Department of Computer Science
University of California, Los Angeles

Computer Science 134: Distributed Systems

Fall 2024
Prof. Ryan Rosario

Lecture 5: October 14, 2024

Outline

1. Case Study: MapReduce
2. Time Sychronization

1 Case Study: MapReduce

2 Time Sychronization

MapReduce

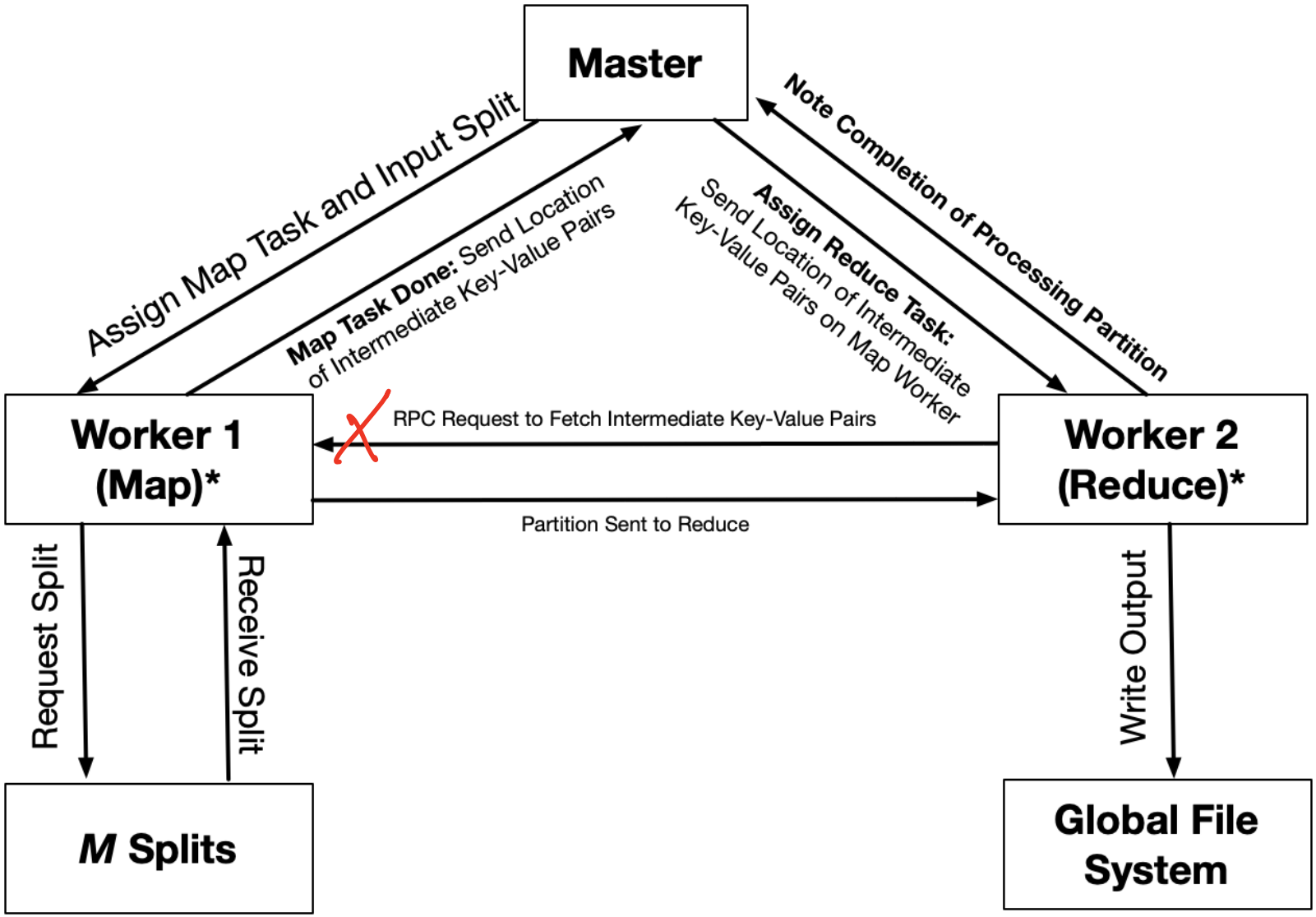
Last time we discussed the Google MapReduce paper. We learned:

- The purpose of MapReduce and how it works
- What the map and reduce phases do
- How MapReduce achieves parallelism
- How a master was used to coordinate data sharing and parallelism among workers
- Fault tolerance on worker and master nodes
- Optimizing/minimizing I/O with data locality
- How load balancing is achieved

Partitioner

Today we will finish off by discussing refinements and optimizations.

MapReduce (contd.)



Refinements

A series of optimizations were implemented for efficiency purposes:

1. Partitioning Functions
2. Ordering Guarantees
3. Combiner
4. Input and Output
5. Side Effects
6. Skipping Bad Records
7. Local Execution
8. Status Information
9. Counters

We will only discuss a few. None of these are part of Project 1.

Refinement: Partitioning Functions

Earlier, we defined a very simple partition function. The partition function assigns an intermedia key-value pair to a partition, which is then sent to a reduce task.

Remember that:

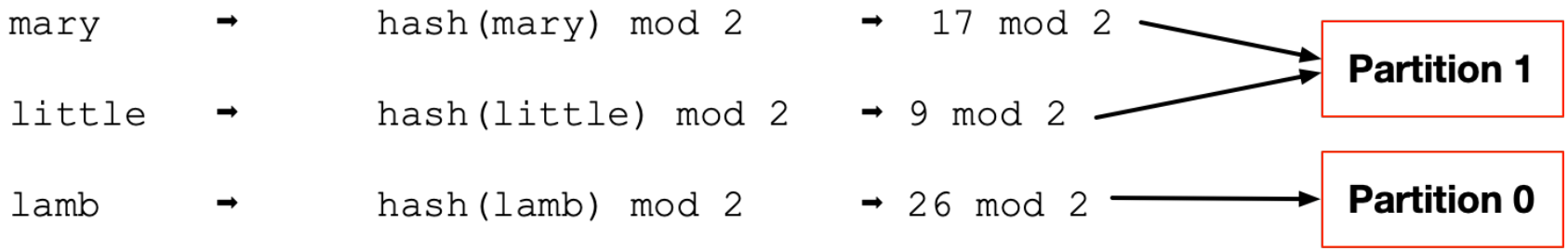
- Multiple unique keys can be found in the same partition (hash).
- All key-value pairs with the same key will be in the same partition.
- A reduce task processes a single partition.

Refinement: Partitioning Functions (contd.)

The simplest partition function is just

$$\texttt{hash(key) mod } R$$

Refinement: Partitioning Functions (contd.)



But there may be cases where this is not what we want.

Refinement: Partitioning Functions (contd.)

Suppose we have millions of URLs and the number of times that URL was visited:

```
("http://www.ucla.edu/donations", 500000)
("http://www.ucla.edu/admissions/apply", 80500000)
("http://www.ucla.edu/alumni", 427)
("http://www.google.com/search", 10000000000)
("http://www.google.com/wave", 2)
("http://www.google.com/gemini/pro", 10000000)
```

Refinement: Partitioning Functions (contd.)

Example/Question: Consider the case where the key is a URL. What is the default behavior of the partitioner?

Question: Might we want to partition these URLs differently? How? Why? Will this mess up the reduce task?

Refinement: Partitioning Functions (contd.)

Exercise: What other major problem might we have, where a partitioner can be useful?

Refinement: Partitioning Functions (contd.)

Partitioners can be implemented using the `Partitioner` class in Hadoop or the `Partitioner` class in Spark.

```
public static class CaderPartitioner extends Partitioner < Text, Text >
{
    @Override
    public int getPartition(Text key, Text value, int numReduceTasks)
    {
      String[] str = value.toString().split("\t");
      int age = Integer.parseInt(str[2]);

      if(numReduceTasks == 0) { return 0; }

      if(age<=20) { return 0; }
      else if(age>20 && age<=30) { return 1 % numReduceTasks; }
      else { return 2 % numReduceTasks; }
    }
}
```

Refinement: Ordering Guarantees

Within a given partition, all intermediate key-value pairs are sorted in ascending order by key.

This means that the output will also be sorted in ascending order by key.

This is useful for downstream use cases where we need to support random access lookup by key, such as an index for a data store or data querying platform (e.g. Hive).

Refinement: Combiner

There can be a lot of repetition and redundancy in the intermediate key-value pairs formed from the map phase.

For example, suppose our input split looks like:

```
mary had a little lamb little lamb little lamb
the quick mary jumps over the lazy lamb
```

Refinement: Combiner (contd.)

The intermediate key-value pairs are:

```
("mary", 1), ("had", 1), ("a", 1), ("little", 1),
("lamb", 1), ("little", 1), ("lamb", 1),
("little", 1), ("lamb", 1), ("the", 1),
("quick", 1), ("mary", 1), ("jumps", 1), ("over", 1),
("the", 1), ("lazy", 1), ("lamb", 1)
```

Note the repetitiveness. This is too much. It wasts memory, disk space and network I/O.

Refinement: Combiner (contd.)

Instead, we can execute a **combiner** that executes at the end of each map task. It does not affect the reduce phase. Less memory and disk use on the map worker, and less network I/O.

```
("mary", 2), ("had", 1), ("a", 1), ("little", 3),
("lamb", 4), ("the", 2), ("quick", 1), ("jumps", 1),
("over", 1), ("lazy", 1)
```

Typically the combiner uses the same code as the reducer. Consider it a map-side pre-reducer. Combiners are implemented in Hadoop.

Refinement: Combiner (contd.)

```
int main(int argc, char** argv) {
  ParseCommandLineFlags(argc, argv);

  MapReduceSpecification spec;

  // Store list of input files into "spec"
  for (int i = 1; i < argc; i++) {
    MapReduceInput* input = spec.add_input();
    input->set_format("text");
    input->set_filepattern(argv[i]);
    input->set_mapper_class("WordCounter");
  }

  // Specify the output files:
  //    /gfs/test/freq-00000-of-00100
  //    /gfs/test/freq-00001-of-00100
  //    ...
  MapReduceOutput* out = spec.output();
  out->set_filebase("/gfs/test/freq");
  out->set_num_tasks(100);
  out->set_format("text");
  out->set_reducer_class("Adder");

  // Optional: do partial sums within map
  // tasks to save network bandwidth
  out->set_combiner_class("Adder");

  // Tuning parameters: use at most 2000
  // machines and 100 MB of memory per task
  spec.set_machines(2000);
  spec.set_map_megabytes(100);
  spec.set_reduce_megabytes(100);
```

Refinement: Input and Output Types

MapReduce can work with many different kinds of data.

