CONSISTENCY IN DISTRIBUTED SYSTEMS

Introduction to Consistency

- Need replication in DS to enhance reliability/performance.
- In *Multicasting Example*, major problems to keep replicas consistent.
- Must ensure *all* copies kept updated else replicas won’t be the same.
- **Consistency models** assume that multiple procs access shared data.
- Look at consistency here in terms of what processes expect when reading/updating shared data, knowing others are accessing it too.
- Also must consider *how* to implement consistency.
- There are two independent issues we need to consider:
  1. *Managing replicas* (handle placing replica servers, content distribution)
  2. *Ensuring replica consistency* (ie update one, must update other copies)
- Hard implementing efficiently on large-scale DS, use simpler models
• **Performance and scalability**

  **Consistency (/2)**

  **Main issue:**
  - For replica consistency, must ensure all conflicting operations done in same order everywhere – ‘Tight Consistency’.

  **Conflicting operations:**
  - From the world of Transactions:
    - Read–write conflict: read & write operations act concurrently
    - Write–write conflict: two concurrent write operations.
  - **Ideal:** Update is an atomic transaction – but this causes bandwidth problems in large-scale networks:
    - $P$ accesses local replica $N$ times/s, but replica updated $M$ times/s.
    - Assume that update totally refreshes previous version of local replica.
    - If $N \ll M$, i.e. very low access-to-update ratio, $P$ won’t access many updated replica versions => useless/waste of bandwidth
    => better installing local replica close to $P$

  **Issue**
  - Ensuring global ordering on conflicting operations can be costly, downgrading scalability
  - **Solution:** weaken consistency requirements so that hopefully global synchronization can be avoided

---

**Consistency (/3)**

• **Data-centric** consistency models:

  **Consistency model:**
  - A contract between a (distributed) data store and processes,
  - Have a range of consistency models
    - Those with major restrictions on what read results of the last write operation are easy to use, whereas those with minor restrictions are sometimes difficult.
  - Data store specifies precisely what results of R/W operations are in the presence of concurrency

---

**General organization of logical data store, physically distributed & replicated across multiple processes.**
Consistency (/4)

- **Data-centric** consistency models: *Continuous Consistency*
  
  - **Observation**: Can actually talk about a degree of consistency:
    1. Replicas may differ in their *numerical value*
      - E.g. replicas of stock data should not deviate by more than $0.02
      - i.e. a maximum numerical deviation
    2. Replicas may differ in their relative *staleness*
      - E.g. normal users only need weather data O(hours) old usually
      - But other weather use-cases (e.g. mountaineering) need O(mins)
    3. Also possibly differences in number and order of performed update operations
      - These are updates which are applied tentatively to a local copy, awaiting global agreement from all replicas.
      - A result is some updates may need to be rolled back and applied in a different order before becoming permanent.
      - Intuitively, ordering deviations are harder to grasp than the other two metrics.

Consistency (/5)

- **Data-centric** consistency models: *Sequential consistency*
  
  - **Definition**
    - Result of any execution same as if operations of all processes were executed in some sequential order, and
    - Operations of each individual process appear in this sequence in order specified by its program.
    - i.e. when procs run concurrently on different machines, any valid operation (r/w) is acceptable if all procs see same interleaving.
Consistency (/6)

- **Data-centric** consistency models: **Causal consistency**:
  - **Definition**
    - Writes that are potentially causally related must be seen by all processes in the same order.
    - Concurrent writes may be seen in different order by different procs.
    - Weaker variant to Sequential Consistency as distinguishes between events that are potentially causally related and those that aren’t.

```
(a) Incorrect sequence of events in causally consistent data stores.
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

(b) Correct sequence of events in causally consistent data stores.
P1: W(x)a
P2: W(x)b
P3: R(x)b R(x)a
P4: R(x)a R(x)b
```

Consistency (/7)

- **Grouping operations**:
  - **Definition**
    - Granularity imposed by consistency models above frequently does not match granularity provided by applications
    - Normally accesses to shared variables provided by synchronization variables on critical sections **Acq** / **Rel** are sequentially consistent.
    - No access to a synchronization variable can be performed until all previous writes have completed everywhere.
    - No data access is allowed to be performed until all previous accesses to synchronization variables have been performed.
  - **Basic idea**
    - Don’t care that r/w of a series of operations are immediately known to other processes, just that effect of the series itself to be known.

