CSE 5306
Distributed Systems

Consistency and Replication
Reasons for Replication

• Data are replicated for
  – the reliability of the system

• Servers are replicated for performance
  – Scaling in numbers
  – Scaling in geographical area

• Dilemma
  – Gain in performance
  – Cost of maintaining replication

  • Keep the replicas up to date and ensure consistency
Data-Centric Consistency Models (1)

- Consistency is often discussed in the context of read and write on
  - shared memory, shared databases, shared files
- A more general term is: data store
  - A data store is distributed across multiple machines
  - Each process can access a local copy of the entire data store

The general organization of a logical data store, physically distributed and replicated across multiple processes.
Data-Centric Consistency Models (2)

• A consistency model is essentially a contract between processes and the data store
  - A process that performs a read operation on a data item expects to the value written by the last write operation

• However, due to the lack of global clock, it is hard to define which write operation is the last one

• Will discuss
  - Continuous consistency model
  - Sequential consistency model
  - Causal consistency model
Continuous Consistency

• Defines three independent axes of inconsistency
  - Deviation in numerical values between replicas
    • E.g., the number and the values of updates
  - Deviation in staleness between replicas
    • related to the last time a replica was updated
  - Deviation with respect to the ordering of update operations
    • E.g. the number of uncommitted updates

• To measure inconsistency, we define “conit”
  - A conit specifies the unit over which consistency is to be measured
  - E.g., a record representing a stock, a weather report
Measuring Inconsistency: An Example

Replica A

Conit
\[ x = 6; y = 3 \]

Operation | Result
--- | ---
< 5, B> x := x + 2 | \[ x = 2 \]
< 8, A> y := y + 2 | \[ y = 2 \]
<12, A> y := y + 1 | \[ y = 3 \]
<14, A> x := y + 2 | \[ x = 6 \]

Vector clock A = (15, 5)
Order deviation = 3
Numerical deviation = (1, 5)

Replica B

Conit
\[ x = 2; y = 5 \]

Operation | Result
--- | ---
< 5, B> x := x + 2 | \[ x = 2 \]
<10, B> y := y + 5 | \[ y = 5 \]

Vector clock B = (0, 11)
Order deviation = 2
Numerical deviation = (3, 6)

An example of keeping track of consistency deviation.
• **Requirement:** two replicas may differ in no more than ONE update

  (a) When each data item in Replica 1 is updated, the data items in replica 2 has to be updated as well

  (b) It is not necessary to update replica 2 since they differ in only one update per conit
Sequential Consistency

• The symbols for read and write operations

<table>
<thead>
<tr>
<th>P1:</th>
<th>W(x)a</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2:</td>
<td>R(x)NIL R(x)a</td>
</tr>
</tbody>
</table>

• A data store is sequential consistent if
  - The result of any execution is the same, as if
  - the (read and write) operations 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
Example 1

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

(a)

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

(b)

(a) A sequentially consistent data store.
(b) A data store that is not sequentially consistent.
Example 2

<table>
<thead>
<tr>
<th>Process P1</th>
<th>Process P2</th>
<th>Process P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x \leftarrow 1; )</td>
<td>( y \leftarrow 1; )</td>
<td>( z \leftarrow 1; )</td>
</tr>
<tr>
<td>\text{print}(y, z);</td>
<td>\text{print}(x, z);</td>
<td>\text{print}(x, y);</td>
</tr>
</tbody>
</table>

Four valid execution sequences for the above processes

1. \( x \leftarrow 1; \) \text{print}(y, z); \text{print}(x, y); \quad \text{Prints: 001011} \quad \text{Signature: 001011} \quad (a)
2. \( x \leftarrow 1; \) \text{print}(x, z); \text{print}(y, z); \quad \text{Prints: 101011} \quad \text{Signature: 101011} \quad (b)
3. \( y \leftarrow 1; \) \text{print}(x, y); \text{print}(x, z); \quad \text{Prints: 010111} \quad \text{Signature: 110101} \quad (c)
4. \( z \leftarrow 1; \) \text{print}(y, z); \text{print}(x, z); \quad \text{Prints: 111111} \quad \text{Signature: 111111} \quad (d)
Causal Consistency

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

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

This sequence is causally consistent, but not sequentially consistent.
Another Example

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

(a) A violation of a causally-consistent store.

