# 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

# Today's Lecture

**Data Distribution Algorithms**

- Anti-Entropy
- Gossip Protocol
- Coding Demonstration

**MapReduce**

- Primitive Operations: Map, Shuffle, Reduce
- Coding Demonstration

# Data in Distributed Computing

# Propagating Data in Distributed Systems

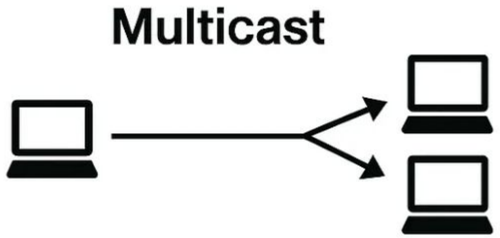
How do we reliably update data across 10,000+ nodes in a decentralized network?

*Traditional Reliable Multicast* relies on acknowledgments (**ACK**s) or negative acknowledgments (**NAK**s).

As the number of receivers **N** grows, the sender becomes swamped with feedback messages.

Centralized directories (like basic DNS) create bottlenecks and single points of failure.

We need a mechanism that relies only on *local* information yet achieves *global* convergence!



# Data Distribution as an Infectious Disease

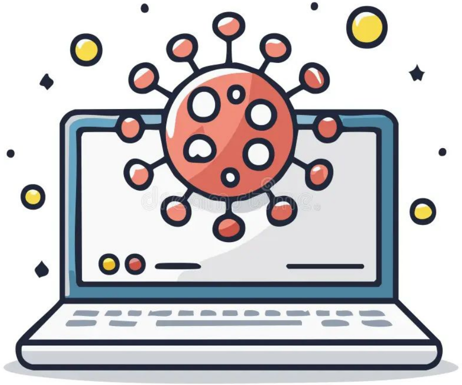
**Epidemic Theory:**

- Modeled after the spread of infectious diseases.
- Instead of a cold, we are spreading updates or information.

**Goal:** "Infect" 100% of the population with the update as fast as possible.

**Key Terminology:**

- **Infected:** A node that holds the update and is willing to share it.
- **Susceptible:** A node that has not yet seen the update.
- **Removed**: A node that has the update but is no longer sharing it (lazy/inactive).



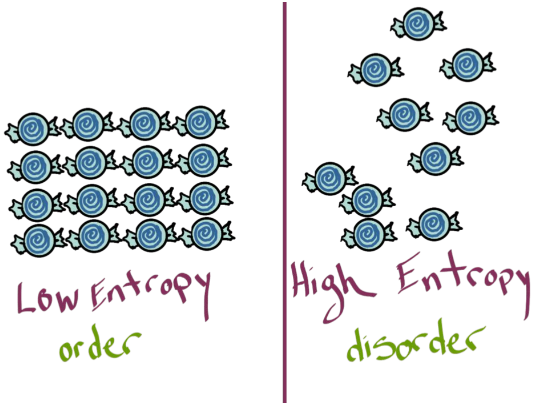
# Anti-Entropy

Nodes regularly choose a random neighbor to exchange state with.

**Goal:** Resolving inconsistencies (entropy) between nodes.

Three Approaches:

- **Push: P** sends updates to **Q**. (Efficient when few are infected).
- **Pull: P** asks **Q** for updates. (Efficient when many are infected).
- **Push-Pull:** Both exchange updates. (Optimal convergence: O(log N) rounds).




Anti-Entropy Guarantees eventual consistency but can be bandwidth intensive (sending large state differences).

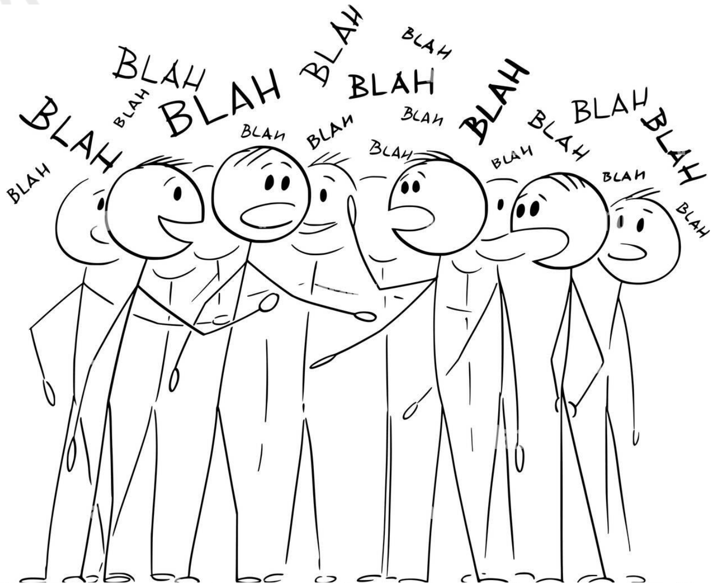
# Gossip Method Protocol

If a node receives a new update, it tries to push it to a random neighbor.

If the neighbor already knows the update, the sender loses interest. The sender stops spreading the rumor with probability **1 / k**.

**Pros:** Extremely rapid distribution; low bandwidth when the system is dormant.

**Cons:** Probabilistic. There is a small chance some nodes remain Susceptible (never receive the update) if the "gossip dies out" too quickly



# Deletion in Distributed Computing

Why is deleting hard?

- In a stateless system, a "deletion" looks like a missing file.
- If Node **A** deletes Item **X**, and later syncs with Node **B** (who still has **X**), Node B might "update" Node **A** by sending **X** back.

**The Solution:** Death Certificates

- Don't delete the item immediately.
- Replace the item with a Death Certificate: A timestamped record stating "Item **X** is deleted."
- Spread the certificate via gossip like any other update.

