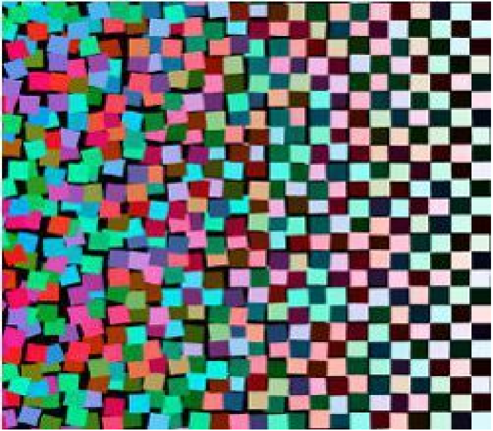
# Consistency and Replication



# Replicas and Consistency???

# Replicas and Consistency???

# Replicas and Consistency???

*Tatiana Maslany in the show Orphan Black: The story of a group of clones that discover each other and the secret organization Dyad, which was responsible for their creation.*

# Reasons for Replication

- To increase the reliability of a system.

# Reasons for Replication

- To increase the reliability of a system.
- To increase the performance of a system. This can be of two types:

# Reasons for Replication

- To increase the reliability of a system.
- To increase the performance of a system. This can be of two types:
  - Scaling in numbers
  - Scaling in geographical area

# Reasons for Replication

- To increase the reliability of a system.
- To increase the performance of a system. This can be of two types:
  - Scaling in numbers
  - Scaling in geographical area
- Having multiple copies leads to the problem of consistency. When and how the copies are made consistent determines the price of replication.

# Reasons for Replication

- To increase the reliability of a system.
- To increase the performance of a system. This can be of two types:
  - Scaling in numbers
  - Scaling in geographical area
- Having multiple copies leads to the problem of consistency. When and how the copies are made consistent determines the price of replication.
- **Example**: Client caching of web pages in web browser gains performance for the client at the cost of consistency.

# Replication as a Scaling Technique

- Placing copies of data close to client processes can help with scaling. But keeping copies up to date requires more network bandwidth. Updating too often may be a waste. Not updating often enough is the flip side.

# Replication as a Scaling Technique

- Placing copies of data close to client processes can help with scaling. But keeping copies up to date requires more network bandwidth. Updating too often may be a waste. Not updating often enough is the flip side.
- How to keep the replicas consistent? Use global ordering using Lamport timestamps or use a coordinator. This may require a lot of communication for a large system.

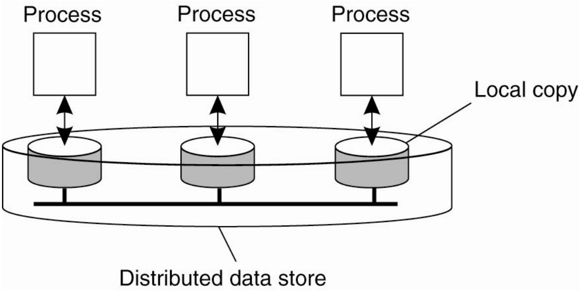
# Replication as a Scaling Technique

- Placing copies of data close to client processes can help with scaling. But keeping copies up to date requires more network bandwidth. Updating too often may be a waste. Not updating often enough is the flip side.
- How to keep the replicas consistent? Use global ordering using Lamport timestamps or use a coordinator. This may require a lot of communication for a large system.
- In many cases, the real solution is to loosen consistency constraints. E.g. The updates do not have to be atomic. To what extent we can loosen depends highly on the access and update patterns as well as the application.

# Replication as a Scaling Technique

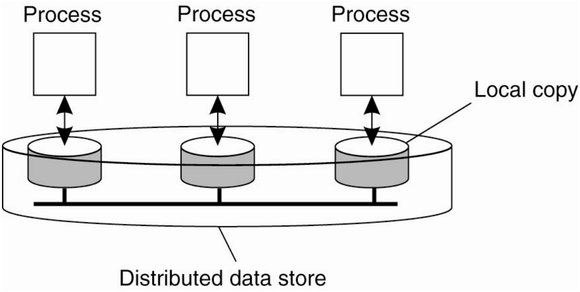
- Placing copies of data close to client processes can help with scaling. But keeping copies up to date requires more network bandwidth. Updating too often may be a waste. Not updating often enough is the flip side.
- How to keep the replicas consistent? Use global ordering using Lamport timestamps or use a coordinator. This may require a lot of communication for a large system.
- In many cases, the real solution is to loosen consistency constraints. E.g. The updates do not have to be atomic. To what extent we can loosen depends highly on the access and update patterns as well as the application.
- A range of consistency models are available.

# Data Store



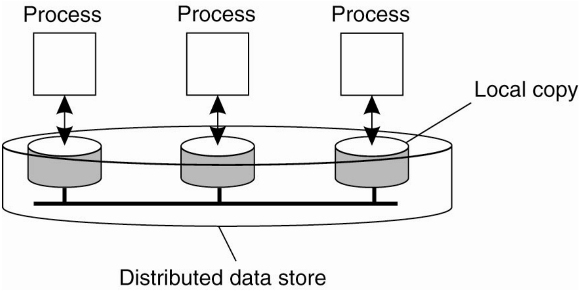

- Examples:
  - distributed shared memory
  - distributed shared database
  - distributed file system

# Data Store




- Examples:
  - distributed shared memory
  - distributed shared database
  - distributed file system
- Each process has a local or nearby copy of the whole store or part of it.

# Data Store




- Examples:
  - distributed shared memory
  - distributed shared database
  - distributed file system
- Each process has a local or nearby copy of the whole store or part of it.
- A data operation is classified as a write when it changes the data, and is otherwise classifies as a read operation.

# What is a Consistency Model?

- A consistency model is essentially a contract between processes and the data store. It says that if processes agree to obey certain rules, the store promises to work correctly.

# What is a Consistency Model?

- A consistency model is essentially a contract between processes and the data store. It says that if processes agree to obey certain rules, the store promises to work correctly.
- Models with minor restrictions are easy to use while models with major restrictions are more difficult to use. But then easy models don't perform as well. So we have to make trade-offs.

# Consistency Models

- Data-centric consistency models
  - Continuous consistency
  - Sequential consistency
  - Causal consistency
  - Entry Consistency with Grouping operations
- Client-centric consistency models
  - Eventual consistency
  - Monotonic reads
  - Monotonic writes
  - Read your writes
  - Writes follow reads

# Sequential Consistency (1)

A data store is sequentially consistent when:

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 ***some sequential order*** and the operations of each individual process appear in this sequence in the order specified by its program.

# Sequential Consistency (2)

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

- Behavior of two processes operating on the same data item. The horizontal axis is time.

# Sequential Consistency (3)

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

(a)

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

(b)

(a) A sequentially consistent data store.

(b) A data store that is not sequentially consistent.

