Replica Management

CS425 /ECE428 – DISTRIBUTED SYSTEMS – SPRING 2019

Material derived from slides by I. Gupta, M. Harandi, J. Hou, S. Mitra, K. Nahrstedt, N. Vaidya
Objective

Understand replication management

- Goals
- Model
- Group communication & views
- Consistency
- Passive vs. active

Readings

- §15.1–15.3 (4th ed)
- §18.1–18.3 (5th ed)
Replication

Enhances a service by replicating data

- **Increased Availability**
  - Of service. When servers fail or when the network is partitioned.

- **Fault Tolerance**
  - Under the fail-stop model, if up to $f$ of $f+1$ servers crash, at least one is alive.
  - *(later: Byzantine faults, survive $f$ faults with $3f+1$ servers)*

- **Load Balancing**
  - One approach: Multiple server IPs can be assigned to the same name in DNS, which returns answers round-robin.

\[
P: \text{probability that one server fails} = 1 - P = \text{availability of service. e.g. } P = 5\% \Rightarrow \text{service is available 95\% of the time.}
\]

\[
P^n: \text{probability that } n \text{ servers fail} = 1 - P^n = \text{availability of service. e.g. } P = 5\%, n = 3 \Rightarrow \text{service available 99.875\% of the time}
\]
Goals of Replication

Replication Transparency
- User/client need not know that multiple physical copies of data exist.

Replication Consistency
- Data is consistent on all of the replicas (or is converging towards becoming consistent)
Replication Management

Request Communication
- Requests can be made to a single RM or to multiple RMs

Coordination: The RMs decide
- whether the request is to be applied
- the order of requests
  - **FIFO ordering:** If a FE issues r then r’, then any correct RM handles r and then r’.
  - **Causal ordering:** If the issue of r "happened before" the issue of r’, then any correct RM handles r and then r’.
  - **Total ordering:** If a correct RM handles r and then r’, then any correct RM handles r and then r’.

Execution: The RMs execute the request (often they do this tentatively – why?).
Replication Management

Agreement: The RMs attempt to reach consensus on the effect of the request.
- E.g., Two phase commit through a coordinator
- If this succeeds, effect of request is made permanent

Response
- One or more RMs responds to the front end.
- The first response to arrive is good enough because all the RMs will return the same answer.
- Thus each RM is a replicated state machine
  - "Multiple copies of the same State Machine begun in the Start state, and receiving the same Inputs in the same order will arrive at the same State having generated the same Outputs." [Wikipedia, Schneider 90]
Need consistent updates to all copies of object
- Linearizability
- Sequential consistency
Linearizability

Let the sequence of read and update operations that client $i$ performs in some execution be $o_{i1}, o_{i2}, \ldots$

- "Program order" for the client

A replicated shared object service is linearizable if for any execution (real), there is some interleaving of operations (virtual) issued by all clients that:

- meets the specification of a single correct copy of objects
- is consistent with the real times at which each operation occurred during the execution

Main goal: any client will see (at any point of time) a copy of the object that is correct and consistent
Sequential Consistency

The real-time requirement of linearizability is hard

A less strict criterion is **sequential consistency**: A replicated shared object service is sequentially consistent if for any execution (real), there is some interleaving of clients' operations (virtual) that:

- meets the specification of a single correct copy of objects
- is consistent with the program order in which each individual client executes those operations.

This approach does not require absolute time or total order. Only that for each client the order in the sequence be consistent with that client's program order (~ FIFO).

Linearizability implies sequential consistency. Not vice-versa!

Challenge with guaranteeing seq. cons.?

- Ensuring that all replicas of an object are consistent.
Passive (Primary-Backup) Replication

Request Communication: the request is issued to the primary RM and carries a unique request id.

Coordination: Primary takes requests atomically, in order, checks id (resends response if not new id.)

Execution: Primary executes & stores the response

Agreement: If update, primary sends updated state/result, req-id and response to all backup RMs

Response: primary sends result to the front end
Fault Tolerance in Passive Replication

What to do if a primary fails?
- Use leader election to choose a new primary

How do we make sure new primary has all committed data?
- RAFT: commit requires ACK from majority of followers, election restriction ensures log entries don’t get lost
- Alternative: view-synchronous communication
Views

A **group membership service** maintains group *views*: a list of current members
- Each member maintains its own *local* view

A view $V_p(g)$ is process $p$'s understanding of its group (list of members)
- Example: $V_{p,0}(g) = \{p\}$, $V_{p,1}(g) = \{p, q\}$, $V_{p,2}(g) = \{p, q, r\}$, $V_{p,3}(g) = \{p, r\}$
- The second subscript indicates the "view number" received at $p$ (≈ terms in Raft)

A new group view is disseminated, throughout the group, whenever a member joins or leaves.
- Member detecting failure of another member multicasts a "view change" message
- Likewise for newly joined member
Views

An event is said to occur in a view $v_{p,i}(g)$ if the event occurs at $p$, and at the time of event occurrence, $p$ has delivered $v_{p,i}(g)$ but has not yet delivered $v_{p,i+1}(g)$.

