File Systems

2024 Semester 2 COMPSCI 340: Operating Systems
Talia Xu

Lecture 1
1.0.0

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License

What can happen on a crash?

Assume we are creating and writing to a new file /foo/bar
Write in general (not just the first write)

Operation	data bitmap	inode Bitmap	root inode	foo's inode	bar's inode	root data	foo data	bar data[0]	bar data[1]	Step #
create(bar)			read							1
						read				2
				read						3
							read			4
		read								5
		write								6
							write			7
					read					8
					write					9
				write						10
write()					read					11
	read									12
	write									13
									write	14
					write					15

What can happen on a crash?

Assume we are creating and writing to a new file /foo/bar

A crash after the completion of:

- Step 1-5: No corruption.
- Step 6: i-node bitmap points to garbage data.
- Step 7-9: Inconsistent metadata.
- Step 10-12: No corruption.
- Step 13: data bitmap points to garbage data.
- Step 14: Corrupted data.
- Step 15: The new data cannot be found.

What can happen on a crash?

Lost Files: New files being created might not be fully registered in the directory structure, leading to them being ”lost” even if their data exists on disk.

Inaccessible Files: Existing files might become inaccessible due to corrupted metadata or lost pointers to their data blocks.

Incorrect File Sizes: The file system might report incorrect file sizes due to incomplete writes or metadata updates.

Inconsistent Metadata: Applications trying to access affected files might crash, freeze, or produce errors due to the inconsistencies.

Recovery from failure

Consistency checking compares data in directory structure with data blocks on disk, and tries to fix inconsistencies.

- Tools: fsck for Unix systems, chkdsk for Windows.
- Scanning the entire disk, slow, very slow ...

Journaling: A system that keeps track of changes that will be made in the file system to ensure data integrity.

Versioning and Snapshots: Maintain historical versions or states of files and systems to roll back when needed.

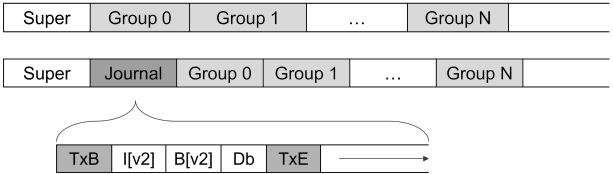
Backup and Restore: Periodic backups to external devices or cloud storage, ensuring data can be restored from a known good state.

Journaling (a.k.a Write-Ahead Logging)

Used by many modern file systems:

- Linux (ext3, ext4), Windows (NTFS), IBM (JFS)

First write down what you are going to do, and then carry out the action



Journaling (a.k.a Write-Ahead Logging)

Physical journal

- all changed data is recorded in the journal first

Logical journal

- only changed metadata (like inode tables, directory structures) is recorded

Journaling (a.k.a Write-Ahead Logging)

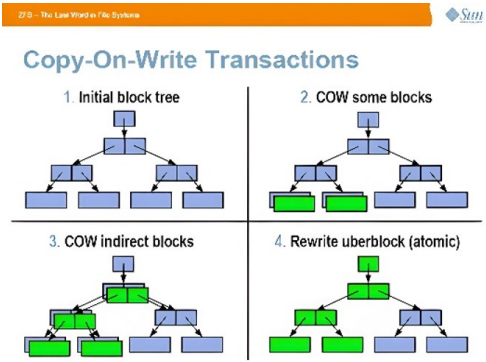
What if absolute data integrity is crucial?

- for example, in banking, aviation systems, stock exchange?

How do we ensure that data integrity is always maintained?

ZFS

What happens if system crashes before each step can be completed?



Why distributed systems?

What if we need more performance than a single CPU?

Vertical Scaling: use more powerful systems / multiprocessors

- But these have scaling limits and cost $

Horizontal Scaling: distributed load across more systems

- Distributed systems allow us to achieve massive performance

Why distributed systems?

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.

Why distributed system? An example

2004: Facebook started on a single server

- Web server front end to assemble each user's page.
- Database to store posts, friend lists, etc.

