TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Consistency and Replication

Corso di Sistemi Distribuiti e Cloud Computing
A.A. 2025/26

Valeria Cardellini

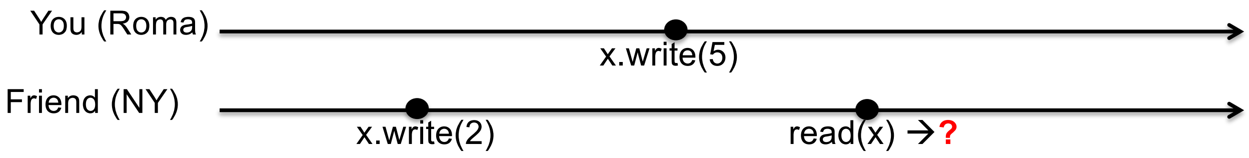
Laurea Magistrale in Ingegneria Informatica

Why replicate data?

- To increase DS availability when servers fail or network is partitioned
  - p = probability that 1 server fails
  - p^n = probability that n servers fail
  - $1-p^n$ = availability of service/system with n servers
    - p=5% and n=1 => service is available 95% of time
    - p=5% and n=3 => service is available 99.9875% of time
- To increase DS fault tolerance
  - Under *fail-stop model*, if up to k of k+1 servers crash, at least one is alive and can be used
    - Fail-stop: failed component simply stops functioning without any additional erroneous behavior
  - Protect against corrupted data
- To improve DS performance through scalability
  - Scale with size and geographical areas

Replication cons

- What does data replication entail?
  - Having multiple copies of same data
- We need to keep replicas **consistent**
  - When one copy is updated we need to ensure that the other copies are updated as well; otherwise the replicas will no longer be the same



Data center in North Carolina
Data center in Ireland

Consistency: the fundamental issue

- Consistency maintenance is itself an issue
- How and when to update replicas?
- How to avoid significant performance loss due to consistency, especially in large scale DS?
  - Latency is non-negligible
    - Inter-data center latency: from 10 ms to 250 ms
    - Even inside data center: ~1 ms
  - and impacts performance
    - Amazon said: *just an extra one tenth of second (i.e., 100 ms) on the response times will cost 1% in sales*
    - Google said: *a half a second (i.e., 500 ms) increase in latency will cause traffic to drop by a fifth*

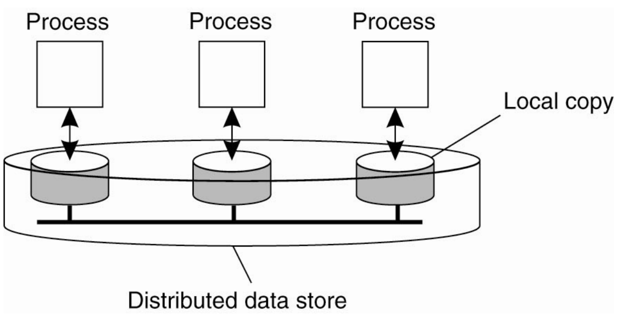
Consistency: what we need

- To keep replicas consistent, we generally need to ensure that all conflicting operations on the same data are done in the the same order everywhere
- Conflicting operations (from transactions world):
  - Read-write conflict: a read operation and a write operation act concurrently
  - Write-write conflict: two concurrent write operations
- Guaranteeing global ordering on conflicting operations may be too costly (requires global synchronization), thus downgrading scalability
- *Solution*: weaken consistency requirements so that global synchronization can be avoided and we get a "consistent" and efficient system

➡ **Different consistency models**

Consistency models

- **Distributed data store**: distributed collection of storage, physically distributed and replicated across multiple processes
  - E.g., distributed database, distributed file system, Cloud storage



- **Consistency model** (or consistency semantics)
  - Contract between a distributed data store and processes, in which the data store specifies precisely what the results of read and write operations are in presence of concurrency

Consistency models

- All consistency models try to return *the last write* operation on the data as a result of data read operation
- Consistency models differ in *how* the last write operation is determined/defined and with respect *to whom*
- **Data-centric** consistency models
  - Goal: provide a system-wide view of a consistent data store
- **Client-centric** consistency models
  - Goal: provide a view of a consistent data store at a single client level
  - Faster but less accurate consistency management than data-centric consistency

Choosing a consistency model

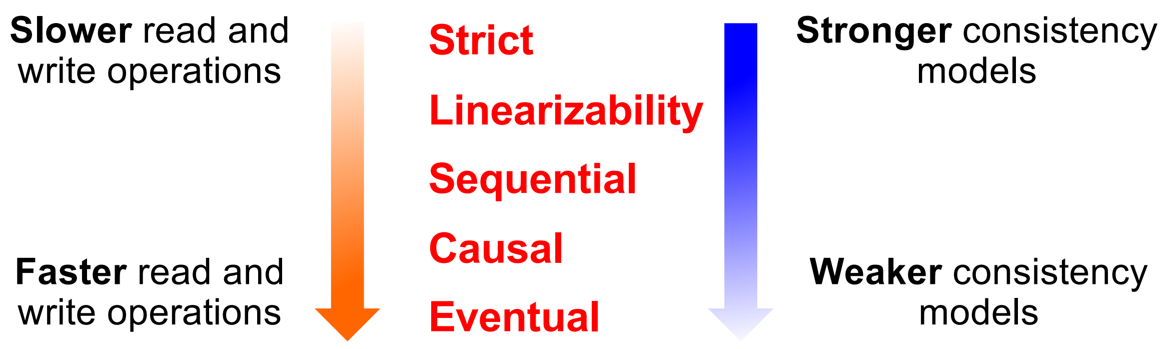
- No right or wrong consistency model
  - There is no unique general solution (i.e., consistency model that fits well all situations) but rather multiple solutions, that are suitable to applications with different consistency requirements
- Non-trivial trade-off among easy of programmability, cost/efficiency, consistency, and availability
  - Low consistency is cheaper but it might result in higher operational cost (e.g., overselling of products in a Web shop)
- Not all data need to be treated at the same level of consistency
  - Consider a Web shop: credit card and account balance information require higher consistency levels, whereas user preferences (e.g., “users who bought this item also bought… ”) can be handled at lower consistency levels

Data-centric consistency models

- Consistency models describe *how* and *when* different data store replicas see operations order
  - Replicas must agree on the global ordering of operations before making them persistent

Data-centric consistency models we study

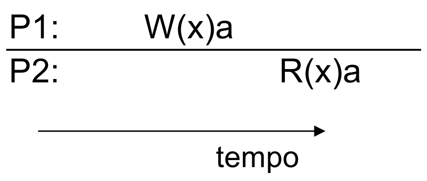
- Main consistency models based on ordering of read and write operations on shared and replicated data


