Consistency Models

- Background – Replication Motivation
- Data-Centric Consistency Models
  - Continuous Consistency
  - Consistent Ordering of Operations
- Client-Centric Consistency Models
  - Eventual Consistency
  - Monotonic Reads
  - Monotonic Writes
  - Read Your Writes
  - Writes Follow Reads
Replication

- Replication Benefits
  - Replication enhances tolerance to failure
  - Performance can be improved in different ways
    - Placing replicas in the proximity of the process using them
    - Replicating servers and subsequently dividing the work among them

- Consistency Requirements – Replication may lead to consistency problems
  - Modifications need to be carried out on all copies
  - When and how modifications are carried out is the price of replications

Replication and Scalability

- Replication and caching for performance are widely used as scaling techniques
  - Adding replicas improves scalability, but incurs the overhead of keeping the replicas up-to-date

- How to efficiently synchronize all of the replicas created to solve the scalability issue?
  - A challenging problem!
Performance and Scalability

- Main issue – To keep replicas consistent, we generally need to ensure that all conflicting operations are done in the same order, across all servers

- **Conflicting Operations** – Concurrent Transactions
  - **Read–Write Conflict**: a read operation and a write operation act concurrently
  - **Write–Write Conflict**: two concurrent write operations

- Guaranteeing global ordering on conflicting operations may be a costly operation, with impact on scalability

- **Potential Solution** – Weaken consistency requirements to avoid global synchronization, when possible

Weakening Consistency Requirements

- **What does it mean to “weaken consistency requirements”?**
  - Relax the requirement that “updates need to be executed as atomic operations”
  - Do not require global synchronizations
  - Replicas may not always be the same everywhere

- **To what extent can consistency be weakened?**
  - Depends highly on the access and update patterns of the replicas
  - Depends on the replicated data use patterns
    - Application requirements and behavior
Data-Centric Consistency Models

A data store is physically distributed and replicated across multiple machines.

- A data-store can be read from or written to by any process
- A local copy of the data-store (replica) can support “fast reads”
- A write to a local replica needs to be propagated to *all* remote replicas

Various consistency models can be used to ensure “correct” operation, as agreed upon by access rules.

What is a Consistency Model?

A “consistency model” is a *contract* between a distributed data-store and its processes.

- If the processes agree to the rules, the data-store will perform correctly and as advertised.
Data-Centric Consistency Models – Strong and Weak Models

- **Strong consistency models**: Operations on shared data are synchronized – Do not require synchronization operations
  - **Strict consistency** – Related to absolute global time
  - **Sequential Consistency** – Also known as serializability
  - **Linearizability** – To achieve atomicity
  - **Causal Consistency** – To maintain only causal relations
  - **FIFO Consistency** – To maintain only individual ordering

- **Weak consistency models**: Synchronization occurs only when shared data is locked and unlocked – Rely on synchronization operations
  - **Weak Consistency**
  - **Release Consistency**
  - **Entry Consistency**

- **Note** – Weaker the consistency model are more scalable

Consistency Model Diagram Notation

- **\( W_i(x)a \)** – A write operation performed by process ‘i’ on item ‘x’ with a value of ‘a’.
  - Process ‘\( P_i \)’ sets ‘x’ to ‘a’.
- **\( R_i(x)b \)** – A read operation performed by process ‘i’ on item ‘x’ producing the value ‘b’.
  - Process ‘\( P_i \)’ read operation on ‘x’ returns ‘b’.
- In all diagrams, time moves from left to right
**Strict Consistency Diagrams**

- **With Strict Consistency**, all writes are visible instantaneously to all processes.

- Any read to a shared data item X returns the value stored by the most recent write operation on X.

- Strict Consistency relies on absolute global time, making it difficult to implement in a distributed system.

---

**Sequential Consistency – Definition**

Sequential consistency – The result of any execution is the same as if the read and write operations by all processes were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program.

- Any valid interleaving of read and write operations is OK, but all processes must see the same interleaving.
  - The events observed by each process must globally occur in the same order, or it is not sequentially consistent.
    - It doesn't actually matter if the events don't really agree with clock time, as long as they are consistent.
Sequential Consistency – Interleaving

<table>
<thead>
<tr>
<th>Sequentially Consistent Data Store</th>
<th>Sequentially Not Consistent Data Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: W(x)a</td>
<td>P1: W(x)a</td>
</tr>
<tr>
<td>P2: W(x)b</td>
<td>P2: W(x)b</td>
</tr>
<tr>
<td>P3: R(x)b R(x)a</td>
<td>P3: R(x)b R(x)a</td>
</tr>
<tr>
<td>P4: R(x)b R(x)a</td>
<td>P4: R(x)u R(x)b</td>
</tr>
</tbody>
</table>

