COEN6731 Distributed Software Systems

Week 1: Introductions and fundamentals

Gengrui (Edward) Zhang, PhD
Web: gengruizhang.com

Who am I?

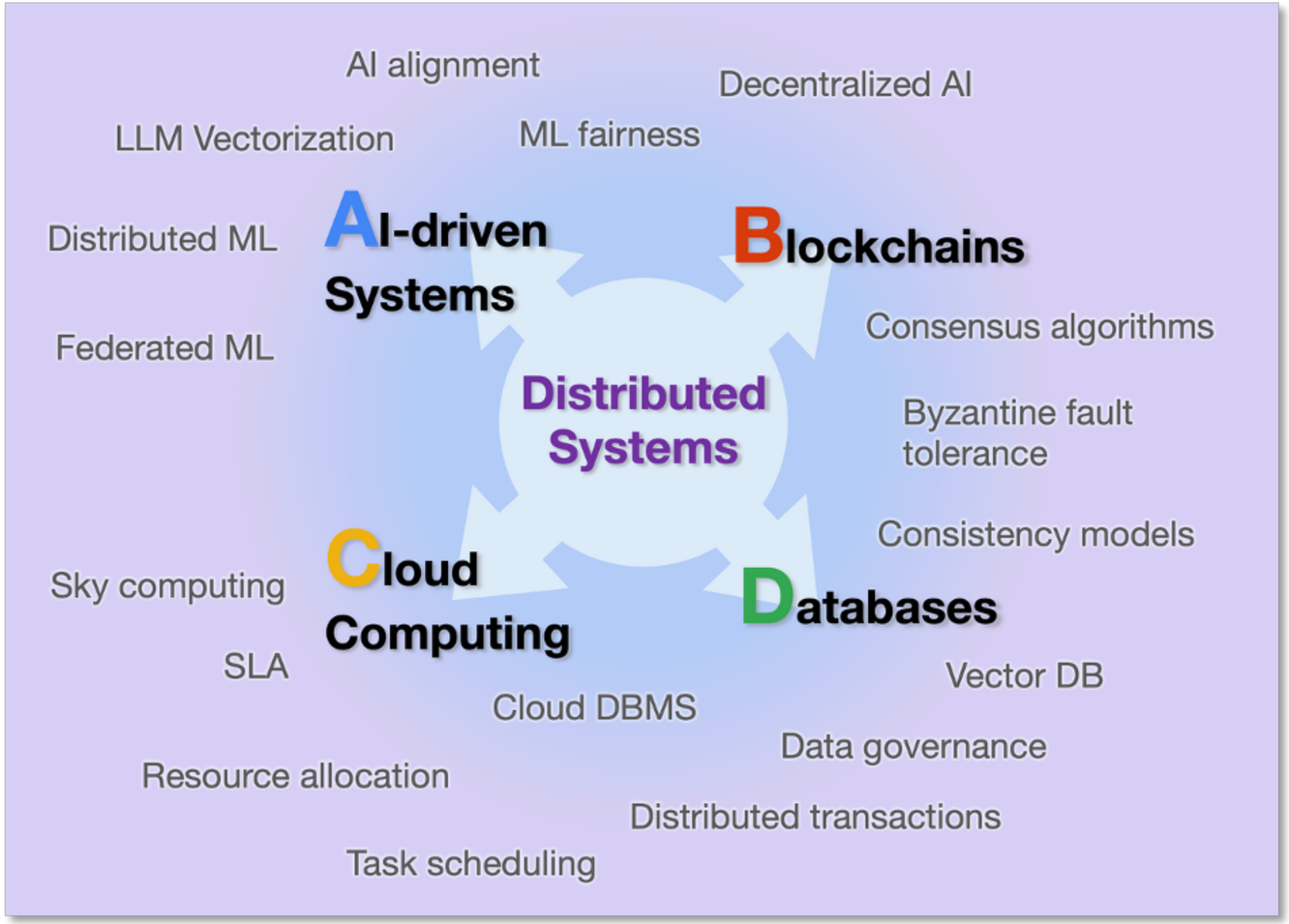
https://www.gengruizhang.com/

My research is rooted in **distributed systems**, powering and advancing **A**I, **B**lockchains, **C**loud computing, and **D**atabase systems

- AI/Agentic systems
- Blockchains
- Cloud computing
- Distributed databases

Who am I?

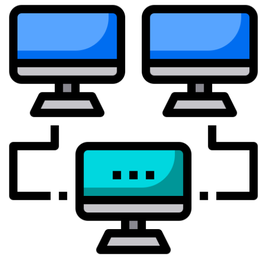
https://www.gengruizhang.com/



How about you?

Operating systems

Databases



Computer networking

Course outline

- From Week 1 to Week 6, we will learn some fundamental concepts and designs of distributed systems. We will especially focus on **consensus algorithms** and **fault tolerance**, which pillars blockchain applications
- Starting Week 6, we will discuss research papers. Each lecture will include two or three papers followed by discussion.
- Around Week 6 (subject to change), course projects will start with initial proposal presentations. There will also be midterm progress presentations and final project presentations with demos.
- This course is a discussion-based course! **Participation is important! Questions are always welcome!** You will learn system design trade-offs through the discussions!

Grading scheme

- **Presentations and deliverables (30%)**
  - In-class presentation
  - Slides
- **Participation and discussion (10%)**
- **Course project (60%)**
  - Proposal
  - Progress report
  - Final project presentation and demo
  - Final report

Why read papers?

