KVS

November 25, 2021

KVS Abstract Scala Types database wich allow to build the storage schemes around the linked list of entities (data feeds). Powered by several backend storage engines it can suite variety of needs. With the RING backend its a great tool for managing the distributed data and provide sequential consistency when used with FeedServer.

https://github.com/zero-deps/kvshttps://github.com/zero-deps/kvs

KVS - Abstract Scala Data Types Key-Value Storage

The Key-Value Database for storing scala value types. It has the simple put/get API plus some extended operation to manage the data feeds.

Backend Storage Engine

The KVS can be backed by the various storage engines.

RING For distributed, scalable, fault tolerant key-value store.

LevelDB For usage in simple non-clustered environment

Memory In memory storage for chache or testing purposes

FS Filesystem storage.

Services Handlers

Particular Services can operate with specific data types. Storing every specific type may require some additional logic. You should think about the handlers as some kind of serializators/picklers/marshallers for KVS.

Datastore JMX interface

KVS application registers MBeans called Kvs and Ring. MBean is object that simlular to JavaBean that represent resourse that can be management using JMX techlology. The Java Management Extensions (JMX) technology is a part of Java Platform that gives abillity to manage aplication remotely. In order to connect to Kvs MBean you can use standart application as jconsole that provided with JDK and located in \$JDK$_{H}OME/bin, orothersthatcompliestotheJMXspecification.$

KVS JMX interfase

Read all feed as string allStr(fid:String):String

Return string representation of all entities in spesified feed.

Export all data Current version of KVS using RNG application as backend layer. As far as RNG is distributed data store it's not possible to backup or migrate data form server to server only with copy-past directory with persisted data. Even more any copy of RNG's data will has no any sense becase data in RNG storage is partitioned and particular node can be acquired only for defined range of keys in store (approximately equals N from quorum configuration divided by number of reachable nodes). Use save method to merge data from all nodes. This method returns path to zip file with dump.

```
Kvs:save
```

Be aware that RNG become readonly after save trigger in order to keep consistency. When archive will composed RNG become writable and readable again.

Load file Archive from save operation can be loaded to application with load method.

```
Kvs:load(path)
```

After load opperation triggered, RNG state become readonly. Data from loaded file has higher priority compare to already stored. KVS become readable and writeable when loading is finished. Be sure that quorum write configuration should be satisfied otherwise data will be ignored on write opperation. This condition can be checked by comparing W property from quorum configuration and currently reachable nodes in cluster. Important note that loading of file that was created on differ version of services can lead to broken data. It's caused because RNG persist data in bytes so all entities that serilisated before saving goes throught serialisation - deserialisation . That's why KVS not supported compatability between differ version of schema.

RNG JMX interfase RNG JMX interface is not for production usage. As far as RNG is lowest level of application, it's not aware about data schema and works only with arrays of Byte. Those method can be used only if caller aware about data schema and keys composition.

Ring:get(key:String): String Get value by key.

Ring:put(key:String, data: String):String This method has value only for testing. Put value associated with key.

Ring:delete(key:String):Unit Delete value associated with key.

KVS its the Key-Value storage framework which provide high level API for handling ‘feeds‘ of the data. Something about sequential consistency...last operation for the operation. Features: Managing linked list various backend support:leveldb, ring sequential consistency via feed server basic schema extendable schema? secondary indexes? mupltiple backends the application is kind of framework for seq consistency via feed server or something else. all the operation that are make without secondary indexes,i.e not linked can be used without feed server. The data in kvs presented as plain scala case classes

iterators, all records can be chained into double-linked list. so you can inherit from entry class and provide the data type. kvs use scala pickling library for serialization so the picklers must be defined on compile time for you kind of data.

the table will support add/remove linked list operation.

kvs.entries(kvs.get(feed,users), user, undefined)

kvs.all(user) - real flat values of all keys from table.

containers are just boxed for storing top of lists

database init kvs.join

put operation is mean put into the database, but add is adding to the list

leveldb - for secondary indexes consistency - put each in the box

statically typed. mutable, key-value,case class store schemas for each container loc process should be spawned - handle feed operations ordering and consistency

after handler for entries with particular payload is defined it can be reused for specific tagged type Scalaz tygged type is used for specify the ”new type” from existing type without the needs to actually create the types.

For example the Message and Metrics are the same kind of entry with string payload. Typicaly they are marked by some empty trait

trait Msg trait Mtr

and the tagged types can be created as follows

type Message = En[String] @@ Msg type Metrics = En[String] @@ Mtr

actually as soon as you defined the En[String] handler implicitly you are ready to define the new handlers. type tags will help to catch the error about incompatible operations on compile time.

Ring Scala implementation of Kai AP, key-value storage which is inspired by Amazon's Dynamo. It's service distributed datastore that follows CAP theorem and provides consistency, partition tolerance and data availability among cluster. Currently it is a part of KVS codebase.

