Distributed Systems (5DV147)

Consistency

Fall 2014
Intuition

- A replicated system is correct when:
  - It maintains execution despite failures
  - Clients can’t tell the difference between the results from a system that uses replicated data from those obtained from a system with a single correct replica
  - In general we expect a read to return the last value written
    ... but which is the last value written since we don’t have a global clock?
# Example

<table>
<thead>
<tr>
<th>Client 1</th>
<th>Client2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$setBalance_B(x,1)$</td>
<td></td>
</tr>
<tr>
<td>$setBalance_A(y,2)$</td>
<td>$readBalance_A(y) = 2$</td>
</tr>
<tr>
<td></td>
<td>$readBalance_A(x) = 0$</td>
</tr>
</tbody>
</table>

- Local replica of Client 1 is B
- Local replica of Client 2 is A
Consistency problem

Replication improves reliability and performance

... but

when a replica is updated, it becomes different from the others

... so

we need to propagate updates in a way that temporal inconsistencies are not noticed

... however

this may degrade performance severely
Consistency models
Consistency model

- It is a contract between processes and a data store
  - Processes agree to follow certain rules, and the store promises to work correctly (Tanenbaum and van Steen, 2002)
  - What to expect when reading and updating shared data (while others do the same)
  - Models restrict the values that a read can return.
    - Minor restrictions’ models are easy to use but have low performance
    - Major restrictions’ models are hard to use but offer better performance
Types of Consistency Models

Consistency models can be divided into two types:

- **Data-Centric Consistency Models**
  - These models define how the data updates are propagated across the replicas to keep them consistent.

- **Client-Centric Consistency Models**
  - These models assume that clients connect to different replicas at different times.
  - The models ensure that whenever a client connects to a replica, the replica is brought up-to-date with the replica that the client accessed previously.
Data-centric consistency models
Introduction

- These models provide a system wide consistent view of the data store
  - Concurrent processes can simultaneously update the data store
  - A data store is distributed across a number of machines
  - Writes are propagated to other replicas
- These models are concerned with consistently ordering operations to the data store
Strict consistency

- Every read of $x$ returns a value corresponding to the result of the most recent write to $x$
- **True** replication transparency, every process receives a response that is consistent with the real time
- All writes are instantaneously visible to all processes
Strict consistency

- Each operation is stamped with a global wall-clock time

- Rules:
  - Rule 1: Each read gets the latest written value
  - Rule 2: All operations at one CPU are executed in order of their timestamps
Example

\[
\begin{array}{ll}
A: & W(x) \ a \\
B: & R(x) \ a
\end{array}
\]

\[
\begin{array}{ll}
A: & W(x) \ a \\
B: & R(x) \ NIL \ R(x) \ a
\end{array}
\]

Strictly consistent

Not strictly consistent

In general, \(A: \text{write}_t(x,a)\) then \(B: \text{read}_{t'}(x,a)\); \(t' > t\)

(regardless on the number of replicas of \(x\))
- So, strict consistency has very intuitive behavior
  - Essentially, the same semantic as on a uniprocessor!
- But how to implement it efficiently?
Implementing Strict Consistency

- To achieve, one would need to ensure:
  - Each read must be aware of, and wait for, each write
  - Real-time clocks are strictly synchronized...

- Unfortunately:
  - Time between instructions $< <$ speed-of-light...
  - Real-clock synchronization is tough

- So, strict consistency is tough to implement efficiently
Linearizability

- Interleaving of reads and writes into a single total order that respects the local ordering of the operations of every process (i.e., program order must be maintained)
  - A trace is consistent when every read returns the latest write preceding the read

- A trace is linearizable when
  - It is consistent
  - If $t_1$, $t_2$ are the times at which $p_i$ and $p_j$ perform operations, and $t_1 < t_2$, then the consistent trace must satisfy the condition that $t_1 < t_2$
Example

<table>
<thead>
<tr>
<th>A:</th>
<th>W(x) 1</th>
<th>R(y) 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>B:</td>
<td>W(y) 1</td>
<td>R(x) 1</td>
</tr>
</tbody>
</table>