Reading papers is a skill

Get research ideas

Employ the state of the art

Presentation guidelines

- You may team up with another student (optional)
- Pick a paper from the paper list (appeared on course weekly schedule)

Presentation guidelines

- You may team up with another student (optional)
- Pick a paper from the paper list (appeared on course weekly schedule)
- Each presentation should be around 30-35 minutes (20-25 slides), followed by 20-30 minutes of Q&A
  - All students are expected to participate in the Q&A (participation and discussion)

Presentation guidelines

- You may team up with another student (optional)
- Pick a paper from the paper list (appeared on course weekly schedule)
- Each presentation should be around 30-35 minutes (20-25 slides), followed by 20-30 minutes of Q&A
  - All students are expected to participate in the Q&A (participation and discussion)
- You **must upload your slides** on Moodle at least 24 hours before your presentation. Your slides will be made available to the class

Presentation guidelines

- Introduce background and motivation: **why the problem is important?**

Presentation guidelines

- Introduce background and motivation: **why the problem is important?**
- Present the design
  - Design overview: provide a **high-level summary** of proposed approach
  - Design details: dive **deeper into the specific mechanisms**
  - **Use at least two examples** to illustrate how the proposed approach works

Presentation guidelines

- Introduce background and motivation: **why the problem is important?**
- Present the design
  - Design overview: provide a **high-level summary** of proposed approach
  - Design details: dive **deeper into the specific mechanisms**
  - **Use at least two examples** to illustrate how the proposed approach works
- Discuss implementation and evaluation
  - Experimental setup: what's the environment, testbed, workloads, benchmark, etc?
  - Results: summarize key findings

Presentation guidelines

- Introduce background and motivation: **why the problem is important?**
- Present the design
  - Design overview: provide a **high-level summary** of proposed approach
  - Design details: dive **deeper into the specific mechanisms**
  - **Use at least two examples** to illustrate how the proposed approach works
- Discuss implementation and evaluation
  - Experimental setup: what's the environment, testbed, workloads, benchmark, etc?
  - Results: summarize key findings
- Discuss **at least three strong and weak points** of the paper

Course project

- Suggested topics will be provided
- Four major components:
  - Proposal (2 pages max, including references) + Presentation (5 minutes)
    - **Tentative schedule: Week 5**
  - Progress report (Presentation)
  - Final presentation with demos
  - Submit final project report (6 pages max, including references)

Today's Outline

DS Fundamentals
DS Example: Key-value store
Time in DS

Today's Outline

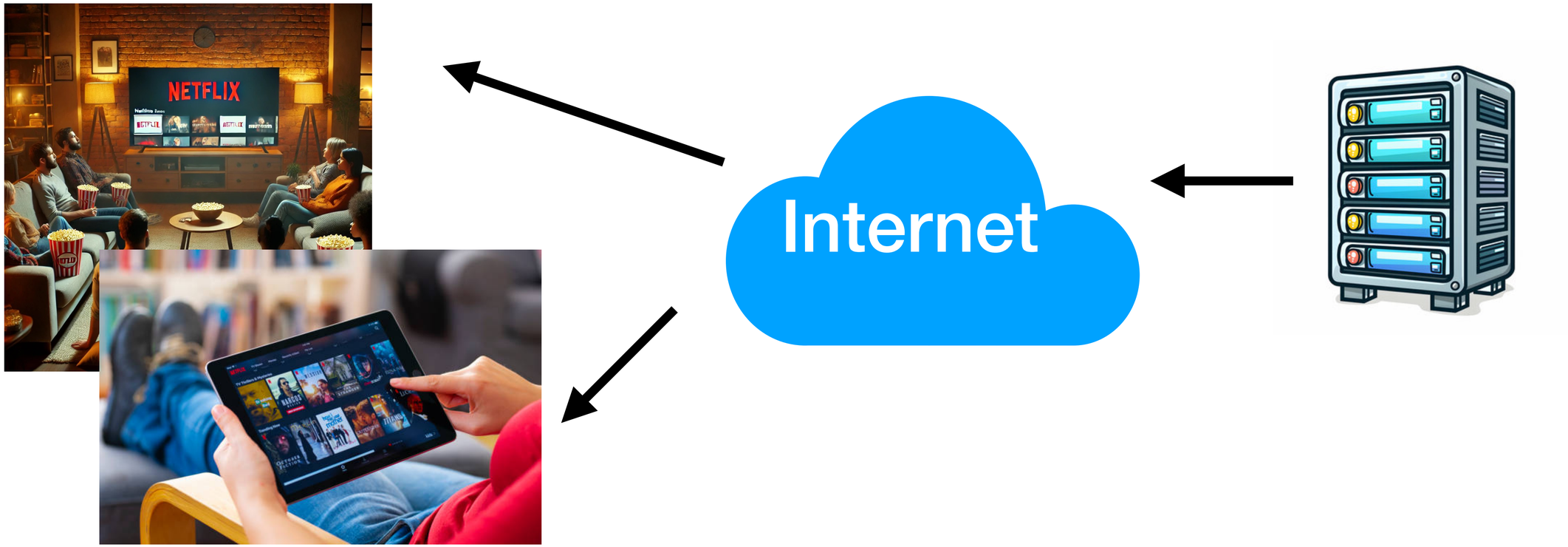
DS Fundamentals

DS Example: Key-value store

Time in DS

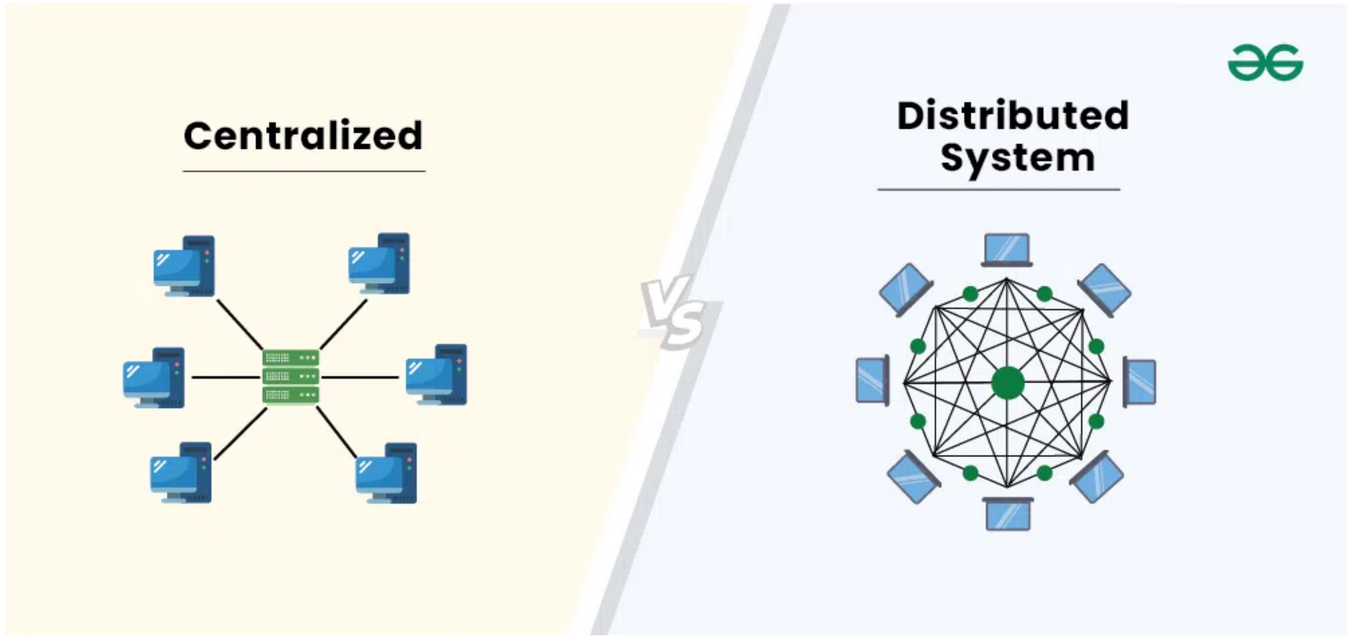
Why build a distributed system?

