BITS Pilani
Pilani Campus

Lectuer-4
Big Data Systems(SEZG522)

Slides: Courtesy:.Prof. Anindya

BITS Pilani
Pilani Campus

First Semester

2022-23

Lecture -4 Contents

- Characteristics of Big Data Systems
  - ✓Failures - Reliability and Availability
  - ✓Consistency
- Consistency, availability, partition tolerance
- CAP theorem

Distributed computing – living with failures

Failures of nodes and links is a common concern in Distributed Systems

- Essential to have fault tolerance aspect in design

Fault tolerance is a measure of

- How a distributed system functions in the presence of failures of system components
- Tolerance of component faults is measured by 2 parameters
  - Reliability - An inverse indicator of failure rate
    - How soon a system will fail
  - Availability - An indicator of fraction of time a system is available for use
    - System is not available during failure

Metrics

MTTF - Mean Time To Failure

- MTTF = 1 / failure rate = Total #hours of operation / Total #units
- MTTF is an averaged value. In reality failure rate changes over time because it may depend on age of component.

Failure rate = 1 / MTTF (assuming average value over time)

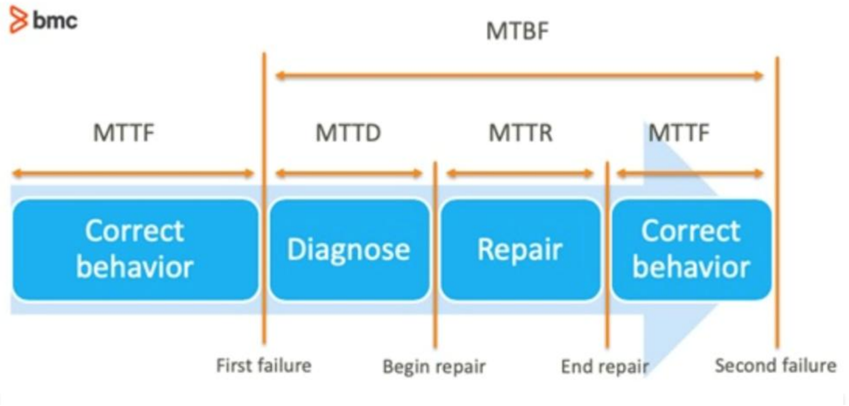
MTTR - Mean Time to Recovery / Repair

- MTTR = Total #hours for maintenance / Total #repairs

MTTD - Mean Time to Diagnose

MTBF - Mean Time Between Failures

- MTBF = MTTD + MTTR + MTTF



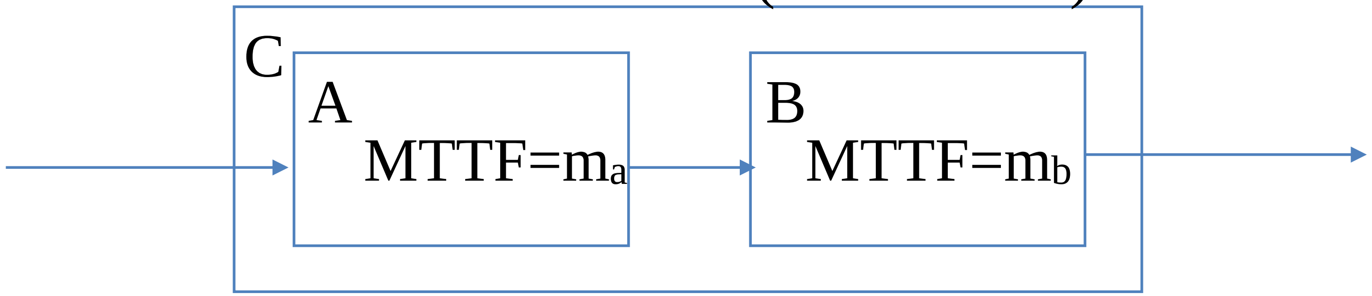
Reliability - serial assembly

MTTF of a system is a function of MTTF of components

Serial assembly of components

- Failure of any component results in system failure
- Failure rate of C = Failure rate of A + Failure rate of B = $1/m_a + 1/m_b$

MTTF $m_c=1/(1/m_a + 1/m_b)$



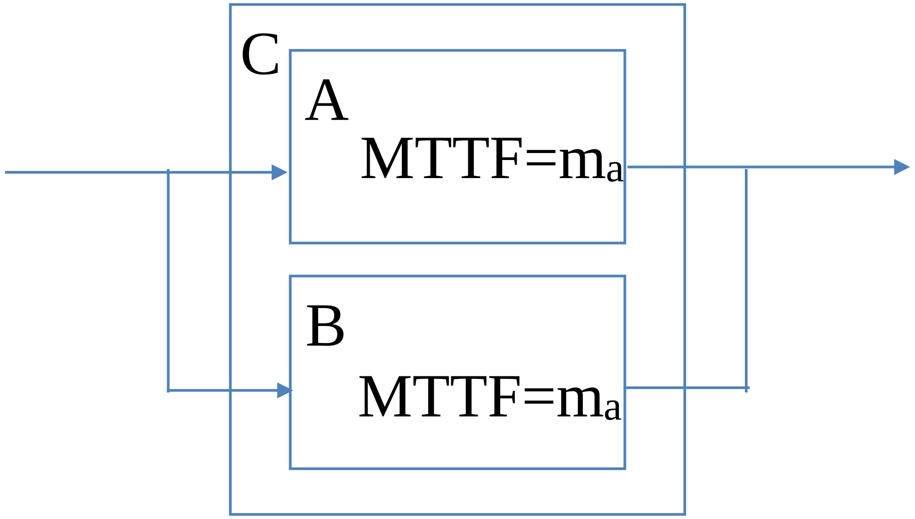
- MTTF of system = 1 / SUM ($1/\text{MTTF}_i$) for all components i
- Failure rate of system = SUM($1/\text{MTTF}_i$) for all components i

Reliability - parallel assembly

In a parallel assembly, e.g. a cluster of nodes

- MTTF of C = MTTF A + MTTF B because both A and B have to fail for C to fail

MTTF of system = SUM(MTTF$_i$) for all components i



MTTF m_c=m_a + m_b

Availability

Availability = Time system is UP and accessible / Total time observed

Availability = MTTF / (MTTD* + MTTR + MTTF)

or

Availability = MTTF / MTBF

A system is highly available when

- MTTF is high
- MTTR is low

* Unless specified one can assume MTTD = 0

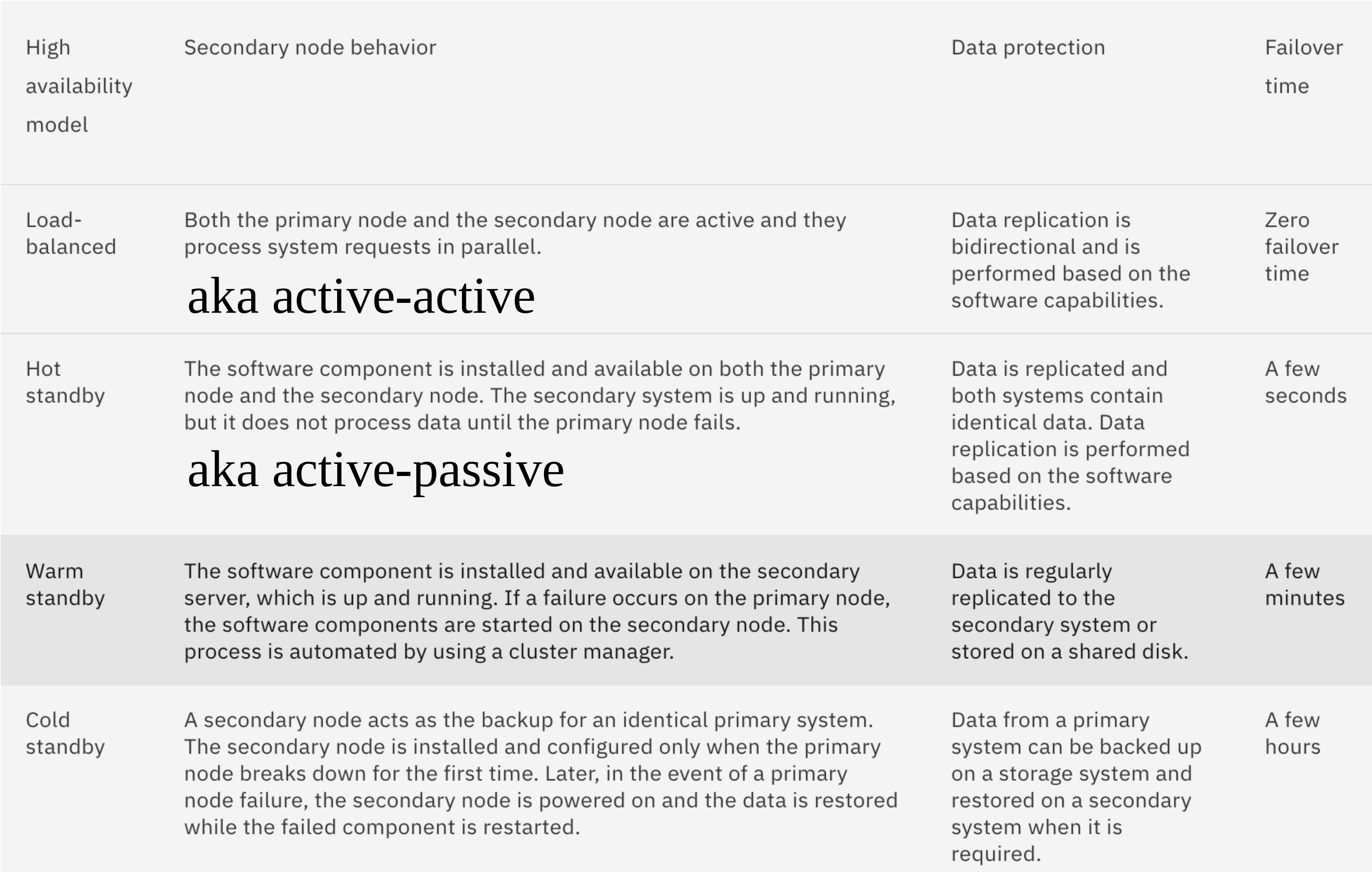
Example

- A node in a cluster fails every 100 hours while other parts never fail. On failure of the node the whole system needs to be shutdown, faulty node replaced and system. This takes 2 hours. The application needs to be restarted, which takes 2 hours.
- What is the availability of the cluster ?
- If downtime is $80k per hour, the what is the yearly cost ?