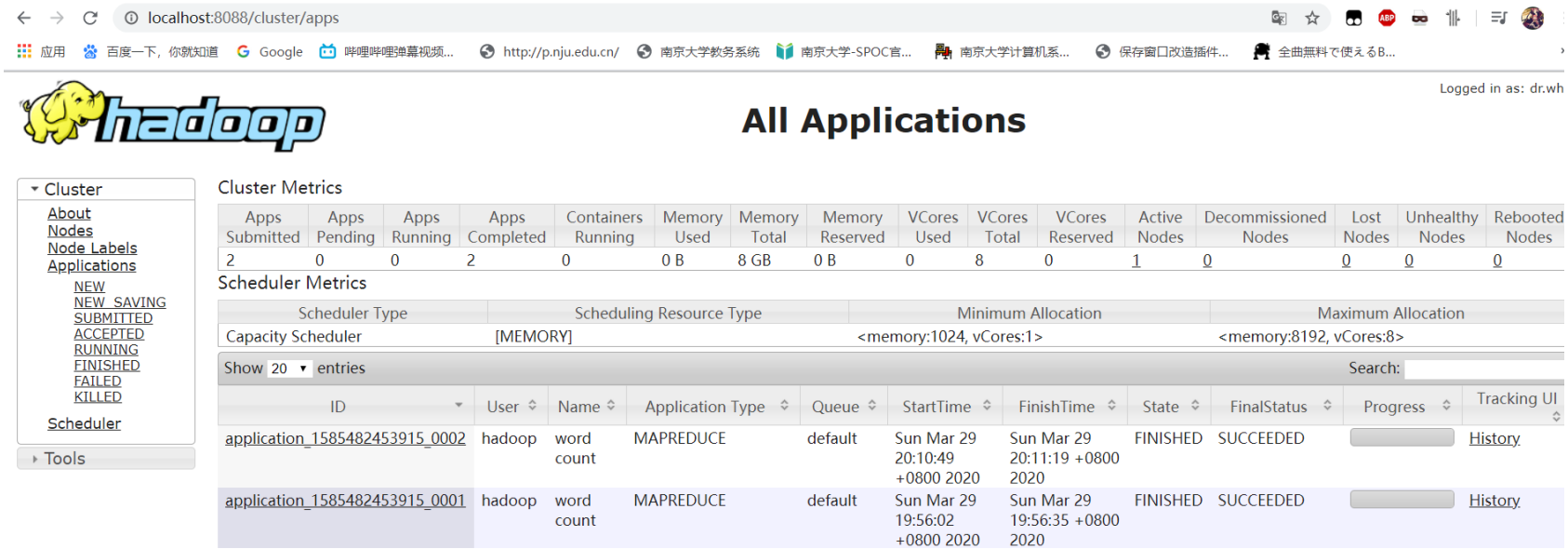
The simplest file format is one record per line, and this is the default in Hadoop.

Users can write custom *readers* that specify how to find the beginning and ending of a record, so that records can be organized into splits effectively.

Likewise, the user may wish to write output to formats other than one record per line: binary files, JSON etc.

Refinement: Status Information

MapReduce implementations include a web service that allows users to see statistics, logs and progress information for various aspects of jobs and tasks.



Uses of MapReduce at Google

MapReduce was used for a long time at Google for various use cases:

1. large-scale machine learning
2. clustering problems for Google News and Froogle
3. processing data reports of popular queries
4. extracting properties of web pages for new products
5. large-scale graph computations
6. large scale indexing

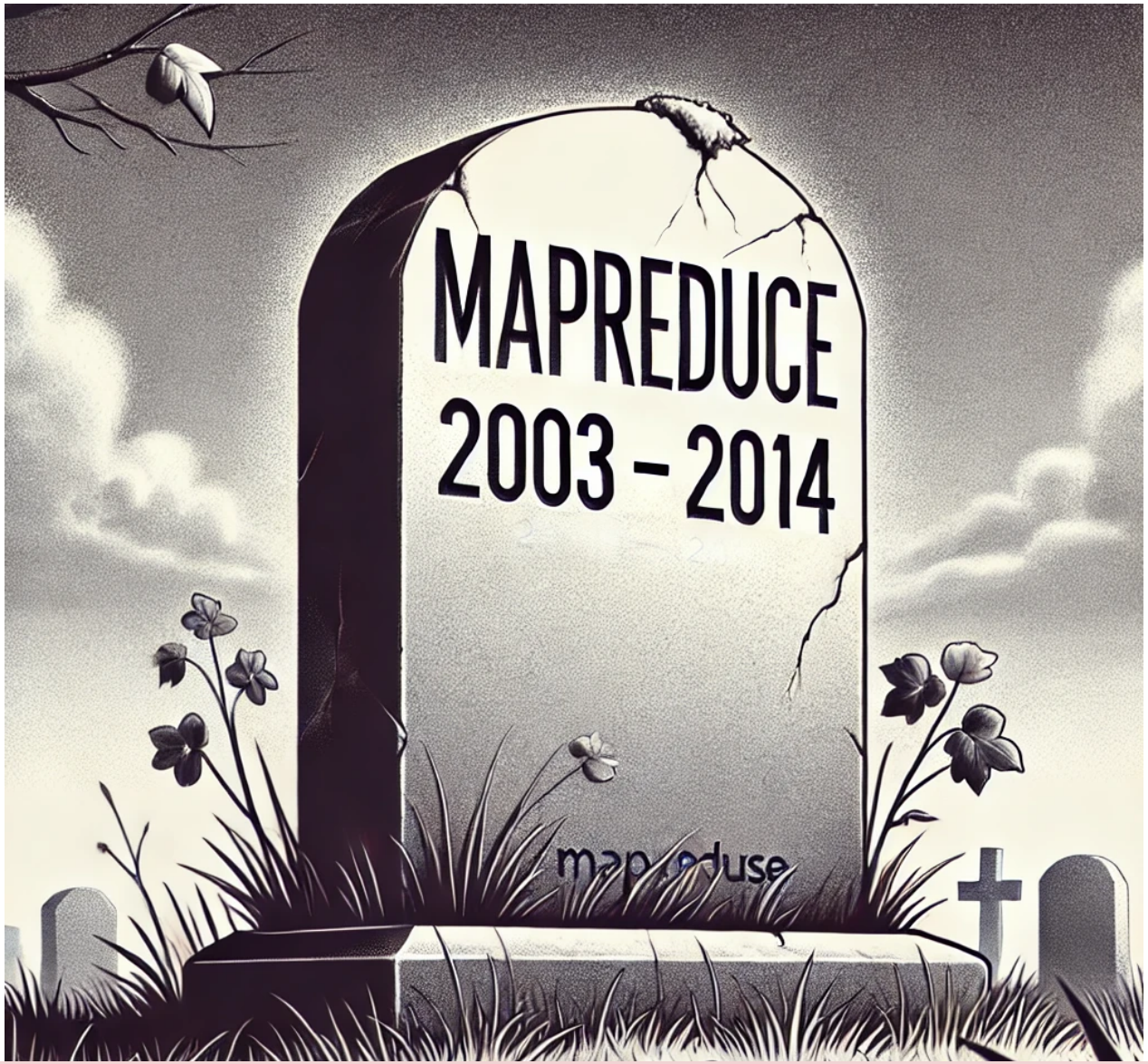
RIP MapReduce at Google

Google deprecated MapReduce internally in 2014. It had a long life, but problems became harder to solve and did not fit nicely into a map phase and a reduce phase, particularly iterative problems in machine learning. Google now uses:

1. **Flume** as the batch processing framework;
2. **Millwheel and Dataflow** for stream processing

Apache Beam was developed from MapReduce, Flume and Millwheel.

RIP MapReduce at Google (contd.)



1 Case Study: MapReduce

2 Time Sychronization
- Time
- Clock Synchronization
- Ordering of Messages
- Logical Time

Time

Today we are going to talk about **time**... our constant enemy.

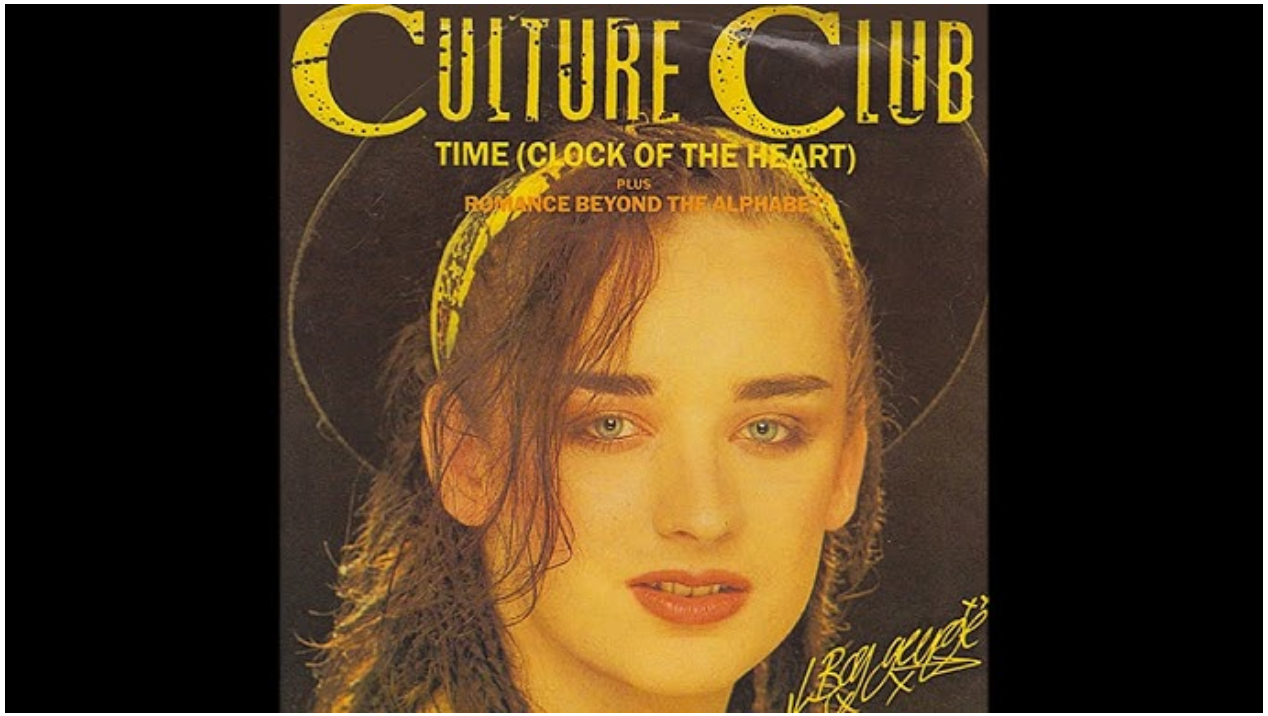
Time (contd.)