The linearizable trace is $W_A(x,1), W_B(y,1), R_A(y,1), R_B(x,1)$
Sequential consistency

“\textit{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}” (Lamport 1979)

- Is not concerned with real time
- All processes see the same interleaving of operations
- Requires that interleaving preserving local temporal order of reads and writes are consistent traces
Sequential consistency example

**Sequence of operations:**

\[ W_2(x)b, R_3(x)b, R_4(x)b, W_1(x)a, R_3(x)a, R_4(x)a \]

**Sequentially consistent**

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

**Not sequentially consistent**

**Must be seen in the same order by all processes**

Figure adapted from Tanenbaum & Van Steen, Distributed Systems: Principles and Paradigms, (c) 2002 Prentice-Hall, Inc.- based on Figure 6.6
Sequential consistency Rules

- There exists a total ordering of operations such that
  - Rule 1: Each machine’s own ops appear in order
  - Rule 2: All machines see results according to total order (i.e. reads see most recent writes)
More Examples

What’s a global sequential order that can explain these results?

wall-clock ordering

This was also strictly

wall-clock time

P1: w(x)a

P2: w(x)b

P3: r(x)b r(x)b

P4: r(x)b r(x)b

What’s a global sequential order that can explain these results?

w(x)a, r(x)a, w(x)b, r(x)b, ...

This wasn’t strictly
More Examples

No global ordering can explain these results...
=> not seq. consistent

No global sequential global ordering can explain these results...
E.g.: the following global ordering doesn’t preserve P1’s ordering
w(x)c, r(x)c, w(x)a, r(x)a, w(x)b, …
Seq. Consistency Is Easier To Implement Efficiently

- No notion of real time
- System has some leeway in how it interleaves different machines' ops
  - Not forced to order by op start time, as in strict consistency
- Performance is still not great
  - Once a machine's write completes, other machines' reads must see new data
  - Thus communication cannot be omitted or much delayed
  - Thus either reads or writes (or both) will be expensive
Sequential Consistency Requirements

- Each processor issues requests in the order specified by the program
  - Do not issue the next request unless the previous one has finished

- Requests to an individual memory location (storage object) are served from a single FIFO queue.
  - Writes occur in a single order
  - Once a read observes the effect of a write, it’s ordered behind that write
Causal consistency

- All writes that are potentially causally related must be seen by every process in the same order, and reads must be consistent with this order.
- Writes that are not causally related to one another (concurrent) can be seen in any order.
- No constraints on the order of values read by a process if writes are not causally related.
  - Reads are fresh only w.r.t. the writes that they are causally dependent on.
### Example

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

- **Causally consistent**

\[
W_1(x)a \rightarrow R_2(x)a \rightarrow W_2(x)b \\
W_1(x)a \rightarrow W_1(x)c
\]
Example

**Data centric consistency**

Only per-process ordering restrictions:
- $w(x)b < r(x)b$; $r(x)b < r(x)a$; …
- $w(x)a \parallel w(x)b$, hence they can be seen in $\neq$ orders by $\neq$ processes

This wasn’t sequentially

Having read $c$ ($r(x)c$), $P3$ must continue to read $c$ or some newer value (perhaps $b$), but can’t go back to $a$, b/c $w(x)c$ was conditional upon $w(x)a$ having finished
More Examples

---

P1: \( w(x)a \)

P2: \( w(x)b \)

P3: \( r(x)b \quad r(x)a \)

P4: \( r(x)a \quad r(x)b \)

---

w(x)b is causally-related on r(x)a, which is causally-related on w(x)a. Therefore, system must enforce \( w(x)a < w(x)b \) ordering. But P3 violates that ordering, b/c it reads a after reading b.
Why Causal Consistency?

