Active-Active Geo-Distribution (CRDTs-Based)

Active-Active Geo-Distributed topology is achieved by implementing CRDTs (conflict-free replicated data types) in Redis Enterprise using a global database that spans multiple clusters. This is called a “conflict-free replicated database” or “CRDB.”

CRDB provides three fundamental benefits over other geo-distributed solutions:

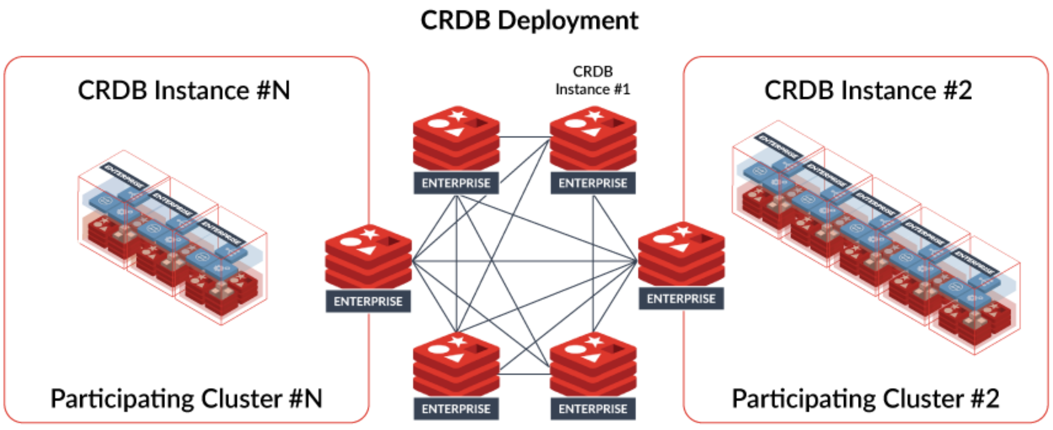
- It offers local latency on read and write operations, regardless of the number of geo-replicated regions and their distance from each other.
- It enables seamless conflict resolution (“conflict-free”) for simple as well as complex data types like those of Redis core.
- Even if the majority of geo-replicated regions in a CRDB (for example 3 out of 5) are down, the remaining geo-replicated regions are uninterrupted and can continue to handle read and write operations.

Deployments and topologies

A CRDB is a database that is created across multiple Redis Enterprise clusters, which typically reside in different data centers around the world. The database in each one of the participating clusters is called a “CRDB instance.” As long as a CRDB dataset fits into the CRDB instance memory, each CRDB instance can be configured differently—composed from a different number of shards and run on a different number or type of cluster nodes.

Furthermore, one CRDB instance may run on a multi-availability-zone (AZ) cluster configuration with high availability and data persistence enabled, while a second CRDB instance may run on a single-AZ cluster configuration without high availability or data persistence. This high level of flexibility lets you optimize infrastructure costs and database performance for your particular use case.

An application that uses a CRDB is connected to the local CRDB instance endpoint. Bi-directional replication is used between all CRDB instances in a mesh-like topology, i.e. all writes by the application to a local instance are replicated to all other instances, as illustrated here:



Note: Future releases of Redis Enterprise will feature additional topologies, such as ring.

High-level architecture

CRDT layer

The CRDB architecture is based on an alternative implementation of most Redis commands and data types (called CRDTs, as explained above). In Redis Enterprise, our CRDB implementation is based on a proprietary Redis module that is built with the Redis module data type API.

Read commands are handled locally using the local CRDB instance. The inherent consensus-free mechanism of the CRDT layer does not require “read repairs” common to other active-active implementations.

Write commands are processed in two steps, following the principles of operation-based CRDT:

1. In the *prepare* step, the user’s request is processed using the local CRDB instance and a resulting *effect* is created.
2. In the *effect* step, the *effect* data generated above is distributed to all instances (including the local one) and applied.

Operation-based CRDTs requires *effect* updates to be delivered to all CRDB instances with guarantees to be delivered exactly once, in a source FIFO consistency. To fulfill these guarantees, the CRDB generally relies on the Redis replication mechanism, with a few modifications.

Peer replication

CRDB replication is implemented by syncers that contact remote masters and request *peer replication* (as illustrated below), a new replication mechanism introduced especially for this purpose. Peer replication works in a similar way to standard Redis replication:

- PSYNC and a replication offset are used for resuming broken links
- A fallback to full SYNC is always possible

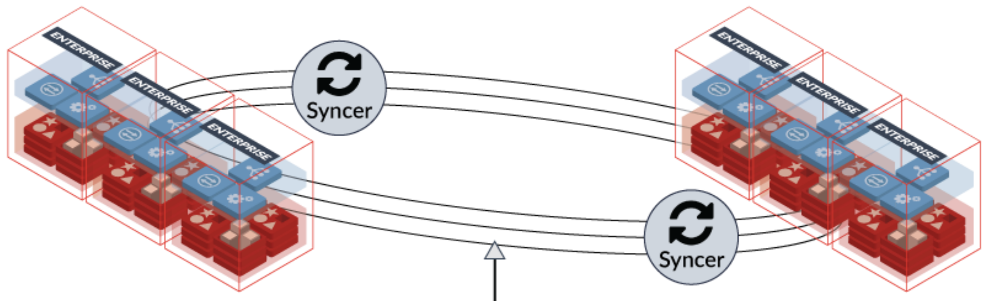
Once a peer has established the replication link, only CRDT effects generated by the CRDT module are propagated. Additional filtering may be applied so that only specific updates will be included, depending on the topology. When using a mesh topology, peer replication will carry only effects generated on the local CRDB instance.