"Time is a precious thing. Never waste it." – Arthur O'Shaughnessy
(popularized by Willy Wonka, 1971)

Time (contd.)

"Time, why you punish me?" – Hootie and the Blowfish

Time (contd.)



"Time is like a clock in my heart." – Culture Club

Time (contd.)

And clocks...

Time: A Mystery

On June 30/July 1, 2012, many online services and systems crashed simultaneously, locked up and stopped responding.

Airlines could not process check-ins or reservations for hours. **What happened?**

Standby...

Time in Distributed Systems

An accurate concept of time is required for distributed systems to work effectively:

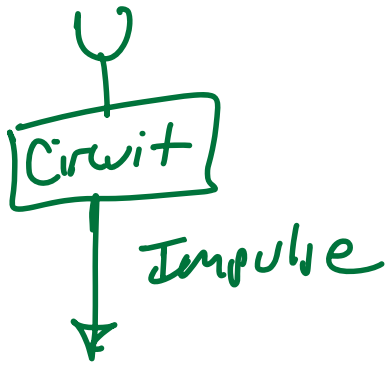
- Schedulers (`cron`), timeouts, failure detection, retry timers
- Performance measurement, statistics, profiling
- Log files and databases
- Data that are time-sensitive (e.g. cache) TTL
- **Determining the order of events across a series of nodes.**

Two types: **physical clocks** and **logical clocks**.

Quartz Clocks

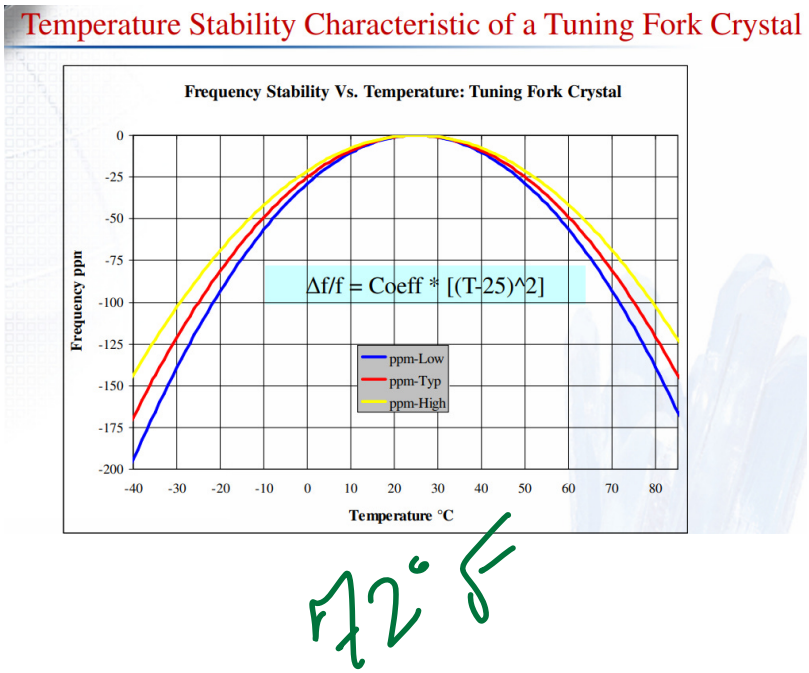
PHYSICAL

While analog clocks are based on pendulum-like devices, digital clocks like in a digital watch use a vibrating crystal. It oscillates 32,768 times per second.



Quartz Clocks (contd.)

Quartz clocks are very cheap, but also not very accurate. Due to manufacturing flaws, some clocks run fast or slow. Also, temperature affects the speed of the clock:



- Drift is measured in parts per million (ppm).
- 1 ppm $=$ 1 μs / s $=$ 86 ms/day $=$ 32s/year
- Most computer clocks accurate to $\approx$ 50ppm

Atomic Clocks

If we need more accuracy, we can use an atomic clock which is based on quantum mechanics of cesium-133 or rubidium.

- One second is equivalent to 9,192,631,770 periods of a particular frequency of cesium-133.
- Accuracy ≈ 1 in 10^{-14}, 1 second error in 3 million years.
- Price: \$25,000 for cesium-133 clock or $\approx$ \$1300 for rubidium.

GPS as a Time Source

Several satellites orbit the Earth and transmit highly precise time from an atomic clock to receivers.

- 31 satellites, each with an atomic clock
- satellite broadcasts time and its location
- calculate position from speed-of-light delay between satellite and receiver
- correct for atmospheric effects, relativity etc.

Definition of Time

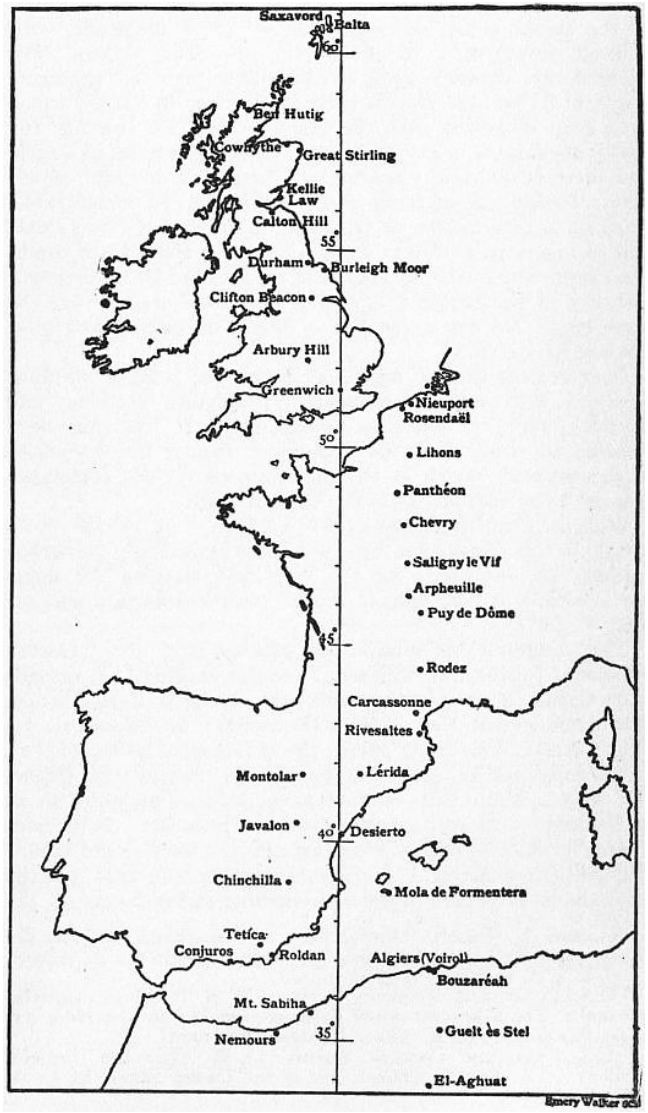
Atomic clocks are extremely accurate, but they use a different concept of time, from quantum mechanics.

One rotation of the Earth around its axis does not take $365 \times 24 \times 60 \times 60 \times 9192631770$ periods of cesium-133's resonant frequency. The speed of Earth's rotation is not constant!

Two definitions: one based on quantum mechanics and other from astronomy.

Coordinated Universal Time (UTC)

- **Greenwich Mean Time (GMT)**. It's noon when the sun is in the south, as seen from the meridian.
- **International Atomic Time (TAI)**. 1 day is $24 \times 60 \times 60 \times 9,192,631,770$ periods of cesium-133's resonant frequency.
- **Issue:** Earth's rotation speed is not constant.
- **Solution**. UTC is TAI with corrections for Earth's rotation
- Time zones and daylight saving time are offsets to UTC.



Leap Seconds

To adjust for the Earth's rotation, UTC features *leap seconds*. Every year on June 30th and December 31st, at 23:59:59 UTC, one of three things can happen:

1. **Negative Leap Second**: the clock automatically jumps to 00:00:00 skipping that last second.
2. **Positive Leap Second**: the clock moves to 23:59:60 after one second and then to 00:00:00 after another.
3. **Nothing Interesting**: the clock moves to 00:00:00 after one second

Leap Seconds (contd.)

Leap seconds are announced to interested parties months in advance.

See more breathtaking captures of leap seconds here.

Leap Seconds (contd.)

In 2022, it was decided that leap seconds should be deprecated by 2035.

The next leap second is on June 30, 2025.

Leap seconds have caused all kinds of havoc in the 21st century.

Leap Seconds (contd.)

So, how does most software deal with leap seconds? **It doesn't!**

But operating systems and distributed systems need very accurate times with sub-second precision.

23:59:60

On June 30, 2012, a bug in the Linux kernel caused a **livelock** on the leap second causing servers to be overwhelmed.

Solution: Spread out, or **smear** the leap second over the course of the day.

Solving the Mystery

On June 30, 2012, there was a positive leap second and some software did not recognize `23:59:60`.

