TUTORIAL

Developing Apps Using Active-Active Redis Enterprise

Roshan Kumar, Redis

CONTENTS

Introduction

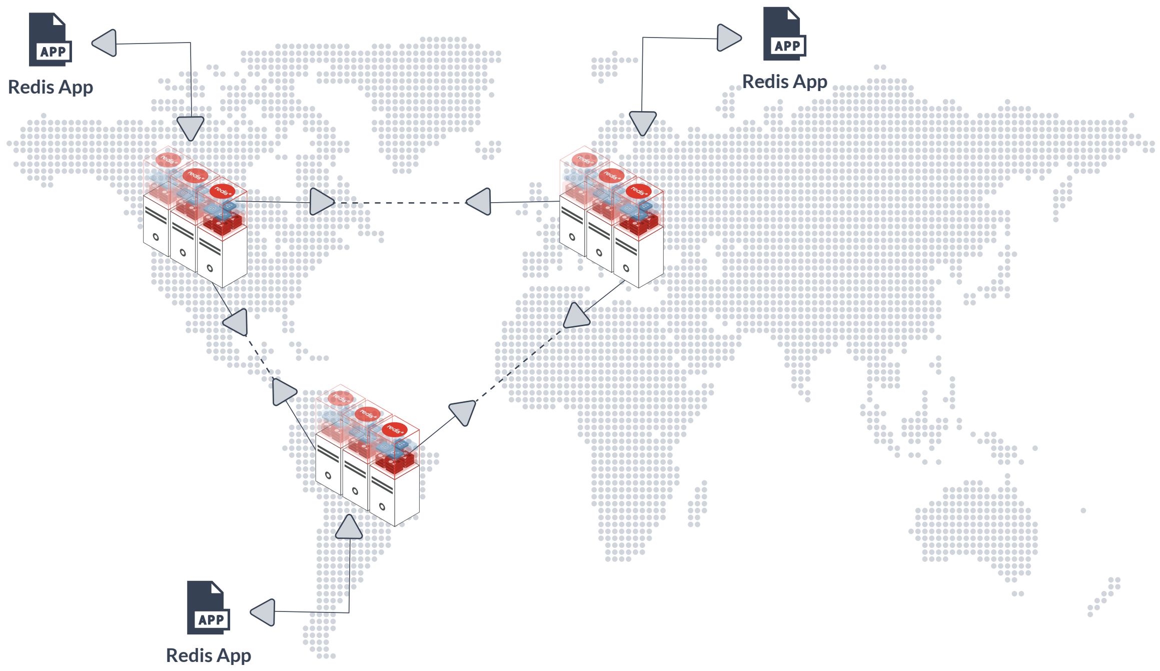
Bending consistency and availability as described by the CAP theorem has been a great challenge for the architects of geo-distributed applications. Network partition is unavoidable; the high latency between data centers always results in some disconnect, even if only for a short period of time. Traditional solutions' architecture for geo-distributed applications follow the CAP theorem and are designed to either give up data consistency or take a hit on availability. Unfortunately, you cannot sacrifice availability when building interactive user applications. In recent times, the architects have taken a shot at consistency and settled down with the "eventual consistency" model. In this model, applications depend on the database management system to merge all local copies of the data to make them eventually consistent.

Everything looks fine and dandy with the eventual consistency model until there are data conflicts. A few weak eventual consistency models promise best effort to fix the conflicts, but fall short of guaranteeing strong consistency. The good news is that models built around conflict-free replicated data types (CRDTs) deliver strong eventual consistency. CRDTs achieve this through a predetermined set of conflict resolution rules and semantics. Applications built on top of CRDT-based databases must be designed to accommodate the conflict resolution semantics. In this paper, we will explore how to design, develop and test geo-distributed applications using Redis CRDTs. We'll also illustrate these techniques with four sample use cases: counters, distributed caching, shared sessions and multi-region data ingest.

Sample geo-distributed architecture

For the rest of this paper we will assume a geo-distributed application deployed in different data centers around the world. As shown in the figure, each region has a local Redis Enterprise cluster. A conflict-free distributed database is deployed across all regions. In this setup, the database in each region is a replica of the databases in the other regions. On top of that, each database is an active database, which means it supports both read and write operations.

The local instances of the geo-distributed application connect to the nearest database to deliver local latency for read and write operations. With Redis CRDTs, Redis Enterprise delivers strong eventual consistency.



Redis CRDTs

What are CRDTs?

CRDTs are special data types that converge the data from all database replicas. The popular CRDTs are G-counters (grow-only counters), PN-counters (positive-negative counters), registers, G-sets (grow-only sets) and OR-sets (observed-remove sets). Behind the scenes, they rely on the following mathematical properties to converge the data:

1. Commutative property: a D b = b Da
2. Associative property: a D (b D c) = (a D b) D c
3. Idempotence: a D a = a

A G-counter is a perfect example of an operational CRDT that merges the operations. Here, a + b = b + a and a + (b + c) = (a + b) + c. The replicas exchange only the updates (additions) with each other. The CRDT will merge the updates by adding them up. A G-set, for example, applies idempotence ({a, b, c} U {c} = {a, b, c}) to merge all the elements—this avoids duplication of elements added to a data structure as they travel and converge via different paths.

Advantage Redis

Redis has a head start with CRDTs—it already has a rich portfolio of data structures: Strings, Hashes, Lists, Sets, Sorted Sets, Bitfields, Geo, Hyperloglog and Streams. Redis Enterprise extends some of these popular data structures as CRDTs. It also introduces a new data type, Counter, which is not available in Redis. Redis CRDTs apply converging and conflict resolution rules that are relevant to specific use cases. Here's a list of Redis CRDTs, along with their conflict resolution techniques:

1. **Counters** can be used as both G-counters and PN-counters
2. **Strings (Registers)** apply "last writer wins" (LWW) for conflict resolution
3. **Hashes** act as registers for strings and counters when the keys are set using hincrby
4. **Sets** support G-sets and observed-remove sets with "add" winning over "delete" during conflicts
5. **Sorted Sets** combine the semantics of both sets and counters (for score)
6. **Lists** ensure all the copies have the same sequence, but the final order is not based on the order in which the items are pushed to the list

How to architect your solution with Redis CRDTs?

