Handling Network Partitions

Table of Contents

Chapter 1. Setting Up Partition Handling

1.1. Partition handling

An Infinispan cluster is built out of a number of nodes where data is stored. In order not to lose data in the presence of node failures, Infinispan copies the same data — cache entry in Infinispan parlance — over multiple nodes. This level of data redundancy is configured through the `numOwners` configuration attribute and ensures that as long as fewer than `numOwners` nodes crash simultaneously, Infinispan has a copy of the data available.

However, there might be catastrophic situations in which more than `numOwners` nodes disappear from the cluster:

Split brain

Caused e.g. by a router crash, this splits the cluster in two or more partitions, or sub-clusters that operate independently. In these circumstances, multiple clients reading/writing from different partitions see different versions of the same cache entry, which for many application is problematic. Note there are ways to alleviate the possibility for the split brain to happen, such as redundant networks or IP bonding. These only reduce the window of time for the problem to occur, though.

`numOwners` nodes crash in sequence

When at least `numOwners` nodes crash in rapid succession and Infinispan does not have the time to properly rebalance its state between crashes, the result is partial data loss.

The partition handling functionality discussed in this section allows the user to configure what operations can be performed on a cache in the event of a split brain occurring. Infinispan provides multiple partition handling strategies, which in terms of Brewer's CAP theorem determine whether availability or consistency is sacrificed in the presence of partition(s). Below is a list of the provided strategies:

Strategy	Description	CAP
DENY_READ_WRITES	If the partition does not have all owners for a given segment, both reads and writes are denied for all keys in that segment.	Consistency
ALLOW_READS	Allows reads for a given key if it exists in this partition, but only allows writes if this partition contains all owners of a segment. This is still a consistent approach because some entries are readable if available in this partition, but from a client application perspective it is not deterministic.	Consistency

Strategy	Description	CAP
ALLOW_READ_WRITES	Allow entries on each partition to diverge, with conflict resolution attempted upon the partitions merging.	Availability

The requirements of your application should determine which strategy is appropriate. For example, DENY_READ_WRITES is more appropriate for applications that have high consistency requirements; i.e. when the data read from the system must be accurate. Whereas if Infinispan is used as a best-effort cache, partitions maybe perfectly tolerable and the ALLOW_READ_WRITES might be more appropriate as it favours availability over consistency.

The following sections describe how Infinispan handles split brain and successive failures for each of the partition handling strategies. This is followed by a section describing how Infinispan allows for automatic conflict resolution upon partition merges via merge policies. Finally, we provide a section describing how to configure partition handling strategies and merge policies.

1.1.1. Split brain

In a split brain situation, each network partition will install its own JGroups view, removing the nodes from the other partition(s). We don't have a direct way of determining whether the has been split into two or more partitions, since the partitions are unaware of each other. Instead, we assume the cluster has split when one or more nodes disappear from the JGroups cluster without sending an explicit leave message.

Split Strategies

In this section, we detail how each partition handling strategy behaves in the event of split brain occurring.

ALLOW_READ_WRITES

Each partition continues to function as an independent cluster, with all partitions remaining in AVAILABLE mode. This means that each partition may only see a part of the data, and each partition could write conflicting updates in the cache. During a partition merge these conflicts are automatically resolved by utilising the ConflictManager and the configured EntryMergePolicy.

DENY_READ_WRITES

When a split is detected each partition does not start a rebalance immediately, but first it checks whether it should enter **DEGRADED** mode instead:

- If at least one segment has lost all its owners (meaning at least *numOwners* nodes left since the last rebalance ended), the partition enters DEGRADED mode.
- If the partition does not contain a simple majority of the nodes (floor(numNodes/2) + 1) in the *latest stable topology*, the partition also enters DEGRADED mode.
- Otherwise the partition keeps functioning normally, and it starts a rebalance.

The *stable topology* is updated every time a rebalance operation ends and the coordinator determines that another rebalance is not necessary.

These rules ensures that at most one partition stays in AVAILABLE mode, and the other partitions enter DEGRADED mode.