In particular the `hrtimer` subsystem is used when application is sleeping while waiting for the OS to do another task. It sets an alarm to start these sleeping tasks when the OS is spending too much time with its other work.

Async

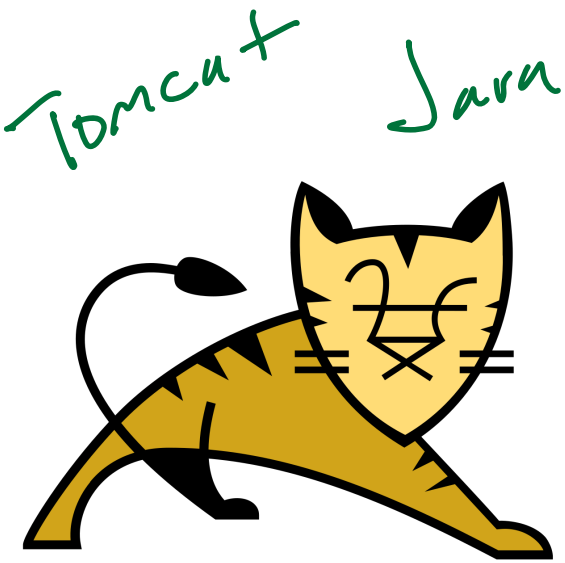
Issues a callback

By adding this extra second, *many* processes were deemed as "taking too long" and a bunch of sleeping processes woke up overloading the CPU.

Solving the Mystery (contd.)

Over at Reddit, Cassandra was faling to pause its Java processes and getting stuck in spinning loops, overwhelming the CPU.

Solving the Mystery (contd.)



Source: Wired

Life would be so much easier if Earth just didn't rotate, right?

Clock Synchronization

Computers use quartz clocks to track physical time/UTC. It has a battery that continues to run even when the machine is shutdown.

CMOS Can cause crashes.

The clock's error will gradually increase over time, which is called **clock drift**.

The difference between two clocks at a given point in time is **clock skew**.

So, how do we keep the time up to date? We can install an authoritative time source (e.g atomic clock, GPS), right? Easy!

Clock Synchronization (contd.)

Atomic clocks are too expensive and cumbersome to install into a computer or phone, so cheaper but less accurate quartz clocks are used instead.

Since these clocks drift, they need to be adjusted from time to time using something more authoritative:

- Network Time Protocol (NTP) ✓
- Precision Time Protocol (PTP)

Network Time Protocol (NTP)

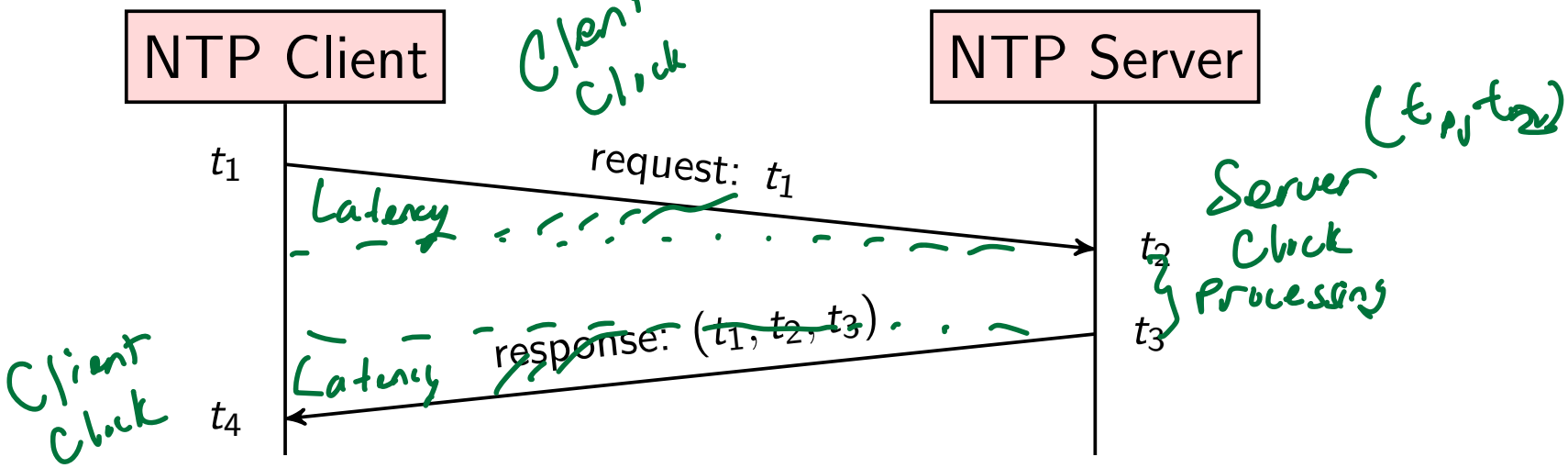
Most operating system vendors run NTP servers and the OS is configured to use them.

Servers are organized into strata :

- **Stratum 0:** atomic clock of GPS receiver
- **Stratum 1:** synced directly with device in stratum 0
- **Stratum 2:** servers that sync with stratum 1 etc.

NTP

Estimating Time Over a Network



Round-trip network delay: $\delta = (t_4 - t_1) - (t_3 - t_2)$

Esimate of server time when client receives response: $t_3 + \frac{\delta}{2}$

Estimated clock skew: $\theta = \left(t_3 + \frac{\delta}{2}\right) - t_4 = t_3 + \frac{(t_4 - t_1) - (t_3 - t_2)}{2} - t_4$

But network latency and request/response processing are unpredictable:

Correcting Clock Skew

So, let's let C be the client's current time. Since we have calculated θ, all we need to do is set $C := C + \theta$, right?

Typically, no. That is too drastic! **Why?**

Providers might miss certain milestones.
01:30:00 → 01:35:00
without triggering at 01:32:00

Providers wake up simultaneously
→ livelock

Correcting Clock Skew (contd.)

Smaller probability of missing time milestones.

We adjust the client's clock to be in line with the server.

We can adjust the client's clock speed either faster or slower, gently, which will bring the clock in line with the server's, usually eliminating the skew in a few minutes.

This is called **slewing** the clock.

~ 5 minutes

Correcting Clock Skew (contd.)

Once we have estimated the clock skew θ our method of resolution depends on how large it is.

- If $|\theta| < 125$ms, **slew** the clock. Gently speed the clock faster or slower. Takes $\approx$ 5mins.
- If 125ms $\leq |\theta| < 1000$ms, **step** the clock. Simply reset the clock to match the server as slewing takes too long. This should be rare as there are unintended consequences.
- If $|\theta| \geq 1000$ms, **panic** and do nothing. A human operator must intervene.

Any system that relies on clock synchronization must monitor clock skew!

Get to Stepping: Real-Time vs. Monotonic Clocks

As we discussed earlier, **stepping** the clock can be problematic. In software, we have two different kinds of clocks:

1. **Time-of-day** or **Real-time clock**
Based on physical time 02:00:00
BAD: Leap second, skew
2. **Monotonic clock**
Only count forward.
# of seconds since an arbitrary event:
Process start, boot the system...

Get to Stepping: Real-Time vs. Monotonic Clocks (contd.)

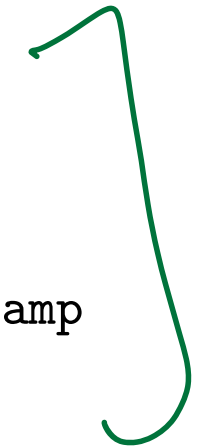
When computing elapsed time, the decision of which to use has serious consequences:

- **Time-of-day** is susceptible to stepping. Elapsed times may be significantly larger than expected, or even negative. (`currentTimeMillis()` in Java)
- **Monotonic** always increases... it's monotonic! Even if NTP steps, it does not matter. It only counts seconds. (`nanoTime()` in Java)

Both are affected by skewing, which does introduce some error.

Code Example

```java
// Don't do this:
long startTimestamp = System.currentTimeMillis();
foo();    // Do something. NTP client steps the clock during execution
long endTimestamp = System.currentTimeMillis();
long elapsedMs = endTimestamp - startTimestamp;
// elapsedMs may be negative.

// Do this instead:
long startTimestamp = System.nanoTime();
foo();
long endTimestamp = System.nanoTime();
long elapsedNs = endTimestamp - startTimestamp
// elapsedNs >= 0
```



Summary

Real-time clocks:

- Time since a fixed date (e.g. January 1, 1970)
- Susceptive to stepping forwards or backwards and leap seconds
- Can be compared across nodes as long as they are in sync.

Monotonic clocks:

- Time since an arbitrary point (e.g. system boot, process start)
- Monotonically increasing, always moves forward, almost at constant rate
- Good for measuring elapsed time on a node
- Cannot be compared across nodes

Summary (contd.)

Checkpoint: Up to this point, we have been discussing physical clocks.

Both time-of-day/real-time and monotonic clocks measure some kind of actual, physical time (e.g. seconds).

Where we Are Going: We do not always need this, and it can actually cause a lot of problems, such as with skew etc.

Ordering Messages

In the previous section, we discussed many aspects of time and clocks and how to improve clock skew.

We discussed the challenges of measuring elapsed time as well.

But there is a much more challenging aspect that we need to discuss: ordering events in a distributed system.

Ordering Messages (contd.)