If your application already uses Redis as a local, non active-active database, then you have all the necessary libraries in your application stack. However, you cannot just reconfigure your database to be CRDT-based and expect the application to work as before. The data lifecycle in a distributed database is different from that of a centralized one. Therefore:

1. **Make your application stateless.** A stateless application is typically API-driven. Every call to an API results in reconstructing the complete message from scratch. This ensures that you always pull a clean copy of data at any point in time. The low local latency offered by CRDT-based Redis Enterprise makes reconstructing messages faster and easier.

2. **Select the right CRDT that fits your use case.** The Counter is the simplest of the CRDTs; it can be applied for use cases such as global voting, tracking active sessions, metering, etc. However, if you want to merge the state of distributed objects, then you must consider other data structures too. For example, for an application that allows users to edit a shared document, you may want to preserve not just the edits, but also the order in which they were performed. In that case, saving the edits in a List would be a better solution than storing them in a String or a Set CRDT. It's also important that you understand the conflict resolution semantics enforced by the CRDTs, and that your solution conforms to the

rules.

3. **A CRDT isn't a one-size-fits-all solution.** While CRDTs are indeed great tools for many use cases, they may not be the best for all use cases (ACID transactions, for example). CRDT-based Redis Enterprise fits well with microservices architecture where you have a dedicated database for each microservice. In such a case, you could apply Redis CRDTs for only those microservices that require CRDT-based use cases.

The main takeaway from this is that your application should focus on the logic and delegate the data management and synchronization complexity to Redis Enterprise. You may view Redis Enterprise as a RAM memory extension for your application.

Developing and testing applications with Redis Enterprise

Redis Enterprise is designed to reduce the complexity of developing applications and taking them to market.

Client libraries

The good news is that Redis CRDTs are compatible with Redis data types. Therefore, all Redis clients are compatible with Redis CRDTs. If you are porting your existing application from a stand-alone Redis database to a CRDT-based Redis Enterprise database, you do not need to change your client libraries.

Development and testing environment

To achieve faster go-to-market, we recommend that you have a consistent development, testing, staging and production setup. The easiest way to configure a miniaturized environment for development and testing is to have a Docker-based setup. Follow the instructions in Appendix B to deploy a three-node, distributed, CRDT-based Redis Enterprise database on three different subnets of a Docker-based setup.

Testing your application

Testing applications with a distributed multi-master database may sound complex, but all you are testing for, most of the time, is data consistency and application availability in two situations: (1) when the distributed databases are connected and (2) when there is a network partition between the databases.

By setting up a three-node distributed database in your development environment, you can automate and cover most of the testing scenarios in the unit testing itself. These are the recommended guidelines for testing your applications:

1. Test cases when the network connectivity is on and the latency between the nodes is low. Your test cases must have more emphasis on simulating conflicts. Typically, you do this by updating the same data across different nodes many number of times. Incorporate steps to pause and verify the data across all nodes. Even though Redis Enterprise clusters are synchronizing continuously, to test the eventual consistency you will need to pause your test and check the data. For validation, verify the data across all the Redis nodes for two things: (1) that all nodes have the same data and (2) that when conflict occurred, the conflict resolution was consistent with the design.

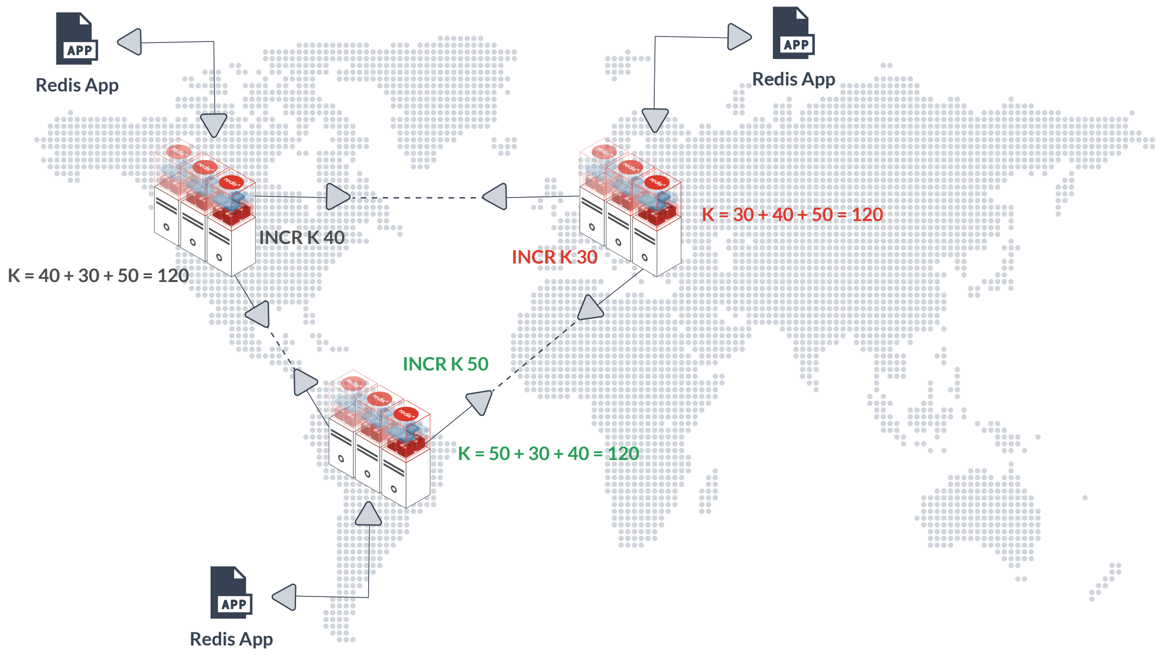
2. Test cases for partitioned networks. You'll want to test a situation where the databases are unable to synchronize with each other, so execute the same test cases as earlier, but use a partitioned network. When the network is split, the database doesn't merge all the data. Therefore, your test case must assume that you are reading only a local copy of the data.

