What is a Distributed System?

A collection of independent, autonomous hosts connected through a communication network.

- No shared memory (must use the network)
- No shared clock
Collection of independent computers that appears as a single system to the user(s)

- Independent = autonomous
- Single system: user not aware of distribution
Classifying parallel and distributed systems
Flynn’s Taxonomy (1966)

Number of instruction streams and number of data streams

**SISD**
- traditional uniprocessor system

**SIMD**
- array (vector) processor
  - Examples:
    - GPUs – Graphical Processing Units for video
    - AVX: Intel's Advanced Vector Extensions
    - GPGPU (General Purpose GPU): AMD/ATI, NVIDIA

**MISD**
- Generally not used and doesn’t make sense
- Sometimes (rarely!) applied to classifying fault-tolerant redundant systems

**MIMD**
- multiple computers, each with:
  - 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 & Multicommputers

- **Multiprocessors**
  - Shared memory
  - Shared clock
  - All-or-nothing failure

- **Multicomputers** (networks of computers)
  - No shared memory
  - No shared clock
  - Partial failures
  - Inter-computer communication mechanism needed: the network
    - Traffic much lower than memory access
Why do we want distributed systems?
1. Scale
2. High availability
3. Fault tolerance
4. Reduced latency
5. Delegation operations
6. Collaboration
7. Mobility
8. Incremental cost
Scale: Increased Performance

• Computers are getting faster

• Moore's Law
  – Prediction by Gordon Moore that the number of transistors in an integrated circuit doubles approximately every two years.
  – Commonly described as performance doubling every 18 months because of faster transistors and more transistors per chip

• Not a real law – just an observation from the 1970s
Moore’s Law – The number of transistors on integrated circuit chips (1971-2016)

Moore’s law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore’s law.

The data visualization is available at OurWorldInData.org. There you find more visualizations and research on this topic. Licensed under CC-BY-SA by the author Max Roser.
• Getting harder for technology to keep up with Moore's law
  – More cores per chip → requires multithreaded programming
  – There are limits
    • Intel Broadwell Xeon CPU: 22 cores per chip (> $4,000/chip!)
    • NVIDIA GeForce GTX1080: 3,584 CUDA cores per chip
      – Special purpose apps: Graphics rendering, neural networks

• We want to build systems that can process billions of request per day
  – Google gets over 40,000 queries per second
  – Facebook:
    > 2 billion searches per day; 13 million database queries per second

• We also want access to exabytes of storage
How can you get massive performance?

• Multiprocessor systems don’t scale high

• Example: movie rendering
  – Toy Story 3: average frame takes 7 hours to render
    • Some can take 39 hours
  – Monsters University: an average of 29 hours per frame
    • 2,000 computers with 12,500 cores
    • Total time: over 100 million CPU hours
    • 3,000 to over 5,000 AMD processors; 10 Gbps and 1 Gbps networks
  – Disney's Frozen
    • 30,000 core renderfarm – 60M render hours – 5 PB storage

• Google
  – Over 40,000 search queries per second on average
  – Index >50 billion web pages
  – Uses hundreds of thousands of servers to do this
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/
High availability

• **Redundancy** = replicated components
  – Service can run even if some systems die

Reminder

\[
P(A \text{ and } B) = P(A) \times P(B) \\
P(A \text{ or } B \text{ or both}) = P(A) + P(B)
\]

• If \( P(\text{any one system down}) = 5\% \)
  \[P(2 \text{ systems down at the same time}) = 5\% \times 5\% = 0.25\%
\]

• BUT if we need all systems up to provide a service
  \[P(\text{any system down}) \approx 100\% !\]
Availability requires fault tolerance

• Fault tolerance
  – Identify & recover from component failures

• Recoverability
  – Software can restart and function
  – May involve restoring state
Reduced Latency

- Replication across geographies
- Cache data close to where it is needed
- Example: Akamai
Delegated operations

- Offload responsibility
  - Let someone else manage systems
  - Use third-party services

- Modularize services on different systems
  - Dedicated systems for storage, email, etc.