When a partition is in DEGRADED mode, it only allows access to the keys that are wholly owned:

- Requests (reads and writes) for entries that have all the copies on nodes within this partition are honoured.
- Requests for entries that are partially or totally owned by nodes that disappeared are rejected with an `AvailabilityException`.

This guarantees that partitions cannot write different values for the same key (cache is consistent), and also that one partition can not read keys that have been updated in the other partitions (no stale data).

To exemplify, consider the initial cluster `M = {A, B, C, D}`, configured in distributed mode with `numOwners = 2`. Further on, consider three keys `k1`, `k2` and `k3` (that might exist in the cache or not) such that `owners(k1) = {A,B}`, `owners(k2) = {B,C}` and `owners(k3) = {C,D}`. Then the network splits in two partitions, `N1 = {A, B}` and `N2 = {C, D}`, they enter DEGRADED mode and behave like this:

- on `N1`, `k1` is available for read/write, `k2` (partially owned) and `k3` (not owned) are not available and accessing them results in an `AvailabilityException`
- on `N2`, `k1` and `k2` are not available for read/write, `k3` is available

A relevant aspect of the partition handling process is the fact that when a split brain happens, the resulting partitions rely on the original segment mapping (the one that existed before the split brain) in order to calculate key ownership. So it doesn't matter if `k1`, `k2`, or `k3` already existed cache or not, their availability is the same.

If at a further point in time the network heals and `N1` and `N2` partitions merge back together into the initial cluster `M`, then `M` exits the degraded mode and becomes fully available again. During this merge operation, because `M` has once again become fully available, the ConflictManager and the configured EntryMergePolicy are used to check for any conflicts that may have occurred in the interim period between the split brain occurring and being detected.

As another example, the cluster could split in two partitions `O1 = {A, B, C}` and `O2 = {D}`, partition `O1` will stay fully available (rebalancing cache entries on the remaining members). Partition `O2`, however, will detect a split and enter the degraded mode. Since it doesn't have any fully owned keys, it will reject any read or write operation with an `AvailabilityException`.

If afterwards partitions `O1` and `O2` merge back into `M`, then the ConflictManager attempts to resolve any conflicts and `D` once again becomes fully available.

ALLOW_READS

Partitions are handled in the same manner as DENY_READ_WRITES, except that when a partition is in DEGRADED mode read operations on a partially owned key WILL not throw an AvailabilityException.

Current limitations

Two partitions could start up isolated, and as long as they don't merge they can read and write inconsistent data. In the future, we will allow custom availability strategies (e.g. check that a certain node is part of the cluster, or check that an external machine is accessible) that could handle that situation as well.

1.1.2. Successive nodes stopped

As mentioned in the previous section, Infinispan can't detect whether a node left the JGroups view because of a process/machine crash, or because of a network failure: whenever a node leaves the JGroups cluster abruptly, it is assumed to be because of a network problem.

If the configured number of copies (`numOwners`) is greater than 1, the cluster can remain available and will try to make new replicas of the data on the crashed node. However, other nodes might crash during the rebalance process. If more than `numOwners` nodes crash in a short interval of time, there is a chance that some cache entries have disappeared from the cluster altogether. In this case, with the DENY_READ_WRITES or ALLOW_READS strategy enabled, Infinispan assumes (incorrectly) that there is a split brain and enters DEGRADED mode as described in the split-brain section.

The administrator can also shut down more than `numOwners` nodes in rapid succession, causing the loss of the data stored only on those nodes. When the administrator shuts down a node gracefully, Infinispan knows that the node can't come back. However, the cluster doesn't keep track of how each node left, and the cache still enters DEGRADED mode as if those nodes had crashed.

At this stage there is no way for the cluster to recover its state, except stopping it and repopulating it on restart with the data from an external source. Clusters are expected to be configured with an appropriate `numOwners` in order to avoid `numOwners` successive node failures, so this situation should be pretty rare. If the application can handle losing some of the data in the cache, the administrator can force the availability mode back to AVAILABLE via JMX.