2008: 100M users

2010: 500M users

2012: 1B users

2019: 2.5B users

An example scenario

For a web-based service to be available, both the server and the web server need to be operational.

In this scenario, both the server and the web server are hosted on a single machine.

This machine has a 10% chance of failure.

What is the overall availability of the service, assuming no other components fail?

An example scenario

Now the server and the web server are each hosted on a different machine.

Each machine has a 10% chance of failure.

What is the overall availability of the service, assuming no other components fail?

An example scenario

As the service grows, the user base expands, and more content is required.

To support this, 10 machines are dedicated to the servers, and another 10 are allocated for the web servers.

Each machine has a 10% chance of failure.

What is the overall availability of the service, assuming no other components fail?

An example scenario

The service provider aims to reduce their failure rate to below 1% by creating replicas of their servers and web servers.

How many replicas are needed?

Why distributed system?

Nature of the application

- Multiplayer games
- Collaborative Projects & Cloud App

Availability despite unreliable components

- A service shouldn't fail when one computer does. •

Conquer geographic separation

- A web request in NZ is faster served by a server in NZ than by a server in US.

Scale up capacity

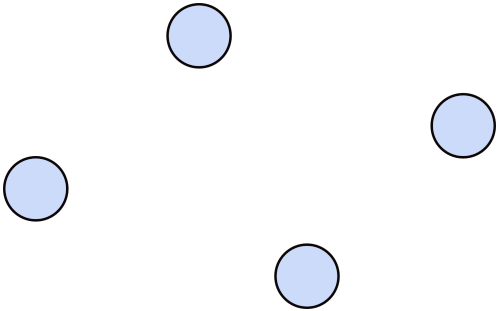
- More CPU cycles, more memory, more storage

What is a distributed system?

“A collection of loosely coupled nodes interconnected by a communication network.”

— textbook

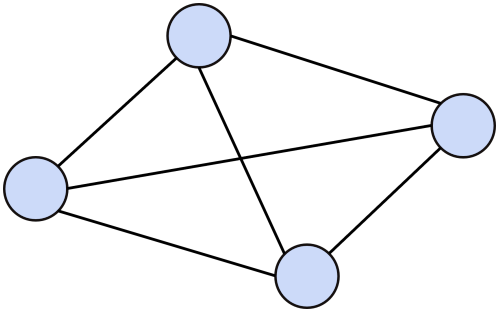
What is a distributed system?



Independent components or elements

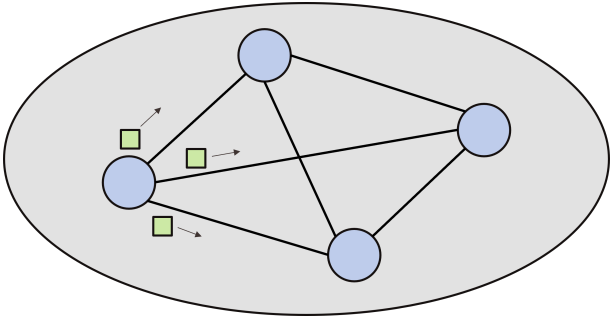
- software processes or hardware used to run a process, store data, etc.

What is a distributed system?



Independent components or elements that are **connected by a network.**

What is a distributed system?



Independent components or elements that are **connected by a network** and **communicate by passing messages** to achieve a common goal, appearing as **a single coherent system**.

When one (or more) component fails, the system does not fail.

Core challenges in distributed systems design

Cuncurrency

Latency

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.

Concurrency

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

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.

Latency

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

Latency

Asynchronous network model

- Messages can take arbitrarily long to reach a peer node.
- This is what we get from the Internet!

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 cause messages to arrive in a different order.

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.

Partial 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.

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 is how long the system can run without failing.

- Includes ensuring data does not get lost.
- includes security.

If a system recovers quickly from filure, is it highly available? is it highly reliable?

Increasing availability through redundancy

Redundancy comes from repliated components.

- Service can run even if some systems die.