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

(b) A correct sequence of events in a causally-consistent store
Grouping Operations

- Sequential and causal consistency is defined at the level of read and write operations
  - However, in practice, such granularity does not match the granularity provided by the application
    - Concurrency is often controlled by synchronization methods such as mutual exclusion and transactions

- A series of read/write operations, as one single unit, are protected by synchronization operations such as ENTER_CS and LEAVE_CS
  - This atomically executed unit then defines the level of granularity in real-world applications
Synchronization Primitives

• Necessary criteria for correct synchronization:
  - An acquire access of a synchronization variable, not allowed to perform until
    • all updates to guarded shared data have been performed with respect to that process.
  - Before exclusive mode access to synchronization variable by process is allowed to perform with respect to that process,
    • no other process may hold synchronization variable, not even in non-exclusive mode.
  - After exclusive mode access to synchronization variable has been performed,
    • any other process’ next non-exclusive mode access to that synchronization variable is performed respect to that variable’s owner.
Entry Consistency

- It requires
  - the programmer to use acquire and release at the start and end of each critical section, respectively.
  - each ordinary shared variable to be associated with some synchronization variable

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

A valid event sequence for entry consistency.
Consistency v.s. Coherence

- **Consistency** deals with a set of processes operate on
  - a set of data items (they may be replicated)
  - This set is consistent if it adheres to the rules defined by the model

- **Coherence** deals with a set of processes operate on
  - a single data item that is replicated at many places
  - It is coherent if all copies abide to the rules defined by the model
Eventual Consistency

• In many distributed systems such as DNS and World Wide Web,
  – updates on shared data can only be done by one or a small group of processes
  – most processes only read shared data
  – a high degree of inconsistency can be tolerated

• Eventual consistency
  – If no updates take place for a long time, all replicas will gradually become consistent
  – Clients are usually fine if they only access the same replica

• However, in some cases, clients may access different replicas
  – E.g., a mobile user moves to a different location

• Client-centric consistency:
  – Guarantee the consistency of access for a single client
Monotonic-Read Consistency

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

In other words,

- if a process has seen a value of \( x \) at time \( t \), it will never see an older version of \( x \) at any later time
An Example

- Notations
  - $x_i[t]$: the version of $x$ at local copy $L_i$ at time $t$
  - $WS(x_i[t])$: the set of all writes at $L_i$ on $x$ since initialization

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

• 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.

• In other words
  – A write on a copy of x is performed only if this copy is brought up to date by means of
    • any preceding write on x, which may take place at other copies, by the same process.
An Example

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.
Read-Your-Write Consistency

• 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.

• In other words,
  – A write operation is always completed before a successive read operation by the same process
    • no matter where that read takes place
An Example

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

(b) A data store that does not.
Write-Follow-Read Consistency

• 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.

• In other words,
  – Any successive write operation by a process on a date item \( x \) will be performed on a copy of \( x \) that
    • is up to date with the value most recently read by that process
An Example

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

(b) A data store that does not provide writes-follow-reads consistency.
Replica Management

• Two key issues for distributed systems that support replication
  
  - Where, when, and by whom replicas should be placed? Divided into two subproblems:
    - Replica server placement: finding the best location to place a server that can host a data store
    - Content placement: find the best server for placing content
  
  - Which mechanisms to use for keeping replicas consistent
Replica-Server Placement

• Not intensively studied
  - more of a management and commercial issue than an optimization problem

• Some typical approaches
  - Select K out N: select the one that leads to the minimal average distance to all clients, and repeat
  - Ignore the client, only consider the topology, i.e., the largest AS, the second largest AS, so and so forth
  - However, these approaches are very expensive

• Region-based approach
  - A region is identified to be a collection of nodes accessing the same content, but for which the internode latency is low
Region-Based Approach

Too small

Cell

Too large

Just right

Choosing a proper cell size for server placement.
Content Replication and Placement

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

- Observe the client access pattern and dynamically add or remove replicas to improve the performance
- One example algorithm
  - Count the access requests of F from clients and combining points
  - If the request drops significantly, delete replica F
  - If a lot of requests from one combining point, replicate F at such point
Client-Initiated Replicas

- Mainly deals with client cache,
  - i.e., a local storage facility that is used by a client to temporarily store a copy of the data it has just requested