# Sequential Consistency (4)

| Process P1 | Process P2 | Process P3 |
| --- | --- | --- |
| $x \leftarrow 1$; | $y \leftarrow 1$; | $z \leftarrow 1$; |
| print(y, z); | print(x, z); | print(x, y); |

- Three concurrently-executing processes.

# Sequential Consistency (5)

| (a) | (b) | (c) | (d) |
|---|---|---|---|
| $x \leftarrow 1$; | $x \leftarrow 1$; | $y \leftarrow 1$; | $y \leftarrow 1$; |
| print(y, z); | $y \leftarrow 1$; | $z \leftarrow 1$; | $x \leftarrow 1$; |
| $y \leftarrow 1$; | print(x, z); | print(x, y); | $z \leftarrow 1$; |
| print(x, z); | print(y, z); | print(x, z); | print(x, z); |
| $z \leftarrow 1$; | $z \leftarrow 1$; | $x \leftarrow 1$; | print(y, z); |
| print(x, y); | print(x, y); | print(y, z); | print(x, y); |
| Prints: 001011 | Prints: 101011 | Prints: 010111 | Prints: 111111 |
| Signature: 001011 | Signature: 101011 | Signature: 110101 | Signature: 111111 |

- Four valid execution sequences for the three processes. The vertical axis is time.
- Signature is output of P1, P2, P3 concatenated.

# Causal Consistency (1)

For a data store to be considered causally consistent, it is necessary that the store obeys the following condition:

- **Causal Consistency**: Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes by different processes may be seen in a different order on different machines. Write by the same process are considered to also be causally related.
- Weaker than sequential consistency.

# Causal Consistency (2)

| | | | | | | |
|---|---|---|---|---|---|---|
| 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 |

- This sequence is allowed with a causally-consistent store, but not with a sequentially consistent store. Why?

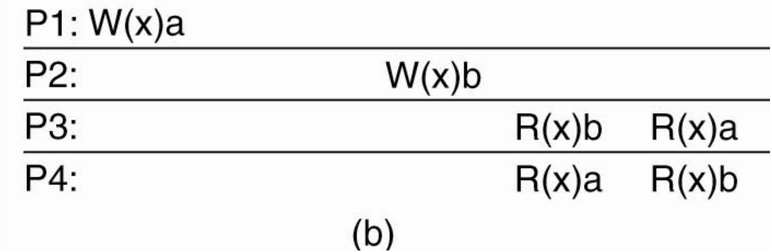
# Causal Consistency (3)

| | | | | |
|---|---|---|---|---|
| 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 |

(a)

- A violation of a causally-consistent store. Why?

# Causal Consistency (4)




- A valid sequence of events in a causally-consistent store. Concurrent writes do not have to be globally ordered.

# Causal Consistency (5)

- Implementing causal consistency requires keeping track of which processes have seen which writes.

# Causal Consistency (5)

- Implementing causal consistency requires keeping track of which processes have seen which writes.
- It effectively means a dependency graph of which operation is dependent on which other operations must be constructed and maintained.

# Causal Consistency (5)

- Implementing causal consistency requires keeping track of which processes have seen which writes.
- It effectively means a dependency graph of which operation is dependent on which other operations must be constructed and maintained.
- This can be done using *vector timestamps*.

# Entry Consistency with Grouping Operations (1)

Necessary criteria for correct synchronization:

- Acquiring a lock can succeed only when all updates to its associated shared data have completed.

# Entry Consistency with Grouping Operations (1)

Necessary criteria for correct synchronization:

- Acquiring a lock can succeed only when all updates to its associated shared data have completed.
- Exclusive access to a lock can succeed only if no other process has exclusive or nonexclusive access to that lock.

# Entry Consistency with Grouping Operations (1)

Necessary criteria for correct synchronization:

- Acquiring a lock can succeed only when all updates to its associated shared data have completed.
- Exclusive access to a lock can succeed only if no other process has exclusive or nonexclusive access to that lock.
- Nonexclusive access to a lock is allowed only if any previous exclusive access has been completed, including updates on the lock's associated data.

This, in effect, is linearizing the usage of locks, adhering to sequential consistency.

# Entry Consistency with Grouping Operations (2)

P1: L(x) W(x)a L(y) W(y)b U(x) U(y)

P2: L(x) R(x)a R(y) NIL

P3: L(y) R(y)b

- A valid event sequence for entry consistency.

# Client-Centric Consistency Models

- Being able to handle concurrent operations on shared data while maintaining sequential consistency is fundamental to distributed systems. For performance reasons, this may be guaranteed only when processes use synchronization mechanisms.

# Client-Centric Consistency Models

- Being able to handle concurrent operations on shared data while maintaining sequential consistency is fundamental to distributed systems. For performance reasons, this may be guaranteed only when processes use synchronization mechanisms.
- A special, important class of distributed data stores have the following characterization:

# Client-Centric Consistency Models

- Being able to handle concurrent operations on shared data while maintaining sequential consistency is fundamental to distributed systems. For performance reasons, this may be guaranteed only when processes use synchronization mechanisms.
- A special, important class of distributed data stores have the following characterization:
  - Lack of simultaneous updates (or when such updates happen, they can be easily resolved)

# Client-Centric Consistency Models

- Being able to handle concurrent operations on shared data while maintaining sequential consistency is fundamental to distributed systems. For performance reasons, this may be guaranteed only when processes use synchronization mechanisms.
- A special, important class of distributed data stores have the following characterization:
  - Lack of simultaneous updates (or when such updates happen, they can be easily resolved)
  - Most operations involve reading data.

# Client-Centric Consistency Models

- Being able to handle concurrent operations on shared data while maintaining sequential consistency is fundamental to distributed systems. For performance reasons, this may be guaranteed only when processes use synchronization mechanisms.
- A special, important class of distributed data stores have the following characterization:
  - Lack of simultaneous updates (or when such updates happen, they can be easily resolved)
  - Most operations involve reading data.
- A very weak consistency model, called eventual consistency, is sufficient in such cases. This can be implemented relatively cheaply by client-centric consistency models.

# Eventual Consistency (1)

- Consider the following example scenarios:
  - A database where most operations are reads. Only one, or very few processes perform update operations. The question then is how fast updates should be made available to only-reading processes.

# Eventual Consistency (1)

- Consider the following example scenarios:
  - A database where most operations are reads. Only one, or very few processes perform update operations. The question then is how fast updates should be made available to only-reading processes.
  - DNS system has an authority for each domain so write-write conflicts never happen. The only conflicts we need to handle are read-write conflicts. A lazy propagation is sufficient in this case.

# Eventual Consistency (1)

- Consider the following example scenarios:
  - A database where most operations are reads. Only one, or very few processes perform update operations. The question then is how fast updates should be made available to only-reading processes.
  - DNS system has an authority for each domain so write-write conflicts never happen. The only conflicts we need to handle are read-write conflicts. A lazy propagation is sufficient in this case.
  - Web pages are updated by a single authority. There are normally no write-write conflicts to resolve. Clients such as Web browsers and proxies cache pages so they may return out-of-date Web pages. Users tolerate this inconsistency.

