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. 2022/23

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

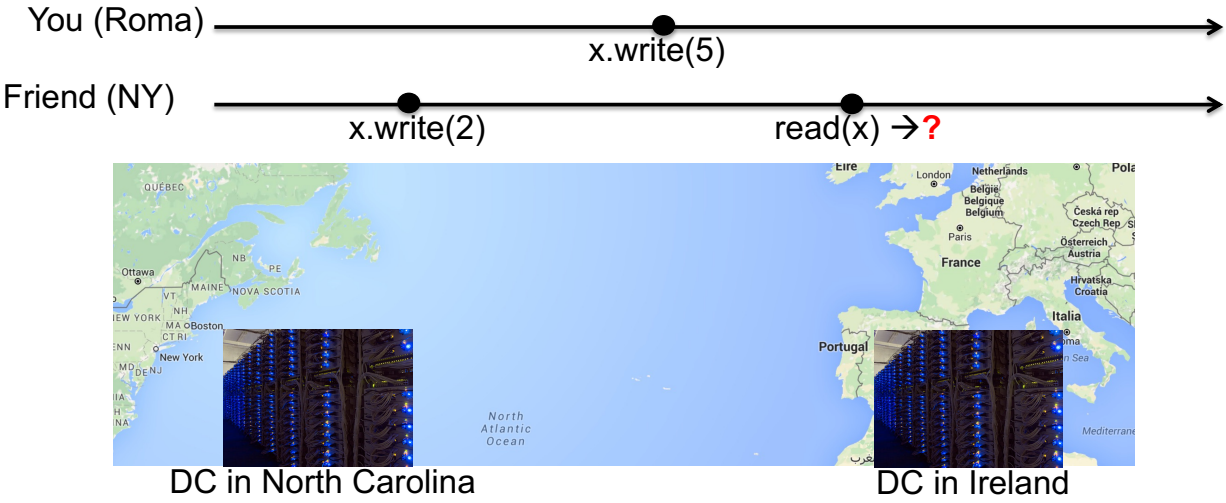
Pros of replication

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 the fail-stop model (see slides on fault tolerance), if up to k of k+1 servers crash, at least one is alive and can be used
  - Protect against corrupted data
- To improve DS performance through scalability
  - Scale with size and geographical areas

Cons of replication

- What does data replication entail?
  - Having multiple copies of the 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



DC in North Carolina DC in Ireland

Consistency issues

- Consistency maintenance is itself an issue
- How and when to update the replicas?
- How to avoid significant performance loss due to consistency, especially in large scale DS?
  - Remember that latency is non-negligible…
    - Inter-data center latency: from 10 ms to 250 ms
      https://medium.com/@sachinkagarwal/public-cloud-inter-region-network-latency-as-heat-maps-134e22a5ff19
    - Even inside data center: ~1 ms
  - and may seriously impact on 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*

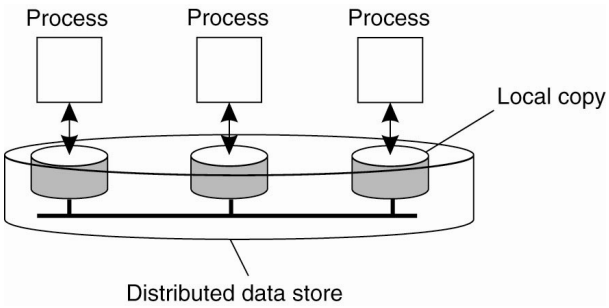
Consistency: what we need

- 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 a costly operation (since requires global synchronization), thus downgrading scalability
- *Solution*: weaken consistency requirements so that hopefully 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 the 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
- A range of 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 policy than data-centric consistency

Choosing a consistency model

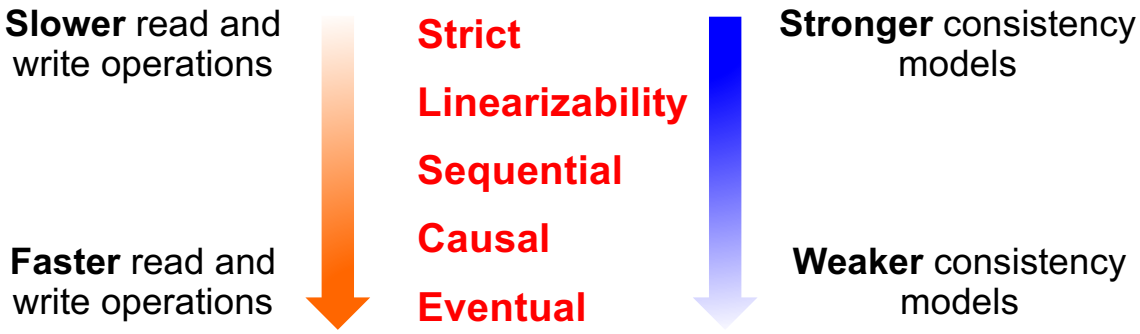
- No right or wrong consistency model
  - There is no unique general solution (i.e., consistency model that fits well all situations) but rather multiple solutions, that are suitable to applications with different consistency requirements
- Non-trivial trade-off among easy of programmability, cost/efficiency, consistency and availability
  - Low consistency is cheaper but it might result in higher operational cost because of, 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

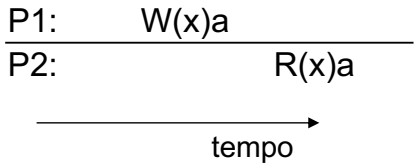
- Main consistency models based on ordering of read and write operations on shared and replicated data



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

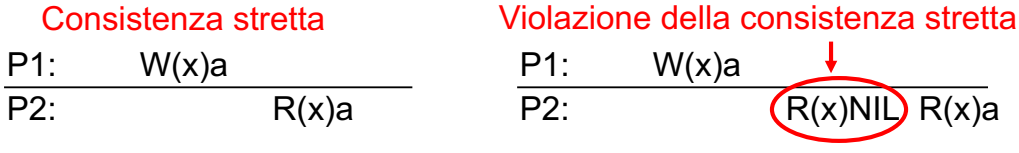
Modelli di consistenza: notazione

- 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
- La consistenza stretta impone un ordinamento temporale assoluto di tutti gli accessi all'archivio di dati e richiede un clock fisico globale
  - Nessuna 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 $W_1(x)a$
    - Real-time clocks must be strictly synchronized
  - But time between instructions << communication time
- Therefore, 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

