POLITECNICO
MILANO 1863

Consistency and Replication

Alessandro Margara
alessandro.margara@polimi.it
http://home.deib.polimi.it/margara

Replication: why?

- Improve performance
  - Sharing of workload …
    average number of user you can serve in a unit of time
  - … to increase the *throughput* of served requests and reduce the *latency* for individual requests
    - For example, replicate a Web server to sustain a higher number of users
  - Replicate data close to the users …
    geographic region/data
  - … to reduce the *latency* for individual requests
    - For example, cache in processors, local cache in browsers, geo-replicated services, …

Don't forget the speed of light …

	OR	VA	TO	IR	SY	SP	SI
CA	22.5	84.5	143.7	169.8	179.1	185.9	186.9
OR		82.9	135.1	170.6	200.6	207.8	234.4
VA			202.4	107.9	265.6	163.4	253.5
TO				278.3	144.2	301.4	90.6
IR					346.2	239.8	234.1
SY						333.6	243.1
SP							362.8

(c) Cross-region (CA: California, OR: Oregon, VA: Virginia, TO: Tokyo, IR: Ireland, SY: Sydney, SP: São Paulo, SI: Singapore)

Table 1: Mean RTT times on EC2 (min and max highlighted)

Bailis et al. "Highly-available transactions, virtues and limitations" VLDB '13.

Replication: why?

[ˌɪntəˈmɪtəntli]
adv. 间歇地

- Increase availability
  - Data may be available only intermittently in a mobile setting
    - A local replica in the mobile node can provide support for disconnected operations
  - Data might become unavailable due to excessive load
    - For example, release of a new operating system
- Achieve fault tolerance
  - Related to availability
  - Data may become unavailable due to failure
    - Availability = $1 - p^n$
    - p probability of failure, n # of replicas
  - It becomes possible to deal with incorrect behaviors through redundancy

Examples

- Domain Name Service (DNS) why it can be scalable
- Content delivery networks (CDN) netflix
  - Geographically distributed network that replicate the content to better serve end users
- Distributed file systems multiple machine share the same file system
  - File replication allows for faster access and disconnected operations
    - User files are reconciled against servers upon reconnection
    - Assumption: conflicts (files modified by more than a user) are infrequent
      for Dropbox may be not this case
      用户文件在重新连接时与服务器进行协调
- Platforms for Big Data
  - Rely on distributed file systems
  - Performance through local access
    - Don't move the data, move the computation!
  - Fault tolerance through replication

Challenges

- Main problem: *consistency* across replicas
  - Changing a replica demands changes to all the others
  - What happens if multiple replicas are updated concurrently?
    - Write-write conflicts / read-write conflicts
    - What is the behavior in the case of conflicts?

cost of consistency and the level of consistency

- Goal: provide consistency with limited communication overhead

Challenges

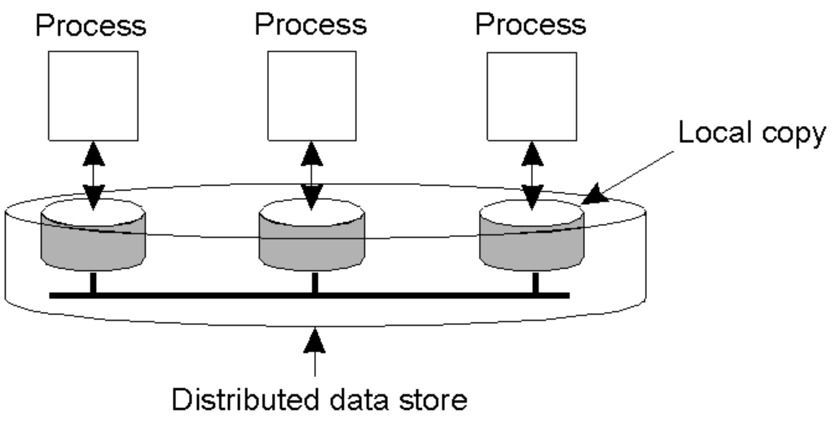
- Scalability vs. performance
  - Replication may actually degrade performance!