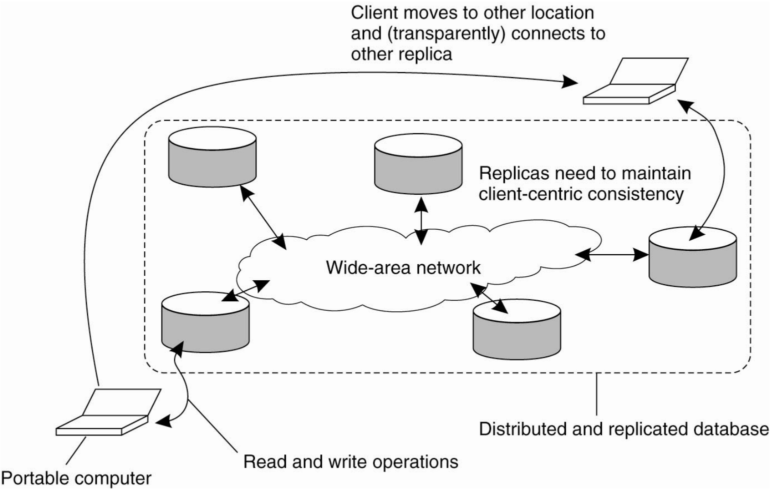
# Eventual Consistency (1)

- Consider the following example scenarios:
  - A database where most operations are reads. Only one, or very few processes perform update operations. The question then is how fast updates should be made available to only-reading processes.
  - DNS system has an authority for each domain so write-write conflicts never happen. The only conflicts we need to handle are read-write conflicts. A lazy propagation is sufficient in this case.
  - Web pages are updated by a single authority. There are normally no write-write conflicts to resolve. Clients such as Web browsers and proxies cache pages so they may return out-of-date Web pages. Users tolerate this inconsistency.
- If no updates take place, all replicas will gradually become consistent. This form of consistency is called eventual consistency.

# Eventual Consistency (1)

- Consider the following example scenarios:
  - A database where most operations are reads. Only one, or very few processes perform update operations. The question then is how fast updates should be made available to only-reading processes.
  - DNS system has an authority for each domain so write-write conflicts never happen. The only conflicts we need to handle are read-write conflicts. A lazy propagation is sufficient in this case.
  - Web pages are updated by a single authority. There are normally no write-write conflicts to resolve. Clients such as Web browsers and proxies cache pages so they may return out-of-date Web pages. Users tolerate this inconsistency.
- If no updates take place, all replicas will gradually become consistent. This form of consistency is called eventual consistency.
- Eventual consistency essentially requires only that updates are guaranteed to propagate to all replicas. Write-write conflicts are easy to solve when assuming that only a small group of processes can perform updates. Eventual consistency is therefore often cheap to implement.

# Eventual Consistency (2)




- The above problem can be alleviated by using *client-centric consistency*, which provides guarantees for a single client consistency in accessing the data store.

# Client-centric Consistency Model Notation

- Version $x_i$ of a data item is the result of a series of write operations that took place since initialization.
- We denote these operations by the write set $W(x_i)$
- By appending write operations to that series, we obtain another version $x_j$ and say that $x_j$ follows from $x_i$. We denote this as $W(x_i; x_j)$.
- If we do not know if $x_j$ follows from $x_i$, we use the notation $W(x_i \mid x_j)$.

# Monotonic Reads (1)

A data store is said to provide monotonic-read consistency if the following condition holds:

- If a process reads the value of a data item $x$, then any successive read operation on $x$ by that process will always return that same value or a more recent value.
- But no guarantees on concurrent access by different clients.

# Monotonic Reads (2)

| L1: | $W_1(x_1)$ | | $R_1(x_1)$ | |
|---|---|---|---|---|
| L2: | | $W_2(x_1;x_2)$ | | $R_1(x_2)$ |

- The read operations performed by a single process P at two different local copies of the same data store. A monotonic-read consistent data store.
- $L_1$ and $L_2$ are two local data stores.
- $W_1(x_1)$: Process $P_1$ produces version $x_1$ without knowing anything about other versions.
- $W_2(x_1; x_2)$: Process $P_2$ produces version $x_2$ that follows from $x_1$.

# Monotonic Reads (3)

| | | | | |
|---|---|---|---|---|
| L1: | $W_1(x_1)$ | | $R_1(x_1)$ | |
| L2: | | $W_2(x_1 \| x_2)$ | | $R_1(x_2)$ |

- The read operations performed by a single process P at two different local copies of the same data store. A data store that does not provide monotonic reads.
- $W_2(x_1 \mid x_2)$ denotes that process $P_2$ produced version $x_2$ concurrently to version $x_1$, a potential write-write conflict.

# Monotonic Writes (1)

In a monotonic-write consistent store, the following condition holds:

- 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 (2)

L1: $W_1(x_1)$

L2: $W_2(x_1;x_2)$ $W_1(x_2;x_3)$

- A monotonic-write consistent data store.

# Monotonic Writes (3)

| | | |
|---|---|---|
| L1: | $W_1(x_1)$ | |
| L2: | $W_2(x_1\|x_2)$ | $W_1(x_1\|x_3)$ |

- A data store that does not provide monotonic-write consistency.

# Monotonic Writes (4)

L1: $W_1(x_1)$

L2: $W_2(x_1|x_2)$ $W_1(x_2;x_3)$

- A data store that does not provide monotonic-write consistency as we have both $WS(x_1 \mid x_2)$ and $WS(x_1 \mid x_3)$.

# Monotonic Writes (5)

| | | | |
|---|---|---|---|
| L1: | $W_1(x_1)$ | | |
| L2: | | $W_2(x_1|x_2)$ | $W_1(x_1;x_3)$ |

- Consistent because $WS(x_1;x_3)$ although $x_1$ has apparently overwritten $x_2$.

# Read Your Writes (1)

A data store is said to provide read-your-writes consistency, if the following condition holds:

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

# Read Your Writes (2)

L1: $W_1(x_1)$

L2: $W_2(x_1;x_2)$ $R_1(x_2)$

- A data store that provides read-your-writes consistency.

# Read Your Writes (3)

L1: $W_1(x_1)$

L2: $W_2(x_1|x_2)$ $R_1(x_2)$

- A data store that does not follow *Read-Your-Writes* consistency model.

# Writes Follow Reads (1)

A data store is said to provide writes-follow-reads consistency, if the following holds:

- 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 a more recent value of $x$ that was read.

# Writes Follow Reads (2)

| | | | | |
|---|---|---|---|---|
| L1: | $W_1(x_1)$ | | $R_2(x_1)$ | |
| L2: | | $W_3(x_1;x_2)$ | | $W_2(x_2;x_3)$ |

- A *Writes-Follow-Reads* consistent data store.

