جامعة الامير سلطان
PRINCE SULTAN UNIVERSITY

REPLICATION & FAULT TOLERANCE

CS435 Distributed Systems

Basit Qureshi PhD, FHEA, SMIEEE, MACM
https://www.drbasit.org/

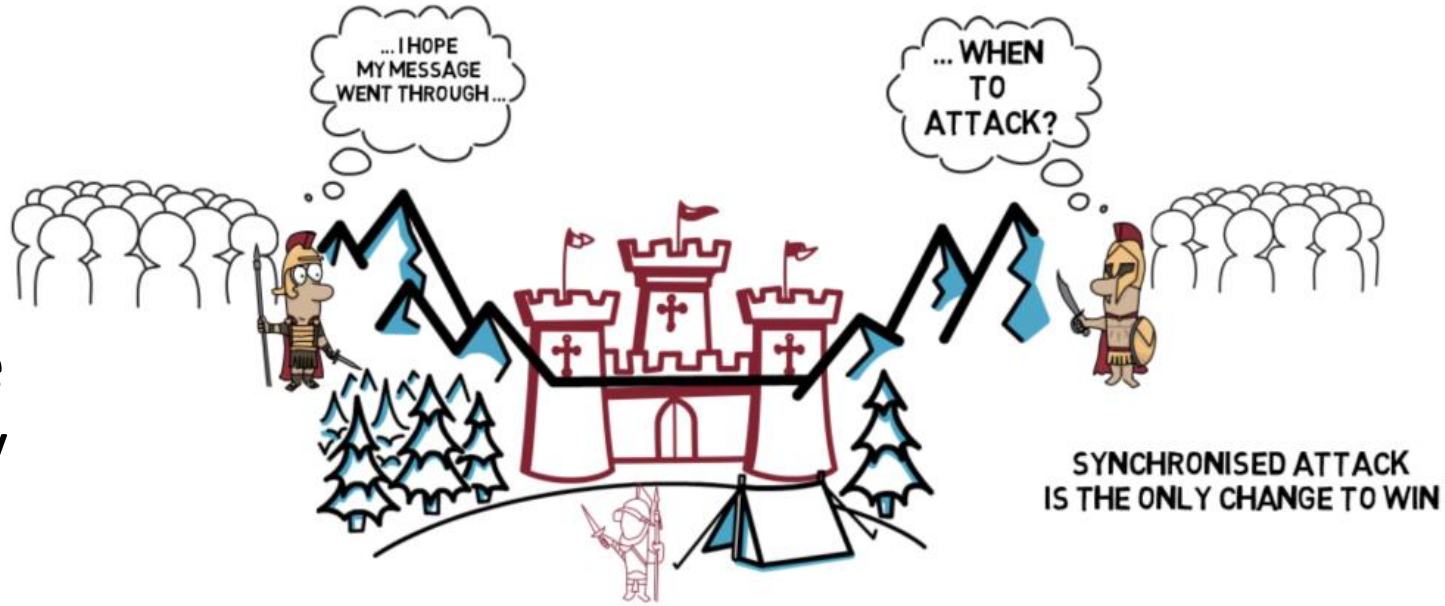
TOPICS

- The two generals problem
- Byzantine Generals problem
- Failure model
- Fault tolerance and availability
- Replication
- Ordering

THE 2 GENERALS PROBLEM

THE 2 GENERALS PROBLEM

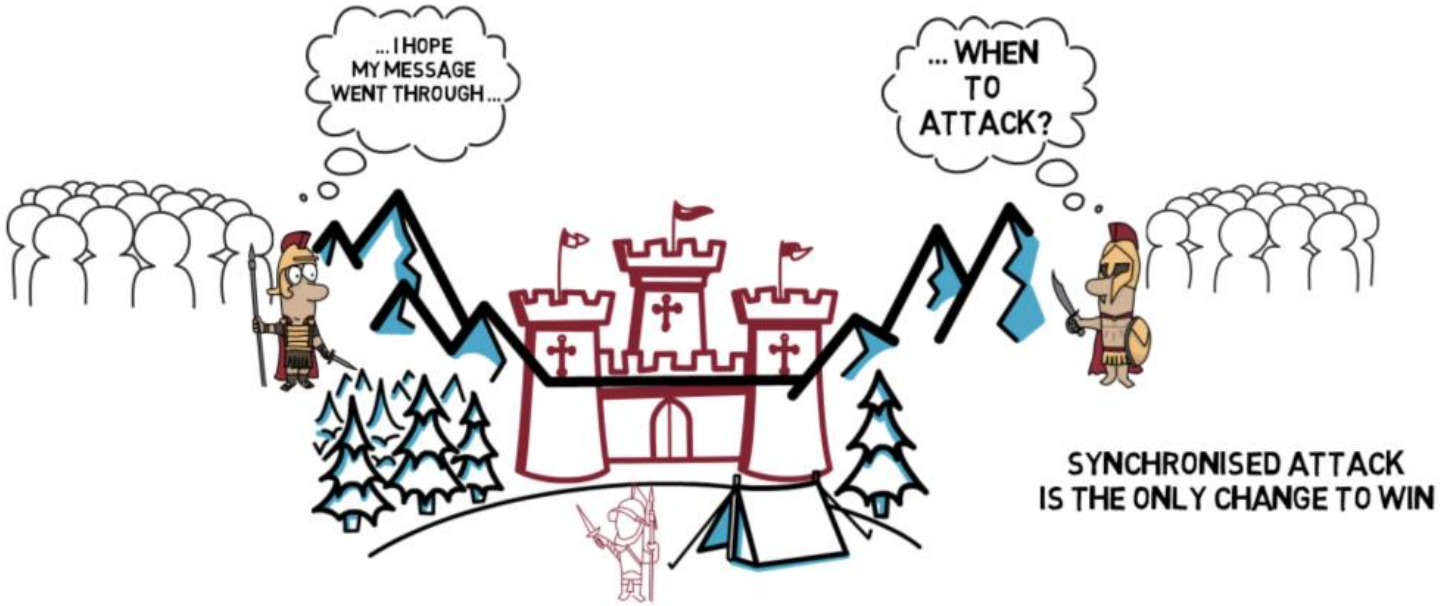
- The Two Generals' Problem is a thought experiment and theoretical problem in Dist. Systems.
- The city's defenses are strong, and if only one of the two armies attacks, the army will be defeated.
- However, if both armies attack at the same time, they will successfully capture the city.



https://finematics.com/two-generals-problem/

THE 2 GENERALS PROBLEM

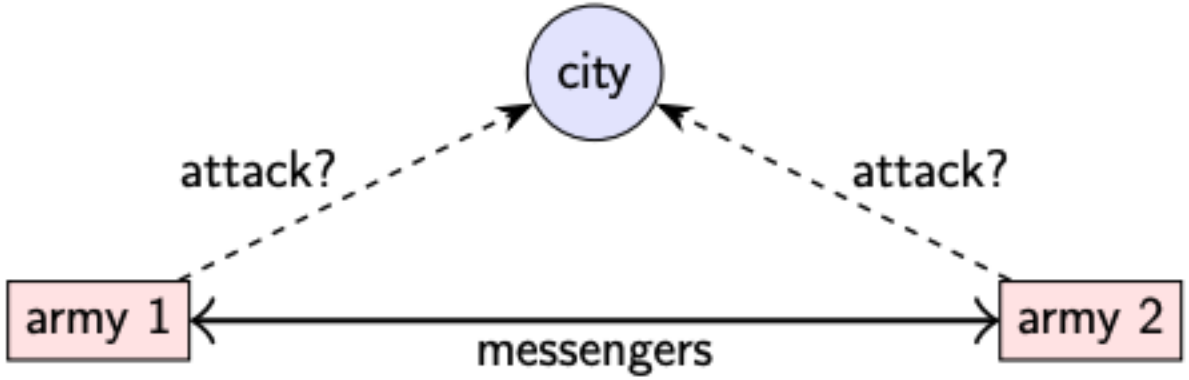
- Need to coordinate attack
- Communicate by sending a messenger through enemy territory.
- Agree on a time to launch an attack.
- Messenger could be captured!



https://finematics.com/two-generals-problem/

THE 2 GENERALS PROBLEM

- Communicate by sending a messenger through enemy territory.
- Agree on a time to launch an attack
- Messenger could be captured!