CRDB instances can push different replication streams to different peers, so the peer replication mechanism may manage many different replication backlogs.

Auto GZIP compression and SSL encryption

When a remote public IP is recognized, GZIP compression is automatically applied to the peer replication to better utilize the WAN link. Furthermore, if an SSL handshake is recognized during the connection establishment, encryption will automatically be set.

The peer replication operation is illustrated here:



High availability and disaster recovery

In a case where one or more CRDB instances fail, other instances under the global CRDB continue to read and write. Even if the majority of CRDB instances (for example 3 out of 5) are down, the remaining CRDB instances are uninterrupted and can continue to take both reads and writes. In these types of regional failures, users that can't connect to a local CRDB instance are typically diverted to other data centers that point to one of the available CRDB instances. This provides uninterrupted availability for application reads and writes, even when a user's local CRDB instance is down.

In rare occasions, a CRDB instance may experience full data loss and need replication from scratch. This condition requires special handling, as the recovering CRDB instance may have sent updates to some of its peers. Since no further updates can be expected, we cannot assume all peers will eventually converge (some effect messages may have been received by some peers but not others). In this scenario, Redis Enterprise implements a reconciliation mechanism that involves all of the relevant CRDB instances. Once reconciled, the recovering instance can simply do a full sync from any other replica.

Consistency model

Multiple consistency characteristics are applied in a CRDB deployment:

- For local CRDB instance operations, near strong consistency is achieved when the WAIT command is implemented; otherwise, operations are categorized as having weak consistency.

- For global CRDB operations, Strong Eventual Consistency (SEC) characteristics ensure that any two CRDB instances that have received the same (unordered) set of updates will be in the same state without requiring a consensus protocol. Unlike eventual consistency systems that provide only a liveness guarantee (i.e. updates will be observed eventually), SEC adds a safety guarantee.

Conflict (free) resolution

CRDB conflict resolution is based on three CRDT principles:

- The outcome of concurrent writes is predictable and based on a set of rules.
- Applications don’t need to do anything about concurrent writes (but must be CRDB “dialect” compatible).
- The dataset will eventually converge to a single, consistent state.

How conflict resolution works

Each CRDB instance separately maintains a vector clock for each dataset object/sub-object. This vector clock is updated upon any update operation at the instance level, or when another update operation for the same object arrives from another CRDB instance.

The following process is carried out separately at each CRDB instance upon receiving an update operation (and vector clock) from another instance:

Stage 1: Classify the update operation

A received update operation can represent (1) a new update, (2) an old update, or (3) a concurrent update.

The classification algorithm works as follows:

When Instance A receives an update from Instance B regarding object X:

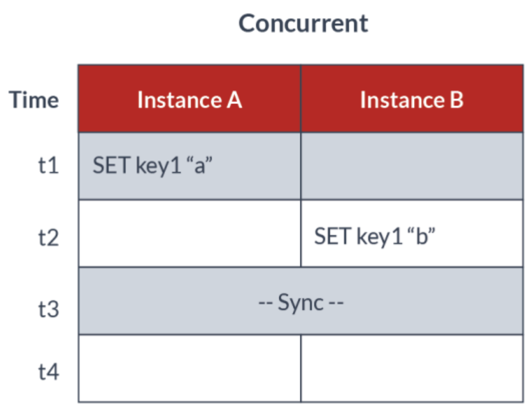
- if: x_vc[b] > x_vc[a] – this is “new” update
- if: x_vc[b] < x_vc[a] – this is an “old” update

- if: x_vc[b] ≸ x_vc[a] – this is a “concurrent” update

Where x_vc[a] is the vector clock of object X at Instance A and x_vc[b] is the vector clock of object X at Instance B.

