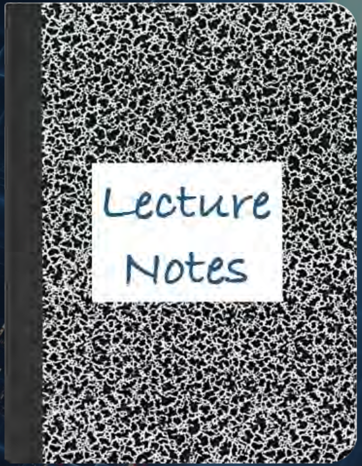


CS 417 – DISTRIBUTED SYSTEMS

Week 1: Part 1

Introduction to distributed systems

Paul Krzyzanowski

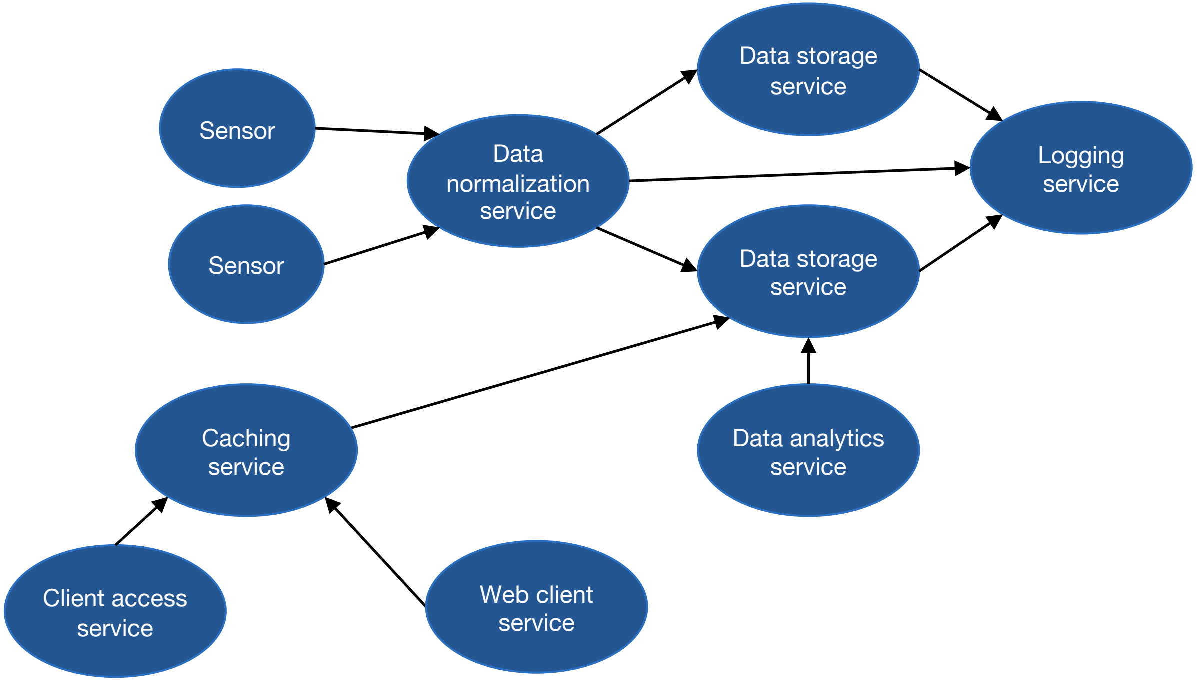
What is a Distributed System?

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

What is a Distributed System?

A distributed system is a collection of services accessed via network interfaces



Single System Image

Collection of independent computers that appears as a single system to the user(s)

Independent = autonomous, self-contained
Single system = user not aware of distribution

Classifying parallel and distributed systems

Flynn’s Taxonomy (1966)

Classify computer architectures by looking at the number of instruction streams and number of data streams

1. **SISD** — Single Instruction, Single Data stream
   – Traditional uniprocessor systems
2. **SIMD** — Single Instruction, Multiple Data streams
   – Array (vector) processors
   – Examples:
     - GPUs – Graphical Processing Units for computer graphics, GPGPU (General Purpose GPU): AMD/ATI, NVIDIA
     - AVX: Intel’s Advanced Vector Extensions
3. **MISD** — Multiple Instructions, Single Data stream
   – Sometimes (rarely!) applied to classifying fault-tolerant redundant systems
4. **MIMD** — Multiple Instruction, Multiple Data streams
   – Multiple computers, each with a program counter, program (instructions), data
   – **Parallel and distributed systems**

Subclassifying MIMD

Memory

- Shared memory systems: **multiprocessors**
- No shared memory: networks of computers, **multicomputers**

