5. Consistency and Replication

Sistemes Distribuïts en Xarxa (SDX)
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)
2018/2019 Q2

FIB

Contents

- **Introducing replication & consistency**
- Data-centric consistency models
- Client-centric consistency models
- Distribution protocols
- Consistency protocols

Introduction

- Reasons for replication
  A. Increase **availability**: data is available despite server failures and network partitions
  B. Enhance **reliability**: data is correct on the presence of faults (e.g. protection against data corruption, Byzantine failures, and stale data)
  C. Improve **performance**: this directly supports the goal of enhanced **scalability**
     - Size: Replicate data and distribute work instead of having one single server
     - Geographical: Replicate data close to where it is used (e.g. data caching)

Introduction

- What are the issues with replication?
  - Replication should be **transparent**
    - Clients are not aware that multiple copies of data exist
  - Replicated data should be **consistent**
    - Clients see the same data despite the replica they read

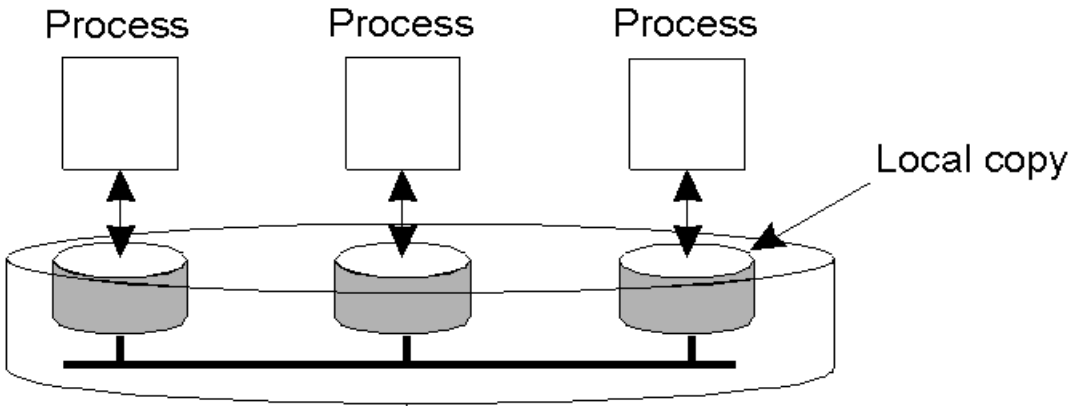
  ⇒ How do we transparently (and efficiently) keep all the replicas up-to-date and consistent?
- Dilemma: replication improves scalability, but incurs the overhead of synchronizing replicas
  - The solution often results in a relaxation of the consistency constraints

Contents

- Introducing replication & consistency
- **Data-centric consistency models**
- Client-centric consistency models
- Distribution protocols
- Consistency protocols

Consistency models

- Consistency models in a data store
  - Each process has a local replica of the data store
  - Write operations to a local replica need to be propagated to all remote replicas
  - Model specifies precisely what the results of read and write operations are with concurrency



Consistency models

- Diagram notation
  - Time axis drawn horizontally with time increasing from left to right in all diagrams
  - $W_i(x)a$ – a write by process 'i' to item 'x' with a value 'a'
    - i.e. 'x' is set to 'a'
    - Note: The process is often shown as 'P_i'
  - $R_i(x)b$ – a read by process 'i' from item 'x' returning the value 'b'
    - i.e. reading 'x' returns 'b'

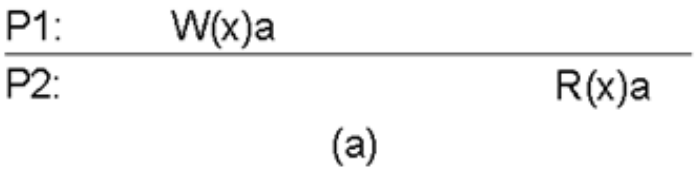
Contents

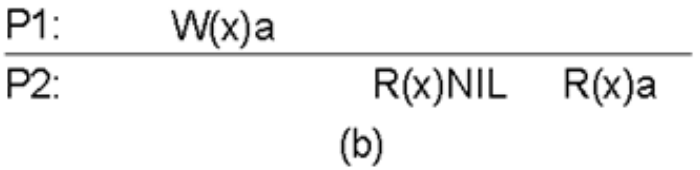
- Introducing replication & consistency
- **Data-centric consistency models**
  - **Strong consistency models**
  - Relaxed consistency models
- Client-centric consistency models
- Distribution protocols
- Consistency protocols

Strict consistency

- Definition:
  - Any read on a data item 'x' returns the value of the **most recent** write on 'x', regardless of in which replica the write occurred
    - Assuming non-blocking read and write operations
- An absolute global time order is maintained
  - ↓ Writes must be instantaneously visible everywhere
    - Not feasible due to network latencies
  - ↓ Requires perfectly synchronized clocks
  - ↓ Needs timestamps with infinite precision to ensure that at most one operation occurs at a time

Strict consistency





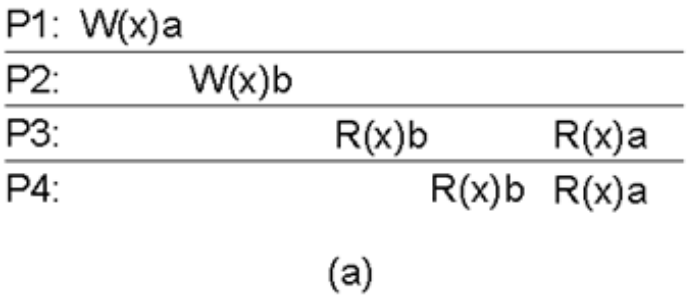
a) Data store with strict consistency
b) Data store that is not strictly consistent