Slower read and write operations
Faster read and write operations
Strict
Linearizability
Sequential
Causal
Eventual
Stronger consistency models
Weaker consistency models

- Strict consistency: strongest model
- Linearizability, sequential, causal and eventual consistency: progressive weakening of strict consistency

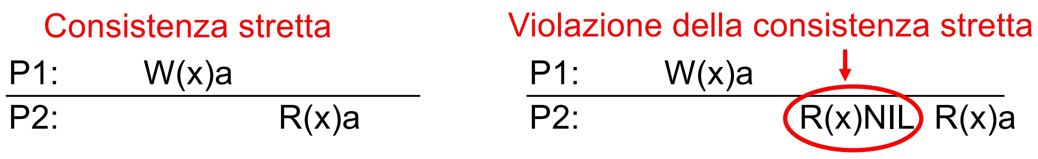
Modelli di consistenza: notazione

- Rappresentiamo il comportamento dei processi che eseguono operazioni di lettura o scrittura sui dati condivisi
  - $W_i(x)a$: operazione di scrittura da parte del processo P_i sul dato x con valore scritto a
  - $R_i(x)b$: operazione di lettura da parte del processo P_i sul dato x con valore letto b



Consistenza stretta: il modello ideale

Qualsiasi read su un dato x ritorna un valore corrispondente al risultato più recente della write su x



- Write eseguita su tutte le repliche come singola operazione atomica
  - E’ come se ci fosse una copia unica, ovvero la write è vista istantaneamente da tutti i processi
- Ordinamento temporale assoluto delle operazioni: richiede un clock fisico globale
  - No ambiguità su “più recente”

Implementing strict consistency

```
P1:      W(x)a
P2:                R(x)a
```

- To achieve it, one would need to ensure:
  - Each read must be aware of, and wait for, each write
    - $R_2(x)a$ aware of $W_1(x)a$
    - Clocks must be strictly synchronized
- But time between instructions << communication time: strict consistency is tough to implement efficiently
- Solution: linearizability and sequential consistency
  - Slightly weaker models than strict consistency
  - Still provide the illusion of single copy
    - From the outside observer, the system should (almost) behave as if there's only a single copy

Linearizability

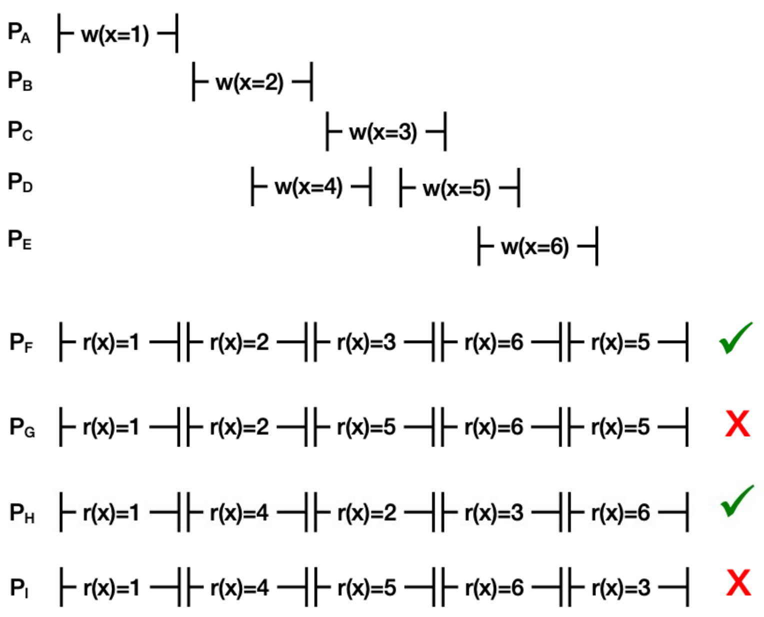
Each operation appears to take effect instantaneously at some point between its start and completion, as if there is a global timeline for all operations

- Operations (op = read, write) receive global timestamp using synchronized clock (e.g., NTP) sometime during their execution
- All replicas execute operations in some total order
- That total order preserves the real-time ordering between operations (and each process' own local ordering)
  - If op A completes before op B begins, then A is ordered before B in real-time
  - If neither A nor B completes before the other begins, then there is no real-time order. But there must be some total order (i.e., same order for A and B)

Linearizability: properties

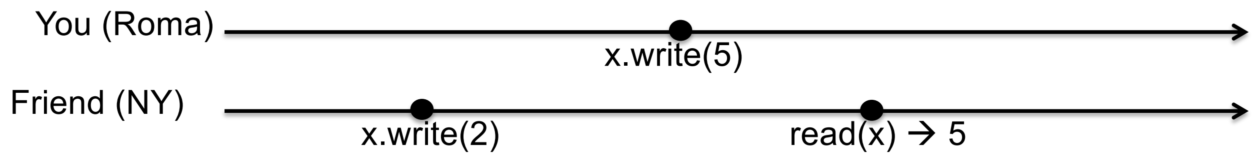
- Weaker model than strict consistency
- Still provides single-client, single-copy semantics
  - That is, read/write behave as if there were
    - a single client making all the requests in a given order
    - over a single copy
- A read op returns the most recent write, regardless of the clients
- All subsequent reads should return the same result until the next write, regardless of the clients
- Does not mandate any particular order for *overlapping operations*
  - The system needs to provide an ordering of ops
  - The ordering should give an illusion that there is a single copy

Linearizability: example



P_A w(x=1)
P_B w(x=2)
P_C w(x=3)
P_D w(x=4) w(x=5)
P_E w(x=6)
P_F r(x)=1 r(x)=2 r(x)=3 r(x)=6 r(x)=5 ✓
P_G r(x)=1 r(x)=2 r(x)=5 r(x)=6 r(x)=5 X
P_H r(x)=1 r(x)=4 r(x)=2 r(x)=3 r(x)=6 ✓
P_I r(x)=1 r(x)=4 r(x)=5 r(x)=6 r(x)=3 X

Linearizability: performance



- How to implement linearizability? (see slide 2)
  - Clients send read/write requests to Ireland DC (primary)
  - Ireland datacenter propagates write to North Carolina DC
  - Read never returns until propagation is done
  - Linearizability? Yes
  - Performance? No, because of WAN latencies
- Linearizability requires *complete* synchronization of multiple replicas before write returns
- It makes less sense in global setting, but still makes sense in local setting (e.g., within a single data center)

Consistenza sequenziale

Il risultato di una qualunque esecuzione è uguale a quello ottenuto se le operazioni (di read e write) da parte di tutti i processi sull'archivio di dati fossero eseguite