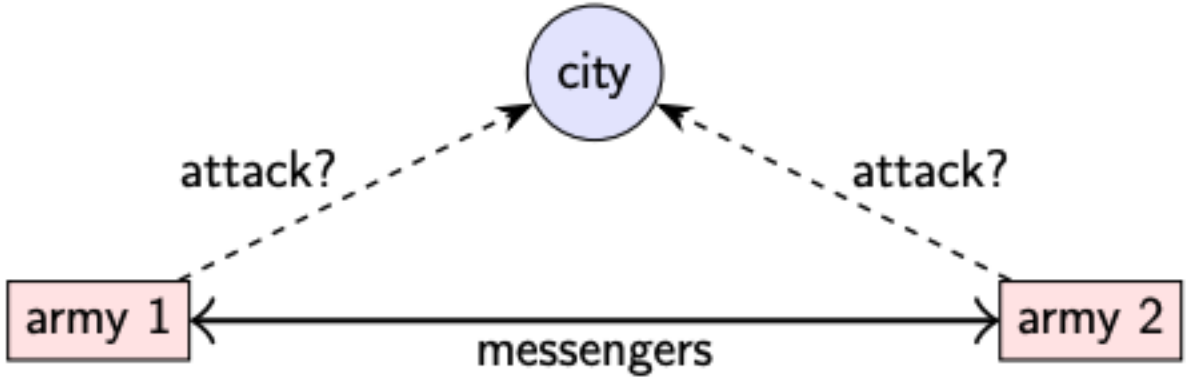


army 1	army 2	outcome
does not attack	does not attack	nothing happens
attacks	does not attack	army 1 defeated
does not attack	attacks	army 2 defeated
attacks	attacks	city captured

THE 2 GENERALS PROBLEM

PROBLEM

- General1 sends a message. Messenger is captured-> Message Not received.
- General1 sends a message. General2 receives the message. On the way back, the Messenger is captured -> Not received.
- Cannot confirm attack, unless messenger reaches General1.



army 1	army 2	outcome
does not attack	does not attack	nothing happens
attacks	does not attack	army 1 defeated
does not attack	attacks	army 2 defeated
attacks	attacks	city captured

THE 2 GENERALS PROBLEM

SOLUTION?

- OPTION 1: General 1 always attacks, even if no response is received?
  - Send lots of messengers to increase probability that one will get through
  - If all are captured, general 2 does not know about the attack, so general 1 loses
- OPTION 2: General 1 only attacks if positive response from general 2 is received?
  - Now general 1 is safe BUT general 2 knows that general 1 will only attack if general 2's response gets through
  - Now general 2 is in the same situation as general 1 in option 1

THE 2 GENERALS PROBLEM

The problem is that no matter how many messages are exchanged, neither general can ever be certain that the other army will also turn up at the same time.

Repeated sequence of back-and-forth acknowledgements can build up but the generals cannot reach certainty by exchanging any finite number of messages.

THE 2 GENERALS PROBLEM

An analogy: Ordering food using a food-delivery app

- Customer Orders food
- The bank charges payment
- The restaurant dispatches food

Restaurant	Bank	outcome
Doesnot dispatch food	Does not charge	Nothing delivered
Dispatches food	Does not charge	Restaurant looses money
Doesnot dispatch food	Charges	Customer complains
Dispatches food	Charges	Everyone is happy

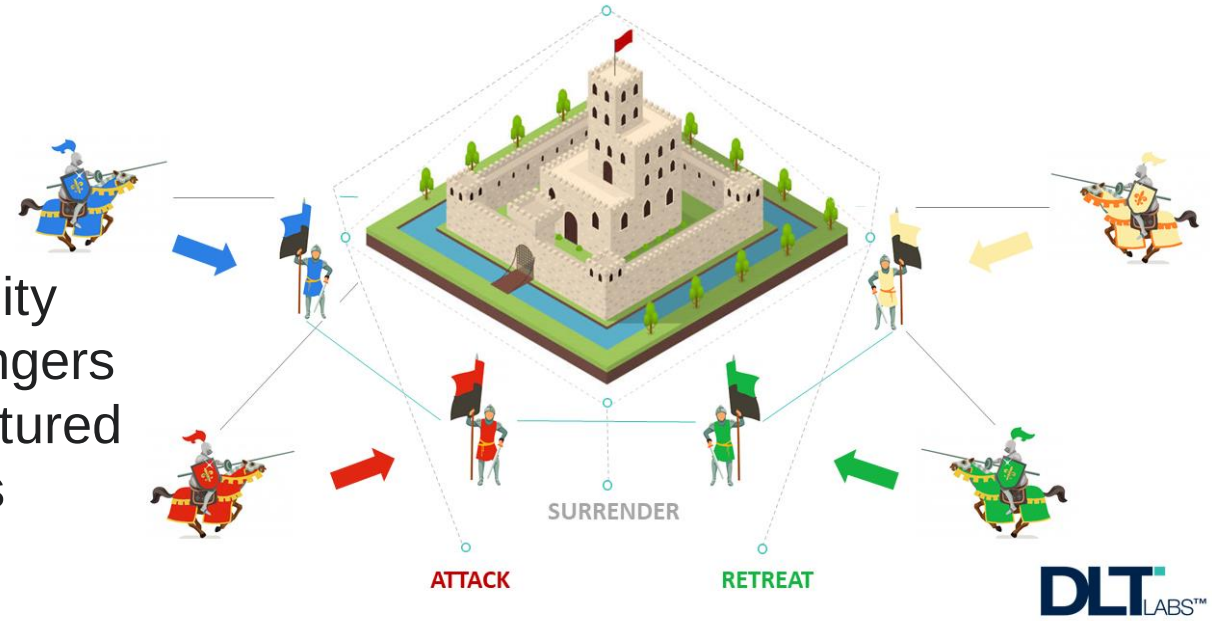
THE BYZANTINE GENERALS PROBLEM

THE BYZANTINE GENERALS PROBLEM

- A game theory problem: How to decentralized parties arrive at a consensus without a trusted central party?
- Similar to Two Generals' Problem

Differences

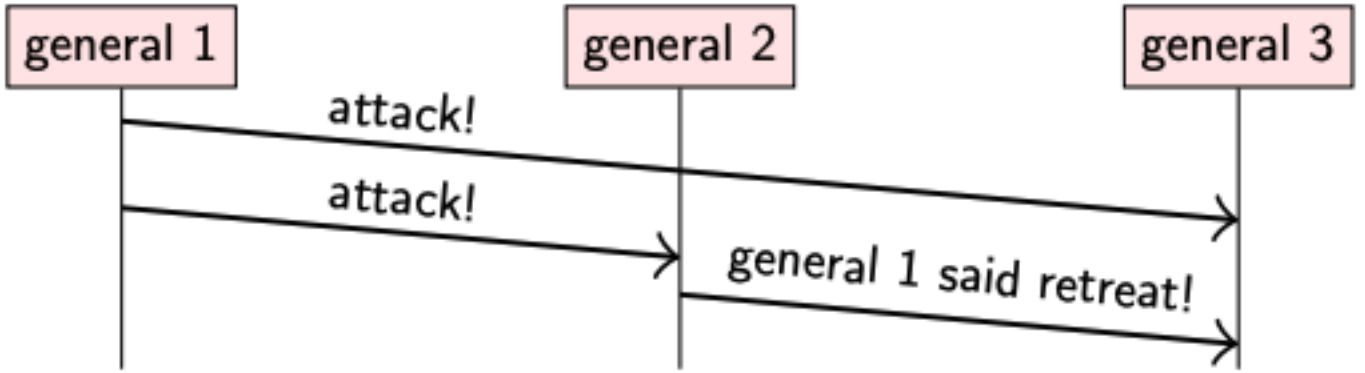
- 3 or more armies wanting to capture a city
- Generals communicate through messengers
- We assume messengers cannot be captured
- Problem: Some generals can be traitors



https://dltlabs.medium.com/the-byzantine-generals-problem-8552e24abe02

THE BYZANTINE GENERALS PROBLEM

- Generals behavior
  - A “honest” general colludes with other generals to attack the city
  - A “traitor” general deliberately misleads and confuses others
- 3 generals
  - Gen1 to Gen2 and Gen3: attack
  - Gen2 to Gen3: retreat!

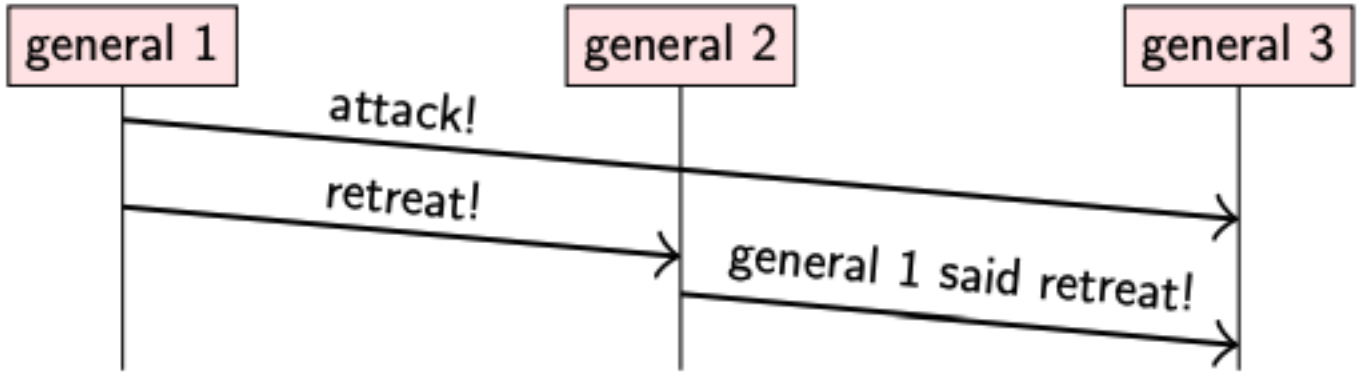


Gen2: Traitor

Gen3: Which message to trust?