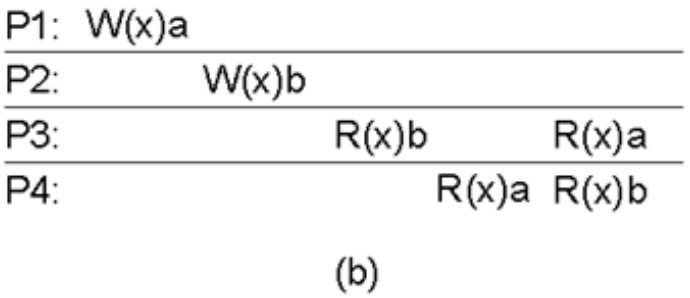
- It is the ideal consistency model
  - Corresponds to true replication transparency
- But it is impossible to achieve within a distributed system

Sequential consistency

- Definition:
  - The result of any execution is the same as if the (read and write) operations by all processes on the data store were executed in the same sequential order and the operations of each individual process appear in this sequence in the order specified by its program
- In other words: all processes see the same interleaving of operations, regardless of what that interleaving is
- Real time is not taken into consideration

Sequential consistency



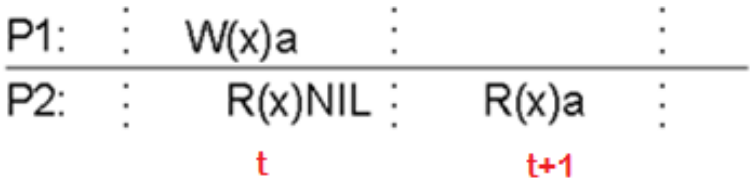


a) Data store with sequential consistency
b) Data store that is not sequentially consistent

- Weaker than strict consistency
- Easier to implement
  - Using total+FIFO multicast
  - See 'Consistency protocols' lesson for details

Linearizability (a.k.a. strong consistency)

- Operations receive a timestamp with a global available clock that is loosely synchronized
  - We have finite precision, thus two operations can be assigned with the same timestamp
- Def: sequential consistency + if ts(x) < ts(y) then op(x) precedes op(y) in the interleaving
  - strict > linearizability > sequential consistency
  - Can be implemented if clocks are synchronized with real time, and message delays are bounded



Other strong consistency models

- Causal consistency
  - Writes that are potentially causally related must be seen by all processes in the same order
    - Concurrent writes may be seen in a different order by different processes
  - Implemented using causally-ordered multicast
- FIFO consistency
  - Writes done by a single process are seen by all the others in the order in which they were issued
    - Writes from different processes may be seen in a different order by different processes
  - Implemented using FIFO-ordered multicast

Contents

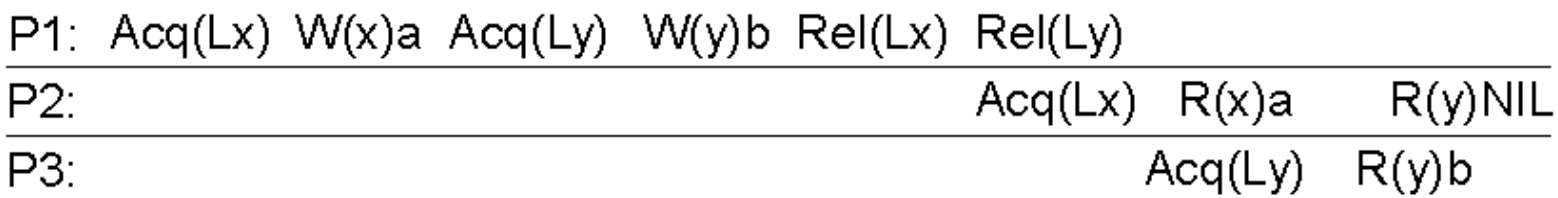
- Introducing replication & consistency
- **Data-centric consistency models**
  - Strong consistency models
  - **Relaxed consistency models**
- Client-centric consistency models
- Distribution protocols
- Consistency protocols

Relaxed consistency models

- Not all the applications need to see all writes
- Weaker semantics by enforcing consistency on groups of operations (i.e. critical section) instead of individual reads and writes
  - Each process performs operations on its local copy of the data store
  - Process propagates only the results at the end of the critical section
    - Do not worry about propagating intermediate results
  - Critical section is delimited by means of **synchronization variables**

Relaxed consistency models

- Operate in the variable to synchronize all the copies in the data store
  - **Acquire**: local copy of the protected data is updated to be consistent with the remote ones
  - **Release**: protected data that have been changed are propagated out to the remote copies
- Relaxed models: weak consistency, release consistency, entry consistency



Contents

- Introducing replication & consistency
- Data-centric consistency models
- **Client-centric consistency models**
- Distribution protocols
- Consistency protocols