P3 and P4 do see the same interleaving of writes

Sequential Consistency – Example

- Three processes, executing concurrently.
- The 6 statements shown can be ordered (and executed) in 6! = 720 possible ways, with most of the orderings are invalid.
- Analysis shows that 90 possible valid execution sequences exist.

```c
x = 1;
print(y, z);
print(x, z); // Invalid because it violates program
y = 1;       // order of process P2.
z = 1;
print(x, y);
```
Sequential Consistency – Interleaved Execution Sequence

- The *signature* is the output of P₁, P₂, and P₃, in that order.
- Signature can be used to determine whether a given execution sequence is valid.

```
x = 1;
print (y, z);
y = 1;
print (x, z);
z = 1;
print (x, y);
```

Prints: 001011
Sign: 001011
(a)

```
x = 1;
y = 1;
print (x, z);
print (y, z);
z = 1;
print (x, y);
```

Prints: 101011
Sign: 101011
(b)

```
y = 1;
z = 1;
print (x, y);
print (x, z);
x = 1;
print (y, z);
```

Prints: 010111
Sign: 110101
(c)

```
y = 1;
x = 1;
z = 1;
print (x, z);
print (y, z);
print (x, y);
```

Prints: 111111
Sign: 111111
(d)

---

All four interleaved execution sequences are valid

---

Linearizability

- In sequential consistency, absolute time is somewhat irrelevant – The order of events is most important.
- Linearizability – Sequential ⊕ Operations are ordered according to a global time
- A data store is said to be **linearizable** when each operation is timestamped and the following condition holds:
  - “The result of any execution is the same as if the 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.
  - In addition, if \( ts_{\text{Op}}(x) < ts_{\text{Op}}(y) \), then \( \text{Op}_1(x) \) should precede \( \text{Op}_2(y) \) in this sequence.”
**Sequential consistency vs. Linearizability**

- Linearizability is weaker than strict consistency, but stronger than sequential consistency.
- Linearizability has proven useful for reasoning about program correctness but has not typically been used otherwise.
- Sequential consistency is implementable and widely used but has poor performance.
- To get around performance problems, weaker models that have better performance have been developed.

**Causal Consistency**

- **Necessary condition** – 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 on different machines.
  - If event A is a direct or indirect result of another prior event B, then all processes should observe event A before observing event B.
    
    ```
    A = A + 1; // First two events are causally related,
    B = A * 5; // because B reads A after A was written.
    C = C * 3; // This is a concurrent statement.
    ```
Causal Consistency - Sequence

- It is assumed that $W_2(x)b$ and $W_1(x)c$ are concurrent
- Not strictly consistent because $P_3$ and $P_4$ "R(x)" operations don’t return most recent write on x, in all cases.
- Not sequentially consistent because $P_3$ and $P_4$ don’t read the same values in the same order.

- The sequence is allowed with a causally-consistent store

Causal Consistency

- (a) A violation of a causally-consistent store
  - $W_1(x)a$ and $W_2(x)b$ are causally dependent – all processes must them in the same order
- (b) A correct sequence of events in a causally-consistent store
  - $W_1(x)a$ and $W_2(x)b$ are concurrent.
FIFO Consistency

- Necessary Condition – Writes performed by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.

<table>
<thead>
<tr>
<th>P1: W(x)a</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2: R(x)a W(x)b W(x)c</td>
</tr>
<tr>
<td>P3: R(x)b R(x)a R(x)c</td>
</tr>
<tr>
<td>P4: R(x)a R(x)b R(x)c</td>
</tr>
</tbody>
</table>

- A valid sequence of events of FIFO consistency.
  - P2’s writes are seen in the correct order.
  - FIFO consistency is easy to implement.

Weak Consistency – Motivation

- Not all applications need to see all writes, let alone seeing them in the same order
- This leads to “Weak Consistency”
  - It is primarily designed to work with distributed critical regions
  - This model introduces the notion of a “synchronization variable”, which is used to update all copies of the data-store
Weak Consistency - Criteria

- The following criteria must be met
  - Accesses to synchronization variables associated with a data store are sequentially consistent – all processes see the synchronization calls in the same order.
  - No operation on a synchronization variable is allowed to be performed until all previous writes have been completed everywhere.
  - No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed.

Weak Consistency - Semantics

- The weak consistency models enforce consistency on a group of operations, as opposed to individual reads and writes as is the case for strict, sequential, causal and FIFO consistency models.
- A synchronize(S) operation by P, causes all writes by P to be propagated to all other replicas and all external writes are propagated to P.
  - Process P forces the just written value out to all the other replicas.
  - Process P can be sure it’s getting the most recently written value before it reads.
Weak Consistency - Example

A valid sequence of events for weak consistency -- This is because $P_2$ and $P_3$ have yet to synchronize, so there’s no guarantees about the value in ‘x’.