THE BYZANTINE GENERALS PROBLEM

- Generals behavior
  - A “honest” general colludes with other generals to attack the city
  - A “traitor” general deliberately misleads and confuses others
- 3 generals
  - Gen1 to Gen3: attack
  - Gen1 to Gen2: retreat!
  - Gen2 to Gen3: retreat!



Gen1: Traitor
Gen3: Which message to trust?

THE BYZANTINE GENERALS PROBLEM

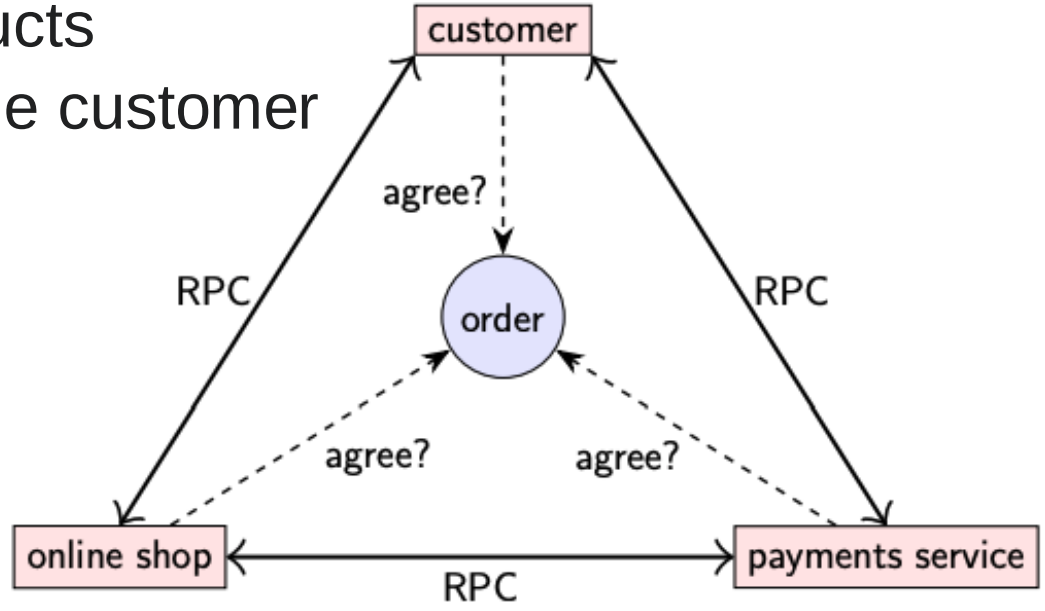
- Honest generals do not know which generals are traitors
- Traitor generals can collude to secretly coordinate actions
- Don’t know if honest generals are “honest”; they can be controlled by the adversary!
- So who to **Trust??**

THE BYZANTINE GENERALS PROBLEM

- In Dist Systems, there are **complex trust relationships**. To understand, lets use an analogy:
- Online shopping
  - Customer trusts Online shop and shares credit card information
  - Customer trusts Online shop to deliver items
  - Online shop trusts the payment service to complete the payment
  - Online shop trusts the delivery to deliver products
  - Payment service trusts the customer to pay dues
  - Payment service trusts the online shop to complete payments

THE BYZANTINE GENERALS PROBLEM

- Dis-trust?
- Online shopping
  - Customer uses stolen Credit cards to pay for Online shop
  - Customer suspects Online shop will deliver wrong items
  - Online shop payments are declined by the payment service
  - Delivery company does not deliver products
  - Payment service declines payment by the customer



THE BYZANTINE GENERALS PROBLEM

- In distributed systems, some systems explicitly deal with the possibility that some nodes may be controlled by a malicious actor, and such systems are called **Byzantine fault tolerant**.
- Popular with Blockchain and cryptocurrencies

FAILURE MODEL

FAILURE MODEL

- When designing a distributed algorithm, a system model is how we specify our assumptions about what faults may occur.

- Failure model:
  1. Network failure (e.g. loss etc)
  2. Node behavior (crashes, slow etc)
  3. Timing (e.g. latency, etc)

FAILURE MODEL

- Networks are NOT reliable
- Common problems:
  - Configuration errors
  - Shark bites! Line damage
  - Hardware failure
  - Intrusions
  - Power loss
  - Traffic spikes
  - Cellular (WAN) failure
  - Government restrictions (5G banned?)

FAILURE MODEL

- **Networks failure model**
- Communication modes: point-to-point, unicast, multicast, broadcast communication.
- Lets assume, we mostly use point-to-point communication between two nodes
  - **Reliable link**: Perfect links, messages are received 100% guaranteed.
  - **Fair-loss link**: Message may be lost, but can be duplicated, re-ordered. We keep re-trying until all messages eventually get through
  - **Arbitrary link**: A malicious adversary interferes with the messages (e.g. spoofing, replay etc).

We always assume that Network partitions can occur

FAILURE MODEL

- **Node behavior model**
- A node in a distributed system exhibits these behavior
  - **Crash-stop**: A node is faulty if it crashes. After crashing it stops executing forever.
  - **Crash-recovery**: A node may crash at any moment losing all of its memory. A restart is possible, however all memory operations are lost. Local disk storage survives the crash.
  - **Byzantine**: A node is faulty if it does not follow algorithm/rules (faulty, malicious).

A node that is not faulty is “Correct”

FAILURE MODEL

- **Timing (Synchrony) model**
- We assume one of the following for network and node behavior
  - **Synchronous**: Messages are delivered within an upper bound time frame. Node execute the tasks/algorithm with a known speed.
  - **Partial-Synchronous**: The system is asynchronous for a short/finite (but unknown) periods of time. It is synchronous otherwise.
  - **Asynchronous**: Messages can be delayed. Nodes can “pause”. No guarantees to deliver messages at all.

FAILURE MODEL

- Fault tolerant distributed systems need address:
- Networks failure
  - Reliable link, Fair-loss link, Arbitrary link
- Node behavior
  - Crash-stop, Crash-recovery, Byzantine
- Timing (Synchrony)
  - Synchronous, Partial-Synchronous, Asynchronous

FAULT TOLERANCE AND AVAILABILITY

FAULT TOLERANCE AND AVAILABILITY

- Availability
  - Online store wants to sell products 24/7
  - Service unavailability = LOSS of money

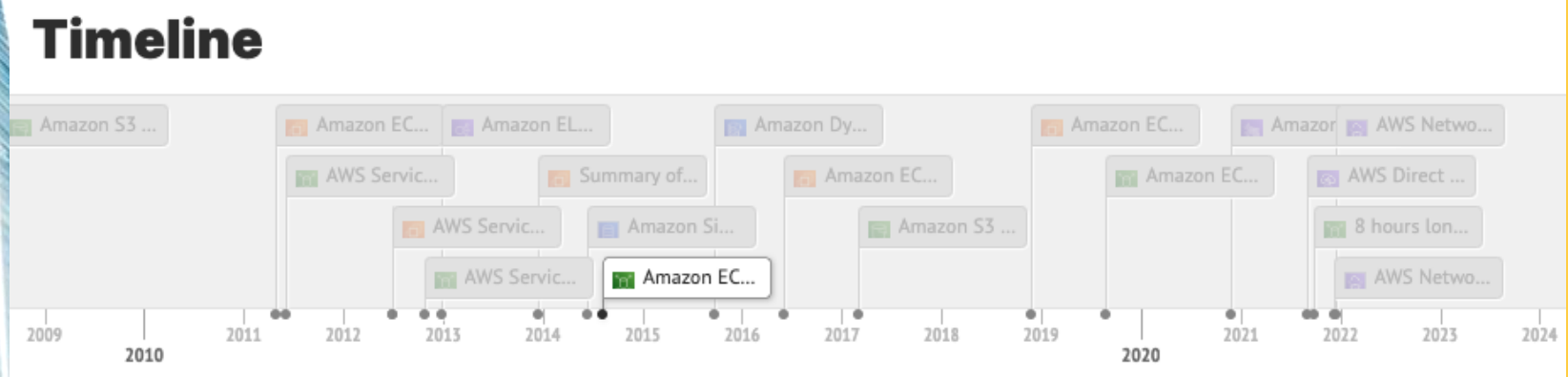
- Availability = uptime = % of time service is functional
  - "Two nines" = 99% up = down 3.7 days/year
  - "Three nines" = 99.9% up = down 8.8 hours/year
  - "Four nines" = 99.99% up = down 53 minutes/year
  - "Five nines" = 99.999% up = down 5.3 minutes/year