Certificates are eventually cleaned up after a maximum propagation time.

## Hands-on Gossip Deletion Example

https://drive.google.com/file/d/14piiD1Pem_o5es4b9Pc7QUSI40iWmLa5/view?usp=sharing

# MapReduce

# Working with Big Data

In modern Distributed Computing and AI, we're often working with internet-scale datasets. How do we process so much data?

**Traditional Approach (Serial Processing):**

- Store data on one massive storage server (SAN/NAS).
- Pull data across the network to a supercomputer.
- Process line-by-line.

**The Bottlenecks:**

- Network Bandwidth: Moving 10PB takes forever.
- Single Point of Failure: If the processor crashes at 99%, you start over.
- Disk I/O: Reading linearly is slow.

# Divide and Conquer

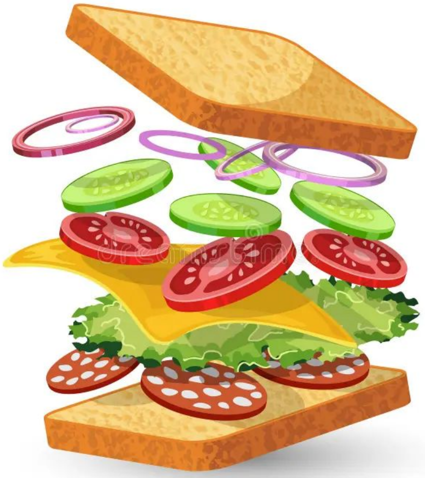
Google addressed this problem back in 2004:

- Don't buy one supercomputer. *Buy 1,000 cheap commodity servers.*
- Data Locality: Don't move data to the CPU. *Move the code to the data.*

**The Abstraction:**

- Hide the messy details (network failures, disk crashes, load balancing).
- Expose two simple functions to the programmer: Map and Reduce.

# Analogy: Sandwich Shop



**Goal:** Make 1,000 Sandwiches as fast as possible.

**Phase 1: MAP (The Prep Cooks)**

- 5 cooks working in parallel.
- They don't make full sandwiches. They just output intermediate parts.
- Output: Piles of "Sliced Tomato," "Chopped Lettuce," "Cooked Bacon."

**Phase 2: SHUFFLE (The Runners)**

- The messy part. Runners grab all tomatoes from all 5 cooks and put them in the "Tomato Bin."
- Grouping: Ensure all instances of the same ingredient end up in the same place.

**Phase 3: REDUCE (The Assemblers)**

- One chef stands at the Tomato bin, one at the Bacon bin.
- They process the piles into the final result.

# The Technical Approach

**Input:** Split into chunks (usually 64MB or 128MB).

**MAP: (Key, Value) -> List(Key, Value)**

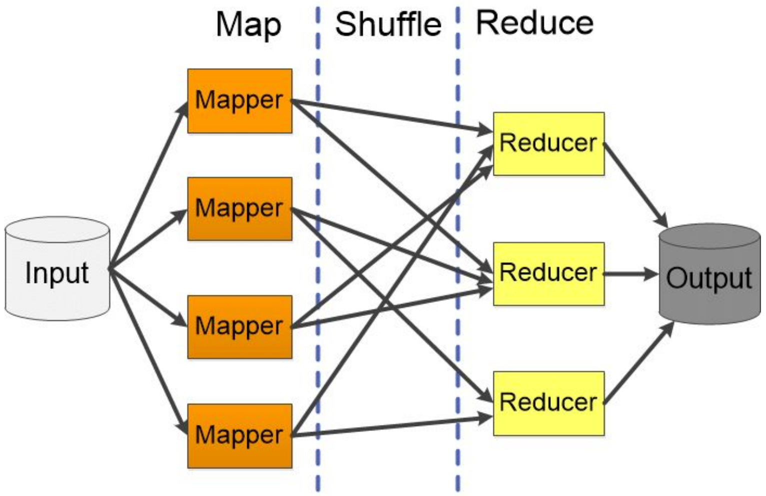
- Takes raw data, filters/parses it, outputs intermediate pairs.
- Example: (Doc1, "Hello World") -> [("Hello", 1), ("World", 1)]

**SHUFFLE & SORT: (Key, Value) -> (Key, List[Values])**

- The system groups all values associated with the same key.
- Example: ("Hello", [1, 1, 1, 1])

**REDUCE: (Key, List[Values]) -> (Key, Result)**

- Aggregates the list into a single output.
- Example: ("Hello", 4)



# Example: Word Count

**Input Data:** "Deer Bear River" "Car Car River" "Deer Car Bear"

**Map Phase (Parallel):**

- Worker 1: (Deer, 1), (Bear, 1), (River, 1)
- Worker 2: (Car, 1), (Car, 1), (River, 1)
- Worker 3: (Deer, 1), (Car, 1), (Bear, 1)

**Shuffle Phase (Network Intensive):** Moves data so keys group together.

- Bear: [1, 1]
- Car: [1, 1, 1]
- Deer: [1, 1]
- River: [1, 1]

**Reduce Phase:** Sum the lists: Bear: 2, Car: 3, Deer: 2, River: 2

# Overhead

**Synchronization Barrier**

- The Reduce phase cannot start until the slowest Map worker is finished.
- Analogy: You can't count total "Tomatoes"  until the last prep cook finishes chopping.

**Network Congestion:**

- The "Shuffle" phase requires moving massive amounts of data between servers simultaneously.

**Fault Tolerance:**

- What if Worker 5 dies? The Master node must detect it and re-assign that chunk of data to Worker 2.

## Hands-On MapReduce Demonstration

https://drive.google.com/file/d/1PpZDwDgH_0-R4ArOOFlUdybn82GQUm2_/view?usp=sharing

# Questions?