An invalid sequence for weak consistency – $P_2$ has synchronized, so it cannot read ‘a’ from ‘x’ and should be getting ‘b’.

This S ensures that $P_2$ sees all updates

Release Consistency – Motivation

Weak consistency has the problem that when a synchronization variable is accessed, the data store does NOT know what caused this access:

1. Process finished writing the shared data, or
2. Process is about to start reading data

Consequently, the data store must take the actions required in both cases

1. Make sure that all locally initiated writes have been completed, thereby propagated to other replicas
2. Gathering in all writes from other replicas

If the data store could tell the difference between entering a critical region or leaving one, a more efficient implementation might be possible.
Release Consistency Basic Idea

- Divide access to a synchronization variable into two parts: an acquire and a release phase.
  1. When a process is **about to start accessing data** - Acquire forces a requester to wait until the shared data can be accessed
  2. When a process **finished accessing the shared data** - Release sends requester’s local value to other servers in data store.

Release Consistency Definition

- A distributed data-store is “Release Consistent” if it obeys the following rules:
  - Before a read or write operation on shared data is performed, all previous acquires done by the process must have completed successfully.
  - Before a release is allowed to be performed, all previous reads and writes by the process must have completed
  - Accesses to synchronization variables are FIFO consistent
    - Sequential consistency is not required
Release Consistency – Example

- A valid event sequence for release consistency
- Process $P_3$ has not performed an acquire, so there are no guarantees that the read of ‘x’ is consistent.
  - The data-store is simply not obligated to provide the correct answer
- Process $P_2$ does perform an acquire, so its read of ‘x’ is consistent

<table>
<thead>
<tr>
<th>Process</th>
<th>Event Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>Acq(L) W(x)a W(x)b Rel(L)</td>
</tr>
<tr>
<td>$P_2$</td>
<td>Acq(L) R(x)b Rel(L)</td>
</tr>
<tr>
<td>$P_3$</td>
<td>Acq(L) R(x)a</td>
</tr>
</tbody>
</table>

No Consistency Guarantee

Entry Consistency – Introduction

- With release consistency, all local updates are propagated to other replica servers during release of shared data.
- With entry consistency, each shared data item is associated with a synchronization variable.
- In order to access consistent data, each synchronization variable must be explicitly acquired.

Release consistency affects all shared data but entry consistency affects only those shared data associated with a synchronization variable.
Entry Consistency

- Acquire and release are still used, and the data-store meets the following conditions:
  - An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.
  - Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.
  - After an exclusive mode access to a synchronization variable has been performed, any other process's next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.

Entry Consistency – Semantics

- At an acquire, all remote changes to guarded data must be brought up to date.
- Before a write to a data item, a process must ensure that no other process is trying to write at the same time.

P1: \text{Acq} (Lx) \ W(x) a \ \text{Acq} (Ly) \ W(y) b \ \text{Rel} (Lx) \ \text{Rel} (Ly)

P2: \begin{array}{ccc}
\text{Acq} (Lx) & \text{R} (x) a & \text{R} (y) \text{NIL}
\end{array}

P3: \begin{array}{ccc}
\text{Acq} (Ly) & \text{R} (y) b
\end{array}

- Locks associate with individual data items, as opposed to the entire data-store.
- \(P_2\)'s read on ‘y’ returns NIL as no locks have been requested.
**Summary of Consistency Models**

<table>
<thead>
<tr>
<th>Consistency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strict</td>
<td>Absolute time ordering of all shared accesses matters.</td>
</tr>
<tr>
<td>Linearizability</td>
<td>All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp.</td>
</tr>
<tr>
<td>Sequential</td>
<td>All processes see all shared accesses in the same order. Accesses are not ordered in time.</td>
</tr>
<tr>
<td>Causal</td>
<td>All processes see causally-related shared accesses in the same order.</td>
</tr>
<tr>
<td>FIFO</td>
<td>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.</td>
</tr>
</tbody>
</table>

(a) Consistency models not using synchronization operations.

(b) Models with synchronization operations.

---

**Client-Centric Consistency Models**

- Data-centric consistency models aim at providing the system-wide view on a data store.
- Client-centric consistency models are generally used for applications that lack simultaneous updates
  - Most operations involve reading data.
- **Weak, client-centric** consistency models
  - Eventual consistency
  - Monotonic reads
  - Monotonic writes
  - Read your writes
  - Writes follow reads
Client-Centric Consistency Models

**Goal:** Show how we can perhaps avoid system-wide consistency, by concentrating on what specific clients want, instead of what should be maintained by servers.