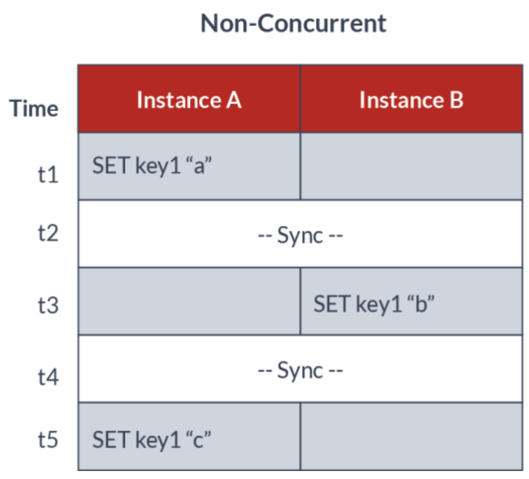
Stage 2: Update the object locally

- If the update was classified as “new,” update the object value in the local CRDB instance
- If the update was classified as “old,” do not update the object value in the local CRDB instance
- If the update was classified as “concurrent,” perform conflict resolution to determine if and how to update the object value in this CRDB instance



Concurrent

Time	Instance A	Instance B
t1	SET key1 “a”	
t2		SET key1 “b”
t3	-- Sync --	
t4		



Non-Concurrent

Time	Instance A	Instance B
t1	SET key1 “a”	
t2	-- Sync --	
t3		SET key1 “b”
t4	-- Sync --	
t5	SET key1 “c”	

The CRDB conflict resolution algorithm is based on two main processes:

Process 1: Conflict resolution for a conflict-free data type/operation

In many concurrent update state cases, an update can be processed completely conflict-free based on the properties of the applicable data type. Here are a few examples:

- When the data type is Counter (mapped to a CRDT’s Counter), all operations are commutative and conflict-free. Example: Counters tracking article popularity, shares or retweets across regions.
- When the data type is Set (mapped to a CRDT’s Add-Wins Observed-Removed Set) and the concurrent updates are ADD operations, all operations are either associative or idempotent and

are conflict-free. Example: In fraud detection, when the application is tracking dubious events associated with an ID or credit card. The Redis Set cardinality associated with the ID or credit card is used to trigger an alert if it reaches a threshold.

- When the data type is Hash (mapped to a CRDT's map) and the concurrent updates are over different fields of the Hash, all operations are conflict-free, as if they were implemented on different objects. Example: A shared corporate account with multiple users, using a plan in different locations or at different rates. The HASH object per account could contain innumerable fields per user, and individual user usage could be updated conflict-free, even when concurrent.

In all of these cases, the object value in this instance is updated according to the data-type policy.

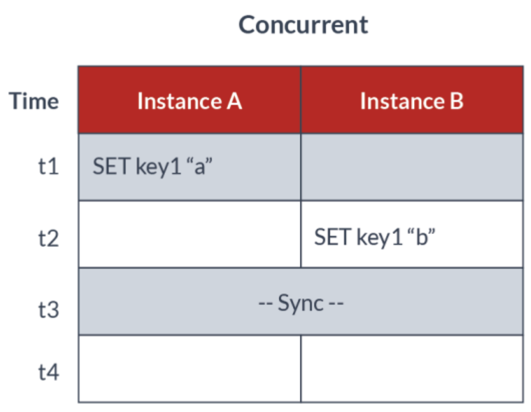
Process 2: Conflict resolution using the Last Write Wins (LWW) mechanism

In cases of concurrent updates in a non conflict-free data type, such as Redis String (mapped to a CRDT's register), a conflict-resolution algorithm should be applied. We have chosen to use the LWW approach to resolve such situations by leveraging the operation timestamp as a tiebreaker.

Note that our solution works in a strong eventually consistent manner, even if there is a timestamp skew between regions. For example, assume Instance A's timestamp is always ahead of the other instances' timestamps (i.e. in case of a tiebreaker, Instance A always wins). This ensures behavior that is eventually consistent. Example: A password is changed for a user account that is accessed by multiple geographically distributed entities. In this case, the change would result in logging out other users, which might be the right behavior for license enforcement scenarios.

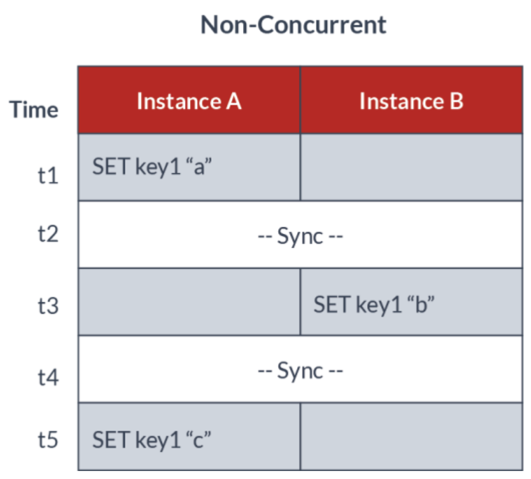
Conflict resolution by examples: Strings

Data type: Strings
Use case: Non-concurrent SETs
Conflict resolution: Conflict-free