In the second half of the test, reconnect all the networks to test how the merge occurs. If you are following the same test cases as in the previous section, you want to confirm that the eventual final data is the same as in the previous set of steps.

Sample Implementations

1. Counters: polling, likes, heart/emoji counts

Counters have many applications. You may have a geo-distributed application that's collecting the votes, measuring the number of 'likes' on an article, or tracking the number of emoji reactions to a message. In this example, the application local to each geographical location connects to the nearest Redis Enterprise cluster. It then updates the counter and reads the counter with local latencies.



Design

Data type

Counter

Sample keys

Counter - `poll:[pollId]:counter`

Hash counter - `message:[messageId]:emoji hearts`

Sample code:

```
// 1. Counter shared across all users
void countVote(String pollId){
      // Increments the counter.  Redis Enterprise synchronizes
      // the counter by merging the counts across across all locations
      // Redis command: INCR poll:[pollId]:counter
}
// 2. Read the global count
long getVoteCount(String pollId){
      // Redis command: GET poll:[pollId]:counter
}
// 3. Count the number of likes of a particular message
void incrementMessageReaction(String messageId, String emoji){
      // Increment the count inside the Hash data structure
associated
      // with the message
      // message:[messageId]:emojis is the key of the Hash
      // [emoji] is a Hash item that tracks the number of reactions

      // Redis command: HINCRBY message:[messageId]:emojis [emoji] 1
}
```

Test cases for connected network:

1. Increment count at **one region**
   a. Run your app in all regions; increment the counter at one location.
   b. Stop the counter; note the individual counts of each region.
   c. Your app should be consistent across all regions, showing the latest count.

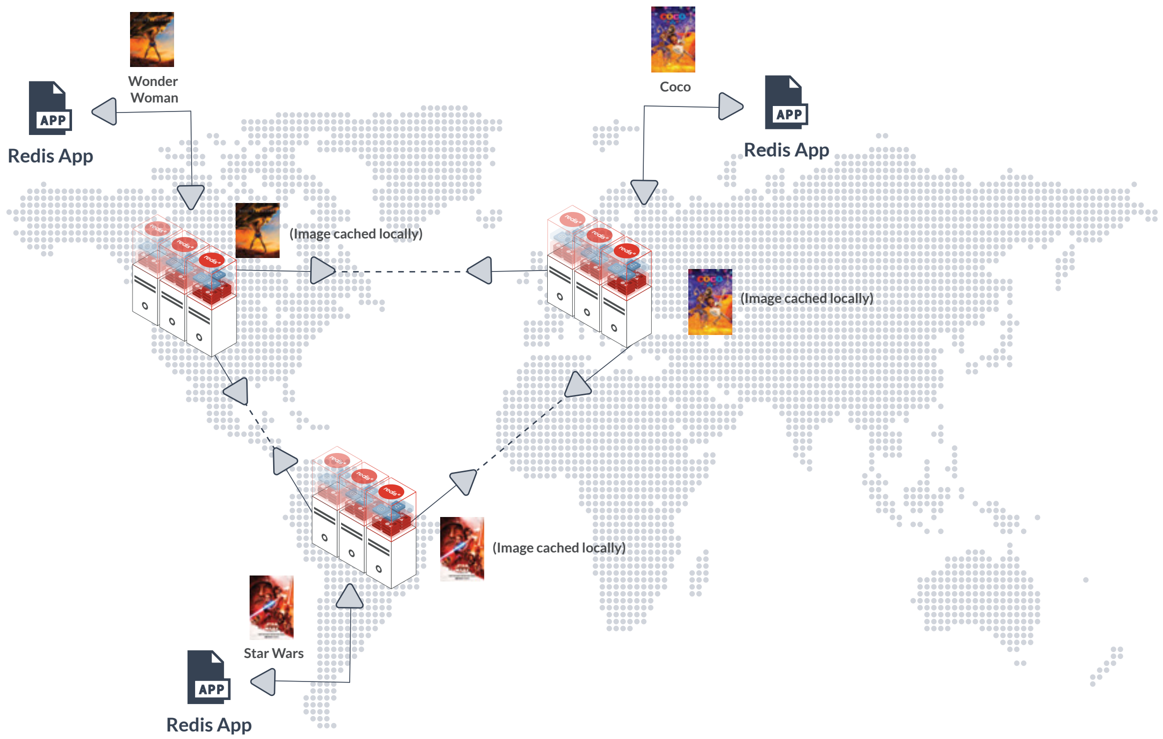
2. Increment count at **multiple regions**
   a. Run your app in all regions; increment the counter at all locations.
   b. Stop the counter; note the individual counts of each region.
   c. Your app should be consistent across all regions, showing the same count.

Test cases for partitioned network:

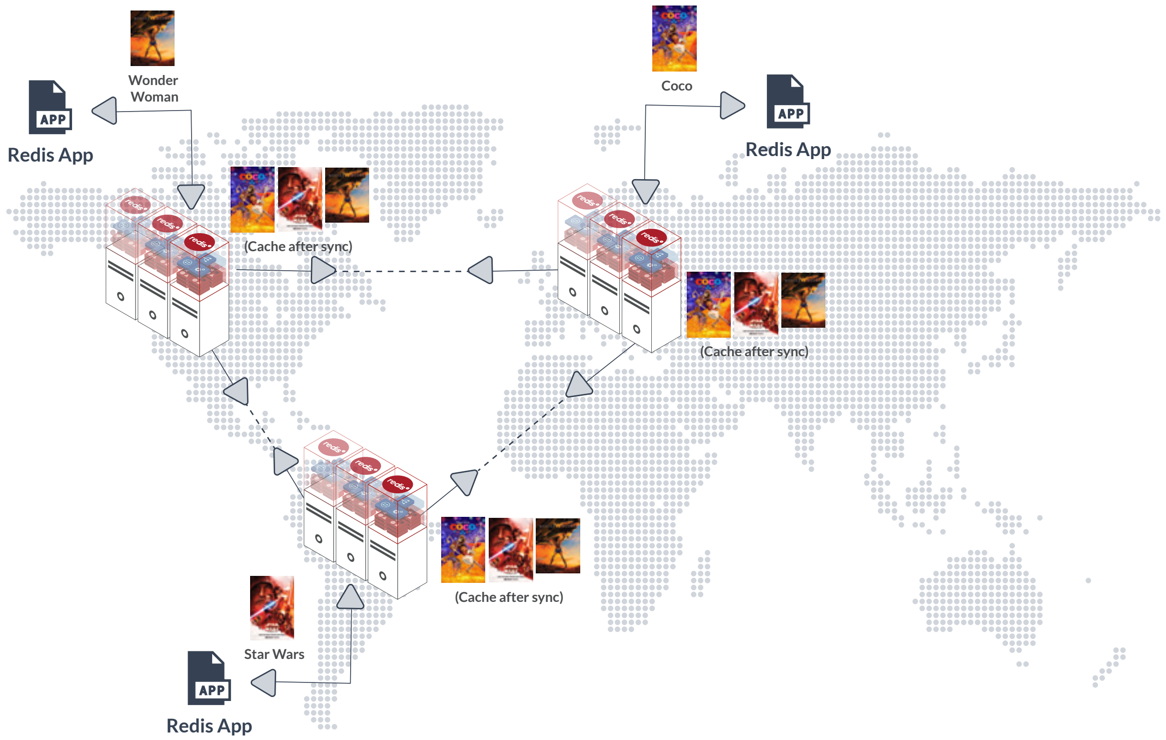
1. Increment count at **multiple regions**
   a. Isolate Redis Enterprise clusters.
   b. Run your app in all regions; increment the counter at all locations.
   c. Stop the counter; note the individual counts of each region.
   d. Your app should show only the local increments.
   e. Reconnect the networks.
   f. All locations should show the updated count; your app should adjust to this behavior.

2. Distributed caching

The caching mechanism for a distributed cache is the same as the one used in local caching: your application tries to fetch an object from the cache. If the object doesn't exist in the cache, the app retrieves it from the primary store and saves it in the cache with an appropriate expiration time. In case of CRDT-based Redis Enterprise, the cached object gets replicated automatically across all the regions. In the following example, the poster image for each movie is cached locally and distributed to all the locations.



Step 1: The poster image is stored in the cache, locally.



Step 2: Redis Enterprise synchronizes the cache across all the regions.

Design

Data type: String
Note that a String in Redis is implemented as a byte array. A String acts like a register with "last writer wins" (LWW) strategy. Should two clusters update the same string, the one with the later timestamp survives in the cache. If your caching solution updates the expiration time of an object and if there's a conflict with the time, then the greatest time is chosen as the value.

Data type: Hash
You may use a Hash data structure as a cache if you are storing metadata or multiple items for the same object.

Sample keys
String: `object:[objectId]`
Hash: `object:[objectId][key1] [value1] [key2] [value2] [key3] [value3]`

```
Sample code:
// 1. Cache an object as a string
void cacheString(String objectId, String cacheData, int ttl){
      // Redis command: SET object:[objectId] [cacheData] ex [ttl]
}
// 2. Get a cached item as a string
String getFromCache(String objectId){
      // Redis command: GET object:[objectId]
}
```

Test cases for connected network

1. Run your app in all regions; set a new cache object in all regions.

2. Verify that your application can pull cached objects from the local clusters, even if the objects were cached at other locations.

Test cases for partitioned network

1. Simulate network partition and set values.

2. Reconnect and verify that your application is designed to handle the following scenarios:

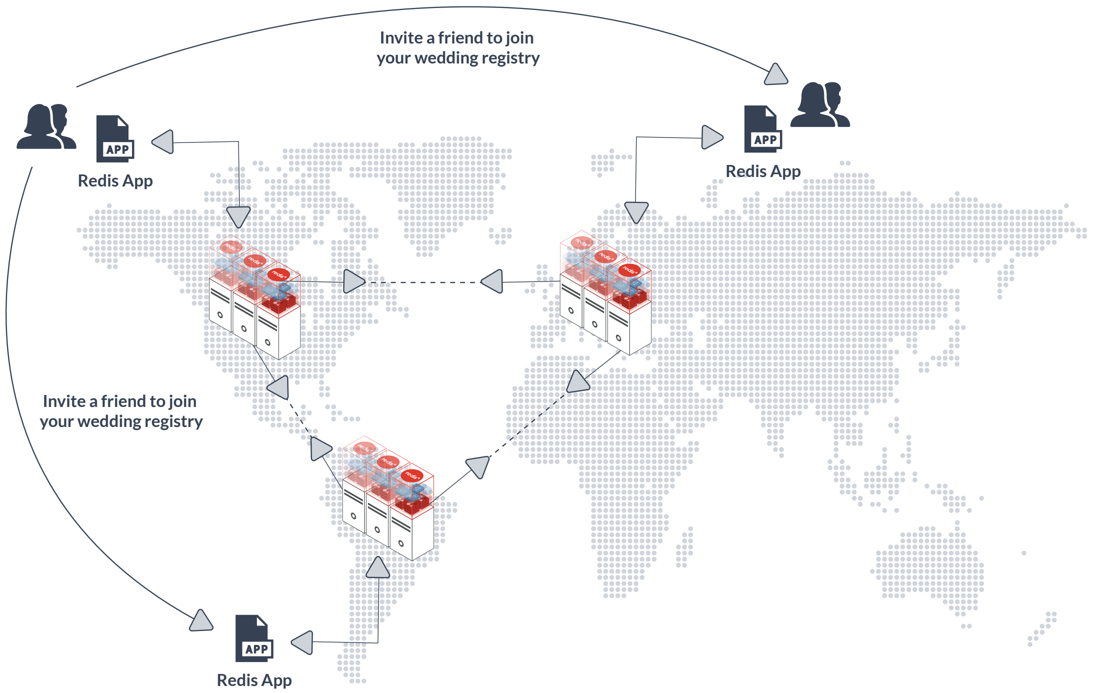
    a. Updating the same key results in "last writer wins"