- Different consistency requirements depending on the application scenario
  - Data-centric vs. client-centric

Consistency models

- Focus on a (distributed) data store
  - Shared memory, filesystem, database, …
  - The store consists of multiple items
    - Files, variables, …
- Ideally, a read should show the result of the last write
  - What does *last* mean?
  - Impossible to determine without a global clock

each process work by reading/writing on the local copy and there are protocols to make the changes consistent



Consistency models

- A consistency model is a contract between the processes and the data store
  - Stricter guarantees simplify the development but incur higher costs
  - Weaker guarantees reduce the cost but make development difficult sometimes read data that is not the latest one
  - Tradeoffs: guarantees, performance, ease of use

Consistency models

- Several different models
  - For individual operations or groups of operations
  - Guarantees on content
    - Maximum "difference" on the versions stored at different replicas

      but there are some guarantees such that the value never go below to 0
  - Guarantees on staleness
    - Maximum time between a change and its propagation to all replicas

      maybe lose some work, but we never lose more than one day of work
  - Guarantees on the order of updates
    - Constrain the possible behaviors in the case of conflicts
    - Data-centric vs client-centric

      order of

      order of ... in clients

Consistency protocols

- Consistency protocols implement consistency models
- The devil is in the details!
- Different strategies for different assumptions/configurations
  - Passive vs active
    for passive, just store but cannot serve us
  - Single leader vs multiple leader vs leaderless
    dropbox(multiple nodes write concurrently)
  - Synchronous vs asynchronous
    means you can lose the data
  - "Sticky" clients vs mobile clients

    if the data can move (can connect to FB
    through multiple data centers)

Consistency protocols

- Passive replication
  - All the operations go through a master
  - The master propagates all the changes to one or more backup replicas
  - If the master fails, one of the replicas take over
- Provides fault–tolerance related to availability
  - If the propagation of changes occurs synchronously
- No sharing of workload!
  - For this reason, many solutions adopt active replication
  - Replicas can also process user requests
  - We will focus on active replication in these slides

Consistency protocols

- Single leader
  - One of the replicas is designated as the leader
    - When clients want to write to the datastore, they must send the request to the leader, which first writes the new data to its local storage
  - The other replicas are known as followers
    - Whenever the leader writes new data to its local storage, it also sends the data to all of its followers
  - When a client wants to read from the database, it can query any replica (either the leader or a follower)

Consistency protocols

- Synchronous
  - The write operation completes after the leader has received a reply from all the followers
- Asynchronous
  - The write operation completes when the new value is stored on the leader
  - Followers are updated asynchronously
- Semi-synchronous
  - The write operation completes when the leader has received a reply from at least *k* replicas

Consistency protocols

- Synchronous (or semi-synchronous with *k* followers) replication is safer
  - Even if k-1 replicas fail, we still have a copy of the data

- What happens if the leader fails?
  - We can elect a new leader
  - But the protocol can be very complex
    - To deal with situations in which the old leader comes back alive and other tricky situations …
    - Out of the scope of this lecture

Consistency protocols

- Multiple leaders
  - Writes are carried out at different replicas concurrently
  - No leader means that there is no single entity that decides the order of writes
  - It is possible to have write-write conflicts in which two clients update the same value almost concurrently
    - How to solve conflicts depends on the specific consistency model
    - We will discuss several of them later

Consistency protocols

- Leaderless replication
  - The client directly contacts several replicas to perform the writes/reads
  - Quorum-based protocols
    - Similar to a voting system
    - We need a majority of replicas to agree on the write
    - We need an agreement on the value to read

quorum 英 [ˈkwɔːrəm]
会议的）法定人数

Consistency protocols

- Considering clients mobility can introduce further issues

- In some configurations, a client can connect to a new replica and might not find previous writes it previously performed on another replica

DATA-CENTRIC CONSISTENCY MODELS

Data-centric consistency models

- We now review the most widely used data-centric consistency models
- Graphical convention
  - One line for each process
  - Operations of each process appear in temporal order

    in time order
  - W(x)a means that the value a is written on the data item x
  - R(x)a means that the value a is read from the data item x

Strict consistency

"Any read on data item x returns the value of the most recent write on x"

we don't have a single clock
there is no way to synchronize the operations

this one is never used

Strict consistency

single processor machine

- All writes are instantaneously visible, global order is maintained
- "Most recent" is ambiguous without global time
- In practice, possible only within a uniprocessor machine

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

Consistent

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

NOT Consistent

why this is not consistent

Sequential consistency

“The result is the same as if the operations by all processes were executed in some sequential order, and the operations by each process appear in this sequence in the order specified by its program”

Sequential consistency

- Operations within a process may not be re-ordered
- All processes see the same interleaving
- Does not rely on time

we can just write the sequence

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

Consistent

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

NOT Consistent

for all processes, they are performed on the same order

Sequential consistency

- Why sequential consistency?

- Shared memory on multi-processor computers

- "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs"
  - L. Lamport, ACM Transactions on Programming Languages and Systems, 1979

- C++, Java memory models
  - Are they sequential?