- Causal consistency is strictly weaker than sequential consistency and can give weird results, as you’ve seen
  - If system is sequentially consistent => it is also causally consistent

- BUT: it also offers more possibilities for concurrency
  - Concurrent operations (which are not causally-dependent) can be executed in different orders by different people
  - In contrast, with sequential consistency, you need to enforce a global ordering of all operations
  - Hence, one can get better performance than sequential

- Not very popular in industry
<table>
<thead>
<tr>
<th>Consistency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strict</td>
<td>Absolute time ordering on all shared accesses, thought to be impossible to implement it in distributed systems until recently. Example System: Google's Spanner</td>
</tr>
<tr>
<td>Linearizability</td>
<td>All processes see all shared accesses in the same order. Accesses are ordered based on a global timestamp. Good for reasoning about correctness of concurrent programs but not really used for building programs</td>
</tr>
<tr>
<td>Sequential</td>
<td>All processes see all shared accesses in the same order. Accesses are not ordered in time. Feasible and popular but has poor performance Example: Microsoft’s Niobe DFS</td>
</tr>
<tr>
<td>Causal</td>
<td>All processes see causally-related shared accesses in the same order. There is no globally agreed upon view of the order of operations. Example: COPS and Eiger</td>
</tr>
</tbody>
</table>

Figure adapted from Tanenbaum & Van Steen, Distributed Systems: Principles and Paradigms, (c) 2002 Prentice-Hall, Inc.- based on Figure 6.18.a
Trade-offs in Maintaining Consistency

- Maintaining consistency should balance between the strictness of consistency versus efficiency
- Good-enough consistency depends on your application

Loose Consistency
- Easier to implement, and is efficient

Strict Consistency
- Generally hard to implement, and is inefficient
Applications that Can Use Data-centric Models

Data-centric models are applicable when many processes are concurrently updating the data-store.

But, do all applications need all replicas to be consistent?

Data-Centric Consistency Model is too strict when:
- One client process updates the data
- Other processes read the data, and are OK with reasonably stale data
Client-centric consistency models
Motivation

- Data stores characterized by lack of simultaneous updates (or updates that are easily resolved)
  - Read-Write, Write-Read, Read-Read, **Write-Write**
  - Most operations involve reading data
- If updates are infrequent, eventually all replicas will obtain the update and become identical
  - Good if clients always access the same replica
- Predominant case for current large-scale Internet services
  - CAP theorem (Consistency, Availability, Partition tolerable)
Eventual consistency

- Maintains consistency for *individual clients*, not considering concurrent access by different clients.
- Ensure that replicas are brought up to date with data that has been manipulated by a client and that probably resides at another replica sites:
  - If there are no updates, eventually all replicas will be consistent.
  - Easier if clients access a single replica (more difficult if clients access different replicas over a short period of time).
  - Delay resolving conflicts, but updates are guaranteed to propagate to all replicas.

Several variations ...
Client centric consistency

- Clients are unaware of which replica they are accessing
- Clients may access different replicas
- Updates need to be propagated or otherwise there is inconsistent behavior
- Avoid write-write conflicts if data objects have a single owner (that can update the object)

Figure adapted from Tanenbaum & Van Steen, Distributed Systems: Principles and Paradigms, (c) 2002 Prentice-Hall, Inc.- based on Figure 6.19
Notation

- $X_i[t]$: data item $X$, at replica $L_i$, at time $t$
- $WS(X_i[t])$: Writing set, i.e., series of write operations until $X$ is at version $[t]$
- $WS(X_i[t_1]; X_j[t_2])$: Operations in $WS(X_i[t])$ were also made at replica copy $j$ at time $t_2$
Monotonic-read consistency

- If a process has seen a value of (data item) $x$ at a certain time, it will never see an older version of $x$ at a later time.

<table>
<thead>
<tr>
<th>L1:</th>
<th>WS($x_1$)</th>
<th>R($x_1$)</th>
<th>L1:</th>
<th>WS($x_1$)</th>
<th>R($x_1$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2:</td>
<td>WS($x_1$; $x_2$)</td>
<td>R($x_2$)</td>
<td>L2:</td>
<td>WS($x_2$)</td>
<td>R($x_2$)</td>
</tr>
</tbody>
</table>

A monotonic-read consistent data store

A not monotonic reads data store

The state has been copied to L2

Only the state in L2 is considered
Monotonic-write consistency

- A write to data item \( x \) is completed before any successive write to \( x \) by the same process.

