Replication

Consistency Models

Pedro F. Souto (pfs@fe.up.pt)

May 24, 2015

Roadmap

Replication and Consistency

Sequencial Consistency

Weak Consistency Models

Implementation of Session-based Consistency

Replication for Performance

Replication (1/2)

What is? The use of multiple copies of data or services (and associated state)

Why? Mainly for 3 reasons:

Availabilitiy the service may be provided, even if some replicas fail

Reliability the service state may survive, even if the state kept by some replicas is lost (e.g. because of an earth quake)

Performance/scalability the service may be able to service more requests per time unit, by sharing the load among different replicas

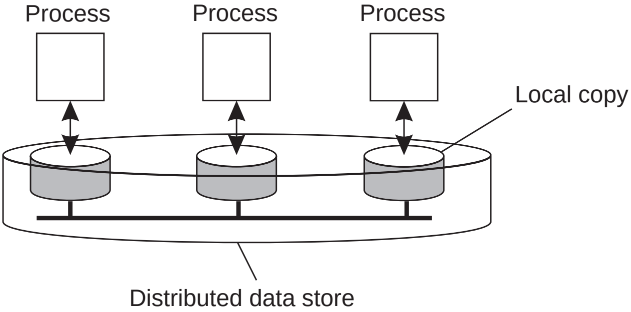
Replication (2/2)

- All replication protocols studied so far
  - State machine replication with Paxos
  - Primary backup replication with VSC
  - Quorum-based with transactions

  are targeted for **fault tolerance NOT performance**
- They require synchronization among all replicas to ensure that all replicas apply all operations in the same order
  - Most likely, they are less performant than using a single service copy
- Basically, they all ensure the **sequential consistency model**

Replication and Consistency

- When we have multiple replicas their value may diverge, i.e. **consistency** issues may arise
- Ideally, a replicated service should behave as a non-replicated service



- As a minimum, the value of all replicas should converge in the absence of further updates. But:
  - What will the final value be?
  - Which values can be returned by the service operations?

Consistency: Concurrency vs. Replication

- Multiple threads accessing **concurrently** shared data may lead to consistency issues, as studied in OS classes
- A single thread executing different operations on **different replicas** may also cause consistency issues

```
Repl. 1                     Repl. 2
(5)   // initial value      (5)     // initial value
x = 2;
(2)   // final value
                            x += 3;
                            (8)    // final value
```

- In distributed systems, consistency issues are harder to deal with because there is often concurrency and replication
  - Multiprocessor systems also have replication induced consistency issues
    - The processor caches may also store replicas of data objects

Consistency Model: What is?

Definition A **consistency model** is a contract between the developers and the users of a system specifying:

- the **result** of the system operations in the presence of replication
- the **rules** the users must follow when accessing the system

Note This is akin to requiring mutual exclusion in the access to critical sections to ensure consistency in concurrent systems

- I.e. that a concurrent system behave as one with a single thread of execution, except for the performance

- Developers should choose/design a consistency model intuitive and with good performance
- Users need to follow the rules mandated by the model, or else the system may behave unexpectedly

Roadmap

Sequential Consistency Model (Lamport 79)

Definition An execution is **sequential consistent iff** it is identical to a sequential execution of all the operations in that execution such that

- all operations executed by any thread, appear in the order in which they were executed by the corresponding thread

Observation This is the model provided by a multi-threaded system on a uniprocessor

Counter-example Consider the following operations executed on two replicas of variables x and y, whose initial values are 2 and 3, respectively

```
Répl. 1       Répl. 2
(2,3)         (2,3)       /* Initial values */
x = y + 2;    y = x + 3;
(5,5)         (5,5)       /* Final values */
```

If the two operations are executed sequentially, the final result cannot be $(5, 5)$

One-copy Serializability (Transaction-based Systems)

Definition The execution of a set of transactions is **one-copy serializable** **iff** its outcome is similar to the execution of those transactions in a **single** copy

Observation 1 Serializability used to be the most common consistency model used in transaction-based systems

- DB systems nowadays provide weaker consistency models to achieve higher performance

Observation 2 This is essentially the sequential consistency model, when the operations executed by all processors are transactions

