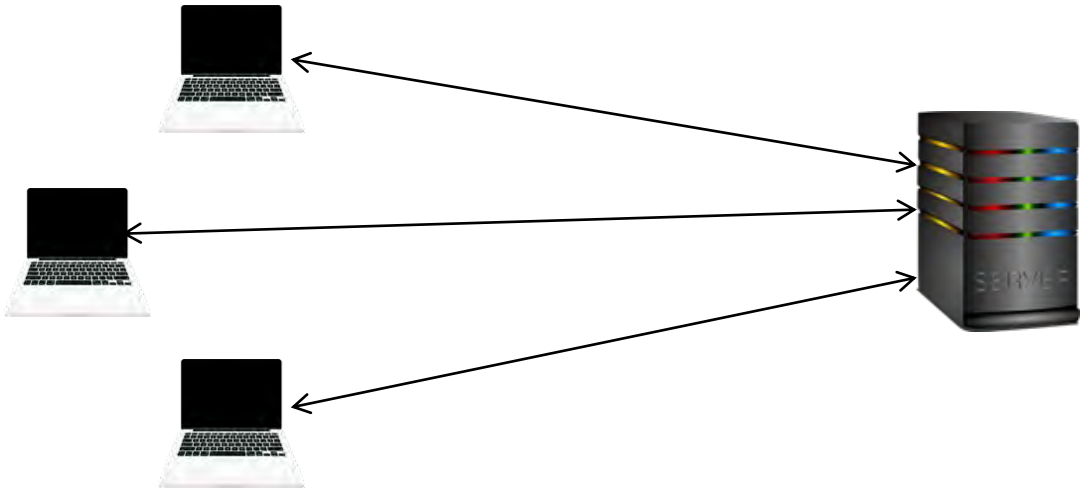
Service Architectures

Centralized model

- Traditional time-sharing system
- Single workstation/PC with one or more cores/CPUs
- Relies on vertical scaling
- Limiting factor: number of CPUs in system
  - Contention for same resources (memory, network, devices)

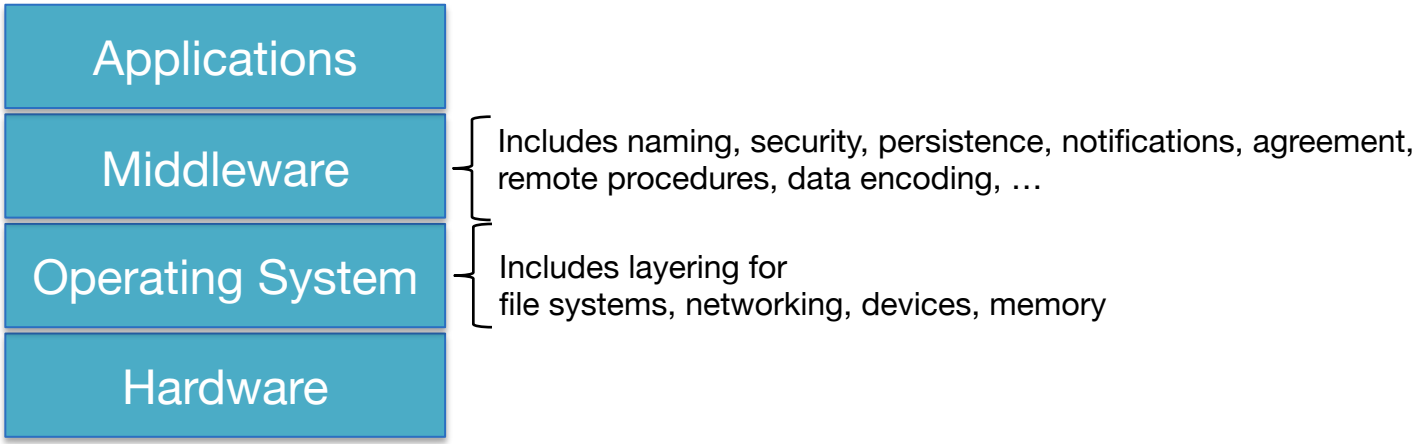
Client-Server model

- Clients send requests to servers
- A server is a system that runs a service
- Clients do not communicate with other clients