# Writes Follow Reads (3)

| | | | | |
|---|---|---|---|---|
| L1: | $W_1(x_1)$ | | $R_2(x_1)$ | |
| L2: | | $W_3(x_1\|x_2)$ | | $W_2(x_1\|x_3)$ |

- A data store that does not follow *Writes-Follow-Reads* consistency model.

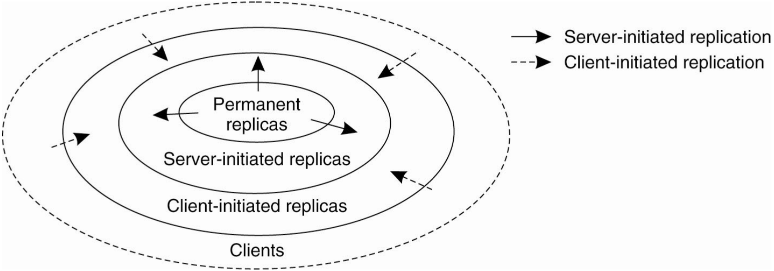
# Replica Management

- A key issue for any distributed system that supports replication is to decide where, when and by whom replicas should be placed, and subsequently the mechanisms to use for keeping the replicas consistent.

# Replica Management

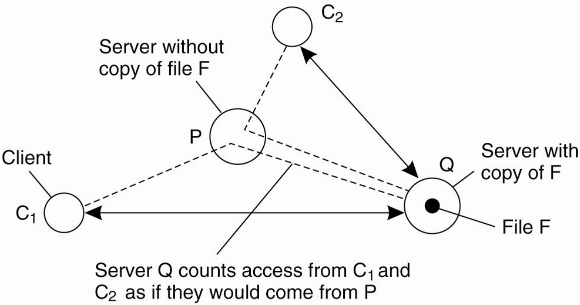
- A key issue for any distributed system that supports replication is to decide where, when and by whom replicas should be placed, and subsequently the mechanisms to use for keeping the replicas consistent. Two separate problems:
- *placing replica servers*: handled automatically by data centers
- *placing content*: permanent replicas, server-initiated, client-initiated

# Content Replication and Placement




- The logical organization of different kinds of copies of a data store into three concentric rings.

# Server-Initiated Replicas




- Counting access requests from different clients.
- Three possible actions: migrate, delete, replicate.

# Client-Initiated Replicas

- Same as client caches.
- Used to improve access times.
- Sharing a cache between clients may or may not improve the hit rate. E.g. Web data, file servers
- Shared caches can be at departmental or organization level

# Content Distribution (1)

- *Design issue*: What is actually to be propagated?

# Content Distribution (1)

- *Design issue*: What is actually to be propagated?
  - Propagate only a notification of an update (better for low read-to-write ratios)

# Content Distribution (1)

- *Design issue*: What is actually to be propagated?
  - Propagate only a notification of an update (better for low read-to-write ratios)
  - Transfer data from one copy to another (better for high read-to-write ratios)

# Content Distribution (1)

- *Design issue*: What is actually to be propagated?
  - Propagate only a notification of an update (better for low read-to-write ratios)
  - Transfer data from one copy to another (better for high read-to-write ratios)
  - Propagate the update operation to other copies (active replication)

# Content Distribution (1)

- *Design issue*: What is actually to be propagated?
  - Propagate only a notification of an update (better for low read-to-write ratios)
  - Transfer data from one copy to another (better for high read-to-write ratios)
  - Propagate the update operation to other copies (active replication)

# Content Distribution (2)

- Push Protocol: Updates are propagated to other replicas without being asked. This is useful for keeping a relatively higher degree of consistency.

# Content Distribution (2)

- Push Protocol: Updates are propagated to other replicas without being asked. This is useful for keeping a relatively higher degree of consistency.
- A push-based approach is efficient when the read-to-update ratio is relatively high.

# Content Distribution (2)

- Push Protocol: Updates are propagated to other replicas without being asked. This is useful for keeping a relatively higher degree of consistency.
- A push-based approach is efficient when the read-to-update ratio is relatively high.
- Pull Protocol: A server or a client requests another server to send it any updates it has at the moment. A pull-based approach is efficient when the read-to-update ratio is relatively low.

# Content Distribution (2)

- Push Protocol: Updates are propagated to other replicas without being asked. This is useful for keeping a relatively higher degree of consistency.
- A push-based approach is efficient when the read-to-update ratio is relatively high.
- Pull Protocol: A server or a client requests another server to send it any updates it has at the moment. A pull-based approach is efficient when the read-to-update ratio is relatively low.
- A comparison between push-based and pull-based protocols in the case of multiple-client, single-server systems.

| Issue | Push-based | Pull-based |
|---|---|---|
| State at 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 |

# Content Distribution (3)

- A lease is a promise by the server that it will push updates to the client for a specified time.

# Content Distribution (3)

- A lease is a promise by the server that it will push updates to the client for a specified time.
- When a lease expires, the client is forced to poll the server for updates and pull in the modified data if necessary. We can use leases to dynamically switch between push and pull.

# Content Distribution (3)

- A lease is a promise by the server that it will push updates to the client for a specified time.
- When a lease expires, the client is forced to poll the server for updates and pull in the modified data if necessary. We can use leases to dynamically switch between push and pull.
- Types of leases:
  - Age-based leases. Grant long lasting leases to data that is expected to remain unmodified.

# Content Distribution (3)

- A lease is a promise by the server that it will push updates to the client for a specified time.
- When a lease expires, the client is forced to poll the server for updates and pull in the modified data if necessary. We can use leases to dynamically switch between push and pull.
- Types of leases:
  - Age-based leases. Grant long lasting leases to data that is expected to remain unmodified.
  - Renewal-frequency based leases. Give longer leases to clients where its data is popular.

# Content Distribution (3)

- A lease is a promise by the server that it will push updates to the client for a specified time.
- When a lease expires, the client is forced to poll the server for updates and pull in the modified data if necessary. We can use leases to dynamically switch between push and pull.
- Types of leases:
  - Age-based leases. Grant long lasting leases to data that is expected to remain unmodified.
  - Renewal-frequency based leases. Give longer leases to clients where its data is popular.
  - State-space overhead based leases. Start lowering expiration times as space runs low.

# Content Distribution (3)

- A lease is a promise by the server that it will push updates to the client for a specified time.
- When a lease expires, the client is forced to poll the server for updates and pull in the modified data if necessary. We can use leases to dynamically switch between push and pull.
- Types of leases:
  - Age-based leases. Grant long lasting leases to data that is expected to remain unmodified.
  - Renewal-frequency based leases. Give longer leases to clients where its data is popular.
  - State-space overhead based leases. Start lowering expiration times as space runs low.
- Using multicasting can be much more efficient than unicasting for the updates.

# Implementations: Consistency Protocols

- Bounding numerical deviations

# Implementations: Consistency Protocols

- Bounding numerical deviations
  - Implement using a push model based on views on values of variables.