- Consider watching Netflix
  - If a server goes down, what should we expect?



Why build a distributed system?

- Centralized system is simpler in all regards
  - Local memory, storage
  - Failure model
  - Maintenance
  - Data security



Why build a distributed system?

- But …
  - Vertical scaling costs more than horizontal scaling
  - Availability and redundancy
  - Single point of failure
  - Many resources are inherently distributed
  - Many resources used in a shared fashion

Distributed vs. Parallel systems

- **Parallel systems/computing**
- Multiple processors/cores work on different parts of the same task simultaneously. These processors are usually part of a single machine or a tightly-coupled cluster.
  - **Shared memory** or tightly-coupled processors
- E.g, high-performance computing (HPC), and graphics processing (e.g., GPUs)

Distributed vs. Parallel systems

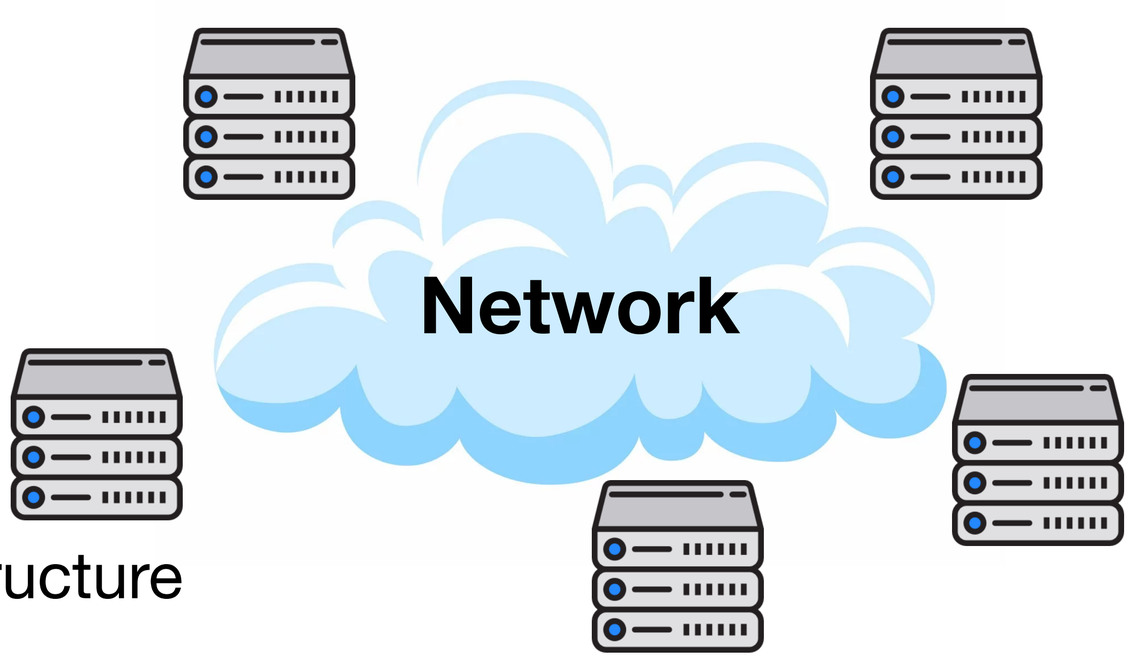
- **Parallel systems/computing**
- Multiple processors/cores work on different parts of the same task simultaneously. These processors are usually part of a single machine or a tightly-coupled cluster.
  - **Shared memory** or tightly-coupled processors
- E.g, high-performance computing (HPC), and graphics processing (e.g., GPUs)

- **Distributed systems/computing**
- Multiple **independent machines** (nodes) work together, connected by a network and can be geographically dispersed
  - Independent machines (nodes) communicate over **message passing**
- E.g., cloud computing, blockchains, and distributed databases

Distributed system definitions & views

- A distributed system is a system that is comprised of several **physically disjoint compute resources** interconnected by a **network**

Map Reduce
(Hadoop)



Peer-to-peer
(Bitcoin, BitTorrent)

Google infrastructure
(BigTable)

World Wide Web

Other definitions & views

- A distributed system is one in which **hardware** or **software components** located at **networked computers communicate** and **coordinate** their actions only by **passing messages**

  — *By Coulouris et al.*

- A distributed system is a **collection of independent computers** that appears to its users as **a single coherent system**

  — *By Tanenbaum & van Steen.*

Focus on practical perspective; we do not need rigorously formalized definitions or binary precisions

Terminology

- Strive to use the term ***node*** or ***server*** to refer to a physically separable computing node in our systems

Terminology

- Strive to use the term ***node*** or ***server*** to refer to a physically separable computing node in our systems
- Other often synonymously used terms
  - Process, client (?), server, machine, container, …

Terminology

- Strive to use the term ***node*** or ***server*** to refer to a physically separable computing node in our systems
- Other often synonymously used terms
  - Process, client (?), server, machine, container, …
- Strive to use the term **message** or **value** to refer to the unit of communication among nodes

