Topics to be covered
- Introduction
- Consistency Models
- Distribution Protocols
- Consistency Protocols

Introduction

- Reliability
- Performance

- Consistency Problems: keep replica consistent - in general, ensure that all conflicting operations (e.g., from the world of transactions: RW, and WW) are executed in the same order everywhere.

Guaranteeing global ordering is a costly operation, downgrade scalability.

Weaken consistency requirements

Organization of a distributed remote object shared by two different clients.

Object Replication

a) A remote object capable of handling concurrent invocations on its own.
b) A remote object for which an object adapter is required to handle concurrent invocations.
a) A distributed system for replication-aware distributed objects.
b) A distributed system responsible for replica management.
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. All processes see the same interleaving of operations.

Example

P1: W(x)
P2: W(y)
P3: R(y) R(x)
P4: R(y) R(x)
P5: R(x)
P6: R(x) R(y)

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

Note: a process sees writes from all processes but only its own reads.

Relationship between transaction serializability and sequential consistency: Similar only difference in granularity (transactions vs single read and write operations).

Data-Centric Consistency Models

- Relationship between (transaction) serializability and sequential consistency: Similar only difference in granularity (transactions vs single read and write operations).
- Serializability for replicated data:
  - x: a logical data item
  - x1, x2, ..., an physical data items
- Replica control protocols: maps each read/write on a logical data item x to a read/write on one (or more) of the physical data items.
- One-copy serializability (equivalence with a serial execution on an one-copy database - view equivalence: same reads-from and same set of final writes)
- (assumption: unique reads-from relationships on data items)

Linearizability

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. In addition, if \( \text{ts}(P1(x)) < \text{ts}(P2(y)) \) then operation \( P1(x) \) should precede \( P2(y) \) in this sequence.

- A linearizable data store is also sequentially consistent.
- The additional requirements of ordering according timestamps makes it more expensive to implement.

Example: Four valid execution sequences for the processes of the previous slide.

- \( x = 1; \)
- \( y = 1; \)
- \( z = 1; \)
- \( \text{print}(x, y); \)
- \( \text{print}(x, z); \)
- \( \text{print}(y, z); \)
- Prints: 111111

Signature:
111111

- \( x = 1; \)
- \( y = 1; \)
- \( z = 1; \)
- \( \text{print}(x, y); \)
- \( \text{print}(x, z); \)
- \( z = 1; \)
- \( \text{print}(y, z); \)
- Prints: 010111

Signature:
110101

- \( x = 1; \)
- \( \text{print}(x, z); \)
- \( y = 1; \)
- \( \text{print}(x, z); \)
- \( z = 1; \)
- \( \text{print}(x, y); \)
- Prints: 101011

Signature:
111111

- \( x = 1; \)
- \( \text{print}(y, z); \)
- \( y = 1; \)
- \( \text{print}(x, z); \)
- \( z = 1; \)
- \( \text{print}(x, y); \)
- Prints: 001011

Signature:
001011

90 different valid statement orderings produce a variety of different signatures.

WAYS TO EXPRESS CONSISTENCY

Consider an associated history (execution)

<table>
<thead>
<tr>
<th>Process P1</th>
<th>Process P2</th>
<th>Process P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: W(x)a</td>
<td>P2: R(x)b</td>
<td>P3: R(x)a</td>
</tr>
<tr>
<td>P4: R(x)a</td>
<td>P5: R(x)b</td>
<td>P6: R(x)c</td>
</tr>
</tbody>
</table>

Merge individual histories to get the execution history \( H \):

\[ H = W(x)W(x)bR(x)bR(x)bR(x)bR(x)bR(x)bR(x) \]

Legal history \( H \), if:

Rules:
1. Present program order (order of individual histories)
2. A read to \( x \) must always return the value most recently written to \( x \) (data coherency)
Sequential Consistency (2nd definition)

All legal values for history H must:
(i) Maintain the program order
(ii) Data coherency must be respected

Sequential Consistency (2nd definition)

It has been proved that: for any sequentially consistent store, changing the protocol to improve read performance makes write performance worse and vice versa.

Data-Centric Consistency Models

Data coherency: a R(x) must return the value most recently written to x, that is, the value written by the W(x) immediately preceding it in H

Coherence examines each data item in isolation
Called memory coherence when dealing with memory locations instead of data items

Data-Centric Consistency Models

Data coherency: a R(x) must return the value most recently written to x, that is, the value written by the W(x) immediately preceding it in H

Sequential Consistency (2nd definition)

All legal values for history H must:
(i) Maintain the program order
(ii) Data coherency must be respected

It has been proved that: for any sequentially consistent store, changing the protocol to improve read performance makes write performance worse and vice versa.