Concurrent

Time	Instance A	Instance B
t1	SET key1 “a”	
t2		SET key1 “b”
t3	-- Sync --	
t4		

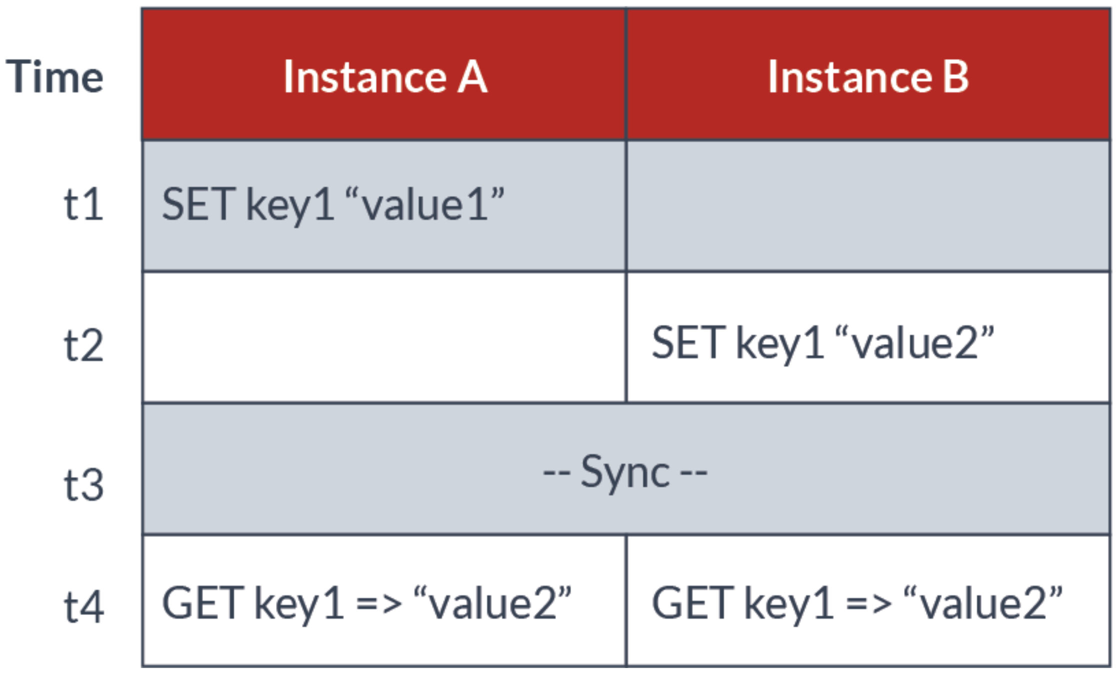


Non-Concurrent

Time	Instance A	Instance B
t1	SET key1 “a”	
t2	-- Sync --	
t3		SET key1 “b”
t4	-- Sync --	
t5	SET key1 “c”	

Note: no conflict, *key1,* was last set to “*value3”* at *t5.*

Data type: Strings
Use case: Concurrent SETs
Conflict resolution: Last Write Wins (LWW)



Time	Instance A	Instance B
t1	SET key1 “value1”	
t2		SET key1 “value2”
t3	-- Sync --	
t4	GET key1 => “value2”	GET key1 => “value2”

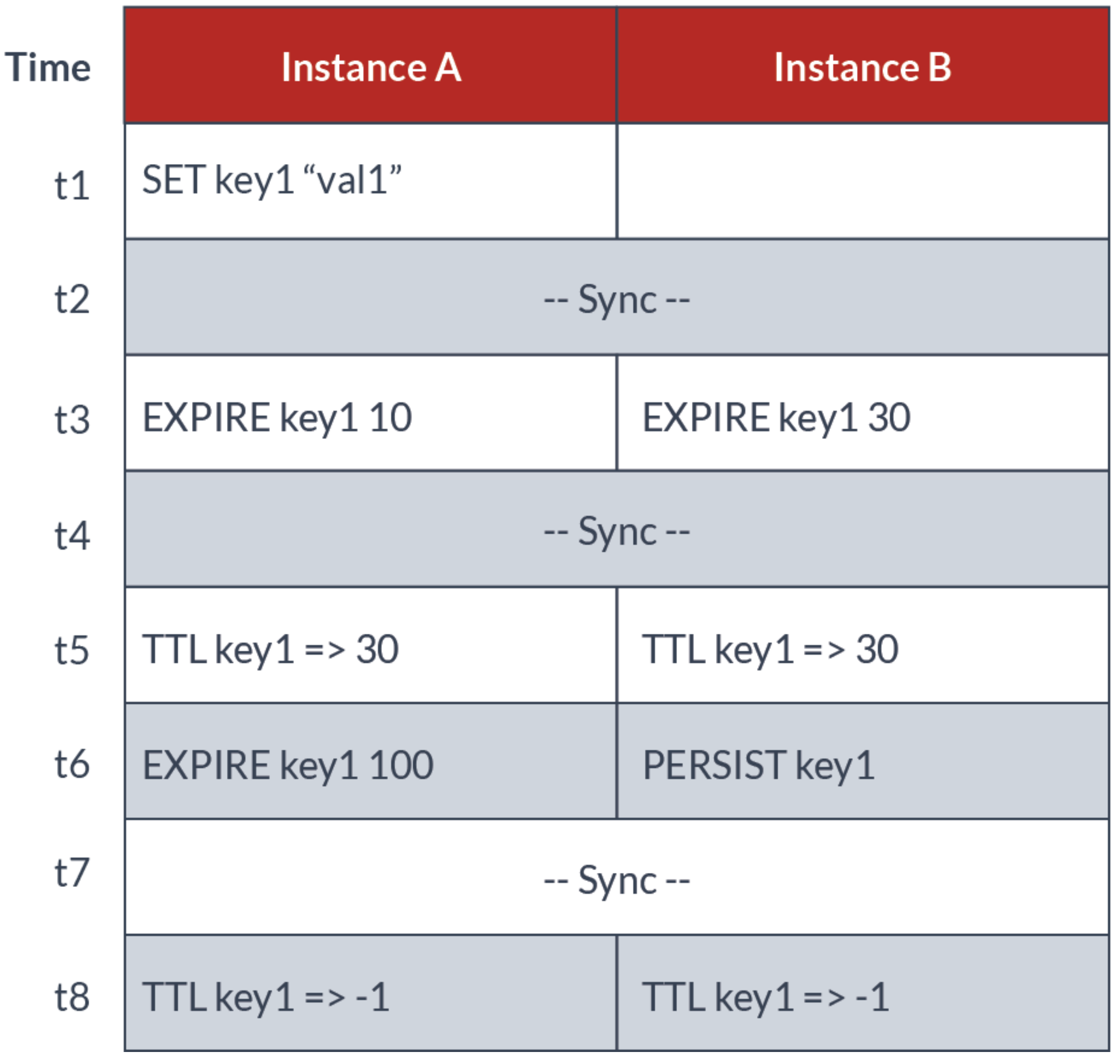
Note: *t2>t1* and due to LWW *key1* was set to *“value2.”*