Consistenza sequenziale

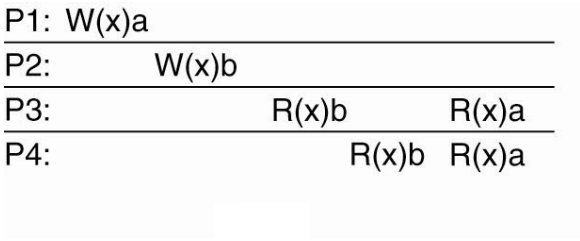
Il risultato di una qualunque esecuzione è uguale a quello ottenuto se le operazioni (di read e write) da parte di tutti i processi sull'archivio di dati fossero eseguite

- *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: example

- A sequentially consistent data store



- Allowable operation interleavings that satisfy the program order of each process

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

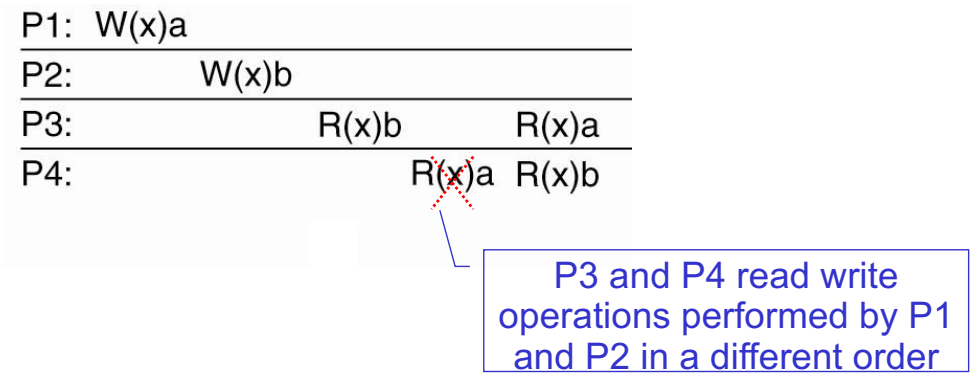
  $W_2(x)b\ R_4(x)b\ R_3(x)b\ W_1(x)a\ R_4(x)a\ R_3(x)a$

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

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

Sequential consistency: example

- A data store that is not sequentially consistent



- We cannot find an allowable interleaving, 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: properties

- Weaker model than strict consistency
  - No global clock
- Read/write should behave as if there were
  - a single client making all the (combined) requests in a given sequential order
  - over a single copy

Linearizability

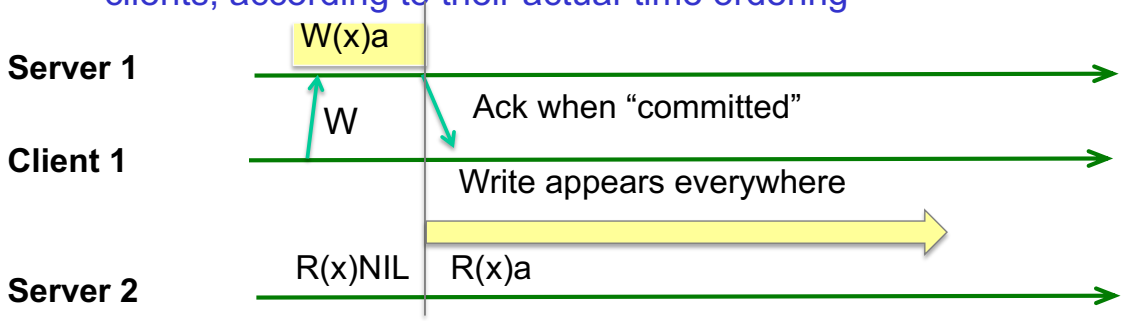
Each operation should appear to take effect instantaneously at some point between its start and completion

- All operations (OP = read, write) receive a global timestamp using a synchronized clock (e.g., NTP) sometime during their execution
- Requirements for sequential consistency, plus *operations are ordered according to a wall-clock time*
  - Timestamp-based ordering: if $ts_{OP1}(x) < ts_{OP2}(y)$, then OP1(x) appears before OP2(y) in the order
- Therefore, linearizability is weaker than strict consistency, but stronger than sequential consistency

Strict > Linearizability > Sequential

Linearizability

- Linearizability (like sequential consistency) provides single-client, single-copy semantics
  - Plus: a read returns *the most recent* write, regardless of the clients, according to their actual-time ordering

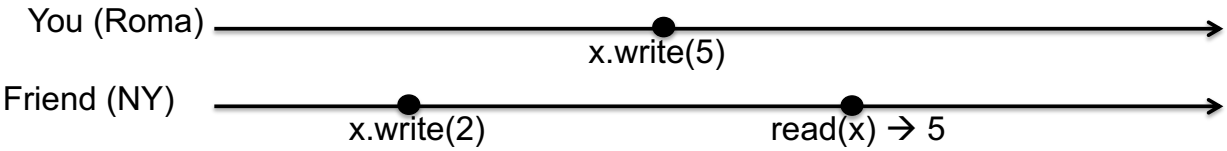


- However, linearizability does not mandate any particular order for *overlapping operations*
  - You can implement a particular ordering strategy
  - As long as there is a single, interleaving ordering for overlapping operations, it's fine

Linearizability vs sequential consistency

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

Performance of linearizability



- How to implement linearizability?
  - Clients send all read/write requests to Ireland datacenter (primary)
  - Ireland datacenter propagates write to North Carolina datacenter
  - Read never returns until propagation is done
  - Correctness (linearizability)? Yes
  - Performance? No, must wait for WAN write
- Linearizability typically requires *complete* synchronization of multiple copies before a write operation returns
- It makes less sense in global setting, but still makes sense in local setting (e.g., within a single datacenter)