Interconnect

- Bus
- Switch

Delay/bandwidth

- Tightly coupled systems
- Loosely coupled systems

Multiprocessors & Multicomputers

Multiprocessors

- Shared memory
- Shared clock
- Shared operating system
- All-or-nothing failure

Multicomputers (networks of computers) ⇒ *distributed systems*

- No shared memory
- No shared clock
- Partial failures
- Inter-computer communication mechanism needed: the network
  - Traffic volume much lower than memory access

Why do we want 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

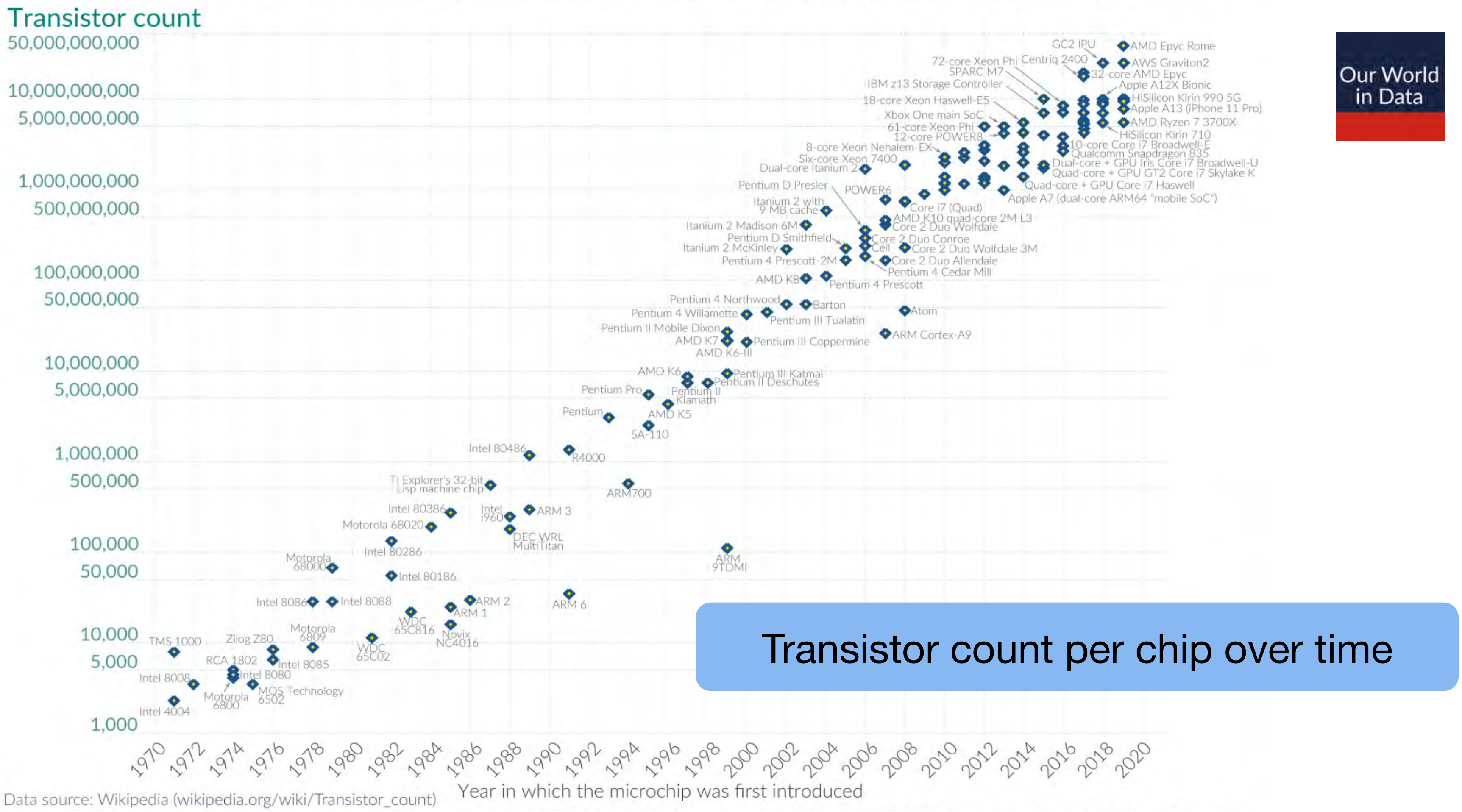
Scale: Increased Performance

Computers are getting faster

Moore's Law

Prediction: performance doubles approximately every 18 months because of faster transistors and more transistors per chip