Sequential consistency (Java)

Thread 1

multi-core
multi cache

Thread 2

int x = 0;
int y = 0;
…
…
…
…
x = 1;
y = 1;

cache coherency protocols
are for single computers

…
…
…
…
…
…
…
…
…
read y = 1;
read x: which values are allowed?

0 is allowed!

Java is not even sequencial consistent
volatile!
Because it's too expensive. If we propogate
the updates to all the caches

Sequential consistency (Java)

- Values can be read from the local cache …
- … and any old value is allowed

- Unless
  - The variable is declared as volatile, or
  - There is a synchronized block
    - Synchronized blocks are sequentially consistent

- In practice, synchronization is almost always controlled through synchronized blocks
  - Cases like the example in the previous slide should never occur

Sequential consistency

- Consider the following program

Process P1	Process P2	Process P3
x = 1;	y = 1;	z = 1;
print (y, z);	print (x, z);	print (x, y);

- All the following (and many other) executions are sequentially consistent not possible to reverse the sequences in one processor

x = 1;	x = 1;	y = 1;	y = 1;
print (y, z);	y = 1;	z = 1;	x = 1;
y = 1;	print (x,z);	print (x, y);	z = 1;
print (x, z);	print (y, z);	print (x, z);	print (x, z);
z = 1;	z = 1;	x = 1;	print (y, z);
print (x, y);	print (x, y);	print (y, z);	print (x, y);
Prints: 001011	Prints: 101011	Prints: 010111	Prints: 111111

Sequential consistency: implementation

- All the replicas need to agree on a given order of operations

- Solutions
  - Distributed agreement (see previous lectures)
  - Using a single coordinator
    - Single leader replication
    - In practice, one of the reference implementations for sequential consistency
      - MySQL, PostgreSQL, MongoDB, …

- Assumption
  - Sticky clients not move one replica to another

Sequential consistency: implementation

- Sequential consistency limits availability
  - We need to contact the leader
    - Which might be further away from the client
  - The leader must propagate the update to the replicas
    - In a synchronous way if we want to be fault-tolerant

- Two main problems related to availability
  - High latency
  - Clients are blocked in the case of network partitions
    - Can only proceed if they can contact the leader
    - The leader can only proceed if it can contact the followers

consistency availability partition

Leaderless protocols

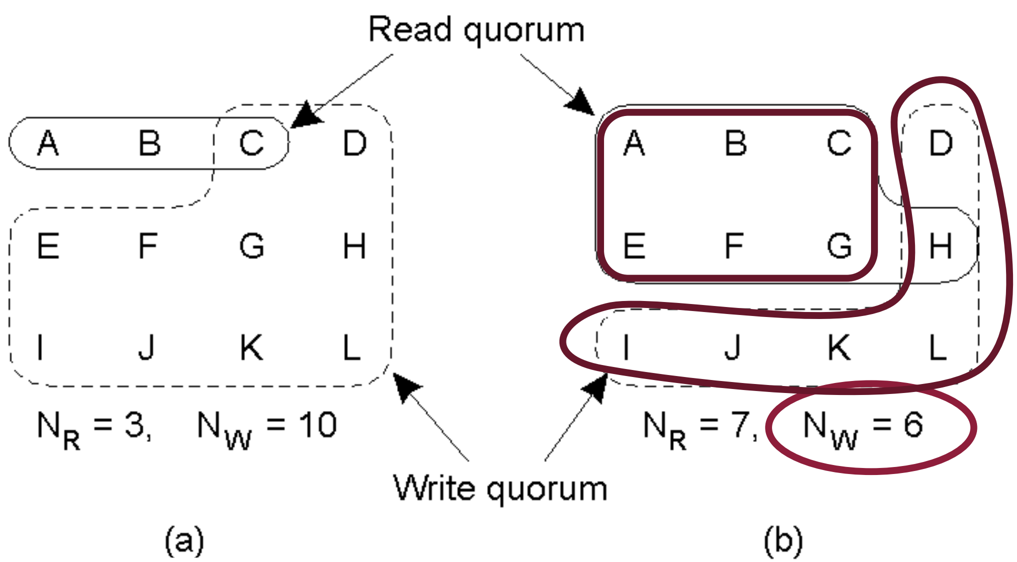
- Quorum-based
  - An update occurs only if a quorum of the servers agrees on the version number to be assigned
  - Reading requires a quorum to ensure latest version is being read

    number of replica to read NR
    number of replica to write NW
  - Typically:
    - NR + NW > N
    - NW > N/2

Avoids read-write conflicts

Avoids write-write conflicts

Leaderless protocols



A correct choice of read and write set

there are some overlap, so there are some nodes contains the latest nodes

A choice that may lead to write-write conflicts

maybe two processes write on separately 7 nodes

A correct choice, known as ROWA (read one, write all)

when you are the client, write to all of the other

Linearizability

"The system is sequentially consistent; moreover, if $ts_{OP1}(x) < ts_{OP2}(y)$ then operation OP1(x) precedes OP2(y) in the operation sequence."