    b. Upon merge, the keys get the longest value of TTL

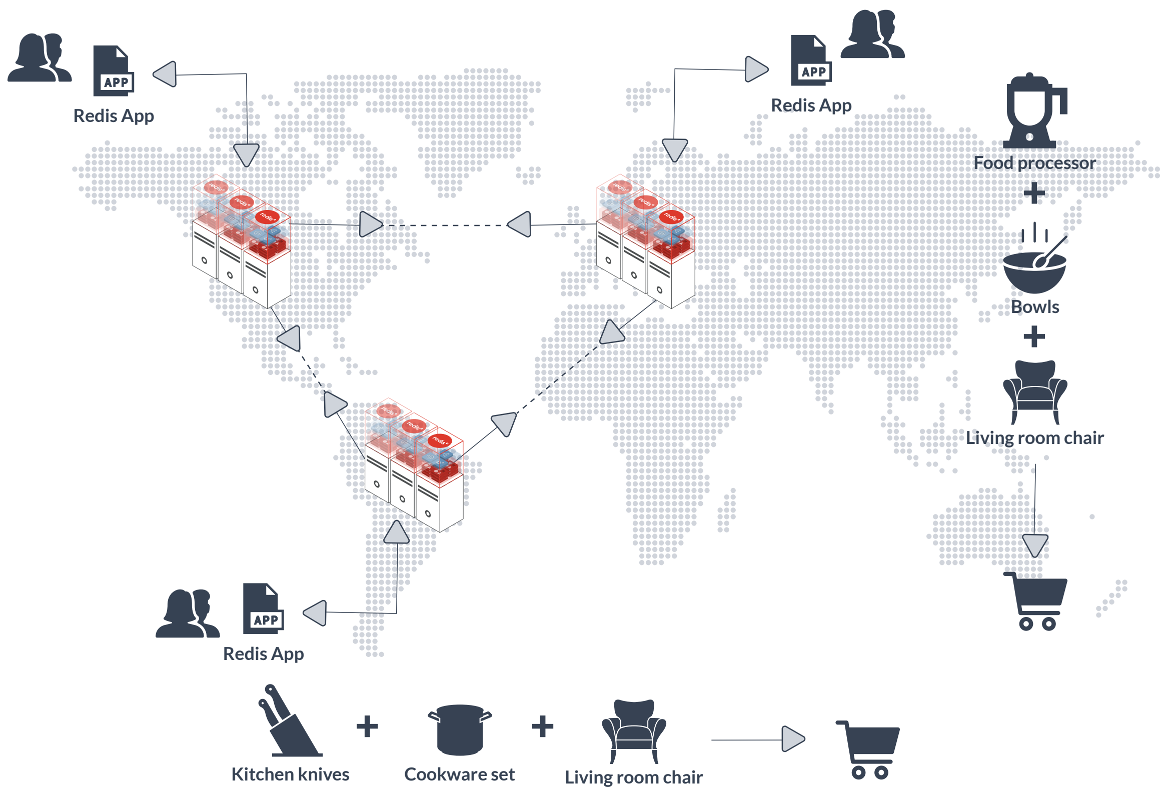
3. Collaboration using shared session data

CRDTs were initially developed to support multi-user document editing. Shared sessions are used in gaming, e-commerce, social networking, chat, collaboration, emergency response and many more applications. In the following example, we demonstrate how you can develop a simple wedding registry application. In this application, all the well wishers of a newly engaged couple add their gifts to a shopping cart that's managed as a shared session.

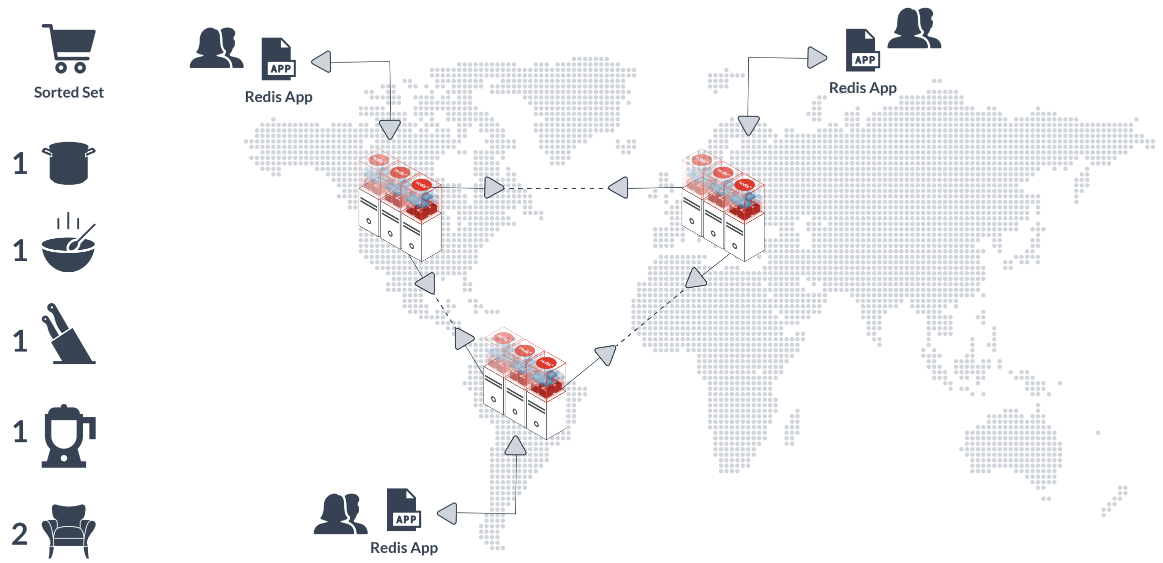
The registry application is a geo-distributed application with each instance connecting to the local database. To begin a session, the owners of the registry invite their friends located around the world. Once the invitees accept the invitation, they all get access to the session object. Later, they shop and add items to the shopping cart.


