Distributed vs Replicated Cache

Posted on **March 14, 2021**

Caching facilitates faster access to data that is repeatedly being asked for. The data might have to be fetched from a database or have to be accessed over a network call or have to be calculated by an expensive computation. We can avoid multiple calls for these repeated data-asks by storing the data closer to the application (Generally, in memory or local disc). Of course, all of this comes at a cost. We need to consider the following factors when cache has to be implemented:

1. Additional memory is needed for applications to cache the data.
2. What if the cached data is updated? How to invalidate the cache? (Needless to say now that caching works great when the data to be cached does not change often)
3. We need to have Eviction Policies (LRU, LFU etc.) in place to delete the entries when cache grows bigger.

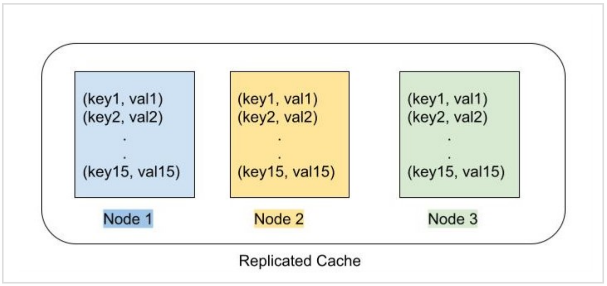
Caching becomes more complicated when we think of distributed systems. Let us assume we have our application deployed in a 3-node cluster:

1. What happens to the cached data when the data is updated (by a REST API call or through a notification). Data gets updated in only one of the nodes where the API call is received or the notification is processed. Data cached in other nodes became stale now.
2. What if the data to be cached is too huge that it does not fit in the application heap?

Usage of distributed/replicated caches would help us in addressing the above problems. Here we will see when to use a distributed vs replicated cache and the pros-cons of each of them.

Replicated Cache

In a replicated cache, all nodes in cluster hold all cached entries. If an entry exists on one node, it will also exist on all other nodes too. So the size of cache is uniform across all the nodes of cluster. As the data is stored in multiple nodes, it can contribute towards higher availability of application.



When a data entry has to be updated in cache, this kind of cache implementation also provides mechanisms to replicate the data on other nodes either synchronously or asynchronously. We have to be mindful of the fact that in case of asynchronous replication, the data stored in cache is inconsistent in nodes for a smaller duration until the replication is completed. Typically this duration is negligible when the data to be transferred across the network is smaller.

Ehcache provides different mechanisms for replicating cache across multiple nodes. Here we will see how JGroups can be used as the underlying mechanism for the replication operations in Ehcache.

```java
package com.prasna.cache.replicated;

import com.prasna.cache.PrasnaReaderCache;
import com.prasna.cache.CacheConstants;
import net.sf.ehcache.Cache;
import net.sf.ehcache.CacheManager;
import net.sf.ehcache.Element;

import java.util.Objects;

public class PrasnaEhCache implements PrasnaReaderCache {
    private final CacheManager cacheManager;
    private final Cache cacheStore;

    public PrasnaEhCache(){
        cacheManager = CacheManager.create(new EhCacheConfig().build());
        cacheStore = cacheManager.getCache(CacheConstants.DEFAULT_CACHE_NAME);
    }

    @Override
    public void put(T key, R value) {
        try {
            cacheStore.acquireWriteLockOnKey(key);
            cacheStore.put(new Element(key, value));
        } finally {
            cacheStore.releaseWriteLockOnKey(key);
        }
    }

    @Override
    public R putIfAbsent(T key, R value) {
        Element element = cacheStore.putIfAbsent(new Element(key, value));
        return element == null ? null : (R)element.getObjectValue();
    }

    @Override
    public R get(T key) {
        Element element = cacheStore.get(key);
        return element == null ? null : (R)element.getObjectValue();
    }

    @Override
    public boolean remove(T key) {
        return cacheStore.remove(key);
    }

    @Override
    public boolean remove(T key, R value) {
        Element existing = cacheStore.removeAndReturnElement(key);
        Object asset = existing != null ? existing.getObjectValue() : null;
        boolean valid = Objects.equals(value, asset);
        if(!valid){
            cacheStore.put(existing);
        }
        return valid;
    }

    @Override
    public void clear() {
        cacheStore.removeAll();
    }
}
```

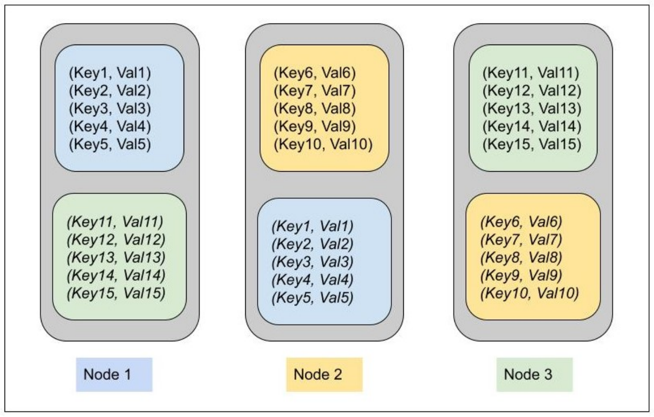
```java
package com.prasna.cache.replicated;

import com.prasna.cache.CacheConstants;
import net.sf.ehcache.config.CacheConfiguration;
import net.sf.ehcache.config.Configuration;
import net.sf.ehcache.config.FactoryConfiguration;
import net.sf.ehcache.store.MemoryStoreEvictionPolicy;

class EhCacheConfig {
    Configuration build(){
        String ehCacheJGroupsPort= System.getenv("prasna.eh_jgroups_port");
        FactoryConfiguration factoryConfiguration = new FactoryConfiguration()
                .className("net.sf.ehcache.distribution.jgroups.JGroupsCacheManagerPee
                .properties("connect=TCP(bind_addr=localhost;bind_port="+ehCacheJGroup
                .propertySeparator("::");

        CacheConfiguration.CacheEventListenerFactoryConfiguration eventListenerFactory
                new CacheConfiguration.CacheEventListenerFactoryConfiguration();
        eventListenerFactoryConfig.className("net.sf.ehcache.distribution.jgroups.JGro
                .properties("replicateAsynchronously=true, replicatePuts=true, replica
                        "replicateUpdatesViaCopy=true, replicateRemovals=true");
        CacheConfiguration defaultCache =
                new CacheConfiguration(CacheConstants.DEFAULT_CACHE_NAME, CacheConstan
                .memoryStoreEvictionPolicy(MemoryStoreEvictionPolicy.LRU)
                .timeToLiveSeconds(CacheConstants.TTL).timeToIdleSeconds(CacheConstant
                .cacheEventListenerFactory(eventListenerFactoryConfig);
        return new Configuration().name("Sample EhCache Cluster")
```

```
                .cacheManagerPeerProviderFactory(factoryConfiguration)
                .cache(defaultCache);
    }
}
```