Data-Centric Consistency Models

No need to preserve the order of non-related (that is, of concurrent) events (= writes in our case).

Casual relation = related say by happened-before

Casual Consistency

Example:

A violation of a casually-consistent store. (b) A correct sequence of events in a casually-consistent store. – assume W2(x)b depends on W1(x)a

Data-Centric Consistency Models

Pipelined RAM

In other words: There are no guarantees about the order in which different processes see writes, except that two or more writes of the same process must arrive in order (that is, all writes generated by different processes are concurrent).

Also called PRAM consistency in the case of distributed shared memory
Pipeline RAM

Pipelined RAM
A valid sequence of events of FIFO consistency but not for casual
Data-Centric Consistency Models

Example:  

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

Implementation: need just to guarantee that writes from the same process  
arrive in order, tag writes with (process-id, sequence-number)

FIFO Consistency

Statement execution as seen by the three processes from the previous slide. The  
statements in bold are the ones that generate the output shown.

(c) P3's view

Process P1  
Process P2

x = 1;  
if (y == 0) kill (P1);  
y = 1;  
if (x == 0) kill (P1);

Initially, x = y = 0

Weak Consistency

Don't care that the reads and writes of a series of operations are  
immediately known to other processes. Just want the effect of  
the series itself to be known

A synchronization variable S with one associated operation  
synchronize(S) which synchronizes all local copies of the data store.  
When the data store is synchronized all local copies of process P are  
propagated to the other copies, whereas writes by other processes  
are brought into P's copies.

Strong Consistency Models: Operations on shared data are  
synchronized:

- Strict consistency (related to time)  
- Sequential Consistency (similar to database  
  serializability, what we are used to)  
- Causal Consistency (maintains only causal relations)  
- FIFO consistency (maintains only individual ordering)
A program fragment in which some variables may be kept in registers.

```c
int a, b, c, d, e, x, y; /* variables */
int *p, *q; /* pointers */
int f( int *p, int *q); /* function prototype */
a = x * x; /* a stored in register */
b = y * y; /* b as well */
c = a * a + b * b + a * b;
d = a * a * c; /* used later */
p = &a; /* p gets address of a */
q = &b; /* q gets address of b */
e = f(p, q) /* function call */
```

Weak Consistency

- A program fragment in which some variables may be kept in registers.

Data-Centric Consistency Models

Weak Consistency

- Example

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

(a) A valid sequence of events for weak consistency. (b) An invalid sequence for weak consistency.

Weak consistency implies that we need to lock and unlock data (implicitly or not).

Data-Centric Consistency Models

Release Consistency

Divide access to a synchronization variable into two parts: an acquire (for entering a critical region) and a release (for leaving a critical region) phase.

- Acquire: forces a requester to wait until the shared data can be accessed.
- Release: sends requester's local value to other servers in data store.

1. When a process does an acquire, the store will ensure that all the local copies of the protected data are brought up to date.
2. When a release is done, protected data that have been changed are propagated to other local copies of the store.

Example

<table>
<thead>
<tr>
<th>P1</th>
<th>Acq(L)</th>
<th>W(x)</th>
<th>W(y)</th>
<th>Re(L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>Acq(L)</td>
<td>R(x)</td>
<td>R(y)</td>
<td>Re(L)</td>
</tr>
<tr>
<td>P3</td>
<td></td>
<td></td>
<td></td>
<td>R(x)</td>
</tr>
</tbody>
</table>

A valid event sequence for release consistency.

Data-Centric Consistency Models

Entry Consistency

With entry consistency, all local updates are propagated to other copies/servers during release of shared data.

- When acquiring the synchronization variable, the most recent of its associated shared data are fetched.
- Whereas release consistency affects all data, entry consistency affects only those shared data associated with a synchronization variable.

Entry Consistency

1. Any acquire access to 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.
2. 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.
3. After an exclusive mode access to a synchronization variable has been performed, any other process' next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.
**Data-Centric Consistency Models**

**Entry Consistency**

Example:

\[ P_1: \text{Acq}(Lx) \text{ W}(i) \text{ Acq}(Lx) \text{ W}(i) \text{ R}(i) \text{ R}(i) \text{ R}(i) \text{ R}(i) \]  

A valid event sequence for entry consistency.