Eventual consistency

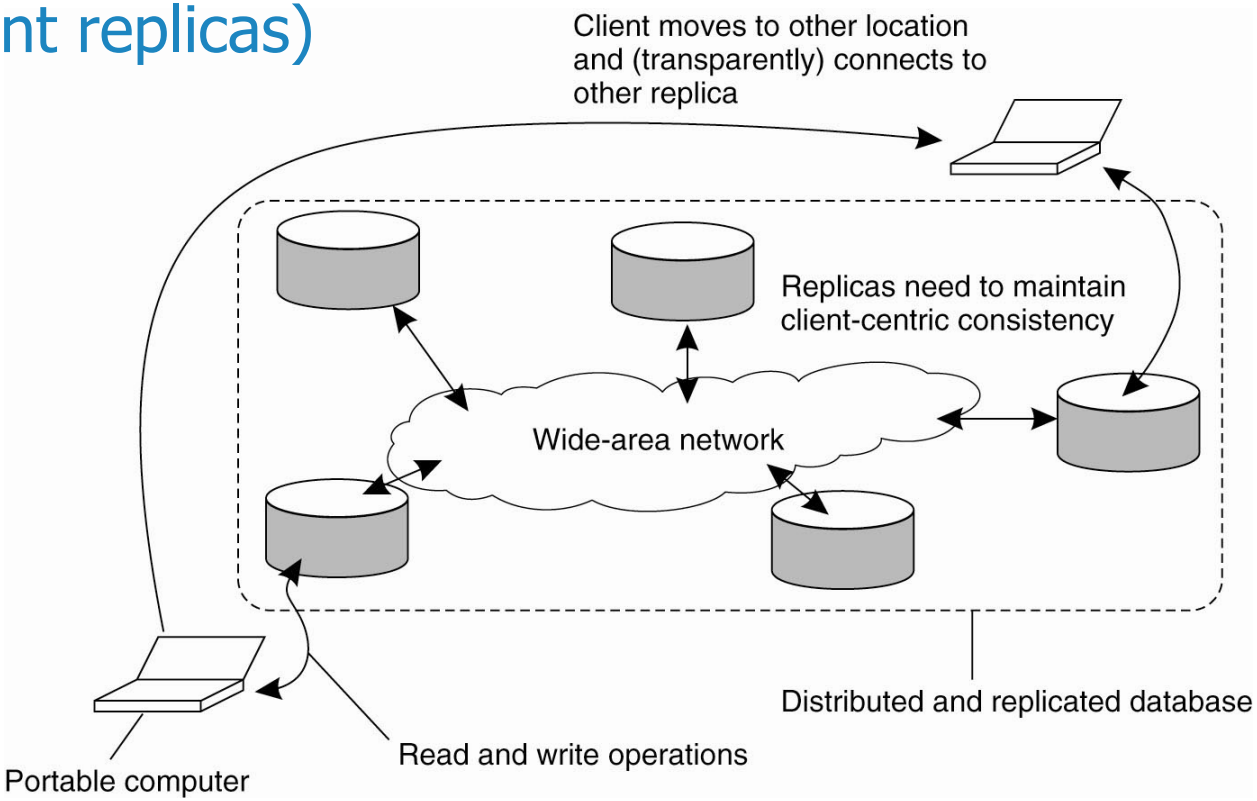
- Data-centric models aim to provide a system-wide consistent view on data stores with concurrent write operations
  - These models require highly available connections
- **Eventual consistency** targets stores that:
  - Execute mainly read operations and have none or few write-write conflicts
  - Have users that often accept reading stale data
  - Deal with disconnected users, network partitions, and users preferring availability vs. consistency
  - e.g. DNS, Amazon Dynamo, Dropbox

Eventual consistency

- In absence of new updates, eventually all accesses will return the last updated value
  - Only requires that all updates are guaranteed to propagate to all replicas … eventually!
- Writes are not ordered when executed, which might create write-write conflicts
  - Conflict resolution is needed
- Eventual consistency works well as long as every client always updates the same replica

Eventual consistency

- Things are harder if clients are mobile (they operate on different replicas)



⇒ Use **client-centric** consistency models

Client-centric consistency models

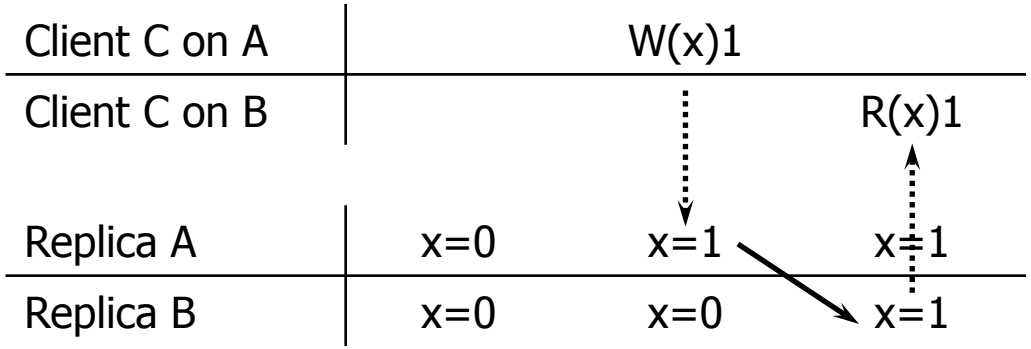
- Guarantee that **a single client** sees its accesses to the data store in a consistent way
  - No guarantees are given concerning concurrent accesses by multiple clients
- There are four typical client-centric models

  i) Read Your Writes, ii) Monotonic Reads,

  iii) Monotonic Writes, iv) Writes Follow Reads
  - For their implementation, we keep track of two sets of writes for each client
    - **Read set:** writes relevant (whose effects were visible) for the read operations performed by that client
    - **Write set:** writes performed by that client