Linearizability

- Stronger than sequential, weaker than strict
  - It assumes globally available clocks but only with finite precisions
  - Useful if the application logic needs to enforce some ordering between operations

- The previous example of sequential is also linearizable if we assume the write at P1 has a timestamp greater than the write at P2

Causal consistency

- Consider a group chat discussion
  - A says "Distributed systems are the best!"
  - B says "No way! They are too complex …"
- If C first sees B and then A, she cannot understand what is going on
  - Therefore, everybody else must see A's message before B's one

Causal consistency

- Consider other two messages
  - A says "Apple released a new MacBook"
  - B says "I'm having a lot of fun learning about consistency and replication!"
- The two messages are not related to each other
  - The order in which they are seen from other members does not matter

Causal consistency

"Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in any order at different machines."

Causal consistency

- Weakens sequential consistency based on Lamport's notion of happened-before
  - Lamport's model deals with message passing
  - Here causality is between reads and writes

Causally related writes

Concurrent writes

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

Consistent

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

NOT Consistent
(it becomes consistent without P2's R(x))

Causal consistency

- Causal consistency defines a causal order among operations
- More precisely, causal order is defined as follows:
  - A write operation W by a process P is causally ordered after every previous operation O by the same process
    - Even if W and O are performed on different variables
  - A read operation by a process P on a variable x is causally ordered after a previous write by P on the same variable
  - Causal order is transitive

?

- It is not a total order
  - Operations that are not causally ordered are said to be concurrent

Causal consistency

- Why causal consistency?

- Easier to guarantee within a distributed environment
  - Smaller overhead

- Easier to implement

- "Causal memory meets the consistency and performance needs of distributed applications!"
  - Ahamad et al., 1994

Causal consistency: implementation

- Multi-leader implementations are possible (which enable concurrent updates)
  - Writes are timestamped with Lamport's vector clocks
  - Vector clocks define what the process knew when it performed the write
    - Which are the possible causes of the write
  - An update U is applied to a replica only when all the write operations that are possible causes of U have been received and applied
    - Otherwise a read always returns the previous value

Causal consistency: implementation

- The above implementation enables a high degree of availability
  - Clients can continue to interact with the store even if they are disconnected from other replicas
  - The local replica will return an old value …
  - … but it avoids violation of causality
  - New writes can also be performed
    - The rest of the world will not be informed
    - The writes that occur in the rest of the world will be concurrent
    - This is clearly not possible under sequential consistency!

- Note: this implementation works only if clients are sticky!

FIFO consistency

"Writes done by a single process are seen by all others in the order in which they were issued; writes from different processes may be seen in any order at different machines."

FIFO consistency

- In other words, causality across processes is dropped
  因果关系
- Also called PRAM consistency (Pipelined RAM)
  - If writes are put onto a pipeline for completion, a process can fill the pipeline with writes, not waiting for early ones to complete

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

Consistent

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

Not Consistent

FIFO consistency: implementation

- Very easy to implement
  - Even with multi-leader solutions (concurrent updates)

- The updates from a process P carry a sequence number
  - A replica performs an update U from P with sequence number S only after receiving all the updates from P with sequence number lower than S

Consistency models and synchronization

- FIFO consistency still requires all writes to be visible to all processes, even those that do not care
- Moreover, not all writes need be seen by all processes
  - E.g., those within a transaction/critical section

Consistency models and synchronization

- Some consistency models introduce the notion of synchronization variables
  - Writes become visible only when processes explicitly request so through the variable
  - Appropriate constructs are provided (e.g., synchronize(S))

- It is up to the programmer to force consistency when it is really needed, typically
  - At the end of a critical section, to distribute writes
  - At the beginning of a "reading session" when writes need to become visible

Weak consistency

1. *Access to synchronization variables is sequentially consistent;*

2. *No operation on a synchronization variable is allowed until all previous writes have completed everywhere;*

3. *No read or write to data are allowed until all previous operations to synchronization variables have completed.*

Weak consistency

- It enforces consistency on a group of operations
- It limits only the time when consistency holds, rather than the form of consistency
- Data may be inconsistent in the meanwhile

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

Consistent

P1:	W(x)a	W(x)b	S		
P2:				S	R(x)a

NOT Consistent

Release consistency

- Problem: the data store cannot distinguish between a synchronization request for disseminating writes or for reading consistent data
  - Unnecessary overhead