**Weak Consistency Models**: Synchronization occurs only when shared data are locked and unlocked:

- General Weak Consistency
- Release Consistency
- Entry consistency

**Consistency**

<table>
<thead>
<tr>
<th>Consistency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>Sequential</td>
<td>All processes see causal-related shared accesses in the same order.</td>
</tr>
<tr>
<td>Linearizability</td>
<td>All processes see all shared accesses in the same order. Accesses are not ordered in time.</td>
</tr>
<tr>
<td>Serial</td>
<td>All processes see all shared accesses in the same order. Accesses are ordered according to a monotonically increasing timestamp.</td>
</tr>
</tbody>
</table>

**Client-Centric Consistency Models**

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

**Monotonic Reads**

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

**Notation**

- \( W(i) \): the set of write operations (at site \( L_i \)) that lead to version \( x_i \) of \( x \) (at time \( T_i \))
- \( W(i) \cup W(j) \): indicates that is known that \( W(i) \cup W(j) \) is part of \( W(x) \)

**Example**: Reading incoming email while on the move; each time you connect: monotonic reads.
The read operations performed by a single process $P$ at two different local copies of the same data store.

(a) A monotonic-read consistent data store
(b) A data store that does not provide monotonic reads.

### Monotonic Reads

| L1: $W(x)$ | R($x$) |
| L2: $W(x_2)$ | R($x_2$) |

(b) $L1: W(x)$ $R(x)$ $L2: W(x_2)$ $R(x_2)$ $W(x_2)$

The read operations performed by a single process $P$ at two different local copies of the same data store.

(a) A monotonic-read consistent data store
(b) A data store that does not provide monotonic reads.

### 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.

| L1: $W(x)$ | R($x$) |
| L2: $W(x_2)$ | R($x_2$) $W(x_2)$ |

(b) $L1: W(x)$ $L2: W(x_2)$ $R(x_2)$ $W(x_2)$

The write operations performed by a single process $P$ at two different local copies of the same data store.

(a) A monotonic-write consistent data store
(b) A data store that does not provide monotonic-write consistency.

### Examples

#### Monotonic Reads
- Updating a program on server $S2$ and ensuring that all components on which compilation and linking depend are also placed at $S2$.

#### Monotonic Writes
- Maintaining versions of replicated files in the correct order everywhere (propagate the previous version to the server where the newest version is installed).

#### Reads Follow Writes

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.

| L1: $W(x)$ | R($x$) |
| L2: $W(x_2)$ | R($x_2$) |

(b) $L1: W(x)$ $L2: W(x_2)$ $R(x_2)$

Example: See reactions to posted articles only if you have the original posting (a read "pulls in" the corresponding write operation).

### Distribution Protocols

- Replica Placement
- Update Propagation
- Epidemic Protocols
**Replica Placement**

- Where, when and by whom data copies are placed in a distributed system?

**Permanent Replicas**

The initial set of replicas that constitute a distributed data store.

**Server-Initiated Replicas**

Copies of data to enhance performance, created by the servers.

- Keep track of access counts per file, aggregated by considering server closest to requesting clients.

  - Number of accesses drop below threshold D: drop file.
  - Number of accesses exceed threshold R: replicate file.
  - Number of accesses between D and R: file can only be migrated (no drop or replication), when?

Example, when two clients (C1 and C2) share the same closest server (P).

**Client-Initiated Replicas**

Client initiated replicas or (client) caches.

- Generally kept for a limited amount of time (replaced or become stale).
- Cache hit.
- Share caches among clients.
- Typically placed at the same machine as the client.

**Update Propagation**

- State vs Operation
  - Propagate only notification/invalidation of update.
  - Often used for caches.
  - Invalidation protocols work well when read-to-write ratio is small.
  - Transfer values/copies from one copy to the other.
  - Lag the changes, aggregate updates.
  - Propagate the update operation to other copies (aka active replication).

- Push vs Pull
  - Push or server based (update is propagated without a client request).
  - Pull or client based.

<table>
<thead>
<tr>
<th>Issue</th>
<th>Push-based</th>
<th>Pull-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>State of server</td>
<td>List of client replicas and caches</td>
<td>None</td>
</tr>
<tr>
<td>Messages sent</td>
<td>Update (and possibly fetch update later)</td>
<td>Poll and update</td>
</tr>
<tr>
<td>Response time at</td>
<td>Immediate (or fetch-update time)</td>
<td>Fetch-update time</td>
</tr>
</tbody>
</table>

**Comparison between push-based and pull-based protocols in the case of multiple client-single server systems.**
**Update Propagation**

**A Hybrid Protocol: Leases**

- **Lease**: A contract in which the server promises to push updates to the client until the lease expires.