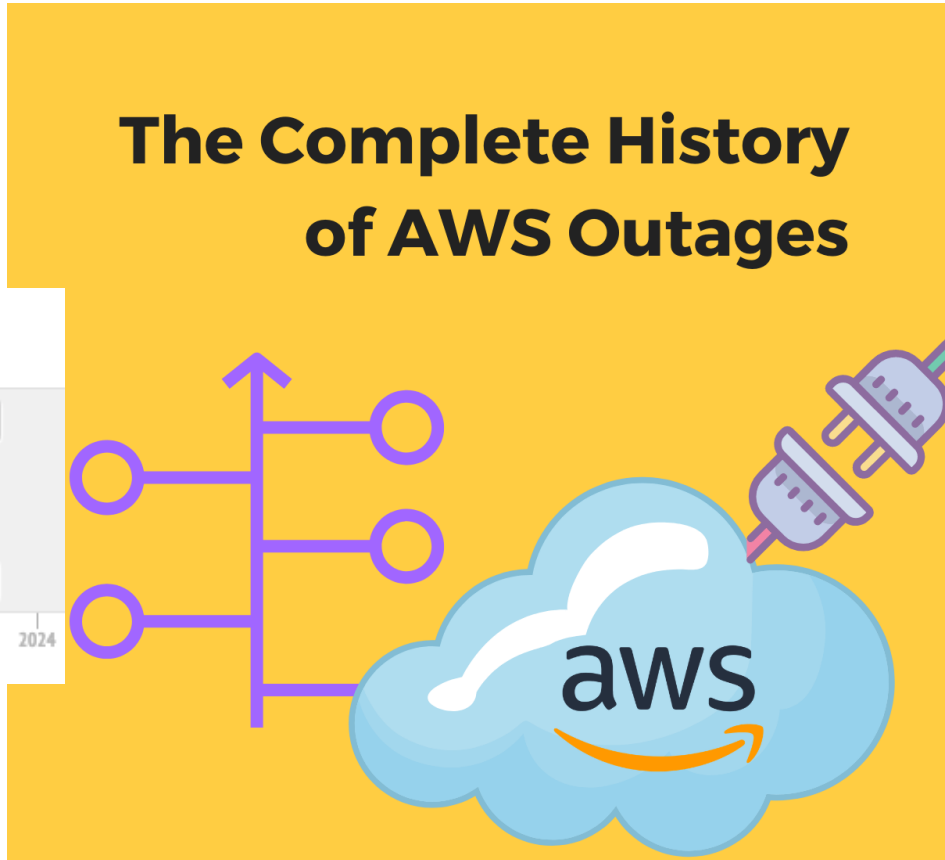
Service-Level Objective (SLO): e.g. "99.9% of requests in a day get a response in 200 ms"
Service-Level Agreement (SLA): contract specifying some SLO, penalties for violation

FAULT TOLERANCE AND AVAILABILITY

- Case-study: A complete History of Amazon AWS outages
- A good resource / time-line for AWS service failures/outages

https://awsmaniac.com/aws-outages/





FAULT TOLERANCE AND AVAILABILITY

- Achieving High Availability => Fault tolerance

- Failure: system as a whole isn't working
- Fault: some part of the system isn't working
  - Node fault: crash (crash-stop/crash-recovery), Byzantine?
  - Network fault: dropping or significantly delaying messages
- Fault tolerance:
  - System as a whole continues working, despite faults (up to some maximum number of faults)
- Single point of failure (SPOF):
  - Node/network link whose fault leads to failure

FAULT TOLERANCE AND AVAILABILITY

- **Failure detection**

- Goal: Detect failure before it happens!

- **Problem**: *Cannot tell the difference between crashed node, temporarily unresponsive node, lost message, and delayed message*

FAULT TOLERANCE AND AVAILABILITY

- **Failure detection**
- **Synchronous systems:** Perfect timeout-based failure detector program can exist only in a synchronous crash-stop system with reliable links.
- **Partial-Synchronous systems:**
  - Temporarily label a node “crashed”, even though it is “correct”
  - Temporarily label a node “correct”, even though it is “crashed”
  - Eventually label a node “crashed”, if and only if, it is “crashed”
  - Detection may not be immediate, and may require various timeouts

The additional cost of achieving higher availability exceeds the cost of occasional downtime. So accepting a certain amount of downtime can be economically acceptable?!!?

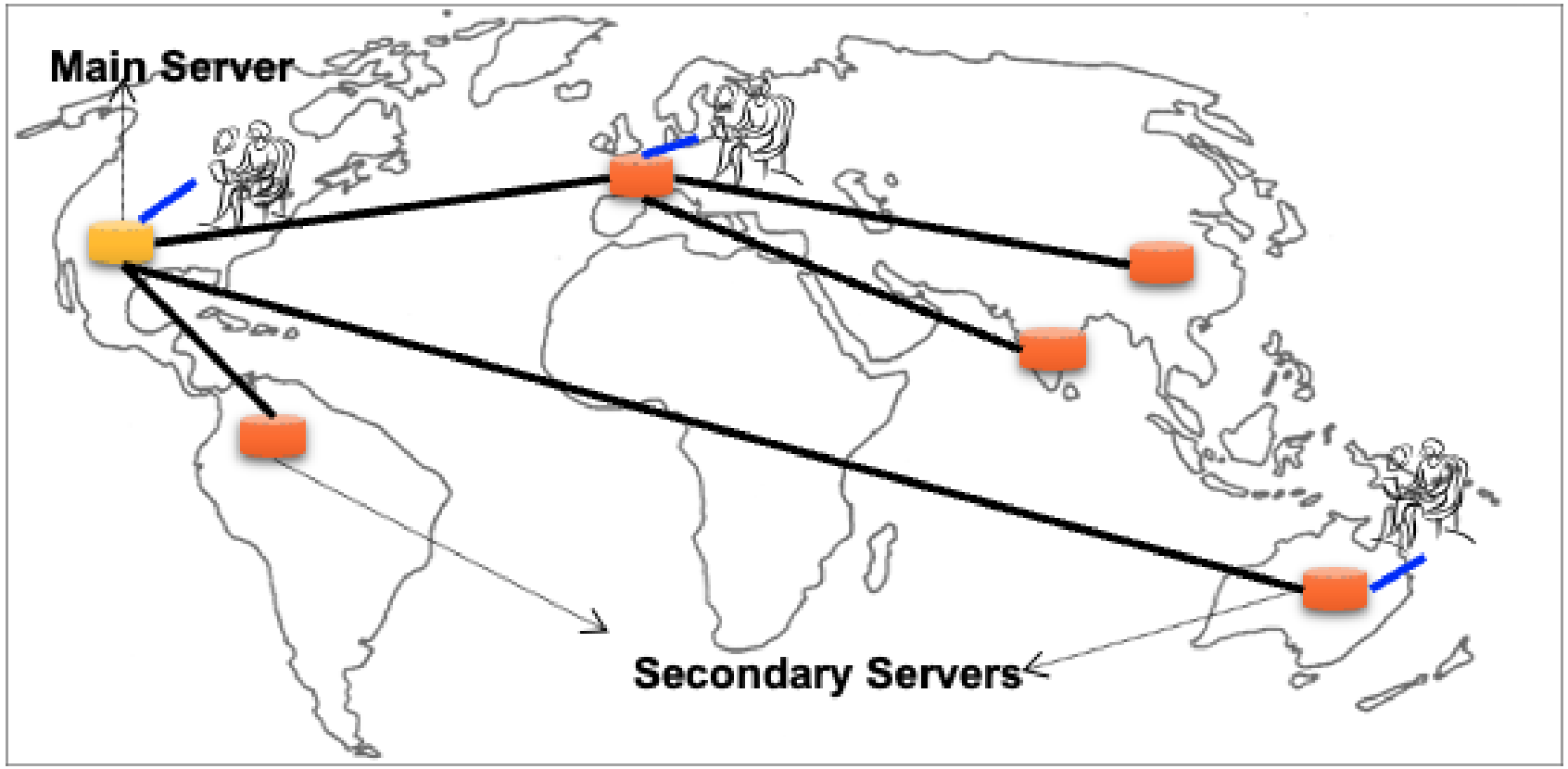
REPLICATION

REPLICATION

- Replication = An object has identical copies, each maintained by a separate server
  - Copies are called “replicas”
- Why replication?
  - Fault-tolerance: With *k* replicas of each object, can tolerate failure of any (*k-1*) servers in the system
  - Load balancing: Spread read/write operations out over the *k* replicas => load lowered by a factor of *k* compared to a single replica
  - Replication => Higher Availability