# Implementations: Consistency Protocols

- Bounding numerical deviations
  - Implement using a push model based on views on values of variables.
- Bounding staleness deviations

# Implementations: Consistency Protocols

- Bounding numerical deviations
  - Implement using a push model based on views on values of variables.
- Bounding staleness deviations
  - Use Vector Clocks, where on server $S_k$, $RVC_k[i] = t_i$ implies that $S_k$ has seen all writes that have been submitted to $S_i$ up to time $t_i$.

# Implementations: Consistency Protocols

- Bounding numerical deviations
  - Implement using a push model based on views on values of variables.
- Bounding staleness deviations
  - Use Vector Clocks, where on server $S_k$, $RVC_k[i] = t_i$ implies that $S_k$ has seen all writes that have been submitted to $S_i$ up to time $t_i$.
  - Whenever a server notices that $t_k - RVC_k[i]$ is about to exceed a specified limit, it simply starts pulling in writes from $S_i$ with a timestamp later than $RVC_k[i]$.

# Implementations: Consistency Protocols

- Bounding numerical deviations
  - Implement using a push model based on views on values of variables.
- Bounding staleness deviations
  - Use Vector Clocks, where on server $S_k$, $RVC_k[i] = t_i$ implies that $S_k$ has seen all writes that have been submitted to $S_i$ up to time $t_i$.
  - Whenever a server notices that $t_k - RVC_k[i]$ is about to exceed a specified limit, it simply starts pulling in writes from $S_i$ with a timestamp later than $RVC_k[i]$.
- Bounding ordering deviations

# Implementations: Consistency Protocols

- Bounding numerical deviations
  - Implement using a push model based on views on values of variables.
- Bounding staleness deviations
  - Use Vector Clocks, where on server $S_k$, $RVC_k[i] = t_i$ implies that $S_k$ has seen all writes that have been submitted to $S_i$ up to time $t_i$.
  - Whenever a server notices that $t_k - RVC_k[i]$ is about to exceed a specified limit, it simply starts pulling in writes from $S_i$ with a timestamp later than $RVC_k[i]$.
- Bounding ordering deviations
  - Can be bounded by specifying the maximal length of the queue of tentative writes.
  - When this is exceeded, the server no longer accepts writes, but will instead attempt to commit tentative writes by negotiating with other servers in which order its writes should be executed.

# Implementations: Primary-based Protocols

- Implementations tend to prefer simpler consistency models.

# Implementations: Primary-based Protocols

- Implementations tend to prefer simpler consistency models.
- Primary-based protocols are used for implementing sequential consistency.

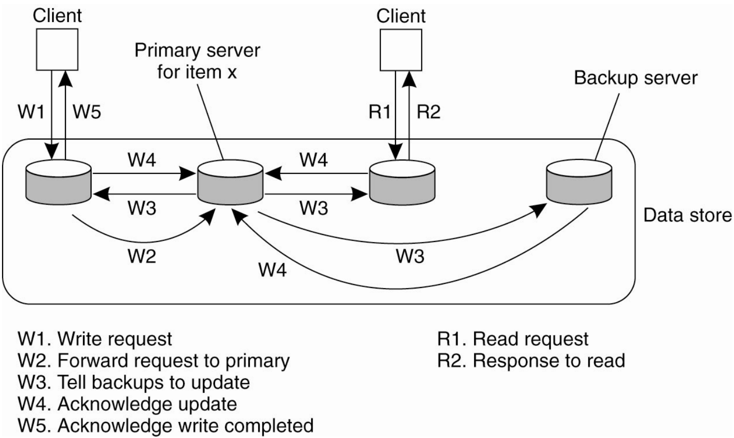
# Implementations: Primary-based Protocols

- Implementations tend to prefer simpler consistency models.
- Primary-based protocols are used for implementing sequential consistency.
- In a primary-based protocol, each data item $x$ in the data store has an associated primary, which is responsible for write operations on $x$

# Implementations: Primary-based Protocols

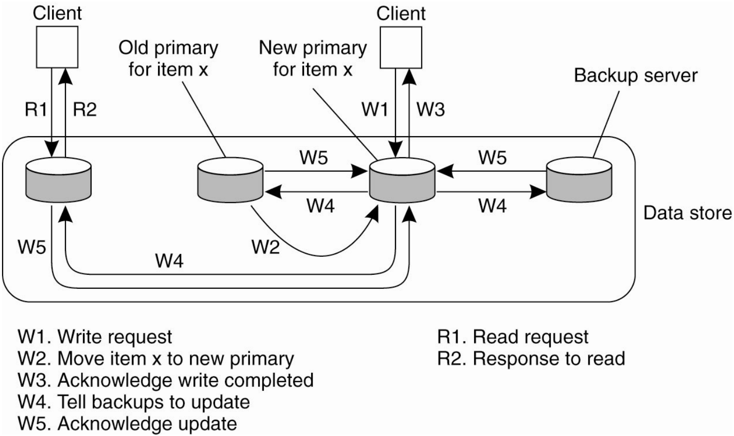
- Implementations tend to prefer simpler consistency models.
- Primary-based protocols are used for implementing sequential consistency.
- In a primary-based protocol, each data item $x$ in the data store has an associated primary, which is responsible for write operations on $x$
- Primary can be *fixed at a remote server* or write operations can be carried out *locally after moving the primary to the process* where the write operation was initiated.

# Remote Write Protocol




- Also known as a primary-backup protocol. Implements sequential consistency.
- Non-blocking version: the primary acknowledges after updating its copy and informs backup servers afterwards

# Local Write Protocol




- Primary-backup protocol in which the primary migrates to the process wanting to perform an update. Updates have to be propagated back to other replicas.

# Examples of Local Write Primary-Backup Protocols

- *Mobile devices*
  - Before disconnecting, the mobile device becomes the primary server for each data item it expects to update.

# Examples of Local Write Primary-Backup Protocols

- *Mobile devices*
  - Before disconnecting, the mobile device becomes the primary server for each data item it expects to update.
  - While disconnected, update operations happen locally on the device while other processes can still perform read operations (but without updates).

# Examples of Local Write Primary-Backup Protocols

- *Mobile devices*
  - Before disconnecting, the mobile device becomes the primary server for each data item it expects to update.
  - While disconnected, update operations happen locally on the device while other processes can still perform read operations (but without updates).
  - Later, when connecting again, the updates are propagated from the primary to backups, bringing the data store in a consistent state again.

# Examples of Local Write Primary-Backup Protocols

- *Mobile devices*
  - Before disconnecting, the mobile device becomes the primary server for each data item it expects to update.
  - While disconnected, update operations happen locally on the device while other processes can still perform read operations (but without updates).
  - Later, when connecting again, the updates are propagated from the primary to backups, bringing the data store in a consistent state again.
- *Distributed File Systems*
  - Normally, there may be a fixed central server through which all write operations take place (as in remote write primary backup).