Solution

- MTTF = 100 hours
- MTTR = 2 + 2 = 4 hours
- Availability = 100/104 = 96.15%
- Cost of downtime per year = 80000 x 3.85 * 365 * 24 / 100 = USD 27 million

Fault tolerance configurations - standby options

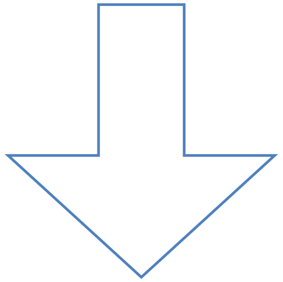


High availability model	Secondary node behavior	Data protection	Failover time
Load-balanced	Both the primary node and the secondary node are active and they process system requests in parallel. aka active-active	Data replication is bidirectional and is performed based on the software capabilities.	Zero failover time
Hot standby	The software component is installed and available on both the primary node and the secondary node. The secondary system is up and running, but it does not process data until the primary node fails. aka active-passive	Data is replicated and both systems contain identical data. Data replication is performed based on the software capabilities.	A few seconds
Warm standby	The software component is installed and available on the secondary server, which is up and running. If a failure occurs on the primary node, the software components are started on the secondary node. This process is automated by using a cluster manager.	Data is regularly replicated to the secondary system or stored on a shared disk.	A few minutes
Cold standby	A secondary node acts as the backup for an identical primary system. The secondary node is installed and configured only when the primary node breaks down for the first time. Later, in the event of a primary node failure, the secondary node is powered on and the data is restored while the failed component is restarted.	Data from a primary system can be backed up on a storage system and restored on a secondary system when it is required.	A few hours

Decreasing cost

vs

Increasing MTTR

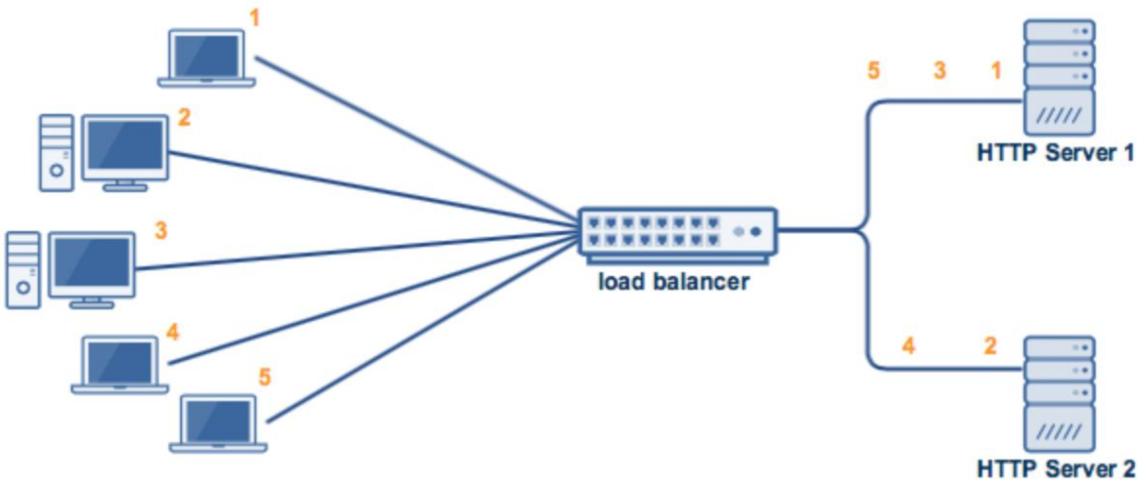


Assumption:
Same software bug or
runtime fault will not
recur in the standby

https://www.ibm.com/docs/en/cdfsp/7.6.0?topic=systems-high-availability-models

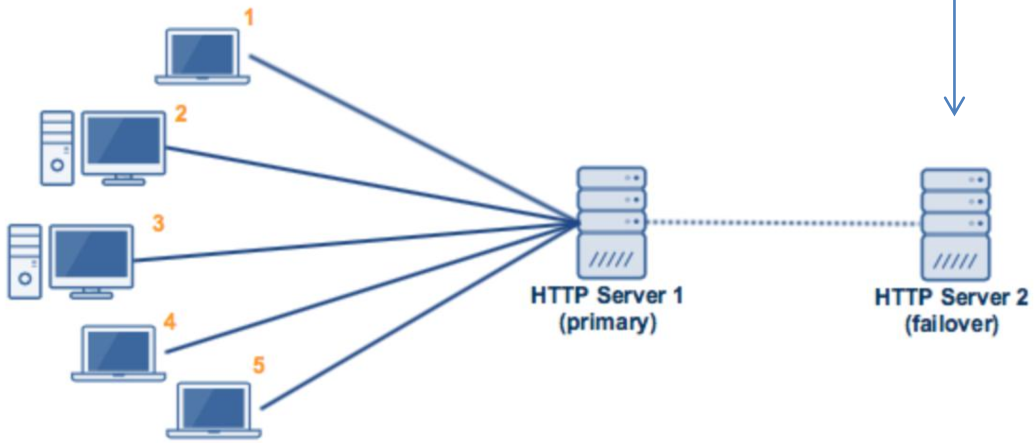
Fault tolerance configurations - standby options

Active-Active



Warm and cold depends on how the passive / failover node is kept updated

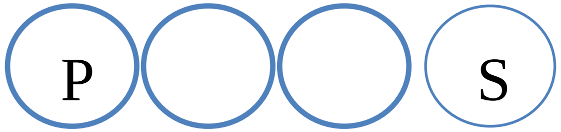
Active-Passive



Fault tolerance configurations - cluster topologies

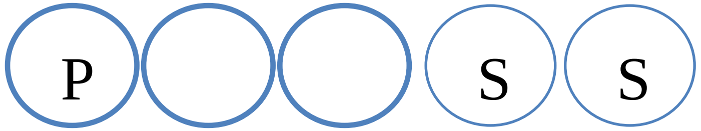
N+1

- One node is configured to take the role of the primary



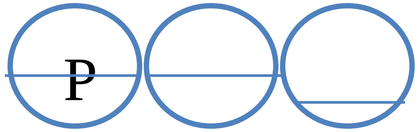
N+M

- M standby nodes if multiple failures are anticipated especially when running multiple services in the cluster



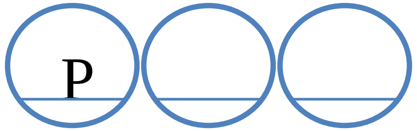
N to 1

- The secondary failover node is a temporary replacement and once primary is restored, services must be reactivated on it



N to N

- When any node fails, the workload is redistributed to the remaining active nodes. So there is no special standby node.



https://www.ibm.com/docs/en/cdfsp/7.6.0?topic=systems-high-availability-models

Fault Tolerant Clusters – Recovery

Diagnosis

- Detection of failure and location of the failed component, e.g. using heartbeat messages between nodes

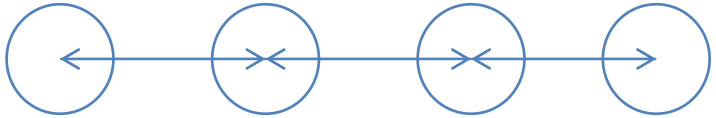
Backward recovery

- periodically do a checkpoint (save consistent state on stable storage)
- on failure, isolate the failed component, rollback to last checkpoint and resume normal operation
- Ease to implement, independent of application, but leads to wastage of execution time on rollback besides unused checkpointing work

Forward recovery

- In real-time systems or time-critical systems cannot rollback. So state is reconstructed on the fly from diagnosis data.
- Application specific and may need additional hardware

checkpoints



rollback on errors

Consistency

- Big Data systems write replicas of a shard / partition
- Any write needs to update all replicas
- Any reads can happen in between

Consistency —

- Do you allow a read of any replica in any thread to always read the latest value written in any thread ?
  - RDBMS / OLTP systems / Systems of Record
  - ACID (Atomicity, Consistency, Isolation, Durability)
- Do you allow reads to return any value and eventually show the latest stable value
  - Some BigData systems / Systems of Engagement e.g. social network comments
  - BASE (Basic Availability, Soft state, Eventual consistency)

Ref: https://www.dummies.com/programming/big-data/applying-consistency-methods-in-nosql/

Consistency clarification

C in ACID is different from C as in CAP Theorem (in next session) *

C in ACID of RDBMS based OLTP systems

- A broader concept at a data base level for a transaction involving multiple data items
- Harder to achieve

C in CAP Theorem for most NoSQL systems

- Applies to ordering of operations for a single data item not a transaction involving multiple data items
- So it is a strict subset of C in ACID
- Typically support of ACID semantics for NoSQL systems defeats the purpose as it involves having a single transaction manager that becomes a bottleneck for large scale sharding to scale

Levels of Consistency

Strict

- Requires real time line ordering of all writes - assumes actual write time can be known. So “reads” read the latest data in real time across threads.