- *secondo un* ***ordine sequenziale***
- ***e*** *le operazioni di ogni singolo processo apparissero in questa sequenza* ***nell'ordine specificato dal suo programma***

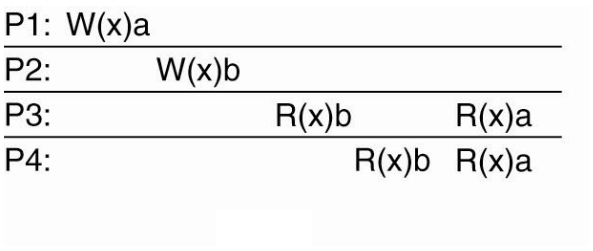
- Quando i processi sono in esecuzione concorrente, qualunque alternanza (*interleaving*) di operazioni è accettabile (purché rispetti l'ordine di programma), ma *tutti i processi vedono la* ***stessa alternanza di operazioni***

Sequential consistency: properties

- Weaker model than linearizability
- Still provides single-client, single-copy semantics
- All replicas execute operations in some total order
- That total order preserves the program order of each process between operations
  - If process P *issues* op A before op B, then A is ordered before B by P's program order (i.e., preserves local ordering)
  - If ops A and B and done by different processes then there is no program order between them. But there must be *some* total order

Sequential consistency: example

- Sequentially consistent data store



- Operation interleavings that satisfy program order of each process (∃ total order + process' program ordering)

$W_2(x)b\ R_3(x)b\ R_4(x)b\ W_1(x)a\ R_4(x)a\ R_3(x)a$

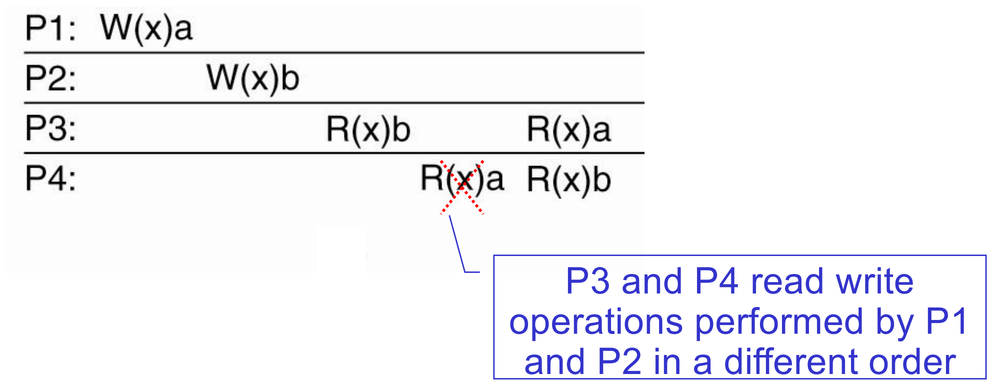
$W_2(x)b\ R_4(x)b\ R_3(x)b\ W_1(x)a\ R_4(x)a\ R_3(x)a$

$W_2(x)b\ R_3(x)b\ R_4(x)b\ W_1(x)a\ R_3(x)a\ R_4(x)a$

$W_2(x)b\ R_4(x)b\ R_3(x)b\ W_1(x)a\ R_3(x)a\ R_4(x)a$

Sequential consistency: example

- Data store that is not sequentially consistent


P1: W(x)a
P2: W(x)b
P3: R(x)b R(x)a
P4: R(x)a R(x)b
P3 and P4 read write operations performed by P1 and P2 in a different order

- We cannot find any interleaving that satisfies the program order of each process, e.g.,
  - $W_1(x)a$ $R_4(x)a$ $R_3(x)a$ $W_2(x)b$ $R_3(x)b$ $R_4(x)b$ violates P3 program order
  - $W_2(x)b$ $R_3(x)b$ $R_4(x)b$ $W_1(x)a$ $R_3(x)a$ $R_4(x)a$ violates P4 program order

Sequential consistency: performance

- Sequential consistency is *programmer-friendly*, but hard to implement efficiently
  - Writes should be applied in the same order across different copies to keep the single-copy illusion

- How to implement sequential consistency?
  - Use a global sequencer (centralized), or
  - a totally ordered multicast protocol (decentralized)

Linearizability vs sequential consistency

- Linearizability is stronger than sequential consistency
- Both provide single-client, single-copy semantics
- With linearizability: interleaving across all processes is pretty much determined on the basis of time
- With sequential consistency: freedom to interleave operations coming from different processes, as long as ordering from each process is preserved

- In a nutshell:
  - Linearizability: ∃ total order + real-time ordering
  - Sequential: ∃ total order + process' program ordering

Casual and eventual consistency

- More relaxed consistency models are often used to increase performance and availability and lower cost
  - **Causal** consistency
  - **Eventual** consistency
- But we lose the single-copy illusion
- Causal consistency
  - We care about ordering causally-related write operations correctly (e.g., Facebook post-like pairs)
- Eventual consistency
  - As long as we can say all replicas converge to the same copy eventually, we're fine

Casual consistency: informal example

- Consider these posts on a social network:
  1. Oh no! My cat just jumped out the window.
  2. [a few minutes later] Whew, the catnip plant broke her fall.
  3. [reply from a friend] I love when that happens to cats!

- Causality violation could result someone else reads:
  1. Oh no! My cat just jumped out the window.
  2. [reply from a friend] I love when that happens to cats!
  3. Whew, the catnip plant broke her fall.

Consistenza causale

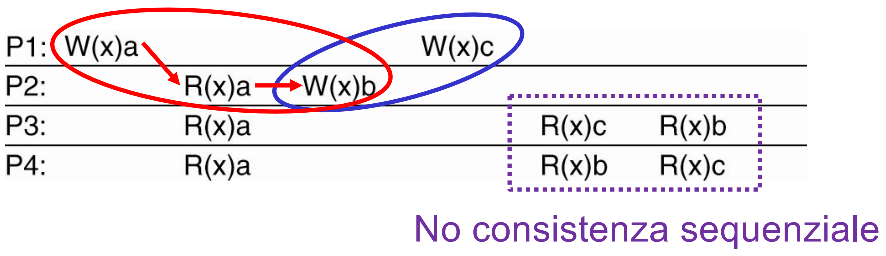
Operazioni di write che sono potenzialmente in relazione di causa/effetto devono essere viste da tutti processi nello stesso ordine. Operazioni di write concorrenti possono essere viste in ordine differente da processi differenti