$m_1 \rightarrow m_2 \Rightarrow \$1000 + \$100 + \$11 = \$1111$

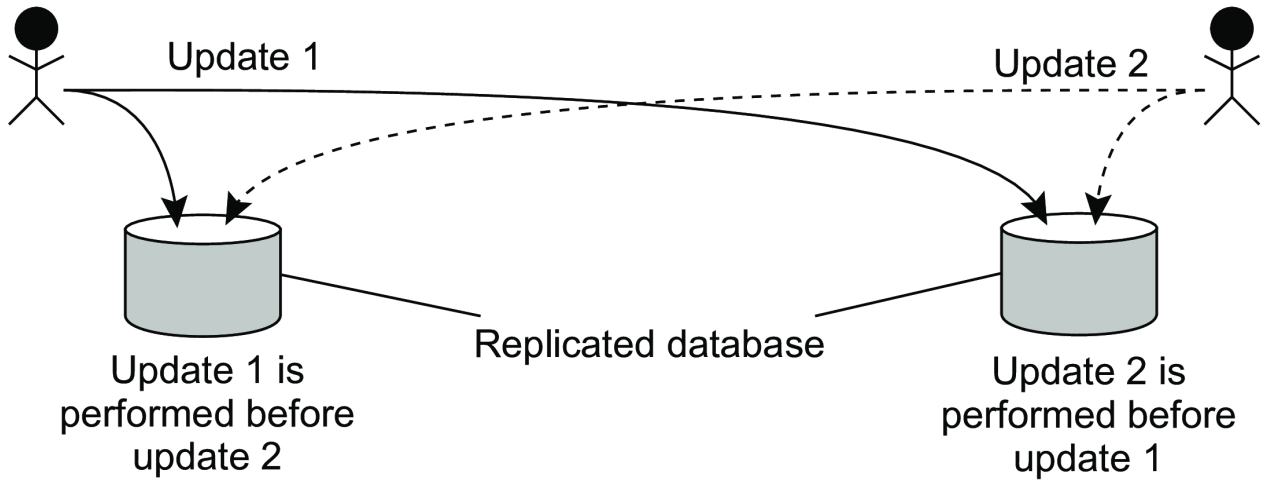
$m_2 \rightarrow m_1 \Rightarrow \$1000 + \$10 + \$100 = \$1110$

Example: A bank's data is replicated across two nodes – one in Los Angeles and the other in New York City.

The bank account contains \$1000. In LA, the user requests to deposit \$100. Meanwhile, in NYC an employee deposits 1% interest into their account.

m_1 Locally $\$1000 + \$100 = \$1100$

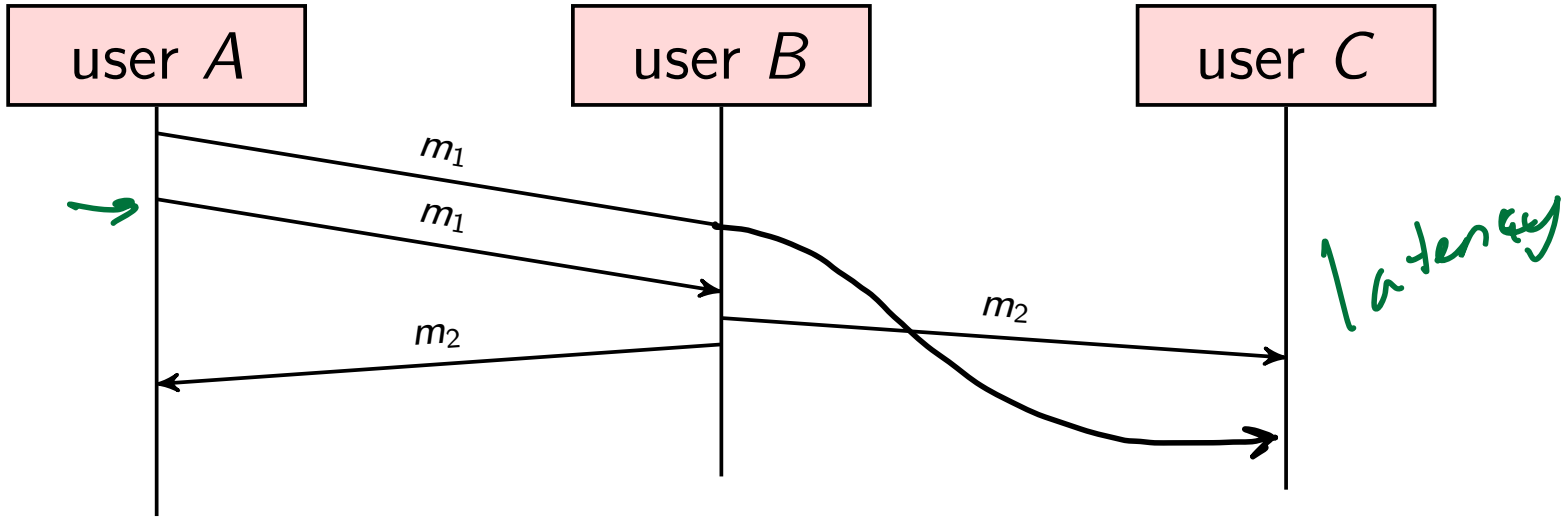
Locally $\$1000 + \$10 = \$1010$



What should the proper order of messages be?

Ordering Messages (contd.)

Let's look at a more concrete example. Recall that even on a reliable link messages may arrive in the wrong order due to latency.



m_1 = "A says: UCLA football shall win the Big 10!"
m_2 = "B says: Oh no it won't!"

User C may se m_2 before m_1 even though m_1 **happened before** m_2.

Ordering Messages (contd.)

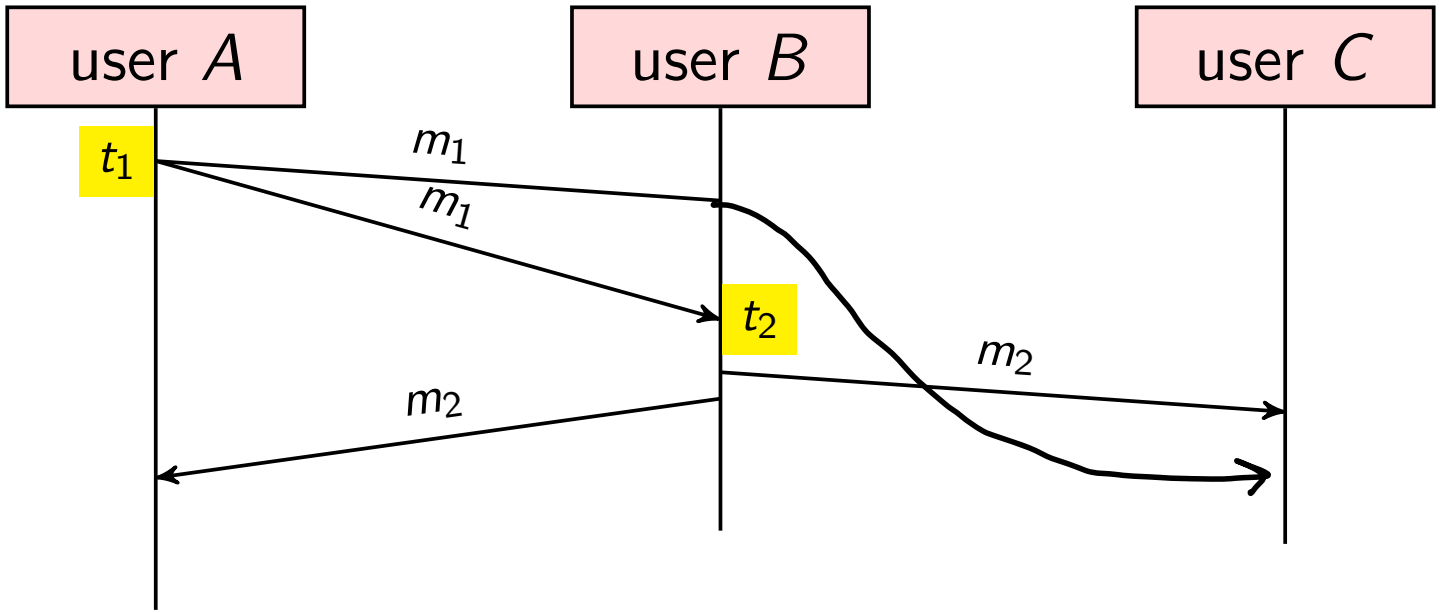
How does poor old C figure out the order of the messages?

A *monotonic clock* won't work. **Why not?** Can't compare time across nodes.

We could perhaps attach the timestamps from a real-time clock to m_1 and m_2. **Would that work?**

Clocks are not synced + Skew

Ordering Messages (contd.)



$m_1 = (t_1, \text{A says: “UCLA football shall win the Big 10!”})$
$m_2 = (t_2, \text{B says: “Oh no it won't!”})$

Problem: It's still possible for $t_2 < t_1$. due to clock skew

The Happens-Before Relation and Causality

What do we mean by *correct* order? We use the **happens before** relation mentioned earlier.

If we assume that each user executes their application in a single thread, there must be an ordering to the messages sent.

Theoretically, we say there is a **strict total order** on the events that occur on a single node. This can be generalized to multithreading by using a different node for each thread.

The Happens-Before Relation and Causality (contd.)

We must define that a message is sent **before** that same message is received by another node.

If we assume that each message is unique, we always know exactly where the message was sent from and when.

The Happens-Before Relation and Causality (contd.)

The **happens-before** relation is a *partial order* in that there may exist events a and b such that neither a happened before b nor b happened before a, yet they still appear in a particular order by timestamp.

Essentially, in that case a and b are independent of each other – that is, they are **concurrent**. The definition of concurrent is different here – there is no sequence of messages leading from one to the other. They are executed in different processes/nodes.