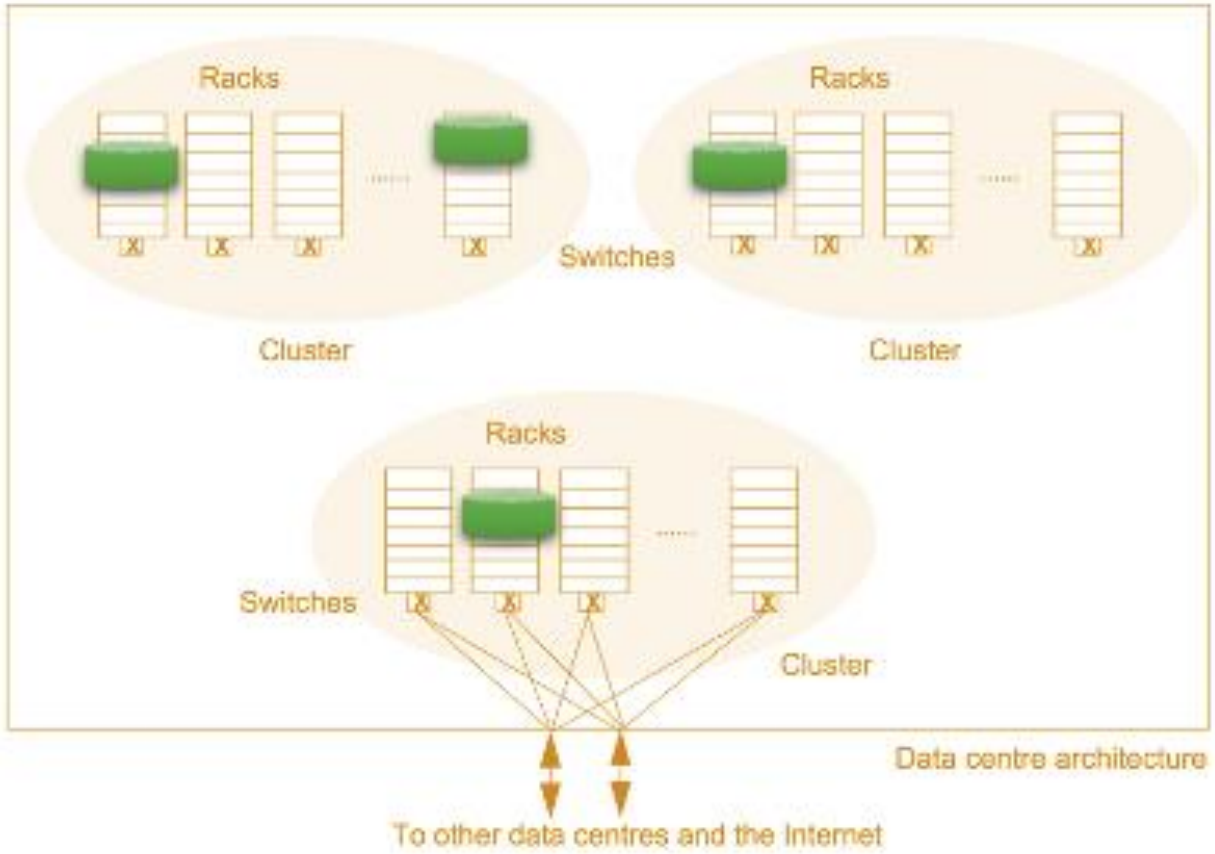
REPLICATION

- Replication is necessary for:
  1. Improving performance
     - A client can access nearby replicated copies and save latency



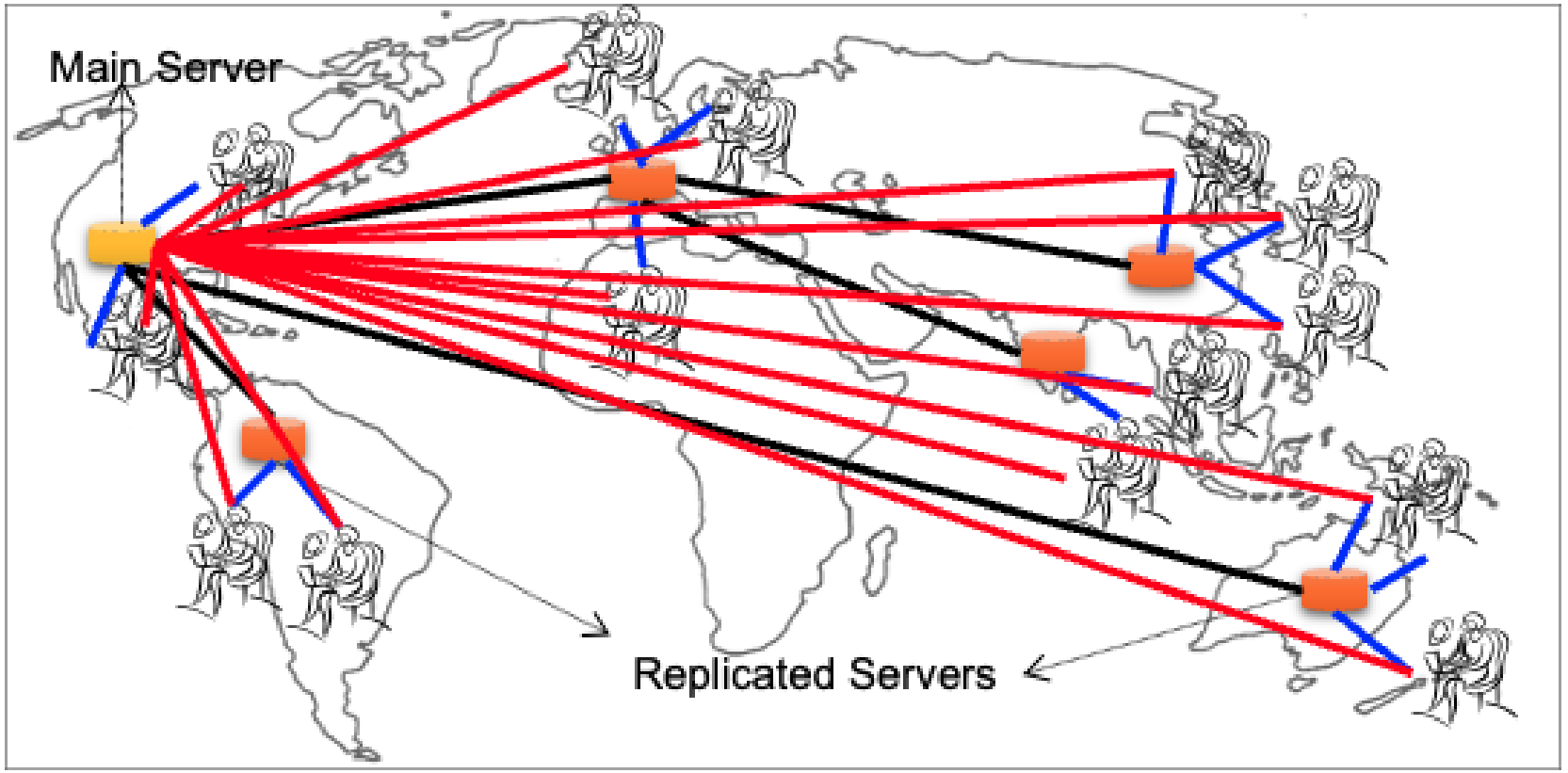
REPLICATION

- Replication is necessary for:
  2. Increasing the availability of services
     - Replication can mask failures such as server crashes and network disconnection



