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. 2023/24

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

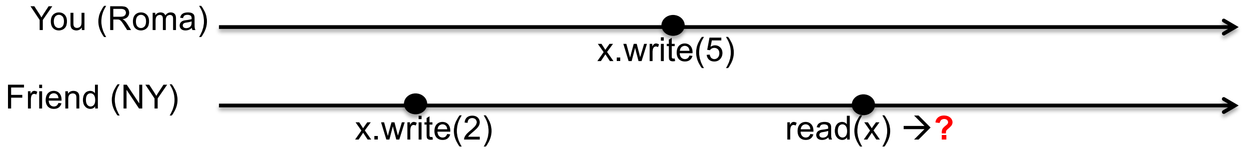
Pros of replication

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, 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



You (Roma)

x.write(5)

Friend (NY)

x.write(2)

read(x) → ?

DC in North Carolina

DC in Ireland

Consistency issues

- 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?
  - 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*

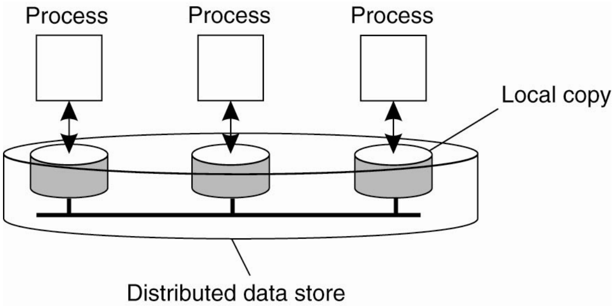
Consistency: what we need

- 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 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

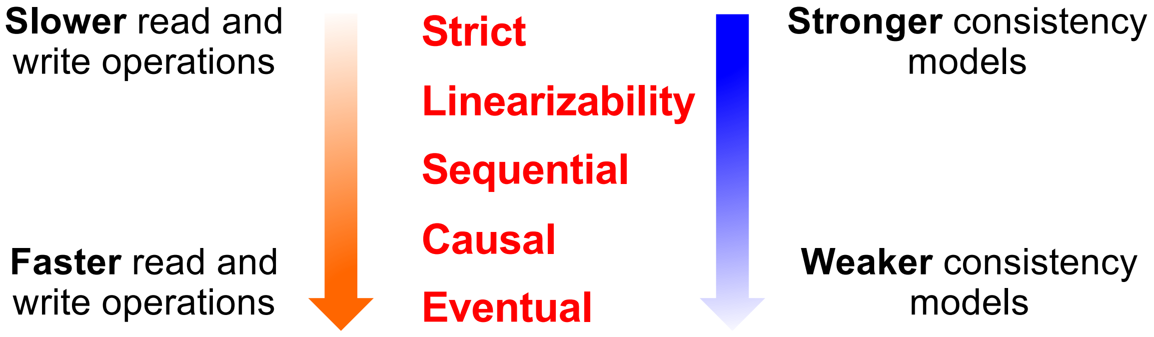
Data-centric consistency models

- 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

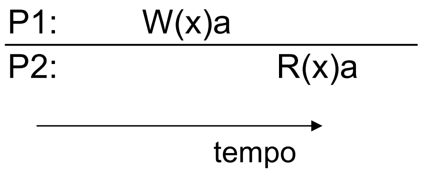
- Main consistency models based on ordering of read and write operations on shared and replicated data



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

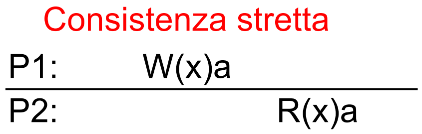
Modelli di consistenza: notazione

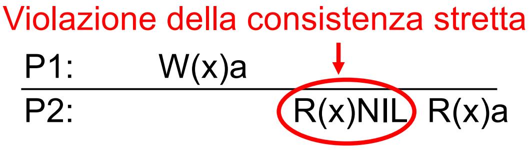
- Rappresentiamo il comportamento dei processi che eseguono operazioni di lettura o scrittura sui dati condivisi
  - $W_i(x)a$: operazione di scrittura da parte del processo P_i sul dato x con valore scritto a
  - $R_i(x)b$: operazione di lettura da parte del processo P_i sul dato x con valore letto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

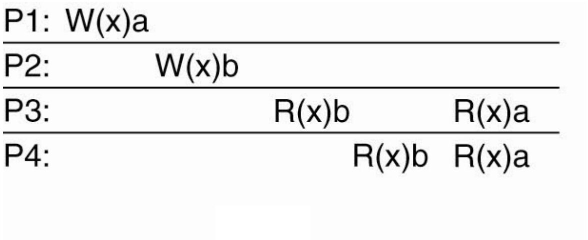
Il risultato di una qualunque esecuzione è uguale a quello ottenuto se le operazioni (di read e write) da parte di tutti i processi sull'archivio di dati fossero eseguite