/ring.htmlDetailed documentation

Ring Datastore

Ring Application (also referred by the inner name RNG) is available as akka extension and makes your system part of the highly available, fault tolerant data distrubution cluster.

To configure RNG application on your cluster the next config options are available:

quorum Configured by array of three integer parameters N,W,R where

- **N** Number of nodes in bucket(in other words the number of copies).
- **R** Number of nodes that must be participated in successful read operation.
- **W** Number of nodes for successful write.

To keep data consistent the quorums have to obey the following rules:

1. R + W > N
2. W > N/2

Or use the next hint:

- single node cluster [1,1,1]
- two nodes cluster [2,2,1]
- 3 and more nodes cluster [3,2,2]

if quorum fails on write operation, data will not be saved. So in case if 2 nodes and [2,2,1] after 1 node down the cluster becomes not writeable and readable.

buckets Number of buckets for key. Think about this as about size of HashMap. In current implementation this value should not be changed after first setup.

virtual-nodes Number of virtual nodes for each physical. In current implementation this value should not be changed after first setup.

hash-length Lengths of hash from key. In current implementation this value should not be changed after first setup.

gather-timeout Number of seconds that requested cluster will wait for response from another nodes.

ring-node-name Role name that mark node as part of ring.

leveldb Configuration of levelDB database used as backend for ring.

- **dir** directory location for levelDB storage.
- **fsync** if true levelDB will synchronise data to disk immediately.

RNG provides default configuration for single node mode:

```
ring {
  quorum = [1, 1, 1]  //N,R,W
  buckets = 1024
  virtual-nodes = 128
  hash-length = 32
  gather-timeout = 3
  leveldb {
    dir = "rng_data_"${akka.remote.netty.tcp.hostname}"_"${akka.remote.netty.tcp.port}
    fsync = false
  }
}
```

Listing 1: Example

In case default values are suitable for particular deployment, rewrite is not needed.

KVS/RING Data Metrics

Services uses the data storage so its helpfull to track the state of the different data distribution metrics. The RING distributed key-value storage also provide the node extension to collect and propogate different metrics.

These various stats give a picture of the general level of activity or load on the node at specified moment.

Disk/Memory usage Available disk space. Used file descriptors. Swap Usage. IO wait.

Read operations Consistent reads coordinated by this node. Number of local replicas participating in secondary index reads. Number of siblings encountered during all GET operations by this node within the specific time.

Write operations Consistent writes coordinated by this node. Object size encountered by this node within the specific time. Abnormally large objects (especially paired with high sibling counts) can indicate sibling explosion.

Network Throughput metrics. Latency metrics. General load/health metrics. Network errors.

Search Documents indexed by search. Search queries on the node. Number of "failed to index" errors Search encountered for specific time.

General Load/Health Metrics Watch for abnormally high sibling counts, especially max ones. Number of unprocessed messages in the vnode message queues of the Search subsystem on this node in the specific time.

The KVS/RING Metrics Extension provide JMX control over data distribution system.

Ring datastore

Scala implementation of Kai (originally implemented in erlang). Kai is a distributed key-value datastore, which is mainly inspired by Amazon's Dynamo. Ring is implemented on top of akka and injected as akka extension.

Overview

To reach fault tolerance and scalability ring resolve next problems:

Problems: Technique

Membership and failure detection: reused akka's membership events that uses gossip for communication. FD also reused from akka.

Data partitioning: consistent hashing.

High availability to wright: vector clocks increase number of write opperation to merge data on read opperation.

Handling nodes failures: gossip protocol.

Consistent hashing

To figure out where the data for a particular key goes in that cluster you need to apply a hash function to the key. Just like a hashtable, a unique key maps to a value and of course the same key will always return the same hash code. In very first and simple version of this algorithm the node for particular key is determined by hash(key) mod n, where n is a number of nodes in cluster. This works well and trivial in implementation but when new node join or removed from cluster we got a problem, every object is hashed to a new location. The idea of the consistent hashing algorithm is to hash both node and key using the same hash function. As result we can map the node to an interval, which will contain a number of key hashes. If the node is removed then its interval is taken over by a node with an adjacent interval.

Vector clocks

Vector clocks is an algorithm for generating a partial ordering of events in a distributed system and detecting causality violations. (from wikipedia.org) Vector clocks help us to determine order in which data writes was occurred. This provide ability to write data from one node and after that merge version of data. Vector clock is a list of pairs of node name and number of changes from this node. When data writs first time the vector clock will have one entity (node-A : 1). Each time data amended the counter is incremented.

Quorum

Quorum determines the number of nodes that should be participated in operation. Quorum-like system configured by values: R ,W and N. R is the minimum number of nodes that must participate in a successful read operation. W is the minimum number of nodes that must participate. N is a preference list, the max number of nodes that can be participated in operation. Also quorum can configure balance of latency for read and write operation. In order to keep data strongly consistent configuration should obey rules:

```
1) R + W > N
2) W > V/2
```