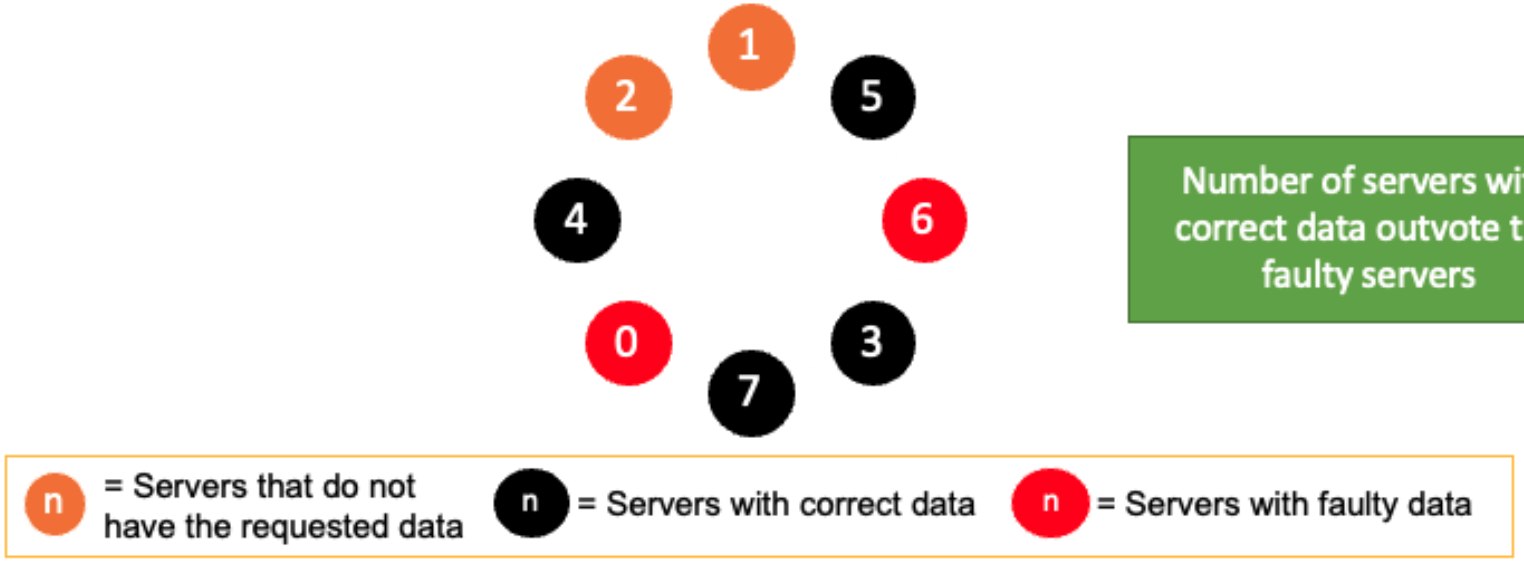
REPLICATION

- Replication is necessary for:
  3. Enhancing the scalability of systems
     - Requests to data can be distributed across many servers, which contain replicated copies of the data



REPLICATION

- Replication is necessary for:
  4. Securing against malicious attacks
     - Even if some replicas are malicious, security of data can be guaranteed by relying on replicated copies at non-compromised servers



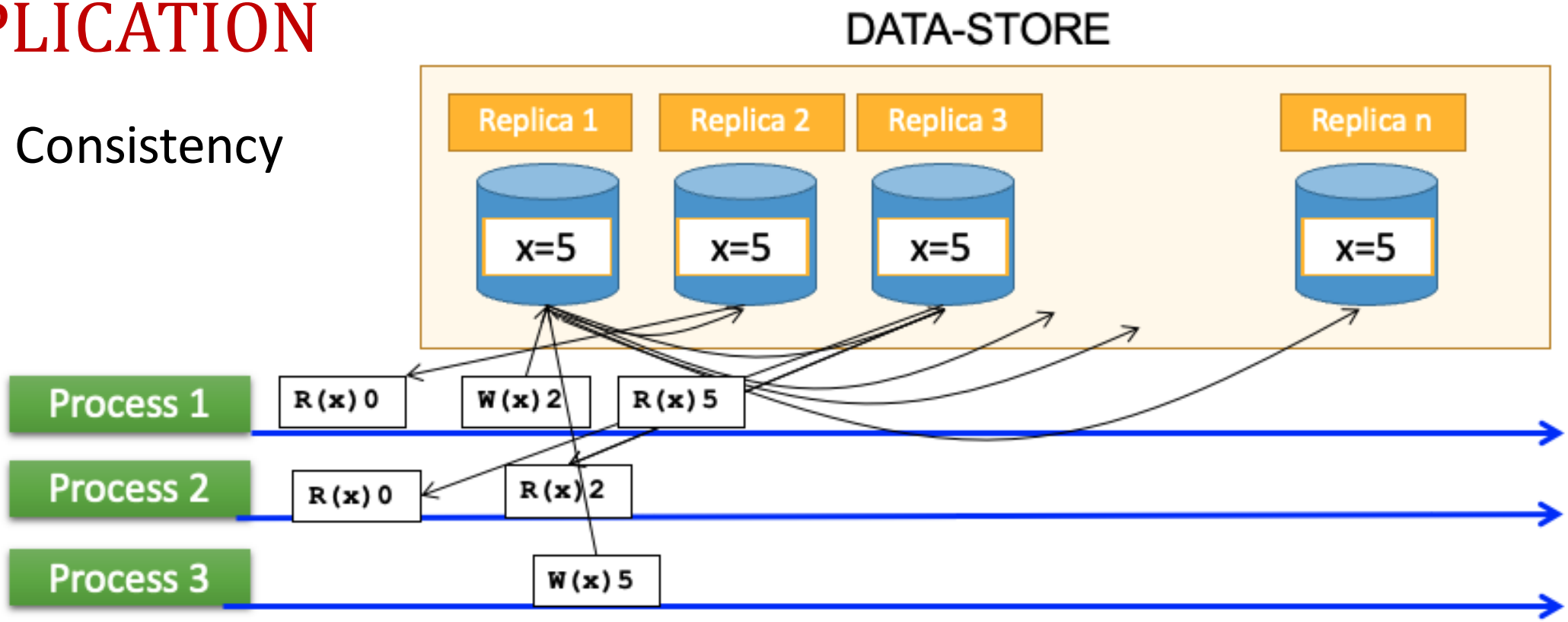
REPLICATION

Easy to implement. Main Challenge: Consistency!

- Server-side replication comes with a cost, which is the necessity for maintaining consistency (or more precisely *consistent ordering of updates*)
  - Strict Consistency
  - Loose Consistency

REPLICATION

Strict Consistency



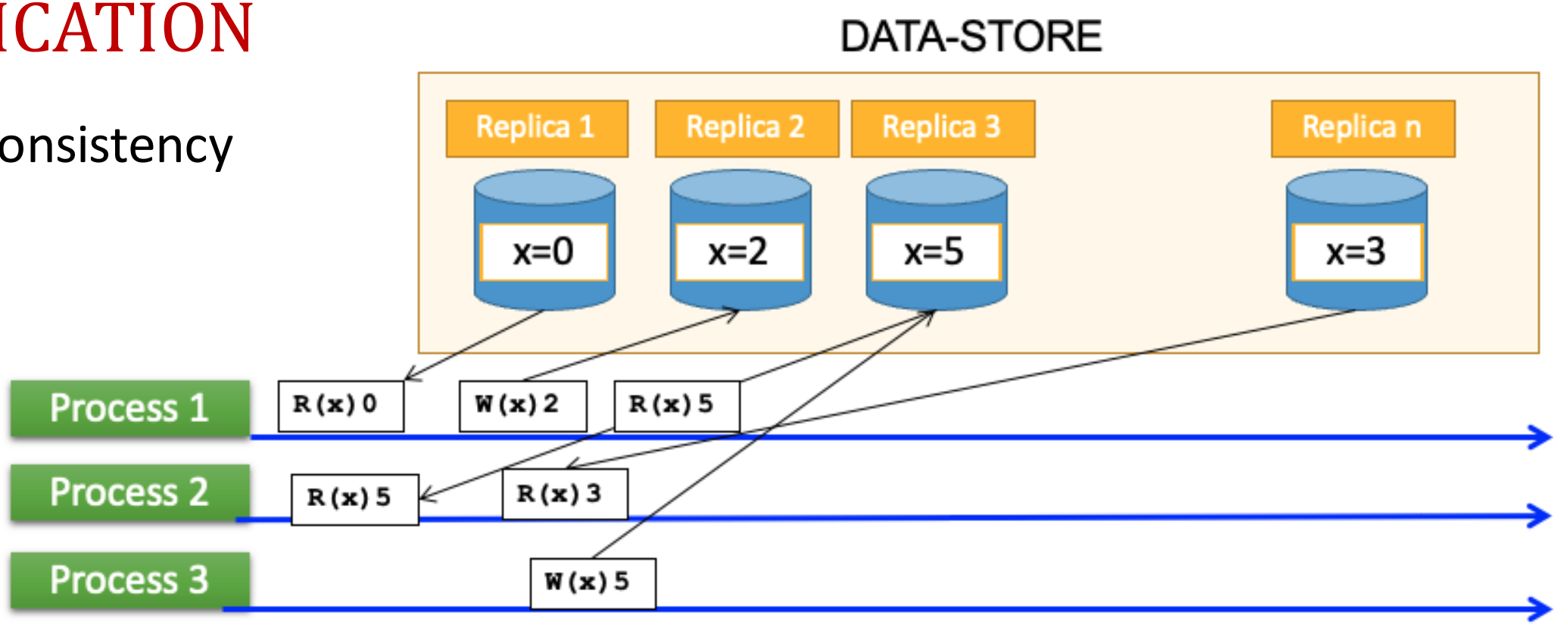
Strict Consistency

- Data is always fresh
  - After a write operation, the update is propagated to all the replicas
  - A read operation will result in reading the most recent write
- If read-to-write ratio is low, this leads to large overheads

P1 =Process P1 → =Timeline at P1 R(x)b =Read variable x; Result is b W(x)b = Write variable x; Result is b