- The cached data may be outdated
  - Let the client checks the version of the data

- Multiple clients may use the same cache
  - Data requested by one client may be useful to other clients as well, e.g., DNS look-up
  - This can also improve the chance of cache hit
Content Distribution

- Deals with the propagation of updates to all relevant replicas
- Two key questions
  - What to propagate (state v.s. operations)
    - Propagate only a notification of an update.
    - Transfer data from one copy to another.
    - Propagate the update operation to other copies.
  - How to propagate the updates
    - Pull v.s. push protocols
    - Unicast v.s. multicast
Pull v.s Push Protocols

- **Push-based approach**
  - It is server-based, updates are propagated (via multicast if possible) to other replicas without those replicas even asking for
  - It is usually used for high degree of consistency

- **Pull-based approach**
  - It is client-based, updates are propagated (often via unicast) when a client or a replication server asks for it

<table>
<thead>
<tr>
<th>Issue</th>
<th>Push-based</th>
<th>Pull-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>State at 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 client</td>
<td>Immediate (or fetch-update time)</td>
<td>Fetch-update time</td>
</tr>
</tbody>
</table>

A comparison between push-based and pull-based protocols in the case of multiple-client, single-server systems
Consistency Protocols

• A consistency protocol describes
  – an implementation of a specific consistency model

• Will discuss
  – Continuous consistency protocols
    • Bounding numerical, staleness, ordering deviation
  – Primary-based protocols
    • Remove-write and local-write protocols
  – Replication-write protocols
    • Active replication and quorum-based protocols
Continuous Consistency Protocols (1)

• **Bounding numerical deviation**
  - The number of unseen updates, the absolute numerical value, or the relative numerical value
  - E.g., the value of a local copy of $x$ will never deviate from the real value of $x$ by a threshold

• **Let us concern about the number of update unseen**
  - i.e., the total number of unseen updates to a server shall never exceed a threshold $\tilde{\delta}$

• **A simple approach for $N$ replicas**
  - Every server $i$ tracks every other server $j$’s state about $i$’s local writes, i.e., the number of $i$’s local writes not been seen by $j$
  - If this number exceeds $\tilde{\delta} / (N-1)$, $i$ will propagate its writes to $j$. 
Continuous Consistency Protocols (2)

- **Bounding staleness deviation**
  - Each server maintains a clock $T(i)$, meaning that this server has seen all writes of $i$ up to time $T(i)$
  - Let $T$ be the local time. If server $i$ notices that $T - T(j)$ exceeds a threshold, it will pull the writes from server $j$

- **Bounding ordering deviation**
  - Each server keeps a queue of tentative, uncommitted writes
  - If the length of this queue exceeds a threshold,
    - the server will stop accepting new writes and
    - negotiate with other servers in which order its writes should be executed, i.e., enforce a globally consistent order of tentative writes
  - Primary-based protocols can be used to enforce a globally consistent order of tentative writes
Remote-Write Protocols

- Problems: it is a blocking operation at the client
- Replace it with a non-blocking update, i.e., update the local copy immediately and then the local server ask the backup server to perform the update
- However, the non-blocking version does not have fault tolerance
Local-Write Protocols

- The difference is that the primary copy migrates between processes
- Benefit: multiple successive writes can be performed locally, while others can still read
  - if a non-blocking protocol is followed by which updates are propagated to the replicas after the primary has finished the update
Replicated-Write Protocols (1)

- Active replication
  - Update are propagated by means of the write operation that cause the update

- The challenge is that the operations have to be carried out in the same order everywhere
  - Need a totally-ordered multicast mechanism such as the one based on Lamport’s logical clocks
    - However, this algorithm is expensive and does not scale

- An alternative is to use a central sequencer
  - However, this central sequencer does not solve the scalability problem
Replicated-Write Protocols (2)

- **Quorum-based protocols**
  - Require a client to get permission from multiple servers before a read or write

- **A simple version**
  - A read or write has to get permission from half plus 1 servers

- **A better version:** a client must get permission from
  - A read quorum: an arbitrary set of $N_r$ servers
  - A write quorum: an arbitrary set of $N_w$ servers
  - such that $N_r+N_w>N$ and $N_w>N/2$
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).