- *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



P1: W(x)a
P2: W(x)b
P3: R(x)b R(x)a
P4: R(x)b R(x)a

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

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

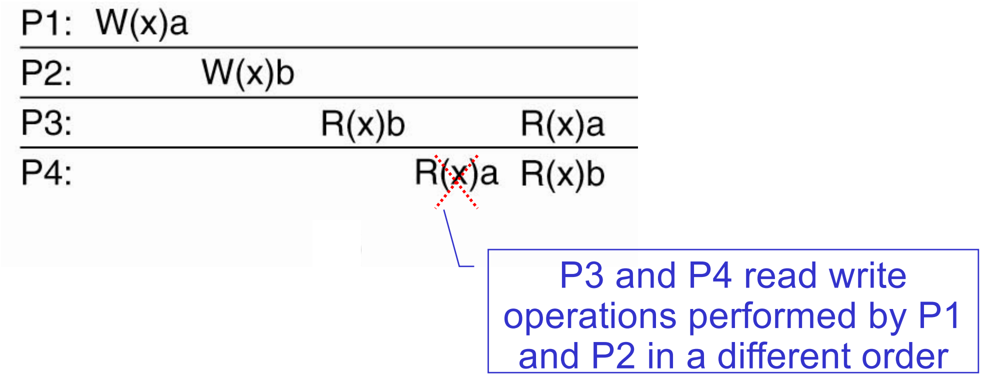
$W_2(x)b$ $R_4(x)b$ $R_3(x)b$ $W_1(x)a$ $R_4(x)a$ $R_3(x)a$

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

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

Sequential consistency: example

- A data store that is not sequentially consistent



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 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

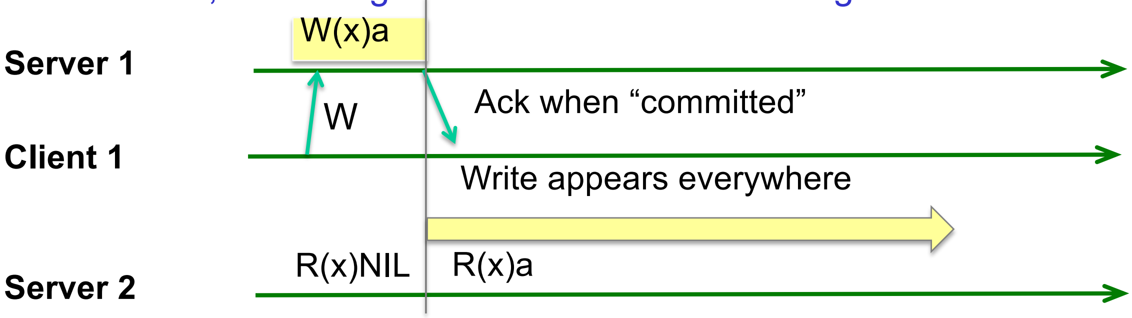
Each operation should appear to take effect instantaneously at some point between its start and completion

- 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



W(x)a

Server 1

W

Ack when “committed”

Client 1

Write appears everywhere

R(x)NIL

R(x)a

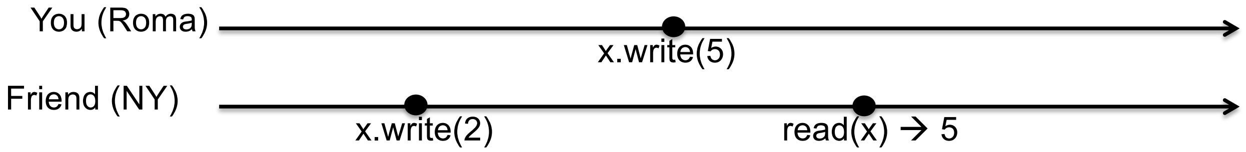
Server 2

- 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

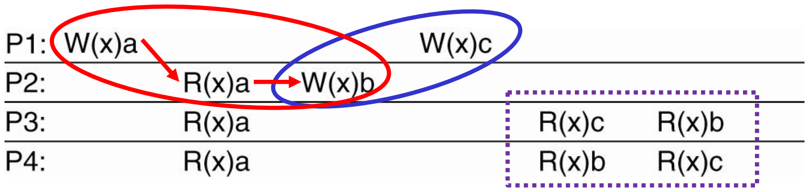
- In relazione di causa/effetto:
    - read seguita da write sullo stesso processo: write è (potenzialmente) causalmente correlata con read
    - write di un dato seguita da read dello stesso dato su processi diversi: read è (potenzialmente) causalmente correlata con write
    - Si applica la proprietà 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