Invite a friend to join your wedding registry
Redis App
Redis App
Invite a friend to join your wedding registry
Redis App

Step1: The couple invites their international friends to their wedding registry.



Step 2: Invitees add their items to the shopping cart (a Sorted Set).



Step 3: The owners of the wedding registry now have all the items that they can check out.

Design

Data type: Sorted Set

This holds the items of a shopping cart.

Data type: Set

The Set data structure holds all the active sessions.

Sample keys

`SharedSession:[sessionId]`

`ShoppingCart:[sessionId]`

Sample code

```
void joinSession(String sharedSessionID, sessionID){
      // Redis command: SADD sharedSession:[sharedSessionId] [sessionID]
}
void addToCart(String sharedSessionId, String productId, int count){
      // Redis command: ZADD sharedSession:[sharedSessionId] productId count
}
getCartItems(String sharedSessionId){
      // Redis command: ZRANGE sharedSession:sessionSessionId 0 -1
}
```

Test whether your app reflects the CRDT rules:

1. Weights behave like a counter; adding the same object twice results in two objects in the Sorted Set.
2. "Add" wins over "remove."

Test cases for connected network

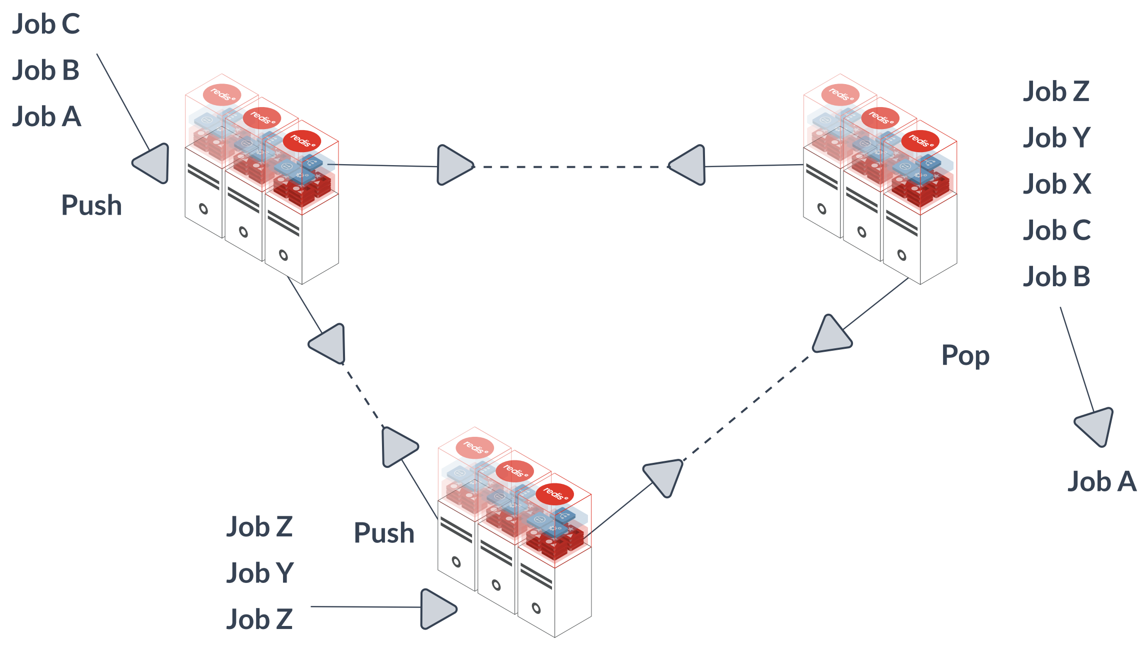
1. Run your app in all regions; add objects to the distributed Sorted Set data structure.
2. Weights behave like a counter; adding the same object twice results in two objects in the Sorted Set.
3. Verify whether your application conforms to these CRDT semantics.

Test cases for partitioned network

1. Simulate network partition; add and remove objects from the local cluster.
2. Reconnect and verify whether your application is designed to handle the CRDT semantics of a Sorted Set.

4. Multi-region data ingest

Lists or queues are used in many applications. In this example, we demonstrate how you can implement a distributed job order processing solution. As shown in the picture, the job order processing system maintains active jobs in a CRDT-based List data structure. The solution collects a jobs at various locations. The distributed application at each location connects to the nearest Redis Enterprise cluster. This reduces the network latency for write operations, which in turn allows the application to support a high volume of job submissions. The jobs are popped out of the List data structure from one of the clusters. This assures that a job is processed only once.



Design

Data type: List

The List data structure is used as a FIFO queue.

Sample keys

`job:[jobQueueId]`

Sample code

```
pushJob(String jobQueueId, String job){
      // Redis command: LPUSH job:[jobQueueId] [job]
}
popJob(String jobQueueId){
      // Redis command: RPOP job:[jobQueueId]
}
```

Test cases for connected network

1. Run your app in all regions; add objects to the List data structure. The CRDT merges objects added to the list.

2. Verify that your application conforms to the CRDT semantics of the List data structure.

Caution: If your application pops an object out of the list at one cluster, then your job processor is assured of processing one job. However, if you pop objects out of all the clusters, then two or more locations may end up processing the same job. If your application requires that you can process a job only once, then you will need to implement your own locking solution at the application layer.