Layered architectures in software design

- Break functionality into multiple layers
- Each layer handles a specific abstraction
  - Hides implementation details and specifics of hardware, OS, network abstractions, data encoding, …



We can apply the same approach to distributed systems design

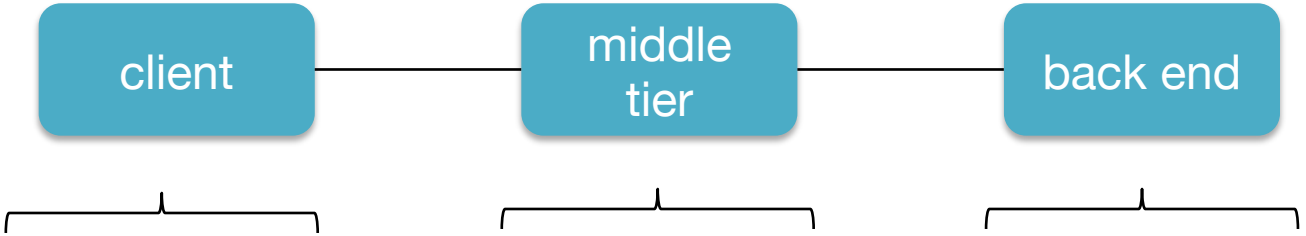
Multi-Tier (N-Tier) Architectures

Distributed systems analogy to a layered architecture

- Each tier (layer)
  - Runs as a network service
  - Is accessed by surrounding tiers

The basic client-server architecture is a two-tier model

Classic three-tier architecture

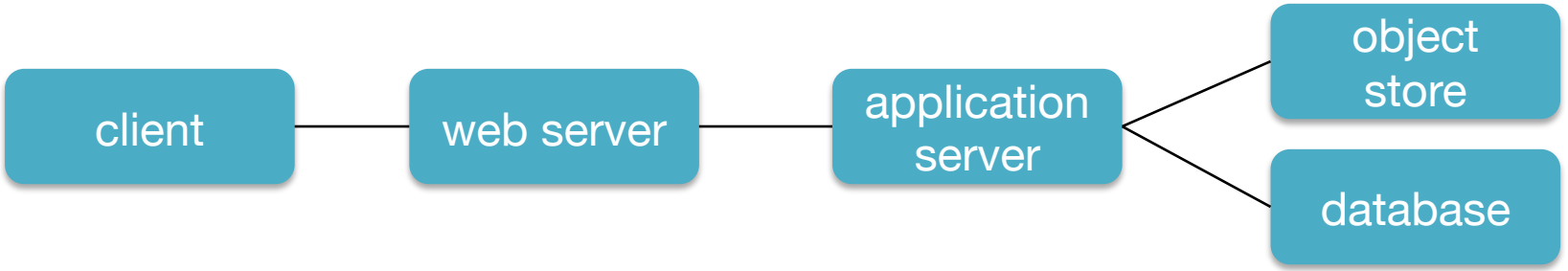


User interface
- Data presentation & validation

Application Logic
- Queuing requests
- Coordinating a transaction among multiple servers
- Managing connections
- Formatting/converting data