1.1.3. Conflict Manager

The conflict manager is a tool that allows users to retrieve all stored replica values for a given key. In addition to allowing users to process a stream of cache entries whose stored replicas have conflicting values. Furthermore, by utilising implementations of the EntryMergePolicy interface it is possible for said conflicts to be resolved automatically.

Detecting Conflicts

Conflicts are detected by retrieving each of the stored values for a given key. The conflict manager retrieves the value stored from each of the key's write owners defined by the current consistent hash. The .equals method of the stored values is then used to determine whether all values are equal. If all values are equal then no conflicts exist for the key, otherwise a conflict has occurred. Note that null values are returned if no entry exists on a given node, therefore we deem a conflict to have occurred if both a null and non-null value exists for a given key.

Merge Policies

In the event of conflicts arising between one or more replicas of a given CacheEntry, it is necessary for a conflict resolution algorithm to be defined, therefore we provide the EntryMergePolicy interface. This interface consists of a single method, "merge", whose returned CacheEntry is utilised as the "resolved" entry for a given key. When a non-null CacheEntry is returned, this entries value is "put" to all replicas in the cache. However when the merge implementation returns a null value, all replicas associated with the conflicting key are removed from the cache.

The merge method takes two parameters: the "preferredEntry" and "otherEntries". In the context of a partition merge, the preferredEntry is the primary replica of a CacheEntry stored in the partition that contains the most nodes or if partitions are equal the one with the largest topologyId. In the event of overlapping partitions, i.e. a node A is present in the topology of both partitions {A}, {A,B,C}, we pick {A} as the preferred partition as it will have the higher topologId as the other partition's topology is behind. When a partition merge is not occurring, the "preferredEntry" is simply the primary replica of the CacheEntry. The second parameter, "otherEntries" is simply a list of all other entries associated with the key for which a conflict was detected.

> EntryMergePolicy::merge is only called when a conflict has been detected, it is not called if all CacheEntrys are the same.

Currently Infinispan provides the following implementations of EntryMergePolicy:

Policy	Description
MergePolicy.NONE (default)	No attempt is made to resolve conflicts. Entries hosted on the minority partition are removed and the nodes in this partition do not hold any data until the rebalance starts. Note, this behaviour is equivalent to prior Infinispan versions which did not support conflict resolution. Note, in this case all changes made to entries hosted on the minority partition are lost, but once the rebalance has finished all entries will be consistent.

Policy	Description
MergePolicy.PREFERRED_ALWAYS	Always utilise the "preferredEntry". MergePolicy.NONE is almost equivalent to PREFERRED_ALWAYS, albeit without the performance impact of performing conflict resolution, therefore MergePolicy.NONE should be chosen unless the following scenario is a concern. When utilising the DENY_READ_WRITES or DENY_READ strategy, it is possible for a write operation to only partially complete when the partitions enter DEGRADED mode, resulting in replicas containing inconsistent values. MergePolicy.PREFERRED_ALWAYS will detect said inconsistency and resolve it, whereas with MergePolicy.NONE the CacheEntry replicas will remain inconsistent after the cluster has rebalanced.
MergePolicy.PREFERRED_NON_NULL	Utilise the "preferredEntry" if it is non-null, otherwise utilise the first entry from "otherEntries".
MergePolicy.REMOVE_ALL	Always remove a key from the cache when a conflict is detected.
Fully qualified class name	The custom implementation for merge will be used Custom merge policy

1.1.4. Usage

During a partition merge the ConflictManager automatically attempts to resolve conflicts utilising the configured EntryMergePolicy, however it is also possible to manually search for/resolve conflicts as required by your application.