Read Your Writes

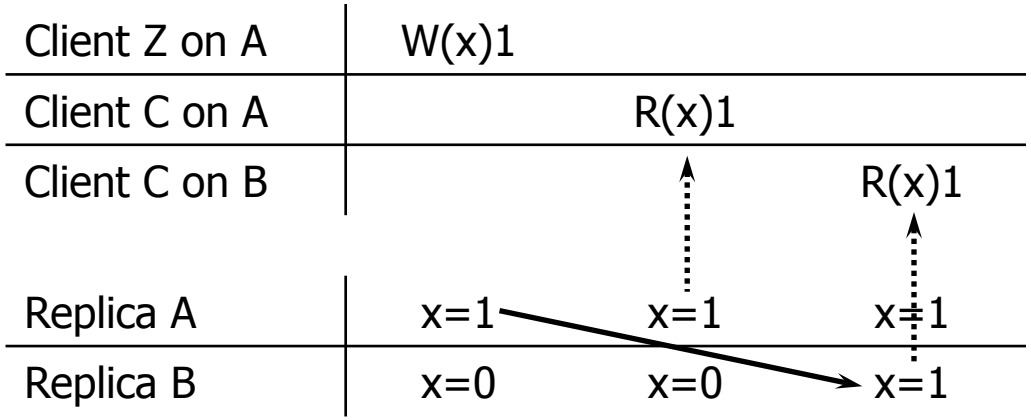
- If client C writes a data item 'x' on replica A, any successive read on 'x' by C on replica B will return the written value or a more recent one
  - e.g. updating your web page



- Replica A updates the client's write set WS; Replica B checks WS before reading to ensure that all writes in the set have taken place locally

Monotonic Reads

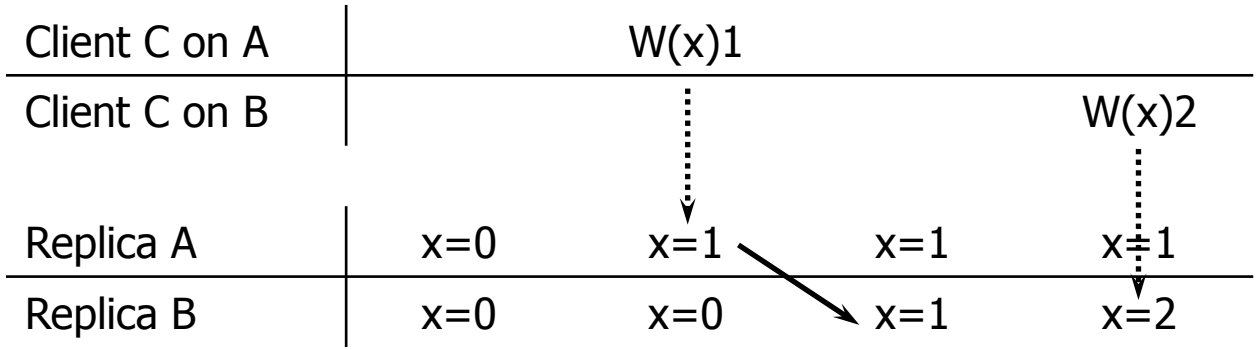
- If client C reads a data item 'x' on replica A, any successive read on 'x' by C on replica B will return the same value or a more recent one
  - e.g. reading email while you are on the move



- Replica A updates the client's read set RS; Replica B checks RS before reading to ensure that all writes in the set have taken place locally

Monotonic Writes

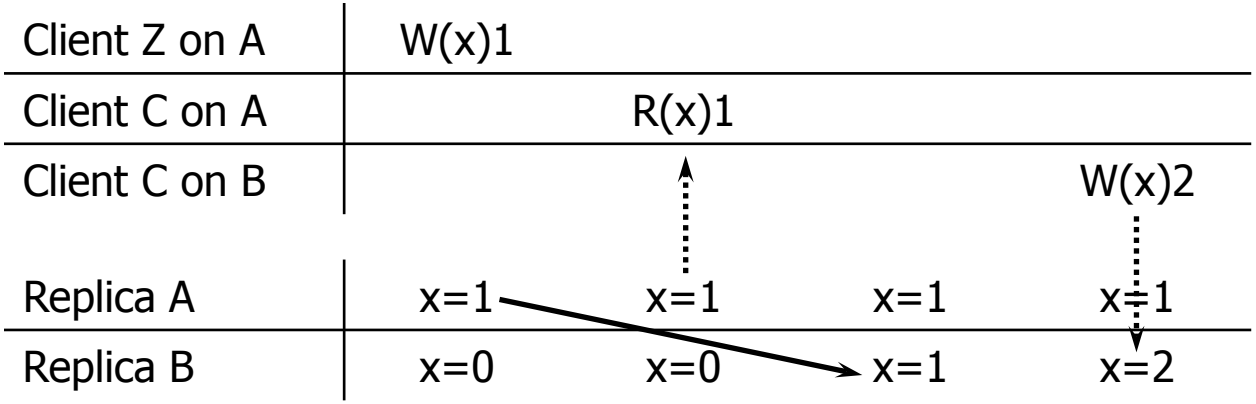
- A write by client C on a data item ‘x’ on replica A is completed also in replica B before C performs any successive write on ‘x’ on replica B
  - e.g. versioning a software library