Performance of sequential consistency

- Sequential consistency is *programmer-friendly*, but difficult to implement efficiently
  - Writes should be applied in the same order across different copies to give the illusion of a single copy
- How to implement sequential consistency? Using:
  - A global sequencer (centralized)
  - A totally ordered multicast protocol (decentralized)
- We will study its implementation (i.e., primary-based and replicated-write protocols)

Casual and eventual consistency

- Even more relaxed consistency models are often used in order to achieve better performance, lower cost and better availability
  - **Causal** consistency
  - **Eventual** consistency
- But we lose the illusion of a single copy
- 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

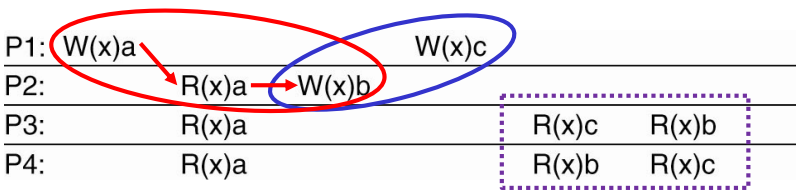
Operazioni di write che sono potenzialmente in relazione di causa/effetto devono essere viste da tutti processi nello stesso ordine. Operazioni di write concorrenti possono essere viste in ordine differente da processi differenti

- 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

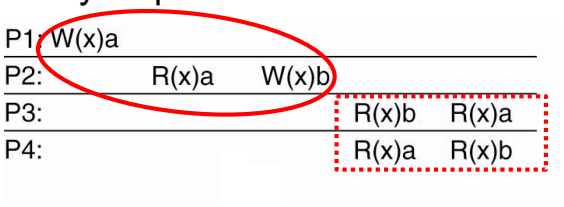
- Esempio di sequenza valida in un archivio di dati causalmente consistente, ma non in un archivio sequenzialmente consistente
  - $W_2(x)b$ e $W_1(x)c$ sono write concorrenti: possono essere viste dai processi in ordine differente
  - $W_1(x)a$ e $W_2(x)b$ sono write in relazione di causa/effetto



No consistenza sequenziale

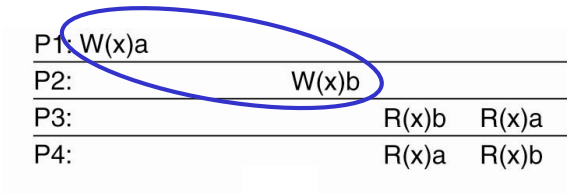
Causal consistency: examples

- 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



Different order

- 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

- We lose the illusion of a single copy
  - 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 requirements, latency is more tractable than in sequential consistency
- 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

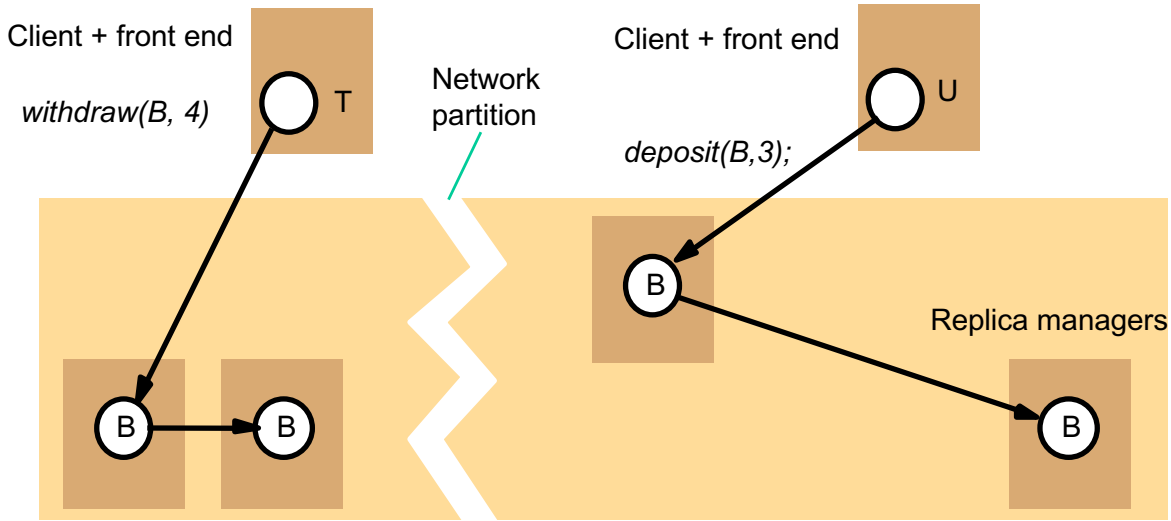
Sintesi dei modelli di consistenza

- Modelli di consistenza data-centrici basati sull'ordinamento delle operazioni

Consistenza	Descrizione
Stretta	Tutti i processi vedono gli accessi condivisi nello stesso ordine assoluto di tempo
Linearizzabile	Tutti i processi vedono gli accessi condivisi nello stesso ordine: gli accessi sono ordinati in base ad un timestamp globale (non unico)
Sequenziale	Tutti i processi vedono gli accessi condivisi nello stesso ordine; gli accessi non sono ordinati temporalmente
Causale	Tutti i processi vedono gli accessi condivisi correlati causalmente nello stesso ordine

Relaxing consistency even further

- Let's just do best effort to make things consistent: **eventual consistency**
  - Popularized by CAP theorem
- Main problem is network partitions



Dilemma with network partitions

- In the presence of 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

- Which kind of consistency in a large scale distributed system?
- CAP theorem
  - Conjecture first proposed by E. Brewer in 2000 and formally proved by S. Gilbert and N. Lynch in 2002 under certain conditions
- 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)