Linearisable

- Acknowledges that write requests take time to write all copies - so does not impose ordering within overlapping time periods of read / write

Sequential

- All writes across threads for all data items are globally ordered. All threads must see the same order. But does not need real-time ordering.

Causal

- Popular and useful model where only causally connected writes and reads need to be ordered. So if a write of a data item (Y) happened after a read of same or another data item (X) in a thread then all threads must observe write X before write to Y.

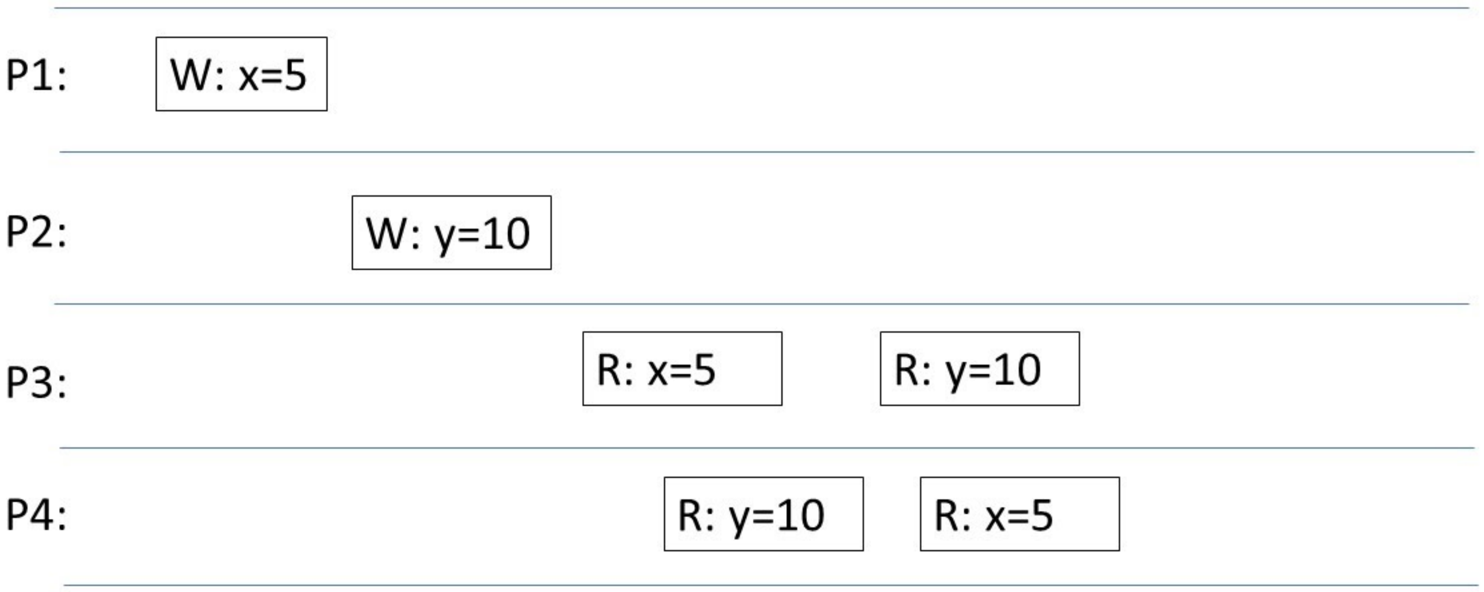
Eventual (most relaxed as in BASE)

- If there are no writes for “some time” then all threads will eventually agree on a latest value of the data item

ref: http://dbmsmusings.blogspot.com/2019/07/overview-of-consistency-levels-in.html

Example: Strictly Consistent

(Initial value of X and Y are 0)



P1: W: x=5

P2: W: y=10

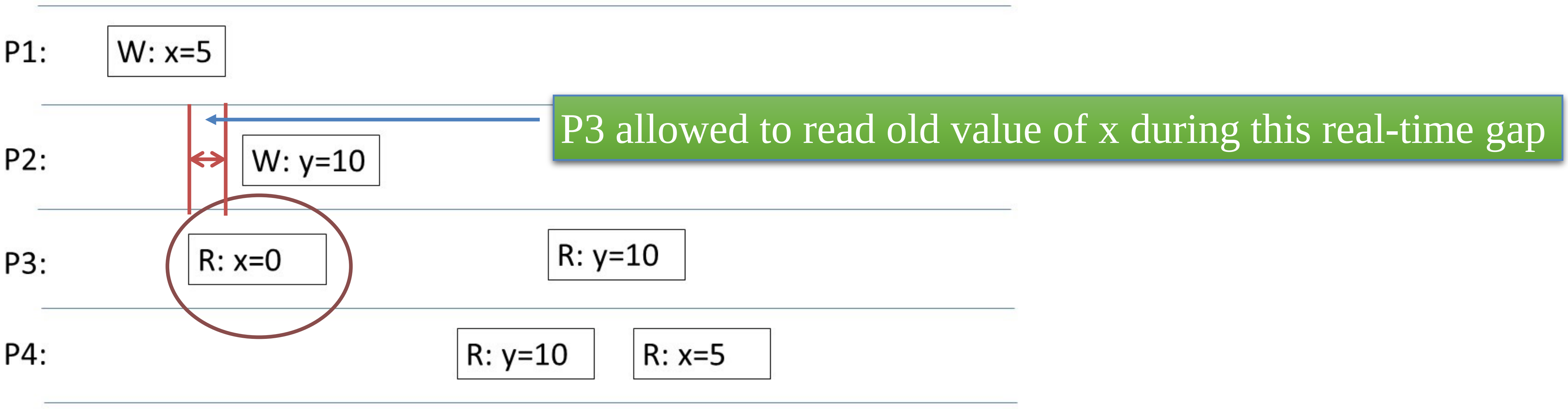
P3: R: x=5 R: y=10

P4: R: y=10 R: x=5

This schedule is sequentially consistent, causally consistent, linearizable and strictly consistent

Example: Linearizability

(Initial value of X and Y are 0)



P1: W: x=5

P2: W: y=10

P3: R: x=0 R: y=10

P4: R: y=10 R: x=5

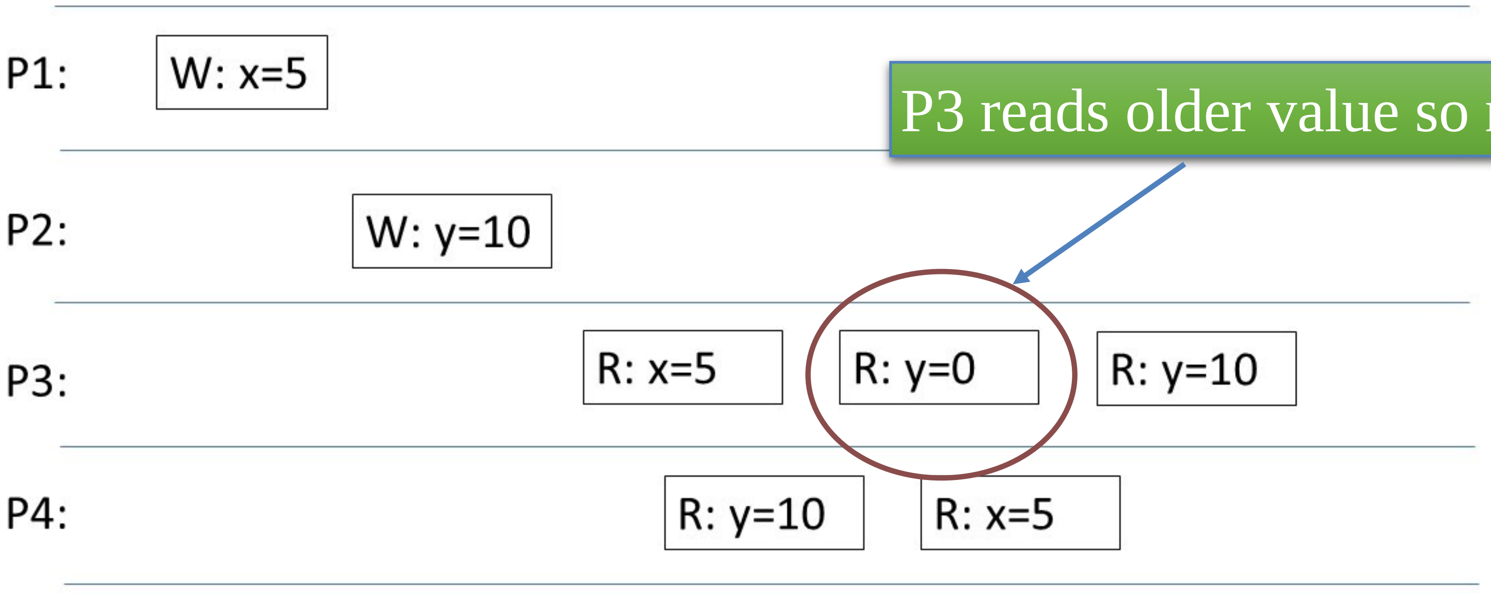
P3 allowed to read old value of x during this real-time gap

This schedule is sequentially consistent, causally consistent, and linearizable, but **not** strictly consistent

Linearizability takes into account the overlapping time when P3 could not read the latest write of x.

