CSC 36000:
Modern Distributed Computing *with AI Agents*

By Saptarashmi Bandyopadhyay
Email: sbandyopadhyay@ccny.cuny.edu
Assistant Professor of Computer Science
City College of New York and Graduate Center at City University of New York

November 10, 2025 CSC 36000

Today's Lecture

Scalability vs. Reliability for Distributed Computing

Quantifying Reliability for Distributed Computing

Reliability of Distributed Architectures

Scalability

What is Scalability?

Scalability refers to a system's ability to handle a growing amount of work by adding resources.

Example: Let's say you're preparing a dinner for 10 friends that has 4 dishes and it needs to be ready in 2 hours. If you try to do this by yourself, this is impossible.

Solution: Have your friends help!

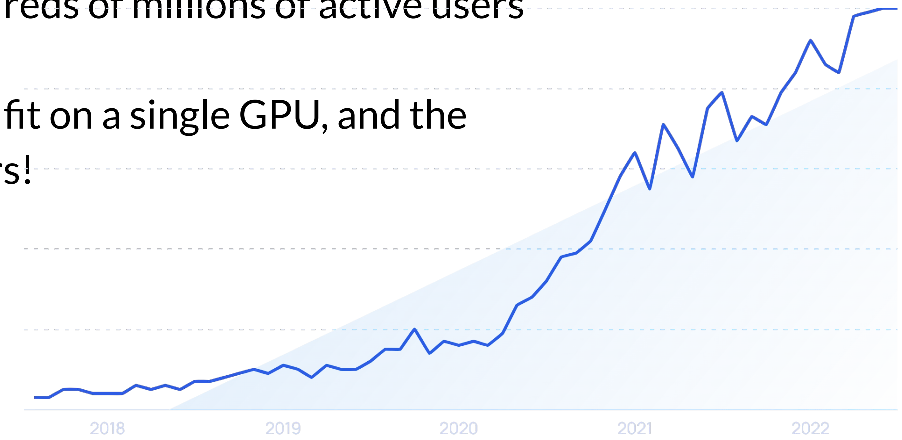
In this case, each person could be thought of as an inference node, distributing the total workload (the dishes) so that no one node is overburdened.

Why would we need to scale AI?

Data: AI training needs *huge* amounts of data and inference likewise needs to handle a constant stream of high-volume multimodal data

User Growth: An AI service could have hundreds of millions of active users

Model Size: AI models are often too large to fit on a single GPU, and the largest AI models have trillions of parameters!



How do we scale AI?

Data Parallelism:

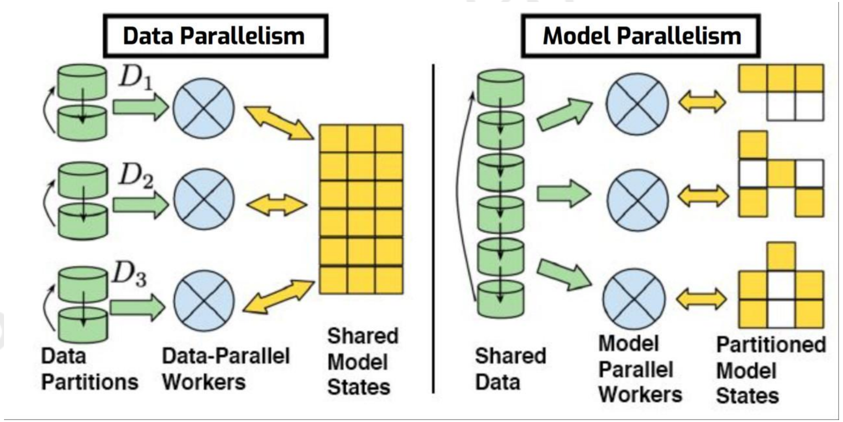
- Same model, different data
- E.g. Ring All-Reduce