- The isolation property ensures that the outcome of the concurrent execution of a set of transactions is equal to some sequential execution of those transactions

Observation 3 (Herlihy ... sort of) Whereas

Serializability Was proposed for databases, where there is a need to preserve complex application-specific invariants

Sequential consistency Was proposed for multiprocessing, where programmers are expected to reason about concurrency

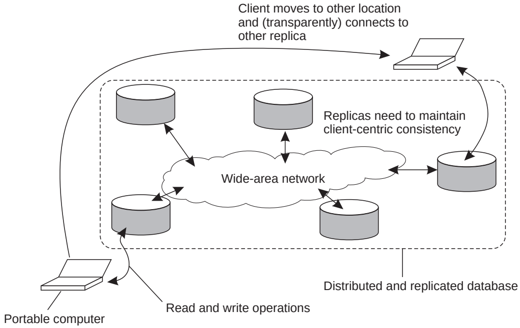
Roadmap

Weak Consistency: Rationale

- Strong consistency models usually provide an environment that facilitates the tasks of the programmers/users
- However, their implementation usually requires tight synchronization among replicas, this adversely affects:
  - **Scalability (performance)**
  - **Availability**
- Weakly consistency models strive to:
  - Improve scalability and availability
  - While providing set of guarantees that is useful for their users
    - Weak consistency models are usually application domain dependent

Session-based Consistency Models

- These models were designed for **mobile computing** applications



- They consider a basic **read/write** interface

Session is an abstraction for a sequence of read and write operations performed on a data store by an application.

- It is not intended to be an atomic transaction
- It is intended to represent the execution of an application, possibly multiple execution instances

Replicated Data Store Model/Assumptions (1/2)

- The data store is composed of a set of servers each holding a full copy of the data stored
- Clients execute applications that try to access the data in the store
- The data store offers two main operations:

  Read that return the values of data items in the store

  Write that updates (creates/modifies/deletes) data items in the store

  - Each write has a globally unique identifier, the **WID**

  Each server applies writes in sequence, which determines its state

  $DS(S,t)$ represents the state of the store in server S at time t, as a sequence of writes

  - If there is no confusion, we can drop the t: $DS(S)$
  - $DS(S,t)$ is just a representation of the state, the actual order of the writes may differ, as long as the state is not affected

Replicated Data Store Model/Assumptions (2/2)

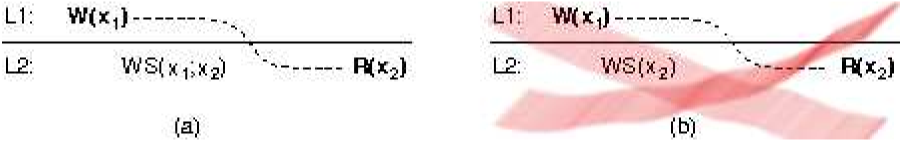
- The data store is assumed to ensure **eventual consistency**, i.e.
  - In the absence of updates, the state of the different replicas converges to a single value
- Eventual consistency can be achieved by:
  1. Total propagation of the writes
  2. Consistent ordering of the writes, i.e. by ensuring that **non-commutative** writes are applied in the same order in all replicas

     WriteOrder(W1,W2) is a predicate that is true if W2 should be applied only if W1 was previously applied

     - Therefore, it is assumed that the system hones the *WriteOrder*() predicates
- Eventual consistency is a liveness property
  - It is very hard to make it precise

Read Your Writes

Read Your Writes read operations on a variable x reflect previous writes on that variable



Notation

x_i Value of variable x at site i, after the operation

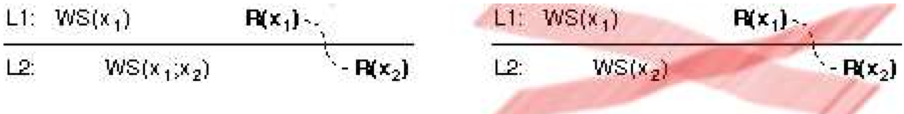
x_i's relevant-writes set ($WS(x_i)$) (minimum) set of write operations that lead to x_i