# Examples of Local Write Primary-Backup Protocols

- *Mobile devices*
  - Before disconnecting, the mobile device becomes the primary server for each data item it expects to update.
  - While disconnected, update operations happen locally on the device while other processes can still perform read operations (but without updates).
  - Later, when connecting again, the updates are propagated from the primary to backups, bringing the data store in a consistent state again.
- *Distributed File Systems*
  - Normally, there may be a fixed central server through which all write operations take place (as in remote write primary backup).
  - However, the server temporarily allows one of the replicas to perform a series of local updates to speed up performance.

# Examples of Local Write Primary-Backup Protocols

- *Mobile devices*
  - Before disconnecting, the mobile device becomes the primary server for each data item it expects to update.
  - While disconnected, update operations happen locally on the device while other processes can still perform read operations (but without updates).
  - Later, when connecting again, the updates are propagated from the primary to backups, bringing the data store in a consistent state again.
- *Distributed File Systems*
  - Normally, there may be a fixed central server through which all write operations take place (as in remote write primary backup).
  - However, the server temporarily allows one of the replicas to perform a series of local updates to speed up performance.
  - When the replica server is done, the updates are propagated to the central server, from where they can be distributed to other replica servers.

# Replicated-Write Protocols

- Write operations can be carried out at multiple replicas instead of just one.

# Replicated-Write Protocols

- Write operations can be carried out at multiple replicas instead of just one.
- Active replication: the write operation is forwarded to all replicas. It requires operations to be carried out in the same order everywhere. So it requires totally ordered multicasts using either Lamport timestamps or a central coordinator.

# Replicated-Write Protocols

- Write operations can be carried out at multiple replicas instead of just one.
- Active replication: the write operation is forwarded to all replicas. It requires operations to be carried out in the same order everywhere. So it requires totally ordered multicasts using either Lamport timestamps or a central coordinator.
- Quorum-based protocol: the write operation is done by majority voting.

# Quorum-based Replicated-Write Protocol (1)

## Quorum-based protocol

- To read a file with $N$ replicas, a client needs to assemble a read quorum, an arbitrary collection of $N_R$ servers. Similarly, to write to a file, a write quorum of at least $N_W$ servers. The values of the quorums are subject to the following rules:
  - $N_R + N_W > N$
  - $N_W > N/2$

# Quorum-based Replicated-Write Protocol (1)

Quorum-based protocol

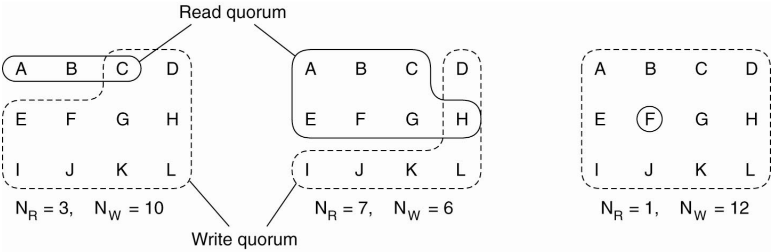
- To read a file with $N$ replicas, a client needs to assemble a read quorum, an arbitrary collection of $N_R$ servers. Similarly, to write to a file, a write quorum of at least $N_W$ servers. The values of the quorums are subject to the following rules:
  - $N_R + N_W > N$
  - $N_W > N/2$
- Only one writer at a time can achieve write quorum.

# Quorum-based Replicated-Write Protocol (1)

Quorum-based protocol

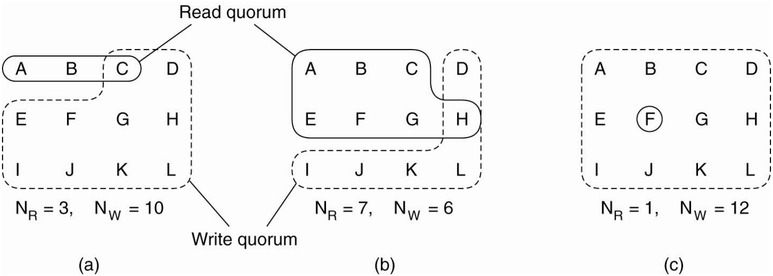
- To read a file with $N$ replicas, a client needs to assemble a read quorum, an arbitrary collection of $N_R$ servers. Similarly, to write to a file, a write quorum of at least $N_W$ servers. The values of the quorums are subject to the following rules:
  - $N_R + N_W > N$
  - $N_W > N/2$
- Only one writer at a time can achieve write quorum.
- Every reader sees at least one copy of the most recent read (takes one with most recent version number or logical timestamp)

# Quorum-based Replicated-Write Protocols (2)



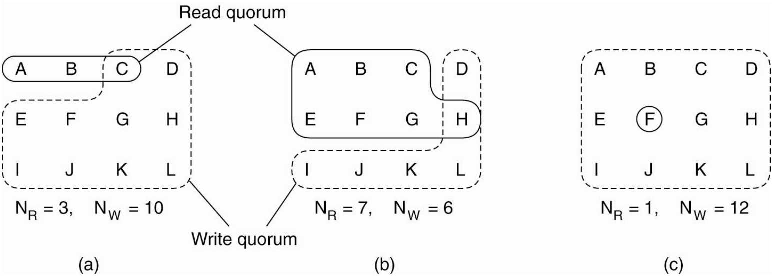
(a) (b) (c)

# Quorum-based Replicated-Write Protocols (2)




- Quorum-based Protocol. Three examples of the voting algorithm.
  (a) A correct choice of read and write set.
  (b) A choice that may lead to write-write conflicts.
  (c) A correct choice, known as ROWA (read one, write all).

# Quorum-based Replicated-Write Protocols (2)




- Quorum-based Protocol. Three examples of the voting algorithm.
  - (a) A correct choice of read and write set.
  - (b) A choice that may lead to write-write conflicts.
  - (c) A correct choice, known as ROWA (read one, write all).
- **In-class Exercise**. A file is replicated on 10 servers. List all combinations of read and write quorums that are permitted by the voting algorithm.

# Quorum-based Replicated-Write Protocols (3)

- ROWA: $R = 1$, $W = N$
  Fast reads, slow writes
- RAWO: $R = N$, $W = 1$
  Fast writes, slow reads
- Majority: $R = W = N/2 + 1$
  Both moderately slow, but extremely high availability
- Weighted voting:
  Give more votes to "better" replicas

# Implementing Client-Centric Consistency (1)

- Each write operation has a globally unique identifier. Such an identifier can be assigned by the server where the write was submitted.

# Implementing Client-Centric Consistency (1)

- Each write operation has a globally unique identifier. Such an identifier can be assigned by the server where the write was submitted.
- Each client tracks identifiers for two sets of writes:
  - read set: consists of writes relevant for the read operations performed by the client.
  - write set: consists of the writes performed by the client.