Terminology

- Strive to use the term ***node*** or ***server*** to refer to a physically separable computing node in our systems
- Other often synonymously used terms
  - Process, client (?), server, machine, container, …
- Strive to use the term **message** or **value** to refer to the unit of communication among nodes
- Other often synonymously used terms
  - Packet(s), communication, data, RPC, …

Terminology

- Strive to use the term ***node*** or ***server*** to refer to a physically separable computing node in our systems
- Other often synonymously used terms
  - Process, client (?), server, machine, container, …
- Strive to use the term **message** or **value** to refer to the unit of communication among nodes
- Other often synonymously used terms
  - Packet(s), communication, data, RPC, …
- It is not just us, it’s the literature and who you talk to

Characteristics of distributed systems

- Reliable
- Fault-tolerant
- Highly available
- Recoverable
- Consistent
- Scalable
- Predictable performance
- Secure

*Many of the characteristics still pose **significant challenges** in theory and practice!*

Reliability

- Reliability means that a system can continue to operate **correctly over time** without **unexpected failures**. It measures how long a system can **consistently perform** its intended functions.
  - Rarely fails
  - When it does fail, it does so gracefully (doesn't lose data or crash unexpectedly)

Reliability

- Reliability means that a system can continue to operate **correctly over time** without **unexpected failures**. It measures how long a system can **consistently perform** its intended functions.
  - Rarely fails
  - When it does fail, it does so gracefully (doesn't lose data or crash unexpectedly)
- Reliability = **No data loss** or **corruption**

Availability

- Availability means that a system is **accessible and operational** when you need it. A system is considered **available** if users can successfully make requests and get responses within the expected time.

Availability

- Availability means that a system is **accessible and operational** when you need it. A system is considered **available** if users can successfully make requests and get responses within the expected time.
- Examples:
  - A website is available if users can visit it and get a response.
  - If a server crashes but a backup takes over, the system remains available.

Availability

- Availability means that a system is **accessible and operational** when you need it. A system is considered **available** if users can successfully make requests and get responses within the expected time.
- Examples:
  - A website is available if users can visit it and get a response.
  - If a server crashes but a backup takes over, the system remains available.
- Availability is often expressed as a percentage of uptime:
  - $$\textit{Availability} = \frac{\textit{Uptime}}{\textit{Total time}} \times 100$$

Availability

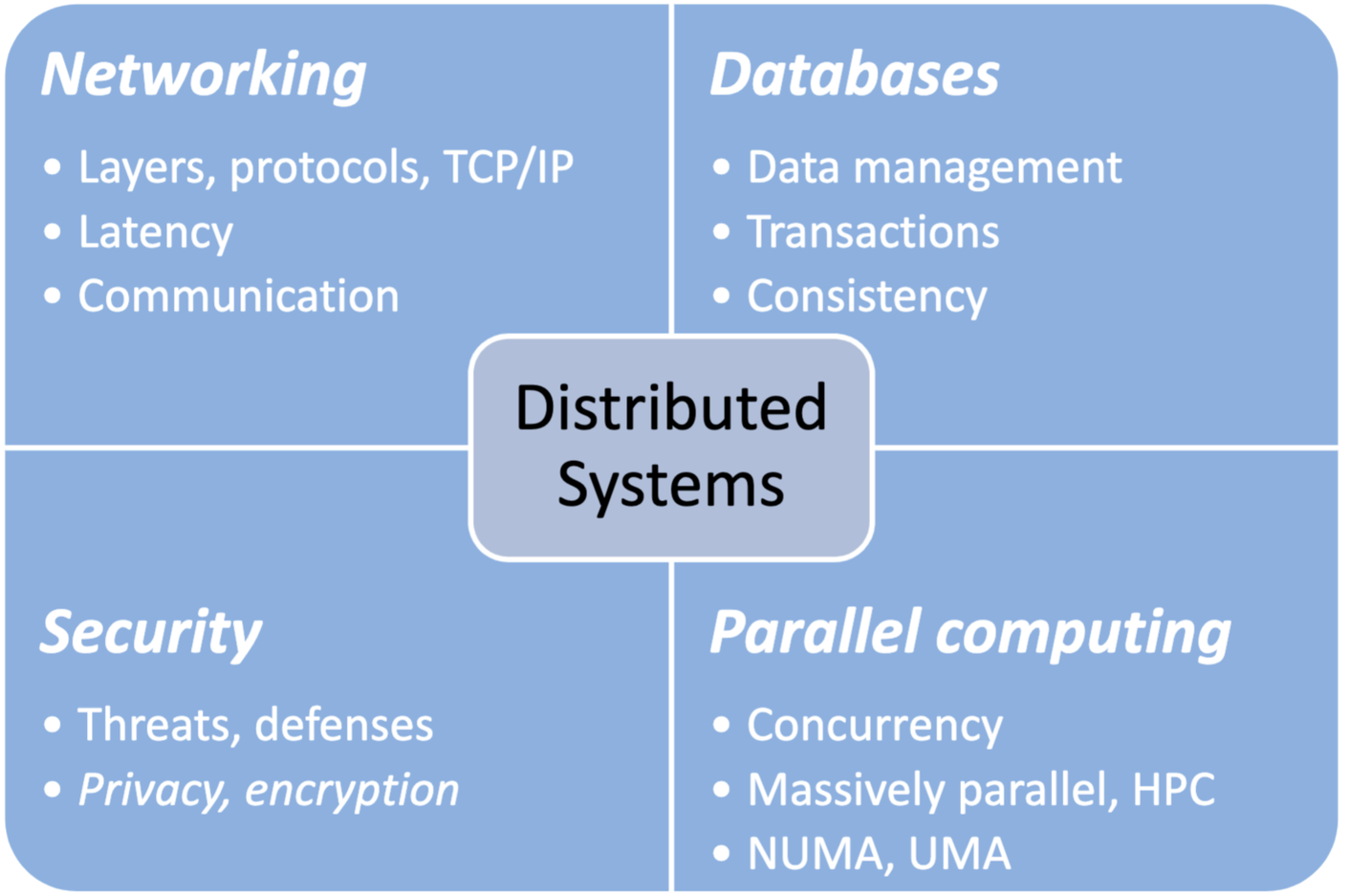
- Availability means that a system is **accessible and operational** when you need it. A system is considered **available** if users can successfully make requests and get responses within the expected time.
- Examples:
  - A website is available if users can visit it and get a response.
  - If a server crashes but a backup takes over, the system remains available.
- Availability is often expressed as a percentage of uptime:
  - $$Availability = \frac{Uptime}{Total\ time} \times 100$$