Data type: Strings
Use case: APPEND vs. DEL
Conflict resolution: Add wins

Time	Instance A	Instance B
t1	SET key1 “Hello”	
t2	-- Sync --	
t3	APPEND key1 “There”	
t4		DEL key1
t5	-- Sync --	
t6	GET key1 => “HelloThere”	GET key1 => “HelloThere”

Note: *APPEND* is an ‘update’ operation treated like ‘add’ and therefore wins the *DEL* operation.

Data type: Strings
Use case: Concurrent expiration
Conflict resolution: Larger TTL wins

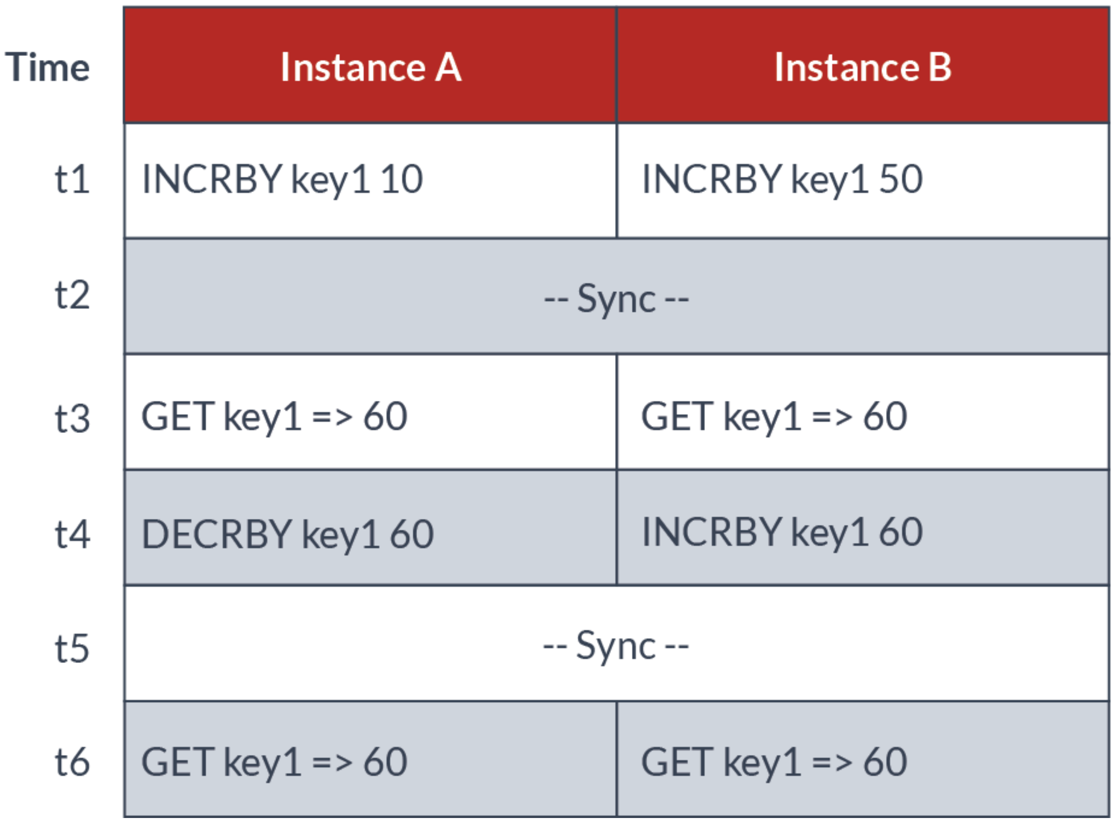


Time	Instance A	Instance B
t1	SET key1 “val1”	
t2	-- Sync --	
t3	EXPIRE key1 10	EXPIRE key1 30
t4	-- Sync --	
t5	TTL key1 => 30	TTL key1 => 30
t6	EXPIRE key1 100	PERSIST key1
t7	-- Sync --	
t8	TTL key1 => -1	TTL key1 => -1