- In relazione di causa/effetto:
  - read seguita da write sullo stesso processo: write è (potenzialmente) causalmente correlata con read
  - write di un dato seguita da read dello stesso dato su processi diversi: read è (potenzialmente) causalmente correlata con write
  - Si applica la proprietà transitiva: se P1 scrive x e P2 legge x e usa il valore letto per scrivere y, la lettura di x e la scrittura di y sono causalmente correlate
- Se due processi scrivono simultaneamente, le due write non sono causalmente correlate (*write concorrenti*)

- Indebolimento della consistenza sequenziale
  - Distingue tra operazioni che sono potenzialmente in relazione di causa/effetto e quelle che non lo sono

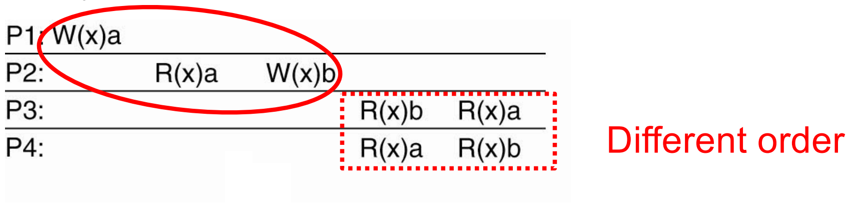
Consistenza causale: esempi

- Esempio di sequenza valida in un archivio di dati causalmente consistente, ma non in un archivio sequenzialmente consistente
  - $W_2(x)b$ e $W_1(x)c$ sono write concorrenti: possono essere viste dai processi in ordine differente
  - $W_1(x)a$ e $W_2(x)b$ sono write in relazione di causa/effetto

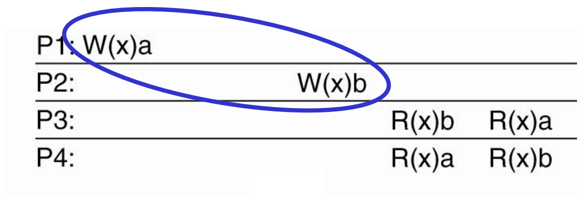


Causal consistency: examples

- Example 1: sequence of operations which is not valid in a causally consistent data store
  - $W_1(x)a$ and $W_2(x)b$ are causally related: must be seen in same order by all processes



- Example 2: sequence of operations which is valid in a causally consistent data store
  - $W_1(x)a$ and $W_2(x)b$ are concurrent: can be seen in different order
  - But not valid in a sequentially consistent data store



Implementing causal consistency

- No longer single-copy illusion
  - Concurrent writes can be applied in different orders across copies
  - Causally-related writes do need to be applied in the same order for all copies
- Thanks to relaxed requirement on writes, latency is less problematic
- However, we need a mechanism to keep track of causally-related writes (i.e., which processes have seen which writes)
  - Build and maintain a dependency graph showing which operations depend on which other operations
  - Or use vector clocks: more amenable for computation

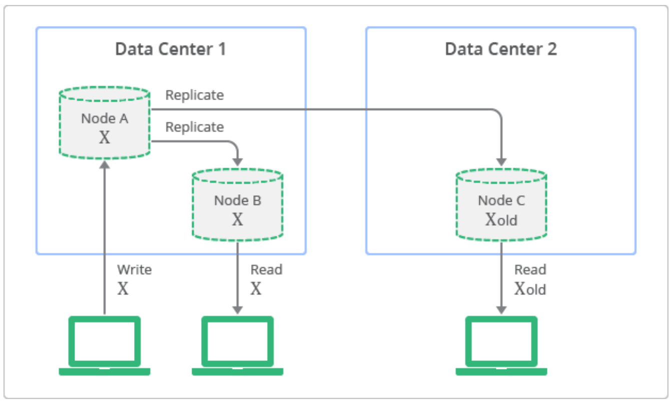
Relaxing consistency further:

- Let’s just do best effort to make things consistent: **eventual consistency**
  - Popularized by CAP theorem

Eventual consistency

- In a distributed data store characterized by
  - Lack of concurrent updates (write-write conflicts) or easy resolution of conflicts
  - Strong prevalence of reads compared to writes (i.e., mostly read)
- we can adopt a relaxed consistency model, called eventual consistency (in Italian: eventual=*finale*)
- What it guarantees: if no new updates occur, *eventually* all reads will return the last updated value
  - That is, all replicas gradually become consistent within a time window (called *inconsistency window*)
  - Without failures, length of inconsistency window depends on: communication latency, number of replicas, system load
- Model used in some Cloud storage services and NoSQL data stores

Eventual consistency vs. strong consistency

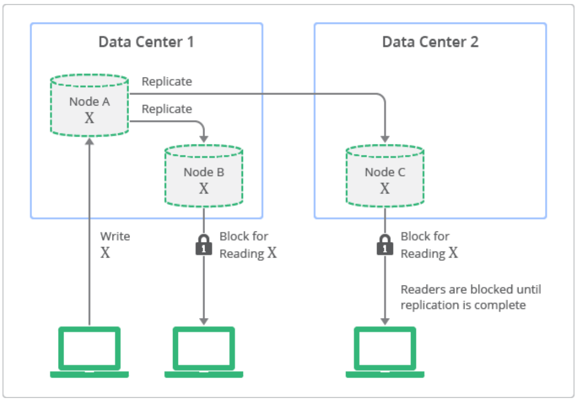


Eventual consistency

- Replicas are always available to read
- But some replica (e.g., C) may be inconsistent with the latest write

Strong consistency (e.g., linearizability)

- Replicas are always consistent
- But replicas are not available until update completes



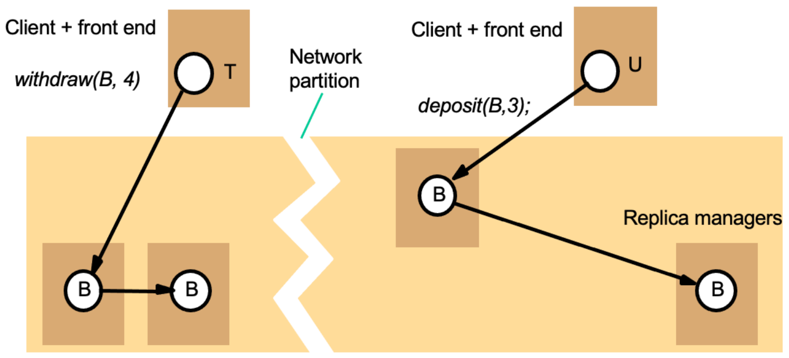
Eventual consistency: pros and cons

- Pros
  - ✓ Simple and inexpensive to implement
  - ✓ Fast reads and writes on local replica
  - – E.g., used by DNS: when authoritative name server updates a resource record, other name servers store it for TTL period
- Cons
  - ✗ No illusion of single copy
  - ✗ Possible data inconsistency (*staleness*) caused by conflicting writes: conflicts must be resolved by means of reconciliation strategy
  - ✗ Can make applications more complex to write: providing stronger consistency falls on developer's shoulder
    - Developer must know which consistency model is provided by data store
    - If read does not return the value of the most recent write, developer must decide whether such inconsistency is acceptable to application

