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
- No shared operating system (almost always)

What is a Distributed System?

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

Data normalization service
Data storage service
Data analytics service
Logging service

Single System Image

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 Advanced Vector Extensions
  - GPU (General Purpose GPU, also used for HPC)

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 & Multicomputers

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

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
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 CPU: 28 cores per chip ($2,999/chip!)
  • 8 billion transistors, 255 w @ 3.1-4.3 GHz
  – NVIDIA GeForce RTX 2080 Ti: 4,352 CUDA cores per chip
  • Special purpose apps: Graphics rendering, neural networks

More performance

What if we need more performance than a single CPU?
• Combine them ⇒ multiprocessors
• But there are limits and the cost goes up quickly

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
– Pixar render farm – 2,000 systems with 24,000 cores
– Toy Story 4 (2019) – 60-160 hours to render a frame
– Disney/Pixar's Coco (2017) – Up to 100 hours to render one frame
– How to Train a Dragon (2010) – 90 million CPU hours to render
– Big Hero 6 (2014) – average 83 hours/frame; 199 million CPU core hours
– Monsters University (2013) – an average of 29 hours per frame
  – 2,000 computers with 12,500 cores – total time: over 100 million CPU hours
• 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/

Collaboration & Content

• Collaborative work & play
• Social connectivity
• Commerce
• News & media
  – Disney
  – Amazon
  – Netflix
  – Apple TV+
  – Spotify
  – Pandora
  – YouTube
  – Hulu
  – HBO NOW

2. Collaboration
Metcalfe'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!

3. 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, Dropbox

4. Mobility

3.5 billion smartphone users

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

IoT = Internet of Things
- 2017: more IoT devices than humans

5. High availability & Fault tolerance
High availability

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

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

If \( P(\text{any one system down}) = 5\% \)
\[ P(\text{two systems down at the same time}) = 5\% \times 5\% = 0.25\% \]
Uptime = 1 – downtime = 1 – 0.0025 = 99.75%

February 11, 2020 © 2014-2020 Paul Krzyzanowski 26

Availability requires fault tolerance

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

February 11, 2020 © 2014-2020 Paul Krzyzanowski 28

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

• eBay
  – Perl code on one hosted FreeBSD server – flat files or Berkeley DB
• 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

February 11, 2020 © 2014-2020 Paul Krzyzanowski 30
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.
- Cloud, network attached storage

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

Why are distributed systems different … and challenging?

Core issues in distributed systems design

1. Concurrency
2. Latency
3. Partial Failure
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
- Replication adds complexity
  - All operations must appear to occur in the same order on all replicas

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, $T$
- Asynchronous network model
  - Messages can take arbitrarily long to reach a peer node
  - This is what we get from the Internet!

Latency

- 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
  - Underlying data can change → cache needs to be invalidated
  - System using the cache may change the data → propagate results
  - Write-through cache
    - Sit updates take time → can lead to inconsistencies (incoherent views)
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

Need to deal with detection, recovery, and restart

Availability = fraction of time system is usable
  - Achieve with redundancy
  - But then consistency is an issue!
Reliability: data must not get lost
  - Includes security

System Failure Types

- Fail-stop
  - Failed component stops functioning
  - Ideally, it may notify other components first
  - 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: 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
Network & System Failure Types

• **Fail-stop (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 – fault tolerance**
  - Failures are inevitable – try to design systems that can handle them
  - Avoid single points of failure
  - A single failed component causes the entire system to not work

---

Redundancy

• We deal with failures by adding redundancy
  - Replicated components

• But this means we need to keep the state of those components replicated

---

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 → address fault tolerance

• **Cache**
  - Local storage of frequently-accessed data to reduce latency → 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**

---

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

---

Other design considerations
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, cryptography (hashes, 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

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

• 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

• 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, ...

**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 basic 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

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
  - 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: email, productivity, games, …
    - Salesforce.com, Google Apps, Microsoft Office 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