Note: At *t6, Instance B* persisted (with PERSIST *key1*), meaning its TTL was set to infinite (i.e. -1) larger than *100* that was set by *Instance A*.

Conflict resolution by examples: Counters

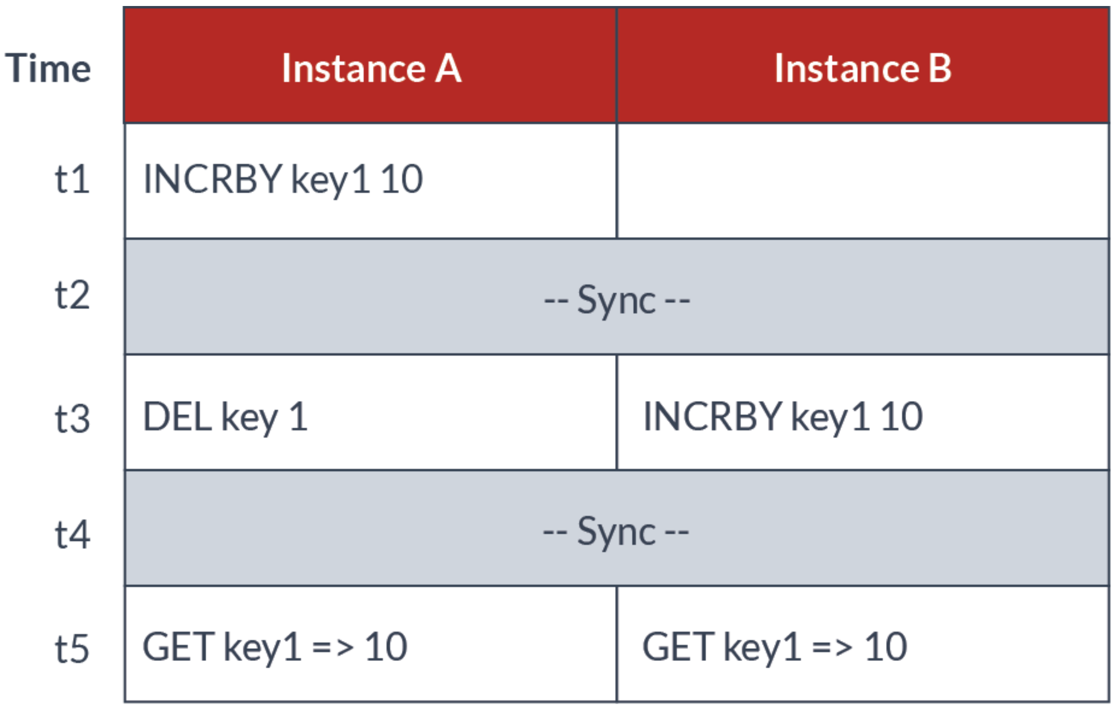
Data type: Counters
Use case: Concurrent increment/decrement operations
Conflict resolution: Conflict-free



Time	Instance A	Instance B
t1	INCRBY key1 10	INCRBY key1 50
t2	-- Sync --	
t3	GET key1 => 60	GET key1 => 60
t4	DECRBY key1 60	INCRBY key1 60
t5	-- Sync --	
t6	GET key1 => 60	GET key1 => 60

Note: In the CRDT layer, the counter value is the SUM of all operations.

Data type: Counters
Use case: Concurrent delete and increment operations (Observed Remove in Counter Value)
Conflict resolution: Add wins