<table>
<thead>
<tr>
<th>L1:</th>
<th>W(( x_1 ))</th>
<th>------</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2:</td>
<td>WS(( x_1 ))</td>
<td>( W( x_2 ) )</td>
</tr>
</tbody>
</table>

A monotonic-write consistent data store

The last write is reflected at L2

<table>
<thead>
<tr>
<th>L1:</th>
<th>W(( x_1 ))</th>
<th>------</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2:</td>
<td>( W( x_2 ) )</td>
<td></td>
</tr>
</tbody>
</table>

A Not monotonic-write data store

The latest write is not updated at L2
Read-your-writes consistency

- A process will never see a previous value of $x$ after a write to that data item $x$

L1: $W(x_1)$

L2: $WS(x_1; x_2)$ $R(x_2)$

A data store that provides read-your-writes

L2 performs updates to the last write

L1: $W(x_1)$

L2: $WS(x_2)$ $R(x_2)$

A data store that does not

L2 is not updated to the last write
Write-follow-reads consistency

- A write to $x$ following a previous read by the same process, is guaranteed to take place on the same or a more recent value of $x$ that was read.

<table>
<thead>
<tr>
<th>L1:</th>
<th>WS($x_1$)</th>
<th>R($x_1$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2:</td>
<td>WS($x_1$; $x_2$)</td>
<td>W($x_2$)</td>
</tr>
</tbody>
</table>

A writes-follow-reads consistent data store

Write operations are moved to L2

<table>
<thead>
<tr>
<th>L1:</th>
<th>WS($x_1$)</th>
<th>R($x_1$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2:</td>
<td>WS($x_2$)</td>
<td>W($x_2$)</td>
</tr>
</tbody>
</table>

A data store that does not

The writes from L2 are not consistent with those in L1
Consistency protocols
Consistency protocols

- Describes an implementation of a specific consistency model
- Sequential consistency
  - Passive replication  ➔ remote-write protocols and local-write protocols (primary-based)
  - Active replication  ➔ sequencer and quorum-based protocols
Primary-based protocol: remote-write

- Updates are blocking operations
- non-blocking operations improve performance but, problem ⇒ Fault tolerance

W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

Figure adapted from Tanenbaum & Van Steen, Distributed Systems: Principles and Paradigms, (c) 2002 Prentice-Hall, Inc.- based on Figure 6.28
Primary-based protocol: local-write

- Primary migrates between processes that wish to perform an operation
- Optimization → carry out multiple successive writes locally
  - But the requests need to be non-blocking

W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read

Figure adapted from Tanenbaum & Van Steen, Distributed Systems: Principles and Paradigms, (c) 2002 Prentice-Hall, Inc.- based on Figure 6.30
Quorum-based protocols - 1

- Assign a number of votes to each replica
- Let $N$ be the total number of votes
- Define $R =$ read quorum, $W =$ write quorum
  - $R + W > N$
  - $W > N/2$
- 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)
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 protocols - 3**

- **ROWA**: $R=1$, $W=N$
  - Fast reads, slow writes (and easily blocked)
- **RAWO**: $R=N$, $W=1$
  - Fast writes, slow reads (and easily blocked)
- **Majority**: $R=W=N/2+1$
  - Both moderately slow, but extremely high availability
- **Weighted voting**
  - Give more votes to “better” replicas
Reading

- Required reading:
  - Drop me an email if you have any trouble with this paper
Suggested readings:

- Google's Spanner paper
  - http://research.google.com/archive/spanner.html
- This takes you out of the comfort zone, heck, it makes me nervous to read
  - But you are better than your teachers!
  - If you try reading it and find issues, drop me an email and we can sit together
  - You can watch the presentation too from the link above!
Office hours for Replication and Consistency

- Wednesday, 13:30 to 15:30
- Please drop me an email if you want to drop by some other time!