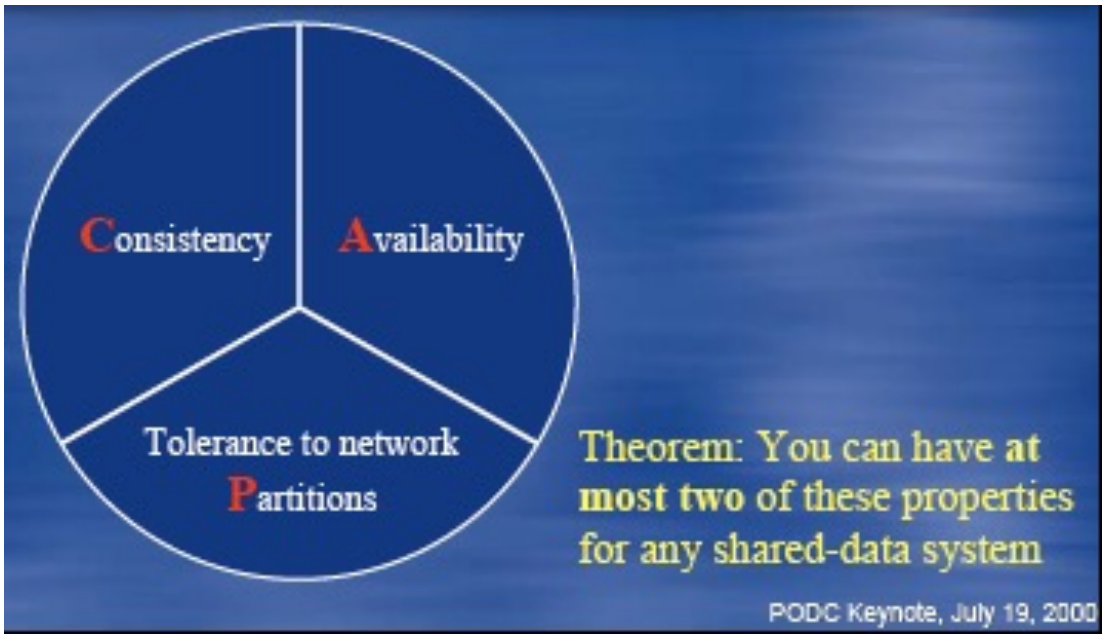
    "All clients can find some replica of data, even in presence of failure."
  - **Tolerance to network partitions** (***P***)

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

Brewer's talk at PODC 2000 http://bit.ly/2sVsYYv

E. Brewer, "CAP twelve years later: how the "rules" have changed", IEEE Comp., Feb. 2012.

CAP theorem



Why is partition-tolerance important?

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

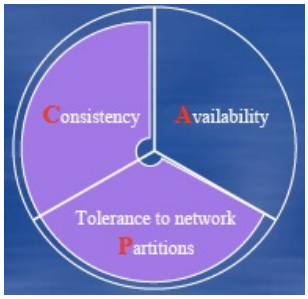
As result of partition, network can lose arbitrarily many messages sent from one node to another

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

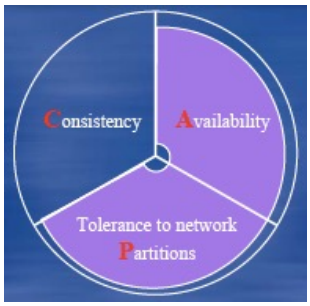
It's a **design choice**

CAP and network partitions

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



- If availability is priority, forfeit consistency: **AP** system
  - Use a relaxed consistency model: **eventual consistency**

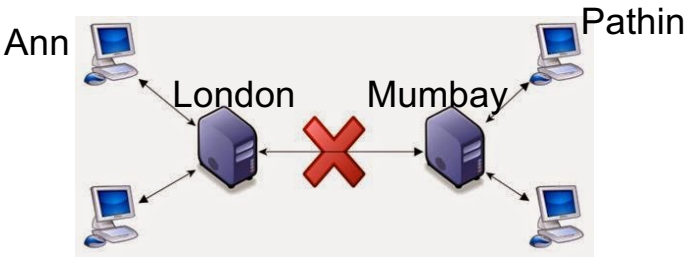


CAP and network partitions

- When using CP and AP systems, the developer needs to be aware of what system is offering
- CP system: may not be available to take a write
  - If write fails because of system unavailability, the developer has to decide what to do with the 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
  - The developer has to decide whether the client requires access to the absolute latest update all the time

CAP: example

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

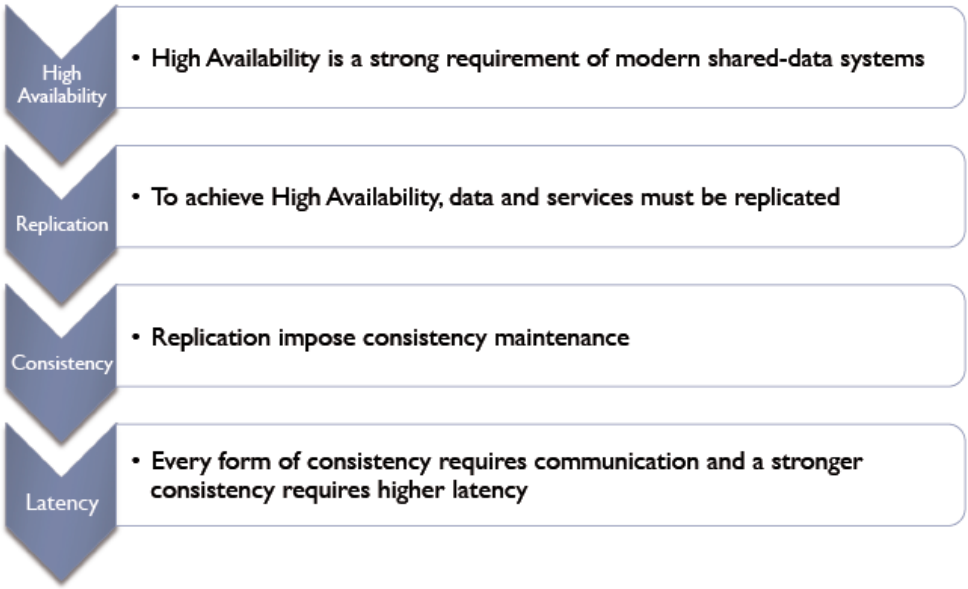


CAP: example

- CA system: neither user can book any hotel room
  - No tolerance to network partitions
- 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!
- Remember that AP choice depends on the application type
  - Blog different from financial exchange or shopping cart