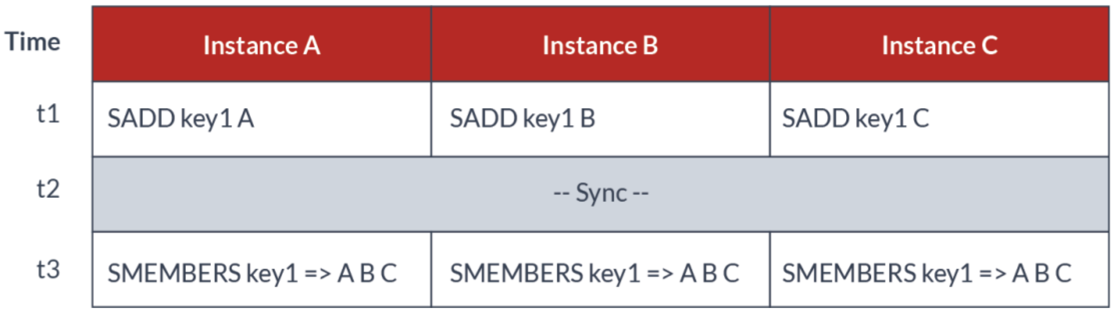


Time	Instance A	Instance B
t1	INCRBY key1 10	
t2	-- Sync --	
t3	DEL key 1	INCRBY key1 10
t4	-- Sync --	
t5	GET key1 => 10	GET key1 => 10

Note: *INCRBY* is an ‘update’ operation treated like ‘add’ and therefore wins the *DEL* operation. In addition, the counter delete operation at *t3*by *Instance A* logically means reset the counter.

Conflict resolution by examples: Sets

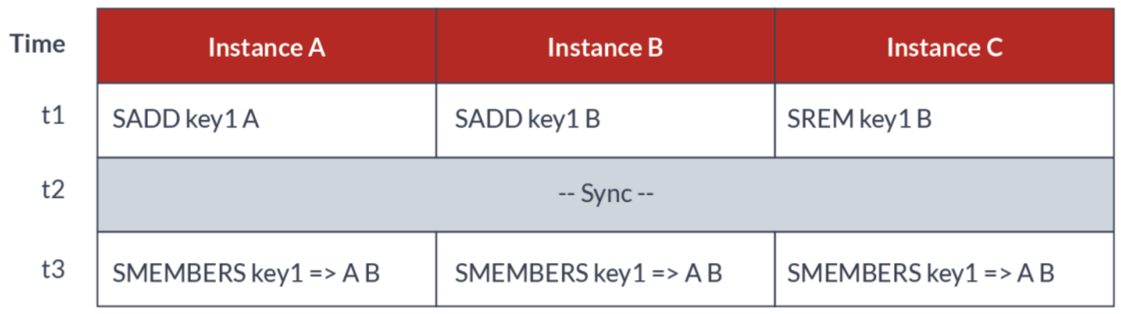
Data type: Sets
Use case: Concurrent SADD operations to a Set
Conflict resolution: Conflict-free



Time	Instance A	Instance B	Instance C
t1	SADD key1 A	SADD key1 B	SADD key1 C
t2	-- Sync --		
t3	SMEMBERS key1 => A B C	SMEMBERS key1 => A B C	SMEMBERS key1 => A B C

Note: This example is conflict-free because *SADD* is an associative operation.

Data Type: Sets
Use Case: Concurrent SADD and SREM operations on a Set
Conflict Resolution: Add wins

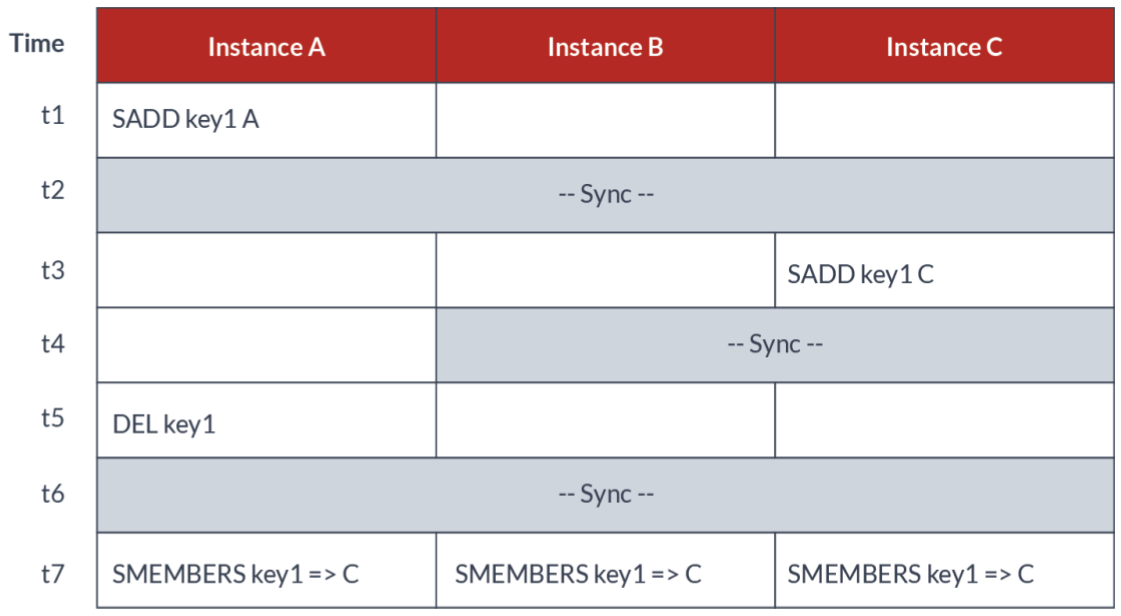


Time	Instance A	Instance B	Instance C
t1	SADD key1 A	SADD key1 B	SREM key1 B
t2	-- Sync --		
t3	SMEMBERS key1 => A B	SMEMBERS key1 => A B	SMEMBERS key1 => A B

Note: Add wins in Set elements.