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

A valid event sequence for entry consistency.
Consistency (8)

• **Client-Centric** Consistency Models:

  – **Definition**
    
    • Avoid *Global Consistency* models, concentrate on consistency from view of a single (mobile?) client.

  – **Consistency for mobile users**
    
    • Consider a distributed database with access through your notebook.
    • Assume your notebook acts as a front end to the database.
      
      – At point A you access DB doing reads and updates.
      – At point B you continue your work, but unless you access same server as that at point A, you may detect inconsistencies:
        
        » your updates at A may not have yet been propagated to B
        » you may be reading newer entries than the ones available at A
        » your updates at B may eventually conflict with those at A

Consistency (9)

• **Client-Centric Consistency** Models:

  – **Note**
    
    • Must ensure entries updated/read at A, are in B as per last seen at A.
    • Here DB will appear consistent to you, (*e.g. eventual consistency*)
    • Consider a distributed database with access through your notebook.
    • Assume your notebook acts as a front end to the database.

Problem: Mobile user accesses different replicas of distributed DB over a short period of time.

**Solution:** Alleviated by CCC. Guarantees consistency for a single client but not concurrent access.
Consistency (/11)

• **Client-Centric Consistency** Models:
  
  — **Monotonic Reads**
    
    • **Notation**
      
      - \( WS(x_1[t_1]) \) is set of writes (at \( L_1 \)) that lead to version \( x_1 \) of \( x \) (at time \( t \))
      
      - \( WS(x_1[t_1]; x_j[t_2]) \) indicates that it is known that \( WS(x_1[t_1]) \) is part of \( WS(x_j[t_2]) \)
    
    • **Example**
      
      - Automatically reading personal calendar writes from different servers.
      
      - Monotonic Reads guarantees that user sees all updates, no matter from which server the automatic reading takes place.
    
    • **Example**
      
      - Reading (not modifying) incoming mail while you are on the move.
      
      - Each time you connect to a different e-mail server, that server fetches (at least) all the updates from server you previously visited.
Consistency (/12)

- **Client-Centric Consistency Models:**
  - **Monotonic Writes**
    - Often writes must be propagated in *correct order* to all DS copies.
    - Write operation by a process on a data item \( x \) is completed before any successive write operation on \( x \) by the same process...

```
<table>
<thead>
<tr>
<th>L1: W(x₁)</th>
<th>L2: WS(x₁) W(x₂)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Set at ( L_1 ) propagated at ( L_2 ) before ( W(x₂) )</td>
<td></td>
</tr>
</tbody>
</table>
```

- **Example**
  - Updating a program at server \( S₂ \), ensuring that all components on which compilation/linking depends, are also placed at \( S₂ \).

```
<table>
<thead>
<tr>
<th>L1: W(x₁)</th>
<th>L2: W(x₂)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writes performed by a single process ( P ) at 2 different local copies of the same data store. (b) Data store not providing monotonic-write consistency.</td>
<td></td>
</tr>
</tbody>
</table>
```

Consistency (/13)

- **Client-Centric Consistency Models:**
  - **Monotonic Writes (cont’d)**
    - **Example**
      - Updating a program at server \( S₂ \), ensuring that all components on which compilation/linking depends, are also placed at \( S₂ \).

```
```

- **Example**
  - Maintaining versions of replicated files in correct order everywhere.
  - i.e. propagate the previous version to server where newest version is installed.
Consistency (/14)

- **Client-Centric Consistency** Models:
  - **Read Your Writes**
    - Write is always completed before successive read by same process no matter where read happens (closely related to Monotonic Reads)
    - 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...

  \[
  \begin{align*}
  L1: & \quad W(x_1) \\
  L2: & \quad WS(x_1;x_2) \rightarrow R(x_2)
  \end{align*}
  \]

  (a) A data store that provides read-your-writes consistency.

  Effects of \( W(x_1) \) propagated before \( R(x_2) \)

  - **Example**
    - Updating Webpage ensuring browser shows updates not cached copy.

Lecture 8: Safe Access to Dist’d Shared Resources  CA4006 Lecture Notes (Martin Crane 2015)  49

---

Consistency (/15)

- **Client-Centric Consistency** Models:
  - **Writes follow Reads**
    - Write on data item \( x \), following previous read on \( x \) by same process, will always occur on same or more recent value of \( x \) that was read

  \[
  \begin{align*}
  L1: & \quad WS(x_1) \\
  L2: & \quad WS(x_1;x_2) \rightarrow W(x_2)
  \end{align*}
  \]

  (a) A writes-follow-reads consistent data store.