Not a real law – just an observation from the mid 1970s



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

Getting harder for technology to keep up with Moore's law

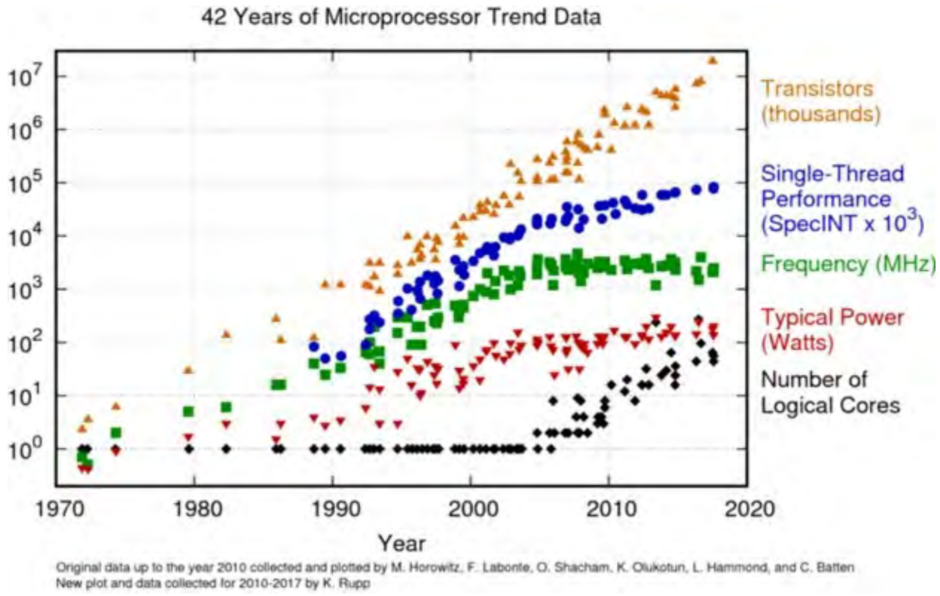
- More cores per chip
  → requires multithreaded programming

- There are limits to the die size and # of transistors
  - **Intel Xeon W-3175X**: 28 cores per chip ($2,999/chip!)
    - 8 billion transistors, 255 watts @ 3.1-4.3 GHz
  - **Apple M1 Ultra**: 20 cores, 64 graphics cores, 32-core neural engine
    - 167 billion transistors, 110 watts (max)
  - **AMD EPYC 7601**: 32 cores per chip ($4,200/chip)
    - 19.2 billion transistors, 180 watts
  - **NVIDIA GeForce RTX 2080 Ti**: 18,432 CUDA cores & 576 Tensor cores per GPU
    - Special purpose apps: Graphics rendering, neural networks

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's law*: 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, cryptoprocessor

2023: system technology co-optimization (STCO) optimizes transistor & interconnect design for each functional component on a chip

How Moore's Law Kept Up Over Time

- Transistor size & operating frequency couldn't keep up with the predictions of Moore's Law
  - Increases in processor performance have not been keeping up with Moore's Law since around 2005 (blue dots in the chart)
  - Another "law": Dennard's Law predicted that performance per watt will increase exponentially. This growth also tapered off in the early 2000s
- Adding more processor cores helped improve performance
  - But only if applications are multithreaded and can make use of the cores
- Heterogeneous computing helped too
  - Adding specialized processing cores: graphics processors (GPU), image signal processors, cryptographic processors, …

More performance

Horizontal scaling

(distributed load across more systems)

vs.

Vertical scaling

(use more powerful systems)

What if we need more performance than a single CPU?

Combine them ⇒ multiprocessors

But these have scaling limits and cost $

Distributed systems allow us to achieve massive performance

Our computing needs exceed CPU advances

Movie rendering

– *Toy Story* (1995) – 117 computers; 45 mins — 30 hours to render a frame
– *Toy Story 4* (2019) – 60-160 hours to render a frame
– Pixar uses 2,000 machines with an aggregate of 24,000 cores

Google

– Over 63,000 search queries per second on average
– Over 130 trillion pages indexed
– Uses hundreds of thousands of servers to do this

Facebook

– Approximately 100M requests per second with 4B users

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.
- 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

Collaboration & Content

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

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

- Cache data close to where it is needed
- *Caching* vs. *replication*
  - Replication: multiple copies of data for increased fault tolerance
  - Caching: temporary copies of frequently accessed data closer to where it's needed
- Some caching services:
  Akamai, Cloudflare, Amazon Cloudfront, Apache Ignite