The Happens-Before Relation and Causality (contd.)

node = thread = process

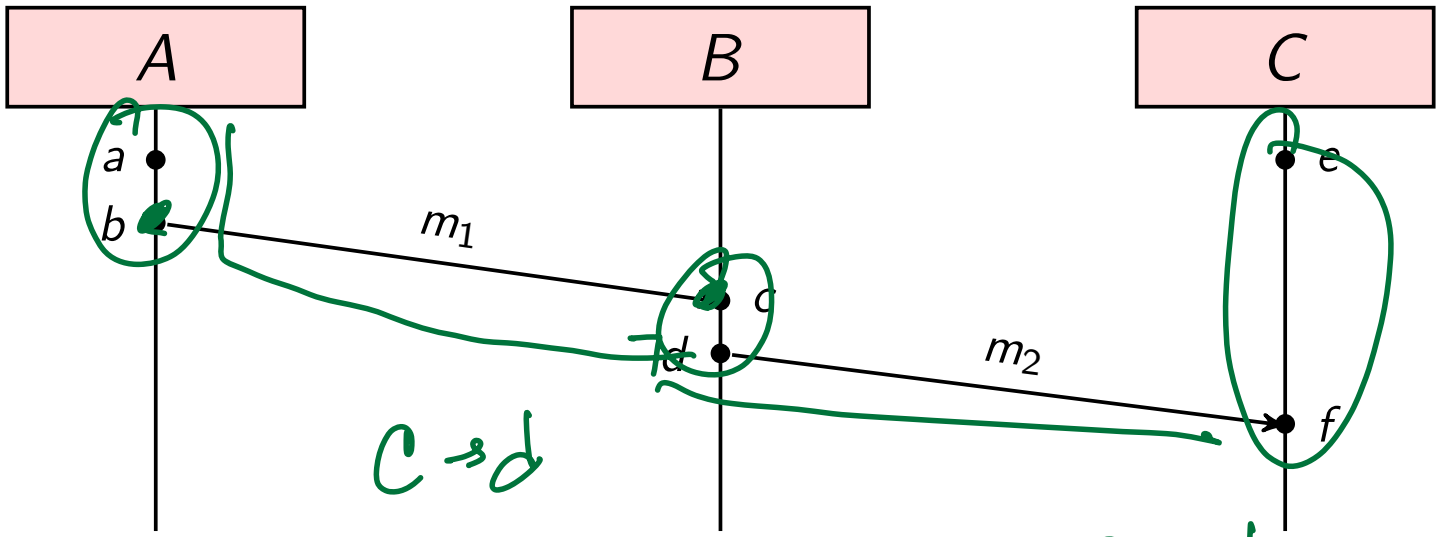
Let's define an **event** as something that is happening on some node. It could be sending or receiving a message, or a step of local execution.

An event a **happens before** b ($a \rightarrow b$) if and only if:

- a and b occurred on the same node and a occurred before b in that node's execution; or
- a is the sending of a message m and b is a process receiving it; or
- there exists an event c such that $a \rightarrow c$ and $c \rightarrow b$.

It is possible that neither $a \rightarrow b$ nor $b \rightarrow a$ and thus a and b are **concurrent** ($a||b$).

The Happens-Before Relation and Causality (contd.)



What can we say about about this scenario?

- $a \rightarrow b, c \rightarrow d, e \rightarrow f$ based on node execution rule
- $b \rightarrow c, d \rightarrow f$ based on messages m_1, m_2
- $a \rightarrow c, a \rightarrow d, a \rightarrow f, b \rightarrow d, b \rightarrow f, c \rightarrow f$ based on transitivity
- $a||e, b||e, c||e, d||e$

The Happens-Before Relation and Causality (contd.)

The happens-before is a way of starting to reason about **causality** in distributed systems.

It is a way of considering whether information could have progressed from one event to another – whether or not one event may have influenced another.

$send(m_1) \rightarrow send(m_2)$ transitive

In our football example, m_1 influenced m_2.

Broadcast Protocols

Broadcast protocols (or multicast protocols) are algorithms for delivering one message to multiple recipients.

This concept is very important for many distributed algorithms.

The main difference among them is **order in wich they deliver messages** so we will discuss how clocks can help us keep track of ordering within the system.

Logical vs. Physical Clocks

Physical timestamps may be inconsistent with causality due to clock skew – we cannot easily establish causality.

Logical clocks are designed to capture causal dependencies:

$$(a \rightarrow b) \implies (T(a) < T(b))$$

- Physical clocks count number seconds elapsed.
- Logical clocks count number of events occurred.

Timestamp

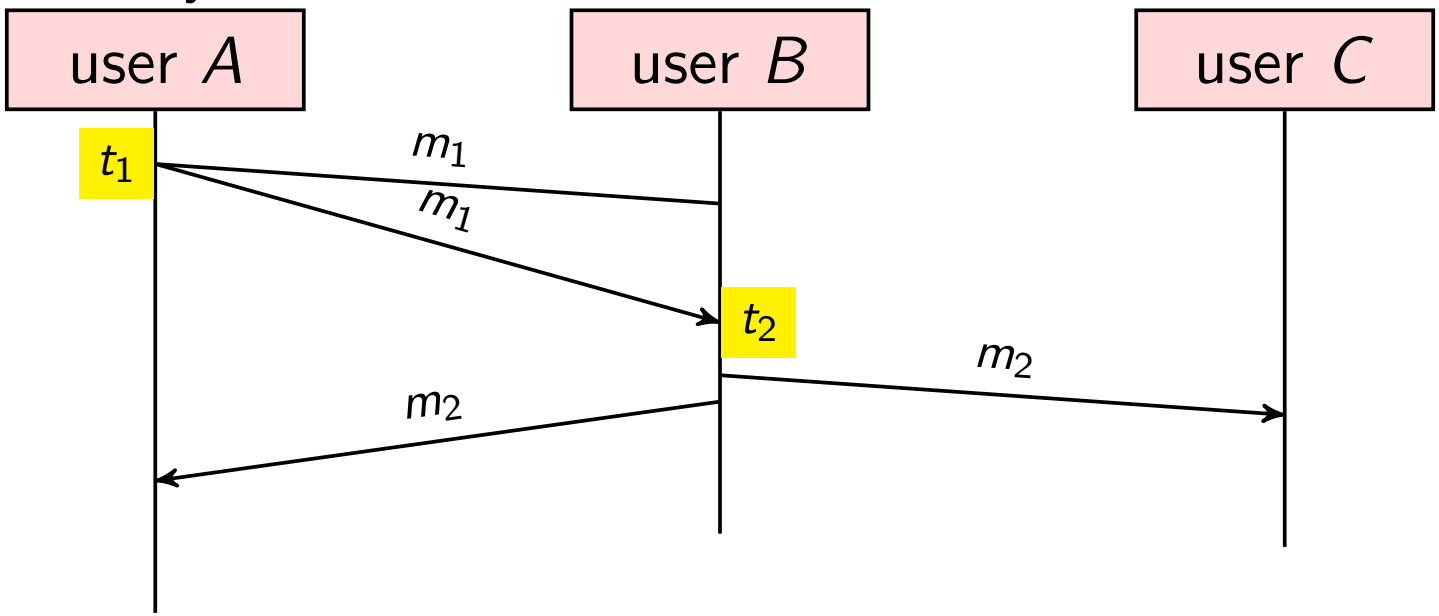
Logical Clocks

Logical clocks focus on correctly capturing the order of events in a distributed sytem. We will consider two logical clocks:

1. Lamport clocks
2. Vector clocks

Logical Clocks (contd.)

But why?

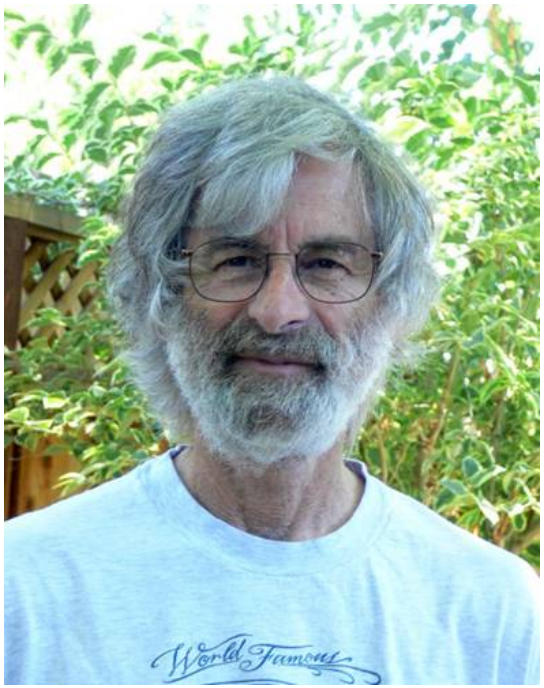


We know that send(m_1) $\rightarrow$ send(m_2) is true, and so we would expect that $t_1 < t_2$ but this is not necessarily the case with a physical clock even if the clocks are synced.

Logical Clocks (contd.)

The purpose of **logical clocks** is to correctly capture the order of events in a distributed system.

Lamport Clocks



1978

The Lamport clock does not use standard timestamps.

It instead makes clever use of sequence numbers that are passed to other processes to maintain a particular order.

Lamport Clocks (contd.)

Assumption: Each node executes only one thread.

Each node has a counter t that is incremented on each local event e. Incremented right before e is executed

Let $L(e)$ be the current value of the counter when event e occurs.

Lamport

Lamport Clocks (contd.)

Here is what we do (in words):

1. On initialization of the communication, set t on each node to be 0.
2. Right before executing an event on node i (e.g. internal event, send a message, delivering message to application), increment t_i.
3. If the event is to transmit a message from node i to another node j:
   1. Send the value of t_i in the message m to node j.
   2. Upon node j receiving the message, it updates its own local clock according to: $t_j = \max(t_i, t_j) + 1$.

Lamport Clocks (contd.)

If we let $L(e)$ be the value of the local counter (sequence number) after being incremented, then we have the following properties:

- If $a \rightarrow b$ then $L(a) < L(b)$
- But $L(a) < L(b)$ does not imply $a \rightarrow b$
- It is possible for $L(a) = L(b)$ for $a \neq b$

where $L(x)$ returns the current counter at the execution of event e.

Lamport Clocks (contd.)

```
on initialization do:
    t := 0
end on

for each event e on a node do:
    if the event occurs only on the local node do:
        t := t + 1
    end on

    if the event sneds a message m to j do:
        t := t + 1
        send(t, m)
    end on

// On node j:
on receiving (t', m) do:
    t := max(t, t') + 1
    deliver m to the application
end on
```

Lamport Clocks (contd.)

So a Lamport Clock is really just a counter of events.