- Solution: introduce different synchronization operations
  - *Acquire* indicates critical region is about to be entered
  - *Release* indicates critical region has just been exited
  - In the standard release consistency definition, acquire and release refer to the entire data store

Release consistency

1. *Before a read or write is performed, all previous acquires done by the process must have completed successfully;*

2. *Before a release is allowed, all previous reads and writes done by the process must have been completed;*

3. *Accesses to synchronization variables are FIFO consistent.*

Release consistency

P1:	Acq(L)	W(x)a	W(x)b	Rel(L)				
P2:					Acq(L)	R(x)b	Rel(L)	
P3:								R(x)a

- P2's acquire must wait for P1's release
- On release, protected data that changed are propagated to other copies
- Two ways to achieve this
  - Eager release consistency
    - On release, all updates are pushed to other replicas
    - Potentially sends data to processes that will not use it
  - Lazy release consistency
    - On release, nothing is sent
    - On acquire, acquiring process must get latest version of the data from other processes
    - Tends to be more bandwidth efficient

Entry consistency

- Explicitly associates each shared data item with a synchronization variable
  - Still accessed through acquire and release
  - Reduces update overhead, increases complexity of access
  - Improves parallelism, enabling simultaneous access to multiple critical sections

- Two modes of access to a synchronized variable
  - Non-exclusive: multiple processes can hold read locks simultaneously
  - Exclusive: only one process holds the lock to the variable; to be granted, must guarantee that no other process holds (even a non-exclusive) lock

Entry Consistency

1. *An acquire access of a synchronization variable is not allowed to perform wrt a process until all updates to the guarded shared data have been performed wrt that process;*

2. *Before an exclusive mode access to a synchronization variable by a process is allowed to perform wrt that process, no other process may hold the synchronization variable, not even in non-exclusive mode;*

3. *After an exclusive mode access to a synchronization variable has been performed, any other process' next non-exclusive mode access to that synchronization variable may not be performed until it has performed wrt to that variable's owner.*

Entry consistency

P1:	Acq(Lx)	W(x)a	Acq(Ly)	W(y)b	Rel(Lx)	Rel(Ly)		
P2:						Acq(Lx)	R(x)a	R(y)NIL
P3:							Acq(Ly)	R(y)b

- On acquire, all guarded data must be made visible
- For exclusive mode to be granted, no other process can be holding any kind of lock
- After a data item has been accessed in exclusive mode, all future accesses by processes (other than the one that did the write) must go through the acquire process
- Complex to use!
  - But may be useful if encapsulated in a distributed object model

Summary of consistency models

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order

Consistency	Description
Weak	Shared data can be counted on to be consistent only after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered

Eventual consistency

- The models considered so far are data-centric
  - Provide a system-wide consistent data view in the face of simultaneous, concurrent updates
- However, there are situations where there are
  - No simultaneous updates (or can be easily resolved)
  - Mostly reads
- Examples: Web caches, DNS, Facebook (geo-distributed data stores)

Eventual consistency

- In these systems, eventual consistency is often sufficient
  - Updates are guaranteed to eventually propagate to all replicas

- Very popular today for three reasons
  - Very easy to implement
  - Very few conflicts in practice
    - E.g., in Facebook a user often accesses and updates the same replica
    - Today's networks offer fast propagation of updates
  - Dedicated data-types (conflict-free replicated data-types)

Eventual consistency

- Conflict-free replicated data types (CRDTs) guarantee convergence even if updates are received in different orders
  - Commutative semantics

- Example: integer counter
  - Replicas do not store only the last value …
  - … but the set of add/remove operations performed
  - These operations can be applied in any order in different replicas, and yield the same result

Eventual consistency

- Another example
  - Set
  - This changes the semantics with respect to a classical set

- Set: add(a), add(a), remove(a)

- In a traditional implementation these operations are not commutative
  - add(a), add(a), remove(a) leaves the set empty
  - add(a), remove(a), add(a) leaves a in the set