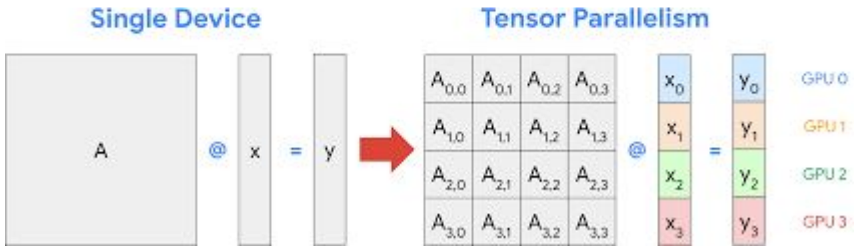
Model Parallelism:

- Same data, partitioned model

Tensor Parallelism:

- Split individual matrix operations (e.g. matrix multiplication)
- Requires intense communication!





Revisiting Reliability

Reliability is the ability to consistently maintain system operations as intended, *even when things go wrong.*

This idea of reliability mixes three concepts we've covered in class:

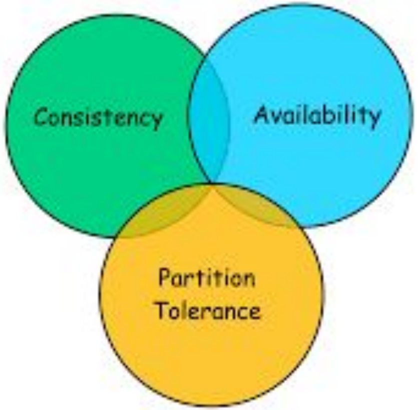
- **Fault Tolerance**: Can it survive a failure? (e.g., Byzantine Fault Tolerance)
- **Availability:** How often is it working? (e.g., 99.99% uptime)
- **Consistency:** Does everyone see the same, correct data?

Reliability’s Internal Trade-off

In a distributed system, we should always have Fault Tolerance , also known as Partition Tolerance (P)

But there’s a conflict between how *available* (A) and how *consistent* (C) we can be!

To start, how can we put a number to reliability?



Quantifying Reliability

Example: AI Startup

Let’s say you create an AI startup where you provide an AI Agent service to your clients.

You promise your clients 99.99% uptime (“Four Nines”)

Question: How much *downtime* does this result in per year?

Example: AI Startup

Solution:

- Total minutes in a year: 365 * 24 * 60 = 525,600
- Allowed downtime: 100% - 99.99% = 0.01% = 0.0001
- Total downtime: 525,600 * 0.0001
- **Answer:** 52.56 minutes each year

Example: AI Startup

It would be very a difficult to ask a single server to stay up for all but one hour of the year!

What if we had *multiple* inference servers and “downtime” only occurs when all of them fail?

Question: How many total servers (in an independent, redundant setup) do you need to achieve "Four Nines" (99.99%) availability if each server has 99% availability?

Example: AI Startup

Solution:

- Recall: In this system, failure only happens when *all* servers fail. For simplicity we assume the failures are independent.
- P(Fail) = 1 - Availability = 1 - 0.99 = 0.01
- For N servers, P(System Fail) = P(Fail) * P(Fail) * … * P(Fail) (N times) = P(Fail) ^ N because we assume independence
- So then 0.01 ^ N = 0.0001 (we want "Four Nines" again)
- **Answer:** N = 2! We need two servers.

Recall:

Independent Events

$$P(A \text{ and } B) = P(A) \cdot P(B)$$

The Problem we Created

We used two nodes (Node A and Node B) to increase the availability of our system. Awesome!

…But what happens in this scenario?

1. A user request comes to **Node A** (active)
2. Node A processes the request (e.g. withdraw $10 -> user's bank balance = $90)
3. Node A **fails** before it can pass this state on to **Node B** (passive)
4. Node B takes over. It's state is *stale* (balance = $100)

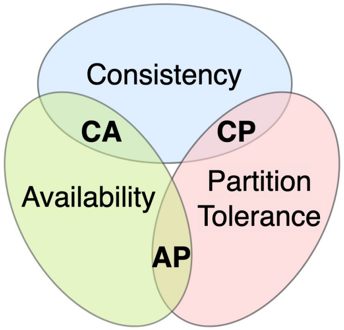
We are *available* but no longer *reliable*! Node B has the *wrong data*.