Requirements for view delivery

- **Order:** If $p$ delivers $v_i(g)$ and then $v_{i+1}(g)$, then no other process $q$ delivers $v_{i+1}(g)$ before $v_i(g)$.
- **Integrity:** If $p$ delivers $v_i(g)$, then $p$ is in all $v_{*,i}(g)$.
- **Non-triviality:** If process $q$ joins a group and becomes reachable from process $p$, then eventually, $q$ will always be present in the views that delivered at $p$. 
Multicast

The following guarantees are provided for multicast messages:

- **Integrity**: If \( p \) delivered message \( m \), \( p \) will not deliver \( m \) again. Also \( p \in \text{group}(m) \), i.e., \( p \) is in the latest view.

- **Validity**: Correct processes always deliver all messages. That is, if \( p \) delivers message \( m \) in view \( v(g) \), and some process \( q \in v(g) \) does not deliver \( m \) in view \( v(g) \), then the next view \( v'(g) \) delivered at \( p \) will not include \( q \).

- **Agreement**: Correct processes deliver the same sequence of views, and the same set of messages in any view.

  - if \( p \) delivers \( m \) in \( V \), and then delivers \( V' \), then all processes in \( V \cap V' \) deliver \( m \) in view \( V \).
Example: View Synchronous Communication
State Transfer

When a new process joins the group, state transfer may be needed (at view delivery point) to bring it up to date

- "state" may be list of all messages delivered so far (wasteful)
- "state" could be list of current server object values (e.g., a bank database) – could be large
- Important to optimize this state transfer
... back to passive replication

Primary uses view-synchronous multicast to send updates to backups
  ◦ Backups execute updates immediately

If primary fails, a view change can be used to select new primary
  ◦ No leader election necessary

May still need to wait for ACKs from backups before replying to front-end
  ◦ Otherwise updates can be lost
Active Replication

Request Communication: The request contains a unique identifier and is multicast to all by a **reliable totally-ordered** multicast.

Coordination: Group communication ensures that requests are delivered to each RM in the same order (but may be at different physical times!).

Execution: Each replica executes the request. (Correct replicas return same result since they are running the same program, i.e., they are replicated protocols or replicated state machines)

Agreement: No agreement phase is needed, because of multicast delivery semantics of requests

Response: Each replica sends response directly to FE
Fault tolerance and Ordering

Faults are tolerated automatically
- All remaining nodes see same set of messages

This system implements sequential consistency
- The total order ensures that all correct replica managers process the same set of requests in the same order.
- Each front end's requests are served in FIFO order (because the front end awaits a response before making the next request).

How many responses should FE wait for?
Transactions on Replicated Data

Client + front end

getBalance(A)

Replica managers

A

A

A

Client + front end

deposit(B,3);

Replica managers

B

B

B

B
One Copy Serialization

In a non-replicated system, transactions appear to be performed one at a time in some order. This is achieved by ensuring a serially equivalent interleaving of transaction operations.

**One-copy serializability:** The effect of transactions performed by clients on replicated objects should be the same as if they had been performed one at a time on a single set of objects (i.e., 1 replica per object).

- Equivalent to combining serial equivalence + replication transparency/consistency
Two Phase Commit Protocol For Transactions on Replicated Objects

Two level nested 2PC

In the first phase, the coordinator sends the canCommit? command to the participants, each of which then passes it onto the other RMs involved (e.g., by using view synchronous communication) and collects their replies before replying to the coordinator.

In the second phase, the coordinator sends the doCommit or doAbort request, which is passed onto the members of the groups of RMs.
Primary Copy Replication

For now, assume no crashes/failures

All the client requests are directed to a single primary RM.
Concurrency control is applied at the primary.

To commit a transaction, the primary communicates with the backup RMs and replies to the client.

View synchronous comm. gives \(\Rightarrow\) one-copy serializability

Disadvantage? Performance is low since primary RM is bottleneck.
Read One/Write All Replication

An FE (client front end) may communicate with any RM.