Consistency/latency tradeoff

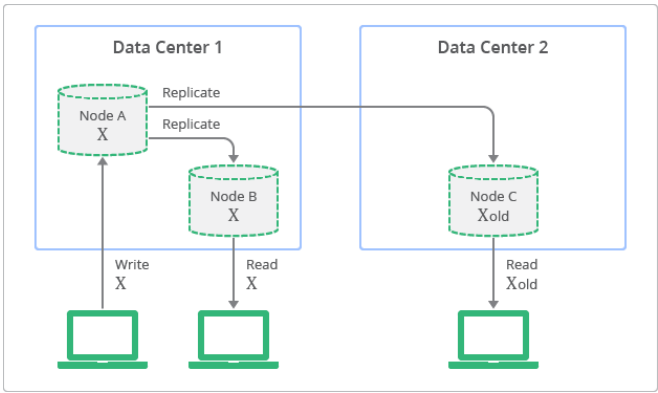
- CAP does not explicitly talk about latency, but latency is crucial to get the essence of CAP
  - Latency is not zero!



Consistenza finale

- In un archivio di dati distribuito caratterizzato da:
  - Mancanza di aggiornamenti simultanei (conflitti write-write) o comunque loro facile soluzione in caso di conflitto
  - Forte prevalenza di letture rispetto alle scritture (mostly read)
- si adotta spesso un modello di consistenza rilassato, detto consistenza finale (*eventual consistency*)
  - Cosa garantisce: ***se*** non si verificano aggiornamenti, tutte le repliche (distribuite geograficamente) diventano gradualmente consistenti **entro una finestra temporale** (detta *inconsistency window*)
  - In assenza di fallimenti, l'ampiezza dell'inconsistency window dipende da: latenza di comunicazione, numero di repliche, carico del sistema

Eventual consistency vs. strong consistency

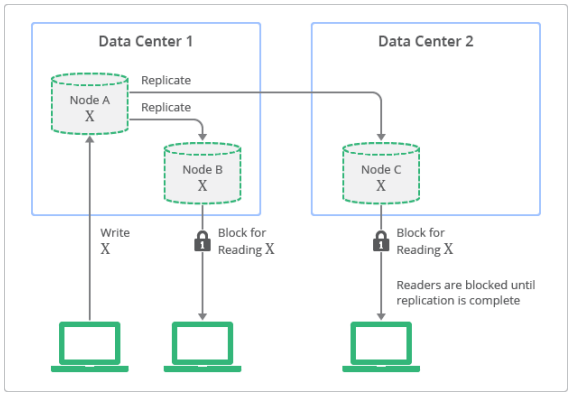


Eventual consistency

- 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 the update completes



Consistenza finale: vantaggi e svantaggi

- Vantaggi:
  - Modello di consistenza semplice e poco costoso da implementare
  - Letture e scritture veloci sulla replica locale
  - Ad es. usato nel DNS: il name server autoritativo aggiorna un dato resource record, altri name server lo memorizzano per la durata del TTL
- Svantaggi
  - No illusione di avere una singola copia
  - Possibile inconsistenza (*staleness*) dei dati causata da scritture conflittuali: occorre risolvere il conflitto tramite un algoritmo di *riconciliazione*

Eventual consistency: reconciliation

- Which strategy to decide *how* to reconcile conflicting versions of the same data that have diverged due to concurrent updates?
  - A widespread strategy is ***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)
  - An alternative is to push the complexity of conflict resolution to the application itself (e.g., Amazon Dynamo) which invokes a user-specified conflict handler
- *When* to reconcile?
  - Usually on read (e.g., Amazon Dynamo) so to provide an "always-writable" experience (but slows down read)
  - Alternatives are: on write (reconcile during write, slowing down it) and asynchronous repair (correction is not part of read or write op)

Consistenza finale e SD

- Modello di consistenza usato spesso nei sistemi distribuiti a larga scala
- Soprattutto per servizi Cloud di storage e data store NoSQL
  - Es.: Amazon Dynamo, AWS S3, CouchDB, Dropbox, git, iPhone sync
- Attenzione: il costo di garantire un modello di consistenza più forte ricade sullo sviluppatore dell'applicazione
  - Lo sviluppatore deve sapere quale grado di consistenza viene offerto dal sistema sottostante l'applicazione
  - Con la consistenza finale, può accadere che una read non restituisca il valore della write più recente: lo sviluppatore deve decidere se tale inconsistenza è accettabile per l'applicazione

V. Vogels, Eventually consistent, *ACM Communications*, 2009.

ACID vs BASE

- ACID and BASE: two design philosophies at opposite ends of the CA spectrum
- **ACID** (**A**tomicity, **C**onsistency, **I**solation, **D**urability)
  - Pessimistic approach: prevent conflicts from occurring
  - Traditional approach in relational DBMSs: Postgres, MySQL, … are examples of CA systems
  - But ACID does not scale well when handling petabytes of data (remember of latency!)

