Resilient Key Value Store Design

1. Introduction

The document describes the design of a replication based resilient key value store. Strong consistency and serializability of the map operations are ensured using a transactional memory mechanism.

2. Application Interface

Users are provided with a HashMap like APIs for storing and manipulating their data. One place creates the map, and passes it to other places that wish to use it as follows:

```
val mapName = “my_map”;
val map = DataStore.getInstance().makeResilientMap(mapName);

for ( p in Place.places()) at (p) async {
    map.put (“some key”, “some value”);
}
```

A user can also perform multiple operations atomically within a single transaction as follows:

```
for ( p in Place.places()) at (p) async {
    val txId = map.startTransaction();
    val x = map.get(txId, “X”);
    val y = map.get(txId, “Y”);
    map.put (txId, “Z”, x+y);
    map.commit(txId);
}
```

The commit operation might fail when places perform conflicting transactions. In that case, the application need to repeat the transaction[1].

3. Data Partitioning

The key/value records are partitioned among places. Currently, the number of partitions equals to Place.numPlaces().[2] Consistent hashing is used for determining the specific key partition.

4. Topology-Aware Replica Placement

Each partition is replicated in at least two places.
A PartitionTable at each place stores the Replicas (i.e. places) that store each partition.
A partition is not allowed to be replicated on places that exist in the same node.

Partition Id	The index of the three arrays represents the partition id 0	1	2	3	4	5	6	7	8
Replica-1	Place(0)	Place(1)	...	...	...	...	...	Place(7)	Place(8)
Replica-2	Place(1)	Place(2)	...	...	...	...	...	Place(8)	Place(0)
Replica-3	Place(2)	Place(3)	...	...	...	...	...	Place(0)	Place(1)

5. Topology-Aware Leaders Assignment

A leader and a deputy leader places are selected to handle the loss of replicas.
Using the topology information, the leader and deputy leader places are selected from different nodes.

1 Possible enhancement: perform automatic repetition upon failure by logging the issued map actions.
2 Possible enhancement: allow configuring some places as Spare.

The following figure shows the main design modules.

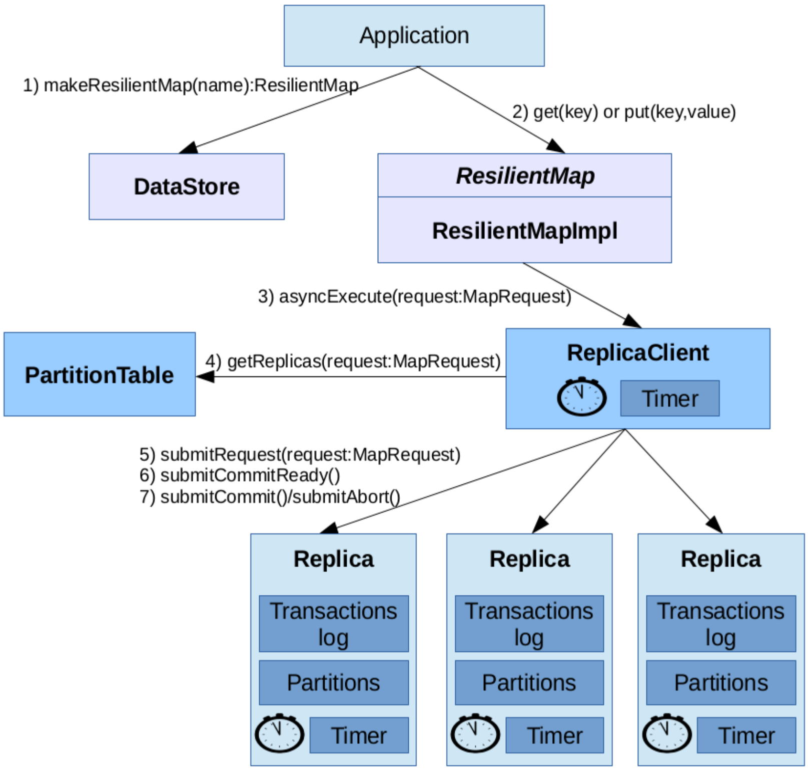


Fig. 1: Main design modules

6. Replica and ReplicaClient

Accessing the replicated key/value records is done using client-server protocols.
The Replica module is responsible for the server side operations, as it is the container of the actual data.

The ReplicaClient is the client side; it performs the map operations (e.g. get and put) on behalf of the application. Using the PartitionTable it identifies the replicas that can serve the application request and starts communicating with them using specific protocols that aim to ensure strong consistency.