Eventual consistency

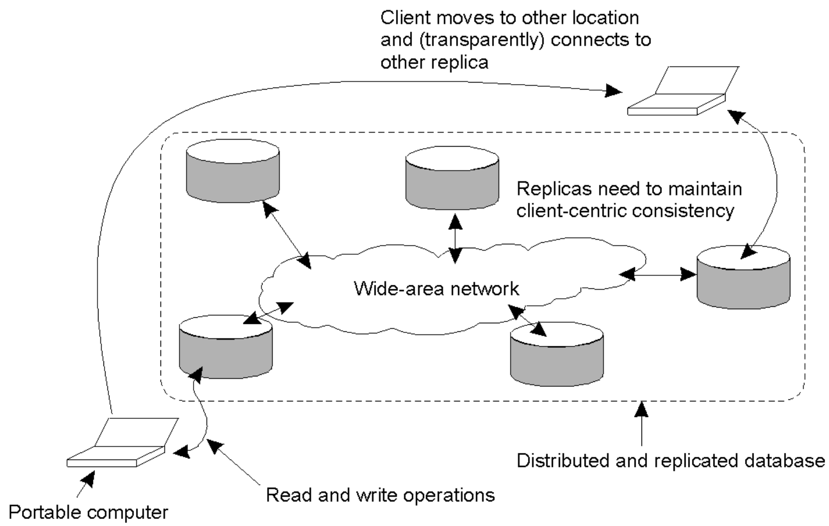
- A commutative set stores all the updates
  - If there are two add and one remove, the element is still there
- Different semantics wrt traditional sets …
- … but guarantees convergence under eventual consistency
- Reasonable trade-off between performance and complexity
  - Often used in geo-replicated data stores

CLIENT-CENTRIC CONSISTENCY MODELS

Client-centric consistency

- What happens if a client dynamically changes the replica it connects to?

- Problem addressed by client-centric consistency models that provide guarantees about accesses to the data store from the perspective of a single client



Monotonic reads

"If a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more recent value."

Monotonic reads

WS xi[t] is the set of write operations at Li that lead to version xi of x (at time t);

- Once a process reads a value from a replica, it will never see an older value from a read at a different replica

The set of writes known at a data store contains the update of x at L1 and the update of x at L2, in this order

Read on x_1, the value of x at data store L1

L1 and L2 are local copies of the data store, accessed by the same process

L1: $WS(x_1)$ $R(x_1)$

L2: $WS(x_1;x_2)$ $R(x_2)$

Consistent

L1: $WS(x_1)$ $R(x_1)$

L2: $WS(x_2)$ $R(x_2)$ $WS(x_1;x_2)$

NOT Consistent

Monotonic Reads

Monotonic reads ensures that if a process performs read r1, then r2, then r2 cannot observe a state prior to the writes which were reflected in r1; intuitively, reads cannot go backwards.

Monotonic reads does not apply to operations performed by different processes, only reads by the same process.

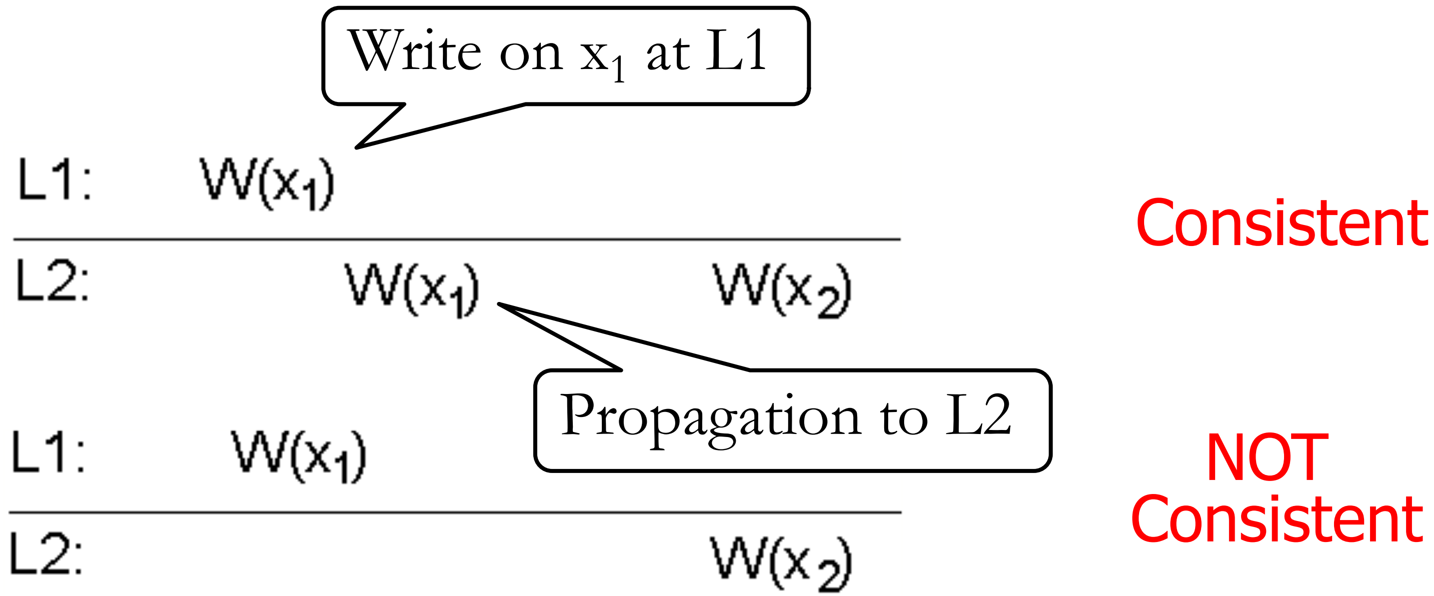
Monotonic reads can be totally available: even during a network partition, all nodes can make progress.