- Cloud, network attached storage
Collaboration & Content

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

-logos of popular companies such as Amazon, Spotify, Pandora, YouTube, Netflix, iTunes, DIRECTV NOW, Amazon Video, Hulu, and HBO NOW
Metcalf's Law

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

This makes networking interesting to us!
Mobility

• Over 2.3 billion smartphone users
• Remote sensors
  – Cars
  – Traffic cameras
  – Toll collection
  – Shipping containers
  – Soda machines
• IoT = Internet of Things
Incremental cost

• Scale also implies cost
• Facebook
  – Started on one rented server at $85/month
• Google
  – Original storage in 1996: 10 4GB drives = 40 GB total
  – 1998 hardware
    • Sun Ultra II, 2 Intel dual-Pentium II servers, quad-processor IBM RS/6000
    • ~ 475 GB of disks
Design Considerations in Distributed Systems
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
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 all-or-nothing
• In distributed systems, we get partial failure
  – Send a request but don't get a response
  – What happened?

• Need to deal with detection, recovery, and restart
  – Availability = fraction of time system is usable
    • Achieve with redundancy
    • But consistency is an issue!
  – Reliability: data must not get lost
    • Includes security
Failure types

• **Fail-stop**
  – Failed component stops functioning
    • Ideally, it may notify other components first
  – **Halting** = stop without notice
  – Detect via timeouts
    • But timeouts aren't reliable if network latency is variable or if the network isn't reliable
    • Sometimes we guess

• **Fail-restart**
  – Component stops but then restarts
  – Danger: stale state
Failure types

- **Omission**
  - Failure to send or receive messages
    - 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
    - Mutual exclusion, timestamped log entries

- **Partition**
  - Network fragments into two or more sub-networks that cannot communicate with each other
Failure types

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

• Goal: avoid single points of failure
No global knowledge

- Nobody has the true global state of a system
  - 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
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

• Cache
  – Local storage of frequently-accessed data to reduce latency
Handling Scale

• 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
Security

• The environment
  – Public networks, remotely-managed services, 3rd party services

• Some issues
  – Malicious interference, bad user input, impersonation of users & services
  – Protocol attacks, input validation attacks, time-based attacks, replay attacks

• Rely on authentication & encryption
  … and good programming!

• Users also want convenience
  – Single sign-on
  – Controlled access to services
Other design considerations

• Algorithms & environment
  – Distributable vs. centralized algorithms
  – Programming languages
  – APIs and frameworks
Main themes in distributed systems

• **Scalability**
  – Things are easy on a small scale
  – But on a large scale
    • Geographic latency (multiple data centers), administration, dealing with many thousands of systems

• **Latency & asynchronous processes**
  – Processes run asynchronously: concurrency
  – Some messages may take longer to arrive than others

• **Availability & fault tolerance**
  – Fraction of time that the system is functioning
  – Dead systems, dead processes, dead communication links, lost messages

• **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
- The server is always on and processes requests from clients
- Clients do not communicate with other clients
- Examples
  - FTP, web, email
Layered architectures

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

![Layered Architecture Diagram]

- Applications
- Middleware
  - Includes naming, security, persistence, notifications, agreement, remote procedures, data encoding, …
- Operating System
  - Includes layering for file systems, networking, devices, memory
- Hardware
Tiered architectures

• **Tiered** (multi-tier) architectures
  – distributed systems analogy to a layered architecture

• Each tier (layer)
  – Runs as a network service
  – Is accessed by surrounding layers

• The “classic” client-server architecture is a two-tier model
  – Clients: typically responsible for user interaction
  – Servers: responsible for back-end services (data access, printing, …)
Multi-tier example

- **Client**
  - User interface
  - Data presentation & validation

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

- **Back End**
  - Database system
  - Legacy software
Multi-tier example
Multi-tier example

Some tiers may be transparent to the application
Peer-to-Peer (P2P) Model

• No reliance on servers

• Machines (peers) communicate with each other

• Goals
  – Robustness
    • Expect that some systems may be down
  – Self-scalability: the system can handle greater workloads as more peers are added

• 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
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
  - Salesforce.com, Google Apps, Microsoft Office 365

- Infrastructure as a Service (IaaS)
  Compute + storage + networking
  - Microsoft Azure, Google Compute Engine, Amazon Web Services

- Platform as a Service (PaaS)
  Deploy & run web applications without setting up the infrastructure
  - Google App Engine, AWS Elastic Beanstalk

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