$WS(x_i, x_j)$ $WS(x_i) \subset WS(x_j)$, i.e. the write operations that lead to x_j comprise those that lead to x_i

Example Update a Web page and ensure that a browser shows the most recent version (and not the version previously cached)

Monotonic Reads

Monotonic Reads Successive reads reflect a non-decreasing sequence of writes

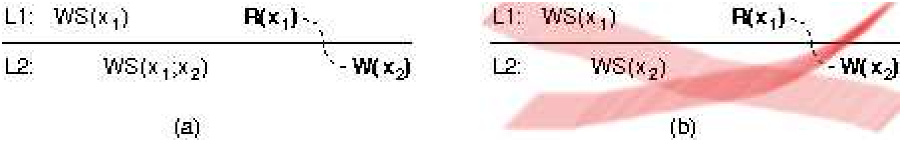


(a) (b)

Example Consider successive reads of a mail box replicated on several servers. Any read, must show all messages shown in previous reads (assuming that no message was deleted).

Writes Follow Reads

Writes Follow Reads (WFR) writes are propagated after writes "seen" by previous reads in the same session



Observation Writes seen by a read in a session, i.e. the relevant writes, **must occur on all servers** before writes that occur in the session after that read

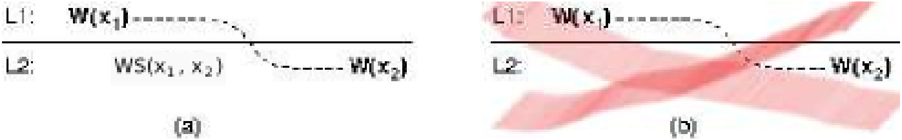
- In contrast with the previous rules, this may affect other clients, even those that have not requested consistency guarantees

Example Consider a bibliographic database replicated on several servers. A user detects an error in one bibliographic element and corrects it by some write operation

- WFR ensures that the correction is applied only after the buggy update has been applied

Monotonic Writes

Monotonic Writes Writes are propagated after previous writes in the same session



Observ. I.e. writes in a session are ordered

Example Consider a replicated document, which is successively updated. An update must be applied to a replica only if all previous updates have been applied.

Session Consistency: Final Remarks

Guarantees

Session consistency properties *are "guaranteed" in the sense that either the storage system ensures them for each read and write operation belonging to a session, or else it informs the calling application that the guarantee cannot be met* (citation)

Food for Thought

Sequential Consistency What is the relationship, if any, between the client-centric consistency models and sequential consistency?

Causal Consistency Causal consistency is a consistency model that ensures that operations are applied in causal order

- Here causality dependencies are induced by reading variables, rather than by receiving messages
- Is there any relationship between causal consistency and the session-based consistency guarantees?

Roadmap

State

WID Each write has an identifier, assigned by the server that **accepts** the write, i.e. the first server to apply the operation

Server keeps one set of WIDs

- **Server Write-Set(WS)** with the WIDs of all write operations that lead to the state of the replica

Client (stub) keeps 2 sets of WIDs for each **session**:

- **Read-Set (CRS)** with the WIDs of all the **relevant writes**, i.e. the writes on which the values read during the session depend
  - In each read, in addition to the value returned, the server has to return the set of WIDs of relevant writes for the returned value
- **Write-Set (CWS)** with the WIDs of all the writes performed during the session
  - In each write operation, the server returns the WID assigned

Implementation of **Read your Writes**

Client

On write Add the WID assigned by the server to the CWS

On read Two alternatives

1. Requests the server's WS
   - If $CWS \not\subseteq WS$, client has to try another server
2. Sends the server its CWS

Server In case 2 above, if $CWS \not\subseteq WS$, there are 2 alternatives

1. Forward the request to another server
2. Update its state so that $CWS \subseteq WS$ before returning the value read

Note The implementation of **monotonic reads** is similar. Except that, on reads:

- Compares the WS with the CRS, instead of the CWS
- Needs to update the CRS with the set of relevant writes returned