# Implementing Client-Centric Consistency (2)

- *Implementing Monotonic Reads*: When a client performs a read operation at a server, that server is handed the client's read set to check whether all the identified writes have taken place locally. If not, it can contact other servers to be brought up to date. Or it can forward the request to another server where the write operations have already taken place.

# Implementing Client-Centric Consistency (2)

- *Implementing Monotonic Reads*: When a client performs a read operation at a server, that server is handed the client's read set to check whether all the identified writes have taken place locally. If not, it can contact other servers to be brought up to date. Or it can forward the request to another server where the write operations have already taken place.
- After the read operation is performed, the write operations that are relevant to the read operation are added to the client's read set.

# Implementing Client-Centric Consistency (2)

- *Implementing Monotonic Reads*: When a client performs a read operation at a server, that server is handed the client's read set to check whether all the identified writes have taken place locally. If not, it can contact other servers to be brought up to date. Or it can forward the request to another server where the write operations have already taken place.
- After the read operation is performed, the write operations that are relevant to the read operation are added to the client's read set.
- How to determine exactly where the write operations in the client's read set have taken place?

# Implementing Client-Centric Consistency (2)

- *Implementing Monotonic Reads*: When a client performs a read operation at a server, that server is handed the client's read set to check whether all the identified writes have taken place locally. If not, it can contact other servers to be brought up to date. Or it can forward the request to another server where the write operations have already taken place.
- After the read operation is performed, the write operations that are relevant to the read operation are added to the client's read set.
- How to determine exactly where the write operations in the client's read set have taken place? The write identifier can contain the server's identifier. Servers can be required to log their writes so they can replayed at another server. The client's can generate a globally unique identifier that is included in the write identifier.

# Implementing Client-Centric Consistency (3)

- *Implementing Monotonic Writes*: Whenever a client initiates a new write operation at a server, the server is handed over the client's write set. It then ensures that the identified write operations are performed first and in the correct order. After performing the new write operation, that write operation's identifier is added to the write set.

# Implementing Client-Centric Consistency (3)

- *Implementing Monotonic Writes*: Whenever a client initiates a new write operation at a server, the server is handed over the client's write set. It then ensures that the identified write operations are performed first and in the correct order. After performing the new write operation, that write operation's identifier is added to the write set.
- *Implementing Read-Your-Writes*: Requires that the server where the read operation is performed has seen all the writes in the client's write set. They can be fetched from other servers or the client can search for the appropriate server to improve performance.

# Implementing Client-Centric Consistency (3)

- *Implementing Monotonic Writes*: Whenever a client initiates a new write operation at a server, the server is handed over the client's write set. It then ensures that the identified write operations are performed first and in the correct order. After performing the new write operation, that write operation's identifier is added to the write set.
- *Implementing Read-Your-Writes*: Requires that the server where the read operation is performed has seen all the writes in the client's write set. They can be fetched from other servers or the client can search for the appropriate server to improve performance.
- *Implementing Writes-Follow-Reads*: Bring the selected server up to date with the write operations in the client's read set, and then later adding the identifier for the write set to the write set, along with the identifiers of the read set (which have now become relevant for the write operation just performed).

# Implementing Client-Centric Consistency (4)

- Problems with the implementation: The read set and write set associated with each client can become very big.

# Implementing Client-Centric Consistency (4)

- Problems with the implementation: The read set and write set associated with each client can become very big.
- Improving efficiency: keep track of the sets only for the length of a session. This isn't sufficient. Why?

# Implementing Client-Centric Consistency (4)

- Problems with the implementation: The read set and write set associated with each client can become very big.
- Improving efficiency: keep track of the sets only for the length of a session. This isn't sufficient. Why?
- The main problem is the representation of the read and write sets. It can be represented more efficiently by means of vector timestamps.

# Representing Read/Write Sets Using Vector Timestamps (1)

- Whenever a server accepts a new write operation $W$, it assigns it a globally unique identifier along with a timestamp $\texttt{ts}(W)$. That server is identified as the $\texttt{origin}(W)$. Subsequent writes to that server get higher timestamps.

# Representing Read/Write Sets Using Vector Timestamps (1)

- Whenever a server accepts a new write operation $W$, it assigns it a globally unique identifier along with a timestamp $\texttt{ts}(W)$. That server is identified as the $\texttt{origin}(W)$. Subsequent writes to that server get higher timestamps.
- Each server $S_i$ maintains a vector timestamp $\texttt{WVC}_i$, where $\texttt{WVC}_i[j]$ is equal to the timestamp of the most recent write operation originating from $S_j$ that has been processed by $S_i$.

# Representing Read/Write Sets Using Vector Timestamps (1)

- Whenever a server accepts a new write operation $W$, it assigns it a globally unique identifier along with a timestamp $\texttt{ts}(W)$. That server is identified as the $\texttt{origin}(W)$. Subsequent writes to that server get higher timestamps.
- Each server $S_i$ maintains a vector timestamp $\texttt{WVC}_i$, where $\texttt{WVC}_i[j]$ is equal to the timestamp of the most recent write operation originating from $S_j$ that has been processed by $S_i$. Assume that the writes from $S_j$ are processed in the order they were submitted.

# Representing Read/Write Sets Using Vector Timestamps (1)

- Whenever a server accepts a new write operation $W$, it assigns it a globally unique identifier along with a timestamp $\texttt{ts}(W)$. That server is identified as the $\texttt{origin}(W)$. Subsequent writes to that server get higher timestamps.
- Each server $S_i$ maintains a vector timestamp $\texttt{WVC}_i$, where $\texttt{WVC}_i[j]$ is equal to the timestamp of the most recent write operation originating from $S_j$ that has been processed by $S_i$. Assume that the writes from $S_j$ are processed in the order they were submitted.
- Whenever a client issues a request to read or write operation at a specific server, that server returns its current timestamp along with the results of the operation.

# Representing Read/Write Sets Using Vector Timestamps (2)

- Read and write sets are represented by vector timestamps. For each session $A$, we construct a vector timestamp $\texttt{SVC}_A$ with $\texttt{SVC}_A[i]$ set equal to the maximum timestamp of all write operations in $A$ that originate from server $S_i$.

$$\texttt{SVC}_A[j] = \max\{\texttt{ts}(W) \mid W \in A \,\&\, \text{origin}(W) = S_j\}$$

# Representing Read/Write Sets Using Vector Timestamps (2)

- Read and write sets are represented by vector timestamps. For each session $A$, we construct a vector timestamp $\texttt{SVC}_A$ with $\texttt{SVC}_A[i]$ set equal to the maximum timestamp of all write operations in $A$ that originate from server $S_i$.

$$\texttt{SVC}_A[j] = \max\{\texttt{ts}(W) \mid W \in A \,\&\, \text{origin}(W) = S_j\}$$

- The timestamp of a session always represents the latest write operations that have been seen by the applications that are executing as part of that session.