Data Type: Sets
Use Case: Concurrent complex operation on a Set (Observed Remove)

Conflict Resolution: Add wins in Set elements

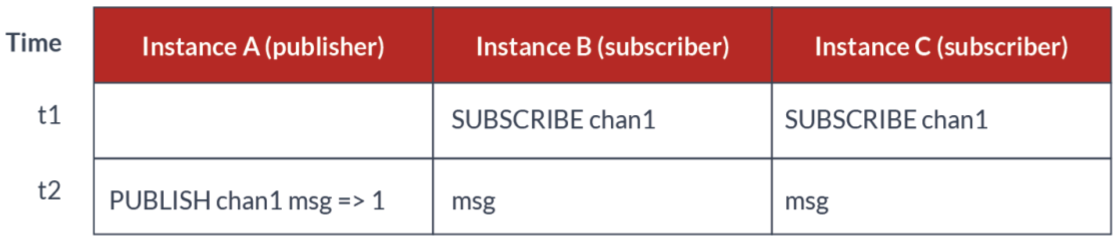


Time	Instance A	Instance B	Instance C
t1	SADD key1 A		
t2	-- Sync --		
t3			SADD key1 C
t4		-- Sync --	
t5	DEL key1		
t6	-- Sync --		
t7	SMEMBERS key1 => C	SMEMBERS key1 => C	SMEMBERS key1 => C

Note: At *t5 Instance A* can only remove elements *A* and *B* from *key1.*

Conflict resolution by examples: PUBLISH/SUBSCRIBE

Data type: Pub/Sub
Use case: Published message propagates to all CRDB instances
Conflict resolution: Conflict-free

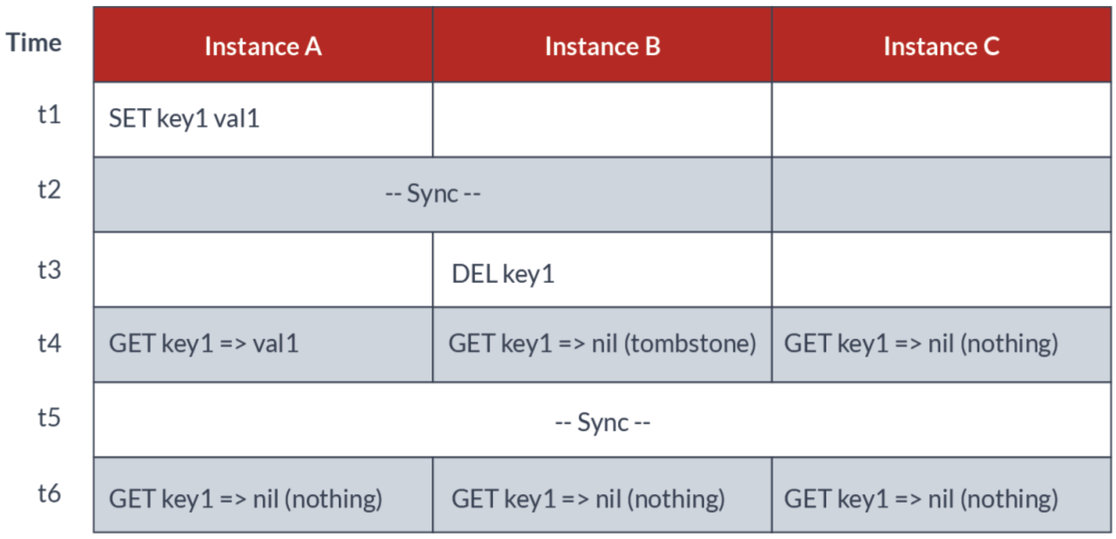


Time	Instance A (publisher)	Instance B (subscriber)	Instance C (subscriber)
t1		SUBSCRIBE chan1	SUBSCRIBE chan1
t2	PUBLISH chan1 msg => 1	msg	msg

Note: In a *PUBLISH* reply, only local subscribers are counted.

Conflict resolution by examples: Garbage collection

Data type: Strings
Use case: Garbage collection
Conflict resolution: Conflict-free



Time	Instance A	Instance B	Instance C
t1	SET key1 val1		
t2	-- Sync --		
t3		DEL key1	
t4	GET key1 => val1	GET key1 => nil (tombstone)	GET key1 => nil (nothing)
t5	-- Sync --		
t6	GET key1 => nil (nothing)	GET key1 => nil (nothing)	GET key1 => nil (nothing)

Note: CRDTs extensively use tombstones. Tombstones become garbage after being observed by all instances.