Example: Sequentially Consistency

(Initial value of X and Y are 0)



P1: W: x=5

P2: W: y=10

P3: R: x=5 R: y=0 R: y=10

P4: R: y=10 R: x=5

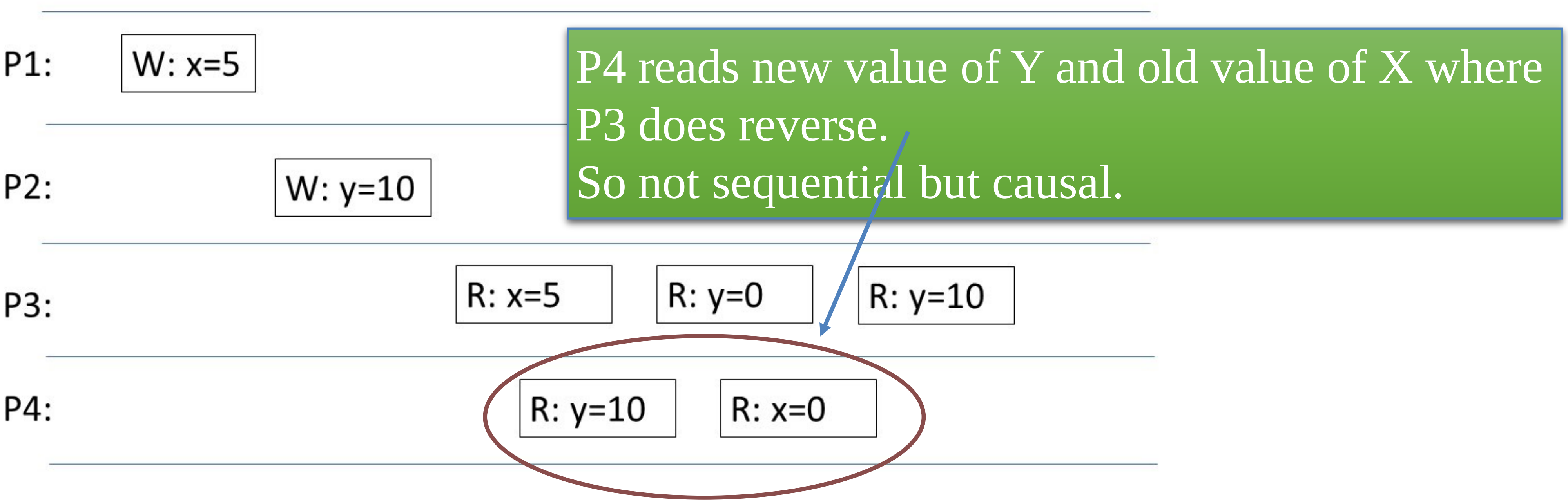
P3 reads older value so not linearizable but sequenti

This schedule is sequentially consistent and causally consistent, but not linearizable or strictly consistent

All writes across threads are globally ordered but not in real-time. Here P3 reads older value of Y in real-time but P1 and P2 writes are deemed parallel in sequential order.

Example: Causal Consistency

(Initial value of X and Y are 0)



P1:	W: x=5						
P2:		W: y=10					
P3:			R: x=5		R: y=0		R: y=10
P4:				R: y=10		R: x=0	

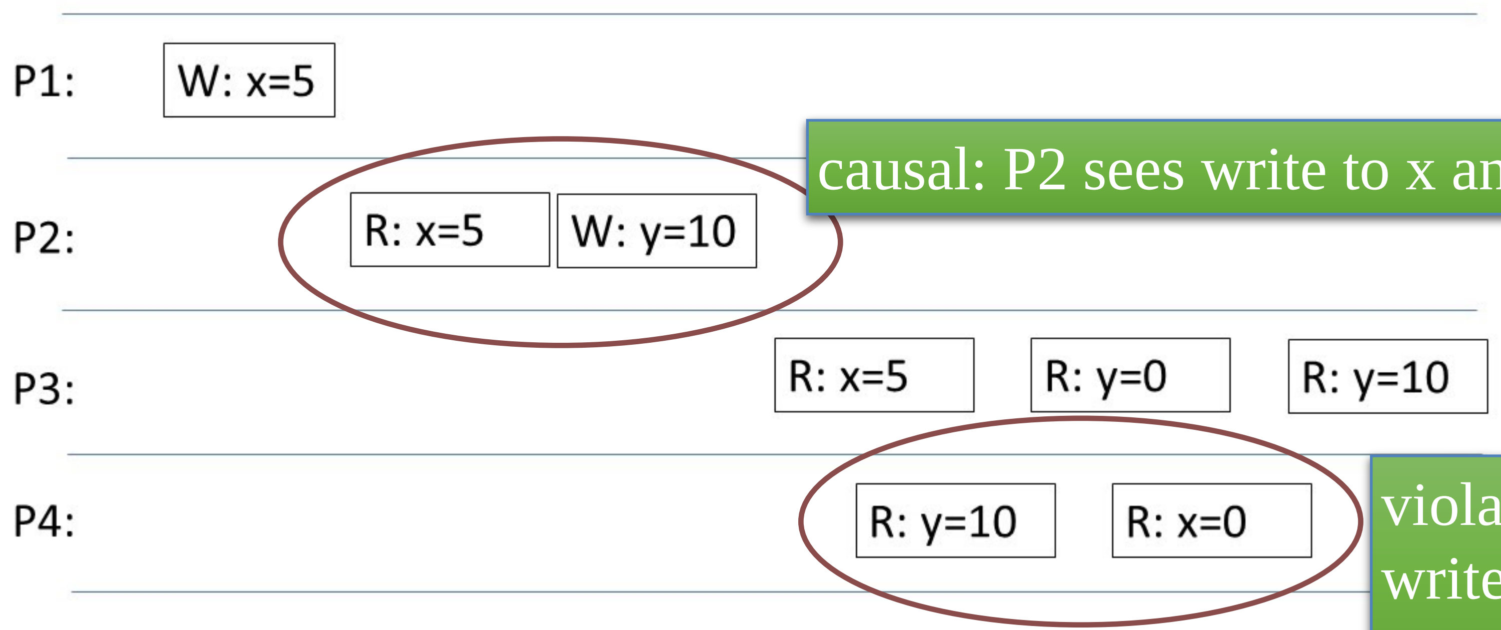
P4 reads new value of Y and old value of X where P3 does reverse.
So not sequential but causal.

This schedule is causally consistent, but not linearizable or strictly consistent or even sequentially consistent

Cannot be sequentially consistent because P3 and P4 read X and Y values in different order. But no causal order violated because x and y are not causally related.

Example: Eventually Consistent

(Initial value of X and Y are 0)



causal: P2 sees write to x and then writes to y

violated : P4 sees write to y but not write to x

This schedule is **not** causally consistent, nor linearizable or strictly consistent or sequentially consistent

CAP

Stands for Consistency (C) , Availability (A) and Partition Tolerance (P)

Triple constraint related to the distributed database systems

Consistency

- ✓ A read of a data item from any node results in same data across multiple nodes