Test cases for partitioned network

1. Simulate network partition; add jobs to the local cluster.

2. Reconnect and verify that your application is designed to handle the CRDT semantics of a List.

A final note

As a final note about Redis Enterprise, there are a few things you need to know before you start designing and developing your applications:

1. **You cannot override the default CRDT conflict resolution semantics.** Some of the databases offer eventual consistency and let the applications override the default conflict resolution semantics with their own. However, Redis Enterprise resolves conflicts beneath the data layer, allowing the application to focus on implementing the logic, and letting the complexity of maintaining data consistency to Redis Enterprise.

2. **Multi-key commands work only if the keys are slotted to the same shard.** CRDT-based Redis Enterprise is deployed in the cluster mode, even if you have a single shard database. Multi-key commands and Lua scripts require that all the keys accessed in a particular call belong to the same shard. There are a few techniques to map the keys to a particular shard. The most popular method is to use a common hashtag within the curly brackets.

3. **Lua script execution follows command synchronization, not script synchronization.** This means that a call to a Lua script at one location will not result in calling the same Lua script at other locations. Instead, as the local Lua script exe-

cutes the Redis commands, only the data changes resulting from those commands are replicated in other regions.

4. **Have a local master-slave setup for high availability.** It's not necessary for your applications to know the global database deployment topology (unless your applications are required to do so by design). The local cluster of Redis Enterprise still has primary-secondary instances on two different nodes. Should a node go down, the auto failure detection and recovery follows the same techniques as a non-CRDT Redis Enterprise database.

5. **If persistence is required, snapshotting is preferred over AOF.** CRDT-based Redis Enterprise supports both snapshotting and AOF configurations. However, recovering from AOF takes a lot more memory and less efficient than snapshotting. Therefore, the recommended practice for persistence is to use snapshotting.

CRDT-based Redis Enterprise enables you to build super-fast, highly engaging geo-distributed applications. In this article, we covered four use cases that apply Redis CRDTs -- global counters, distributed caching, shared session store, and multi-region data ingest. Redis CRDTs enable you to focus on your business logic and not worry about data synchronization between the regions. More than anything, they deliver local latency for engaging applications, and yet promise strong eventual consistency even when there's a network breakdown between data centers.

Appendix 1: Redis CRDT Commands and Conflict Resolution Semantics in Redis Enterprise

For a detailed overview of Redis Enterprise's architecture, which delivers CRDT-based active-active geo distribution, visit the architecture page. This appendix lists the Redis data structures and commands available in CRDT-based Redis Enterprise.

Note: For multi-key operations, the keys must be mapped to the same slot in a cluster. This is usually done via hashtags within curly braces. With CRDTs, the hashslot mapping works the same way as it does with Redis Enterprise clustering.

Data Type	Redis CRDT Commands	Consolidation/Conflict Resolution Semantics
Counter	incr, incrby, incrbyfloat decr, decrby,	Conflict-free merge
	get	Returns the current local copy
	del	"Add/update" wins over "delete." A concurrent delete will leave the key but be treated as decrementing the observed value to zero.
String	set, append, getset, setex, psetex, setnx, setrange, Multi-key operations: mset, msetnx, rename, renamenx	Last writer wins
	get, getrange, mget	Returns the current local copy
	del	"Add/update" wins over "delete"

Data Type	Redis CRDT Commands	Consolidation/Conflict Resolution Semantics
Set	sadd Multi-key operations: sdiffstore, sinterstore, sunionstore, smove	Conflict-free merge, observed remove. Caution: simultaneous smove operations may move the same copy to multiple sets. For example if these commands— smove a b 1 smove a c 1 —are executed on different nodes simultaneously, Item 1 will be available on b and c after sync.
	smembers, sscan, sismemberby, scard, srandmember Multi-key operations: sinter, sunion, sdiff	Performs the operation on the local copy and returns the result.
	del, spop, srem	“Add/update” wins over “delete,” observed remove
Hash	hset, hmset	Conflict-free merge, last writer wins
	hincrby, hincrybyfloat	Works as a counter. Must be initialized using hincrby command. If a key is initialized using hset or hmset, hincrby will not work with that item.
	hget, hgetall, hkeys, hlen, hmget, hscan, hkeys, hvals, hstrln	Performs the operation on the local copy and returns the result
	hdel, delete	“Add/update” wins over “delete,” observed remove
Sorted Set	zadd Multi-key operations: zinterstore, zunionstore	Conflict-free merge
	zincrby	The score is treated as a counter
	zcard, zcount, zrange, zrank, zrevrank, zrevrange, zrangebyscore, zrevrangebyscore, zrangebylex, zrevrangebylex, zlexcount, zscan, zscore	Performs the operation on the local copy and returns the result
	zrem, zremrangebylex, zremrangebyrank, zremrangebyscore	“Add/update” wins over “delete,” observed remove
List	lpush, rpush, linsert	Conflict-free merge
	lrange, lindex	Performs the operation on the local copy and returns the result
	lpop, rpop, blpop, brpop	Returns the local copy. Caution: you may pop the same item at different nodes

Redis data structures NOT supported in Redis Enterprise

Geo
Hyperloglog
Bitfields and Bitmaps

Redis commands that are supported in a CRDT setup

Del (multi-key operation)
Echo
Exists
Expire (during conflict resolution, the longest time wins)
Expireat
Keys (not recommended for production; use Scan instead)
Persist
Pexpire
Pexpireat
Ping
Pttl
slot
Sort (store option results in multi-key operation)
Scan
Touch
Ttl
Type (note the new type “Counter” that’s available with CRDTs)
Unlink
Wait (can be used but has a local effect only)

(Pub/Sub)

Publish
Pubsub (shows channels and subscribers of local cluster)
Psubscribe
Subscribe

Lua scripting, multi-exec -- works as long as the keys are mapped to the same hashslot. Lua scripts are by default (and always) in command replication rather than script replication mode.