***99.9%** available /year -> down time: 8.76 hours*

***99.99%** available /year -> down time: 52.6 minutes*

***99.999%** available /year -> down time: 5.26 minutes (Five nines)*

Related Disciplines



Today's Outline

DS Fundamentals

DS Example: Key-value store

Time in DS

Distributed systems by examples

Massively scalable key-value stores

Key-value store

- A **key-value store**, or **key-value database**, is a type of data storage system that organizes information as a collection of **key-value pairs**. Each **key** serves as a unique identifier, while the corresponding **value** contains the associated data or its location.

```
1 key_value_store = {}
2
3 key_value_store["name"] = "Edward"
4
5 print("Name:", key_value_store["name"])
```

Key-value store

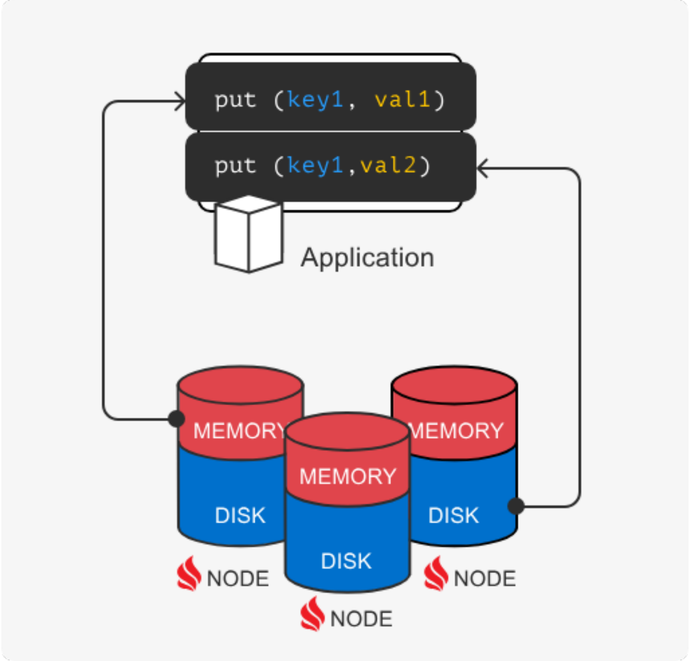
- A **key-value store**, or **key-value database**, is a type of data storage system that organizes information as a collection of **key-value pairs**. Each **key** serves as a unique identifier, while the corresponding **value** contains the associated data or its location.

```
key_value_store = {}

key_value_store["name"] = "Edward"

print("Name:", key_value_store["name"])
```

```
class Person:
    def __init__(self, name, title):
        self.name = name
        self.title = title

    def __str__(self):
        return f"Name: {self.name}, Title: {self.title}"

key_value_store = {}

key_value_store["person1"] = Person("Edward", "Professor")

print("Person1 details:", key_value_store["person1"])

print("Person1's name:", key_value_store["person1"].name)
print("Person1's title:", key_value_store["person1"].title)
```

What are key-value stores?

- Container for key-value pairs (databases)
- Distributed, multi-component, systems
- NoSQL semantics (non-relational)
- KV-stores offer **simpler query semantics** in exchange for i**ncreased scalability**, **speed availability**, and **flexibility**
- Data model not new



DBMS (SQL)

Students Table

Student	ID*
John Smith	084
Jane Bloggs	100
John Smith	182
Mark Antony	219

Activities Table

ID*	Activity*	Cost
084	Swimming	$17
084	Tennis	$36
100	Squash	$40
100	Swimming	$17
182	Tennis	$36
219	Golf	$47
219	Swimming	$15
219	Squash	$40

- Relational data schema
- Data types
- Foreign keys
- Full SQL support

Key-value store

Key	Value
John Smith	{Activity:Name= Swimming}
Jane Bloggs	{Activity:Cost=57}
Mark Anthony	{ID=219}

- No data schema
- Raw byte access
- No relations
- Single-row operations

Why are key-value stores needed?

- Today’s internet applications
  - Huge amounts of stored data
  - Huge number of Internet users
  - Frequent updates
  - Fast retrieval of information
  - Rapidly changing data definitions
- Ever more users, ever more data

Why are key-value stores needed?

- Horizontal scalability
  - User growth, traffic patterns change
  - Adapt to number of requests, data size
- Performance
  - High speed for single-record read and write operations
- Flexibility
  - Adapt to changing data definitions
- Reliability
- Availability and geo-distribution

Key-value store client interface

- Main operations
  - Write/update: **`put(key, value)`**
  - Read: **`get(key)`**
  - Delete: **`delete(key)`**

Key-value store client interface

- Main operations
  - Write/update: **`put(key, value)`**
  - Read: **`get(key)`**
  - Delete: **`delete(key)`**

- Usually no aggregation, no table joins, no transactions!
  - Some KV stores support transactions by implementing features traditionally found in relational databases, such as **atomic operations**, **isolation levels**, and **multi-version concurrency control (MVCC)**

Key-value store in practice

- BigTable by Google
- LevelDB by Google
- RocksDB by Meta
- Apache HBase
- Apache Cassandra
- Redis
- Amazon Dynamo
- Yahoo! PNUTS

Common elements of key-value stores

- Failure detection, failure recovery
  - Crash, omission, timing
- Replication
  - Store and manage multiple copies of data
- Memory store, write ahead log (WAL)
  - Keep data in memory for fast access
- Versioning (time)
  - Store different versions of data
  - Timestamping

Today's Outline

DS Fundamentals

DS Example: Key-value store

Time in DS

Time in distributed systems

- In distributed systems, we require:
  - **Coordination** between nodes: must **agree** on certain things
  - **High degree of parallelism**: nodes should work independently to make progress