Distributed Cache

In a distributed cache, all nodes in cluster do not hold all entries. Each node holds only a subset of the overall cached entries. As in case of replicated cache, here also multiple copies (replicas) of data are maintained to provide redundancy and fault tolerance. The replication count here is generally lesser than the number of nodes in cluster unlike the replicated cache.

The way in which we store multiple copies of data across different nodes in distributed cache differs significantly from the way we store in replicated cache. Here data is stored in partitions that are spread across the cluster. A partition can be defined as a range of hash keys. Anytime, we want to cache a value against a key, we serialize the key to byte array, calculate hash and the value is stored in the partition in whose hash range the calculated key hash falls in. So the data is stored in the corresponding partition (primary) and its replica (secondary) partitions as well. This process of saving data into partitions is commonly referred to as Consistent Hashing.



Distributed Cache

Distributed cache provides a far greater degree of scalability than a replicated cache. We can store any amount of data by adding more number of nodes to the cluster as needed without making any modifications to the existing nodes. This is referred to as horizontally scaling the system. If we want to do the same with replicated cache, we need to do vertical scaling i.e. add more resources to the existing nodes itself as every node will store all the keys.

There are many open source distributed cache implementations like Redis, Hazelcast. Here we will see how Hazelcast IMDG (In Memory Data Grid) can be leveraged for a distributed cache implementation.

```
package com.prasna.cache.distributed;

import com.prasna.cache.PrasnaReaderCache;
import com.prasna.cache.CacheConstants;
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.HazelcastInstance;

import java.util.Map;

public class PrasnaHzCache<T, R> implements PrasnaReaderCache<T, R> {
    private final Map<T, R> cache;

    public PrasnaHzCache() {
        HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance(new HzCacl
        this.cache = hazelcastInstance.getMap(CacheConstants.DEFAULT_CACHE_NAME);
    }

    @Override
    public void put(T key, R value) {
        cache.put(key, value);
    }

    @Override
    public R putIfAbsent(T key, R value) {
        return cache.putIfAbsent(key, value);
    }

    @Override
    public R get(T key) {
        return cache.get(key);
    }

    @Override
    public boolean remove(T key) {
        return cache.remove(key) != null;
    }

    @Override
    public boolean remove(T key, R value) {
        return cache.remove(key, value);
    }

    @Override
    public void clear() {
        cache.clear();
    }
}
```

```
package com.prasna.cache.distributed;

import com.prasna.cache.CacheConstants;
import com.hazelcast.config.*;

class HzCacheConfig {
    Config build() {
        Config config = new Config().setClusterName("Sample Hz Cluster");
        NetworkConfig network = config.getNetworkConfig();
        network.setPortAutoIncrement(true);

        JoinConfig join = network.getJoin();
        join.getMulticastConfig().setEnabled(false);
        join.getTcpIpConfig().setEnabled(true)
                .addMember("192.168.0.107");

        EvictionConfig evictionConfig = new EvictionConfig().setEvictionPolicy(Evictio
                .setSize(CacheConstants.CACHE_SIZE).setMaxSizePolicy(MaxSizePolicy.PER
        MapConfig mapConfig = new MapConfig(CacheConstants.DEFAULT_CACHE_NAME).setEvic
                .setTimeToLiveSeconds(CacheConstants.TTL).setMaxIdleSeconds(CacheConst
        config.getMapConfigs().put(CacheConstants.DEFAULT_CACHE_NAME, mapConfig);
        return config;
    }
}
```

Finally, here is a quick comparison of both the cache types:

	Distributed Cache	Replicated Cache
Availability	Availability is improved as data stored across partitions in multiple nodes and every partition will have its replica partition as well.	Availability improved as the whole data cached is stored in all nodes.
Scalability	Highly scalable as more number of nodes can be added to the cluster.	Scaling will be a problem as we need to add more resources to the existing nodes itself
Consistency	Data reads are in general served by primary partitions here. So even when the data is getting copied to replica partitions, we get to read the correct data from primary.	We might not see a consistent view of data while the data is getting replicated to other nodes. Note that any node can serve the data as there is no concept of primary-secondary or master-slave here.
Predictability	When we don't know exactly the data that can be cached before-hand, distributed cache is preferable as it can scale well.	Better option for a small and predictable number of frequently accessed objects

You can find the source code for this post on GitHub.

This entry was posted in **distributed-systems**, **Java** by **Prasanth Gullapalli**. Bookmark the **permalink [https://prasanthnath.wordpress.com/2021/03/14/distributed-vs-replicated-cache/]** .