Implementation of **Writes follow Reads**

Client

On read Add the set of relevant writes returned by the server to the CRS

On write Two alternatives

1. Requests the server's WS
   - If $CRS \not\subseteq WS$, client has to try another server
2. Sends the server its CRS

Server In case 2 above, if $CRS \not\subseteq WS$, there are 2 alternatives

1. Forward the request to another server
2. Update its state so that $CRS \subseteq WS$ before performing the write
   - The propagation protocol of writes among servers must preserve the order of write operations

Note The implementation of **monotonic writes** is similar. Except on write operations:

- Compares the *WS* with the *CWS* rather than the *CRS*
- Needs to add the *WID* assigned by the server to the *CWS*

Version Vectors (1/2)

Problem The implementation of the sets directly is not efficient

- They increase indefinitely
- Some of them must be exchanged between clients and servers

Solution Use **version vectors**, which are similar to vector clocks, with one element per server

- Each server has a version counter for the replica it manages
  - This counter is incremented every time the server **accepts** a write

Version Vectors (2/2)

Server i Keeps a version vector, V_i, that represents the state of the replica it manages

$V_i[j]$ is the number of **writes accepted by server j** applied by server i to its replica

Obs. $V_i[i]$ is the number of writes server i itself accepted

Obs. Operations accepted by remote servers must be applied according to the $WriteOrder()$ predicates

Client i Keeps two vectors per session

W_i to keep track of the its operations

R_i to keep track of the relevant writes, i.e. the writes on which its read operations depend

Implementation of **Read Your Writes** with VV

On write

Server i

- Updates $V_i[i]$ (and performs write)
- Returns $WID = V_i[i]$ to client

Client j updates its write vector $W_j[i] = WID$

On read (assuming checking @ server)

Client j

1. Sends its W_j (in the read request) to the server
2. Updates its read vector, $R_j = max(R_j, RW)$, where RW is the relevants writes vector returned by the read

Server i

- Checks if V_i **dominates** W_j, i.e. $V_i[k] \geq W_j[k]$, for all k. If not:
  - Either updates its replica
  - Or forwards the request to another server

Note Implementation of **monotonic reads** is similar

Implementation of **Write Follows Reads** with VV

On read

Server i returns also vector with the relevant writes (RW)

- A possible approximation is V_i, but
  - This may create unwarranted dependencies, and make it harder finding a server sufficiently up-to-date

Client j

- Updates its read vector $R_j = max(R_j, RW)$

On write

Client j (assuming check @ client)

Before sending write Requests the server its VV

- If $V_i \not\geq R_j$, must try another server

Upon write return Updates its write vector: $W_j[i] = WID$

Server i

- Updates $V_i[i]$ (and performs write operation)
- Returns $WID = V_i[i]$

Note Implementation of **monotonic writes** is similar.

Performance and Availability Considerations

- The less a client changes the server during a session the best
  - Requests after the first, need no checking of whether the server is up-to-date
- Caching at the client may also improve both:
  - Performance
  - Availability
- However, if on a client there is a single cache shared among different applications using different sessions guarantees:
  - An application can use cached data only if it satisfies its session guarantees

Roadmap

Replication for Performance: Particular Cases

Caches

- Usually managed by the clients
- Try to take advantage of the **locality of reference**
  - Accesses other than the first one to some data uses the cached copy, available locally

Proxy servers

- Also keep a cache, but it is shared among several clients
- As usual, rely on the locality of reference but also on the **homogeneity** of the client population
- Allow, or facilitate:
  - Reduce the response time (latency)
  - Reduce the load on the (original) servers
  - Use the network bandwidth more efficiently
  - Implement global security policies

Further Reading

- Chapter 7 (Consistency and Replication) Tanenbaum and van Steen, *Distributed Systems*, 2nd Ed.
  - Section 7.1: Introduction
  - Section 7.2: Data-centric consistency
  - Section 7.3: Client-centric consistency
  - Section 7.5.5: Implementing client-centric consistency
- L. Lamport, *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*, IEEE Transactions on Computers C-28, 9 (September 1979) 690-691
- D. Terry et al., "Session Guarantees for Weakly Consistent Replicated Data", in Proc. 3rd International Conference on Parallel and Distributed Information Systems, 1994, pp.140-149.