Eventual consistency: reconciliation

- *How* to reconcile conflicting versions of replicas that have diverged due to concurrent updates?
  - – Popular strategy: last write wins
    - Tag data with vector clock as timestamp and use vector clock to capture causality between different versions of data
    - Popular solution in many systems (e.g., Cassandra)
  - – Alternatively, push conflict resolution to application which invokes a user-specified conflict handler (e.g., done by Amazon Dynamo)
- *When* to reconcile?
  - – Usually on read (e.g., Amazon Dynamo) so to provide an "always-writable" experience (but slows down reads)
  - – Alternatives are: on write (reconcile during writes, slowing down them) and asynchronous repair (correction is not part of read or write ops)

Consistency and network partitions



- Dilemma with network partitions
  - To keep replicas consistent, you need to block waiting for replicas update
    - To outside observer, system appears to be unavailable
  - If you don't block and still serve requests from the two partitions, then replicas will diverge
    - System is available, but weaker consistency
- Which choice? **CAP theorem** explains this dilemma

CAP theorem

- CAP theorem
  - Conjecture proposed by E. Brewer in 2000 and formally proved by S. Gilbert and N. Lynch in 2002 under certain conditions
  - One of the most important findings for distributed data stores
- Any networked shared-data system can have **at most two** of the three desirable properties at any given time:
  - **Consistency** (***C***): have a single up-to-date copy of data

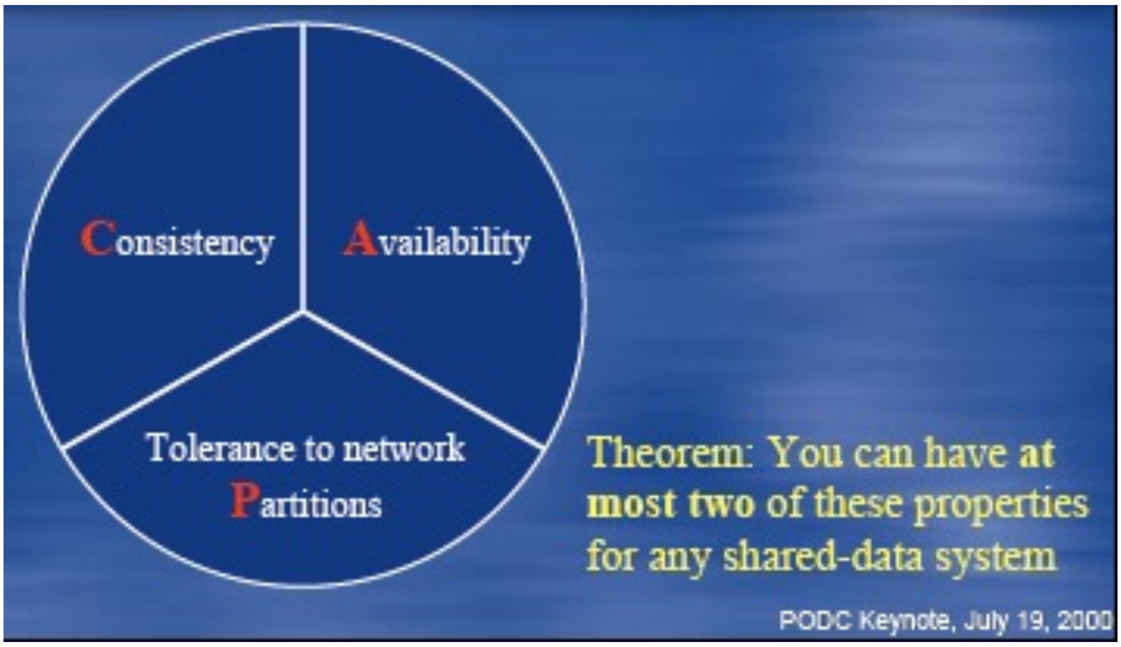
    "All the clients see the same view, even in presence of updates."
  - **Availability** (***A***) of that data (for updates)

    "All clients can find some replica of data, even in presence of failure."
  - **Partition tolerance** (***P***)

    "The system property holds even if the system is partitioned."

Brewer's slides at PODC 2000 https://people.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

CAP theorem


Consistency
Availability
Tolerance to network Partitions
Theorem: You can have at most two of these properties for any shared-data system
PODC Keynote, July 19, 2000

Partition tolerance: why do we care?

- Network partitions can occur across data centers when Internet gets disconnected
  - Internet router outages
  - Under-sea cables cut
  - DNS not working

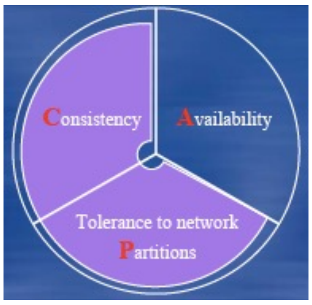
As result of partition, network can lose arbitrarily many messages sent from one node to another

- Network partitions can also occur within a datacenter (e.g., rack switch outage), but less frequently
- We still desire DS to continue functioning normally under network partitions → fix **P**
- According to CAP, consistency and availability cannot be achieved at the same time when partition occurs
- Which one to give up? **C**onsistency or **A**vailability?

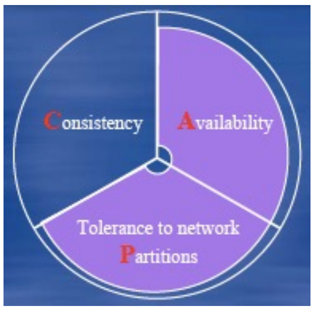
It's a design choice

CAP and network partitions

- If consistency is priority, forfeit availability: **CP** system



- If availability is priority, forfeit consistency: **AP** system
  – Relaxed consistency model: **eventual consistency**