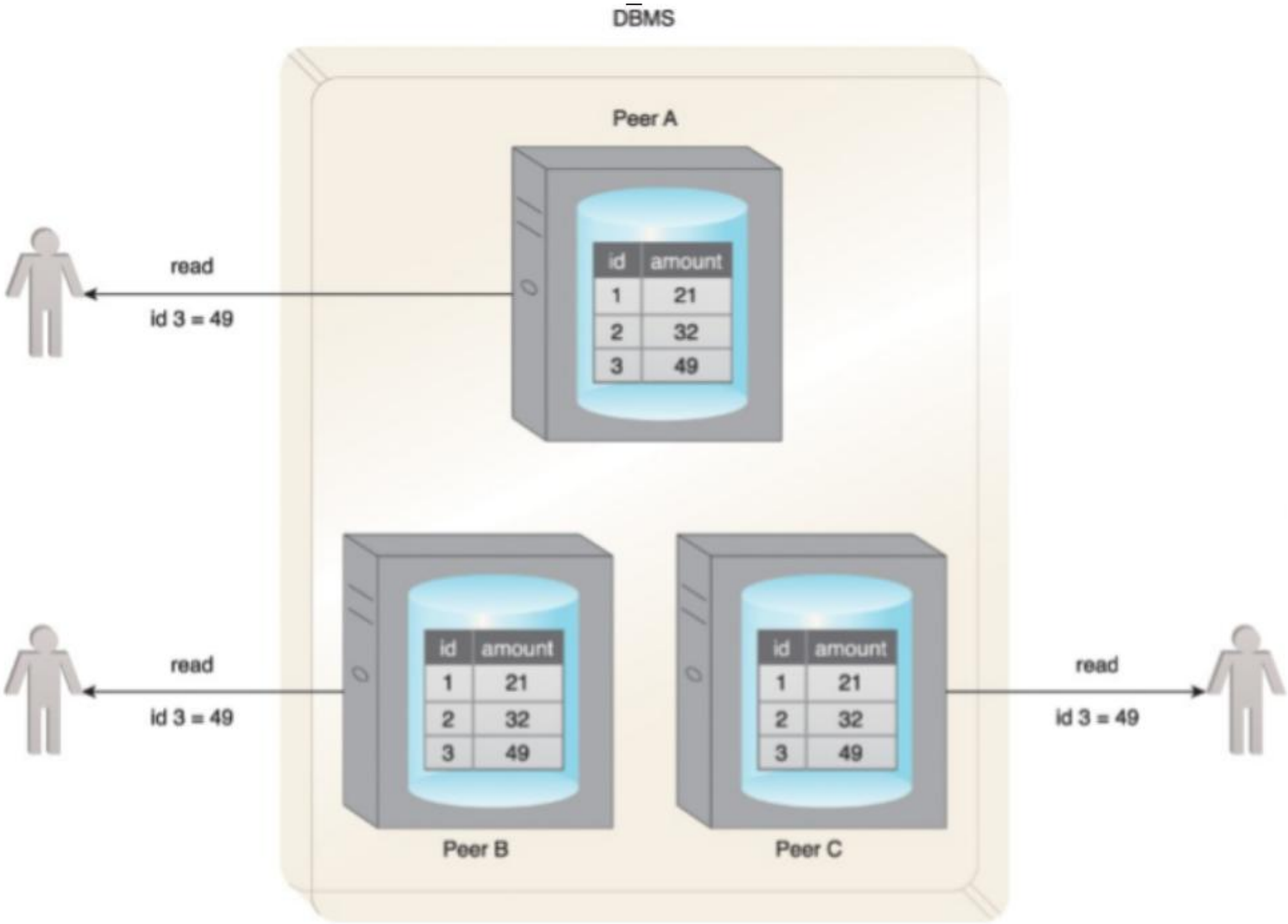
Availability

- ✓ A read/write request will always be acknowledged in form of success or failure in reasonable time

Partition tolerance

- ✓ System can continue to function when communication outages split the cluster into multiple silos and can still service read/write requests

Consistency



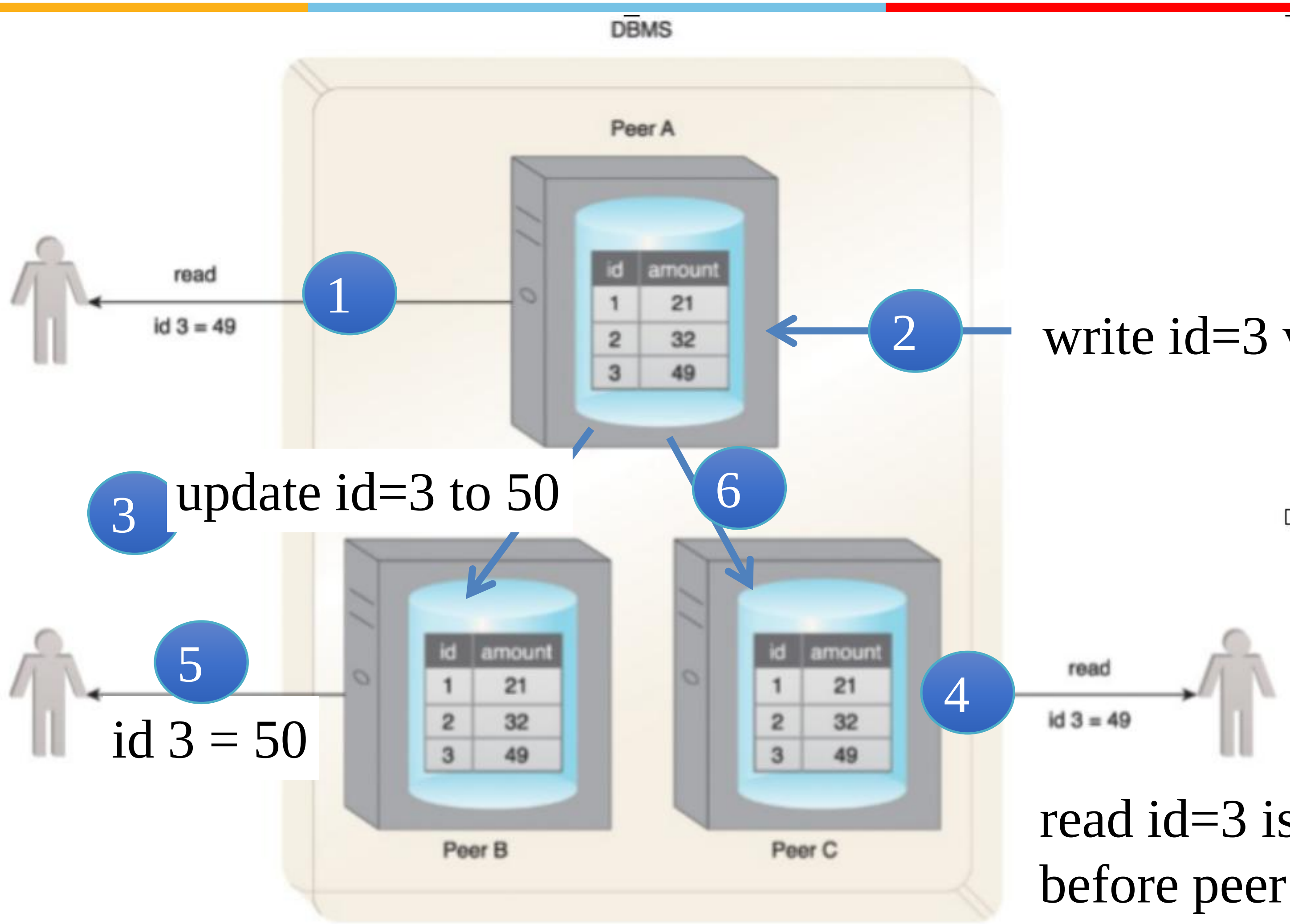
id	amount
1	21
2	32
3	49

id	amount
1	21
2	32
3	49

id	amount
1	21
2	32
3	49

All users get the same value for the amount column even though different replicas are serving the record