Time in distributed systems

- In distributed systems, we require:
  - **Coordination** between nodes: must **agree** on certain things
  - **High degree of parallelism**: nodes should work independently to make progress
- Time gives us:
  - **Point of reference** every machine knows how to keep track of
  - Without need for explicit communication

Why time is important? (Practically speaking)

- Distributed gaming — who grabbed an object first?
- Markets, auctions, trading — who issued order first?
- Multimedia synchronization for real-time teleconferencing
- Target tracking, air traffic control, location positioning

Time tells us the order of events

Ordering observation

1. If two events occurred at the same process p_i $(i = 1,2,\cdots,N)$, then they occurred in the order in which p_i observes them; this order is denoted by the symbol $\rightarrow_i$
   - E.g., $a \rightarrow_i b$, meaning p_i observes that a happened before b
2. Whenever a message is sent between processes, the event of sending the message occurred before the event of receiving the message

Happend-before vs. concurrent

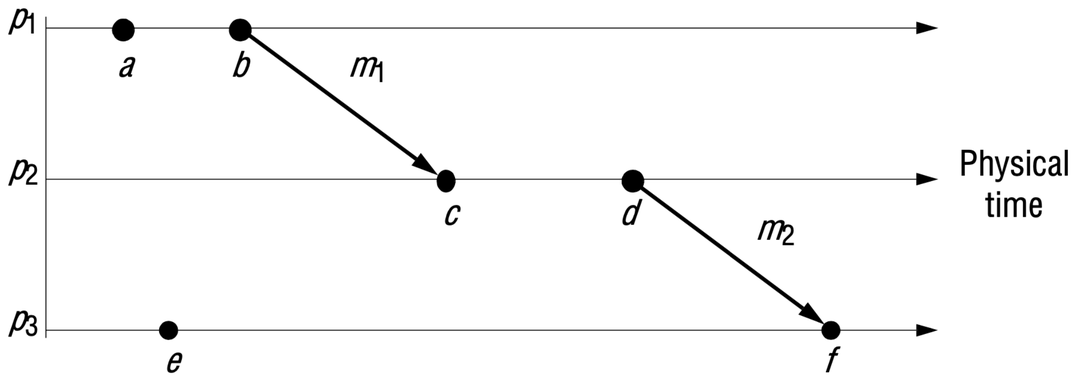
From the ordering observation, we can generalize **happend-before relation**, denoted by $\rightarrow$, as follows:

1. If $\exists$ process $p_i : a \rightarrow_i b$, then $a \rightarrow b$
2. For any message m, $send(m) \rightarrow receive(m)$
3. If a, b, and c are events such that $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$

Happend-before vs. concurrent

From the ordering observation, we can generalize **happend-before relation**, denoted by $\rightarrow$, as follows:

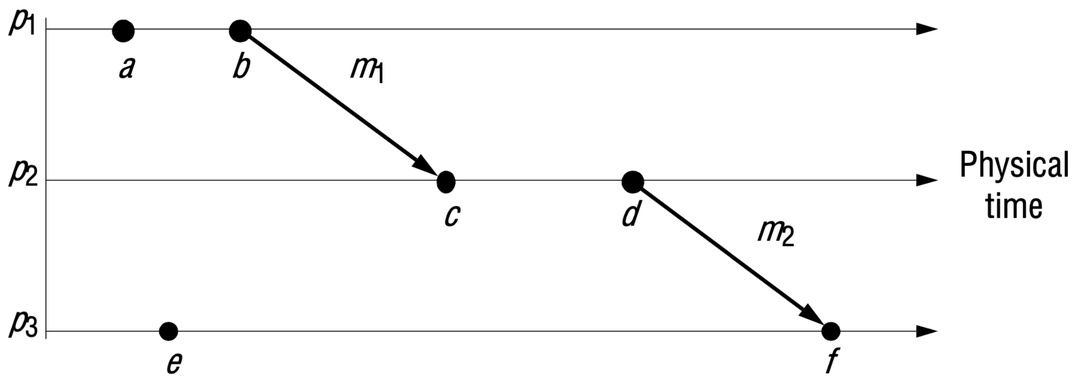
1. If $\exists$ process $p_i : a \rightarrow_i b$, then $a \rightarrow b$
2. For any message m, $send(m) \rightarrow receive(m)$
3. If a, b, and c are events such that $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$



Happend-before vs. concurrent

From the ordering observation, we can generalize **happend-before relation**, denoted by $\rightarrow$, as follows:

1. If $\exists$ process $p_i : a \rightarrow_i b$, then $a \rightarrow b$
2. For any message m, $send(m) \rightarrow receive(m)$
3. If a, b, and c are events such that $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$



a and e occur at different processes and there is no chain of messages intervening between them. We say that such as a and e that are not ordered by $\rightarrow$ are **concurrent** and write $a \,||\, e$

Logical clocks

- **Logical clocks**, invented by Lamport [1978], numerically capture happened-before relation. A logical clock, also called **Lamport clock**, is a **monotonically increasing software counter**.

1. L_i is incremented before each event is issued at process p_i : $L_i = L_i + 1$
2. Two substeps:
   2.1. When a process p_i sends a message m, it piggybacks on m the value $t = L_i$
   2.2. On receiving (m, t), a process p_j computes $L_j = \max(L_j, t)$ and then applies Step 1 before timestamping the event $receive(m)$.

Logical clocks

- **Logical clocks**, invented by Lamport [1978], numerically capture happened-before relation. A logical clock, also called **Lamport clock**, is a **monotonically increasing software counter**.