Redis commands that are NOT supported in a CRDT setup

Dump
Object

(Strings)
Bitcount
Bitfield
Bitop
Bitpos
Getbit
Setbit

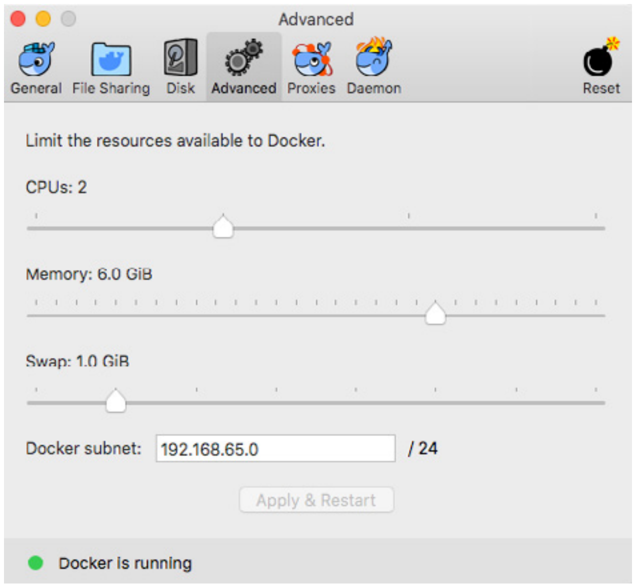
Appendix 2: Setting up a Docker-based Development Environment for CRDT-based Redis Enterprise

Redis Enterprise is available on Docker hub as redis/redis. You can find detailed step-by-step instructions on how to set up Redis Enterprise on Docker at the Redis Enterprise documentation page and the docker hub itself.

As a developer or a tester, your first job is to prepare a miniaturized development environment that mimics your production. Docker is a popular platform among the developers to simulate their production setup. Here we walk you through the steps to create your Docker environment —all through the command line:

1. Set up a database
    a. Create a three-node Redis Enterprise cluster with each node on a separate subnet
    b. Create a CRDT-based Redis Enterprise database
2. Connect to the three different instances
3. Verify your setup
4. Split the networks
5. Restore connections

Before you start, make sure you have enough memory allocated to your docker processes. You could do this by going to Docker -> Preferences -> Advanced.



Limit the resources available to Docker.

CPUs: 2

Memory: 6.0 GiB

Swap: 1.0 GiB

Docker subnet: 192.168.65.0 / 24

Docker is running

1. Set up a database

Run the following script to create a CRDT-based Redis Enterprise database on a 3-node cluster.

```
#!/bin/bash
# Delete the bridge networks if they already exist
docker network rm network1 2>/dev/null
docker network rm network2 2>/dev/null
docker network rm network3 2>/dev/null
# Create new bridge networks
echo "Creating new subnets..."
docker network create network1 --subnet=172.18.0.0/16 --gateway=172.18.0.1
docker network create network2 --subnet=172.19.0.0/16 --gateway=172.19.0.1
docker network create network3 --subnet=172.20.0.0/16 --gateway=172.20.0.1
# Start 3 docker containers. Each container is a node in a separate network
# These commands pull redis/redis from the docker hub. Because of the
# port mapping rules, Redis Enterprise instances are available on ports
# 12000, 12002, 12004
echo ""
echo "Starting Redis Enterprise as Docker containers..."
docker run -d --cap-add sys_resource -h rp1 --name rp1 -p 8443:8443 -p 9443:9443 -p
12000:12000 --network=network1 --ip=172.18.0.2 redis/redis
docker run -d --cap-add sys_resource -h rp2 --name rp2 -p 8445:8443 -p 9445:9443 -p
12002:12000 --network=network2 --ip=172.19.0.2 redis/redis
docker run -d --cap-add sys_resource -h rp3 --name rp3 -p 8447:8443 -p 9447:9443 -p
12004:12000 --network=network3 --ip=172.20.0.2 redis/redis

# Connect the networks
docker network connect network2 rp1
docker network connect network3 rp1
docker network connect network1 rp2
docker network connect network3 rp2
docker network connect network1 rp3
docker network connect network2 rp3

# Sleep while the nodes start. Increase the sleep time if your nodes take
# longer than 60 seconds to start
echo ""
echo "Waiting for the servers to start..."
sleep 60

# Create 3 Redis Enterprise clusters - one for each network. You can login to
# a cluster as https://localhost:8443/ (or 8445, 8447). The user name is
# r@r.com, password is password. Change the user
echo ""
echo "Creating clusters"
docker exec -it rp1 /opt/redis/bin/rladmin cluster create name cluster1.local user-
name r@r.com password test
docker exec -it rp2 /opt/redis/bin/rladmin cluster create name cluster2.local user-
name r@r.com password test
docker exec -it rp3 /opt/redis/bin/rladmin cluster create name cluster3.local user-
name r@r.com password test

# Create the CRDB
echo ""
echo "Creating a CRDB"
docker exec -it rp1 /opt/redis/bin/crdb-cli crdb create --name mycrdb --memory-size
512mb --port 12000 --replication false --shards-count 1 --instance fqdn=cluster1.
```

```
local,username=r@r.com,password=test --instance fqdn=cluster2.local,username=r@r.
com,password=test --instance fqdn=cluster3.local,username=r@r.com,password=test
```

2. Verify Setup

Run redis-cli on ports 12000, 12002 and 12004 and verify that you can connect to all the Redis Enterprise nodes.

```
$ redis-cli -p 12000
127.0.0.1:12000> incr counter
(integer) 1
127.0.0.1:12000> get counter
"1"
```

3. Split the networks

Here we isolate the networks from each node.

```
#!/bin/bash
docker network disconnect network2 rp1
docker network disconnect network3 rp1
docker network disconnect network1 rp2
docker network disconnect network3 rp2
docker network disconnect network1 rp3
docker network disconnect network2 rp3
```

4. Restore connections

This script restores the network connections between the nodes.

```
#!/bin/bash
docker network connect network2 rp1
docker network connect network3 rp1
docker network connect network1 rp2
docker network connect network3 rp2
docker network connect network1 rp3
docker network connect network2 rp3
```