- Replica A updates the client’s write set WS; Replica B checks WS before writing to ensure that all writes in the set have taken place locally

Writes Follow Reads

- If C reads a data item ‘x’ on replica A, relevant writes for that read are completed also in replica B before C performs any successive write on ‘x’ on replica B
  - e.g. distributed newsgroup. A reaction (write) can be posted only after seeing the original post (read)



- Replica A updates the client’s read set RS; Replica B checks RS before writing to ensure that all writes in the set have taken place locally

Summary of strong models

Consistency	Description
Strict	All processes see the result of the most recent write
Linearizability	All processes see the same interleaving of operations, which must be ordered according to a global timestamp
Sequential	All processes see the same interleaving of operations, which must preserve program order but might not be ordered in time
Causal	All processes see causally-related operations in the same order
FIFO	All processes see operations from each other in the order they were issued

- Examples:
  - NewSQL data stores: Google Spanner, Microsoft Yesquel
  - NoSQL data stores: Hyperdex, Apache HBase, MongoDB (*), COPS & Eiger (causal)
  - Distributed shared memory: IVY

(*) extended to support also eventual consistency

Summary of relaxed models

Consistency	Description
Weak	Shared data are made consistent after doing a synchronization
Release	Shared data are made consistent when a critical region is entered and/or exited
Entry	Shared data are made consistent only when a critical region is entered

- They use synchronization variables, hence requiring additional programming constructs
  - Allow programmers to treat the data store as if it is sequentially consistent, when in fact it is not
- Example: Distributed Shared Memory: TreadMarks

Summary of eventual models

Consistency	Description
Monotonic Reads	If a process has seen a particular value for the object, any subsequent accesses will never return any previous values
Monotonic Writes	The writes by the same process are serialized
Read Your Writes	A process, after updating an object, always accesses the updated value and never sees an older value
Writes Follow Reads	A write by a process is ordered after any writes whose effects were seen by previous reads by that process

- Examples:
  - NoSQL data stores: Apache CoughDB, Amazon Dynamo (*), Apache Cassandra (*), Riak (*), LinkedIn Voldemort
  - File synchronizers: Dropbox
  - DNS

(*) extended to support also strong consistency

READING REPORT

[Terry13] Terry, D., *Replicated Data Consistency Explained Through Baseball,* Communications of the ACM, Vol. 56, No. 12, pp. 82-89, December 2013

Contents

- Introducing replication & consistency
- Data-centric consistency models
- Client-centric consistency models
- **Distribution protocols**
- Consistency protocols

Distribution protocols

- Implementation issues regardless of which consistency model is supported

1. Replica placement
   - Decide where, when, and by whom copies of the data store are to be placed
2. Update propagation
   - Decide which mechanisms to use for keeping the replicas consistent
     - What to propagate
     - How to propagate updates

Replica placement

1. **Permanent replicas**
   - Initial set of replicas of the data store
   - Small in number, organized as clusters or mirror sites
2. **Server-initiated replicas**
   - Used to enhance performance
   - Created at the initiative of the owner of the data store
   - e.g. web server locates replicas close to where they are needed most (a.k.a. 'push caches')
3. **Client-initiated replicas (a.k.a. client caches)**
   - Used to improve access times to data
   - Created as a result of client requests
   - e.g. browser caches

Update propagation

- What is actually propagated when a client initiates an update to a data store?
  A. Propagate a notification to other replicas
     - Recipients **invalidate** their copy
     - Good when read to write ratio is low
  B. Transfer the data from one replica to another
     - Recipients **replace** their copy
     - Good when read to write ratio is high
  C. Propagate the update operation to other replicas
     - This is **active replication**
     - Each replica must be capable of 'actively' keeping its data up-to-date by performing operations

Update propagation

A. Push-based/Server-based approach

- Update sent 'automatically' by server
- Useful when a high degree of consistency is needed and the read to write ratio is high
- Often used between permanent and server-initiated replicas

B. Pull-based/Client-based approach

- Updates are requested by the client
  - No request ⇒ no update!
- Useful when the read to write ratio is low
- Used by client caches (e.g. browsers)

Push vs. pull protocols

- Comparison (* when using invalidations)

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

- Hybrid schemes are possible (e.g. **Leases**)
  - Server promises to push updates to client for a period of time. Once lease expires, client reverts to a pull approach (until another lease is issued)
- Adaptive leases
  - Lease expiration time depends on system behavior

Adaptive leases

A. Age-based leases

- Lease depends on the last time the item was modified
- An item that has not changed for a long time, will not change in a near future, so provide a long-lasting lease

B. Renewal-frequency-based leases

- Lease depends on how often a specific client requests a cached item to be refreshed
- The more often, the longer the lease

C. State-based leases

- Lease depends on state-space overhead at the server
- The more loaded a server is, the shorter the expiration times become (server must keep track of fewer clients)

Contents

- Introducing replication & consistency
- Data-centric consistency models
- Client-centric consistency models
- Distribution protocols
- **Consistency protocols**

Consistency protocols

- Consistency protocols describe:
  A. Implementations of specific consistency models
     - We focus on sequential consistency and linearizability
  B. Architectures of replicated processes for fault tolerance
     - Hierarchical and flat process groups
- Consistency protocols discussed:
  1. Primary-based protocols
     - Writes go to a single replica
  2. Replicated-write protocols
     - Writes can go to any replica