4. Mobility

Mobility

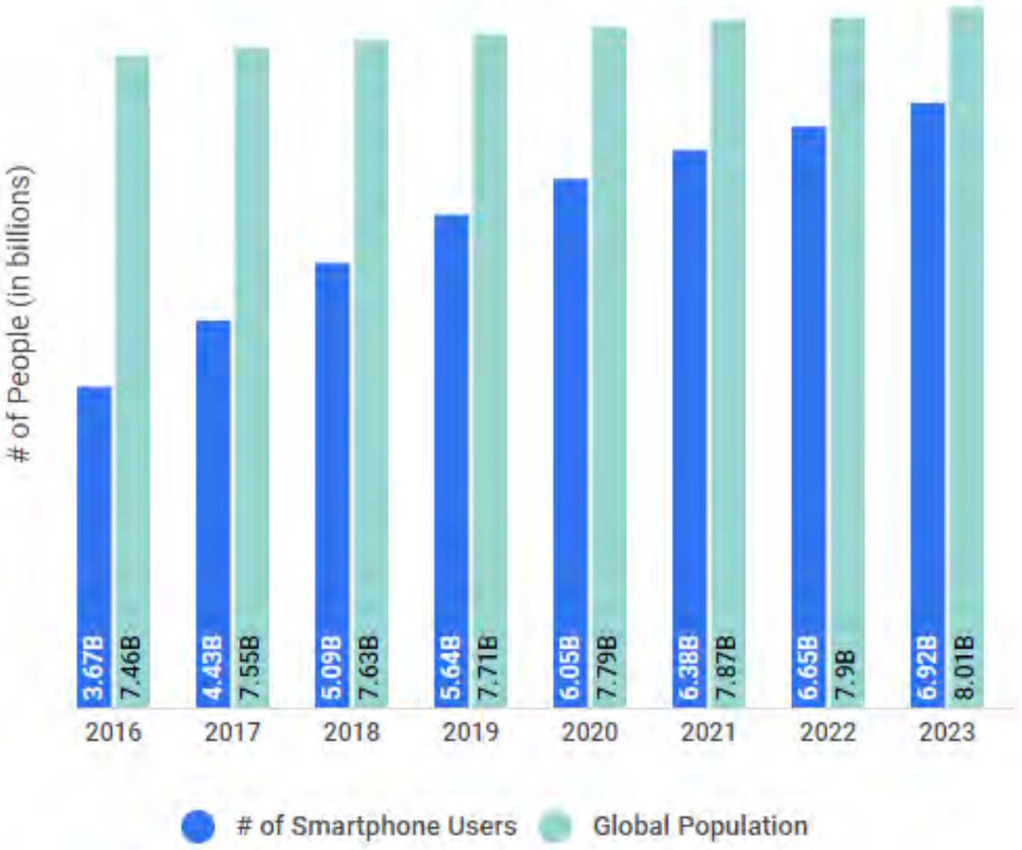
>6 billion smartphone users

Remote sensors

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

IoT = Internet of Things

- Since 2017: more IoT devices than humans



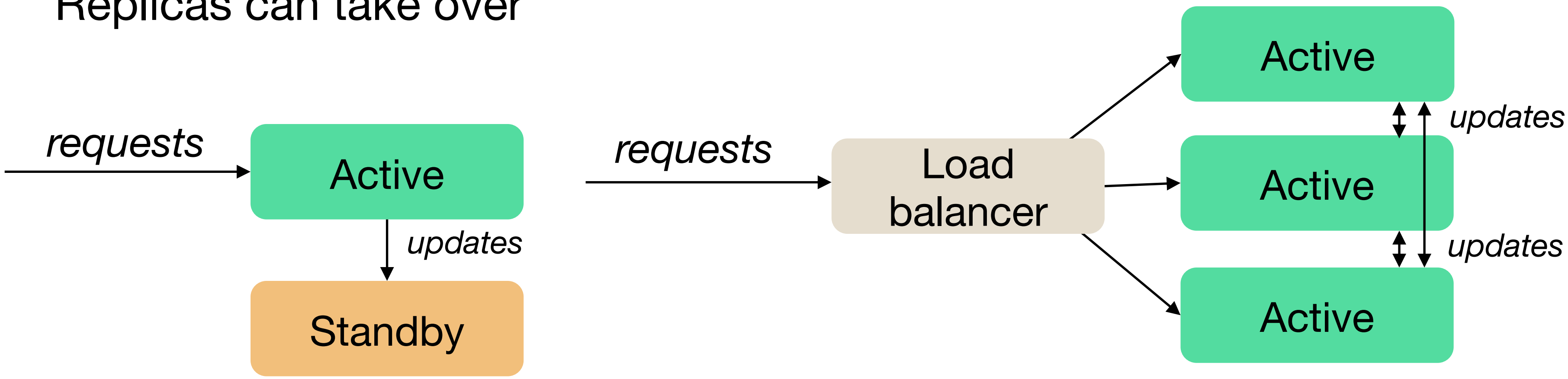
https://www.zippia.com/advice/smartphone-usage-statistics