REPLICATION

Loose Consistency



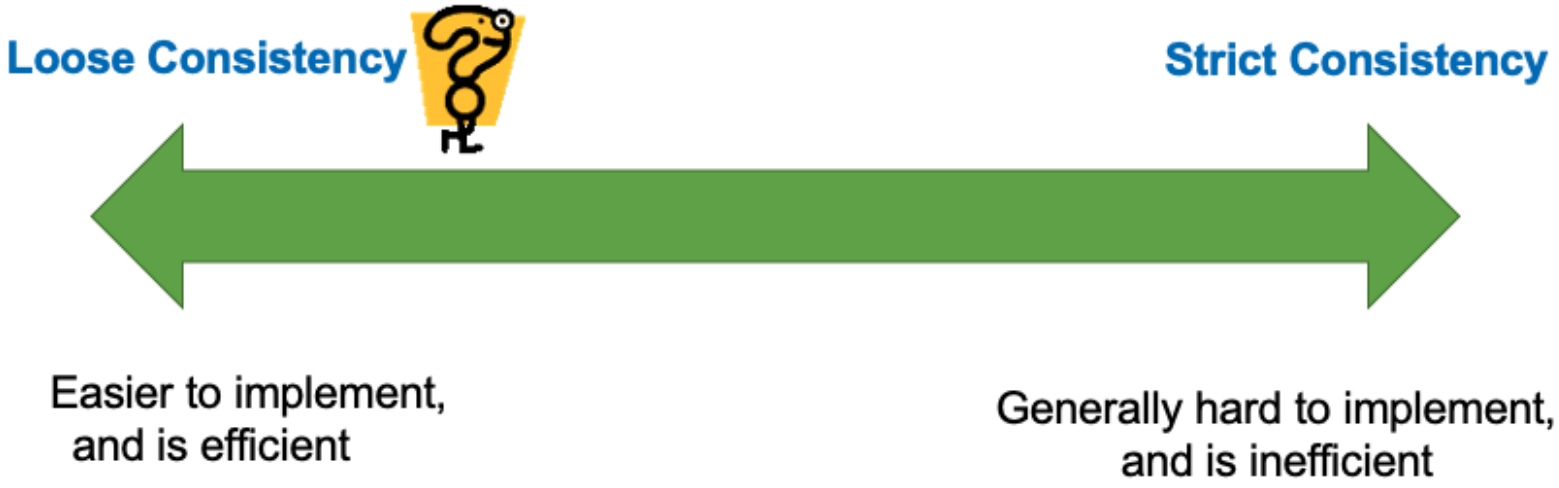
Loose Consistency

- Data might be stale
  - A read operation may result in reading a value that was written long back
  - Replicas are generally out-of-sync
- The replicas may sync at coarse grained time, thus reducing the overhead

P1 =Process P1 → =Timeline at P1 R(x)b =Read variable x; Result is b W(x)b = Write variable x; Result is b

REPLICATION

- Maintaining consistency should balance between the strictness of consistency versus efficiency (or performance)
  - Good-enough consistency depends on your application



ORDERING

ORDERING

- A *consistency model* is a contract between:
  - The **process** that wants to use the data
  - The **data-store**
- Two types
  - **Data-Centric**: How updates are propagated across the replicas to keep them consistent
  - **Client-Centric**: Clients connect to different replicas at different times. They ensure that whenever a client connects to a replica, the replica is brought up to date with the replica that the client accessed previously

ORDERING

Consistent **Ordering** of Operations

- We need to express the *semantics* of parallel accesses when shared data are replicated
- Before updates at replicas are committed, all replicas shall reach *an agreement on a global ordering* of the updates
  - That is, replicas in shared data-stores should agree on a *consistent ordering of updates*
- What consistent ordering of updates can replicas agree on?

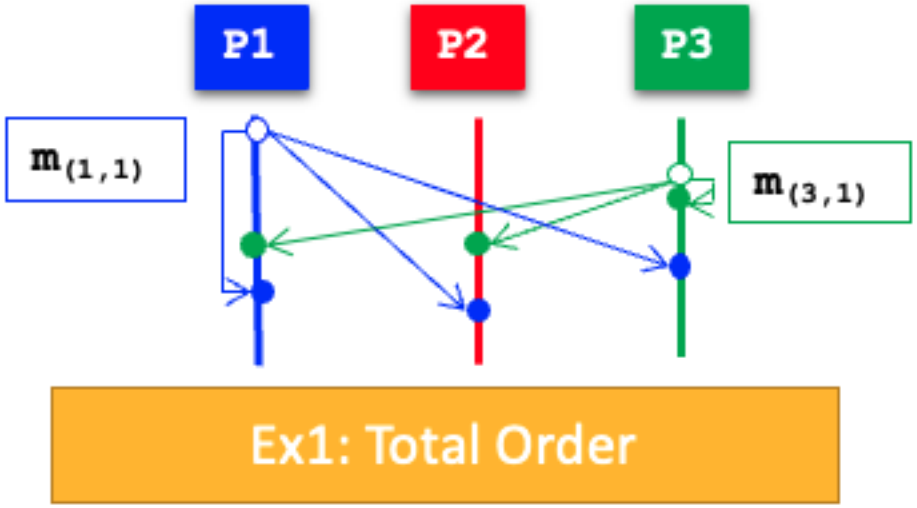
ORDERING

Three major types of orderings:

- Total Ordering
- Sequential Ordering
- Causal Ordering

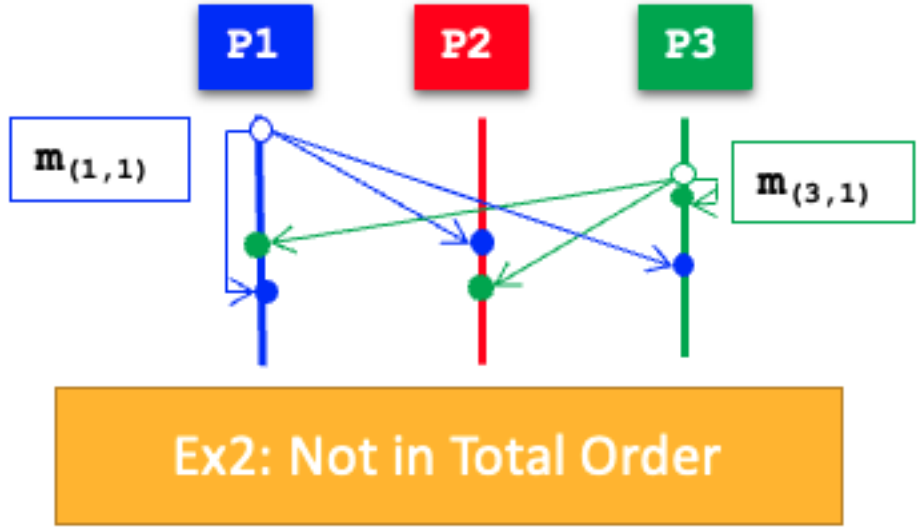
ORDERING

- Total Ordering
- If process P_i sends a message m_i and P_j sends m_j, and if one correct process delivers m_i *before* m_j then every other correct process delivers m_i *before* m_j



Example Ex1,
if P_1 issues the operation $m_{(1,1)}$: `x=x+1;` and
If P_3 issues $m_{(3,1)}$: `print(x);` and
P_1 or P_2 or P_3 delivers $m_{(3,1)}$ before $m_{(1,1)}$
Then, at all replicas P_1, P_2, P_3 the following order of operations are executed