If the probability that any one system down is 5%, what is the probability that two systems are down at the same time?

What is the uptime in both cases?

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

User also wants convenience.

- Single sign-on, no repeated entering of login credentials

Other considerations

Scaling up and 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 and environment

- Distributable vs. centralized algorithms
- Programming languages
- APIs and frameworks

Service Models - Centralized

No networking

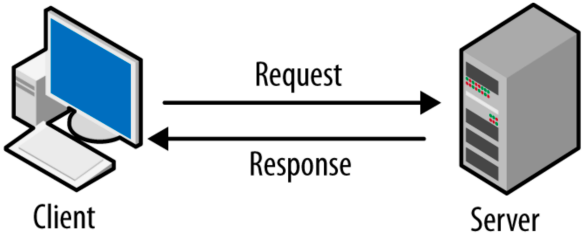
Traditional time-sharing system

Single workstation or PC or direct connection of multiple terminals to a computer.

One or several CPUs.

Not easily scalable.

Service Models - The client-server model

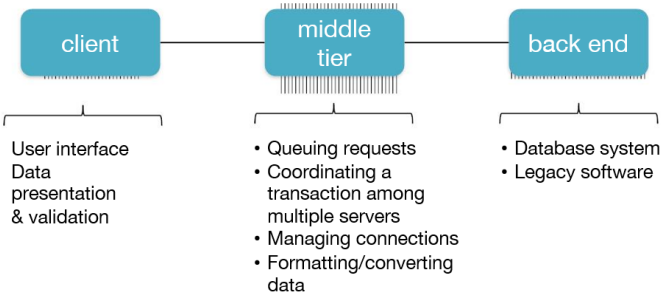


Clients send requests to servers

A server is a system that runs a service

Clients do not communicate with each other.

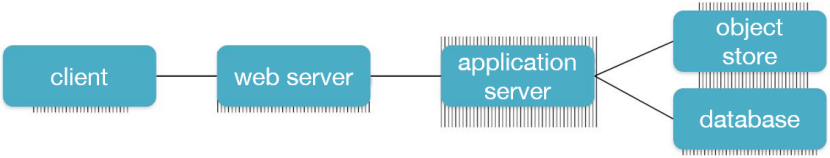
Multi-tier architecture in networked systems



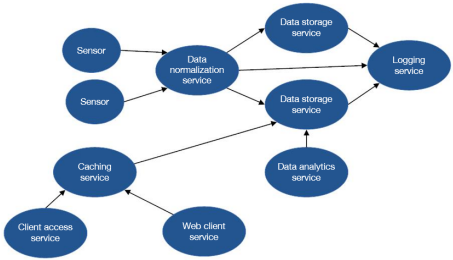
Each tier

- runs as a network service
- is accessed by surrounding tiers

Multi-tier example



Service Models - 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

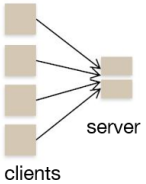
Service Models - peer-to-peer model (P2P)

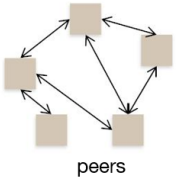
No reliance on servers

Machines (peers) communicate with each other

Goals: Robustness and self-scalability

Examples: BitTorrent, Skype



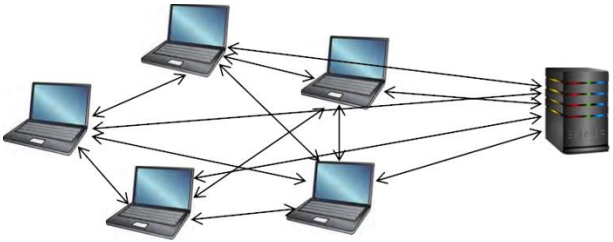


Service Models - hybrid model

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



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)

- Computer + storage + networking: VMs, storage servers, load balancers
- Microsoft Azure, Google Compute Engine, Amazon Web Services

Storage

- remote file storage
- Dropbox, Box, Google Drive, OneDrive, …