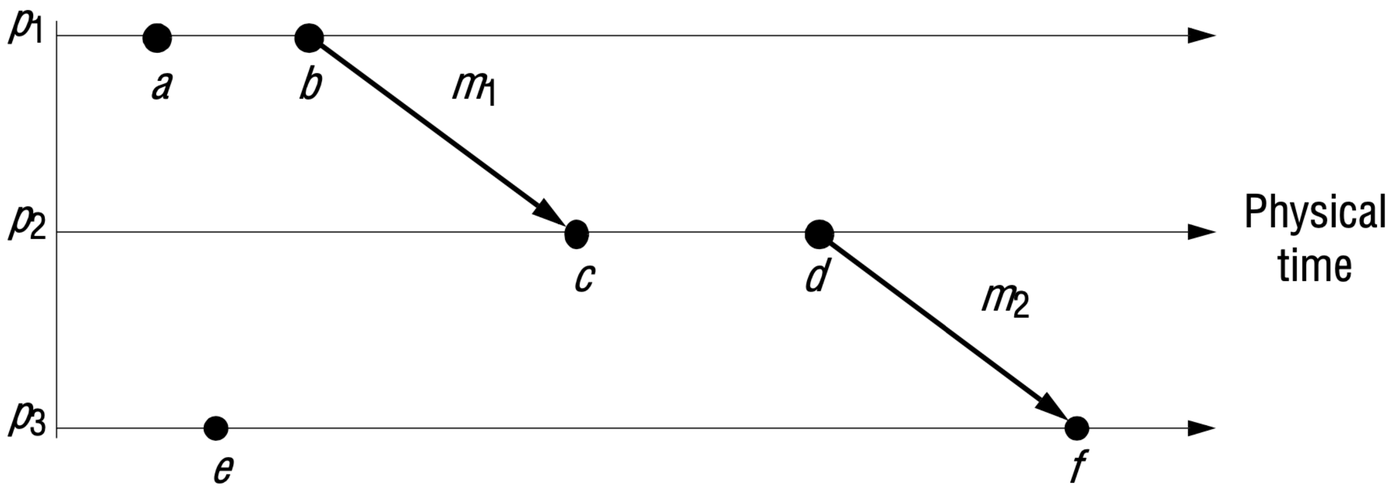
1. L_i is incremented before each event is issued at process p_i : $L_i = L_i + 1$
2. Two substeps:

   2.1. When a process p_i sends a message m, it piggybacks on m the value $t = L_i$

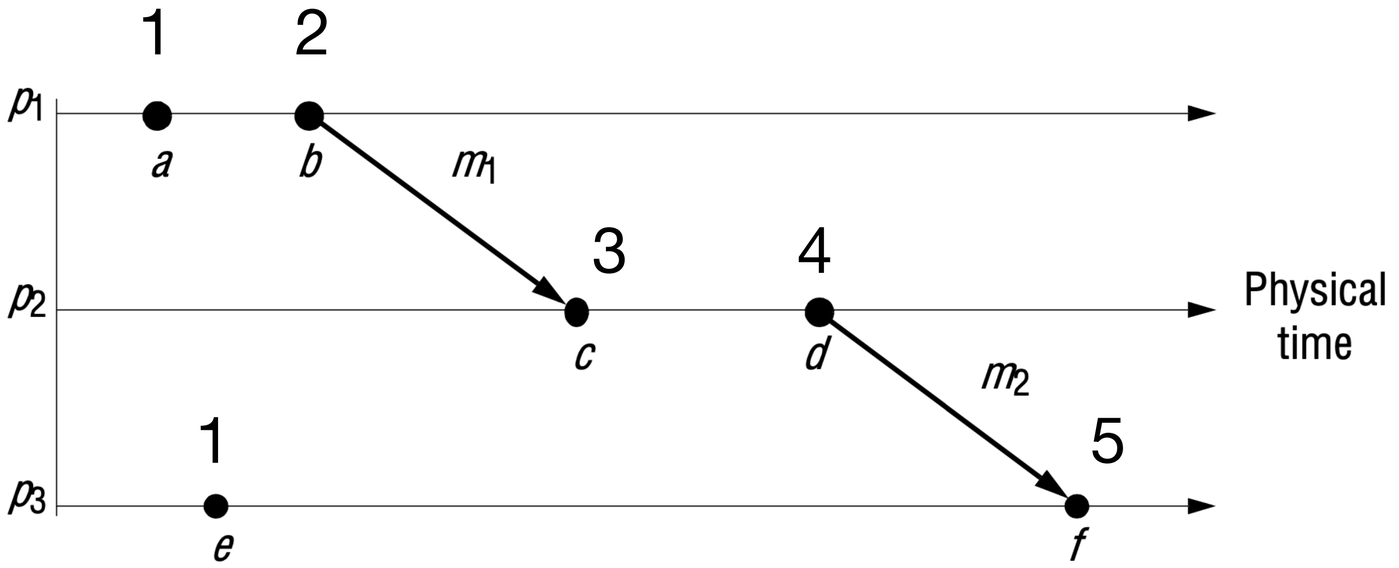
   2.2. On receiving (m, t), a process p_j computes $L_j = \max(L_j, t)$ and then applies Step 1 before timestamping the event $receive(m)$.

Can be any positive value

What are the Lamport timestamps for following events?



What are the Lamport timestamps for following events?



Logical clocks

- We can easily find that:
  - If $a \rightarrow b$, then $L(a) < L(b)$

Logical clocks

- We can easily find that:
  - If $a \rightarrow b$, then $L(a) < L(b)$
- Is the converse true?
  - If $L(a) < L(b)$, we cannot infer that $a \rightarrow b$
  - In our example, $L(b) > L(e)$, but $b \,||\, e$

Vector clocks

- Mattern [1989] and Fidge [1991] developed **vector clocks** to overcome the shortcoming of Lamport's clocks
- A vector clock for a system of N processes is an array of N integers