# Representing Read/Write Sets Using Vector Timestamps (2)

- Read and write sets are represented by vector timestamps. For each session $A$, we construct a vector timestamp $\mathtt{SVC}_A$ with $\mathtt{SVC}_A[i]$ set equal to the maximum timestamp of all write operations in $A$ that originate from server $S_i$.

$$\mathtt{SVC}_A[j] = \max\{\mathtt{ts}(W) \mid W \in A \,\&\, \mathrm{origin}(W) = S_j\}$$

- The timestamp of a session always represents the latest write operations that have been seen by the applications that are executing as part of that session.
- The compactness is obtained by representing all observed writes originating from the same server through a single timestamp.

# Representing Read/Write Sets Using Vector Timestamps (3)

- Suppose a client, as part of session $A$, logs in at server $S_i$ and passes $\texttt{SVC}_A$ to $S_i$.

# Representing Read/Write Sets Using Vector Timestamps (3)

- Suppose a client, as part of session $A$, logs in at server $S_i$ and passes $\texttt{SVC}_A$ to $S_i$.
- Assume that $\texttt{SVC}_A[j] > \texttt{WVC}_i[j]$. That is, $S_i$ has not yet seen all the writes originating from $S_j$ that the client has seen.

# Representing Read/Write Sets Using Vector Timestamps (3)

- Suppose a client, as part of session $A$, logs in at server $S_i$ and passes $\texttt{SVC}_A$ to $S_i$.
- Assume that $\texttt{SVC}_A[j] > \texttt{WVC}_i[j]$. That is, $S_i$ has not yet seen all the writes originating from $S_j$ that the client has seen.
- Depending upon the required consistency, the server may have to fetch these writes before being able to consistently report back to the client.

# Representing Read/Write Sets Using Vector Timestamps (3)

- Suppose a client, as part of session $A$, logs in at server $S_i$ and passes $\texttt{SVC}_A$ to $S_i$.
- Assume that $\texttt{SVC}_A[j] > \texttt{WVC}_i[j]$. That is, $S_i$ has not yet seen all the writes originating from $S_j$ that the client has seen.
- Depending upon the required consistency, the server may have to fetch these writes before being able to consistently report back to the client.
- Once the operation is performed, the server $S_i$ will return its current timestamp $\texttt{WVC}_i$. At that point, $\texttt{SVC}_A$ is adjusted to:

$$\texttt{SVC}_A[j] = \max\{\texttt{SVC}_A[j], \texttt{WVC}_i[j]\}$$

# Exercises (1)

- **Problem 1**. Access to shared Java objects can be serialized by declaring its methods as being synchronized. Is this enough to guarantee serialization when such an object is replicated?

# Exercises (1)

- **Problem 1**. Access to shared Java objects can be serialized by declaring its methods as being synchronized. Is this enough to guarantee serialization when such an object is replicated?
- **Problem 2**. Explain in your own words what the main reason is for actually considering weak consistency models.

# Exercises (1)

- **Problem 1**. Access to shared Java objects can be serialized by declaring its methods as being synchronized. Is this enough to guarantee serialization when such an object is replicated?
- **Problem 2**. Explain in your own words what the main reason is for actually considering weak consistency models.
- **Problem 3**. It is often argued that weak consistency models impose an extra burden for programmers. To what extent is this statement actually true?

# Exercises (1)

- **Problem 1**. Access to shared Java objects can be serialized by declaring its methods as being synchronized. Is this enough to guarantee serialization when such an object is replicated?
- **Problem 2**. Explain in your own words what the main reason is for actually considering weak consistency models.
- **Problem 3**. It is often argued that weak consistency models impose an extra burden for programmers. To what extent is this statement actually true?
- **Problem 4**. What kind of consistency model would you use to implement an electronic stock market? Explain your answer. (*Hint:* Sequential versus Causal)

# Exercises (1)

- **Problem 1**. Access to shared Java objects can be serialized by declaring its methods as being synchronized. Is this enough to guarantee serialization when such an object is replicated?
- **Problem 2**. Explain in your own words what the main reason is for actually considering weak consistency models.
- **Problem 3**. It is often argued that weak consistency models impose an extra burden for programmers. To what extent is this statement actually true?
- **Problem 4**. What kind of consistency model would you use to implement an electronic stock market? Explain your answer. (*Hint:* Sequential versus Causal)
- **Problem 5**. Consider a personal mailbox for a mobile user, implemented as part of a wide-area distributed database. What kind of client-centric consistency model would be the most appropriate?

# Exercises (2)

- **Problem 6**. When using a lease, is it necessary that the clocks of a client and server, respectively, are tightly synchronized?

# Exercises (2)

- **Problem 6**. When using a lease, is it necessary that the clocks of a client and server, respectively, are tightly synchronized?
- **Problem 7**. Consider a non-blocking primary-backup protocol used to guarantee sequential consistency in a distributed data store. Does such a store always provide read-your-writes consistency?

# Exercises (2)

- **Problem 6**. When using a lease, is it necessary that the clocks of a client and server, respectively, are tightly synchronized?
- **Problem 7**. Consider a non-blocking primary-backup protocol used to guarantee sequential consistency in a distributed data store. Does such a store always provide read-your-writes consistency?
- **Problem 8**. For active replication to work in general, it is necessary that all operations be carried out in the same order at each replica. Is this ordering always necessary? (*Hint*: Consider read and idempotent write operations)

## Exercises (2)

- **Problem 6**. When using a lease, is it necessary that the clocks of a client and server, respectively, are tightly synchronized?
- **Problem 7**. Consider a non-blocking primary-backup protocol used to guarantee sequential consistency in a distributed data store. Does such a store always provide read-your-writes consistency?
- **Problem 8**. For active replication to work in general, it is necessary that all operations be carried out in the same order at each replica. Is this ordering always necessary? (*Hint*: Consider read and idempotent write operations)
- **Problem 9**. A file is replicated on 10 servers. List all combinations of read and write quorums that are permitted by the voting algorithm.

# Summary of Consistency Models (1)

**Data-Centric Consistency Models**

- **Sequential Consistency**: 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 ***some sequential order*** and the operations of each individual process appear in this sequence in the order specified by its program.
- **Causal Consistency**: Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes by different processes may be seen in a different order on different machines. Writes by the same process are also considered to be causally related.

# Summary of Consistency Models (2)

**Eventual consistency** *essentially requires only that updates are guaranteed to propagate to all replicas*. **Client-Centric Consistency Models** that are used to implement eventual consistency.

- **Monotonic Reads**: If a process reads the value of a data item $x$, then any successive read operation on $x$ by that process will always return that same value or a more recent value.
- **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.
- **Read Your Writes**: The effect of a write operation by a process on data item $x$ will always be seen by a successive read operation on $x$ by the same process.
- **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 a more recent value of $x$ that was read.

# References

- Eventually Consistent by Werner Vogels (CTO, Amazon)
- How eventual is eventual consistency? posted on Basho Blog
- 10 Lessons from 10 Years of Amazon Web Services by Werner Vogels (CTO, Amazon)