5. High availability & Fault tolerance

High availability through replication

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

Replicas can take over



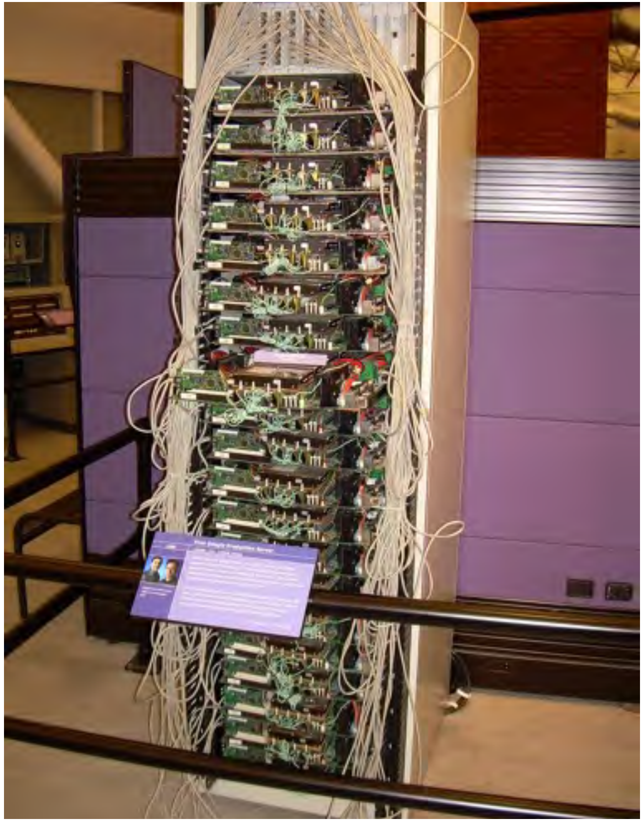
Availability requires fault tolerance

- 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 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



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 as a Design Goal

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

Core challenges in distributed systems design

1. Concurrency
2. Latency
3. Partial Failure

Concurrency

Concurrency

- Lots of requests may occur at the same time

- Need to deal with concurrent requests
  - Need to ensure consistency of all data
  - Understand critical sections & mutual exclusion
  - Beware: mutual exclusion (locking) can affect performance

- We often replicate data (or cache it) – need to update all replicas
  - Replication adds complexity
  - All operations must appear to occur in the same order on all replicas
    - Need to worry about out-of-order messages, undelivered messages, dead replicas

Latency

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 communication 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!**

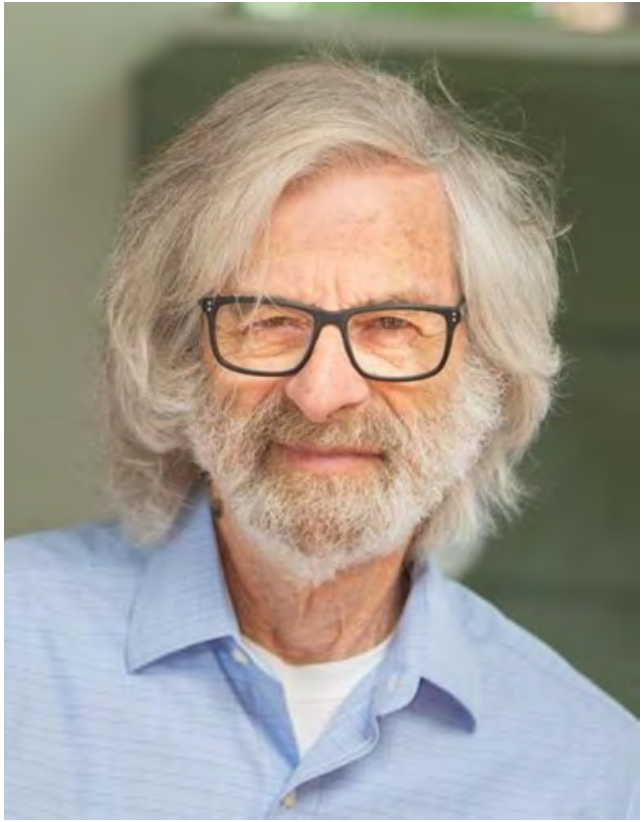
Latency & asynchronous networks

- Asynchronous networks can be a pain
- 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
  - May lead us to assume a service is dead when it isn't
  - May mess with our perception of time
  - May cause messages to arrive in a different order
    … or a different order on different systems

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** (**<u>incoherent</u> views**)