Vector clocks

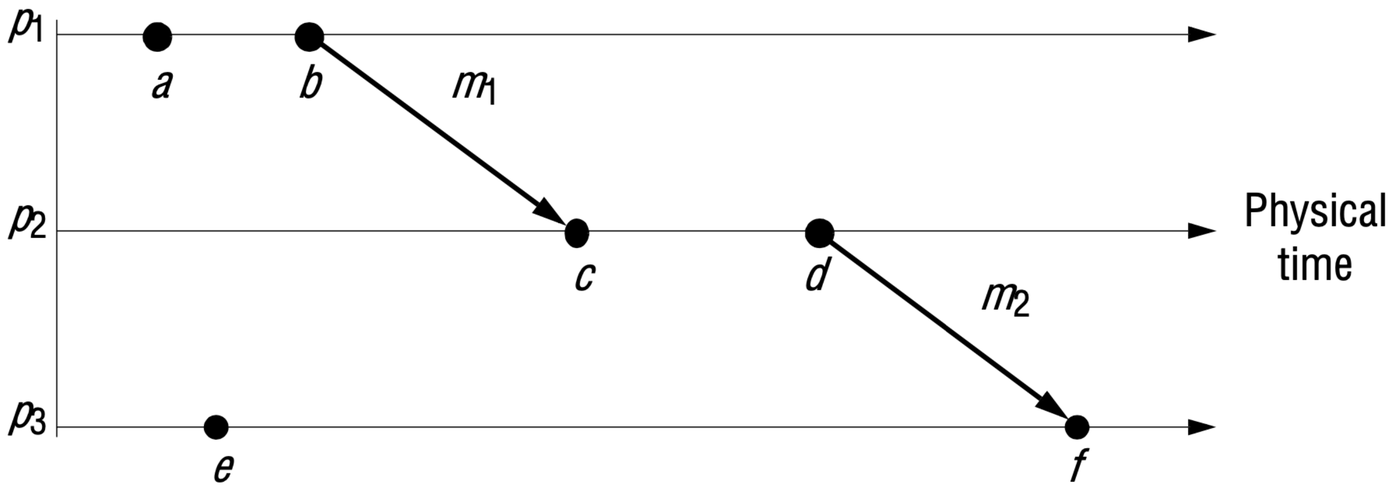
- Mattern [1989] and Fidge [1991] developed **vector clocks** to overcome the shortcoming of Lamport's clocks
- A vector clock for a system of N processes is an array of N integers
- Each process keeps its own vector clock, V_i, to timestamp local event.
  1. Initially, $V_i[j] = 0$, for $i, j = 1,2,\ldots,N$
  2. Just before p_i timestamps an event, it sets $V_i[i] = V_i[i] + 1$
  3. p_i includes the value $t = V_i$ in every message it sends
  4. When p_i receives a timestamp t in a message, it sets $V_i[j] = \max(V_i[j], t[j])$ for $j = 1,2,\ldots,N$

Vector clocks

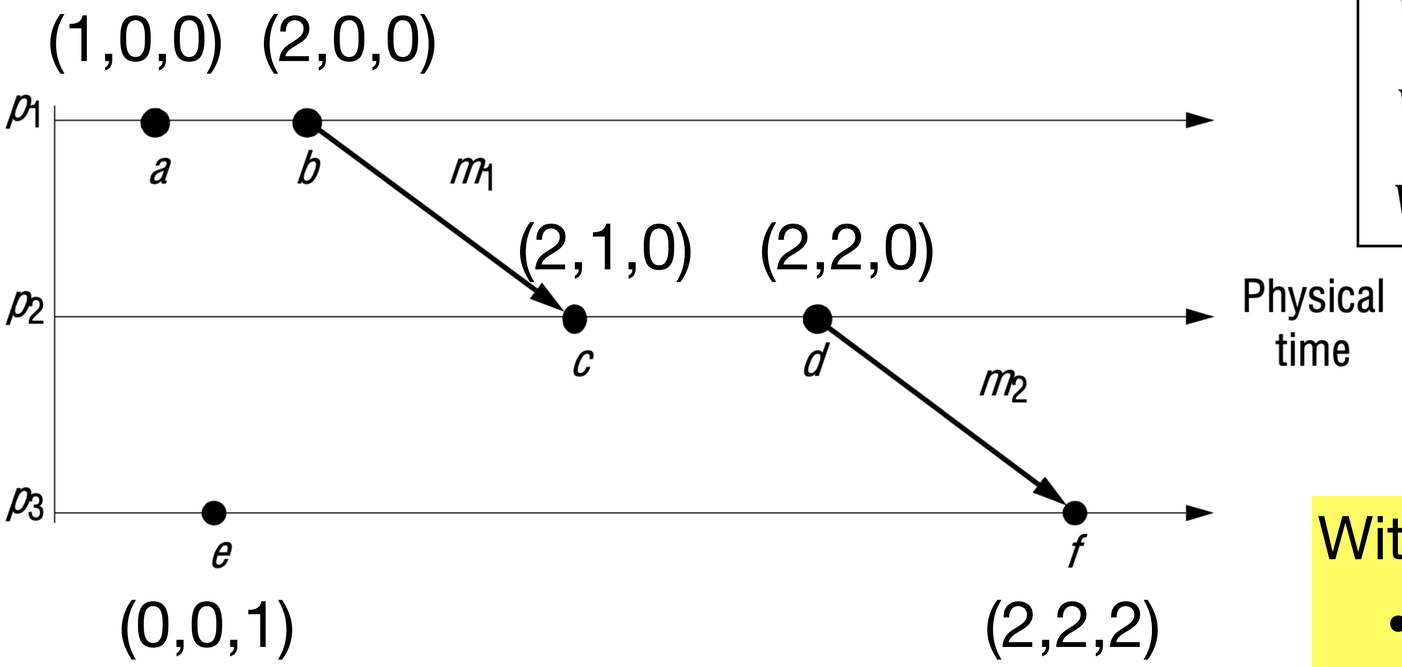
- Mattern [1989] and Fidge [1991] developed **vector clocks** to overcome the shortcoming of Lamport's clocks
- A vector clock for a system of N processes is an array of N integers
- Each process keeps its own vector clock, V_i, to timestamp local event.
  1. Initially, $V_i[j] = 0$, for $i, j = 1,2,\ldots,N$
  2. Just before p_i timestamps an event, it sets $V_i[i] = V_i[i] + 1$
  3. p_i includes the value $t = V_i$ in every message it sends
  4. When p_i receives a timestamp t in a message, it sets $V_i[j] = \max(V_i[j], t[j])$ for $j = 1,2,\ldots,N$

Also called merge operations

What are vector timestamps for following events?



What are vector timestamps for following events?



We may compare vector timestamps as follows:

$V = V'$ *iff* $V[j] = V'[j]$ for $j = 1, 2 \ldots, N$

$V \leq V'$ *iff* $V[j] \leq V'[j]$ for $j = 1, 2 \ldots, N$

$V < V'$ *iff* $V \leq V' \wedge V \neq V'$

With vector timestamps,

- If $a \rightarrow b$, then $V(a) < V(b)$
- If $V(a) < V(b)$, then $a \rightarrow b$

Worksheet