Contents

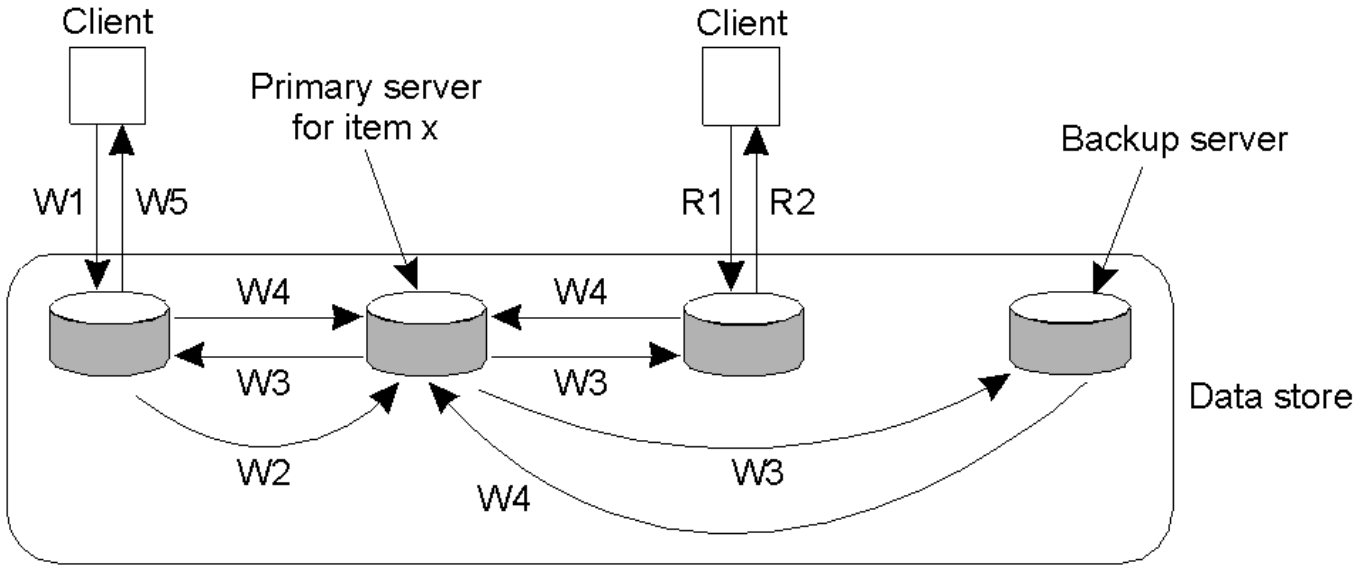
- Introducing replication & consistency
- Data-centric consistency models
- Client-centric consistency models
- Distribution protocols
- **Consistency protocols**
  - **Primary-based protocols**
  - Replicated-write protocols

Primary-based protocols

- Each data item is associated with a **primary replica** which is in charge of coordinating write operations to the data item
  - If the primary fails, one of the replicas is elected to act as the primary
- Also called passive replication protocols
- Two types:
  A. Primary-backup remote-write protocols
  B. Primary-backup local-write protocols

Primary-backup remote-write protocols

- All writes are done at a fixed single replica
- Reads can be carried out locally



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

R1. Read request
R2. Response to read

Primary-backup remote-write protocols

↓ Bad write performance

- Writes can take a long time because a blocking write protocol is used

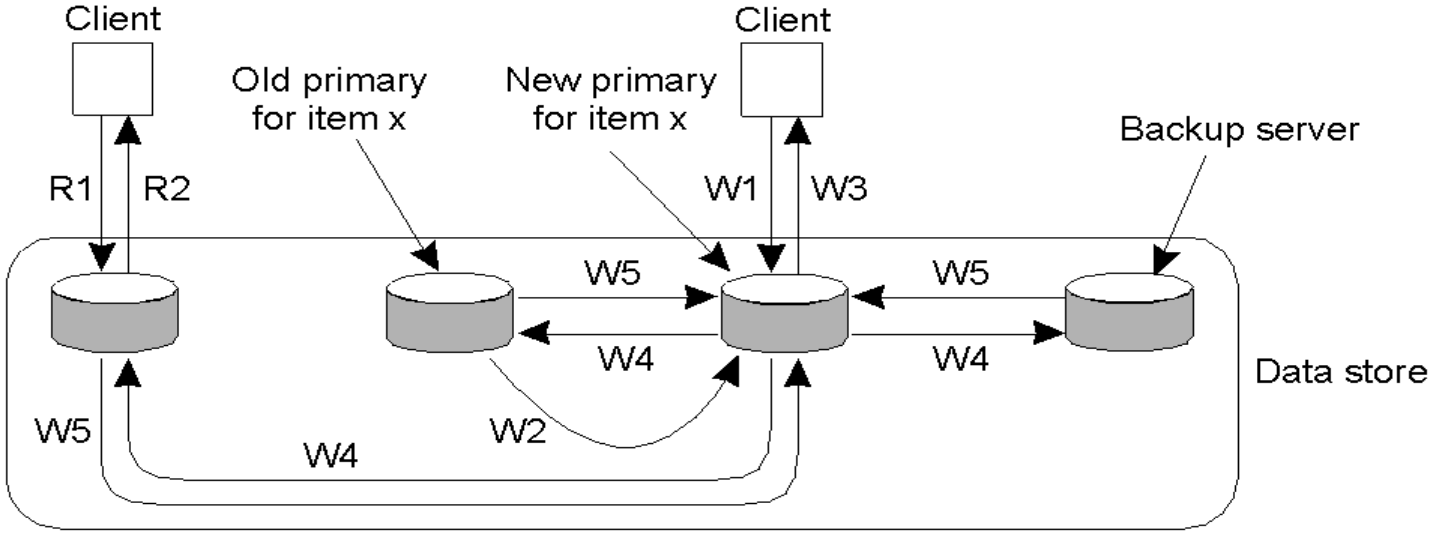
- Alternative: Use a non-blocking write protocol
  - Primary acknowledges the client's replica just after updating its local copy
  - If some replica failed during the update, read-your-writes consistency is not guaranteed