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

Need to ensure the state of the entire system is consistent after a failure

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

Increasing availability through redundancy

Redundancy = replicated components

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

If we need ***all*** systems running to provide a service

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

= 1 - (1-5%) × (1-5%) = 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) …

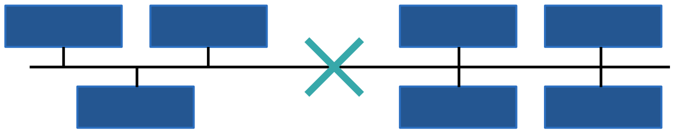
$$P(system\ failure) = \prod_{i}^{n} P_i$$

System Failure Types

- **Fail-stop**
  - Failed component stops functioning
  - **Halting** = stop without notice
  - Detect failed components via **timeouts**
    - But you can't count on timeouts in asynchronous networks
      - And what if the network isn't reliable?
    - Sometimes we guess
- **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

Redundancy requires synchronized state

The **state** of redundant components has to be kept synchronized with the latest version

State, replicas, and caches

- **State**
  - Information about some component that cannot be reconstructed
  - Network connection info, process memory, list of clients with open files, lists of which clients finished their tasks
- **Replicas**
  - Redundant copies of data → *used to address fault tolerance*
- **Cache**
  - Local storage of frequently-accessed data to reduce latency → *used to address latency*

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 foolproof way to detect failure in all cases

Security

Security

- Traditionally managed by operating systems
  - Users authenticate themselves to the system
  - Each user has a unique user ID (UID)
  - Access permissions = f(UID)
- 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

Rely on cryptography (hashes, cryptography) for identity management, authentication, encryption, tamper detection

… and also rely on good defensive programming!

- *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

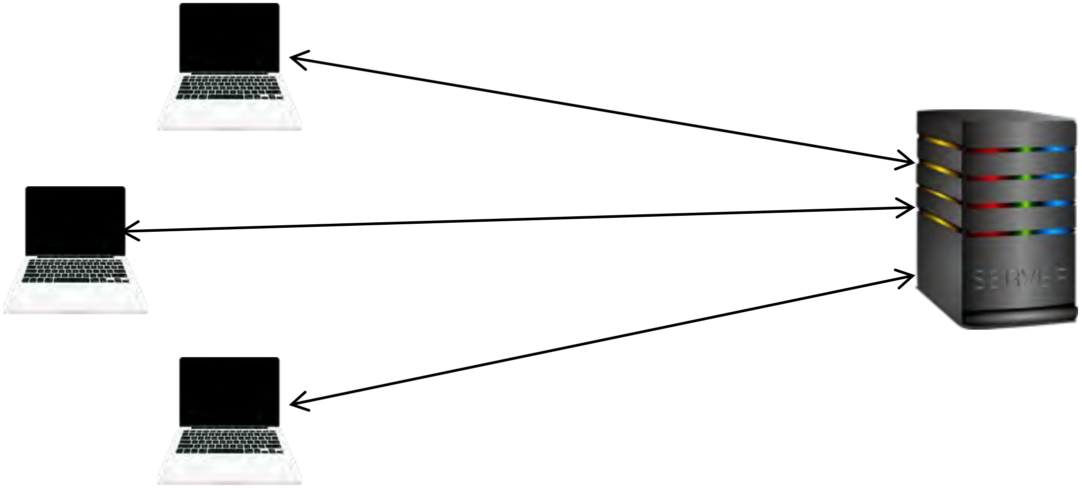
Service Models (Application Architectures)

Centralized model

- No networking
- Traditional time-sharing system
- Single workstation/PC or direct connection of multiple terminals to a computer
- One or several CPUs
- Not easily scalable
- Limiting factor: number of CPUs in system
  - Contention for same resources (memory, network, devices)

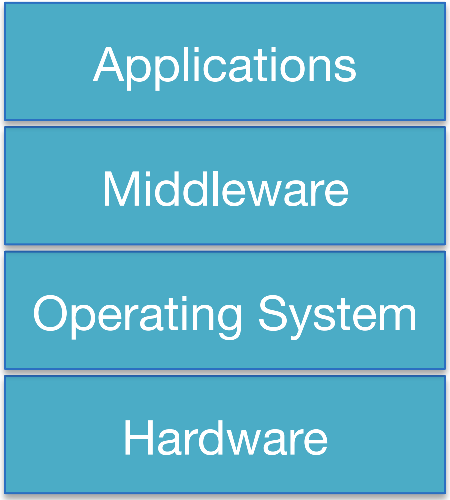
Client-Server model

- 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, …



Includes naming, security, persistence, notifications, agreement, remote procedures, data encoding, …