When can it be in-consistent



Peer A

id	amount
1	21
2	32
3	49

Peer B

id	amount
1	21
2	32
3	49

Peer C

id	amount
1	21
2	32
3	49

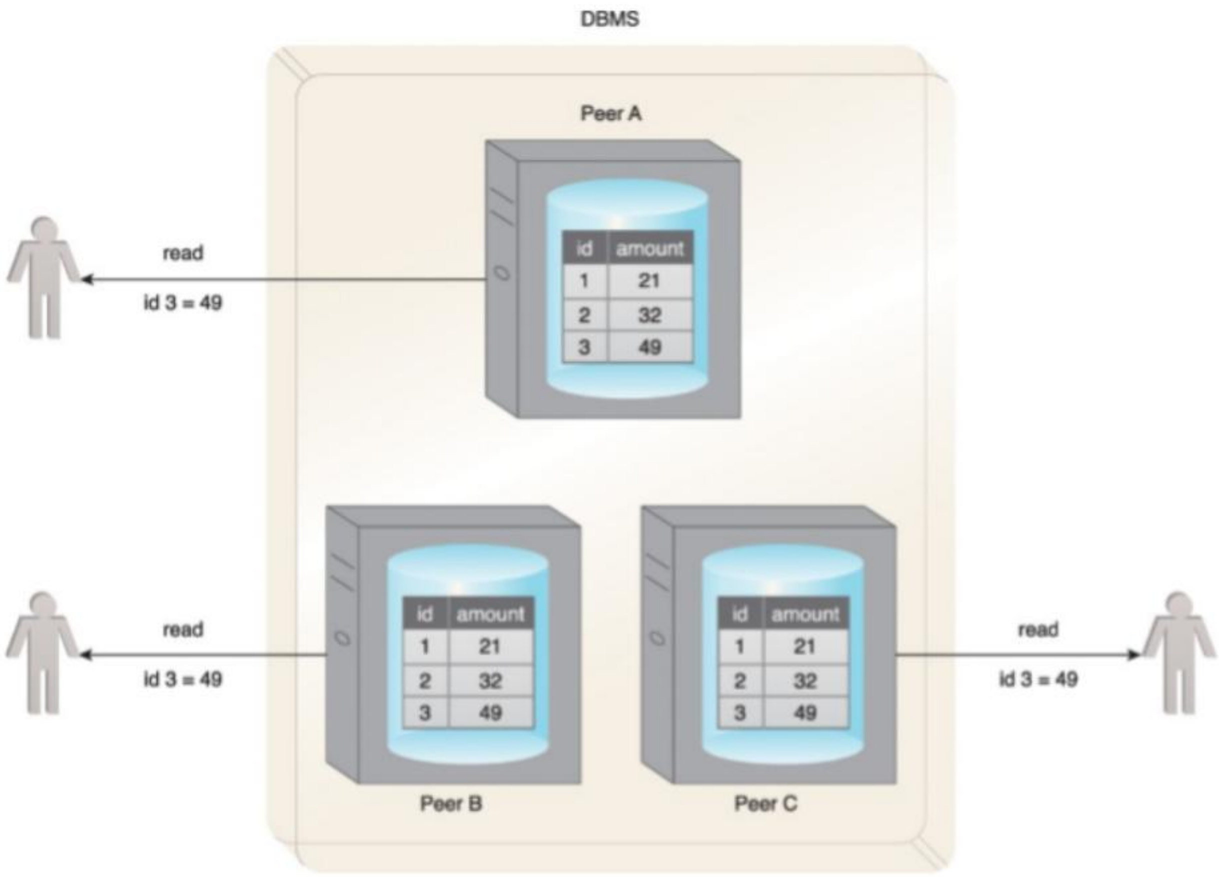
write id=3 value 50

3 update id=3 to 50

id 3 = 50

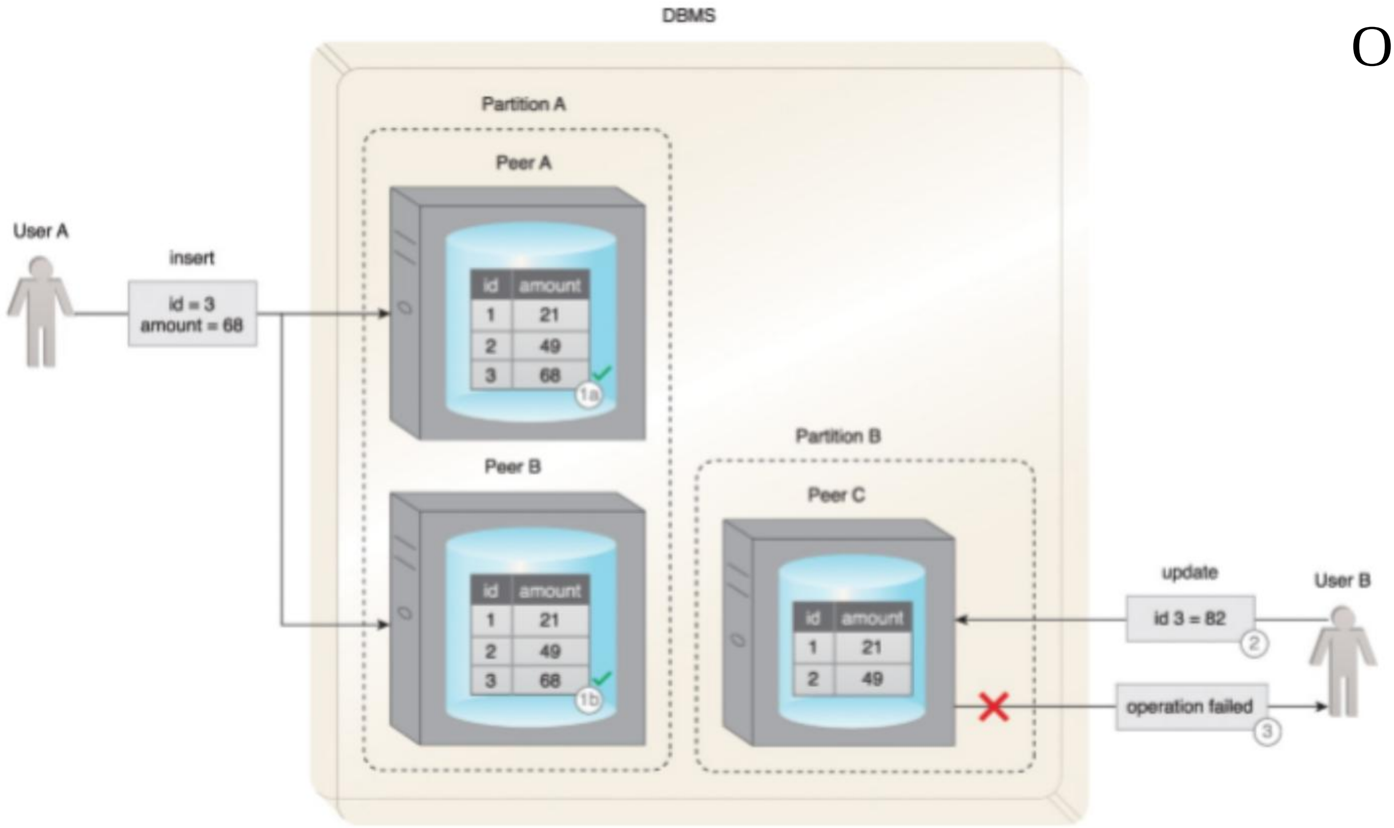
read id=3 is allowed
before peer C is updated
even though write “happens
before” read

Availability



- A request from a user to a peer always responds with a success or failure within a reasonable time.
- Say Peer C is disconnected from the network, then it has 2 options
  - Available: Respond with a failure when any request is made, or
  - Unavailable: Wait for the problem to be fixed before responding, which could be unreasonably long and user request may time out

P: Partition tolerance (1)

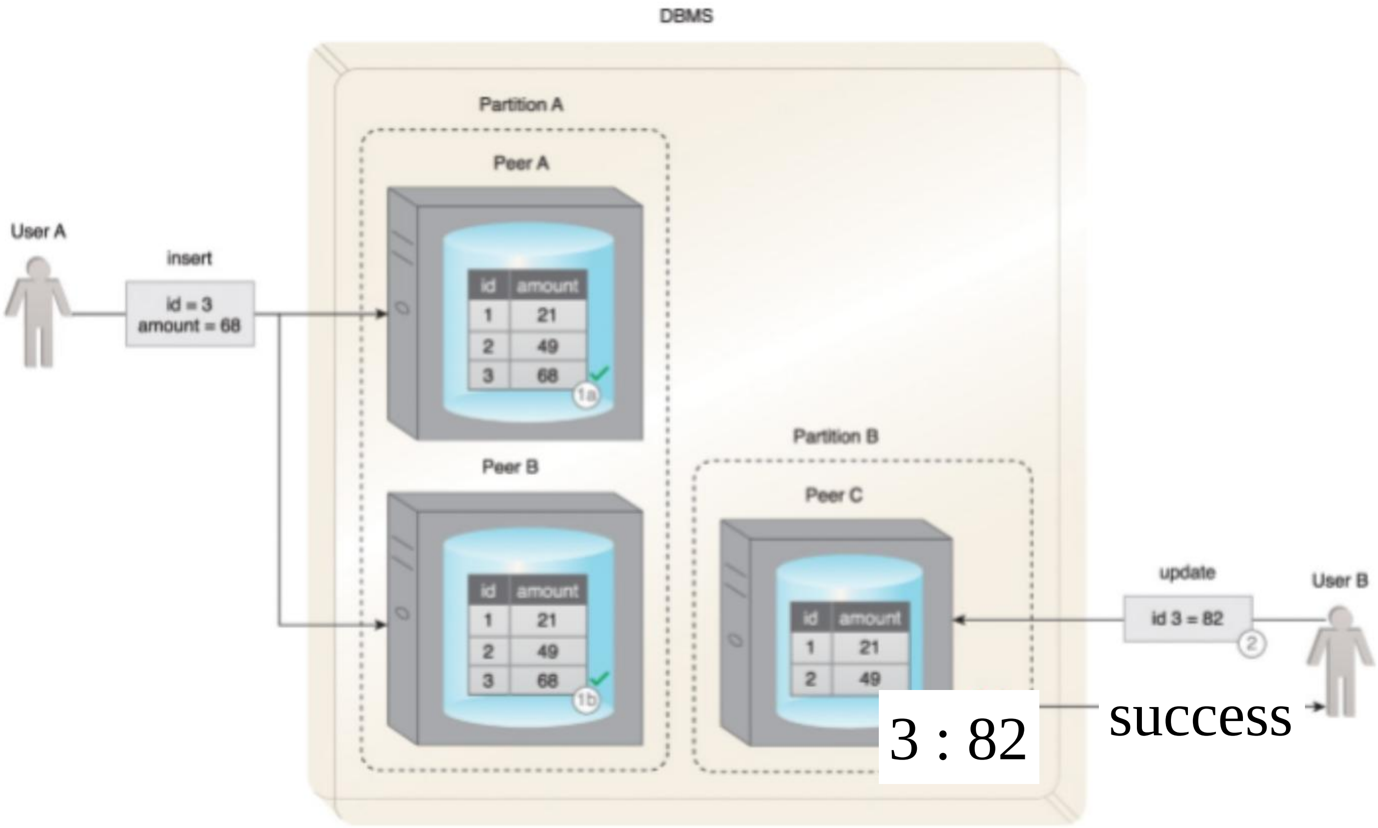


A network partition happens and user wants to update peer C

Option 1:

- Any access by user on peer C leads to failure message response
- System is available because it comes back with a response
- There is consistency because user’s update is not processed
- But there is no partition tolerance
- <u>C and A no P</u>

P: Partition tolerance (2)