Primary-backup remote-write protocols

- ↑ Easy to implement sequential consistency
  - Primary sends the write operations to each replica via FIFO-ordered view-synchronous atomic multicast (see 'Multicast' lesson for details)
    - Having a single primary and FIFO-order ensure that all replicas will see the writes in the same order
    - Virtual synchrony allows the system to take over exactly where it left off upon primary failure
- Implements linearizability if read requests are also forwarded to the primary

Primary-backup local-write protocols

- Primary migrates to the replica that is writing, successive writes are carried out locally, and then the replicas are updated using a non-blocking protocol

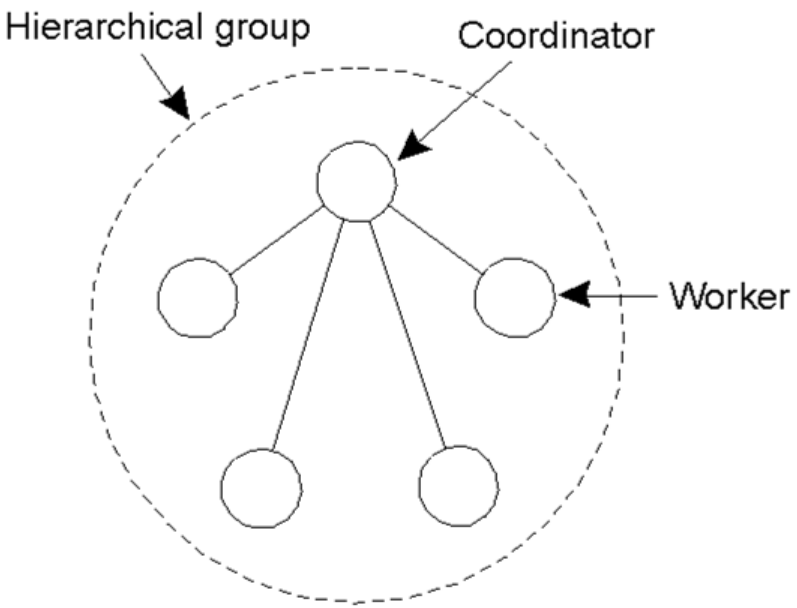


W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read

Hierarchical process groups

- Use process replication to build a hierarchical process group that tolerates process failures
  - Processes are organized in a hierarchical fashion
- Implemented through a **primary-based** protocol
  - The coordinator acts as the primary
  - The workers act as backups



Hierarchical process groups

1. Clients send requests to the coordinator
   - Clients might be able to submit read requests to the workers
2. Coordinator provides the requested service and updates the workers using FIFO-ordered view-synchronous atomic multicast

- If the coordinator fails, one of the workers is promoted to act as new coordinator
  - K + 1 processes are needed to survive K crash or omission process failures
  - Cannot tolerate Byzantine process failures

Contents

- Introducing replication & consistency
- Data-centric consistency models
- Client-centric consistency models
- Distribution protocols
- **Consistency protocols**
  - Primary-based protocols
  - **Replicated-write protocols**

Replicated-write protocols

- Writes can be carried out at **multiple replicas** instead of only one, as occurs in primary-based protocols
- Also called distributed-write protocols
- Two types:
  1. Active replication
  2. Quorum-based protocols (majority voting)