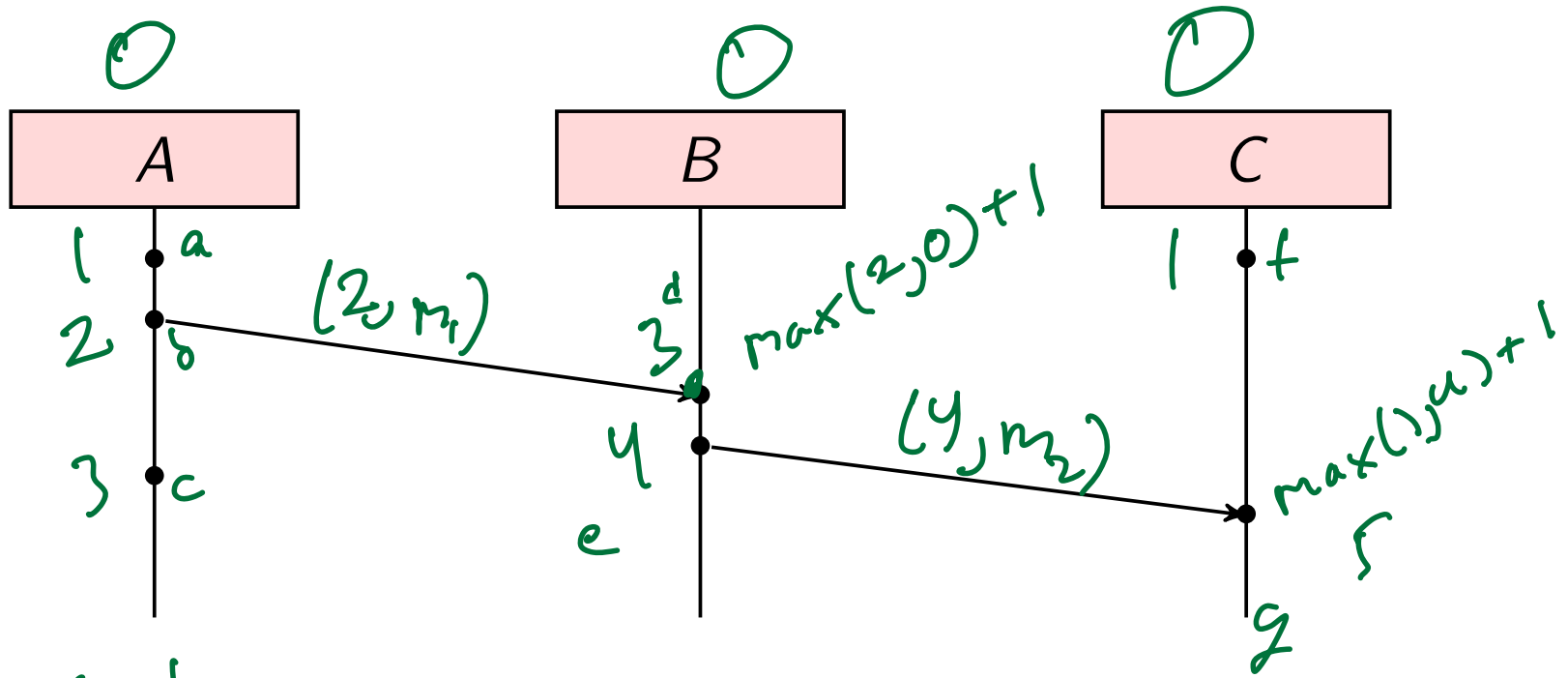
It has no concept of physical time.

Time is always increasing after each event on a per node basis.

Logically Monotonic

Lamport Clocks (contd.)

Let's do an example. Find the Lamport clock for each event:



a, f, b, c, d, e, g

Happens-Before and Lamport Clocks

The **happens-before** relation, and Lamport Clock, imparts a **partial order** over events: Mathematically it doesn't but may say it anyway! I can prove it! ☺

- **Reflexive:** For event a, $a \rightarrow a$. (not true for happens before)
- **Antisymmetric:** If $a \rightarrow b$ and $b \rightarrow a$ then $a = b$.
- ~~**Reflexive:**~~ **transitive** If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$.

This means that *some* of the events are ordered, but there are other events that are not and have duplicate counters.

We can get a **total order** by enforcing no duplicate clocks in the process by attaching the process ID to the counter.

Happens-Before and Lamport Clocks (contd.)

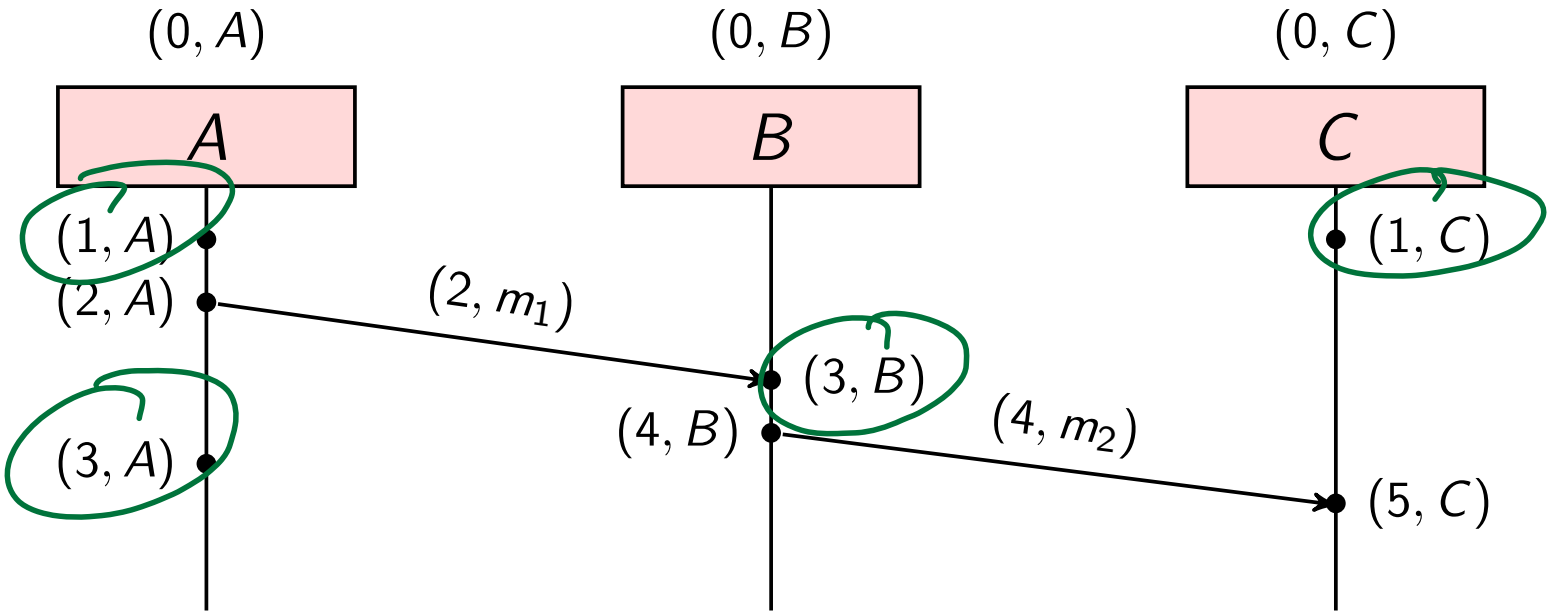
Again:

Total order means that every event is ordered with respect to every other event in the system. Partial order means that some events are not part of this ordering, and this is apparent when we have duplicate clocks.

Lamport Clocks guarantee partial order, but can be extended with process information to enforce total order. **We break ties using the process tag.**

Happens-Before and Lamport Clocks (contd.)

Let's look at an example:



The correct full ordering of events is
$(1, A), (1, C), (2, A), (3, A), (3, B), (4, B), (5, C)$.

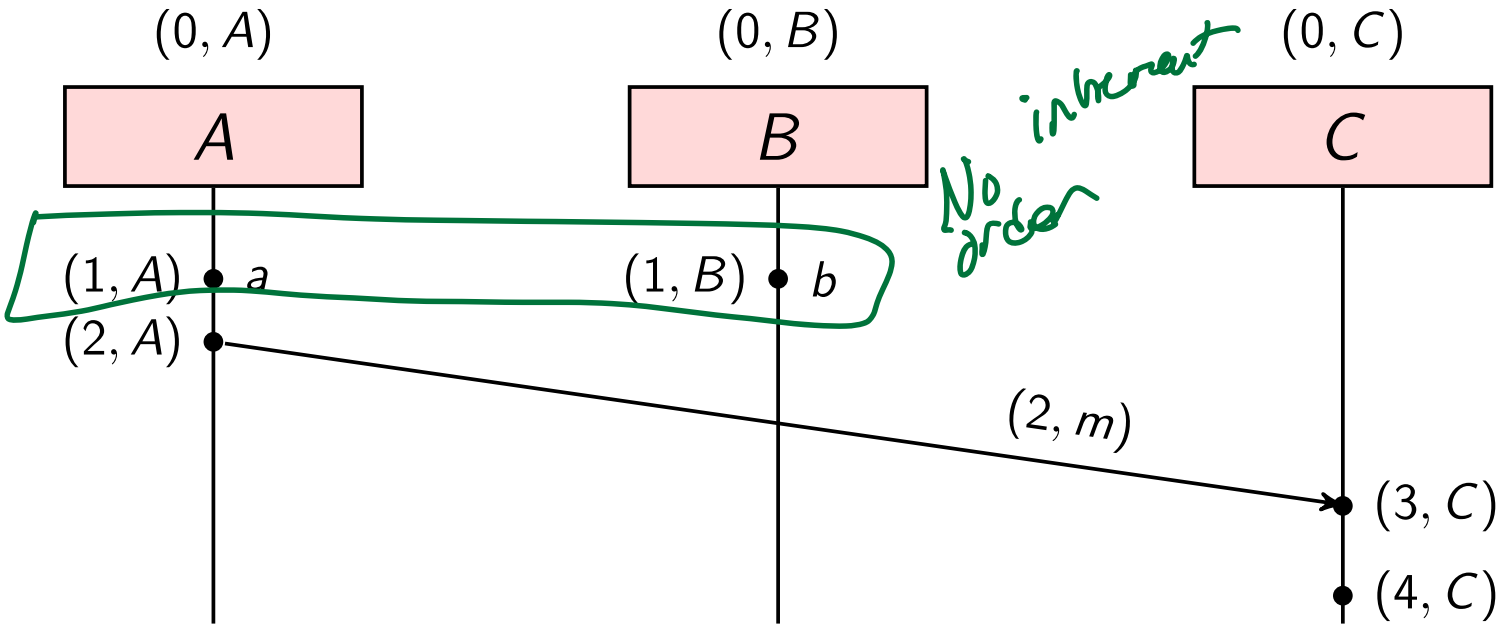
Happens-Before and Lamport Clocks (contd.)