3 : 82

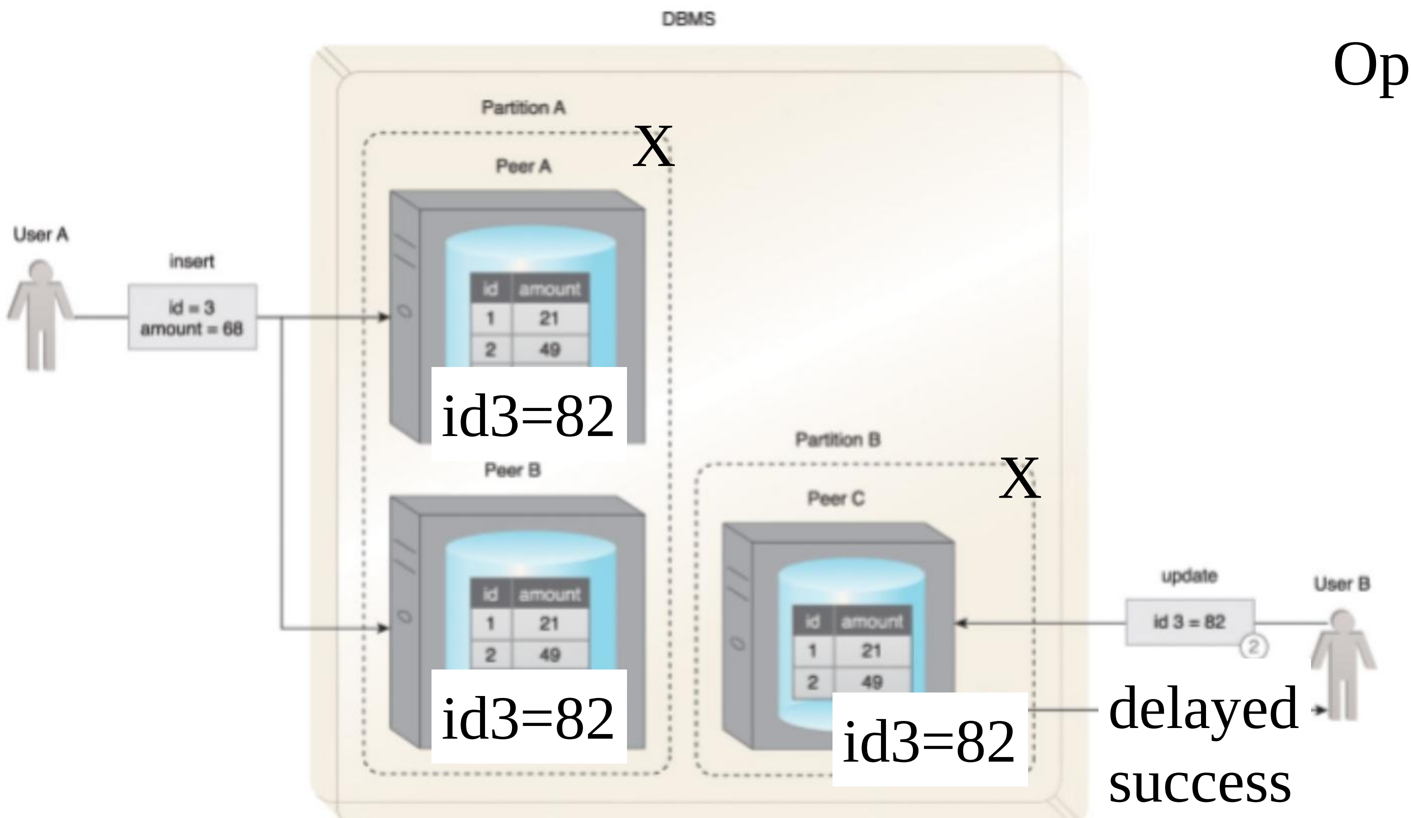
success

A network partition happens and user wants to update peer C

Option 2:

- Peer C records the update by user with success
- System is still available and now partition tolerant.
- But system becomes inconsistent
- <u>P and A but no C</u>

P: Partition tolerance (3)



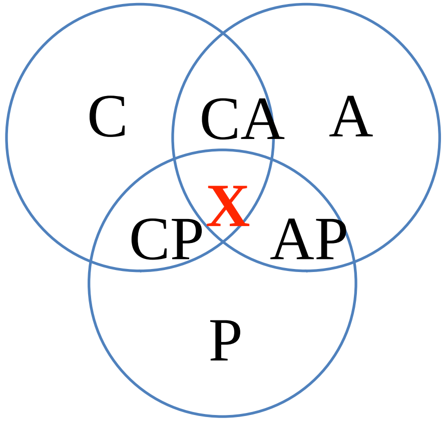
A network partition happens and user wants to update peer C

Option 3:

- User is made to wait till partition is fixed (could be quite long) and data replicated on all peers before a success message is sent
- System appears unavailable to user though it is partition tolerant and consistent
- <u>P and C but no A</u>

CAP Theorem (Brewer's Theorem)

- *A distributed data system, running over a cluster, can only provide two of the three properties C, A, P but not all.*
- So a system can be
  - CA or AP or CP but cannot support CAP
- In effect, when there is a partition, the system has to decide whether to pick consistency or availability.



Consistency or Availability choices

In a distributed database,

- Scalability and fault tolerance can be improved through additional nodes, although this puts challenges on maintaining consistency (C).
- The addition of nodes can also cause availability (A) to suffer due to the latency caused by increased communication between nodes.
  - May have to update all replicas before sending success to client . so longer takes time and system may not be available during this period to service reads on same data item.

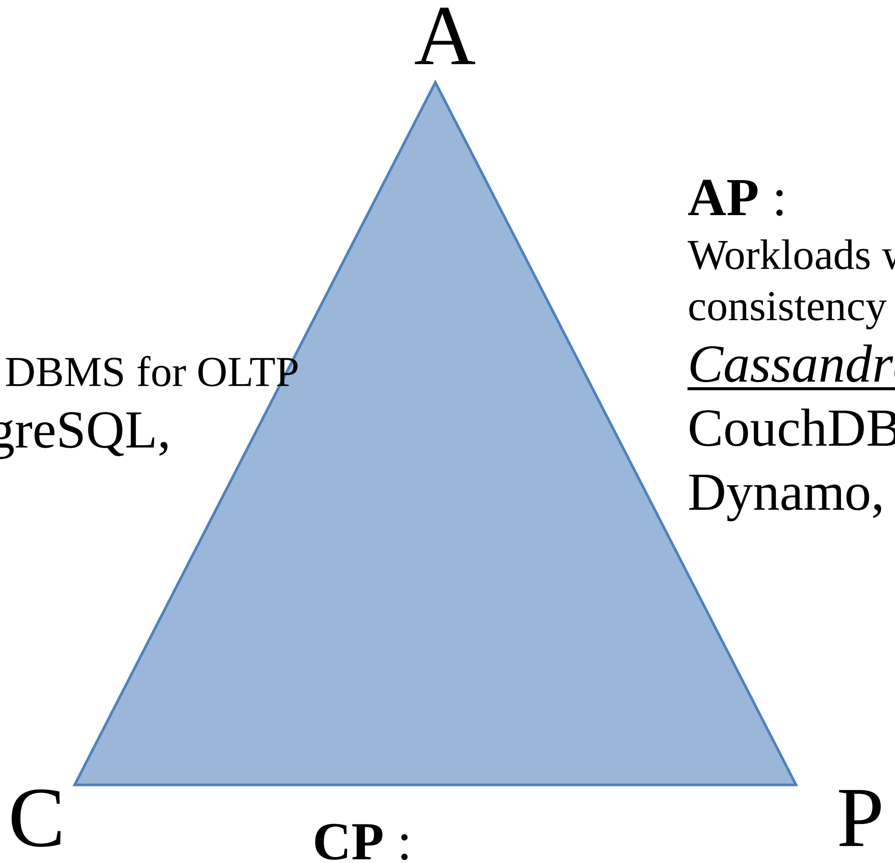
Large scale distributed systems cannot be 100% partition tolerant (P).

- Although communication outages are rare and temporary, partition tolerance (P) must always be supported by distributed database

Therefore, CAP is generally a choice between choosing either CP or AP

Traditional RDBMS systems mainly provide CA for single data items and then on top of that provide ACID for transactions that touch multiple data items.

Database Options



CA :
Traditional DBMS for OLTP
e.g. PostgreSQL,
MySQL

AP :
Workloads where eventual consistency is sufficient
Cassandra,
CouchDB,
Dynamo,