Monotonic writes

"A write operation by a process on a data item x is completed before any successive write operation on x by the same process."

Monotonic writes

- Similar to FIFO consistency, although this time for a single process
- A weaker notion where ordering does not matter is possible if writes are commutative
- x can be a large part of the data store (e.g., a code library)



Read your writes

“The effect of a write operation by a process on a data item x will always be seen by a successive read operation on x by the same process.”

Read your writes

- Examples: updating a Web page, or a password

L1:	$W(x_1)$	
L2:	$WS(x_1;x_2)$	$R(x_2)$

Consistent

L1:	$W(x_1)$	
L2:	$WS(x_2)$	$R(x_2)$

NOT Consistent

Writes follow reads

“A write operation by a process on a data item x following a previous read operation on x by the same process is guaranteed to take place on the same or more recent value of x that was read.”

Writes follow reads

- Example: guarantee that users of a newsgroup see the posting of a reply only after seeing the original article

L1: $WS(x_1)$ $R(x_1)$

L2: $WS(x_1;x_2)$ $W(x_2)$

Consistent

L1: $WS(x_1)$ $R(x_1)$

L2: $WS(x_2)$ $W(x_2)$

NOT Consistent

Does not follow the read on x_1

Client-centric consistency: implementation

- Each operation gets a unique identifier
  - e.g. ReplicaID + sequence number
- Two sets are defined for each client:
  - Read-set: the write identifiers relevant for the read operations performed by the client
  - Write-set: the identifiers of the write performed by the client
- Can be encoded as vector clocks
  - Latest read/write identifier from each replica

Client-centric consistency

- Monotonic-reads: before reading on L2, the client checks that all the writes in the read-set have been performed on L2
- Monotonic-writes: as monotonic-reads but with write-set in place of read-set
- Read-your-writes: see monotonic-writes
- Write-follow-reads: firstly, the state of the server is brought up-to-date using the read-set and then the write is added to the write-set

DESIGN STRATEGIES

Implementing replication

- We have seen some protocols to keep replicas consistent with respect to some consistency model

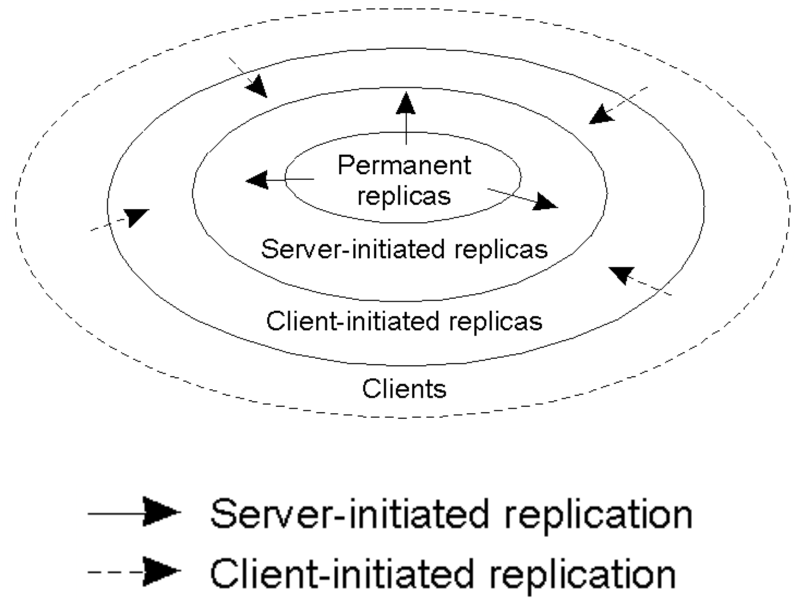
- There are further issues in designing a replicated datastore, including
  - How to place replicas?
  - What to propagate?
  - How to propagate updates between them?