We can construct a full order of events using the modified Lamport clock by appending process IDs to break ties. However, while this gives us a total order, Lamport clocks alone do not precisely capture causal relationships.

Specifically, if we find $L(a) < L(b)$ we only know that a could have occurred before b, but not that a caused b. This is because the tie-breaking process ID has no causal significance. Therefore, it is possible that $a||b$ (they are concurrent), even though a is ordered before b in the total order.

Happens-Before and Lamport Clocks (contd.)

Let's look at this example. Pay attention to the *timestamps* and not the placement of events in the image:



The order is $(1, A), (1, B), (2, A), (3, C), (4, C)$, but $a||b$ thus we cannot conclude that $a \rightarrow b$ based on this scheme.

Happens-Before and Lamport Clocks (contd.)

Remember if $L(a) < L(b)$ we cannot conclude that $a \rightarrow b$.

Thus the confusion is that we cannot tell if a actually happens before b or if a and b are concurrent!

The only cases where we can automatically conclude that $a \rightarrow b$ is when a happens before b in the same process, or a is an event sending a message to another node and b is the event receiving that message.

Checkpoint

b has no dependency on a and a has no dependency on b

We have seen physical clocks which measure in seconds. They suffer from clock skew and there are still inaccuracies even if the clocks are in sync.

Logical clocks are better. Lamport clocks provide a partial ordering over the events. We can modify the algorithm to give a full ordering. This is great, but if $L(a) < L(b)$ we do not know if it is because $a \rightarrow b$ or because $a||b$. We only know that a is before b *in this ordering*.

So we can have a partial order and a full order, but still cannot conclude a causal order.

Vector Clock

$L \leq V$

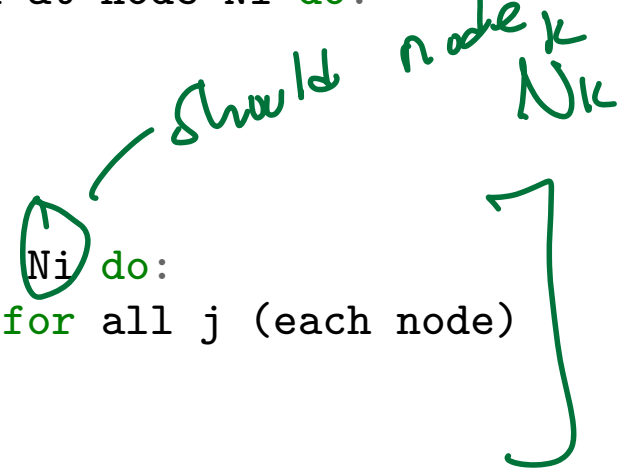
Given Lamport timestamps such that $L(a) < L(b)$ we cannot tell if $a \rightarrow b$ or $a||b$.

To detect which events are concurrent we use a **vector clock**:

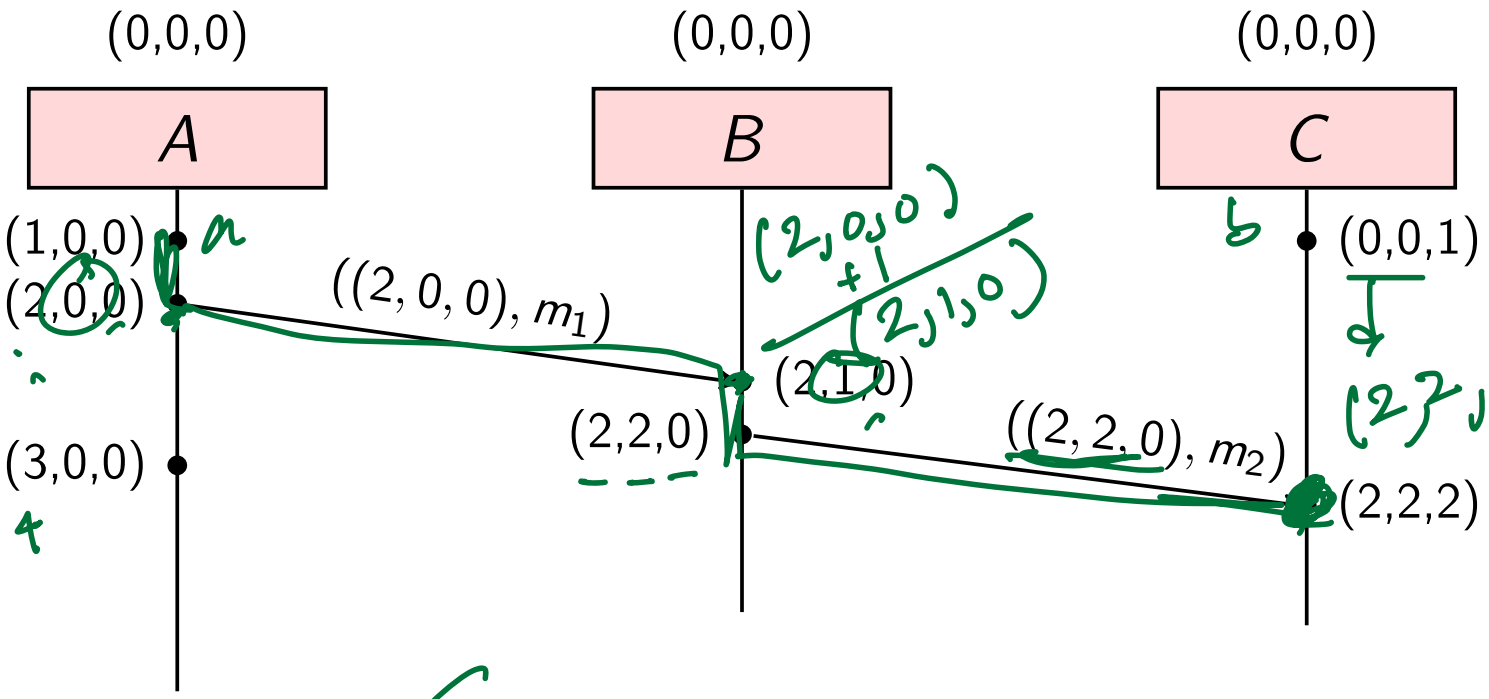
- There are n nodes in the system $N = (N_1, \ldots N_n)$.
- The vector timestamp of event a is $V(a) = (t_1, \ldots, t_n)$.
- t_i is the number of events observed by node N_i.
- Each node has a vector timestamp T.
- On an event at node N_i, increment $T(i)$.
- Attach T to every message.
- Recipient merges T into its own local count vector.

Vector Clock (contd.)

```
on initialization at node Ni do:
    T := (0, 0, ..., 0)
end on

on any event occurring at node Ni do:
    T[i] := T[i] + 1
end on

on request to send message m at node Ni do:
    T[i] := T[i] + 1
    send(T, m)
end on

on receiving (T', m) at node Ni do:
    T[j] = max(T[j], T'[j]) for all j (each node)
    T[i] = T[i] + 1
end on
```



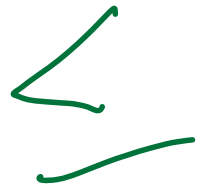
Vector Clock (contd.)



Vector Clock (contd.)

So, how do we *causally* order events? $a \rightarrow b$ iff:

1. Every counter in T_1 is less than or equal to the corresponding counter in T_2.
2. At least one counter in T_1 is strictly less than its corresponding counter in T_2.



Vector Clock (contd.)

Two events a and b are concurrect, $a||b$ iff:

- $a \nrightarrow b$ and $b \nrightarrow a$ where $a \neq b$
- Or in English, when there exists i such that $T_1[i] > T_2[i]$ and there exists j such that $T_2[j] > T_1[j]$.

For example $V(a) = (2, 2, 0)$ and $V(b) = (0, 0, 1)$ are incomparable and $a \nrightarrow b$ and $b \nrightarrow a$ thus $a||b$.

Vector Clock (contd.)

As we move through this process, some of the counters in the vectors should increase. That is, if $a \rightarrow b$ then $V(a) \leq V(b)$.

With Lamport, $V(a) \leq V(b), a \neq b$ could mean that $a \rightarrow b$ or $a||b$, but with Vector, the concurrency possibility is eliminated because of the way we update the vectors.

$$a||b \iff V(a) \not\leq V(b), V(b) \not\leq V(a)$$

Thus, $V(a) \leq V(b) \implies a \rightarrow b$.

monotic

Vector Clock (contd.)

Some definitions:

- $T = T'$ iff $T[i] = T'[i]$ for all $i \in \{1, \ldots, n\}$
- $T \leq T'$ iff $T[i] \leq T'[i]$ for all $i \in \{1, \ldots, n\}$
- $T < T'$ iff $T[i] < T'[i]$, $T \neq T'$ for all $i \in \{1, \ldots, n\}$
- $T || T'$ iff $T \not\leq T'$, $T' \not\leq T$

Which gives us these properties:

- $(V(a) < V(b)) \iff (a \rightarrow b)$
- $(V(a) = V(b)) \iff (a = b)$
- $(V(a) || V(b)) \iff (a || b)$

Checkpoint

So, Lamport clocks gave us a partial or total order over the events. If that is all we care about, we can use a Lamport clock. But, if $L(a) < L(b)$ in an ordering, it could be that $a \rightarrow b$ or $a||b$.

With Vector clocks, $V(a) \leq V(b) \implies a \rightarrow b$ and we can identify cases where $a||b$. **Thus, the ordering is causal.**

What Is The Point?

Timing and synchronization are extremely important in distributed systems. The goal is not to *predict* an ordering of events but to enable processes to understand how many events have causally preceded each one every time it receives a message (Lamport).

With Vector clocks, processes can also determine how many events in **other** processes have preceded the current event.

What Is The Point? (contd.)

This creates an important capability: we can ensure that a message is delivered only if all of the messages that causally precede it have also been received. This allows us to visualize a workflow with dependencies without needing a centralized coordinating process.

This allows us to visualize a workflow with dependencies without having a centralized coordinating process.