The code below shows how to retrieve an EmbeddedCacheManager's ConflictManager, how to retrieve all versions of a given key and how to check for conflicts across a given cache.

```java
EmbeddedCacheManager manager = new DefaultCacheManager("example-config.xml");
Cache<Integer, String> cache = manager.getCache("testCache");
ConflictManager<Integer, String> crm = ConflictManagerFactory.get(cache
.getAdvancedCache());

// Get All Versions of Key
Map<Address, InternalCacheValue<String>> versions = crm.getAllVersions(1);

// Process conflicts stream and perform some operation on the cache
Stream<Map<Address, InternalCacheEntry<Integer, String>>> stream = crm.getConflicts();
stream.forEach(map -> {
   CacheEntry<Object, Object> entry = map.values().iterator().next();
   Object conflictKey = entry.getKey();
   cache.remove(conflictKey);
});

// Detect and then resolve conflicts using the configured EntryMergePolicy
crm.resolveConflicts();

// Detect and then resolve conflicts using the passed EntryMergePolicy instance
crm.resolveConflicts((preferredEntry, otherEntries) -> preferredEntry);
```

Although the `ConflictManager::getConflicts` stream is processed per entry, the underlying spliterator is in fact lazily-loading cache entries on a per segment basis.

1.1.5. Configuring partition handling

Unless the cache is distributed or replicated, partition handling configuration is ignored. The default partition handling strategy is ALLOW_READ_WRITES and the default EntryMergePolicy is MergePolicies::PREFERRED_ALWAYS.

```xml
<distributed-cache name="the-default-cache">
   <partition-handling when-split="ALLOW_READ_WRITES" merge-policy="
PREFERRED_NON_NULL"/>
</distributed-cache>
```

The same can be achieved programmatically:

```java
ConfigurationBuilder dcc = new ConfigurationBuilder();
dcc.clustering().partitionHandling()
                    .whenSplit(PartitionHandling.ALLOW_READ_WRITES)
                    .mergePolicy(MergePolicies.PREFERRED_ALWAYS);
```

Implement a custom merge policy

It’s also possible to provide custom implementations of the EntryMergePolicy

```xml
<distributed-cache name="the-default-cache">
   <partition-handling when-split="ALLOW_READ_WRITES" merge-policy=
"org.example.CustomMergePolicy"/>
</distributed-cache>
```

```java
ConfigurationBuilder dcc = new ConfigurationBuilder();
dcc.clustering().partitionHandling()
                    .whenSplit(PartitionHandling.ALLOW_READ_WRITES)
                    .mergePolicy(new CustomMergePolicy());
```

```java
public class CustomMergePolicy implements EntryMergePolicy<String, String> {

   @Override
   public CacheEntry<String, String> merge(CacheEntry<String, String> preferredEntry,
List<CacheEntry<String, String>> otherEntries) {
      // decide which entry should be used

      return the_solved_CacheEntry;
   }
```

Deploy custom merge policies to a Infinispan server instance

To utilise a custom EntryMergePolicy implementation on the server, it's necessary for the implementation class(es) to be deployed to the server. This is accomplished by utilising the java service-provider convention and packaging the class files in a jar which has a META-INF/services/org.infinispan.conflict.EntryMergePolicy file containing the fully qualified class name of the EntryMergePolicy implementation.

```
# list all necessary implementations of EntryMergePolicy with the full qualified name
org.example.CustomMergePolicy
```

In order for a Custom merge policy to be utilised on the server, you should enable object storage, if your policies semantics require access to the stored Key/Value objects. This is because cache entries in the server may be stored in a marshalled format and the Key/Value objects returned to your policy would be instances of WrappedByteArray. However, if the custom policy only depends on the metadata associated with a cache entry, then object storage is not required and should be avoided (unless needed for other reasons) due to the additional performance cost of marshalling data per request. Finally, object storage is never required if one of the provided merge policies is used.

1.1.6. Monitoring and administration

The availability mode of a cache is exposed in JMX as an attribute in the Cache MBean. The attribute is writable, allowing an administrator to forcefully migrate a cache from DEGRADED mode back to AVAILABLE (at the cost of consistency).

The availability mode is also accessible via the AdvancedCache interface:

```
AdvancedCache ac = cache.getAdvancedCache();

// Read the availability
boolean available = ac.getAvailability() == AvailabilityMode.AVAILABLE;

// Change the availability
if (!available) {
   ac.setAvailability(AvailabilityMode.AVAILABLE);
}
```