The CAP Theorem

Consistency, Availability, Partition Tolerance (CAP)

This brings us to the most important law in distributed computing:

- **C = Consistency**
  - All nodes see the same, most recent data, or you get an error.
- **A = Availability**
  - The system always gives a response (even if the data is stale).
- **P = Partition Tolerance**
  - The system keeps working even if the network fails between nodes.



Distributed Systems exist in a balance between these three key points. In the real world, networks *will* fail, so P is essential. The question we need to ask is: Do we sacrifice C or A?

Example Scenario

Let's explore a situation where the answer might reveal itself to us.

Setup: A video has 100 likes. The database is replicated in two data centers.

- Node N1 (New York): likes = 100
- Node N2 (London): likes = 100

Assume a healthy network link connects them.

Example Scenario: Network Failure

NETWORK PARTITION! The link is cut.

- A user in NY likes the video.
- Node N1 (New York): likes = 101
- Node N2 (London): likes = 100 (N1's update message is lost)

The Dilemma:
A user in London connects to Node N2 and tries to read the like count.
Node N2 knows it is partitioned. It doesn't know if its data (100) is stale.

What should Node N2 do?

Example Scenario: Choose Consistency

If we choose consistency:

- **Action:** N2 cannot guarantee its data is the most recent.
- **Result:** N2 returns an ERROR or TIMEOUT.

Analysis:

- **Pro:** Data integrity is protected. No user ever sees stale data.
- **Con:** The system is unavailable. The user is angry.

This is a system that prioritizes *consistency* above *availability*.

Example Scenario: Choose Availability

If we choose availability:

- **Action:** N2 returns the best (possibly stale) data it has.
- **Result:** N2 returns "100 Likes".

Analysis:

- **Pro:** The system is 100% available. The user is happy.
- **Con:** The data is inconsistent (stale).

This is a system that prioritizes *availability* above *consistency*. ***What do you think is the right choice to make for this scenario?***

Example Scenario: Resolution

What happens *after* the partition?

1. The network link is repaired.
2. N1 (NY) sends its update (101) to N2 (London).
3. N2 (London) updates its state: likes = 101.

We say this system has *converged*. Here, our system demonstrates *Eventual Consistency* because after the faults are repaired, it eventually becomes consistent.

Reliability of Distributed Architectures

Recall: Parameter Server (PS) vs. Ring All-Reduce (AR)

In a previous lecture, we learned about two architectures for distributed training:

- **Parameter Server (PS):** Workers push gradients to a central Server.
- **Ring All-Reduce (AR):** Workers communicate in a logical ring.

Let's re-evaluate them based on Scalability vs. Reliability.

Recall: Parameter Server (PS) vs. Ring All-Reduce (AR)

Architecture	Scalability (Performance)	Reliability (Fault Tolerance)
Parameter Server	**Poor:** Central server is a network bottleneck.	**Mixed:** • **High** tolerance to *worker* failure/stragglers (async mode). • **Low** tolerance to *server* failure (Single Point of Failure).
Ring All-Reduce	**Excellent:** Decentralized & bandwidth-optimal.	**Poor:** • **Zero** tolerance to *worker* failure (breaks the ring). • **Vulnerable** to *stragglers* (slowest node is bottleneck).

PS vs. AR Trade-off

Architecture	Scalability (Performance)	Reliability (Fault Tolerance)
Parameter Server	**Poor:** Central server is a network bottleneck.	**Mixed:** • **High** tolerance to *worker* failure/stragglers (async mode). • **Low** tolerance to *server* failure (Single Point of Failure).
Ring All-Reduce	**Excellent:** Decentralized & bandwidth-optimal.	**Poor:** • **Zero** tolerance to *worker* failure (breaks the ring). • **Vulnerable** to *stragglers* (slowest node is bottleneck).

Parameter Server: Trades Scalability for Worker Reliability (it can handle slow/dead workers).

Ring All-Reduce: Trades Reliability for Scalability (it's much faster, but brittle: one failure halts everything).

Neither is "better." They are different design choices for different problems.

Questions?