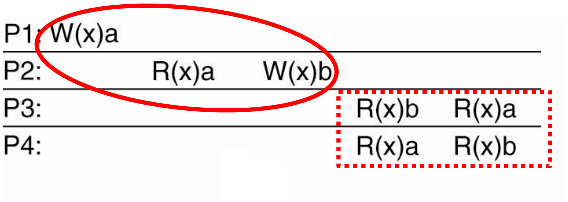


P1:	W(x)a			W(x)c		
P2:		R(x)a	W(x)b			
P3:		R(x)a			R(x)c	R(x)b
P4:		R(x)a			R(x)b	R(x)c

No consistenza sequenziale

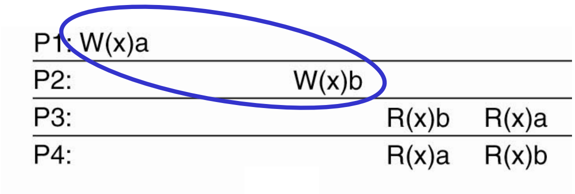
Causal consistency: examples

- 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

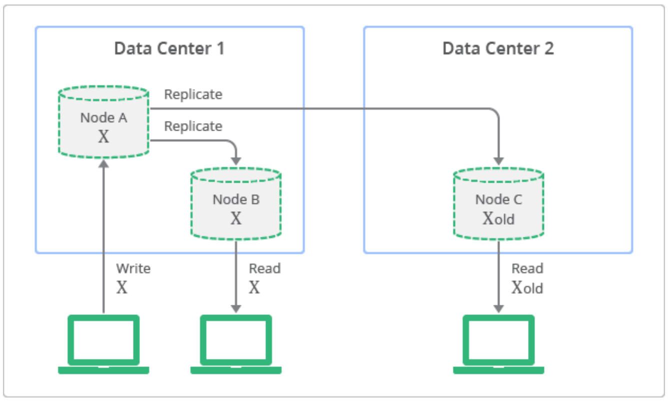
- Let's just do best effort to make things consistent: **eventual consistency**
  - Popularized by CAP theorem

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 può adottare 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

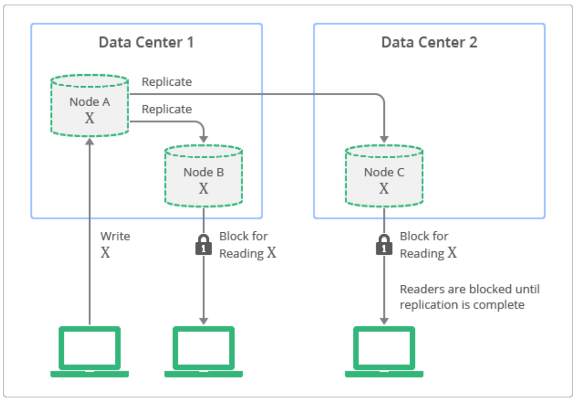


Eventual consistency

- 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*

Consistenza finale: vantaggi e svantaggi

- Svantaggi
  - 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

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)

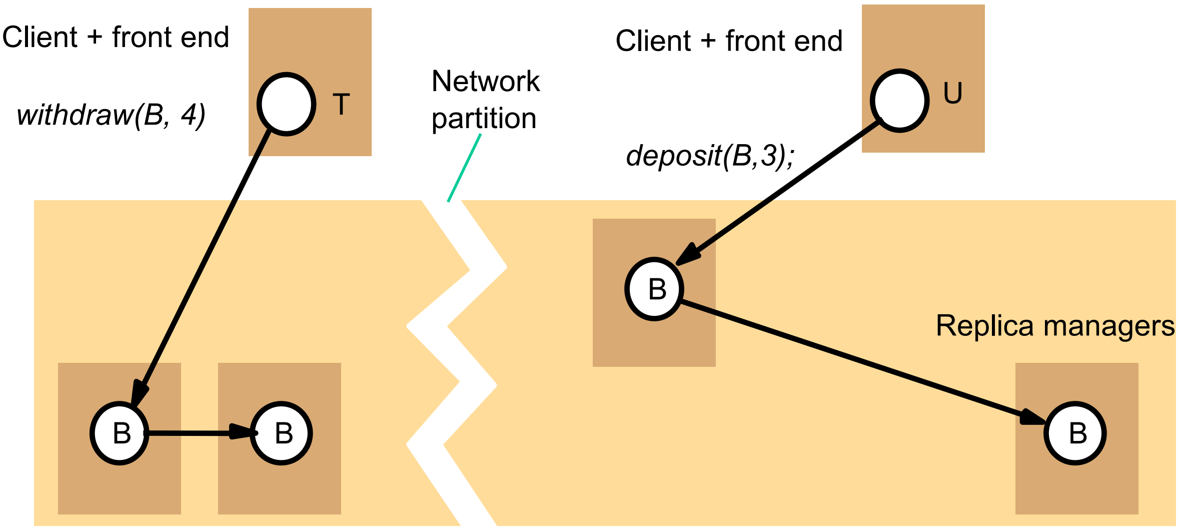
Consistenza finale e SD

- Modello di consistenza frequentemente adottato in sistemi distribuiti a larga scala per servizi di storage e data store NoSQL
  - Es.: Amazon Dynamo, AWS S3, CouchDB, Dropbox, git, iPhone sync