BASE

- **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*: the 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 in the hand of the developer that must take care of refreshing it
    - Data is durable if its changes survive failures and recoveries
  - *Eventually consistent*: the system eventually converges to a consistent state
- Soft state and eventual consistency work well in the presence of partitions and thus promote availability
- BASE is *often* adopted in NoSQL data stores

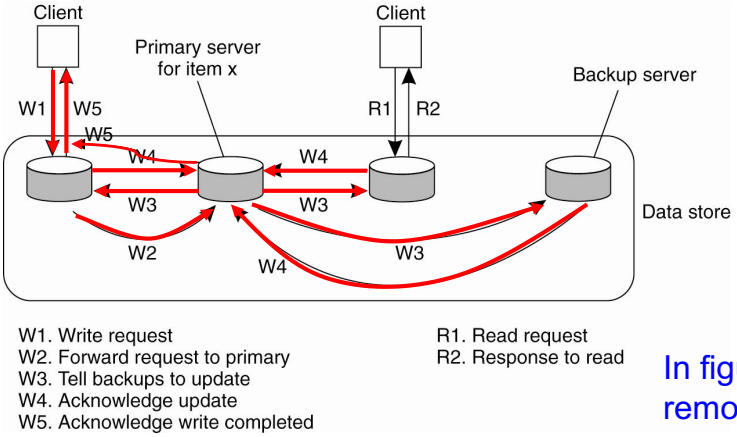
Protocolli di consistenza data-centrica

- 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. sulla 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

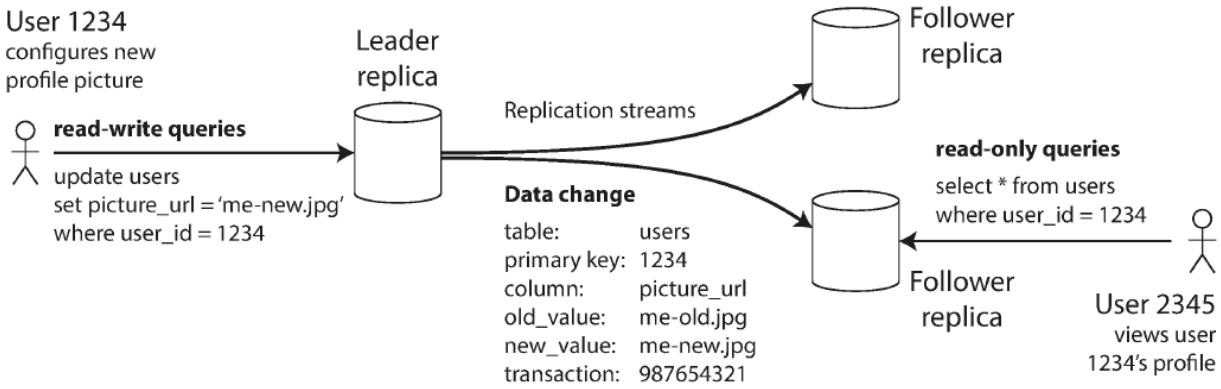
Primary-based: protocolli remote-write



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, RannitMQ) 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

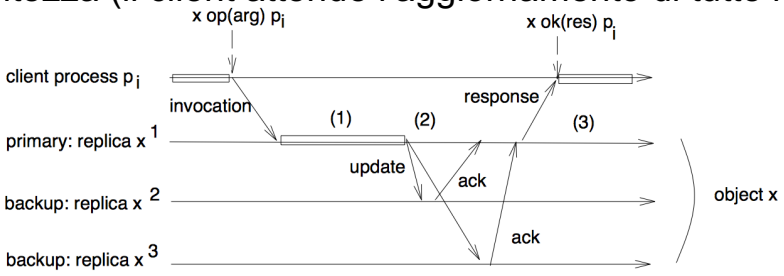
Primary-based remote-write: example



Primary-based: protocolli remote-write

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

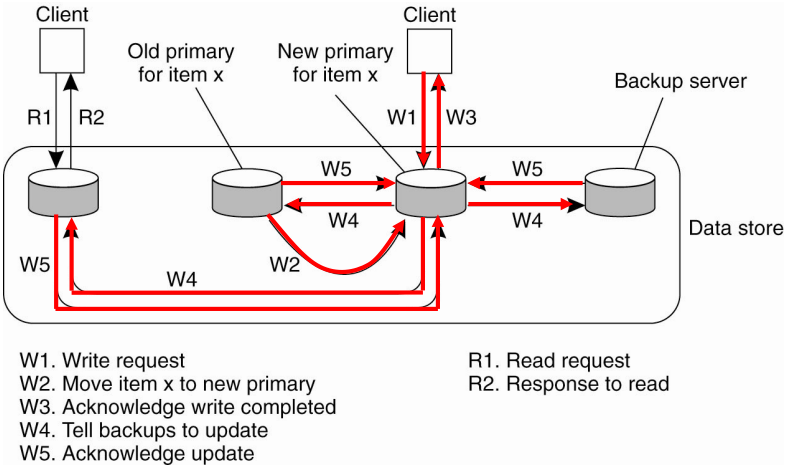
1. Bloccante (o replicazione sincrona):
   - La replica primaria notifica al client che la scrittura è stata completata su tutte le repliche
   - Modello di consistenza: linearizzabilità
   - Vantaggi: maggiore tolleranza a guasti (repliche sincronizzate), incluso crash della replica primaria
   - Svantaggi: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



Primary-based: protocolli remote-write

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

Primary-based: protocolli local-write



- Non bloccante rispetto al client
- Usato ad es. in ambito mobile computing
  - Invio dei dati rilevanti al mobile client prima della disconnessione e successivo aggiornamento alla riconnessione