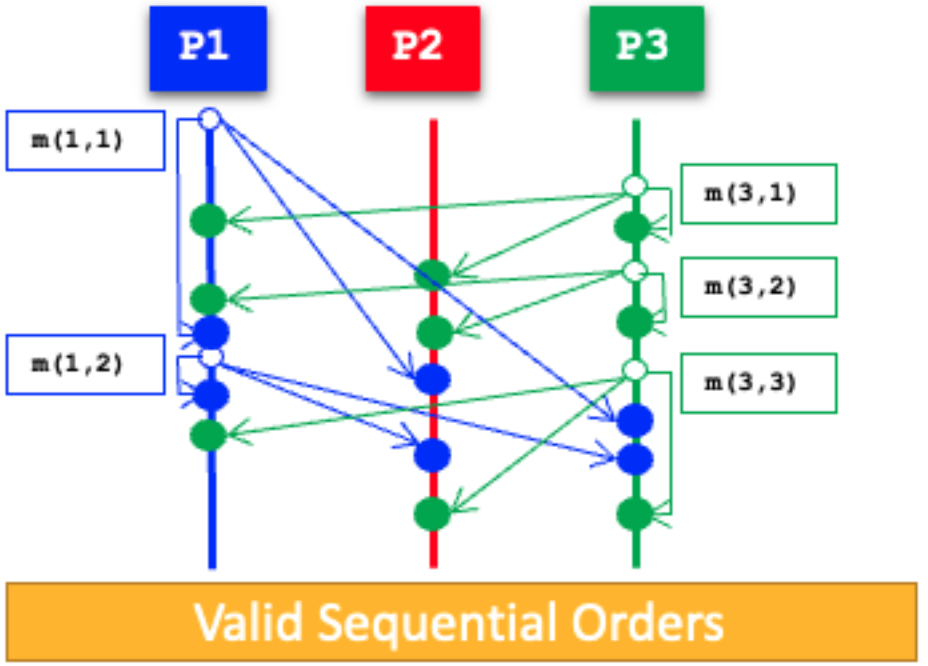
```
print(x);
x=x+1;
```



ORDERING

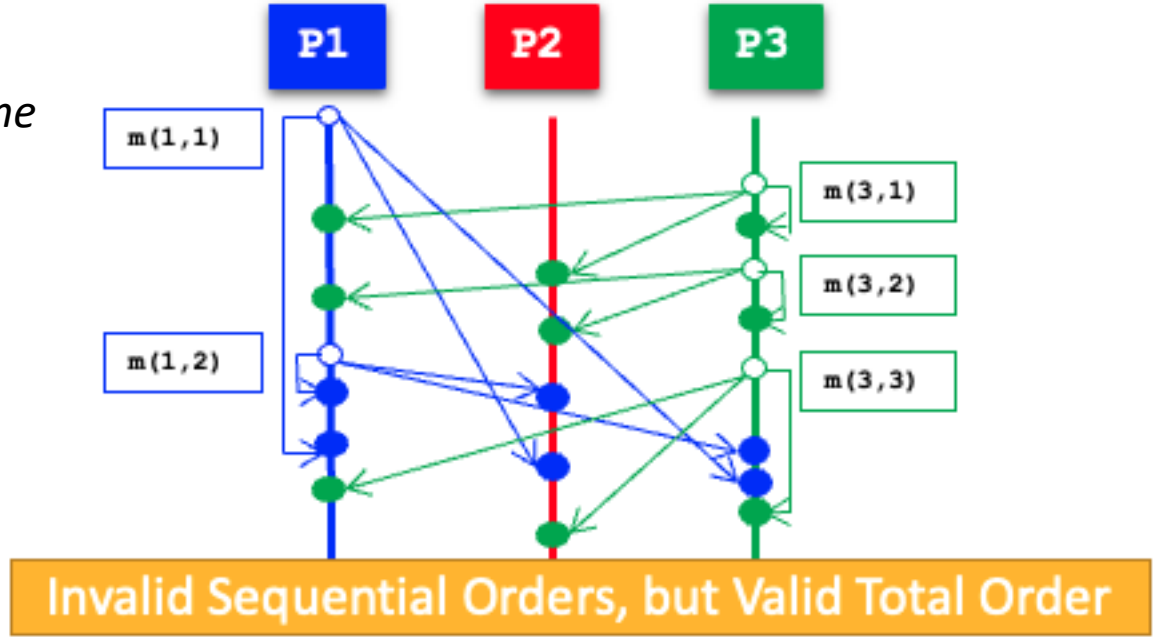
- Sequential Ordering

- If a process **Pi** sends a sequence of messages $m_{(i,1)}, \ldots, m_{(i,ni)}$, and Process **Pj** sends a sequence of messages $m_{(j,1)}, \ldots, m_{(j,nj)}$, Then at any process, the set of messages received are in *some* sequential order



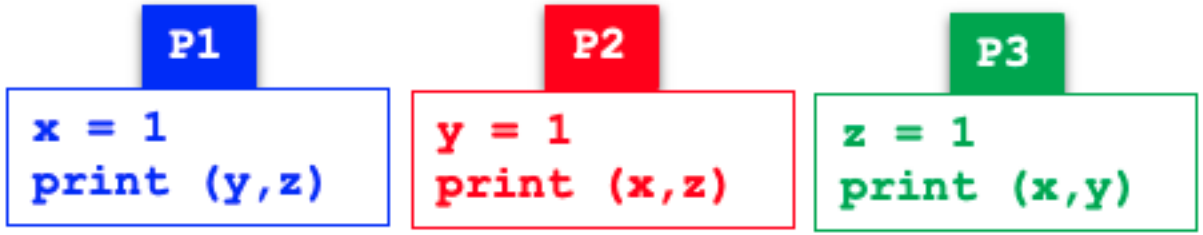
Messages from each individual process should appear *in the same order sent by that process*

- At every process, $m_{i,1}$ should be delivered before $m_{i,2}$, which should be delivered before $m_{i,3}$ and so on...
- At every process, $m_{j,1}$ should be delivered before $m_{j,2}$, which should be delivered before $m_{j,3}$ and so on...



ORDERING

• Sequential Ordering



- Example: Consider three processes $\mathbf{P_1}$, $\mathbf{P_2}$ and $\mathbf{P_3}$ executing multiple instructions on three *shared* variables **x**, **y** and **z**. Assume that **x**, **y** and **z** are set to zero at start
- There are many valid sequences in which operations can be executed, respecting sequential consistency (or *program order*). How can a program identify the wrong sequence among the following sequences?

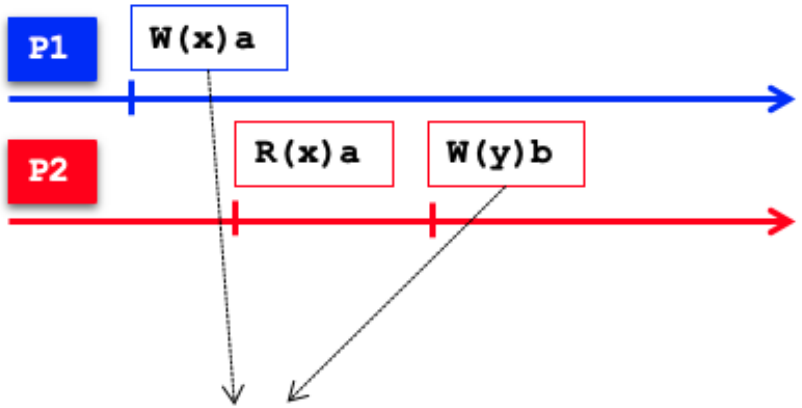
	x = 1 print (y,z) y = 1 print (x,z) z = 1 print (x,y)	x = 1 y = 1 print (x,z) print (y,z) z = 1 print (x,y)	print (y,z) y = 1 print (x,y) x = 1 print (x,z) z = 1	y = 1 z = 1 print (x,y) print (x,z) x = 1 print (y,z)
Output	001011	101011	000110	010111
Signature	001011	101011	001001 ✗	110101 ✗

Number of combinations for a total of n instructions = $O(n!)$

ORDERING

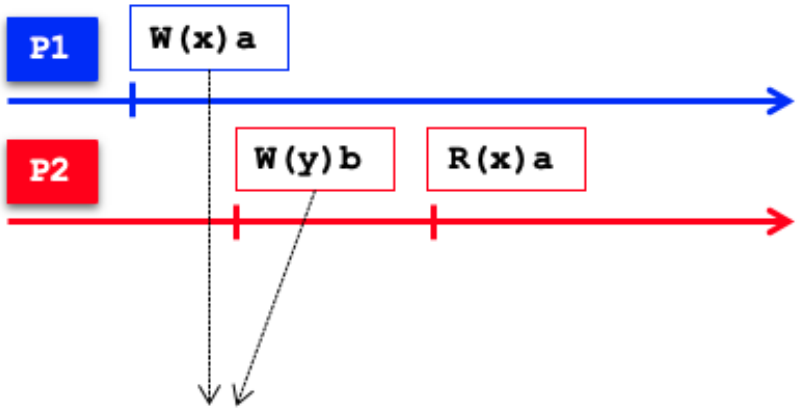
• Causal Ordering

- Consider an interaction between processes $\mathbf{P_1}$ and $\mathbf{P_2}$ operating on replicated data **x** and **y**



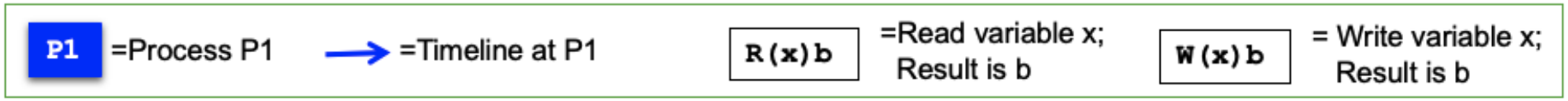
Events are causally related
Events are not concurrent

- Computation of **y** at $\mathbf{P_2}$ *may* have depended on the value of **x** written by $\mathbf{P_1}$



Events are not causally related
Events are concurrent

- Computation of **y** at $\mathbf{P_2}$ does not depend on the value of **x** written by $\mathbf{P_1}$



ORDERING

• Causal Ordering

- If process P_i sends a message m_i and P_j sends m_j, and if $m_i \rightarrow m_j$ (operator '→' is Lamport's **happened-before** relation) then any correct process that delivers m_j will deliver m_i before m_j

- In Ex1:
  - $m_{(1,1)}$ and $m_{(3,1)}$ are in Causal Order
  - $m_{(1,1)}$ and $m_{(1,2)}$ are in Causal Order

- In Ex2:
  - $m_{(1,1)}$ and $m_{(3,1)}$ are NOT in Causal Order