Replica placement

- Permanent replicas
  - Statically configured
  - E.g., Web mirrors

- Server-initiated replicas
  - Created dynamically, e.g., to cope with access load
  - Move data closer to clients
  - Often require topological knowledge

- Client-initiated replicas
  - Rely on a client cache, that can be shared among clients for enhanced performance



Update propagation

- What to propagate?
  - Perform the update and propagate only a notification
    - Used in conjunction with invalidation protocols: avoids unnecessarily propagating subsequent writes
    - Small communication overhead
    - Works best if #reads << #writes
  - Transfer the modified data to all copies
    - Works best is #reads >> #writes
  - Propagate information to enable the update operation to occur at the other copies
    - Also called active replication
    - Very small communication overhead, but may require unnecessary processing power if the update operation is complex

Update propagation

- How to propagate?
  - Push-based approach
    - The update is propagated to all replicas, regardless of their needs
    - Typically used to preserve high degree of consistency
  - Pull-based approach
    - An update is fetched on demand when needed
    - More convenient if #reads << #writes
    - Typically used to manage client caches, e.g., for the Web
  - Leases can be used to switch between the two
    - They were actually developed to deal with replication…

Update propagation

address of all the clients

Issue	Push-based	Pull-based
State of server	List of client replicas and caches	None
Messages sent	Update (and possibly fetch update later)	Poll and update
Response time at client	Immediate (or fetch-update time)	Fetch-update time

?

Comparison assuming one server and multiple clients, each with its own cache

Propagation strategies

- Anti-entropy
  - Server chooses another at random and exchanges updates
  - Options
    - Push (A⇨B), pull (A⇦B), or push-pull (A⇔B)
    - Positive (changes seen) vs. negative (changes missed) notifications
  - Eventually all servers receive all updates
    - Eventual consistency

Propagation strategies

- Gossiping (or rumor spreading) •流言蜚语（或谣言传播）
  - An update propagation triggers another towards a different server
    - If the update has already been received, the server reduces the probability (e.g., 1/k) to gossip further
    - Fast spreading but only probabilistic guarantees for propagation
    - May be complemented with anti-entropy
    - 如果已收到更新，服务器将降低进一步闲谈的可能性（例如1/K）
    - 传播速度快，但只有传播的概率保证
    - 可辅以反熵

Propagation strategies

- How to deal with deletion?
  – Treat it as another update
    - "Death certificates" with expiration

- Several variations available
  – E.g., choosing the gossiping nodes based on connectivity, distance, or entirely at random

Properties of gossiping

- Intrinsically distributed and redundant
  - Scalable
  - Fault-tolerant
  - Resilient to topological changes
- Gossip is not broadcast!
  - Broadcast can be regarded as a special case of gossip, with more overhead and less reliability
- Applied to several fields, including multicast communication (for spreading and/or recovering events), publish-subscribe, resource location, …

Transactional models

- Data consistency is a key aspect in distributed databases, in continuous evolution
- Data can be partially replicated
  - Replicated and partitioned
  - Not all the replicas have all the partitions
- Transactions define groups of operations and provide ACID guarantees
  - Atomic: a transaction either entirely succeeds or entirely fails
  - Consistency: a transaction does not violate application correctness constraints
  - Isolation: transactions do not interfere with each other
  - Durability: changes are permanent

Transactional models

- Different models define different guarantees
  - For instance, they allow some disciplined form of interference

- You will study how to guarantee serializable isolation
  - Similar to sequential consistency
  - Transactions occur as if they were executed in some serial order
  - Incurs similar synchronization costs
  - Various relaxed models have been proposed

Transactional models

- Various trade-offs have become popular over time
- From distributed relational DB
  - Transactions with "strong" guarantees of consistency and isolation
  - Limited performance/availability
- To NoSQL (~2000)
  - No transactions
  - Eventual consistency
  - High performance

Transactional models

- To NewSQL (today)
  - Transactions with strong consistency and isolation
  - Sufficient performance/availability

- Examples
  - Spanner (Google, 2012)
    - Synchronization based on TrueTime (precise GPS/atomic clocks)
  - Calvin (Yale, 2012)
    - Pre-processing to re-order transactions
  - VoltDB (Brown, MIT, Yale, 2009)
    - Users provide explicit information on replication and partitioning
    - This information is used to optimize transactions in distributed settings

Bibliography

- M. Van Steen, A. S. Tanenbaum "Distributed Systems" 3rd edition, 2017
- M. Kleppmann "Designing Data-Intensive Applications: the Big Ideas Behind Reliable, Scalable, and Maintainable Systems", 2017
- P. Viotti, M. Vukolic "Consistency in Non-Transactional Distributed Storage Systems", ACM Computing Surveys, 2016
- P. Bailis et al. "Highly Available Transactions: Virtues and Limitations", VLDB, 2014