**Background:** Most large-scale distributed systems apply replication for scalability, but can support only weak consistency.

**DNS:** Updates are propagated slowly, and inserts may not be immediately visible.

**News:** Articles and reactions are pushed and pulled throughout the Internet, such that reactions can be seen before postings.

**Lotus Notes:** Geographically dispersed servers replicate documents, but make no attempt to keep (concurrent) updates mutually consistent.

**WWW:** Caches all over the place, but there need be no guarantee that you are reading the most recent version of a page.

Eventual Consistency

- Systems such as DNS and WWW can be viewed as applications of large scale distributed and replicated databases that tolerate a relatively high degree of inconsistency

- **Common Assumption** – if no updates take place for a long time, all replicas will gradually and eventually become consistent

- This form of consistency is called **eventual consistency**

- Eventual consistency requires only that updates are guaranteed to propagate to all replicas

- Eventual consistent data stores work fine as long as clients always access the same replica – **what happens when different replicas are accessed?**
Consistency for Mobile Users – Inconsistencies

- Various inconsistencies may occur as a mobile user moves from location A to location B
  - Updates at A may not have yet been propagated to B
  - May be reading newer entries than the ones available at A
  - Updates at B may eventually conflict with those at A
- The consistency model must ensure that the entries updated and/or read at by the mobile user at A, are in B the way you left them in A.
  - This requirement ensures that the database will appear to be consistent to you.
Client-centric Consistency

- For the mobile user example, eventual consistent data stores will not work properly
- Client-centric consistency provides **guarantees for a single client** concerning the consistency of access to a data store by that client
- **No guarantees are given** concerning concurrent accesses by different clients

Client-Centric Consistency – Guarantees

- Guarantees are the responsibility of “session manager”, not servers
- Two sets are maintained
  - Read-set – set of writes that are relevant to session reads
  - Write-set – set of writes performed in session
- Update dependencies captured in read sets and write sets
- Four different client-central consistency models
  - Monotonic reads
  - Monotonic writes
  - Read your writes
  - Writes follow reads
Monotonic-Read Consistency

- A data store is said to be monotonic-read consistent if the following condition holds:
  - 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 or a more recent value.
  - Consequently, if a process has seen a value of x at time t, it will never see an older version of x at a later time.
- Notation: WS(xi[t]) is the set of write operations (at Li) that lead to version xi of x (at time t)
  - WS(xi[t1]; xj [t2]) indicates that it is known that WS(xi[t1]) is part of WS(xj[t2])

---

Monotonic Reads – Consistency

<table>
<thead>
<tr>
<th>L1:</th>
<th>WS(x₁)</th>
<th>R(x₁)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2:</td>
<td>WS(x₁;x₂)</td>
<td>R(x₂)</td>
</tr>
</tbody>
</table>

- **A data store that is monotonic-read consistent**
  - P performs a read operation on x at L1, R(x₁).
  - Later, P performs a read operation on x at L2, R(x₂)
  - Effects of write operations can be seen by succeeding read operations
    - Expressed by WS(x₁;x₂) which states WS(x₁) is part of WS(x₂)
Monotonic Reads – No Consistency

The read operations performed by a single process \( P \) at two different local copies of the same data store.
(b) A data store that does not provide monotonic reads.

Monotonic Writes

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

- A monotonic-write consistent data store.
  - Two write operations performed by a single process P at two different locations
  - \( WS(x_1) \) indicates that previous write operation at L1 has been propagated to L2

Monotonic Writes (3)

- A data store that does not provide monotonic-write consistency
  - Write operation at L1 has not propagated to L2

L1: \( W(x_1) \)
L2: \( WS(x_1) \)
Read Your Writes

- 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

- A data store that provides read-your-writes consistency
  - \( \text{WS}(x_1) \) is the series of write operations that took place since initialization at a local copy.
  - \( \text{WS}(x_1; x_2) \) denotes that operations in \( \text{WS}(x_1) \) also been performed at another local copy that has its set of operations in \( \text{WS}(x_2) \).
Read Your Writes

- A data store that does not provide read-your-writes consistency

writes follow reads

\[ L1: \quad W(x_1) \quad \rightarrow \quad WS(x_2) \quad \rightarrow \quad R(x_2) \]\n
writes follow reads

- 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.
A writes-follow-reads consistent data store.

A data store that does not provide writes-follow-reads consistency.
Consistency Models

- Background – Replication Motivation
- Data-Centric Consistency Models
  - Continuous Consistency
  - Consistent Ordering of Operations
- Client-Centric Consistency Models
  - Eventual Consistency
  - Monotonic Reads
  - Monotonic Writes
  - Read Your Writes
  - Writes Follow Reads