Asynchronous communication: the communication between a Replica and ReplicaClient is asynchronous. However, responses are expected within a specific time period, otherwise, the involved transaction is aborted.

Dead Replica: A ReplicaClient checks for the status of Replicas at the following times:
- Before starting a transaction, and
- Periodically, while waiting for a response from a Replica

When it detects that one replica is dead, it notifies the leader place, terminates the transaction, and tries it again after some time.

Dead ReplicaClient: Each Replica periodically checks the status of ReplicaClients that has non-completed transactions. When a ReplicaClient is found dead, its transactions are aborted. The leader place is ***not*** notified.

7. Distributed Transactional Memory

In order to ensure that the replicas are always synchronized, updating the different replicas is done atomically within a single transaction.

The ACID properties are implemented as follows:
- **Atomicity**: after a transaction starts, it has to complete either by a commit() or an abort() call.
- **Consistency**: at commit time, when two transactions conflict, one of them will be forced to abort.
- **Isolation**: each transaction creates its own copy of the data. Transaction modifications are applied on the copied data and are not visible to other transactions.
- **Durability**: commited transactions are not allowed to abort.

Transaction Code Example:

```
1:    val mapName = "my_map";
2:    val map = DataStore.getInstance().makeResilientMap(mapName);
3:    for ( p in Place.places()) at (p) async {
4:        val txId = map.startTransaction();
5:        try {
6:            val x = map.get(txId, "X");
7:            val y = map.get(txId, "Y");
8:            map.put (txId, "Z", x+y);
9:            map.commit(txId);
10:       }
11:       catch(ex:Exception) {
12:           map.abort(txId)
13:       }
14:   }
```

Transaction Life Cycle:
1) Active
2) Ready to commit
3) Commited
4) Aborted

Starting a Transaction (Line 4): is a local operation. It only generates a globally unique transaction id.

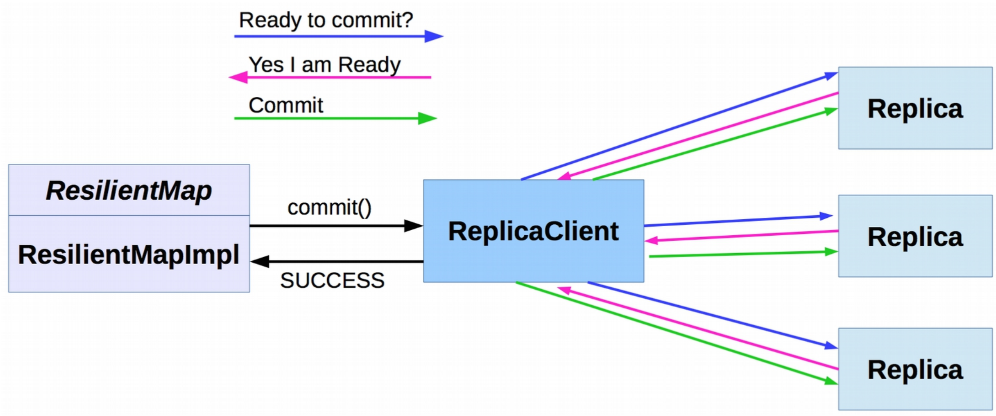
Replicas Transaction Logs: when a map operation is forwarded to a replica with a new transaction id, a new transaction is created with status (Active). A TransLog object is used to store the transaction information.

A TransLog object stores the following information for each accessed key.

TransKeyLog	
key	
initialVersion	The version of the value at transaction start time
value	A copy of the actual value. Updates are performed to this copy (not the actual value).
readOnly	Used for conflict validation.

	Read/Read → no conflict Read/Write or Write/Read or Write/Write → conflict

Distributed Commitment Protocol: we used the 2-phase commit protocol.



In the first phase, the ReplicaClient asks all Replicas to vote about their readiness to commit.
To answer this request, each Replica checks for the transaction conflicts.
If no conflicts exist, the Replica changes the status of the transaction from "Active" to "Ready to Commit", and sents a positive vote to the ReplicaClient.
If all Replicas send a positive vote, the ReplicaClient notifies the Replicas to commit.

Conflict Detection: the following rules are applied to determine a conflict:
1) A transaction with status 'Ready to Commit' has a higher priority than a transaction with status 'Active'.
→ if transaction A conflicts with Transaction B, while B is ready to commit, abort transaction A.

2) If the initial versions of a transaction values have changed, then abort the transaction. This happens when a previous transaction has committed after this transaction has started.

3) Conflicts with other 'Active' transactions are resolved by aborting all transactions except one. The transaction to remain is the one with maximum transaction Id.