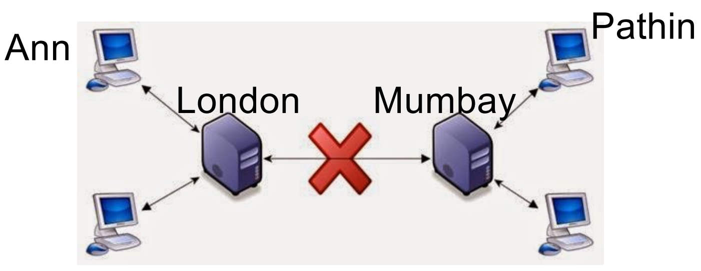


CAP and network partitions

- When using CP and AP systems, developer needs to be aware of what system offers
- CP system: may not be available to take a write
  – If write fails because of system unavailability, developer has to decide what to do with data to be written
- AP system: may always accept a write, but under certain conditions a read will not reflect the result of a recently completed write
  – Developer has to decide whether application requires access to the absolute latest update all the time
- Take-away message: CAP choice depends on application requirements
  – Blog different from financial exchange or shopping cart

CAP: example

- Booking system of Ace Hotel in New York uses a replicated database with master server located in Mumbai and one replica server in London
- Ann is trying to book a room on replica server
- Pathin is trying to do the same on master
- There is only a room available and the network link between the two servers breaks



CAP: example

- CP system:
  - Pathin can book the room
  - Ann can see the room information but cannot book it
- AP system: both servers accept the room booking
  - Overbooking!
- CA system: neither user can book any hotel room
  - No tolerance to network partitions

ACID vs BASE

- ACID and BASE: contrasting approaches to achieving data consistency in DS
- **ACID: A**tomicity, **C**onsistency, **I**solation, **D**urability
  - Pessimistic approach: prevent conflicts from occurring
  - Standard for relational DBMSs: Postgres and MySQL are CA
- **BASE: B**asically **A**vailable, **S**oft state, **E**ventual consistency
  - Optimistic approach: let conflicts occur, but detect them and take action to sort them out
  - *Basically available*: system is available most of the time and there could exist a subsystem temporarily unavailable
  - *Soft state*: data is not durable in the sense that its persistence is left to developer that must take care of it
    - Data is durable if its changes survive failures and recoveries
  - *Eventually consistent*: system eventually converges to a consistent state

ACID vs BASE

- Consistency
  - ACID provides strong consistency, ensuring that data is always in a consistent state
  - BASE provides eventual consistency, allowing temporary inconsistencies but ensuring convergence to a consistent state over time
- Availability
  - ACID system may experience limited availability during certain operations or under system failures, as it prioritizes consistency
  - BASE system prioritizes availability and strives to remain accessible even during failures or network partitions

ACID vs BASE

- Performance
  - ACID system may incur higher latency and performance overhead, requiring synchronous replication and strict consistency enforcement
  - BASE systems can achieve higher throughput and lower latency due to their asynchronous replication and relaxed consistency
- Use cases
  - ACID is well-suited for applications that require strong data integrity and consistency, e.g., financial systems or transactional applications
  - BASE is often used in large-scale DS, NoSQL data stores, and web applications where high availability and horizontal scalability are more critical than strict consistency

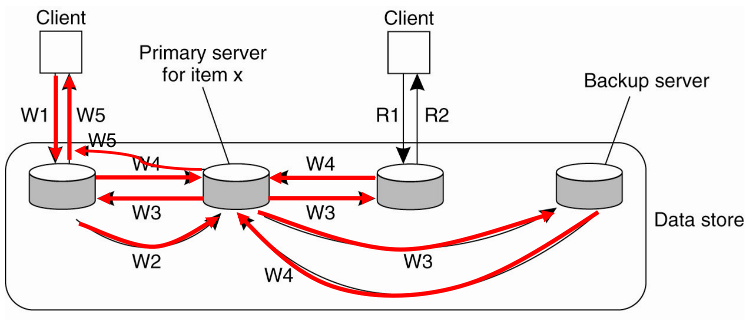
Protocolli di consistenza data-centrica

- Protocollo di consistenza: implementazione di uno specifico modello di consistenza
- Analizziamo protocolli di consistenza data-centrica linearizzabile e sequenziale
  - Protocolli **primary-based**
    - Operazioni di scrittura eseguite su una sola replica (quella primaria), che successivamente assicura che gli aggiornamenti siano opportunamente ordinati ed inoltrati alle altre repliche
  - Protocolli **replicated-write**
    - Operazioni di scrittura eseguite su molteplici repliche

Protocolli primary-based

- Anche detti protocolli primary-backup o di replicazione passiva (o leader-based replication)
  - Ad ogni dato *x* è associata una replica primaria (*leader*) che ha il compito di coordinare le operazioni di *scrittura* di *x* sulle repliche secondarie (*follower*)
  - L'operazione di *lettura* di *x* può essere eseguita su ogni replica (ad es. replica locale al client)
- Protocolli primary-based di tipo remote-write
  - L'operazione di scrittura di *x* è inviata alla replica primaria (eventualmente remota), che poi la inoltra alle repliche secondarie coordinandone l'aggiornamento
- Protocolli primary-based di tipo local-write
  - La copia primaria di *x* *migra* verso la replica locale rispetto al client per l'operazione di scrittura; la replica locale inoltra l'operazione di scrittura alle altre repliche
  - Non esaminato

Primary-based: protocolli remote-write



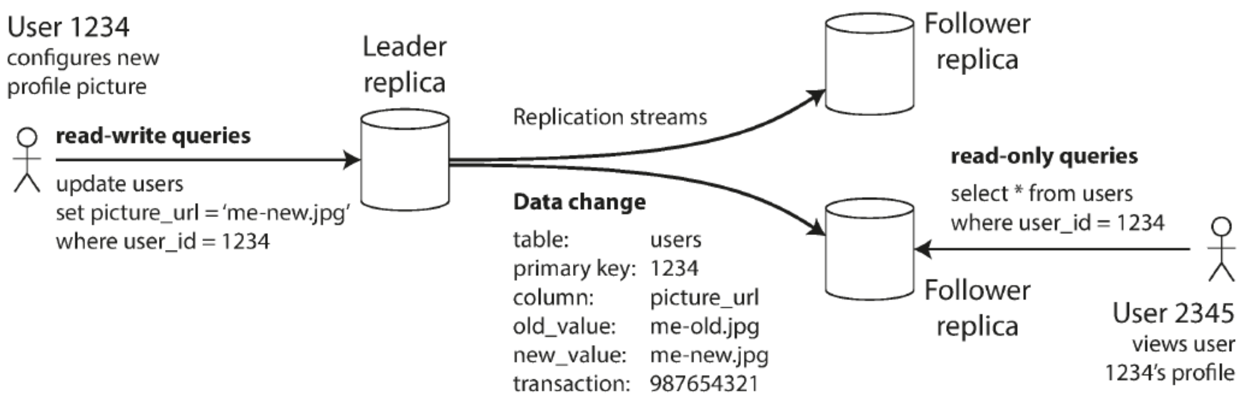
W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