  - **Lease expiration time** is dependent on system behavior (adaptive leases).
  - **Age-based leases**: an object that has not changed for long time, will not change in the near future, so provide a long-lasting lease.
  - **Renewal-frequency based leases**: The more often a client requests a specific object, the longer the expiration time for that client (for that object) will be.
  - **State-based leases**: The more loaded a server is, the shorter the expiration times become.

**Epidemic Algorithms**

**Overview**

- *Basic idea*: assume there are no write-write conflicts (e.g., updates for a specific item are initiated at a single server).
  - Update operations are initially performed at one or only a few replicas.
  - A replica passes its updated state to a limited number of neighbors.
  - Update propagation is lazy, i.e., not immediate.
  - Eventually, each update should reach every replica.
- **Anti-entropy**: Each replica regularly chooses another replica at random, and exchanges state differences, leading to identical states at both afterwards.
- **Gossiping**: A replica that has just been updated (i.e., has been contaminated) tells a number of other replicas about its update (contaminating them as well).

**System Model**

- A collection of servers, each storing a number of objects.
- Each object O has a primary server at which updates for O are initiated.
- An update of an object O at server S is timestamped.

Notation: timestamp \( T(O, S) \), value \( VAL(O, S) \)

**Gossiping**

A server \( S \) having an update to report, contacts other servers. If a server is contacted to which the update has already been propagated, \( S \) stops contacting other servers with probability \( 1/k \).

IF \( s \) is the fraction of susceptible servers (i.e., which are unaware of the updates), it can be shown that with many servers:

\[
 s = e^{-(k+1)(1-s)}
\]

<table>
<thead>
<tr>
<th>( k )</th>
<th>( s )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.2</td>
</tr>
<tr>
<td>2</td>
<td>0.06</td>
</tr>
<tr>
<td>3</td>
<td>0.02</td>
</tr>
<tr>
<td>4</td>
<td>0.007</td>
</tr>
<tr>
<td>5</td>
<td>0.0035</td>
</tr>
</tbody>
</table>

If we really have to ensure that all servers are eventually updated, gossiping alone is not enough.

**Deleting Values**

We cannot remove an old value from a server and expect the removal to propagate. Why?

Treat removal as a special update by inserting a death certificate.

When to remove a death certificate?

- Run a global algorithm to detect whether the removal is known everywhere, and then collect the death certificates (looks like garbage collection).
- Assume that death certificates propagate in finite time, and associate a maximum lifetime for a certificate (can be done at risk of not reaching all servers).
- It is necessary that a removal actually reaches all servers.

**Scalability?**
Consistency Protocols
Primary-Based Protocols
Replicated-Write Protocols
Cache-Coherence Protocols

Implementation of a specific consistency model. We will concentrate on sequential consistency.

Primary-Based Protocols
Each data item $x$ has an associated primary responsible for coordinating write operations on $x$.

Remote-Write protocols

Simplest model: no replication, all read and write operations are forwarded to a single server.

Remote-backup protocol: reads on local copies, but writes at a fixed primary copy.

An update is applied as a blocking operation.

Sequential consistency. Why?

Non-blocking write variant: as soon as the primary has updated its local copy, it returns an ack, then it tells the backup to perform the update as well. Consistency?

Local-Write protocols

Case 1: there is only a single copy of each data item $x$ (no replication), a single copy is migrated between processes.

Useful when writes are expected to come in series from the same client (e.g., mobile computing without replication).

Distributed Systems, Spring 2003
Active Replication: updates are propagated to multiple replicas where they are carried out.

The problem of replicated invocations.

Replicated-Write Protocols

Assign a coordinator on each side (client and server) that ensures that only one invocation and one reply is sent.

Cache Consistency Protocols

Write-through caches
Write-back caches

Number of replica servers jointly implement a causal-consistent data store. Clients normally talk to front ends which maintain data to ensure causal consistency.

The general organization of a distributed data store. Clients are assumed to also handle consistency-related communication.
Performing a read operation at a local copy.

1. \( \text{DEP}(R) = \text{LOCAL}(C) \)
2. \( \text{DEP}(R) \leq \text{VAL}(i) \)
3. Data $\oplus$ VAL(i)
4. \( \text{LOCAL}(C) = \max(\text{LOCAL}(C), \text{VAL}(i)) \)

Performing a write operation at a local copy.

1. \( \text{DEP}(W) = \text{LOCAL}(C) \)
2. \( \text{WORK}(i) = \text{WORK}(i) + 1 \)
3. \( \text{t}(W) = \text{t}(W) + 1 \)
4. \( \text{LOCAL}(C) = \max(\text{LOCAL}(C), \text{t}(W)) \)
5. \( \text{DEP}(W) \leq \text{VAL}(i) \)