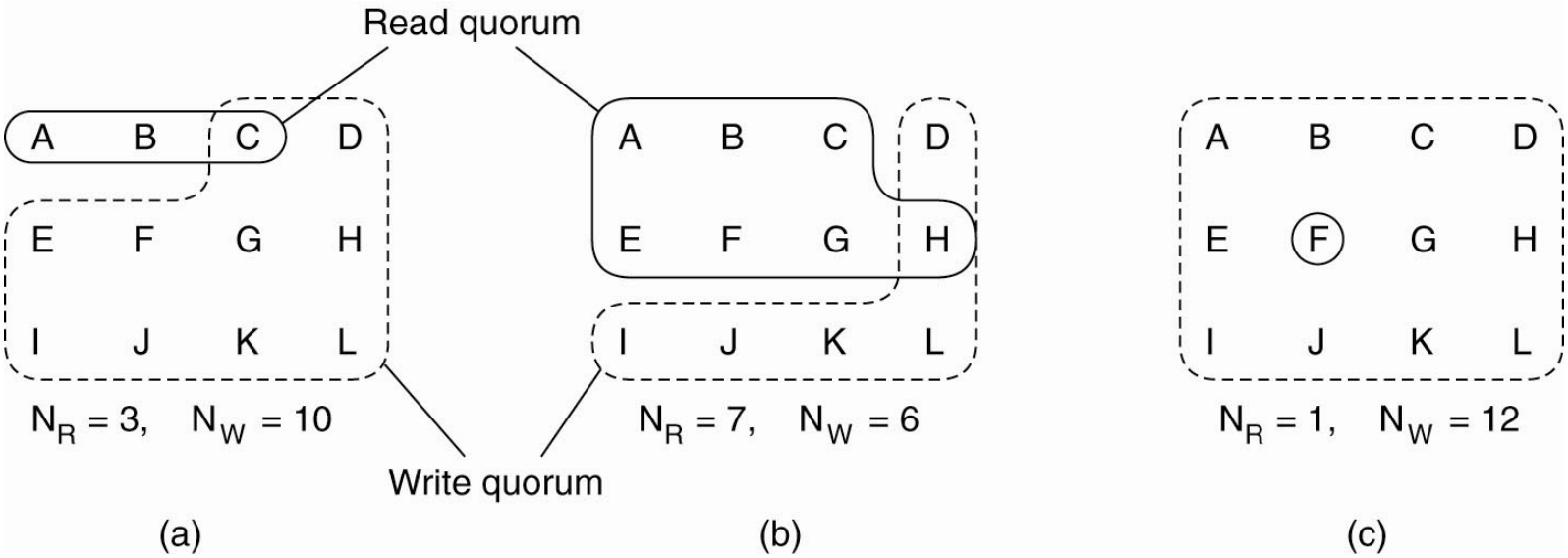
Active replication

- Each operation is forwarded to all replicas
  - Each replica has an associated process to carry out operations
- Operations need to be carried out in the same order everywhere
  - Client uses **total+FIFO-ordered multicast** to send operation to the group of replicas
    - See 'Multicast' lesson for details
  - Total+FIFO-ordered multicast ensures sequential consistency if we multicast only writes, and linearizability if we multicast also reads

Quorum-based protocols

- Clients must get permission from a quorum of the N replicas before either reading or writing
  - Read quorum of N_R replicas before reading
    - Not all of the replicas in N_R need to be up-to-date
    - Any up-to-date replica may be read
  - Write quorum of N_W replicas before writing
    - Update replicas in quorum, assign new version number
    - Remaining replicas are updated as a background task
  - $\mathbf{N_R + N_W > N}$ to avoid read-write conflicts
  - $\mathbf{N_W > N/2}$ to avoid write-write conflicts
  - Writes are serialized and reads return the latest version that was written $\rightarrow$ linearizability

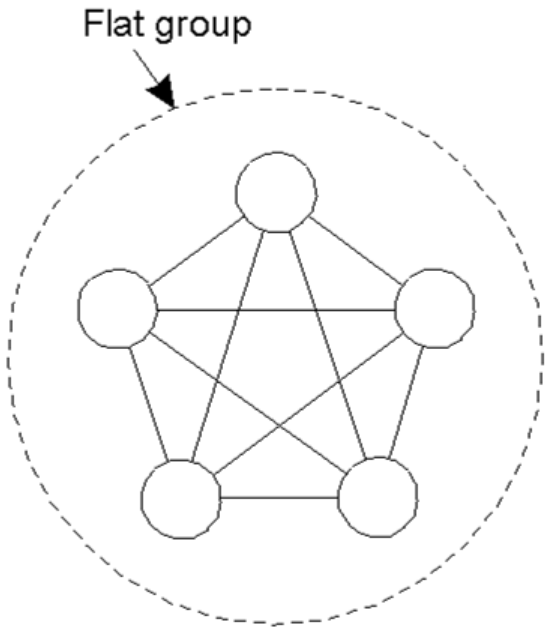
Quorum-based protocols



a) Correct choice of read and write set
b) Choice that may lead to write-write conflicts
c) Correct choice, known as ROWA (read-one,write-all)

Flat process groups

- Use process replication to build a flat process group that tolerates process failures
  - Processes are identical and the group response is defined through **voting**
- Implemented through a **replicated-write** protocol
  - Typically, in the form of active replication



Flat process groups

1. Clients send their requests to all workers using total+FIFO-ordered atomic multicast
2. Workers reply the request independently but identically

- If any worker fails, the others continue to reply in the normal way
  - K + 1 processes are needed to survive K crash or omission process failures
  - 2K + 1 processes are needed to survive K Byzantine process failures
    - K failing processes could generate the same wrong reply, thus we need K+1 correct processes

Summary

- Reasons for replication
  - Improved availability
  - Enhanced reliability
  - Improved performance & scalability
- But replication can lead to inconsistencies …
- How can we propagate updates so that these inconsistencies are not noticed without severely degrading performance?
- Proposed solutions apply for the relaxation of any existing consistency constraints

Summary

- Consistency models
  - Data-centric models
    - Strict, Linearizability, Sequential, Causal, FIFO concern themselves with individual reads/writes to data items
    - Weaker models, which introduce the notion of synchronization variables, concern themselves with a group of reads/writes
  - Client-centric models
    - Concerned with maintaining consistency for a single client access to the distributed data store
    - Models based on Eventual consistency
      - Read your writes, Monotonic reads, Monotonic writes, Writes follow reads

Summary

- We looked at 'distribution protocols' designed to facilitate the propagation of updates
- We looked at 'consistency protocols' as a way to implement consistency models
  - The most common schemes are those that support sequential consistency or linearizability, but eventual consistency is gaining popularity
- Further details:
  - [Tanenbaum]: chapter 7
  - [Coulouris]: chapter 18