In figura: protocollo remote-write *bloccante*

- Tipicamente usati nei DB distribuiti (e.g., MySQL, PostgreSQL), in alcuni data store NoSQL (e.g., MongoDB), nei MQS (Kafka) e file system distribuiti, ovvero quando si richiede un elevato grado di tolleranza ai guasti
- Svantaggi
  - Lentezza in caso di repliche distribuite geograficamente
  - Scarsa scalabilità all'aumentare del numero di repliche

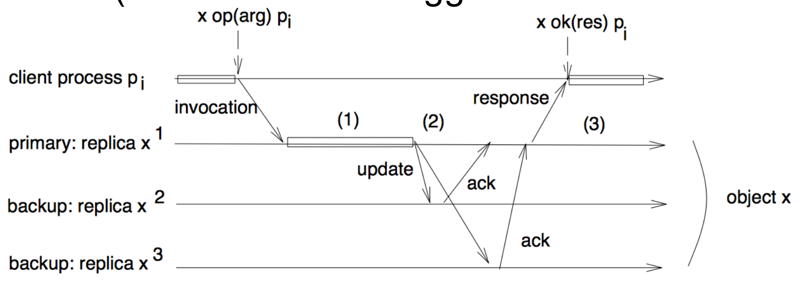
Primary-based remote-write: example



Primary-based: bloccante

- Aggiornamento delle repliche da parte della replica primaria tramite *log shipping*
- Aggiornamento delle repliche in modo *bloccante* o *non bloccante* per il client

1. Bloccante (o replicazione sincrona):
   - La replica primaria notifica al client che la scrittura è stata completata su tutte le repliche
   - Modello di consistenza: linearizzabilità
   - Vantaggi: maggiore tolleranza a guasti (repliche sincronizzate), incluso crash della replica primaria
   - Svantaggi: lentezza (il client attende l'aggiornamento di tutte le repliche)



Primary-based: non bloccante

2. Non bloccante (o replicazione asincrona):
   - La replica primaria notifica al client che la scrittura è stata completata solo su di essa
   - Modello di consistenza: sequenziale
   - Vantaggi: minore attesa per il client, più adatto per repliche in numero elevato e distribuite geograficamente
   - Svantaggi: minore tolleranza ai guasti e perdita della linearizzabilità

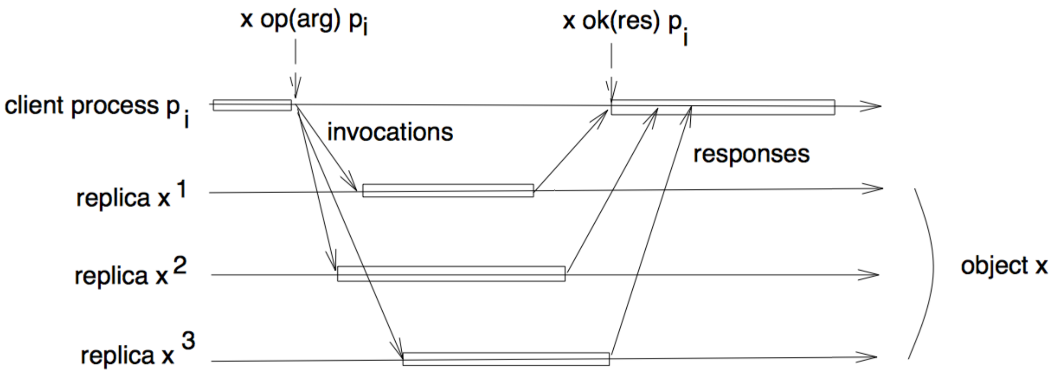
Protocolli replicated-write

- Rispetto ai protocolli primary-based:
  - No controllo centralizzato delle scritture da parte della replica primaria
  - Scritture eseguite su molteplici repliche

- Approcci
  - **Replicazione attiva** (o multi-leader replication)
  - **Protocolli basati su quorum**

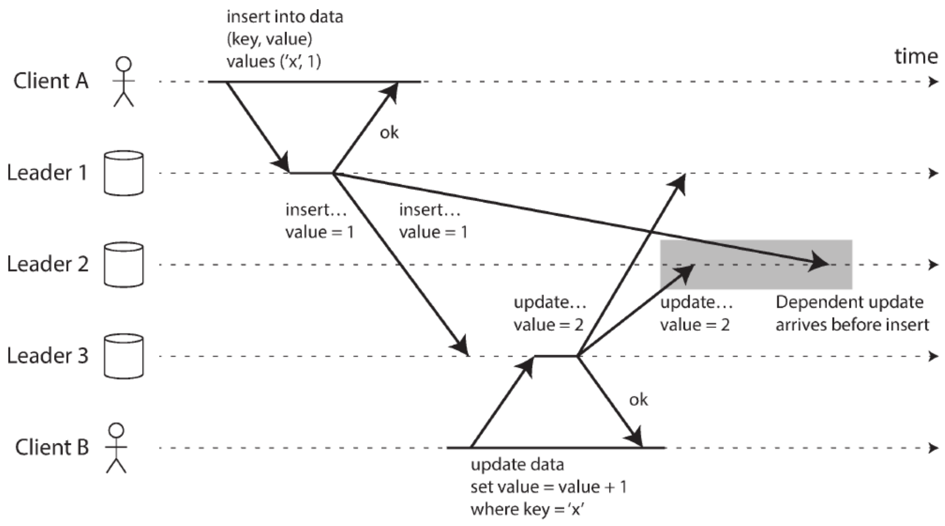
Replicated-write: replicazione attiva

- Replicazione attiva
  - Lettura su replica locale
  - Scrittura su ogni replica
- Soluzione già esaminata (state-machine replication): perché?
  - Per mantenere la consistenza delle repliche, occorre inviare in multicast la scrittura a tutte le repliche



Replicated-write: replicazione attiva

- Qual è il problema da risolvere?
  - Scritture in ordine diverso sulle repliche


insert into data (key, value) values ('x', 1)
time
Client A
ok
Leader 1
insert… value = 1
insert… value = 1
Leader 2
update… value = 2
update… value = 2
Dependent update arrives before insert
Leader 3
ok
Client B
update data set value = value + 1 where key = 'x'

Replicated-write: replicazione attiva

- Occorre eseguire le operazioni di scrittura nello stesso ordine su tutte le repliche
- Come? Multicasting totalmente ordinato
  - Centralizzato tramite sequencer
    - Scarsa scalabilità e single point of failure
  - Decentralizzato usando clock scalare
    - Scalabilità limitata in sistemi a larga scala a causa di elevato numero di messaggi
  - Decentralizzato usando un protocollo di consenso distribuito
    - E.g., Raft
- Modello di consistenza data-centrica supportato
  - Consistenza sequenziale
  - Consistenza linearizzabile