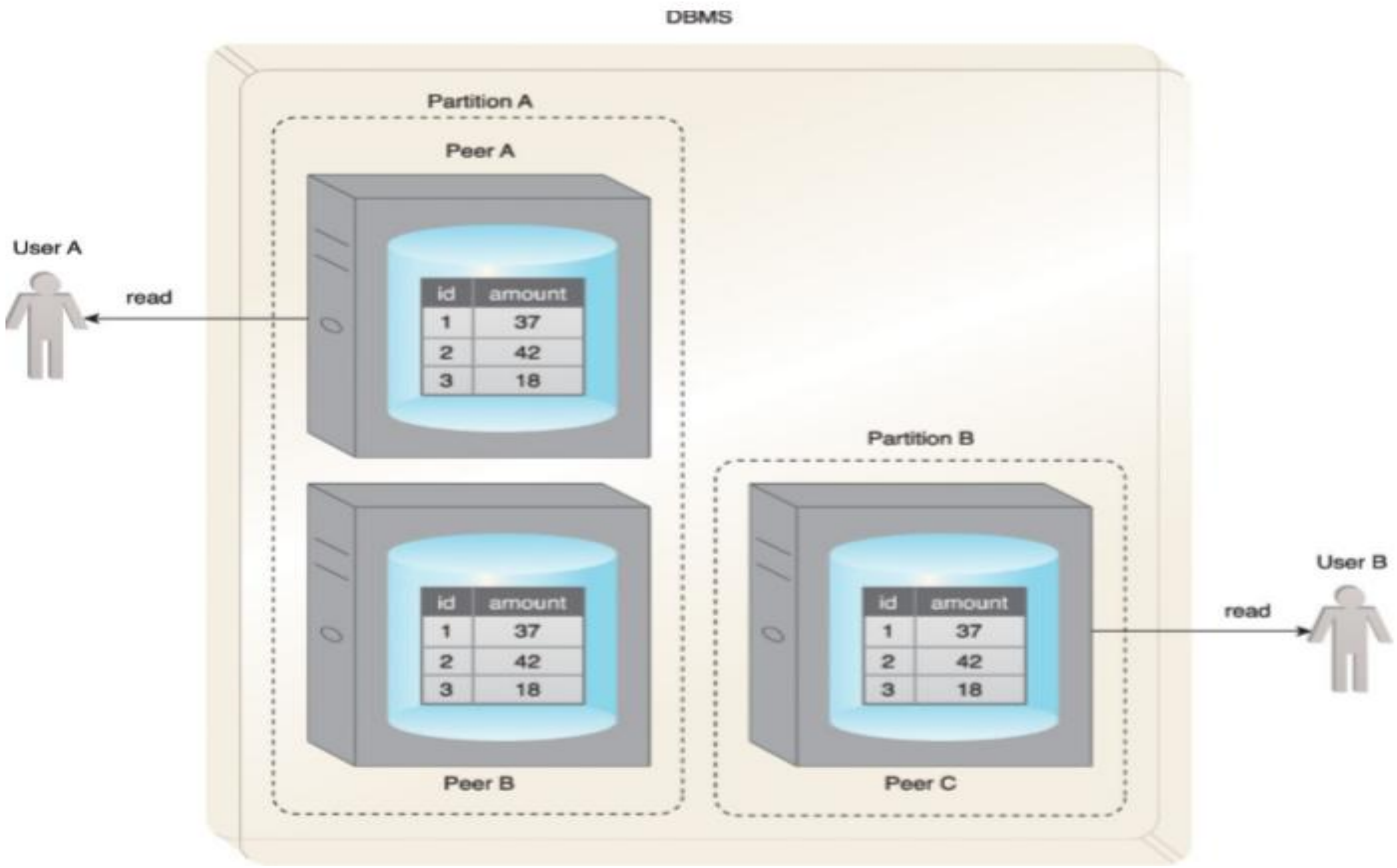
CP :
Can provide consistency with scale.
HBase, *MongoDB*,
Redis

- Different design choices are made by Big Data DBs
- Faults are likely to happen in large scale systems
- Provides flexibility depending on use case to choose C, A, P behavior mainly around consistency semantics when there are faults

BASE

- BASE is a database design principle based on CAP theorem
- Leveraged by AP database systems that use distributed technology
- Stands for
  - ✓ Basically Available (BA)
  - ✓ Soft state (S)
  - ✓ Eventual consistency (E)
- It favours availability over consistency
- Soft approach towards consistency allows to serve multiple clients without any latency albeit serving inconsistent results
- Not useful for transactional systems where lack of consistency is concern
- Useful for write-heavy workloads where reads need not be consistent in real-time, e.g. social media applications, monitoring data for non-real-time analysis etc.

BASE – Basically Available



id	amount
1	37
2	42
3	18

id	amount
1	37
2	42
3	18

id	amount
1	37
2	42
3	18

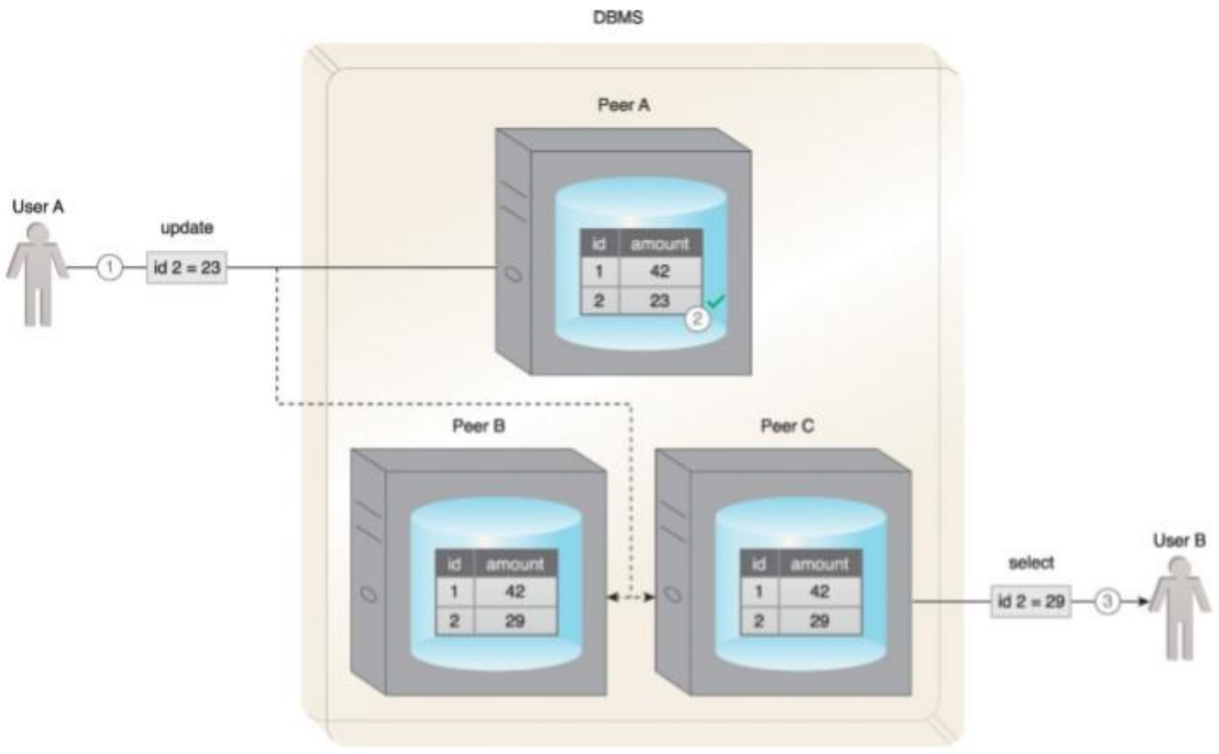
User A and User B receive data despite the database being partitioned by a network failure.

Database will always acknowledge a client’s request, either in form of requested data or a failure notification

BASE – Soft State

- Database may be in inconsistent state when data is read, thus results may change if the same data is requested again
  - ✓ Because data could be uploaded for consistency, even though no user has written to the database between two reads

- User A updated record on node A
- Before the other nodes are updated, User B requests same record from C
- Database is now in a soft state and Stale data is returned to User B



An example of the soft state property of BASE is shown here.

BASE – Eventual Consistency

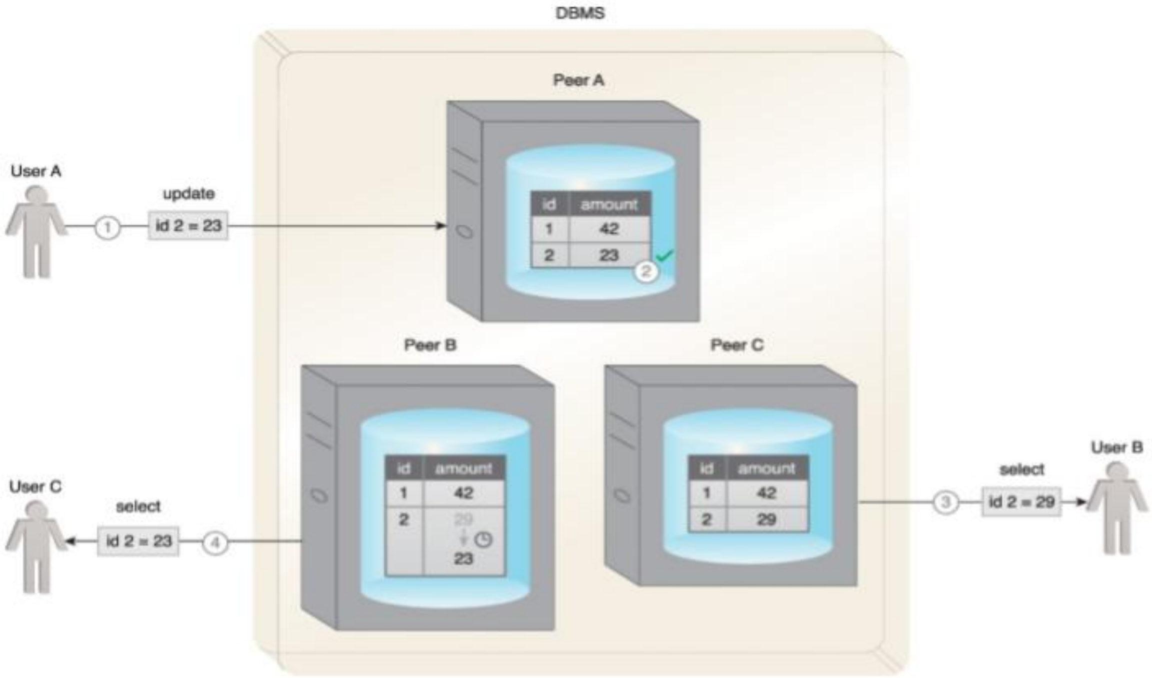
- State in which reads by different clients, immediately following a write to database, may not return consistent results
- Database only attains consistency once the changes have been propagated to all nodes
- While database is in the process of attaining the state of eventual consistency, it will be in a soft state

✓ User A updates a record

✓ Record only gets updated at node A, but before other peers can be updated, User B requests data

✓ Database is now in soft state, stale data is returned to User B from peer C

✓ Consistency is eventually attained, User C gets correct



An example of the eventual consistency property of BASE.

THANK YOU