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

October 6, 2025 CSC 36000

Today's Lecture

Timing in Distributed Systems

Coordination with Mutual Exclusion

Distributed Computing with JAX

Timing in Distributed Systems

Recall: Asynchronous Systems

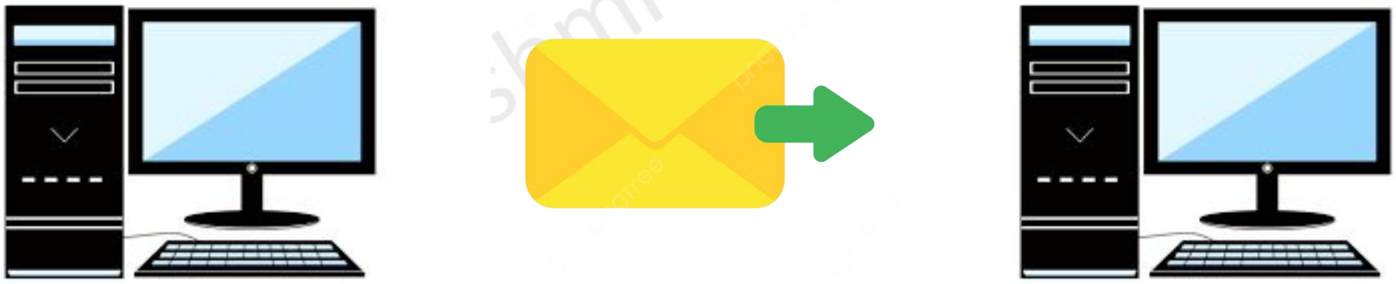
In the vast majority of real-world distributed systems, we don't have access to a **global** clock, only a system-level clock for each process.

Sometimes the time it takes for a message to pass is quite significant

This is a fundamental problem in distributed systems that we need to address!

The Problem with Physical Clocks

- Physical clocks can record the time for a single computer, but cannot be relied upon to maintain the order of messages. This is especially important for applications like banking.



Message Sent: 10:00:00.123

Message Sent: 10:00:00.122

These messages didn't travel back in time, the clocks were out of sync!

The Ordering of Events

- Leslie Lamport - “Time, Clocks, and the Ordering of Events in a Distributed System”
- For many problems, we don’t need to agree on the physical time of an event!
- We only need to agree on the *order*.

The "Happens-Before" Relation (cont.)

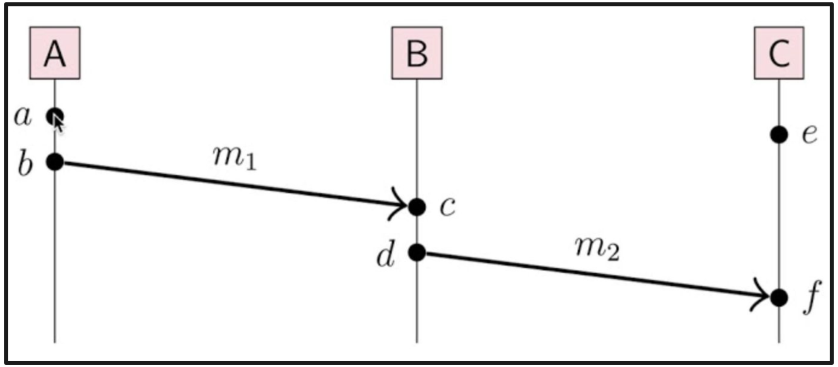
The "happens-before" relation is denoted by an arrow. It's defined by three rules:

1. If two events happen on the same machine, we know their order
2. The act of sending a message must happen before the act of receiving that same message
3. The relation is transitive: if A causes B, and B causes C, then A causes C.

Consider three processes A, B, C and six distinct events a, b, c, d, e, f. In this example:

- a -> b
- b -> c
- b -> f

What else can we say about the order of events in this example?

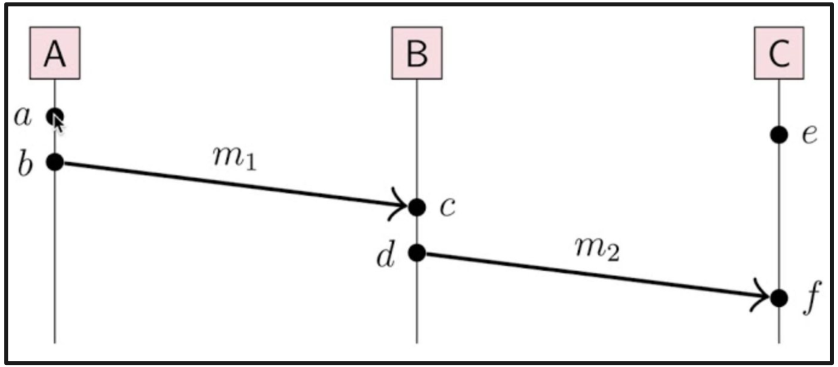


Lamport Timestamps

- Instead of relying on a physical clock, we use a *logical clock*
- Assign a number to each event in the order in which it occurs
    - We call this the *Lamport Timestamp*
- If A -> B (meaning A happens before B) then TS(A) < TS(B)

- In the previous example, we would have:

    TS(a) = 1 TS(b) = 2
    TS(c) = 3 TS(d) = 4
    TS(e) = 1 TS(f) = 5

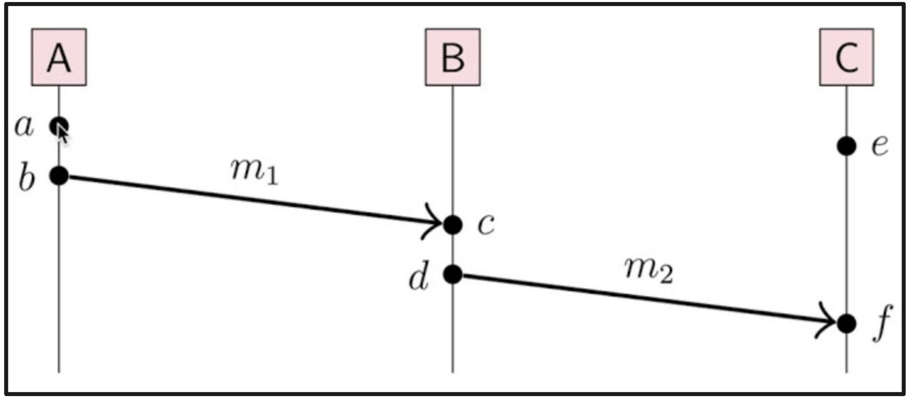


Lamport Clocks

When a process (e.g. A) starts, it's clock is 0
When an event happens, the clock is incremented
When a message is received, the process timestamp becomes the max of its current timestamp and the timestamp of the received message + 1

In the previous example, we would have:

TS(a) = 1
TS(b) = 2
TS(c) = 3
TS(d) = 4
TS(e) = 1
TS(f) = 5



Partial Order vs Total Order

- The "happens-before" relation is a *partial order*, which means concurrent events can have identical timestamps

- Many distributed algorithms require a *total order* to deterministically sequence every event in the system for consistent coordination.

- We can extend the partial order into a total order by using a tie-breaking rule. The standard approach is to use the process ID for events with the same timestamp:

1. If TS(a) < TS(b), event a comes before event b
2. If TS(a) = TS(b) and ID(a) < ID(b), event a comes before event b

Coordination via Mutual Exclusion

Mutual Exclusion

- A "critical section" is a block of code accessing a shared resource (e.g., a file) that cannot be safely executed by multiple processes at the same time.
- We want to enforce *mutual exclusion*: a property that guarantees only one process can enter its critical section at any given time, preventing data corruption.
- On a single computer, this is solved with OS tools like mutexes that rely on shared memory.
  - In most distributed systems, there is no shared memory!
- The only mechanism available for processes to coordinate and solve the mutual exclusion problem is by passing messages to one another.

Approach #1: Centralized

- Designate a single process as a coordinator to manage all access to the critical section.
- A process sends a REQUEST to the coordinator and waits for a GRANT message before entering. When finished, it sends a RELEASE message, allowing the coordinator to grant access to the next process in its queue.
- This is a simple approach, and enforces mutual exclusion
- However, the coordinator represents a critical weakness:
    - If it crashes, the system can no longer access the resource.
    - All requests must be routed through it, creating a bottleneck

Approach #2: Distributed Mutual Exclusion

- A fully distributed, permission-based algorithm with no central coordinator
- Uses the total order of Lamport timestamps to determine who enters the critical section next
- All processes agree to grant access to the request with the earliest timestamp
- Avoids a single point of failure but requires more messages

Distributed Computing with JAX

Introduction to JAX

- JAX is an accelerated Python extension that allows for fast distributed computing
- It looks much the same as NumPy, a very popular Python library for mathematical operations on large arrays of numbers

Example: Numpy to JAX

```
import numpy as np
import jax.numpy as jnp

# JAX arrays look and feel like NumPy arrays
x_np = np.linspace(0, 10, 100)
x_jnp = jnp.linspace(0, 10, 100)

# We can run familiar operations
y_jnp = jnp.sin(x_jnp) * 2.0

print(f"Created a JAX array of shape: {y_jnp.shape}")
print(f"Data type: {y_jnp.dtype}")
```

Recall: Four Key JAX Functions

- `jit`: Just-in-time compilation for speed
- `grad`: Automatic differentiation for training models.
- `vmap`: Automatic vectorization for batching
- `pmap`: Parallel execution across multiple devices

Code Example: `jit`

```
import jax
import jax.numpy as jnp
from jax import jit
import numpy as np

# A function with multiple element-wise
operations
def f(x):
    return jnp.sin(x) + 2 * jnp.cos(x) *
jnp.tanh(x)

# Create a JIT-compiled version using a
decorator
@jit
def f_jit(x):
    return jnp.sin(x) + 2 * jnp.cos(x) *
jnp.tanh(x)
```

```
x = jnp.ones((5000, 5000))

# The first call to f_jit will be
slower due to compilation.
# Subsequent calls will be much
faster than the pure Python version.
print("Running JIT-compiled
version...")
%timeit f_jit(x).block_until_ready()

print("\nRunning pure
Python-dispatched version...")
%timeit f(x).block_until_ready()
```

Code Example: `grad`

```
import jax
import jax.numpy as jnp
from jax import grad

# Define a simple scalar function f(x) = x^3 + 2x
def f(x):
    return x**3 + 2.0 * x

# Use grad to get a new function that computes the derivative, f'(x)
df_dx = grad(f)

# f'(x) = 3x^2 + 2
# Evaluate the derivative at x = 4.0. Expected: 3 * (4^2) + 2 = 50
gradient_value = df_dx(4.0)

print(f"The original function f(4.0) is: {f(4.0)}")
print(f"The gradient df/dx at x=4.0 is: {gradient_value}")
```

Questions?