  Operations in write set at \( L_1 \) have been performed at local copy \( x_2 \)

  - **Example**
    - See replies to posted articles only if you have original posting (a read “pulls in” the corresponding write operation).

Lecture 8: Safe Access to Dist’d Shared Resources  CA4006 Lecture Notes (Martin Crane 2015)  50
REPLICATION IN DISTRIBUTED SYSTEMS

Introduction to Replication

• Need replication in Dist’d Systems to enhance reliability/performance
• Distribution Transparency achieved through either replication of object/state and possible hosting at different locations.
• Also, possible multiplicity of locations that replicas are hosted at used to aide Scalability, another goal of Distributed Systems.
• Replication takes a number of forms:
  1. Replicated file servers/databases
  2. Mirrored Websites
  3. Web caches in browsers & proxies
  4. File caching at server and client
• Caching:
  – Special form of replication except it’s a client decision
  – A Cache normally placed on same as its client (or at least same LAN)
Caching & Replication (/2)

- **Placement Protocols:**
  - Says when/where doc copy is placed/removed - placement can be initiated either by servers or clients,
  - Distinguish three different layers of host servers holding copy of a doc:
    - **Core**: host permanent replicas, often, primary server hosts each page
      - Clusters of Web servers/servers mirroring whole sites typify many permanent replicas.
    - **Middle**: host doc-initiated replicas, mostly created by one permanent replica.
      - On internet, doc-initiated replicas appear in CDNs.
      - Here, content transferred to servers near requesting clients.
    - **Outer**: host client-initiated replicas, a.k.a. cache servers.
      - Creating a cached version of a doc is entirely a local decision.
      - In principle, taken independently from the replication strategy of the doc.
  - However, decision to cache may be subject to many constraints, e.g. client caches only docs expected not to change soon.
  - Also, may have limited disk space available for caching.
  - Web proxy caches typify client-initiated replicas in the Internet.

Caching & Replication (/3)

- **Placement Protocols (cont’d):**

Caching & Replication (/2)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change distribution</td>
<td>- notification</td>
<td>Describes how changes between replicas are distributed. Is only a notification (or invalidation) sent telling that an update is needed, is the full state sent, or only differences, or is the operation sent that is to be carried out to update the receiver's state?</td>
</tr>
<tr>
<td></td>
<td>- full state</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- state differences</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- operation</td>
<td></td>
</tr>
<tr>
<td>Replica responsiveness</td>
<td>- immediate</td>
<td>Describes how quickly a replica reacts when it notices it is no longer consistent with the other replicas. A passive replica will do nothing.</td>
</tr>
<tr>
<td></td>
<td>- lazy (e.g., periodic)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- passive</td>
<td></td>
</tr>
<tr>
<td>Replica reaction</td>
<td>- pull</td>
<td>Describes what a (non passive) replica does when it notices it is inconsistent with other replicas. It either sends or requests updates.</td>
</tr>
<tr>
<td></td>
<td>- push</td>
<td></td>
</tr>
<tr>
<td>Write set</td>
<td>- single</td>
<td>This parameter gives the number of writers that may simultaneously access the document.</td>
</tr>
<tr>
<td></td>
<td>- multiple</td>
<td></td>
</tr>
<tr>
<td>Coherence group</td>
<td>- permanent only</td>
<td>Describes who implements the consistency model: permanent and/or document-initiated replicas. Caches are generally not part of the coherence group.</td>
</tr>
<tr>
<td></td>
<td>- permanent and document-initiated</td>
<td></td>
</tr>
</tbody>
</table>

Summary

- There are 2 main reasons for replicating data in DS: improving the reliability improving performance.
- Replication introduces a consistency problem: whenever a replica is updated, that replica becomes different from the others.
- To keep replicas consistent, need to propagate updates in such a way that temporary inconsistencies are not noticed.
- Unfortunately, doing so may severely degrade performance, especially in large-scale distributed systems.
- To solve this problem various consistency models exist:
  - Data-Centric Consistency e.g. sequential (& weaker) causal consistency
  - Client-Centric Consistency v. weak model (e.g. eventual consistency)