Every write operation must be performed at all of the RMs
  ◦ Each contacted RM sets a write lock on the object.

A read operation can be performed at any single RM
  ◦ A contacted RM sets a read lock on the object.

Consider pairs of conflicting operations of different transactions on the same object.
  ◦ Any pair of write operations will require locks at all of the RMs ➔ not allowed
  ◦ A read operation and a write operation will require conflicting locks at some RM ➔ not allowed
  ◦ One-copy serializability is achieved.

Disadvantage? Failures block the system (esp. writes).
Available Copies Replication

A client's read request on an object can be performed by any RM, but a client's update request must be performed across all available (i.e., non-faulty) RMs in the group.

As long as the set of available RMs does not change, local concurrency control achieves one-copy serializability in the same way as in read-one/write-all replication.

May not be true if RMs fail and recover during conflicting transactions.
The Impact of RM Failure

Assume that:
- RM X fails just after T has performed getBalance; and
- RM N fails just after U has performed getBalance.
- Both failures occur before any of the deposit()'s.

Subsequently:
- T's deposit will be performed at RM's M and P
- U's deposit will be performed at RM Y.

The concurrency control on A at RM X does not prevent transaction U from updating A at RM Y.

Solution: Must also serialize RM crashes and recoveries with respect to entire transactions.
Local Validation (using Our Example)

From T's perspective,
- T has read from an object at X ➔ X must have failed after T's operation.
- T observes the failure of N when it attempts to update the object B ➔ N's failure must be before T.
- Thus: N fails ➔ T reads object A at X; T writes objects B at M and P ➔ T commits ➔ X fails.

From U's perspective,
- Thus: X fails ➔ U reads object B at N; U writes object A at Y ➔ U commits ➔ N fails.

At the time T tries to commit,
- it first checks if N is still not available and if X, M and P are still available. Only then can T commit.
- If T commits, U's validation will fail because N has already failed.

Can be combined with 2PC.

Caveat: Local validation may not work if partitions occur in the network.
Network Partition

Client + front end
withdraw(B, 4)

Network partition

Client + front end
deposit(B, 3);

Replica managers
Dealing with Network Partitions

During a partition, pairs of conflicting transactions may have been allowed to execute in different partitions. The only choice is to take corrective action after the network has recovered

- Assumption: Partitions heal eventually

Abort one of the transactions after the partition has healed

Basic idea: allow operations to continue in partitions, but finalize and commit trans. only after partitions have healed

But to optimize performance, better to avoid executing operations that will eventually lead to aborts...how?
Quorum Approaches

Quorum approaches used to decide whether reads and writes are allowed.
There are two types: pessimistic quorums and optimistic quorums.

In the pessimistic quorum philosophy, updates are allowed only in a partition that has the majority of RM.
- Updates are then propagated to the other RM when the partition is repaired.
Static Quorums

The decision about how many RM\(s\) should be involved in an operation on replicated data is called Quorum selection.

Quorum rules state that:

- At least \(r\) replicas must be accessed for read.
- At least \(w\) replicas must be accessed for write.
- \(r + w > N\), where \(N\) is the number of replicas.
- \(w > N/2\).
- Each object has a version number or a consistent timestamp.

Static Quorum predefines \(r\) and \(w\), & is a pessimistic approach: if partition occurs, update will be possible in at most one partition.
Voting with Static Quorums

Modified quorum:
- Give different replicas different #'s of votes
- e.g., a cache replica may be given a 0 vote

with $r = w = 2$, Access time for write is 750 ms (parallel writes). Access time for read without cache is 750 ms. Access time for read with cache can be in the range 175ms to 825ms – why?

<table>
<thead>
<tr>
<th>Replica</th>
<th>Votes</th>
<th>Access Time</th>
<th>Version Check</th>
<th>P(Failure)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache</td>
<td>0</td>
<td>100ms</td>
<td>0ms</td>
<td>0%</td>
</tr>
<tr>
<td>Replica 1</td>
<td>1</td>
<td>750ms</td>
<td>75ms</td>
<td>1%</td>
</tr>
<tr>
<td>Replica 2</td>
<td>1</td>
<td>750ms</td>
<td>75ms</td>
<td>1%</td>
</tr>
<tr>
<td>Replica 3</td>
<td>1</td>
<td>750ms</td>
<td>75ms</td>
<td>1%</td>
</tr>
</tbody>
</table>
Summary

Transactions
Concurrency Control
Replicated Data
Replication with Transactions
Different types of replication
Quorums