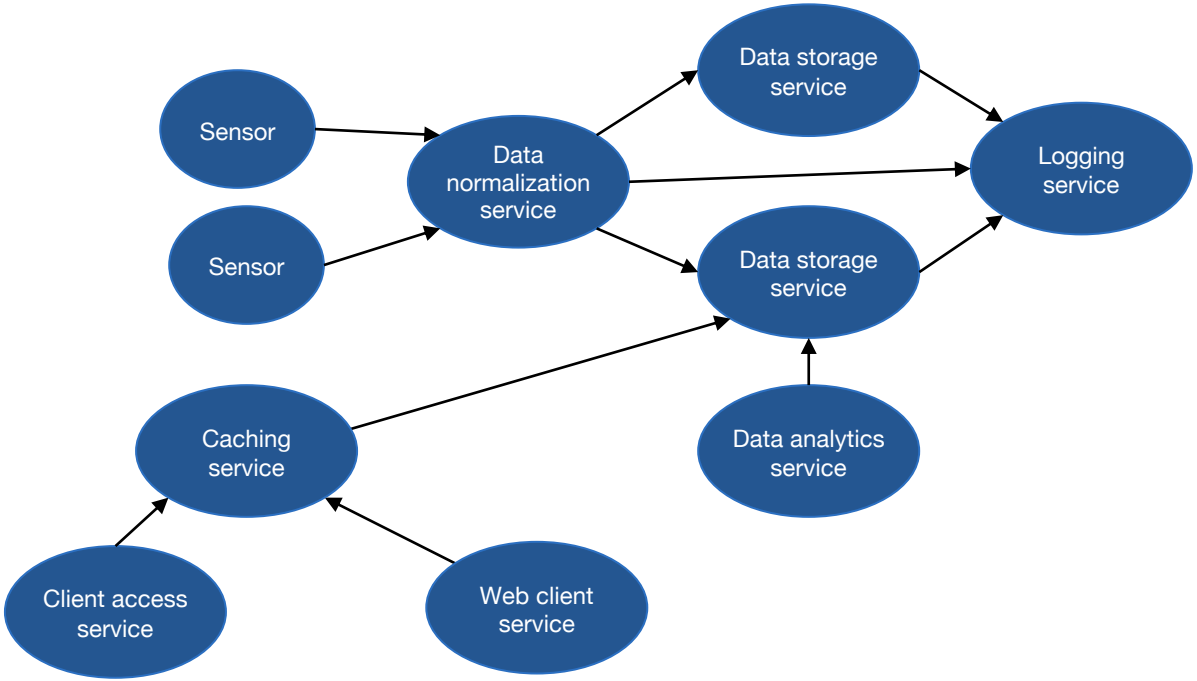
Storage
- Database system
- Legacy software

Multi-tier example



Each tier can be implemented, scaled, and managed independently

Microservices Architecture



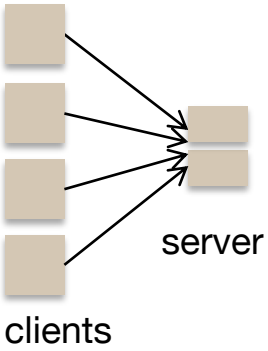
Collection of autonomous services

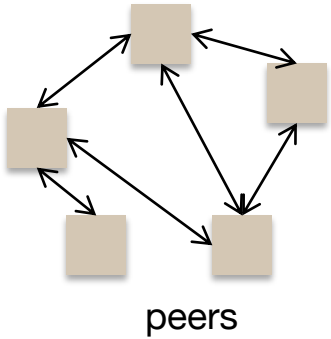
- Each service:
  - Runs independently
  - Has a well-defined interface
  - May be shared by multiple applications
- Main application coordinates interactions

Unlike multi-tier, no strict layering

Peer-to-Peer (P2P) Model

- No reliance on servers
- Machines (peers) communicate with each other
- Goals
  - Robustness: no single point of failure
  - Self-scalability
- Example: BitTorrent





Worker Pools and Compute Clusters

- Collection of CPUs that can be assigned processes on demand
- Similar to hybrid model
  - Coordinator dispatches work requests to available processors
- Render farms, big data processing, machine learning

Cloud Computing

Resources are provided as a network (Internet) service

Software as a Service (SaaS)

Remotely hosted software: email, productivity, games, …

Google Workspace, Microsoft 365, Slack, Zoom

Platform as a Service (PaaS)

Execution runtimes, databases, web servers, development environments: users don't manage servers

Google App Engine, AWS Elastic Beanstalk, Azure App Service

Infrastructure as a Service (IaaS)

Compute + storage + networking: VMs, storage servers, load balancers,…

AWS EC2, Google Compute Engine, Azure Virtual Machines

Storage as a Service, Database as a Service, Function as a Service, …

Various building blocks

Amazon S3/Azure Blob/Google Cloud Storage, Amazon RDS/Azure SQL/DynamoDB, AWS Lambda

The End