Includes layering for file systems, networking, devices, memory

Tiered architectures in networked systems

- Tiered (multi-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

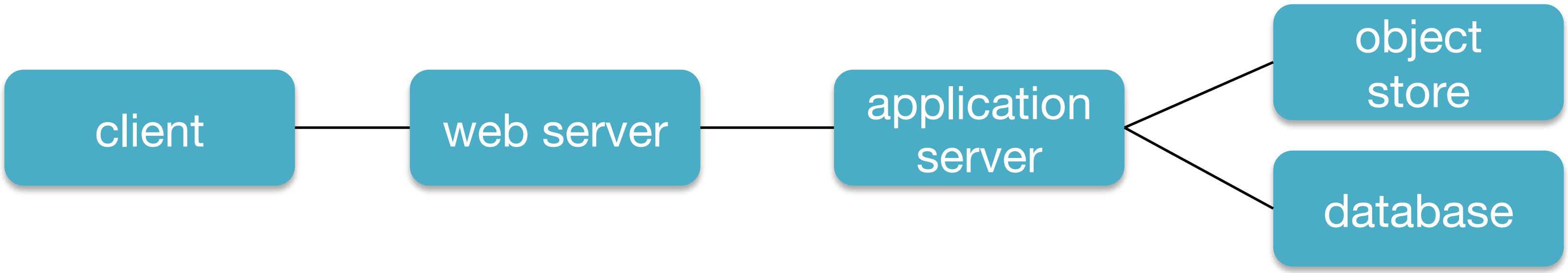
Multi-tier example

User interface
Data presentation
& validation

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

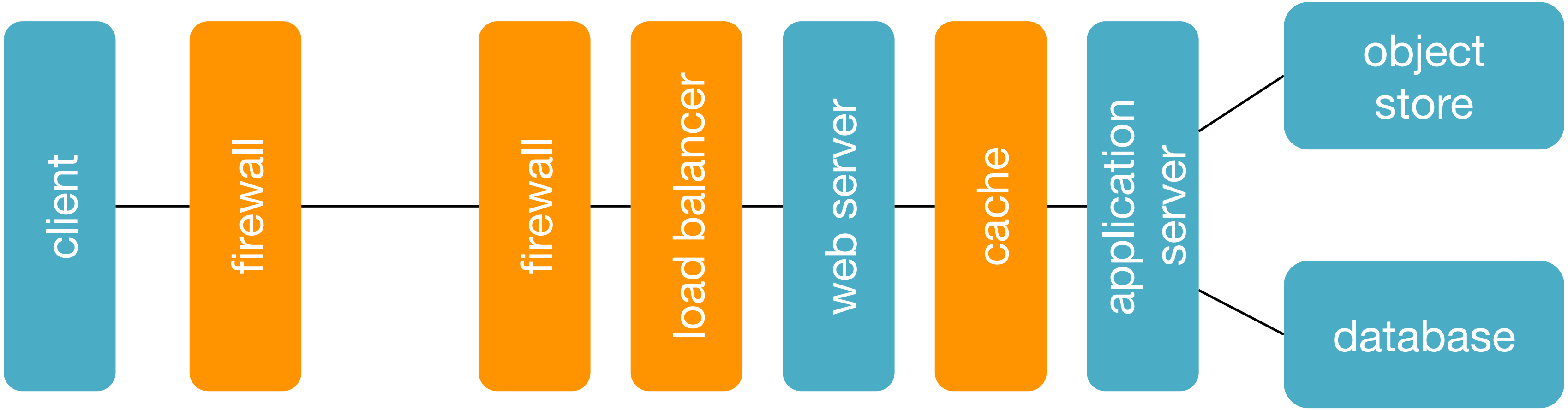
- Database system
- Legacy software

Multi-tier example

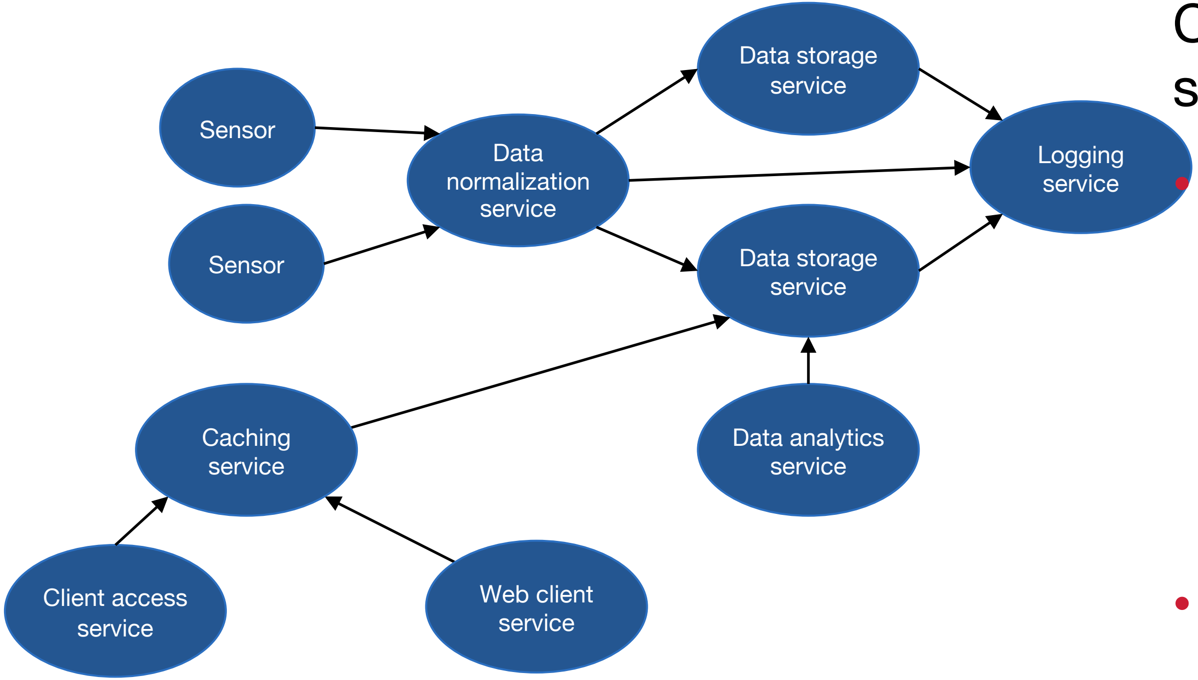


Multi-tier example

Some tiers may be transparent to the application



Microservices Model

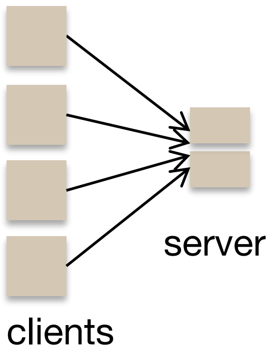


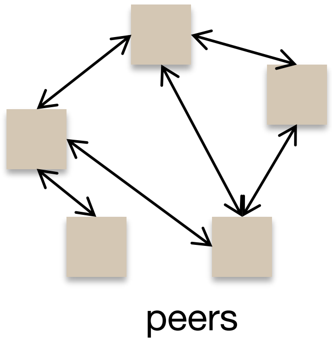
Collection of autonomous services

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

Peer-to-Peer (P2P) Model

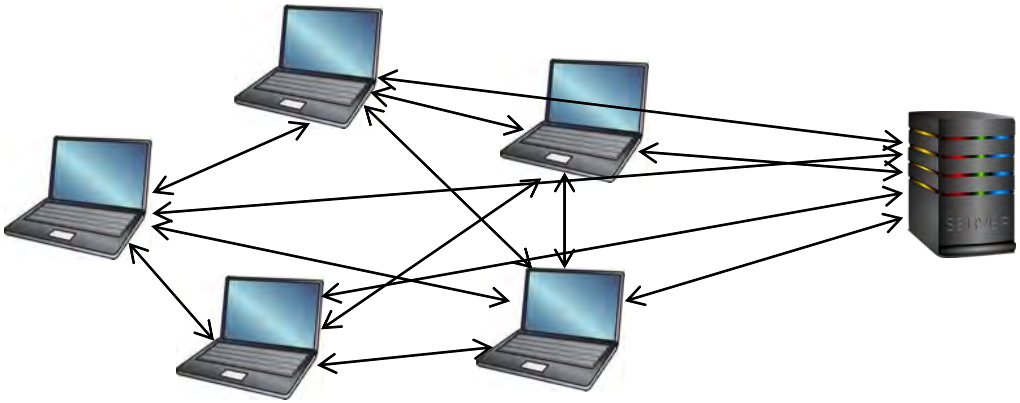
- No reliance on servers
- Machines (peers) communicate with each other
- Goals
  - Robustness
  - Self-scalability
- Examples
  - BitTorrent, Skype





Hybrid model

- Many peer-to-peer architectures still rely on a server
  - Look up, track users
  - Track content
  - Coordinate access
- But traffic-intensive workloads are delegated to peers



Images from: http://clipart-library.com/laptop-cliparts.html

Processor pool model

- 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, …
Salesforce.com, Google Apps, Microsoft 365

Platform as a Service (PaaS)
Execution runtimes, databases, web servers, development environments, …
Google App Engine, AWS Elastic Beanstalk

Infrastructure as a Service (IaaS)
Compute + storage + networking: VMs, storage servers, load balancers
Microsoft Azure, Google Compute Engine, Amazon Web Services

Storage
Remote file storage
- *Dropbox, Box, Google Drive, OneDrive, …*

The End