Resilient Key Value Store Design

1. Introduction

The document describes the design of a replication based resilient key value store. Strong consistency and strict ordering of update operations on the different replicas 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 Replication

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 below 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.

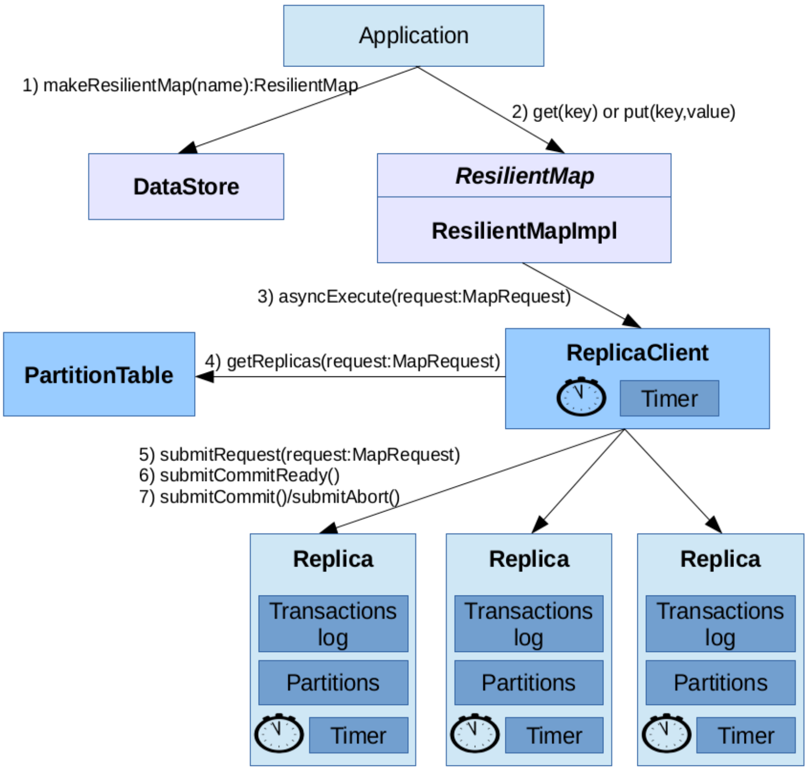

Fig. 1: Main design modules

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 replicas 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.

At the Replica side, a late response from a client results in aborting the transaction.
At the ReplicaClient side, a late response from one of the replicas results in aborting the transaction.

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 replica client is found dead, its transactions are aborted. The leader place is ***not*** notified.

7. Distributed Transactional Memory

- versioned values
- isolation (store the updates in a local copy)
- distributed commitment protocol
  – when a transaction is ready to commit
  ---- abort my self vs abort others