Replicated-write: protocolli quorum-based

- Votazione attuata da un *sottoinsieme* di repliche
- Consideriamo *N* repliche di un dato x
  - Quindi un totale di *N* voti
- Ad ogni dato è associato un numero di versione
  - Ad ogni operazione di scrittura, il numero di versione viene incrementato
- L'operazione di lettura di x richiede un quorum per la lettura N_R per garantire che venga letta l'ultima versione di x
- L'operazione di scrittura su x richiede un quorum per la scrittura N_W per assegnare il numero di versione

Gifford, Weighted voting for replicated data, *Proc. ACM* SOSP 1979
https://dl.acm.org/doi/pdf/10.1145/800215.806583

Protocolli quorum-based: condizioni

- Per N_R e N_W valgono le condizioni

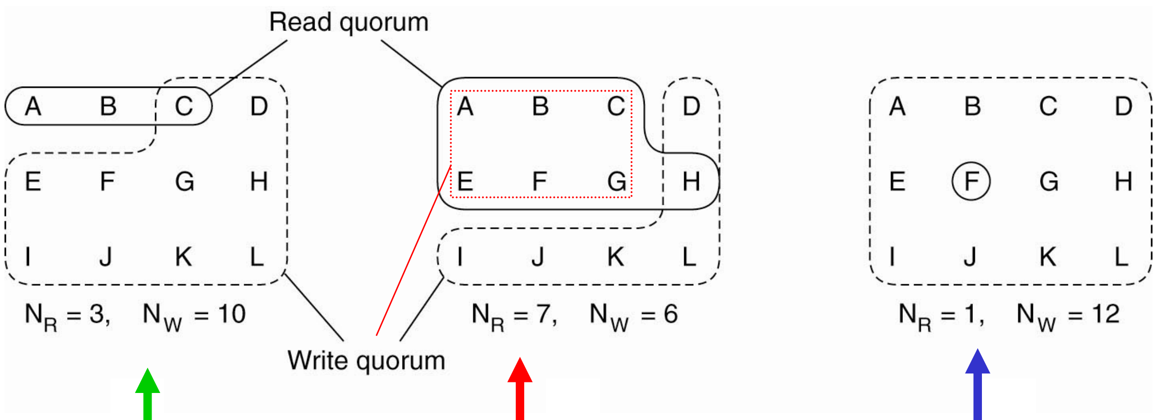
  a) $N_R + N_W > N$

  b) $N_W > N/2$

a) per impedire conflitti lettura-scrittura
b) per impedire conflitti scrittura-scrittura (un solo scrittore alla volta può ottenere il quorum per la scrittura)

Se a) e b) sono entrambe soddisfatte, si garantisce la consistenza sequenziale

Setting read and write quorums



Correct choice for N_R and N_W

Correct choice for N_R and N_W

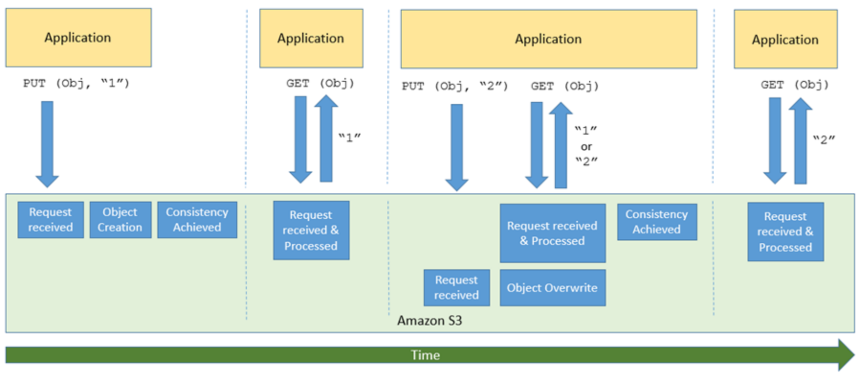
Write-write conflicts may occur (being $N_w <= N/2$)

Setting read and write quorums

- Some specific settings for N_R and N_W
  1. N_R=1 e N_W=N
     - Called *ROWA* (Read Once Write All)
     - Fast reads but slow writes
  2. N_W=1 e N_R=N
     - Called *RAWO* (Read All Write Once)
     - Fast writes but slow reads
     - Be careful: conflicting writes may occur (being $N_W <= N/2$)
  3. N_W= N_R=N/2 +1
     - Called *Majority*
     - Both reads and writes are relatively slow, but high availability
- *Practical use*: quorum-based storage systems allow users to choose between strong and eventual consistency by selecting different read and write quorums (e.g., Cassandra)

Cloud storage and consistency

- Cloud storage services have often adopted weak consistency models
  – Eventual consistency: most popular for a long time
  – New trend towards strong consistency, see AWS S3
- AWS S3 (until 2020)
  – Eventual consistency: after a PUT call, inconsistency window where data has been accepted and durably stored, but not yet visible to all GET or LIST requests



Cloud storage and consistency

- Amazon S3 consistency model (now)
  https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html#ConsistencyModel

  "Amazon S3 provides strong read-after-write consistency for PUT and DELETE requests of objects in your Amazon S3 bucket in all AWS Regions. This behavior applies to both writes to new objects as well as PUT requests that overwrite existing objects and DELETE requests"

  But…

  "Amazon S3 does not support object locking for concurrent writers. If two PUT requests are simultaneously made to the same key, the request with the latest timestamp wins. If this is an issue, you will need to build an object-locking mechanism into your application"

  "Bucket configurations have an eventual consistency model. This means that if you delete a bucket and immediately list all buckets, the deleted bucket might still appear in the list."

Data store systems and consistency

- Some NoSQL data stores offer tunable consistency: user can tradeoff between consistency and latency
- Amazon's DynamoDB: user can choose eventually consistent reads or strongly consistent reads
  https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html
  – Strongly consistent reads experience higher read latency, twofold reduction in read throughput and cost more
- Similarly for Google's Cloud Datastore
  https://cloud.google.com/datastore/docs/articles/balancing-strong-and-eventual-consistency-with-google-cloud-datastore
- Cassandra provides quorum-based consistency, where quorums are configurable
  https://cassandra.apache.org/doc/5.0/cassandra/architecture/dynamo.html#tunable-consistency
  – N+W > R and W >= N/2 +1: strong consistency but higher latency

References

- Sections 7.1, 7.2, and 7.5 of van Steen & Tanenbaum book
- Sections 18.3 of Coulouris et al. book