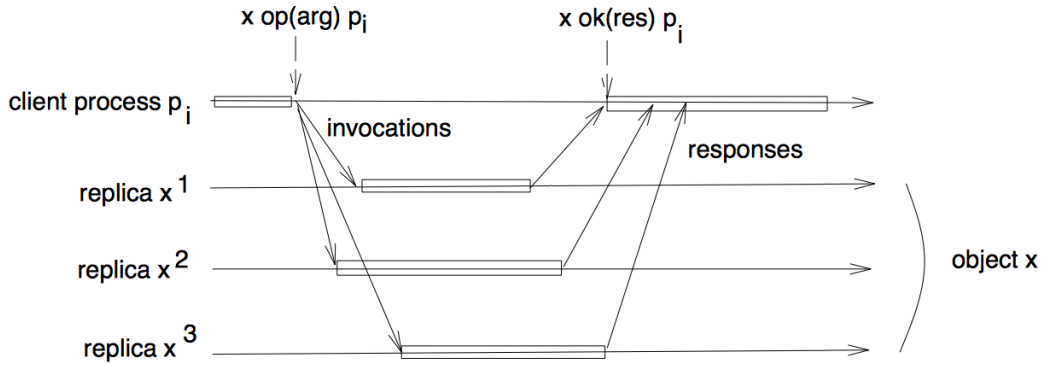
Protocolli replicated-write

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

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

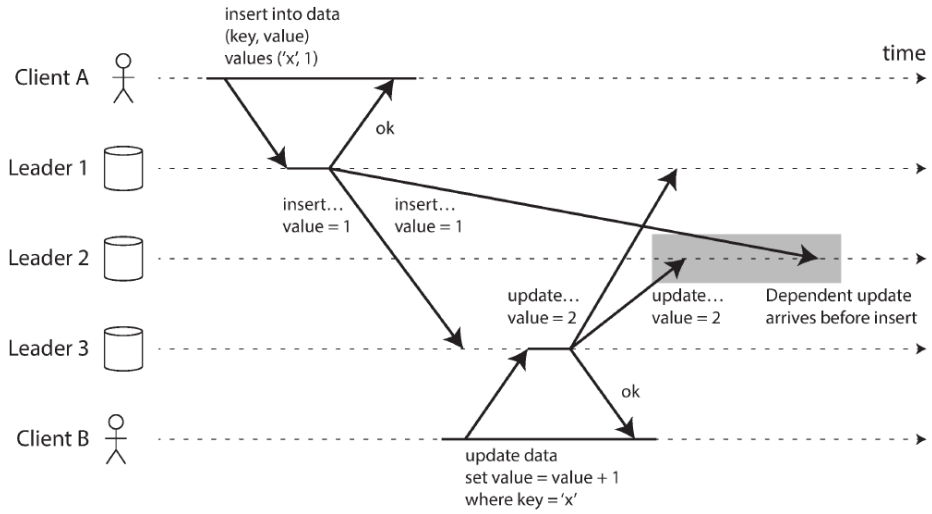
Replicated-write: replicazione attiva

- 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 su repliche diverse



Replicated-write: replicazione attiva

- Occorre eseguire le operazioni di scrittura nello stesso ordine su tutte le repliche
- Come? Multicasting totalmente ordinato
  - Decentralizzato usando clock scalare
    - Scalabilità limitata in sistemi a larga scala a causa di elevato numero di messaggi
  - Centralizzato tramite sequencer
    - Scarsa scalabilità e single point of failure
- Modello di consistenza data-centrica supportato
  - Consistenza sequenziale: stesso ordine su tutte le repliche, ma può essere diverso da quello temporale