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

Consistency and network partitions

- Main problem is network partitions



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

- 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)

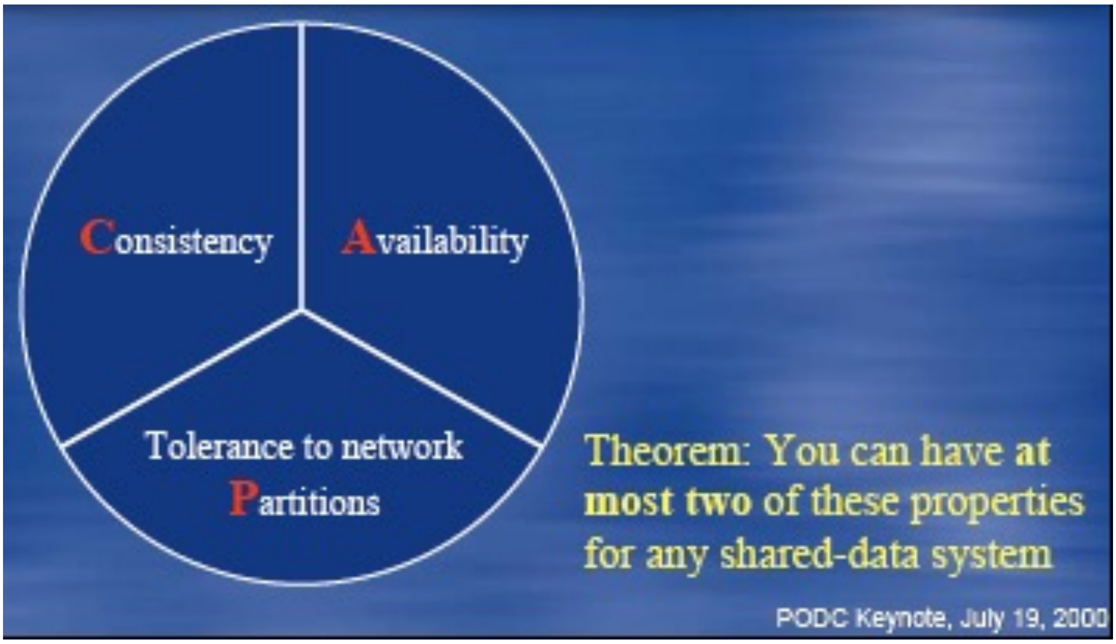
    "All clients can find some replica of data, even in presence of failure."
  - **Tolerance to network partitions** (***P***)

    "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



Theorem: You can have at most two of these properties for any shared-data system

PODC Keynote, July 19, 2000

Why is partition-tolerance important?

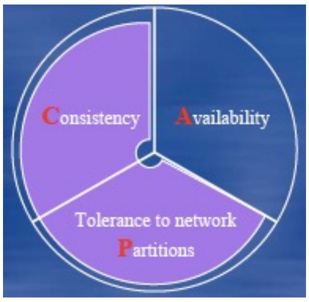
- Network partitions can occur across data centers when Internet gets disconnected
  - Internet router outages
  - Under-sea cables cut
  - DNS not working

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

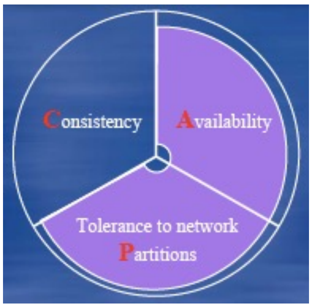
- Network partitions can also occur within a datacenter (e.g., rack switch outage), but less frequently
- Still desire distributed system to continue functioning normally under network partitions → fix **P**
- Therefore, consistency and availability cannot be achieved at the same time when partition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

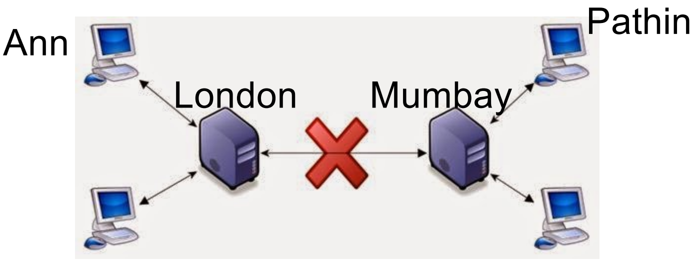
CAP and network partitions

- 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 CAP choice depends on application requirements
  - Blog different from financial exchange or shopping cart

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