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

D.K. Gifford, Weighted voting for replicated data, *Proc. ACM* SOSP 1979

Protocolli quorum-based: condizioni

- Per N_R e N_W valgono le condizioni

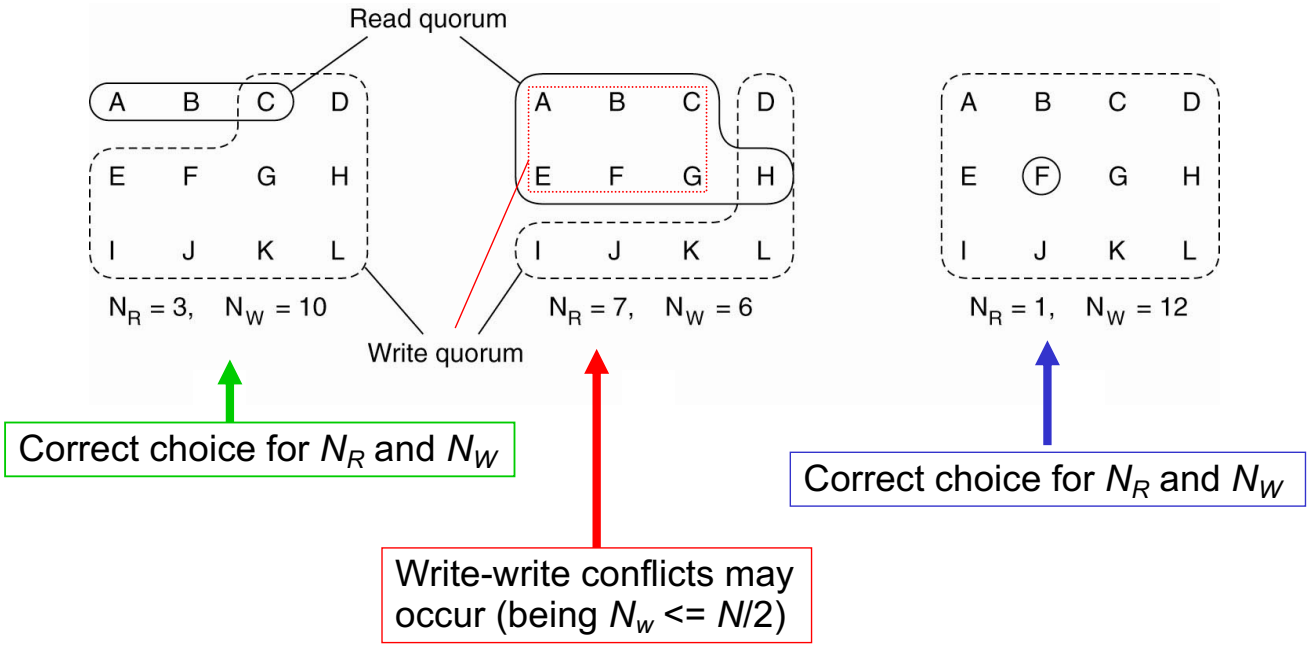
a) $N_R + N_W > N$

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

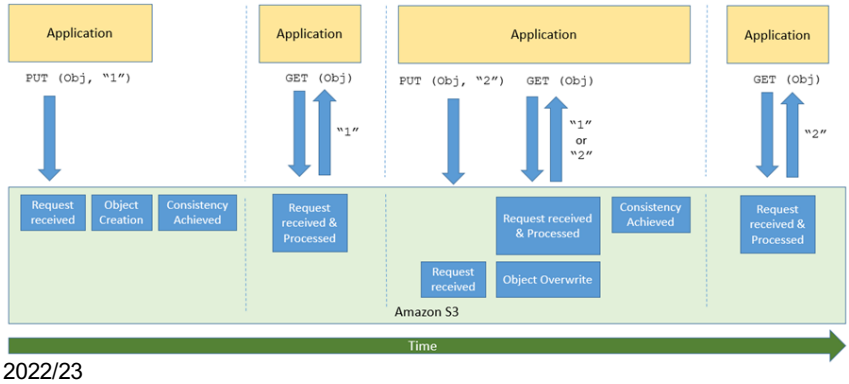


Setting read and write quorums

- 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 app developers to choose between strong and eventual consistency by selecting different read and write quorums (e.g., Cassandra)

Consistency of Cloud data store services

- Cloud storage services often favor weaker levels of consistency
  - Eventual consistency has been for a long time the most popular model for Cloud storage
  - New trend towards strong consistency, let's consider S3
- Amazon S3 (old, 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



Consistency of Cloud data store services

- Amazon S3 consistency model (now)

  "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."

Tunable consistency

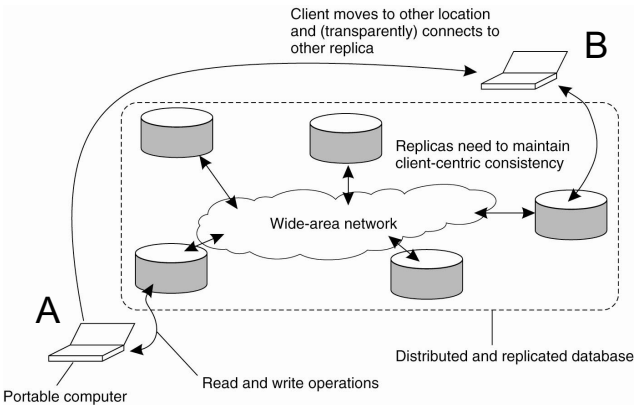
- Some NoSQL data stores offer **tunable** consistency: user can tradeoff between consistency and latency
- Amazon's DynamoDB: both eventually consistent reads and strongly consistent reads
  - Strongly consistent reads experience higher read latency, twofold reduction in read throughput and cost more http://amzn.to/38HoGIn
- Similarly for Google's Cloud Datastore http://bit.ly/2LmBniX
- Cassandra provides quorum-based consistency, where quorums may be tuned
  - N+W > R and W >= N/2 +1: strong consistency but increased latency https://bit.ly/2n26EdE

Consistenza client-centrica

- *Obiettivo*: fornire garanzie ad un singolo client relative alla consistenza degli accessi da parte di quel client ad un archivio di dati distribuito
  - Nessuna garanzia di consistenza relativamente agli accessi concorrenti da parte di altri client

Esempio

- Consideriamo un archivio di dati distribuito a cui un utente accede tramite un dispositivo mobile
  - Nella posizione A l'utente accede alla replica locale dell'archivio, eseguendo letture e scritture
  - Nella posizione B l'utente, a meno che non acceda allo stessa replica della posizione A, può notare inconsistenze nei dati
    - Le scritture in A possono non essere state ancora propagate a B
    - L'utente sta leggendo entry più nuove di quelle disponibili in A
    - Le scritture in B possono essere in conflitto con quelle in A
- L'utente desidera che i dati scritti e/o letti precedentemente in A siano in B nello stesso modo in cui li ha lasciati in A: l'archivio di dati apparirà così consistente all'utente



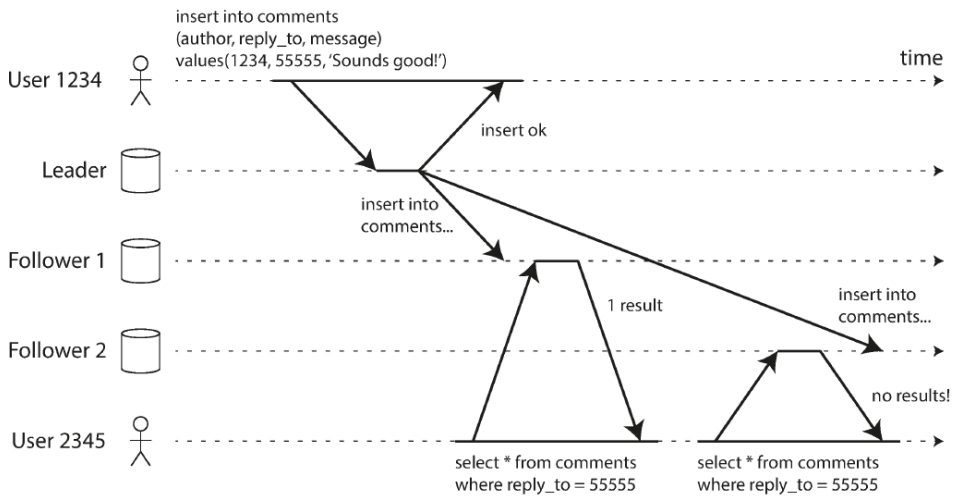
Modelli di consistenza client-centrici

- Monotonic-read (o read-after-read)
  - Le letture non possono tornare indietro per quel client
- Monotonic-write (o write-after-write)
  - Le scritture di quel client non possono tornare indietro
- Read-your-writes (o read-after-write)
  - Equivalente alla consistenza causale per il singolo client
- Writes-follow-reads (o write-after-read)

Client-centric consistency: example

- A user first reads from a fresh replica, then from a stale replica
- Which kind of client-centric consistency is violated?

  Monotonic reads (a.k.a. read-after-read)



Client-centric consistency: example

- A user makes a write, followed by a read from a stale replica
- Which kind of client-centric consistency is violated?

  Read-your-writes (a.k.a. read-after-write)